M64 – Data Science & Scientific Computing 6 : Schémas numériques pour les problèmes de Cauchy

L3 Mathématiques - Université de Toulon

Gloria FACCANONI

04 mars 2026

Table des matières

1. Schémas numériques pour les problèmes de Cauchy 1

I Introduction 3

2. Syllabus du cours DSSC-6 : Schémas numériques pour les problèmes de Cauchy 5
2.1. Introduction : le parcours DSSC . . . 5
2.2. Déroulement du cours DSSC-6 : Schémas numériques pour les problèmes de Cauchy . . . 5
2.2.1. Objectifs . . . 6
2.2.2. Plan des séances . . . 6
2.2.3. Contenu du cours . . . 6
2.2.4. Prérequis . . . 7
2.2.5. Évaluation . . . 7
2.2.6. Informations système . . . 7

II Cours 9

3. Rappels 11
3.1. Équations Différentielles Ordinaires (EDO) . . . 11
3.1.1. EDO d'ordre p . . . 11
3.1.2. EDO d'ordre supérieur à 1 $\rightsquigarrow$ Système d'EDOs d'ordre 1 . . . 13
3.1.3. Conditions Initiales . . . 15
3.2. Problème de Cauchy . . . 15
3.2.1. Intervalle de Validité et Solution Maximale . . . 15
3.3. Méthodes de Résolution . . . 16
3.3.1. EDO du premier ordre à variables séparables . . . 17
3.3.2. EDO linéaires du premier ordre . . . 18
3.3.3. EDO de Bernoulli . . . 20

4. Introduction à l'approximation numérique d'EDO 21
4.1. Position du problème . . . 21
4.2. Solution exacte discrète VS solution approchée . . . 21
4.3. Construction élémentaire des méthodes d'Euler explicite et implicite . . . 23
4.4. Implémentation des schémas d'Euler explicite et implicite . . . 24
4.5. Convergence empirique des schémas d'Euler . . . 26
4.6. Annexe : implémentation d'un schéma "moyen" des deux schémas précédents . . . 29

5. Schémas "classiques" à un pas 31
5.1. Construction de schémas à un pas . . . 32
5.1.1. Schéma d'Euler explicite . . . 32
5.1.2. Schéma d'Euler implicite . . . 33
5.1.3. Schémas d'Euler modifié (EM) ou de type Runge-Kutta (RK1_M) . . . 34
5.1.4. Schéma du trapèze (ou de Crank-Nicolson) . . . 36
5.1.5. Schéma de Heun . . . 37
5.1.6. Remarques . . . 38

5.2. Annexes . . . 38
5.2.1. Schémas basés sur la formule de quadrature de Simpson . . . 38
5.2.2. Annexe : automatisation de la construction de schémas . . . 40

6. Propriétés du problème de Cauchy 43
6.1. Problème **mathématiquement** bien posé : existence et unicité de la solution . . . 43
6.2. Problème **numériquement** bien posé : dépendance continue de la solution par rapport aux données. 46
6.2.1. Exemple de problème BIEN posé numériquement . . . 47
6.2.2. Exemple de problème BIEN posé numériquement . . . 47
6.2.3. Exemple de problème MAL posé numériquement . . . 48
6.2.4. Exemple de problème MAL posé numériquement . . . 49
6.2.5. Exercices . . . 50
6.3. Problème **bien conditionné** (= non raide) : solution approchée en un nombre raisonnable d'itérations. 54
6.3.1. Exemple de problème MAL conditionné (= raide = *stiff*) . . . 54
6.3.2. Exercices . . . 55

7. Construction de schémas multi-pas linéaires 63
7.1. Sympy pour la construction de schémas multi-pas linéaires . . . 65
7.2. Schémas de Adams : intégration sur $[t_n, t_{n+1}]$ . . . 66
7.2.1. Construction des schémas de Adams-Bashford (AB-q) . . . 67
7.2.2. Construction des schémas de Adams-Moulton (AM-q) . . . 70
7.2.3. Automatisation de la construction des schémas de Adam . . . 72
7.3. Schémas de Nyström et de Milne-Simpson : intégration sur $[t_{n-1}, t_{n+1}]$ . . . 75
7.3.1. Construction des schémas de Nyström (N-q) . . . 76
7.3.2. Construction des schémas de Milne-Simpson (MS-q) . . . 78
7.3.3. Automatisation de la construction des schémas de Nyström et de Milne-Simpson . . . 80
7.4. Schémas BDF (Backward Differentiation Formulae) . . . 82
7.4.1. Construction des schémas BDF (BDF-q) . . . 82
7.4.2. Automatisation de la construction des schémas BDF . . . 84
7.5. Combiner des schéma multi-pas avec une approche *predictor-corrector* . . . 85

8. Propriétés d'un schéma sur un intervalle borné 87
8.1. Convergence et ordre de convergence . . . 87
8.2. Consistance et ordre de consistance . . . 92
8.3. Zéro-stabilité . . . 92

9. Propriétés d'un schéma sur un intervalle non borné 95
9.1. Définition de A-stabilité . . . 95
9.2. Lien entre A-stabilité et Problèmes Stiff . . . 99
9.3. Étude de la A-stabilité pour un schéma à un pas . . . 100
9.4. Étude de la A-stabilité pour un schéma à deux pas . . . 107
9.5. Vérification empirique des conditions de A-stabilité . . . 113
9.6. Annexe : A-stabilité d'un système . . . 118

10. Convergence et A-stabilité des schémas multi-pas linéaires 119
10.1. Rappels . . . 119
10.1.1. Schémas multi-pas linéaires (MPL) . . . 119
10.1.2. Généralités : convergence et stabilités . . . 119
10.2. Polynômes caractéristiques . . . 120
10.3. Convergence d'un schéma multi-pas linéaire . . . 121
10.3.1. Consistance . . . 121
10.3.2. Zéro-stabilité . . . 126
10.3.3. Ordre de convergence maximale d'une méthode MPL . . . 127

10.4. A-stabilité . . . 133
10.5. Exercice de synthèse : consistance, zéro-stabilité et A-stabilité des schémas EE, EI, CN . . . 136
10.6. Propriétés des schémas Prédicteur-Correcteur basés sur les méthodes multi-pas . . . 138
10.7. Méthodes cycliques composites : comment surmonter les limites de Dahlquist . . . 138
10.8. Annexe : comment tracer les régions de A-stabilité ? . . . 138

11. Schémas de Runge-Kutta 149
11.1. Schémas . . . 149
11.2. Matrice de Butcher . . . 150
11.3. Convergence et ordre de convergence . . . 151
11.3.1. Calcul explicite de l'ordre de consistance avec Sympy . . . 155
11.4. A-stabilité . . . 159
11.5. Exercice : schémas RK à 1 étage . . . 164
11.5.1. Ordre maximal de convergence . . . 165
11.5.2. Schémas explicites . . . 165
11.5.3. Étude de la A-stabilité . . . 166
11.6. Exercice : schémas RK à 2 étages . . . 166
11.6.1. Schémas semi-implicites . . . 168
11.6.2. Schémas explicites . . . 170
11.6.3. Étude de la A-stabilité des schémas explicites . . . 171
11.7. Quelques schémas RK à 3 étages explicites . . . 173
11.8. Quelques schémas RK à 4 étages explicites . . . 176
11.9. Quelques schémas RK à 5 étages explicites . . . 179
11.10. Quelques schémas RK à 6 étages explicites . . . 181
11.11. Méthodes adapatatives . . . 183
11.11.1. CAS 1 : un unique schéma, mais avec deux pas de discrétisation différents . . . 183
11.11.2. CAS 2 : deux schémas d'ordre différent (méthode de Runge-Kutta emboîtée) . . . 187
11.11.3. Méthode d'Euler explicite adaptative . . . 207
11.11.4. Méthode adaptative avec deux schémas distincts . . . 210

III TP 215

12. Calculs analytiques avec le module sympy 217
12.1. Exemples . . . 217
12.1.1. Exemple de résolution analytique d'une équation différentielle d'ordre 1 . . . 217
12.1.2. Exemple de résolution analytique d'un système d'équations différentielles d'ordre 1 . . . 219
12.1.3. Exemple de résolution analytique d'une équation différentielle d'ordre 2 . . . 221
12.2. Exercices . . . 225
12.2.1. Exercice (EDO d'ordre 1) . . . 225
12.2.2. Exercice (système et EDO d'ordre 2) . . . 226

13. Calcul approché avec la fonction `odeint` du module `SciPy` 231
13.1. Principe d'utilisation de `odeint` . . . 232
13.2. Champ de vecteurs et lignes de courant pour une EDO du premier ordre . . . 233
13.3. Exemples . . . 233
13.3.1. Exemple de résolution approchée d'une équation différentielle d'ordre 1 . . . 233
13.3.2. Exemple de résolution approchée d'un système d'équations différentielles d'ordre 1 . . . 236
13.3.3. Exemple de résolution approchée d'une équation différentielle d'ordre 2 . . . 240
13.4. Exercices . . . 242
13.4.1. Exercice (EDO d'ordre 1) . . . 242
13.4.2. Exercice (système) . . . 243
13.4.3. Exercice (quiver - système) . . . 245
13.4.4. Exercice (système) : étude de la stabilité . . . 247

14. TP 2 - Implémentation des schémas “classiques” et étude de la convergence 251

14.1. Exercice (à compléter) : Implémentation des Schémas Classiques . . . 251
14.2. Exercice (à compléter) : Estimation de l’Ordre de Convergence . . . 256
14.3. Exercice : Fonction intégrale . . . 259
14.4. Exercice : système de 2 EDO avec Invariant . . . 262

15. TP 3 - Implémentation de schémas 273

15.1. Exercice (à compléter) : Implémentation et Comparaison de Schémas . . . 273
15.2. Schémas explicites . . . 274
15.2.1. Schéma AB-1 . . . 274
15.2.2. Schéma AB-2 . . . 274
15.2.3. Schéma AB-3 . . . 275
15.2.4. Schéma AB-4 . . . 275
15.2.5. Schéma AB-5 . . . 276
15.2.6. Schéma N-2 . . . 276
15.2.7. Schéma N-3 . . . 277
15.2.8. Schéma N-4 . . . 277
15.2.9. Schéma EM . . . 278
15.2.10. Schéma RK-4.1 . . . 278
15.2.11. Schéma RK-6.5 . . . 279
15.2.12. Schéma RK-7.6 . . . 279
15.3. Schémas implicites . . . 280
15.3.1. Schéma EI . . . 280
15.3.2. Schéma CN = AM-1 . . . 280
15.3.3. Schéma AM-2 . . . 281
15.3.4. Schéma AM-3 . . . 281
15.3.5. Schéma AM-4 . . . 282
15.3.6. Schéma AM-5 . . . 283
15.3.7. Schéma BDF-2 . . . 283
15.3.8. Schéma BDF-3 . . . 284
15.3.9. Schéma RK1_M . . . 285
15.3.10. Schéma DIRK-2 . . . 285
15.3.11. Schéma Randau-IIa-2 . . . 286
15.4. Schémas predicteur-correcteur . . . 287
15.4.1. Schéma de Heun . . . 287
15.4.2. Schéma AM-2 AB-1 . . . 287
15.4.3. Schéma AM-3 AB-2 . . . 287
15.5. Solution exacte . . . 288
15.6. Tests . . . 288
15.6.1. Méthode 1 . . . 289
15.6.2. Méthode 2 . . . 293
15.6.3. Méthode 2bis . . . 295

16. TP 4 - Étude de la convergence 297

16.1. Exercice (à compléter) : Estimation de l’Ordre de Convergence des Schémas Numériques . . . 297
16.2. Schémas explicites . . . 298
16.2.1. Schéma EE . . . 298
16.2.2. Schéma AB-2 . . . 299
16.2.3. Schéma AB-3 . . . 299
16.2.4. Schéma AB-4 . . . 299
16.2.5. Schéma AB-5 . . . 300
16.2.6. Schéma N-2 . . . 300
16.2.7. Schéma N-3 . . . 301
16.2.8. Schéma N-4 . . . 301
16.2.9. Schéma EM . . . 302
16.2.10. Schéma RK4-1 . . . 302

16.2.11 Schéma RK6_5 . . . 302
16.2.12 Schéma RK7_6 . . . 303
16.3. Schémas implicites . . . 303
16.3.1. Schéma EI . . . 303
16.3.2. Schéma CN . . . 304
16.3.3. Schéma AM-2 . . . 304
16.3.4. Schéma AM-3 . . . 305
16.3.5. Schéma AM-4 . . . 305
16.3.6. Schéma AM-5 . . . 306
16.3.7. Schéma MS-2 = MS-3 . . . 306
16.3.8. Schéma MS-4 . . . 307
16.3.9. Schéma MS-5 . . . 307
16.3.10 Schéma BDF2 . . . 308
16.3.11 Schéma BDF3 . . . 308
16.3.12 Schéma BDF4 . . . 309
16.3.13 Schéma BDF5 . . . 309
16.3.14 Schéma RK1_M . . . 310
16.3.15 Schéma Randau-IIa-2 . . . 311
16.4. Schémas predicteur-correcteur . . . 311
16.4.1. Schéma de Heun (AM-1 AB-1) . . . 311
16.4.2. Schéma AM-4 AB-2/3/4/5 . . . 312
16.5. Convergence . . . 313
16.5.1. Méthode 1 . . . 314
16.5.2. Méthode 2 : version compacte . . . 318
16.5.3. Méthode 2 bis : version compacte . . . 319

IV TD 321

17. TD1 - Exercices sur les schémas à un pas 323
17.1. Exercice : Modèle de Malthus (schéma d'Euler explicite) . . . 323
17.2. Exercice : Modèle de Verhulst (schéma d'Euler explicite) . . . 326
17.3. Exercice : concentration (schéma d'Euler implicite) . . . 329
17.4. Exercice : A-stabilité du schéma de Crank-Nicolson . . . 329
17.5. Exercice : A-stabilité d'un schéma . . . 332
17.6. Exercice : A-stabilité d'un schéma . . . 334
17.7. Exercice : schéma d'Euler explicite et zéro-stabilité . . . 336
17.8. Exercice : A-stabilité d'un schéma . . . 338
17.9. Exercice : A-stabilité du ϑ-schéma . . . 340
17.10. Exercice : Ordre d'un schéma . . . 341

18. TD2 - Exercices sur les schémas multi-pas linéaires et les schémas prédicteur-correcteur 343
18.1. PARTIE I : Étude de schémas multi-pas linéaires . . . 343
18.1.1. Aide-mémoire . . . 343
18.1.2. Exercice : consistance d'un schéma à deux pas explicite . . . 346
18.1.3. Exercice : convergence d'un schéma à 2 pas implicite . . . 347
18.1.4. Exercice : convergence d'un schéma à 3 pas explicite . . . 348
18.1.5. Exercice : schéma à deux pas d'ordre 4 . . . 349
18.1.6. Exercice : convergence d'un schéma à 2 pas implicite . . . 353
18.1.7. Exercice : convergence d'un schéma à 2 pas explicite . . . 358
18.1.8. Exercice : convergence d'un schéma à 2 pas explicite . . . 363
18.2. PARTIE II : Construction de schémas multi-pas linéaires . . . 368
18.2.1. Exercice : Interpolation, Quadrature et EDO . . . 368
18.2.2. Exercice : Interpolation, Quadrature et EDO . . . 369

18.2.3. Exercice : Construction d'un schéma . . . 371
18.2.4. Exercice : construction d'un schéma . . . 372
18.2.5. Exercice : Construction du schéma de Boole-Moulton . . . 373
18.3. PARTIE III : Schémas Prédicteur-Correcteur basés sur deux schémas multi-pas linéaires . . . 376
18.3.1. Aide-mémoire . . . 376
18.3.2. Exercice : convergence d'un schéma predictor-corrector basé sur deux schémas . . . 377
18.3.3. Exercice : convergence d'un schéma predictor-corrector basé sur deux schémas . . . 381
18.3.4. Exercice : Construction d'un schéma Predictor-Corrector . . . 384

19. TD3 - Exercices sur les schémas de Runge-Kutta 387
19.1. Aide-mémoire . . . 387
19.1.1. Conditions pour un schéma explicite avec s = 6 étages . . . 394
19.2. Exercices . . . 396
19.2.1. Exercice : schéma de Ralston ($s = 2$, explicite) . . . 396
19.2.2. Exercice : schéma de Radau IA ($s = 2$, implicite) . . . 400
19.2.3. Exercice : schéma de Radau IIA ($s = 2$, implicite) . . . 405
19.2.4. Exercice ($s = 3$, explicite) . . . 409
19.2.5. Exercice ($s = 3$, implicite) . . . 414
19.2.6. Exercice ($s = 3$, explicite avec paramètres) . . . 419
19.3. Catalogue de quelques schémas Runge-Kutta rencontrés dans la littérature . . . 423
19.3.1. Schémas à $s = 1$ étage . . . 424
19.3.2. Schémas à $s = 2$ étages . . . 426
19.3.3. Schémas à $s = 3$ étages . . . 428
19.3.4. Schémas à $s = 4$ étages . . . 430
19.3.5. Schémas à $s = 5$ étages . . . 431
19.3.6. Schémas à $s = 6$ étages . . . 433
19.3.7. Schémas à $s = 7$ étages . . . 434

20. TD4 - Révisions 435
20.1. Exercice : Étude d'un problème mathématiquement mal posé . . . 435
20.2. Exercice : Étude d'un problème numériquement mal posé . . . 439
20.3. Exercice : EDO d'ordre 2 . . . 441
20.4. Exercice : A-stabilité du schéma d'Euler expicite dans le cas d'un système . . . 442
20.5. Exercice : Étude théorique et numérique du mouvement d'un pendule . . . 444

V Annales 455

21. 2025 Contrôle Continu 457
21.1. EXERCICE 1 - sujet A : Système . . . 457
21.2. EXERCICE 1 - sujet B : Système . . . 461
21.3. EXERCICE 2 - sujet A : Schéma MPL . . . 465
21.4. EXERCICE 2 - sujet B : Schéma MPL . . . 466
21.5. EXERCICE 3 - sujet A : Schéma RK semi-implicite à 2 étages . . . 467
21.6. EXERCICE 3 - sujet B : Schéma RK semi-implicite à 2 étages . . . 472

22. 2025 Contrôle Terminale session 1 479
22.1. EXERCICE 1 : Schéma multi-pas linéaire explicite . . . 479
22.2. EXERCICE 2 : Étude de l'A-stabilité d'un schéma RK explicite à 3 étages . . . 482
22.3. EXERCICE 3 : schéma prédicteur-correcteur . . . 486

23. 2025 Contrôle Terminale session 2 491
23.1. EXERCICE 1 : schéma RK explicite à 3 étage . . . 491

23.2. EXERCICE 2 : schéma multi-pas linéaire explicite 497

24. 2024 Contrôle Continu 501
24.1. EXERCICE 1 : choix d'un schéma selon le problème de Cauchy donné 501
24.2. EXERCICE 2 : schémas de Runge-Kutta à 4 étages explicite d'ordre maximal 503
24.3. EXERCICE 3 : schémas MPL à 2 pas d'ordre maximal 511

25. 2024 Contrôle Terminal session 1 519
25.1. EXERCICE 1 : Choix d'un schéma 519
25.2. EXERCICE 2 : Schémas MPL à 3 pas d'ordre maximal 522
25.3. EXERCICE 3 : Étude d'un schéma RK implicite à 2 étages 534

26. 2024 Contrôle Terminal session 2 543
26.1. EXERCICE 1 : Étude de l'amour entre Roméo et Juliette 543
26.2. EXERCICE 2 : Schéma MPL explicite avec paramètres 546
26.3. EXERCICE 3 : Schéma RK à 3 étages explicite 550

27. 2023 Contrôle Continu 559
27.1. EXERCICE 1 : choix d'un schéma 559
27.2. EXERCICE 2 : étude d'un schéma multipas 560
27.3. EXERCICE 3 : étude d'un schéma RK 563

28. 2023 Contrôle Terminal session 1 571
28.1. EXERCICE 1 : étude d'un système 571
28.2. EXERCICE 2 : étude d'un schéma multipas 574
28.3. EXERCICE 3 : étude d'un schéma RK 577

29. 2023 Contrôle Terminal session 2 587
29.1. EXERCICE 1 : choix d'un schéma 587
29.2. EXERCICE 2 : étude d'un schéma Runge-Kutta 591

30. 2022 Contrôle Continu 601
30.1. EXERCICE 1 : étude d'un schéma multipas 601
30.2. EXERCICE 2 : étude d'un système 605
30.3. EXERCICE 3 : étude de la A-stabilité d'un schéma 610

31. 2022 Contrôle Terminal session 1 615
31.1. EXERCICE 1 : Méthode d'Euler exponentielle 615
31.2. EXERCICE 2 : étude d'un schéma Runge-Kutta 619
31.3. EXERCICE 3 : étude d'un système Hamiltonien 627

32. 2022 Contrôle Terminal session 2 633
32.1. EXERCICE 1 : Méthode d'Euler exponentielle et problèmes raides 633
32.2. EXERCICE 2 : étude d'un schéma Runge-Kutta 636
32.3. EXERCICE 3 : étude d'un schéma multipas 642

33. 2021 Contrôle Continu 647
33.1. EXERCICE 1 - sujet A : implémentation d'un schéma pour un système 647
33.2. EXERCICE 1 - sujet B : implémentation d'un schéma pour un système 650
33.3. EXERCICE 2 - sujet A : étude d'un schéma multipas 653

33.4. EXERCICE 2 - sujet B : étude d'un schéma multipas . . . 657

34. 2021 - Contrôle Terminal session 1 661
34.1. EXERCICE 1 : étude d'un schéma RK . . . 661
34.2. EXERCICE 2 : choix d'un schéma . . . 668
34.3. EXERCICE 3 : choix d'un schéma . . . 670

35. 2021 Contrôle Terminal session 2 675
35.1. EXERCICE 1 : étude d'un schéma . . . 675
35.2. EXERCICE 2 : étude d'un schéma multipas . . . 682

36. 2020 Devoir Maison 687
36.1. EXERCICE 1 : implémentation et comparaison de schémas . . . 687
36.2. EXERCICE 2 : étude d'un schéma RK semi-implicite à 2 étages . . . 693
36.3. EXERCICE 3 : étude d'un schéma Predictor-Corrector . . . 701

37. 2019 Contrôle Continu 707
37.1. EXERCICE : schéma de Runge-Kutta . . . 707
37.1.1. Correction : écriture schéma . . . 708
37.1.2. Correction : étude de la A-stabilité . . . 709
37.1.3. Correction : implémentation et test . . . 711
37.1.4. Correction : étude de l'ordre . . . 713
37.1.5. Correction : solution exacte . . . 714
37.1.6. Correction : étude empirique de l'ordre de convergence . . . 715

38. 2019 Contrôle Terminale session 1 719
38.1. EXERCICE 1 : choix d'un schéma . . . 719
38.2. EXERCICE 2 : choix d'un schéma . . . 719
38.3. EXERCICE 3 : schéma predictor-corrector . . . 719
38.4. EXERCICE 4 : Schéma Runge-Kutta . . . 725

39. 2019 Contrôle Terminale session 2 731
39.1. EXERCICE 1 : Choix d'un schéma . . . 731
39.2. EXERCICE 2 : Circuit LC . . . 733

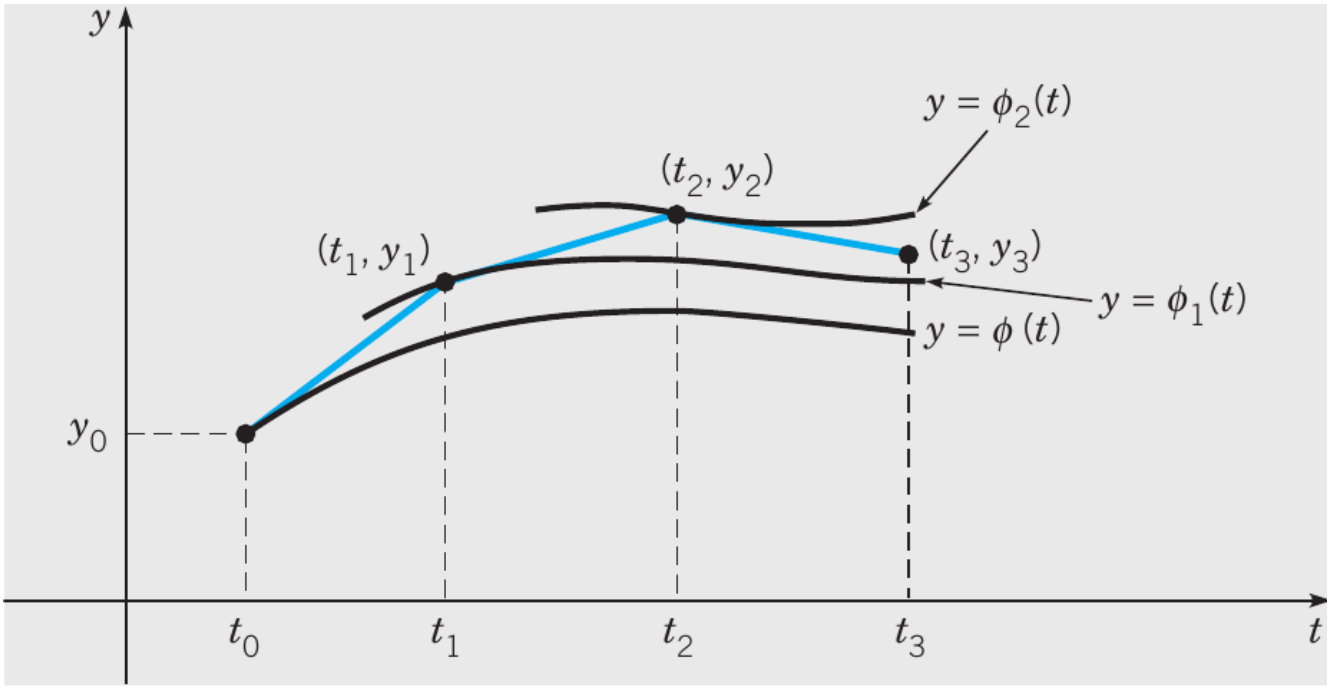
VI Downloads 741

40. Notebook Originaux à télécharger 743
40.1. Cours . . . 743
40.2. Travaux Pratiques (énoncés) . . . 744
40.3. Travaux Pratiques (avec corrections) . . . 744
40.4. Travaux Dirigés (avec corrections) . . . 744
40.5. Annales (avec corrections) . . . 744
40.5.1. A.A. 2025 . . . 745
40.5.2. A.A. 2024 . . . 745
40.5.3. A.A. 2023 . . . 745
40.5.4. A.A. 2022 . . . 746
40.5.5. A.A. 2021 . . . 746
40.5.6. A.A. 2020 . . . 746
40.5.7. A.A. 2019 . . . 747

Chapitre 1.

Schémas numériques pour les problèmes de Cauchy

L3 Mathématiques - Université de Toulon



Première partie

Introduction

Chapitre 2.

Syllabus du cours DSSC-6 : Schémas numériques pour les problèmes de Cauchy

2.1. Introduction : le parcours DSSC

Depuis septembre 2024, l'université de Toulon propose, dans sa licence de Mathématiques, un parcours nommé *Data Science and Scientific Computing*. Il se situe au croisement des mathématiques et de l'informatique. L'objectif principal est d'apprendre à formuler des questions de manière à ce que le calcul informatique puisse fournir des réponses (exactes ou approchées) satisfaisantes.

Ce parcours sur trois ans fournit une base de connaissances et de compétences pratiques en analyse de données et en calcul scientifique, en utilisant principalement **Python** et les **Notebook Jupyter**. On y apprend à manipuler, analyser et visualiser les données, ainsi qu'à résoudre des problèmes scientifiques et mathématiques complexes en utilisant les outils et bibliothèques appropriés :

- **DSSC-1** *Introduction à la programmation scientifique avec Python* : apprentissage des bases de la programmation en Python, notamment les structures de contrôle, les fonctions et les types de données, ainsi que l'utilisation de Matplotlib pour créer des visualisations.
- **DSSC-2** *Approximations et simulations* : étude des bases de l'analyse numérique, avec des comparaisons aux résultats issus de la bibliothèque SciPy, pour résoudre des problèmes scientifiques et d'ingénierie : interpolations, intégrations numériques, zéros d'une fonction. On apprendra également à utiliser des Notebook Jupyter pour expliquer et documenter les calculs.
- **DSSC-3** *Calcul formel avec SymPy* : familiarisation avec SymPy pour réaliser des calculs mathématiques symboliques comme la manipulation d'expressions et la résolution d'équations et sytèmes. Application en analyse mathématique (calcul de limites, dérivées, intégrales symboliques dans $\mathbb{R}^n$) et en algèbre linéaire (applications linéaires, noyeaux, etc.). On apprendra également à rédiger des rapports en LaTeX.
- **DSSC-4** *Manipulation avancée des tableaux avec NumPy* : approfondissement des connaissances sur le calcul matriciel avec NumPy pour des calculs performants.
- **DSSC-5** *Manipulation de données avec Pandas : une initiation* : utilisation de Pandas pour manipuler efficacement les jeux de données, incluant l'importation, le nettoyage, la transformation, le filtrage, l'agrégation et la gestion des valeurs manquantes.
- **DSSC-6** *Approximations d'équations différentielles* : étude des schémas numériques pour l'approximation de problèmes de Cauchy.

Ce polycopié est associé au sixième et dernier cours, initiant à l'étude et à l'implémentation de schémas numériques pour l'approximation de problèmes de Cauchy.

2.2. Déroulement du cours DSSC-6 : Schémas numériques pour les problèmes de Cauchy

Une **équation différentielle ordinaire** (EDO) est une équation mathématique impliquant une ou plusieurs fonctions d'une variable indépendante ainsi que leurs dérivées. On parle d'**équation ordinaire** par opposition aux **équations différentielles partielles** (EDP), qui portent sur des fonctions de plusieurs variables indépendantes et incluent leurs dérivées partielles.

Les EDO et les EDP sont des outils puissants pour modéliser divers phénomènes naturels et systèmes issus de domaines tels que la physique, la biologie, l'économie ou l'ingénierie. En effet, dans de nombreuses applications

réelles, les fonctions représentent des grandeurs physiques, les dérivées traduisent leurs variations, et l'équation établit un lien entre ces éléments. Ce rôle fondamental explique l'importance des équations différentielles dans l'étude des systèmes dynamiques et de nombreuses disciplines scientifiques.

En mathématiques pures, l'étude des équations différentielles se concentre principalement sur leurs **solutions**, c'est-à-dire l'ensemble des fonctions qui satisfont l'équation. Seules les équations les plus simples admettent des **solutions explicites** sous forme de formules. Cependant, même lorsque l'on ne peut pas résoudre une équation de manière analytique, il est souvent possible d'analyser certaines **propriétés** de ses solutions. Dans les cas où une solution exacte est hors de portée, des **méthodes numériques** permettent d'obtenir des approximations avec une précision contrôlée.

2.2.1. Objectifs

Ce cours se concentre sur l'étude et l'implémentation de **schémas numériques pour approcher la solution de problèmes de Cauchy**. Un problème de Cauchy est un problème de valeur initiale pour une EDO. Il consiste à trouver une solution qui satisfait une équation différentielle et une condition initiale. Les méthodes numériques pour résoudre les problèmes de Cauchy sont basées sur la discrétisation de l'intervalle de temps et sur l'approximation de la solution sur cet intervalle. Les méthodes les plus courantes sont les schémas multi-pas linéaires, les schémas Runge-Kutta et les schémas prédicteur-correcteur.

2.2.2. Plan des séances

- CM **6 séances de 2 heures** dans le bâtiment M
- TP **5 séances de 3 heures** dans le bâtiment U1
- TD **2 séances de 2 heures** dans le bâtiment M

Tous les supports de cours sont fournis sous forme de notebooks Jupyter et téléchargeables depuis la page en annexe (menu a gauche ci-contre).

2.2.3. Contenu du cours

- Fondements théoriques des problèmes de Cauchy bien posés mathématiquement, bien posés numériquement et bien conditionnés.
- Construction et analyse de schémas numériques :
  - Schémas multi-pas linéaires (MPL), notamment les schémas des type Adams-Moulton, Adams-Bashford, Nyström, Milne-Simpson et BDF.
  - Schémas Runge-Kutta (RK).
- Schémas prédicteur-correcteur (PC).
- Concepts étudiés :
  - Consistance et ordre de consistance (pour les schémas multipas et RK).
  - Zéro-stabilité (pour les schémas multipas).
  - Convergence.
  - Absolue-stabilité (pour les schémas à un pas et pour les schémas RK).

Lors des séances de travaux pratiques il faudra :

- implémenter chaque schéma étudié en cours et vérifier l'ordre théorique de convergence sur des problèmes pour lesquels la solution exacte est connue (la solution exacte pourra être obtenue par exemple via `sympy`) ;
- implémenter les schémas pour des systèmes d'EDO d'ordre 1 (ou des équations d'ordre supérieur à 1 après les avoir écrit comme des systèmes d'EDO d'ordre 1), avec si possible la recherche et la vérification de la conservation d'invariants théoriques.

2.2.4. Prérequis

- Connaissances de base en **analyse numérique** : interpolation, intégration numérique, zéros d'une fonction, etc. ; cf. **DSSC-2** ;
- **Bases de Python** : types primitifs (int, float, str), structures de données (listes, tuples, dictionnaires), conditions, boucles, fonctions, modules (numpy, matplotlib.pyplot) ; cf. **DSSC-1** ;
- Manipulation de **Notebook Jupyter** : cellules python *vs* markdown, écriture de formules mathématiques en LaTeX, exécution, sauvegarde, export, etc. ; cf. **DSSC-2** ;
- Connaissances de base de **NumPy** : manipulation de tableaux et extraction de sous-tableaux par *slices* ; cf. **DSSC-4**.

Pour manipuler les notebooks Jupyter, on utilisera l'éditeur VS Codium (au lieu de Jupyter Notebook dans le navigateur). Dans les salles de TP, Python 3.12 est installé.

Librairies Python utilisées

- `numpy` pour l'implémentation des schémas numériques.
- `matplotlib.pyplot` pour la visualisation des graphiques.
- Comparaison des résultats avec :
  - la solution exacte obtenue via `sympy`.
  - les solutions approchées obtenues avec `scipy.integrate`.

2.2.5. Évaluation

Le cours est évalué par un contrôle continu et un contrôle terminal.

- **Contrôle continu (CC)** : ~mardi 24 mars 2026~ mardi 7 avril 2026, durée 2 heures, en salle de TP.
- **Contrôle terminal (CT)** : mardi 28 avril 2026, durée 2 heures, en salle de TP.

Les deux évaluations comportent des exercices sur l'étude mathématique et l'implémentation en Python de schémas numériques avec vérification de leurs performances théoriques sur des problèmes de Cauchy donnés. Il faudra déposer sur Moodle un notebook `.ipynb` contenant l'étude mathématique (rédigée en Markdown), les implémentations en Python des schémas numériques demandés ainsi que l'analyse des résultats obtenus.

Vous pourrez utiliser le polycopié, les notebooks fournis dans la section "Sources" et vos notes de cours comme base pour vos implémentations, mais assurez-vous de bien comprendre chaque étape et d'adapter les réponses et le code selon les exigences spécifiques des exercices.

2.2.6. Informations système

```python
import sys  # only needed to determine Python version number
print(f"Python version: {sys.version}")

import numpy as np  # only needed to determine Numpy version number
print(f"Numpy version: {np.__version__}")

import scipy  # only needed to determine Scipy version number
print(f"Scipy version: {scipy.__version__}")

from datetime import datetime
print("Last Updated On:", datetime.now())
```

```
Python version: 3.12.3 (main, Jan  8 2026, 11:30:50) [GCC 13.3.0]
Numpy version: 1.26.4
Scipy version: 1.11.4
Last Updated On: 2026-03-30 09:17:52.313581
```

Deuxième partie

Cours

Chapitre 3.

Rappels

3.1. Équations Différentielles Ordinaires (EDO)

Les équations différentielles décrivent l'évolution de divers phénomènes dans de nombreux domaines. Elles expriment des relations impliquant une ou plusieurs dérivées d'une fonction inconnue. Lorsque ces dérivées sont toutes prises par rapport à une seule variable, on parle d'équation différentielle ordinaire (EDO), tandis que celles impliquant des dérivées partielles sont qualifiées d'équations aux dérivées partielles (EDP).

3.1.1. EDO d'ordre p

Une EDO se présente sous la forme générale :

$$F(y(t), y'(t), y''(t), \dots, y^{(p)}(t)) = g(t).$$

- Les inconnues sont une **fonction** y: $I \subset \mathbb{R} \to \mathbb{R}$ et son **intervalle de définition** I.
- Elle combine la fonction inconnue y et ses dérivées y', y'', ...,$y^{(p)}$ (où p est l'**ordre** de l'équation).

Si la fonction g, appelée "second membre" de l'équation, est nulle, l'équation est dite **homogène**.

Résoudre une équation différentielle consiste à trouver toutes les fonctions définies sur un intervalle $I \subset \mathbb{R}$ qui vérifient cette équation (on dit aussi *intégrer* l'équation différentielle).

Par le terme **solution générale** d'une EDO on désigne un représentant qui englobe l'ensemble des solutions possibles. L'une des solutions de l'EDO sera appelée *solution particulière*. On appelle *courbes intégrales* d'une EDO les courbes représentatives des solutions de l'équation.

> Exemple
>
> La résolution de l'équation différentielle $y'(t) = -y(t)$ consiste à trouver toutes les fonctions $y(t)$ telles que $y'(t) = -y(t)$ pour tout $t \in I$. On peut vérifier que $y(t) = ce^{-t}$ pour tout $t \in \mathbb{R}$ (où c est une constante réelle arbitraire) est une solution de l'EDO. En particulier, pour $c = 0$, on obtient la solution nulle.

3.1.1.1. Représentations graphiques pour une EDO d'ordre 1 : champ de vecteurs et lignes de courant

Bien qu'il soit très rare que l'on puisse résoudre explicitement une équation différentielle donnée, comprendre l'allure des solutions peut être facilité en examinant le champ de vecteurs correspondant. Les graphes des solutions d'une équation différentielle $y'(t) = \varphi(t, y(t))$ sont, par définition, tangents à leur vecteur vitesse $(1, y'(t))$, donc au vecteur dérivé $(1, \varphi(t, y(t))$. Même sans connaître explicitement les solutions, connaître la fonction en chaque point permet de représenter ces vecteurs tangents. En traçant un grand nombre de ces vecteurs uniformément répartis dans le plan, on obtient le champ de vecteurs associé à l'équation différentielle. Cette représentation permet souvent de deviner les graphes des solutions, ces courbes étant tangentes en tous leurs points aux vecteurs du champ.

- La fonction `quiver` du module `matplt.plotlib` permet de tracer un champ de vecteurs : `quiver(x_pos, y_pos, x_dir, y_dir, color)`
- La fonction `streamplt.plot` permet de tracer des lignes de courant représentant un champ vectoriel.

Utilisons-les pour obtenir les tracés associés à l'équation différentielle de notre exemple.

Cf. https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.quiver.html et https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.streamplot.html

```
# On représente en général un champ de vecteurs en des points régulièrement espacés de $
 ↪  I\times\mathbb{R} $ et en normalisant les vecteurs.
# Dessinons alors, en chaque point (t,y) de la grille un vecteur $\mathbf{V}_{t,y}$ de pente
 ↪  $\varphi(t,y)$ : le graphe de $f$ est tangent en chaque point $(t,y)$ au vecteur $\mathbf{V}_{t,y}$.
 ↪
# Remarquer qu'**on n'a pas besoin d'avoir résolu l'équation (analytiquement) pour pouvoir dessiner le
 ↪  champ de tangentes, et ceci permet parfois d'avoir une idée du comportement des solutions.**


import numpy as np
import matplotlib.pyplot as plt

phi = lambda t,y : -y

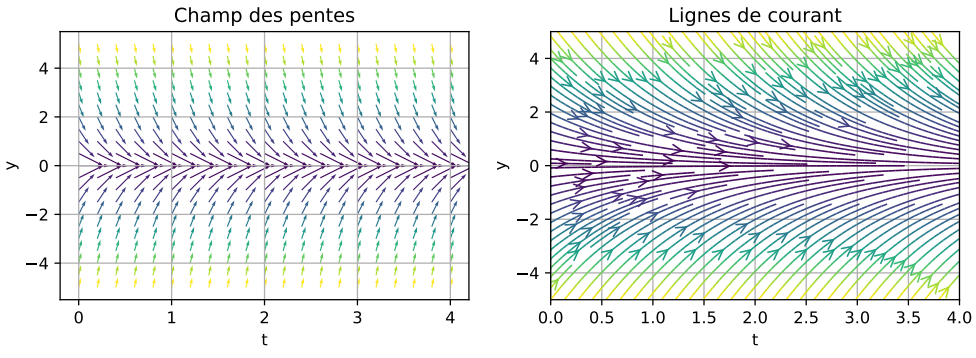
g1  = np.linspace(0,4,21)
g2  = np.linspace(-5,5,21)
T,Y = np.meshgrid(g1,g2)

# compute direction at each point
DT, DY = 1, phi(T,Y)          # compute growth rate on the plt.grid
# M = np.sqrt(DT**2+DY**2)         # norm growth rate
M = np.hypot(DT,DY)

plt.figure(figsize=(10,3))

# Champ des pentes
plt.subplot(1,2,1)
# plt.quiver(T,Y, DT/M, DY/M, M, pivot='mid')
plt.quiver(T,Y, DT/M, DY/M, M, cmap = plt.cm.viridis, scale_units='xy',scale=3.)
plt.grid()
plt.xlabel('t')
plt.ylabel('y')
plt.title('Champ des pentes');

# Lignes de courant
plt.subplot(1,2,2)
# plt.streamplot(T,Y, DT/M, DY/M, color=M)
plt.streamplot(T,Y, DT/M, DY/M, color=M, linewidth=1, cmap=plt.cm.viridis, density=2, arrowstyle='->',
 ↪  arrowsize=1.5)
plt.grid()
plt.xlabel('t')
plt.ylabel('y')
plt.title('Lignes de courant')
plt.show();
```



3.1.2. EDO d'ordre supérieur à 1 $\rightsquigarrow$ Système d'EDOs d'ordre 1

Une EDO d'ordre p est dite *normalisée* lorsqu'elle est exprimée sous la forme :

$$y^{(p)}(t) = f(g(t), y(t), y'(t), y''(t), \dots, y^{(p-1)}(t)).$$

Toute EDO normalisée d'ordre p peut être transformée en un système de p EDO d'ordre 1 de la manière suivante :

$$\begin{cases} z_1'(t) = z_2(t), \\ \dots \\ z_{p-1}'(t) = z_p(t), \\ z_p'(t) = f(g(t), z_1(t), z_2(t), \dots, z_{p-1}(t)). \end{cases}$$

Exemple

Considérons l'équation différentielle d'ordre 2 suivante :

$$y''(t) + ay'(t) + by(t) + c = 0$$

Introduisons les deux fonctions $z_1(t) = y(t)$ et $z_2(t) = y'(t)$. On obtient alors le système :

$$\begin{cases} z_1'(t) = z_2(t), \\ z_2'(t) = -az_2(t) - bz_1(t) - c. \end{cases}$$

3.1.2.1. Représentations graphiques pour un système autonome de 2 EDO d'ordre 1 : portrait de phase

Comme dans le cas d'une EDO scalaire, il est souvent difficile de résoudre explicitement un système différentiel. Cependant, on peut comprendre le comportement qualitatif des solutions en représentant le **champ de vecteurs** associé.

Considérons un système autonome (les fonctions f et g ne dépendent que de la position et jamais du temps t) :

$$\begin{cases} x'(t) = f(x, y) \\ y'(t) = g(x, y) \end{cases}$$

Les solutions sont des courbes $(x(t), y(t))$ du plan.
En chaque point (x, y), le vecteur vitesse est :

$$\mathbf{F}(x, y) = (f(x, y), g(x, y)).$$

Ainsi, **les trajectoires sont tangentes en tout point au champ de vecteurs**.
La représentation de ces trajectoires dans le plan (x, y) s'appelle le **portrait de phase**.

Comme précédemment, on n'a pas besoin d'avoir résolu analytiquement le système pour tracer le champ de vecteurs, ce qui permet d'anticiper le comportement global des solutions.

Par exemple, considérons le système suivant décrivant un oscillateur harmonique linéaire :

$$\begin{cases} x' = y \\ y' = -x \end{cases}$$

```
# On représente un champ de vecteurs dans le plan de phase (x,y).
# En chaque point (x,y) de la grille, on associe le vecteur
# V_{x,y} = (f(x,y), g(x,y)).
# Les trajectoires sont tangentes à ces vecteurs.
# Remarquer qu'**on n'a pas besoin d'avoir résolu le système
# analytiquement pour tracer son portrait de phase.**
```

```
import numpy as np
import matplotlib.pyplot as plt

f = lambda x,y : y
g = lambda x,y : -x

g1 = np.linspace(-3,3,25)
g2 = np.linspace(-3,3,25)
X,Y = np.meshgrid(g1,g2)

# Calcul du champ vectoriel
DX = f(X,Y)
DY = g(X,Y)

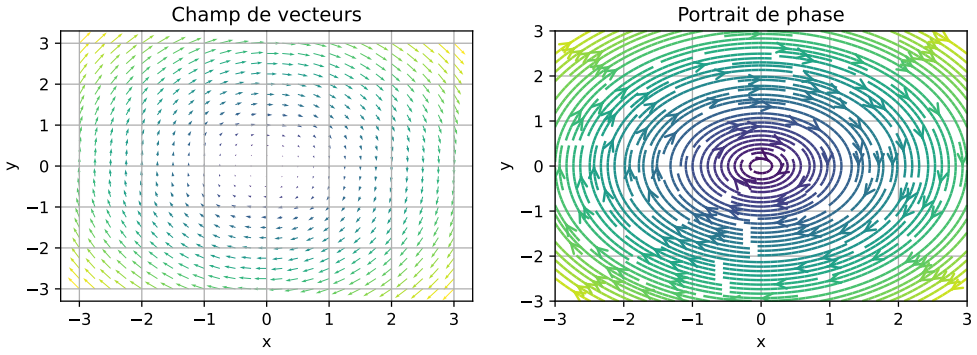
# Norme des vecteurs
M = np.hypot(DX,DY)

plt.figure(figsize=(10,3))

# Champ de vecteurs
plt.subplot(1,2,1)
plt.quiver(X,Y, DX, DY, M, cmap=plt.cm.viridis, scale_units='xy')
plt.grid()
plt.xlabel('x')
plt.ylabel('y')
plt.title('Champ de vecteurs')

# Lignes de courant (portrait de phase)
plt.subplot(1,2,2)
plt.streamplot(X,Y, DX, DY, color=M,
               cmap=plt.cm.viridis,
               density=2,
               arrowstyle='->',
               arrowsize=1.5)
plt.grid()
plt.xlabel('x')
plt.ylabel('y')
plt.title('Portrait de phase')

plt.show()
```



On observe que les trajectoires sont des courbes fermées autour de l'origine : le point $(0, 0)$ est un *centre*, et les solutions sont périodiques.

3.1.3. Conditions Initiales

Les EDO ont généralement une infinité de solutions. Pour déterminer la solution appropriée qui représente le problème physique, il est nécessaire de prendre en compte des données supplémentaires.

Définition

Soit une EDO d'ordre p. Une condition initiale (CI) est un ensemble de relations du type :

$$y(t_0) = y_0, \quad y'(t_0) = y'_0, \dots, \quad y^{(p-1)}(t_0) = y_0^{(p-1)}$$

qui imposent en t_0 les valeurs respectives de la fonction inconnue et de ses dérivées.

3.2. Problème de Cauchy

Le problème associé à une EDO avec une condition initiale est connu sous le nom de *problème de Cauchy* :

Définition : Problème de Cauchy

Considérons un intervalle $I \subset \mathbb{R}$, un point t_0 dans I, une fonction $\varphi \colon I \times \mathbb{R} \to \mathbb{R}$ continue. Le *problème de Cauchy* est défini comme la recherche d'une fonction réelle $y \in \mathcal{C}^1(I)$ telle que

$$\begin{cases} y'(t) = \varphi(t, y(t)), & \forall t \in I, \\ y(t_0) = y_0. \end{cases}$$

avec y_0 une valeur donnée appelée *condition initiale*.

Nous concentrerons notre analyse principalement sur le cas où une seule EDO est présente, c'est-à-dire le cas scalaire.

Résoudre un problème de Cauchy, c'est rechercher toutes les fonctions, définies sur un intervalle $I \subset \mathbb{R}$, qui satisfont l'équation différentielle tout en vérifiant la condition initiale. Cela soulève des questions naturelles telles que :

- trouver toutes les fonctions solutions de l'EDO,
- parmi ces fonctions, choisir celles qui respectent la CI (existence ? unicité ?),
- étudier le domaine de validité (pour chaque fonction trouvée, déterminer le plus grand intervalle contenant t_0).

3.2.1. Intervalle de Validité et Solution Maximale

En règle générale, lorsqu'une équation différentielle est munie d'une condition initiale $y(t_0) = y_0$, nous recherchons un intervalle I qui contient t_0, sur lequel une solution existe, et qui soit *le plus grand possible*. Ce domaine est appelé l'*intervalle de validité* de la solution. Une solution définie sur cet intervalle le plus large possible est appelée *solution maximale*.

Exemple : existence et unicité sur $\mathbb{R}$

Considérons $\varphi(t, y) = 3t - 3y$ et $y_0 = \alpha$. La solution est :

$$y(t) = (\alpha + \frac{1}{3})e^{-3t} + t - \frac{1}{3}.$$

Dans cet exemple, il existe une et une seule solution définie sur $\mathbb{R}$.

Dans cet exemple, il existe une et une seule solution définie sur $\mathbb{R}$. Les choses ne se passent pas toujours si bien. Les exemples ci-dessous montrent que l'étude mathématique de l'existence et de l'unicité des solutions d'un problème de Cauchy peut être une affaire délicate.

Exemple : existence sur un intervalle fini

Pour $\varphi(t,y) = (y(t))^3$ et $y_0 = 1$, la solution est :

$$y(t) = \frac{1}{\sqrt{1-2t}}$$

Cette solution est définie seulement pour $t < \frac{1}{2}$. Cet exemple montre qu'*un problème de Cauchy n'a pas toujours une solution pour tout* $t \in [0;+\infty[$ puisqu'ici la solution explose lorsque t tend vers la valeur $1/2$ (en effet, nous avons $\lim_{t\to(1/2)^-} y(t) = +\infty$) : le graphe de la solution a une asymptote verticale en $t = 1/2$. On parle d'*explosion de la solution en temps fini* ou encore de *barrière*.

Cet exemple illustre un phénomène général : pour une solution d'une EDO, l'unique manière de ne pas être définie sur l'ensemble des réels $\mathbb{R}$ est de présenter une asymptote verticale.

Exemple : non unicité

On se donne $\varphi(t,y(t)) = \sqrt[3]{y(t)}$ et $y_0 = 0$. On cherche une fonction $y\colon t \in \mathbb{R}^+ \mapsto y(t) \in \mathbb{R}$ qui satisfait :

$$\begin{cases} y'(t) = \sqrt[3]{y(t)}, & \forall t > 0, \\ y(0) = 0. \end{cases}$$

On vérifie que les trois fonctions $y_1(t) = 0$ et $y_{2,3}(t) = \pm\sqrt{8t^3/27}$, pour tout $t \geq 0$, sont solutions du problème de Cauchy donné.

Cet exemple montre qu'*un problème de Cauchy n'a pas nécessairement de solution unique* (dans cet exemple, il y en a trois).

Exemple : infinité de solutions

On se donne $\varphi(t,y(t)) = |y(t)|^\alpha$ avec $\alpha \in]0;1[$ et $y_0 = 0$. On cherche une fonction $y\colon t \in \mathbb{R}^+ \mapsto y(t) \in \mathbb{R}$ qui satisfait :

$$\begin{cases} y'(t) = |y(t)|^\alpha, & \forall t > 0, \\ y(0) = 0. \end{cases}$$

On vérifie que, pour tout $c \in \mathbb{R}^+$, les fonctions :

$$y_c(t) = \begin{cases} (1-\alpha)^{1/(1-\alpha)}(x-c)^{1/(1-\alpha)} & \text{si } x \geq c, \\ 0 & \text{si } 0 \leq x \leq c \end{cases}$$

sont solutions du problème de Cauchy donné.
Notons que pour $\alpha \geq 1$, le problème de Cauchy donné admet une et une seule solution, la fonction $y(t) = 0$ pour tout $t \in \mathbb{R}^+$.

Dans ce cours, nous étudierons uniquement des problèmes de Cauchy ayant une unique solution sur l'intervalle indiqué.

3.3. Méthodes de Résolution

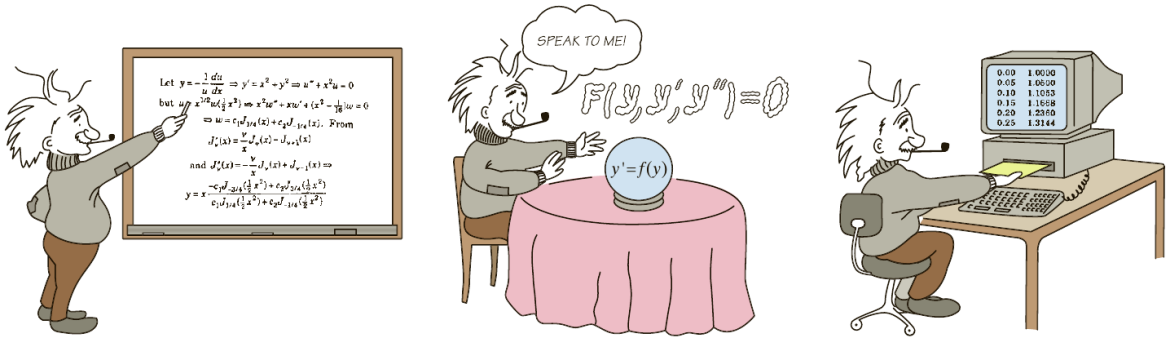
Seules les équations différentielles les plus simples peuvent être résolues par des **formules explicites**.

Dans certains cas, on ne peut exprimer la solution que sous forme implicite. C'est le cas par exemple de l'EDO $y'(t) = \dfrac{y(t)-t}{y(t)+t}$ dont les solutions vérifient la relation implicite

$$\frac{1}{2}\ln(t^2+y^2(t)) + \arctan\left(\frac{y(t)}{t}\right) = C,$$

où C est une constante arbitraire.
Dans d'autres cas, on ne parvient même pas à représenter la solution sous forme implicite. C'est le cas par exemple de l'EDO $y'(t) = e^{-t^2}$ dont les solutions ne peuvent pas s'écrire comme composition de fonctions élémentaires.

Certaines **propriétés** des solutions d'une équation différentielle donnée peuvent être déterminées sans trouver leur forme exacte, et c'est le sujet de l'étude qualitative des EDO. Enfin, si une formule explicite n'est pas disponible, la solution peut au moins être **approchée numériquement**.



(a) analytical **(b)** qualitative **(c)** numerical

Dans cette partie nous allons montrer comment calculer la solution générale de trois types d'EDO d'ordre 1 : les EDO à variables séparables, les EDO linéaires et les EDO de Bernoulli.

3.3.1. EDO du premier ordre à variables séparables

Lorsque l'équation est de la forme

$$f(y(x))y'(x) = g(x)$$

où f et g sont des fonctions données dont on connaît des primitives F et G, on a

$$\underbrace{\int f(y(x))y'(x)\mathrm{d}x}_{=\int f(u)\mathrm{d}u=F(u)} = \underbrace{\int g(x)\mathrm{d}x}_{G(x)+C}$$

donc

$$F(y(x)) = G(x) + C \qquad \text{où } C \in \mathbb{R},$$

et, si F possède une fonction réciproque F^{-1}, on en déduit

$$y(x) = F^{-1}(G(x) + C),$$

relation qui donne toutes les solutions de l'équation. Cette solution générale dépend de la constante d'intégration C.

> Astuce mnémotechnique
>
> En pratique, étant donné que $y'(x) = \mathrm{d}y/\mathrm{d}x$, on peut écrire l'équation $f(y(x))y'(x) = g(x)$ sous la forme
>
> $$f(y)\mathrm{d}y = g(x)\mathrm{d}x,$$
>
> puis intégrer formellement les deux membres

$$\int f(y)\mathrm{d}y = \int g(x)\mathrm{d}x,$$

pour obtenir $F(y) = G(x) + C$ et exprimer y en fonction de x.

⚠ Exemple

On veut résoudre l'équation différentielle

$$y'(x) = xy(x)$$

sur des intervalles à préciser. Il s'agit d'une EDO du premier ordre à variables séparables :

- **Recherche des solutions constantes.**
  Si $y(x) = A$ pour tout x, alors $y'(x) = 0$ pour tout x et l'EDO devient $0 = xA$ pour tout x. Par conséquent $A = 0$: la fonction $y(x) = 0$ pour tout x est l'unique solution constante de l'EDO.
- **Recherche des solutions non constantes.**
  La fonction $y(x) = 0$ pour tout x étant solution, toute autre solution $x \mapsto y(x)$ sera donc non nulle. On peut alors diviser l'EDO par y et procéder formellement comme suit :

$$\begin{aligned}
\frac{y'(x)}{y(x)} = x \quad &\Longrightarrow \quad \frac{\frac{\mathrm{d}y}{\mathrm{d}x}}{y(x)} = x \\
&\Longrightarrow \quad \frac{\mathrm{d}y}{y} = x\mathrm{d}x \\
&\Longrightarrow \quad \int \frac{1}{y}\mathrm{d}y = \int x\mathrm{d}x \\
&\Longrightarrow \quad \ln|y| = \frac{x^2}{2} + C \text{ avec } C \in \mathbb{R}.
\end{aligned}$$

Ainsi, toute solution non nulle est de la forme

$$y(x) = De^{x^2/2} \qquad \text{avec } D \in \mathbb{R}^*.$$

3.3.2. EDO linéaires du premier ordre

Elles sont de la forme

$$a(x)y'(x) + b(x)y(x) = g(x)$$

où a, b et g sont des fonctions données, continues sur un intervalle $I \subset \mathbb{R}$. Pour la résolution, on se place sur un intervalle $J \subset I$ tel que la fonction a ne s'annule pas sur J.

Pour $x \in \mathcal{D}_b \cap \mathcal{D}_g \cap \{x \in \mathcal{D}_a | a(x) \neq 0\}$, toute solution $y(x)$ de cette EDO peut être écrite soit comme somme de deux fonctions (y_H et y_P) soit comme produit de deux fonctions (u et v) :

$$y(x) = \underbrace{Ce^{-A(x)}}_{y_H(x)} + \underbrace{B(x)e^{-A(x)}}_{y_P(x)}$$

avec

- $A(x)$ une primitive de $\dfrac{b(x)}{a(x)}$,
- $B(x)$ une primitive de $\dfrac{g(x)}{a(x)}e^{A(x)}$.

Vérification

Pour vérifier que c'est bien une solution, il suffit de dériver :

$$\begin{aligned} y'(x) &= CA'(x)e^{-A(x)} - B(x)A'(x)e^{-A(x)} + B'(x)e^{-A(x)} \\ &= -A'(x)\left(Ce^{-A(x)} + B(x)e^{-A(x)}\right) + B'(x)e^{-A(x)} \\ &= -A'(x)y(x) + B(x)e^{-A(x)} \\ &= -\frac{b(x)}{a(x)}y(x) + \frac{g(x)}{a(x)}e^{A(x)}e^{-A(x)} = -\frac{b(x)}{a(x)}y(x) + \frac{g(x)}{a(x)} \end{aligned}$$

donc $a(x)y'(x) = -b(x)y(x) + g(x)$.

Exemple

Considérons l'EDO

$$y'(x) - y(x) = x.$$

On a

$$\begin{aligned} a(x) &= 1, \\ b(x) &= -1, \\ g(x) &= x. \end{aligned}$$

Pour $x \in \mathbb{R}$, on a

- $A(x) = \int -1 \mathrm{d}x = -x$,
- $B(x) = \int xe^{-x}\mathrm{d}x = -(1+x)e^{-x}$,

d'où

$$y(x) = \left(C - (1+x)e^{-x}\right)e^{x} = Ce^{x} - (1+x).$$

Remarque

On peut montrer que :

- y_H **est la solution générale de l'EDO homogène associée**, c'est-à-dire de l'EDO $a(x)y'(x) + b(x)y(x) = 0$ (qui est à variables séparables).
  En effet, la fonction $y(x) = 0$ pour tout x étant solution, toute autre solution $x \mapsto y(x)$ sera donc non nulle. On peut alors diviser l'EDO homogène associée par y et procéder formellement comme suit :

$$\begin{aligned} \frac{y'(x)}{y(x)} = -\frac{b(x)}{a(x)} \quad &\Longrightarrow \quad \int \frac{1}{y}\mathrm{d}y = -\int -\frac{b(x)}{a(x)}\mathrm{d}x \\ &\Longrightarrow \quad \ln|y| = -\int \frac{b(x)}{a(x)}\mathrm{d}x. \end{aligned}$$

  Ainsi, toute solution non nulle de l'équation homogène associée est de la forme

$$y_H(x) = Ce^{-A(x)} \qquad \text{où } A(x) = \int \frac{b(u)}{a(u)}\mathrm{d}u$$

  avec C constante arbitraire.
- y_P **est une solution particulière.**
  Cette solution particulière peut être une solution *évidente*, par exemple une solution constante.
  Dans la quête d'une solution évidente (non constante), le *principe de superposition* peut être utile : soient a et b deux réels et $g_1, g_2, \dots, g_n$ des applications continues sur un intervalle I de $\mathbb{R}$.
  Si y_k est une solution particulière de l'EDO $ay'(x) + by(x) = g_k(x)$ alors $\sum_{k=1}^{n} y_k$ est une solution particulière de l'EDO $ay'(x) + by(x) = \sum_{k=1}^{n} g_k(x)$.

Si on ne trouve pas de solution particulière, on peut en chercher une par la **méthode de Lagrange** ou **de variation de la constante**.
Si $y_1(x)$ est une solution non nulle de l'EDO homogène, on introduit une fonction auxiliaire inconnue $B(x)$ telle que $y(x) = B(x)y_1(x)$ soit solution de notre EDO.
On calcule alors $y'(x)$ et on reporte $y'(x)$ et $y(x)$ dans notre EDO. On observe que $K(x)$ disparaît, ce qui fournit une auto-vérification.
Il ne reste que $B'(x)$, ce qui permet de calculer $B(x)$ et donc $y_P(x)$.

3.3.3. EDO de Bernoulli

Les équations différentielles de Bernoulli sont du premier ordre et de la forme

$$u(x)y'(x) + v(x)y(x) = w(x)(y(x))^\alpha, \quad \alpha \in \mathbb{R} \setminus \{0, 1\}$$

où u, v et w sont des fonctions données, continues sur un intervalle $I \subset \mathbb{R}$. Pour la résolution, on se place sur un intervalle $J \subset I$ tel que la fonction u ne s'annule pas sur J et on définit une nouvelle fonction $x \mapsto z(x) = (y(x))^{1-\alpha}$. L'EDO initiale est alors équivalente à l'EDO linéaire du premier ordre suivante :

Formellement $z = y^{1-\alpha}$ implique d'une part $y = zy^\alpha$ et d'autre part $z' = (1-\alpha)y^{-\alpha}y'$, et donc $y' = (1-\alpha)z'y^\alpha$.

$$\underbrace{u(x)}_{a(x)} z'(x) + \underbrace{(1-\alpha)v(x)}_{b(x)} z(x) = \underbrace{(1-\alpha)w(x)}_{g(x)}.$$

Par conséquent, pour $x \in \mathcal{D}_v \cap \mathcal{D}_w \cap \{x \in \mathcal{D}_u \mid u(x) \neq 0\}$, toute solution y s'écrit comme

$$y(x) = [z(x)]^{1/(1-\alpha)}$$

avec :
- $z(x) = \underbrace{Ce^{-A(x)}}_{y_H(x)} + \underbrace{B(x)e^{-A(x)}}_{y_P(x)}$,
- $A(x)$ une primitive de $(1-\alpha)\frac{v(x)}{u(x)}$,
- $B(x)$ une primitive de $(1-\alpha)\frac{w(x)}{u(x)}e^{A(x)}$.

⚠ Exemple

On se propose de résoudre l'équation différentielle

$$y'(x) + \frac{1}{2}y(x) = \frac{1}{2}(x-1)y^3(x).$$

Il s'agit d'une équation différentielle de Bernoulli. Comme $u(x) = 1$ pour tout $x \in \mathbb{R}$, on cherche sa solution générale sur $\mathbb{R}$.

- $A(x) = (1-\alpha)\int \frac{v(x)}{u(x)}\,\mathrm{d}x = -2\int \frac{1/2}{1}\,\mathrm{d}x = -x$,
- $B(x) = (1-\alpha)\int \frac{w(x)}{u(x)}e^{A(x)}\mathrm{d}x = -2\int \frac{(x-1)/2}{1}e^{-x}\mathrm{d}x = \int(1-x)e^{-x}\mathrm{d}x = -(1-x)e^{-x} - \int e^{-x}\mathrm{d}x = xe^{-x}$,
- $z(x) = (C + B(x))\,e^{-A(x)} = (C + xe^{-x})e^x = Ce^x + x$.

On conclut que la solution générale de l'EDO de Bernoulli assignée est

$$y(x) = \frac{1}{\sqrt{x + Ce^x}}.$$

Notons que y n'est définie que si $x + Ce^x > 0$.

Chapitre 4.

Introduction à l'approximation numérique d'EDO

4.1. Position du problème

Problème de Cauchy
Trouver une fonction $y\colon I \subset \mathbb{R} \to \mathbb{R}$ définie sur un intervalle I telle que

$$\begin{cases} y'(t) = \varphi(t, y(t)), \quad \forall t \in I =]t_0, T[, \\ y(t_0) = y_0, \end{cases}$$

avec y_0 une valeur donnée et supposons que l'existence et l'unicité d'une solution y sur I aient été établies.

Nous disposons donc de :

- une valeur initiale y_0 pour $y(t)$ à un certain instant t_0,
- l'EDO elle-même $y'(t) = \varphi(t, y(t))$ qui relie la variation de y par rapport à t à la fonction φ.

Exemple : rechercher la fonction $t \mapsto y(t)$ pour $t \in [0; 2]$ telle que $y'(t) = -2y(t)$ et $y_0 = y(t_0) = 17$. La solution existe, est unique, et est donnée par $y(t) = 17\exp(-2t)$.

Lorsqu'on ne connait pas de solution exacte à un problème de Cauchy qui admet une et une seule solution, on essaye d'en avoir une bonne approximation par des méthodes numériques.

4.2. Solution exacte discrète VS solution approchée

L'idée la plus simple pour résoudre de manière approchée un problème de Cauchy est de discrétiser l'intervalle de temps avec un pas h et d'approcher la dérivée temporelle sur chaque sous-intervalle de longueur h.

Pour $h > 0$ soit $t_n \equiv t_0 + nh$ avec $n = 0, 1, 2, \dots, N$ une suite de $N + 1$ nœuds de I induisant une discrétisation de I en N sous-intervalles $I_n = [t_n; t_{n+1}]$ chacun de longueur $h = \frac{T - t_0}{N} > 0$ (appelé le *pas de discrétisation*).

Pour chaque nœud t_n, on cherche la valeur inconnue u_n qui approche la valeur exacte $y_n \equiv y(t_n)$.

- L'ensemble de $N + 1$ valeurs $\{t_0, t_1 = t_0 + h, \dots, t_N = T\}$ représente les points de la *discrétisation*.
- L'ensemble de $N + 1$ valeurs $\{y_0, y_1, \dots, y_N\}$ représente la *solution exacte discrète*.
- L'ensemble de $N + 1$ valeurs $\{u_0 = y_0, u_1, \dots, u_N\}$ représente la *solution numérique*.

```python
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams.update({'font.size': 14})

def euler_explicit(f, y0, t_values):
    h = t_values[1] - t_values[0]
    y_values = np.empty_like(t_values)
    y_values[0] = y0
    for i in range(len(t_values) - 1):
        y_values[i+1] = y_values[i] + h * f(t_values[i], y_values[i])
```

```
    return y_values

def plot_solutions(n, show_exact_line=True, show_exact_points=True, show_approx_points=True):
    t0 = 0
    tn = 2
    y0 = 17
    phi = lambda t, y : -2 * y
    exact_solution = lambda t : 17 * np.exp(-2 * t)

    # Calcul des solutions
    t_exact = np.linspace(t0, tn, 100)  # pour la solution exacte continue
    y_exact = exact_solution(t_exact)
    t_exact_discrete = np.linspace(t0, tn, n+1)  # pour les solutions exacte et approchée discrètes
    y_exact_discrete = exact_solution(t_exact_discrete)
    y_approx = euler_explicit(phi, y0, t_exact_discrete)

    plt.figure(figsize=(14, 6))
    plt.xlabel('t')
    plt.ylabel('y')
    plt.title('Comparaison des solutions')

    plt.scatter(t_exact_discrete, 0*t_exact_discrete, label='Discrétisation', color='black', marker='x')
    for i in range(len(t_exact_discrete)):
        plt.annotate('$t_{'+str(i)+'}$', (t_exact_discrete[i]-0.04, - 0.75), color='black')

    if show_exact_line:
        plt.plot(t_exact, y_exact, label='Solution exacte')

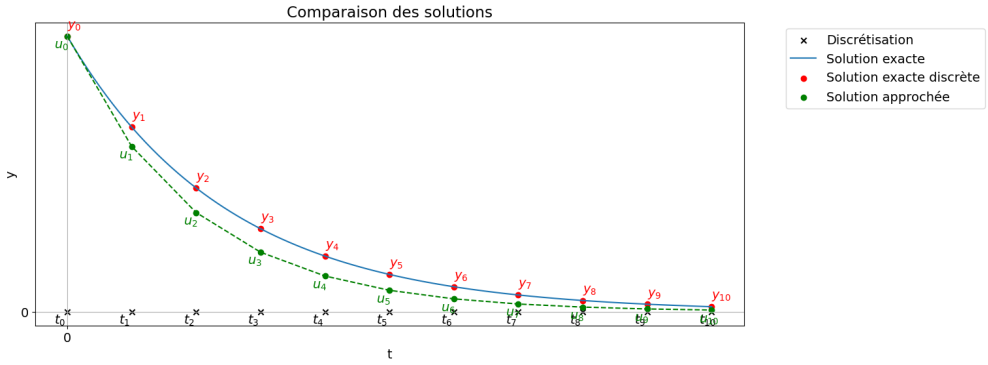
    if show_exact_points:
        plt.scatter(t_exact_discrete, y_exact_discrete, color='red', label='Solution exacte discrète')
        for i in range(len(y_exact_discrete)):
            plt.annotate('$y_{'+str(i)+'}$', (t_exact_discrete[i], y_exact_discrete[i] + 0.5),
↪   color='red')

    if show_approx_points:
        plt.plot(t_exact_discrete, y_approx, 'g--')
        plt.scatter(t_exact_discrete, y_approx, color='green', label='Solution approchée')
        for i in range(len(y_approx)):
            plt.annotate('$u_{'+str(i)+'}$', (t_exact_discrete[i]-0.04, y_approx[i] - 0.75),
↪   color='green')

    plt.xticks([0])
    plt.yticks([0])
    plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
    plt.grid(True)
    plt.show()


# si interactif
# from ipywidgets import interact, widgets
# interact(plot_solutions, n=widgets.IntSlider(min=5, max=20, step=1, value=10),
#          show_exact_line=True, show_exact_points=False, show_approx_points=False);

# si non interactif
plot_solutions(10, show_exact_line=True, show_exact_points=True, show_approx_points=True);
```



4.3. Construction élémentaire des méthodes d'Euler explicite et implicite

Une méthode classique, la **méthode d'Euler explicite** (ou *progressive*, de l'anglais *forward*), est obtenue en considérant l'équation différentielle en chaque nœud t_n et en remplaçant la dérivée exacte $y'(t_n)$ par le taux d'accroissement "à droite" comme suit :

$$\begin{cases} y'(t_n) = \varphi(t_n, y(t_n)) \\ y'(t_n) = \lim\limits_{k\to 0} \dfrac{y(t_n+k)-y(t_n)}{k} = \dfrac{y(t_n+k)-y(t_n)}{k} + \mathcal{O}(k) \end{cases}$$

Pour $k = h > 0$, on obtient

$$\varphi(t_n, y(t_n)) = \frac{y(t_{n+1})-y(t_n)}{h} + \mathcal{O}(h) \quad \rightsquigarrow \quad \varphi(t_n, y(t_n)) \approx \frac{y(t_{n+1})-y(t_n)}{h}.$$

Cela permet de construire une solution numérique par une suite récurrente :

Schéma EE :

$$\begin{cases} u_0 = y(t_0) = y_0, \\ u_{n+1} = u_n + h\varphi(t_n, u_n), \quad n = 0, 1, 2, \dots N-1. \end{cases}$$

On peut donner une intérprétation géométrique de la méthode d'Euler explicite : en un point donné t_n, on avance d'un pas h le long de la droite tangente à la solution (inconnue).

Une autre méthode classique est la **méthode d'Euler implicite** (ou *rétrograde*, de l'anglais *backward*), qui s'obtient en considérant l'équation différentielle en chaque nœud t_{n+1} et en remplaçant la dérivée exacte $y'(t_{n+1})$ par le taux d'accroissement "à gauche" :

$$\begin{cases} y'(t_{n+1}) = \varphi(t_{n+1}, y(t_{n+1})) \\ y'(t_{n+1}) = \lim\limits_{k\to 0} \dfrac{y(t_{n+1}+k)-y(t_{n+1})}{k} = \dfrac{y(t_{n+1}+k)-y(t_{n+1})}{k} + \mathcal{O}(k) \end{cases}$$

Pour $k = -h < 0$, on obtient

$$\varphi(t_{n+1}, y(t_{n+1})) = \frac{y(t_n)-y(t_{n+1})}{-h} + \mathcal{O}(h) \quad \rightsquigarrow \quad \varphi(t_{n+1}, y(t_{n+1})) \approx \frac{y(t_{n+1})-y(t_n)}{h}.$$

ce qui donne encore une suite récurrente pour construire une solution numérique :

Schéma EI :

$$\begin{cases} u_0 = y(t_0) = y_0, \\ u_{n+1} - h\varphi(t_{n+1}, u_{n+1}) = u_n, \quad n = 0, 1, 2, \dots N-1. \end{cases}$$

Ces deux méthodes sont dites *à un pas* : pour calculer la solution numérique u_{n+1} au nœud t_{n+1}, on a seulement besoin des informations disponibles au nœud précédent t_n. Plus précisément,

- pour la méthode d'Euler progressive, u_{n+1} ne dépend que de la valeur u_n calculée précédemment,
- pour la méthode d'Euler rétrograde, u_{n+1} dépend aussi "de lui-même" à travers la valeur de $\varphi(t_{n+1}, u_{n+1})$.

C'est pour cette raison que la méthode d'Euler progressive est dite *explicite* tandis que la méthode d'Euler rétrograde est dite *implicite*.
Les méthodes implicites sont plus coûteuses que les méthodes explicites car, si la fonction φ est non linéaire, un problème non linéaire doit être résolu à chaque temps t_{n+1} pour calculer u_{n+1}. Pour cela on peut utiliser des méthodes itératives comme la méthode de la dichotomie ou une méthode de point fixe comme la méthode de Newton. Néanmoins, nous verrons que les méthodes implicites jouissent de meilleures propriétés de stabilité que les méthodes explicites.

4.4. Implémentation des schémas d'Euler explicite et implicite

Voyons un exemple complet.

Exemple : considérons le problème de Cauchy

$$\begin{cases} y'(t) = 2ty(t), \quad \forall t \in I = [0,1], \\ y(0) = 1. \end{cases}$$

Sachant que la solution est $y(t) = e^{t^2}$, estimer la qualité du schéma en affichant la plus grande erreur commise (en valeur absolue) et varier le pas de discrétisation h pour voir son impact sur l'erreur.

On commence par importer

1. la fonction `fsolve` du module `scipy.optimize` pour résoudre les équations implicites présentes dans le schéma implicite ;
2. le module `matplotlib` et le module `numpy` :
   - soit en important séparément `matplotlib.pyplot` et `numpy` (avec les alias classiques `plt` et `np` respectivement) ;
   - soit (déconseillé) en important `matplotlib.pylab`, ce qui importe à la fois `matplotlib` et `numpy` (avec par exemple l'alias commun `pyl` ou sans alias si on utilise `*`).

Rappels : `numpy` redéfinit toutes les fonctions mathématiques du module `math` et ces fonctions sont vectorisées (*e.g.* on pourra écrire directement `yy = np.sin(xx)` avec `xx` une liste/tuple/array au lieu d'écrire `yy = [np.sin(x) for x in xx]`).

```
from scipy.optimize import fsolve # pour les schémas implicites

#from matplotlib.pylab import *

import numpy as np
import matplotlib.pyplot as plt
# plt.rcdefaults()
plt.rcParams.update({'font.size': 14})
```

On initialise le problème de Cauchy

```
t0     = 0
tfinal = 1
y0     = 1
```

On définit l'équation différentielle : `phi` est une fonction python qui contient la fonction mathématique $\varphi : [t_0, T] \times \mathbb{R} \to \mathbb{R}$ définie par $\varphi(t, y) = 2ty$. Bien noter que φ dépend de **deux** variables : t et y. Dans l'EDO on a $\varphi(t, y(t))$: il s'agit d'une fonction d'**une** seule variable obtenue par composition de la fonction φ avec la fonction $t \mapsto y(t)$.

```
phi = lambda t, y : 2*y*t
```

On introduit la discrétisation : les nœuds d'intégration $[t_0, t_1, \dots, t_N]$ sont contenus dans le vecteur `tt`.
On a $N+1$ points espacés de $h = \frac{t_N - t_0}{N}$.

```
N  = 8
tt = np.linspace(t0,tfinal,N+1)
```

On écrit les schémas numériques : les valeurs $[u_0, u_1, \dots, u_N]$ pour chaque méthode sont contenues dans le vecteur `uu`.

Schéma d'Euler progressif :

$$\begin{cases} u_0 = y_0, \\ u_{n+1} = u_n + h\varphi(t_n, u_n) \quad n = 0, 1, 2, \dots N-1 \end{cases}$$

```
def euler_progressif(phi,tt,y0):
    h  = tt[1]-tt[0]
    N  = len(tt)-1 # len(tt) = N+1 car tt contient N+1 points
    uu = np.empty_like(tt) # array vide de même taille que tt
    uu[0] = y0
    for n in range(N): # range(N) = 0,1,...,N-1
        uu[n+1] = uu[n]+h*phi(tt[n],uu[n])
    return uu
```

Schéma d'Euler régressif :

$$\begin{cases} u_0 = y_0, \\ u_{n+1} = u_n + h\varphi(t_{n+1}, u_{n+1}) \quad n = 0, 1, 2, \dots N-1 \end{cases}$$

Attention :

- u_{n+1} est solution de l'équation $x = u_n + h\varphi(t_{n+1}, x)$, c'est-à-dire un zéro de la fonction (en général non linéaire)

$$x \mapsto -x + u_n + h\varphi(t_{n+1}, x)$$

- la fonction `fsolve` du module `scipy.optimize` requiert deux paramètres : une fonction et un point de départ. Elle renvoie une liste avec la valeur du zéro approché (c'est une liste, car `fsolve` résout un système, ici avec une seule équation). Un choix naturel pour le point de départ est la valeur précédente u_n ou la valeur prédite par la méthode explicite $u_n + h\varphi(t_n, u_n)$.

```
def euler_regressif(phi,tt,y0):
    h  = tt[1]-tt[0]
    N = len(tt)-1 # len(tt) = N+1 car tt contient N+1 points
    uu = np.empty_like(tt) # array vide de même taille que tt
    uu[0] = y0
    for n in range(N): # range(N) = 0,1,...,N-1
        fct_implicit = lambda x: -x + uu[n] + h*phi(tt[n+1], x)
        temp = fsolve( fct_implicit , uu[n] )
        uu[n+1] = temp[0] # fsolve renvoie un array qui ne contient qu'un seul élément
    return uu
```

On calcule les solutions approchées :

```
uu_ep = euler_progressif(phi,tt,y0)
uu_er = euler_regressif(phi,tt,y0)
```

Comme on la connait, on définit la solution exacte pour calculer les erreurs. Pour cela il faut décider si on veut une erreur absolue ou relative et quelle norme utiliser. Ici on choisit l'erreur absolue maximale sur tous les nœuds.

```
sol_exacte = lambda t : y0*np.exp(t**2)
yy = sol_exacte(tt) # = [sol_exacte(t) for t in tt]

# Calcul de l'erreur
# ==========================

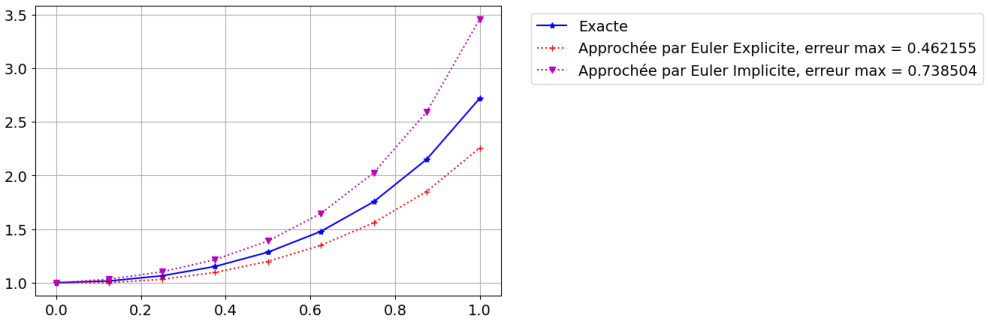
# On peut utiliser la norme infinie : on cherche la plus grande différence entre les deux vecteurs yy et
 ↪  uu en valeur absolue
err_ep = np.linalg.norm(yy-uu_ep,np.inf) # max(np.abs(yy-uu_ep))
err_er = np.linalg.norm(yy-uu_er,np.inf) # max(np.abs(yy-uu_er))

# On peut aussi utiliser la norme 2 : on somme les carrés des différences, on prend la racine carrée et
 ↪  on divise par la norme de yy
# err_ep = np.linalg.norm(yy-uu_ep)/np.linalg.norm(yy)
# err_er = np.linalg.norm(yy-uu_er)/np.linalg.norm(yy)
```

On compare les graphes des solutions exacte (en bleu) et approchées (en rouge) et on affiche le maximum de l'erreur :

```
plt.figure(figsize=(8, 5))

plt.plot(tt,yy   ,'b-*',label='Exacte')
plt.plot(tt,uu_ep,'r:+',label=f'Approchée par Euler Explicite, erreur max = {err_ep:g}')
plt.plot(tt,uu_er,'m:v',label=f'Approchée par Euler Implicite, erreur max = {err_er:g}')
plt.grid()
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left');
```



4.5. Convergence empirique des schémas d'Euler

Définition : schéma convergent et ordre de convergence
Une méthode numérique est convergente si

$$|y_n - u_n| \xrightarrow[h\to 0]{} 0 \qquad \forall n = 0, \dots, N$$

Si $|y_n - u_n| = \mathcal{O}(h^\omega)$ pour $\omega > 0$, on dit que la convergence de la méthode est d'ordre ω.

Remarque : pour un intervalle fixé, $N \to +\infty$ lorsque $h \to 0$.

Exemple : considérons à nouveau le problème de Cauchy

$$\begin{cases} y'(t) = 2ty(t), \quad \forall t \in I = [0,1], \\ y(0) = 1. \end{cases}$$

On se propose d'estimer l'ordre de convergence des méthodes d'**Euler**.

Pour chaque schéma, on calcule la solution approchée avec différentes valeurs de h (ce qui correspond à différentes valeurs de N). On sauvegarde les valeurs de h dans le vecteur `H`.

Pour chaque valeur de h, on calcule le maximum de la valeur absolue de l'erreur et on sauvegarde toutes ces erreurs dans le vecteur `err_schema` de sort que `err_schema[k]` contient $e_k = \max_{i=0,\dots,N_k} |y(t_i) - u_i|$ avec N_k.

```
from numpy.linalg import norm

Ntest = 6

H_ep_er = np.empty(Ntest)
err_ep = np.empty(Ntest)
err_er = np.empty(Ntest)

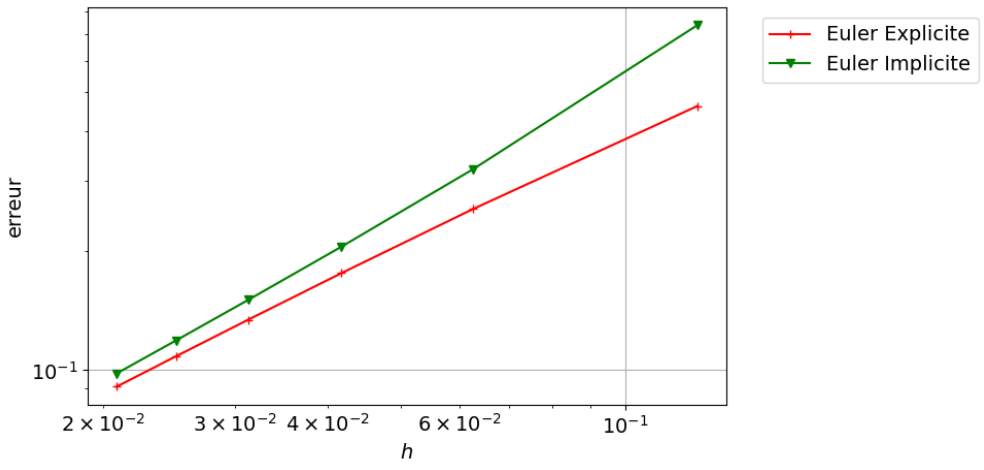

for k in range(1,Ntest+1):

    tt_tmp = np.linspace(t0,tfinal,k*N+1) # on raffine la grille
    yy = sol_exacte(tt_tmp)
    uu_ep_tmp = euler_progressif(phi,tt_tmp,y0)
    uu_er_tmp = euler_regressif(phi,tt_tmp,y0)

    H_ep_er[k-1]= tt_tmp[1]-tt_tmp[0]
    err_ep[k-1] = norm(uu_ep_tmp-yy,np.inf) # idem que max(abs(uu_ep-yy))
    err_er[k-1] = norm(uu_er_tmp-yy,np.inf) # idem que max(abs(uu_er-yy))
```

Pour afficher l'ordre de convergence on affiche les points (`h[k]`,`err_ep[k]`) en échelle logarithmique : on représente $\ln(h)$ sur l'axe des abscisses et $\ln(\text{err})$ sur l'axe des ordonnées. En effet, si $\text{err} = Ch^p$ alors $\ln(\text{err}) = \ln(C) + p\ln(h)$, autrement dit, en échelle logarithmique, p représente la pente de la ligne droite $\ln(\text{err})$.

```
plt.figure(figsize=(8, 5))
plt.loglog(H_ep_er,err_ep, 'r-+', label='Euler Explicite')
plt.loglog(H_ep_er,err_er, 'g-v', label='Euler Implicite')
plt.xlabel('$h$')
plt.ylabel('erreur')
plt.legend(bbox_to_anchor=(1.04,1),loc='upper left')
plt.grid(True);
```



Pour estimer l'ordre de convergence on doit estimer la pente de la droite qui relie l'erreur au pas k à l'erreur au pas $k+1$ en échelle logarithmique. Pour estimer la pente globale de cette droite (par des moindres carrés) on peut utiliser la fonction `polyfit` (du module `numpy`).

```
# ln(e) = a ln(h) + b
a_ep, b_ep = np.polyfit(np.log(H_ep_er),np.log(err_ep), 1) # polyfit ( [liste des abscisses], [liste des
↪  ordonnées], degré du polynome)
print (f'Euler progressif: ordre {a_ep :1.2f}')
a_er, b_er = np.polyfit(np.log(H_ep_er),np.log(err_er), 1)
print (f'Euler regressif: ordre {a_er:1.2f}')
```

```
Euler progressif: ordre 0.91
Euler regressif: ordre 1.12
```

On peut bien-sûr afficher la droite obtenue par régression linéaire en même temps que les points.

Soit on affiche les logarithmes des données et l'équation de la droite avec la commande `plot`, soit on affiche les données en échelle logarithmique et l'équation de l'erreur Ch^ω avec $C = \ln(b)$ et $\omega = a$ avec l'instruction `loglog` :

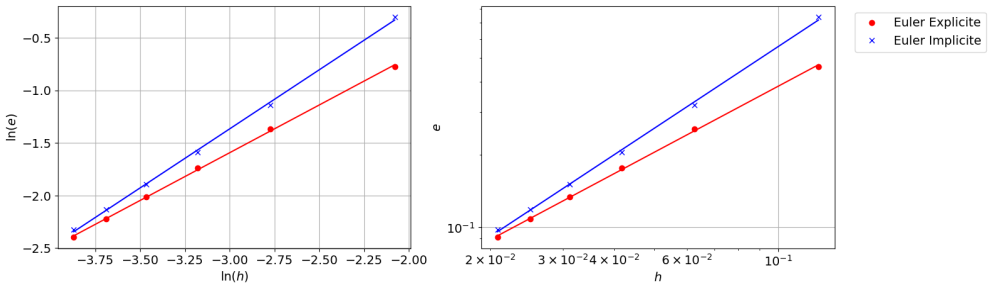
```
plt.figure(figsize=(16, 5))

plt.subplot(1,2,1)
plt.plot( np.log(H_ep_er), np.log(err_ep), 'ro',label='Euler Explicite')
plt.plot( np.log(H_ep_er), a_ep*np.log(H_ep_er)+b_ep, 'r' ) # la droite de régression
plt.plot( np.log(H_ep_er), np.log(err_er), 'bx',label='Euler Implicite')
plt.plot( np.log(H_ep_er), a_er*np.log(H_ep_er)+b_er, 'b' ) # la droite de régression
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
plt.grid(True)

plt.subplot(1,2,2)
plt.loglog(H_ep_er,err_ep, 'ro',label='Euler Explicite')
plt.loglog( H_ep_er, np.exp(b_ep)*(H_ep_er**a_ep), 'r' ) # la droite de régression

plt.loglog(H_ep_er,err_er, 'bx',label='Euler Implicite')
plt.loglog( H_ep_er, np.exp(b_er)*(H_ep_er**a_er), 'b' ) # la droite de régression

plt.xlabel('$h$')
plt.ylabel('$e$')
plt.legend(bbox_to_anchor=(1.04,1),loc='upper left')
plt.grid(True);
```



Remarque

Choix de la norme pour mesurer l'erreur.
Le choix de la norme doit être cohérent avec la norme continue que l'on cherche à approcher.
Pour la norme infinie continue $\|e\|_{L^\infty} = \sup_{t\in[0,1]} |e(t)|$, la norme discrète naturelle est $\max_i |e_i|$, qui ne dépend pas du nombre de points du maillage.

En revanche, pour approcher la norme L^2 continue $\|e\|_{L^2} = \left(\int_0^1 e(t)^2\,dt\right)^{1/2}$, la norme euclidienne discrète $\|e\|_2 = \sqrt{\sum_i e_i^2}$ doit être multipliée par $\sqrt{h}$ (où h est le pas de temps), car l'intégrale est approchée par une somme de Riemann : $\int_0^1 e(t)^2 dt \approx h\sum_i e_i^2$. On obtient donc $\|e\|_{L^2} \approx \sqrt{h}\,\|e\|_2$. Sans ce facteur $\sqrt{h}$, la norme dépend artificiellement du nombre de points et l'ordre de convergence mesuré est incorrect.

4.6. Annexe : implémentation d'un schéma "moyen" des deux schémas précédents

On obtient la **méthode de Crank-Nicolson** (ou *du trapèze*)

Schéma CN :

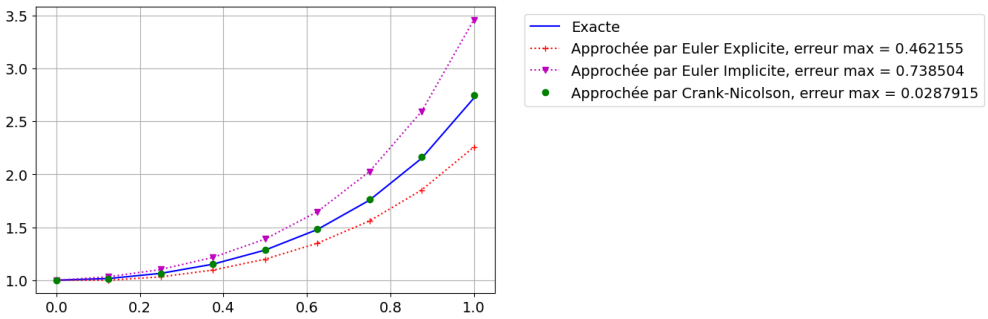
$$\begin{cases} u_0 = y(t_0) = y_0, \\ u_{n+1} = u_n + \dfrac{h}{2}(\varphi(t_n, u_n) + \varphi(t_{n+1}, u_{n+1})), \quad n = 0, 1, 2, \dots N-1. \end{cases}$$

```
def CN(phi,tt,y0):
    h  = tt[1]-tt[0]
    N = len(tt)-1 # len(tt) = N+1 car tt contient N+1 points
    uu = np.empty_like(tt) # array de même taille que tt
    uu[0] = y0
    for i in range(N): # range(N) = 0,1,...,N-1
        fc_implicit = lambda x: -x + uu[i] + h/2*( phi(tt[i],uu[i]) + phi(tt[i+1], x) )
        temp = fsolve( fc_implicit , uu[i] )
        uu[i+1] = temp[0]
        # Rmk: pour ce phi linéaire en y, on peut calculer uu[i+1] directement
        # uu[i+1] = (1+h*tt[i])/(1-h*tt[i+1])*uu[i]
    return uu


N  = 8
tt = np.linspace(t0,tfinal,N+1)
uu_cn = CN(phi,tt,y0)
yy = sol_exacte(tt)

plt.figure(figsize=(8, 5))

plt.plot(tt,yy   ,'b-',label='Exacte')
plt.plot(tt,uu_ep,'r:+',label=f'Approchée par Euler Explicite, erreur max = {max(np.abs(uu_ep-yy)):g}')
plt.plot(tt,uu_er,'m:v',label=f'Approchée par Euler Implicite, erreur max = {max(np.abs(uu_er-yy)):g}')
plt.plot(tt,uu_cn,'go',label=f'Approchée par Crank-Nicolson, erreur max = {max(np.abs(uu_cn-yy)):g}')
plt.grid()
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left');
```



```
H_cn = np.empty(Ntest)
err_cn = np.empty(Ntest)


for k in range(1,Ntest+1):

    tt = np.linspace(t0,tfinal,k*N+1) # on raffine la grille
    yy = sol_exacte(tt) # on calcule la solution exacte sur cette grille
    uu_cn = CN(phi,tt,y0) # on calcule la solution approchée par Crank-Nicolson

    H_cn[k-1] = tt[1]-tt[0] # on stocke le pas de temps
    err_cn[k-1] = np.linalg.norm(uu_cn-yy,np.inf) # on stocke l'erreur (norme infinie)

# ln(e) = a ln(h) + b
a_cn, b_cn = np.polyfit(np.log(H_cn),np.log(err_cn), 1)
print (f'Crank Nicolson: ordre {a_cn :1.2f}')


plt.figure(figsize=(8, 5))

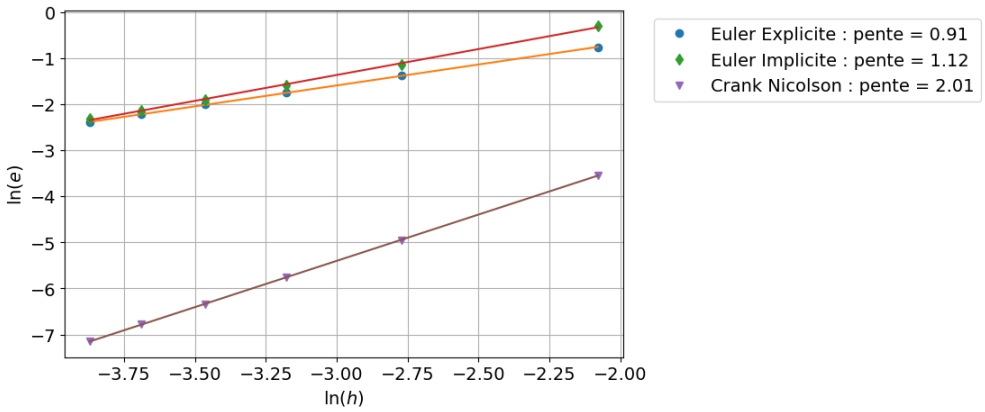
plt.plot(np.log(H_ep_er), np.log(err_ep), 'o',label=f'Euler Explicite : pente = {a_ep:1.2f}')
plt.plot( np.log(H_ep_er), a_ep*np.log(H_ep_er)+b_ep ) # la droite de régression

plt.plot(np.log(H_ep_er),np.log(err_er), 'd',label=f'Euler Implicite : pente = {a_er:1.2f}')
plt.plot(np.log(H_ep_er), a_er*np.log(H_ep_er)+b_er )

plt.plot(np.log(H_cn),np.log(err_cn), 'v',label=f'Crank Nicolson : pente = {a_cn:1.2f}')
plt.plot(np.log(H_cn), a_cn*np.log(H_cn)+b_cn )

plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
plt.legend(loc='upper left',bbox_to_anchor=(1.04,1))
plt.grid(True);
```

```
Crank Nicolson: ordre 2.01
```



Chapitre 5.

Schémas “classiques” à un pas

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import fsolve
import sympy as sp
from IPython.display import display, Markdown
# plt.rcdefaults()
plt.rcParams.update({'font.size': 14})
```

Considérons le problème de Cauchy

$$\begin{cases} y'(t) = \varphi(t, y(t)), \quad \forall t \in I = [t_0, T], \\ y(t_0) = y_0, \end{cases}$$

avec y_0 une valeur donnée et supposons que l'existence et l'unicité d'une solution y sur I aient été établies.

Soit $N \geq 1$ un entier et posons le *pas de discrétisation* $h = \frac{T-t_0}{N} > 0$. On définit alors les $N+1$ nœuds $t_n = t_0 + nh$, pour $n = 0, 1, \dots, N$ qui divisent l'intervalle I en N sous-intervalles $I_n = [t_n, t_{n+1}]$ de même longueur h.

Pour chaque nœud t_n, on cherche une valeur u_n qui approchera la valeur exacte $y_n \equiv y(t_n)$.

- L'ensemble de $N+1$ valeurs $\{t_0, t_1 = t_0 + h, \dots, t_N = T\}$ constitue les points de la *discrétisation*.
- L'ensemble de $N+1$ valeurs $\{y_0, y_1, \dots, y_N\}$ constitue la *solution exacte discrète*.
- L'ensemble de $N+1$ valeurs $\{u_0 = y_0, u_1, \dots, u_N\}$ constitue la *solution numérique* obtenue **via une suite récurrente**.

Les schémas que l'on va construire permettent de calculer (explicitement ou implicitement) u_{n+1} à partir de $u_n, u_{n-1}, ..., u_{n-\kappa}$. Il est ainsi possible de calculer successivement $u_{\kappa+1}, u_{\kappa+2}, \dots, u_N$ à partir des valeurs initiales $u_0, \dots, u_\kappa$ via une formule de récurrence de la forme

$$\begin{cases} u_0 = y_0, \\ u_1 \approx y_1, \\ \vdots \\ u_\kappa \approx y_\kappa, \\ u_{n+1} = \Phi(u_{n+1}, u_n, u_{n-1}, \dots, u_{n-\kappa}), \quad \text{pour } n = \kappa, \kappa+1, \dots, N-1. \end{cases}$$

Méthodes explicites *vs* implicites

- Une méthode est dite *explicite* si la valeur u_{n+1} peut être calculée directement à partir des valeurs précédentes u_k, $k \leq n$ (ou d'une partie d'entre elles).
- Une méthode est dite *implicite* s'il faut résoudre une équation impliquant u_{n+1} et la fonction φ pour obtenir la valeur u_{n+1}.

Méthodes à un pas *vs* multi-pas

- Une méthode est dite *à un pas* si u_{n+1} ne dépend que de u_n (et éventuellement de u_{n+1} lui-même).
- Sinon, le schéma est une méthode *multi-pas*, ou à *pas multiples*, si u_{n+1} dépend de plusieurs valeurs précédentes.

5.1. Construction de schémas à un pas

Si nous intégrons l'EDO $y'(t) = \varphi(t, y(t))$ entre t_n et t_{n+1} nous obtenons

$$\underbrace{\int_{t_n}^{t_{n+1}} y'(t)\mathrm{d}t}_{y_{n+1}-y_n} = \int_{t_n}^{t_{n+1}} \varphi(t, y(t))\mathrm{d}t.$$

En utilisant une formule d'approximation pour approcher le membre de droite, on peut construire un schéma par une formule de récurrence du type

$$\begin{cases} u_0 = y_0, \\ u_{n+1} = u_n + \text{approximation de } \left(\displaystyle\int_{t_n}^{t_{n+1}} \varphi(t, y(t))\mathrm{d}t \right). \end{cases}$$

Pour construire cette approximation, plusieurs stratégies sont possibles. La plus simple est d'utiliser une formule de quadrature numérique. Pour cela, on remplace la fonction $\varphi(t, y(t))$ par un polynôme interpolant $\varphi(t, y(t))$ en certains points de l'intervalle $[t_n, t_{n+1}]$ (on verra plus tard qu'on peut même construire le polynôme en utilisant des points en dehors de cet intervalle). On obtient ainsi le schéma général suivant :

$$\begin{cases} u_0 = y_0, \\ u_{n+1} = u_n + \displaystyle\int_{t_n}^{t_{n+1}} \tilde{f}(t)\mathrm{d}t \quad \text{où } \tilde{f}(t) \text{ est un polynôme interpolant } \varphi(t, y(t)) \end{cases}$$

En L1 vous avez déjà vu certaines formules de quadrature, comme celles de la figure Figure 5.1 . En utilisant ces formules, on peut construire différents schémas à un pas.

5.1.1. Schéma d'Euler explicite

Si on remplace une fonction f par une constante égale à la valeur de f dans la borne gauche de l'intervalle $[a; b]$ (polynôme qui interpole f au point $(a, f(a))$ et donc de degré 0), on a

$$\tilde{f}(x) = f(a)$$
$$\int_a^b f(x)\mathrm{d}x \approx \int_a^b \tilde{f}(x)\mathrm{d}x = (b-a)f(a).$$

Cette formule est dite *formule de quadrature du rectangle à gauche.*

En utilisant cette formule pour approcher la fonction $t \mapsto \varphi(t, y(t))$ on a

$$\int_{t_n}^{t_{n+1}} \varphi(t, y(t))\mathrm{d}t \approx h\varphi(t_n, y(t_n))$$

et on reconnait le **schéma d'Euler progressif (= Euler explicite = Euler en avant)**

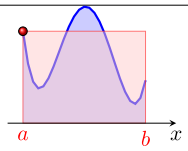
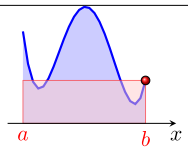
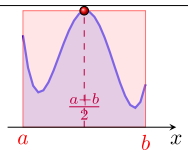
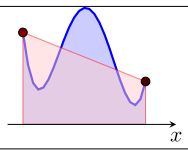
Nom	**Points d'interpolation**	**Polynôme** $p_n \in \mathbb{R}_n[t]$	$\int_a^b p_n(t)\mathrm{d}t$	
Rectangle à gauche	a	$p_0(t) = f(a) \in \mathbb{R}_0[t]$	$(b-a)f(a)$	
Rectangle à droite	b	$p_0(t) = f(b) \in \mathbb{R}_0[t]$	$(b-a)f(b)$	
Point milieu	$\frac{a+b}{2}$	$p_0(t) = f\left(\frac{a+b}{2}\right) \in \mathbb{R}_0[t]$	$(b-a)f\left(\frac{a+b}{2}\right)$	
Trapèze	a, b	$p_1(t) = \frac{f(b)-f(a)}{b-a}(t-a) + f(a) \in \mathbb{R}_1[t]$	$\frac{(b-a)}{2}(f(a)+f(b))$	

FIGURE 5.1.

$$\text{(EE)} \quad \begin{cases} u_0 = y(t_0) = y_0, \\ u_{n+1} = u_n + h\varphi(t_n, u_n) \quad n = 0, 1, 2, \dots N-1 \end{cases}$$

Il s'agit d'un schéma à 1 pas explicite, car il permet d'expliciter u_{n+1} directement en fonction de u_n.

```
def EE(phi,tt,y0):
    h  = tt[1]-tt[0]
    uu = np.empty_like(tt)
    N = len(tt)-1
    uu[0] = y0
    for n in range(N):
        uu[n+1] = uu[n]+h*phi(tt[n],uu[n])
    return uu
```

5.1.2. Schéma d'Euler implicite

Si on remplace une fonction f par une constante égale à la valeur de f en la borne droite de l'intervalle $[a; b]$ (polynôme qui interpole f au point $(b, f(b))$ et donc de degré 0), on a

$$\tilde{f}(x) = f(b)$$
$$\int_a^b f(x)\mathrm{d}x \approx \int_a^b \tilde{f}(x)\mathrm{d}x = (b-a)f(b).$$

Cette formule est dite *formule de quadrature du rectangle à droite.*

En utilisant cette formule pour approcher la fonction $t \mapsto \varphi(t, y(t))$ on a

$$\int_{t_n}^{t_{n+1}} \varphi(t, y(t))\mathrm{d}t \approx (t_{n+1} - t_n)\varphi(t_{n+1}, y(t_{n+1})) = h\varphi(t_{n+1}, y(t_{n+1}))$$



FIGURE 5.2. – Leonhard Euler (1707 - 1783)

et on obtient le **schéma d'Euler rétrograde (=Euler implicite = Euler en arrière)**

$$\text{(EI)} \quad \begin{cases} u_0 = y(t_0) = y_0, \\ u_{n+1} = u_n + h\varphi(t_{n+1}, u_{n+1}) \quad n = 0, 1, 2, \dots N-1 \end{cases}$$

Il s'agit d'un schéma - à 1 pas (car il permet de calculer u_{n+1} en fonction de u_n) - implicite car, pour calculer u_{n+1}, il faudra utiliser un schéma pour le calcul du zéro d'une fonction quelconque. En effet, u_{n+1} est solution de l'équation $x = u_n + h\varphi(t_{n+1}, x)$, c'est-à-dire un zéro de la fonction (en général non linéaire)

$$x \mapsto -x + u_n + h\varphi(t_{n+1}, x).$$

```
def EI(phi,tt,y0):
    h  = tt[1]-tt[0]
    uu = np.empty_like(tt)
    N = len(tt)-1
    uu[0] = y0
    for n in range(N):
        fct_implicite = lambda x: -x+uu[n]+h*phi(tt[n+1],x)
        tmp = fsolve(fct_implicite, uu[n])
        uu[n+1] = tmp[0] # on récupère la valeur scalaire
    return uu
```

5.1.3. Schémas d'Euler modifié (EM) ou de type Runge-Kutta (RK1_M)

Si on remplace une fonction f par une constante égale à la valeur de f au milieu de l'intervalle $[a; b]$ (polynôme qui interpole f au point $\left(\frac{a+b}{2}, f\left(\frac{a+b}{2}\right)\right)$ et donc de degré 0), on a

$$\tilde{f}(x) = f\left(\tfrac{a+b}{2}\right)$$
$$\int_a^b f(x)\mathrm{d}x \approx \int_a^b \tilde{f}(x)\mathrm{d}x = (b-a) f\left(\tfrac{a+b}{2}\right).$$

Cette formule est dite *formule de quadrature du rectangle* ou *du point milieu.*

En utilisant cette formule pour approcher la fonction $t \mapsto \varphi(t, y(t))$ on a

$$\int_{t_n}^{t_{n+1}} \varphi(t, y(t))\mathrm{d}t \approx h\varphi\left(t_n + \frac{h}{2}, y\left(t_n + \frac{h}{2}\right)\right)$$

et on obtient

$$\begin{cases} u_0 = y(t_0) = y_0, \\ u_{n+1} = u_n + h\varphi\left(t_n + \frac{h}{2}, u_{n+1/2}\right) \quad n = 0, 1, 2, \dots N-1 \end{cases}$$

où $u_{n+1/2}$ est une approximation de $y(t_n + h/2)$.

Cependant, $u_{n+1/2}$ est une valeur inconnue, on doit alors en calculer une approximation $\tilde{u}_{n+1/2}$. Pour cela nous pouvons utiliser deux stratégies différentes :

1. Utiliser une prédiction d'Euler progressive sur un demi-pas : $\tilde{u}_{n+1/2} = u_n + (h/2)\varphi(t_n, u_n)$.

   On peut ainsi remplacer $\varphi(t_n + h/2, u_{n+1/2})$ par $\varphi(t_n + h/2, \tilde{u}_{n+1/2})$.

   Nous avons construit ainsi un nouveau schéma appelé **schéma d'Euler modifié** qui s'écrit comme suit et qui est à 1 pas explicite car il permet d'expliciter u_{n+1} en fonction de u_n. Cette technique d'utiliser un schéma pour prédire une valeur de la solution en un point qui n'est pas un nœud de la discrétisation est très utilisée et constitue l'une des bases des méthodes de type Runge-Kutta. Ici on a un schéma de type Runge-Kutta à un étage.

$$\text{(EM)} \quad \begin{cases} u_0 = y(t_0) = y_0, \\ \tilde{u}_{n+1/2} = u_n + \frac{h}{2}\varphi(t_n, u_n), \\ u_{n+1} = u_n + h\varphi\left(t_n + \frac{h}{2}, \tilde{u}_{n+1/2}\right) \quad n = 0, 1, 2, \dots N-1 \end{cases}$$

2. Utiliser une moyenne : $\tilde{u}_{n+1/2} = \dfrac{u_n + u_{n+1}}{2}$.

   On peut ainsi remplacer $\varphi(t_n + h/2, u_{n+1/2})$ par $\varphi(t_n + h/2, \tilde{u}_{n+1/2})$.

   Nous avons construit ainsi un nouveau schéma qu'on appelera **schéma RK1_M** qui s'écrit comme suit et qui est à 1 pas implicite car il ne permet pas d'expliciter u_{n+1} en fonction de u_n. Pour calculer u_{n+1} il faudra utiliser un schéma pour le calcul du zéro d'une fonction quelconque. En effet, u_{n+1} est solution de l'équation $x = u_n + h\varphi(t_n + \frac{h}{2}, \frac{u_n+x}{2})$, c'est-à-dire un zéro de la fonction (en général non linéaire)

$$x \mapsto -x + u_n + h\varphi\left(t_n + \frac{h}{2}, \frac{u_n + x}{2}\right).$$

   À nouveau, il s'agit d'un schéma de type Runge-Kutta à un étage.

$$\text{(RK1_M)} \quad \begin{cases} u_0 = y(t_0) = y_0, \\ u_{n+1} = u_n + h\varphi\left(\dfrac{t_n + t_{n+1}}{2}, \dfrac{u_n + u_{n+1}}{2}\right) \quad n = 0, 1, 2, \dots N-1 \end{cases}$$

```
def EM(phi,tt,y0):
    h  = tt[1]-tt[0]
    uu = np.empty_like(tt)
    N = len(tt)-1
    uu[0] = y0
    for n in range(N):
        u_tilde = uu[n]+0.5*h*phi(tt[n],uu[n])
        uu[n+1] = uu[n]+h*phi(tt[n]+h/2,u_tilde)
    return uu

def RK1_M(phi,tt,y0):
    h  = tt[1]-tt[0]
    uu = np.empty_like(tt)
    N = len(tt)-1
    uu[0] = y0
```



(a) John Crank (1916-2006)



(b) Phyllis Nicolson (1917-1968)

FIGURE 5.3. – Crank et Nicolson

```
    for n in range(N):
        fct_implicite = lambda x: -x+uu[n]+h*phi( (tt[n]+tt[n+1])/2,(uu[n]+x)/2 )
        tmp = fsolve(fct_implicite, uu[n])
        uu[n+1] = tmp[0]
    return uu
```

5.1.4. Schéma du trapèze (ou de Crank-Nicolson)

Si on remplace une fonction f par le segment qui relie $(a, f(a))$ à $(b, f(b))$ (polynôme qui interpole f en les points $(a, f(a))$ et $(b, f(b))$ et donc de degré 1), on a

$$\tilde{f}(x) = \frac{f(b) - f(a)}{b - a}(x - a) + f(a)$$
$$\int_a^b f(x)\mathrm{d}x \approx \int_a^b \tilde{f}(x)\mathrm{d}x = \frac{b-a}{2}\left(f(a) + f(b)\right).$$

Cette formule est dite *formule de quadrature du trapèze*.

En utilisant cette formule pour approcher la fonction $t \mapsto \varphi(t, y(t))$ on a

$$\int_{t_n}^{t_{n+1}} \varphi(t, y(t))\mathrm{d}t \approx \frac{h}{2}\Big(\varphi(t_n, y(t_n)) + \varphi(t_{n+1}, y(t_{n+1}))\Big)$$

et on obtient le **schéma du trapèze ou de Crank-Nicolson**.

$$\text{(CN)} \quad \begin{cases} u_0 = y(t_0) = y_0, \\ u_{n+1} = u_n + \dfrac{h}{2}\Big(\varphi(t_n, u_n) + \varphi(t_{n+1}, u_{n+1})\Big) \end{cases} \quad n = 0, 1, 2, \dots N-1$$

En fait, ce schéma fait la moyenne des schémas d’Euler progressif et rétrograde.

Il s’agit à nouveau d’un schéma à 1 pas implicite, car il ne permet pas d’expliciter directement u_{n+1} en fonction de u_n lorsque la fonction φ n’est pas triviale. Pour calculer u_{n+1} il faudra utiliser un schéma pour le calcul du zéro



FIGURE 5.4. – Karl Heun (1859-1929)

d'une fonction quelconque. En effet, u_{n+1} est solution de l'équation $x = u_n + \frac{h}{2}\left(\varphi(t_n, u_n) + \varphi(t_{n+1}, x)\right)$, c'est-à-dire un zéro de la fonction (en général non linéaire)

$$x \mapsto -x + u_n + \frac{h}{2}\Big(\varphi(t_n, u_n) + \varphi(t_{n+1}, x)\Big).$$

```
def CN(phi,tt,y0):
    h  = tt[1]-tt[0]
    uu = np.empty_like(tt)
    N = len(tt)-1
    uu[0] = y0
    for n in range(N):
        fct_implicite = lambda x: -x+uu[n]+0.5*h*( phi(tt[n],uu[n])+phi(tt[n+1],x) )
        tmp = fsolve(fct_implicite, uu[n])
        uu[n+1] = tmp[0]
    return uu
```

5.1.5. Schéma de Heun

Pour éviter le calcul implicite de u_{n+1} dans le schéma du trapèze, nous pouvons utiliser une prédiction d'Euler progressive et remplacer le u_{n+1} dans le terme $\varphi(t_{n+1}, u_{n+1})$ par $\tilde{u}_{n+1} = u_n + h\varphi(t_n, u_n)$. Nous avons construit ainsi un nouveau schéma appelé **schéma de Heun**. Plus précisément, la méthode de Heun s'écrit

$$\text{(H)} \quad \begin{cases} u_0 = y(t_0) = y_0, \\ \tilde{u}_{n+1} = u_n + h\varphi(t_n, u_n), \\ u_{n+1} = u_n + \dfrac{h}{2}\Big(\varphi(t_n, u_n) + \varphi(t_{n+1}, \tilde{u}_{n+1})\Big) \end{cases} \quad n = 0, 1, 2, \ldots N-1$$

On a ainsi construit un schéma à 1 pas explicite. Cette technique de prédiction-correction est très utilisée pour transformer un schéma implicite en schéma explicite.

```
def heun(phi,tt,y0):
    h  = tt[1]-tt[0]
    uu = np.empty_like(tt)
    N = len(tt)-1
    uu[0] = y0
    for n in range(N):
        k0 = phi( tt[n], uu[n] )
```



FIGURE 5.5. – Thomas Simpson (1710-1761)

```
        k1 = phi( tt[n+1], uu[n] + h*k0  )
        uu[n+1] = uu[n] + 0.5*h*(k0+k1)
    return uu
```

5.1.6. Remarques

1. À première vue, il semble que les schémas d'**Euler progressif** et de **Heun** soient préférable aux schémas d'**Euler rétrograde** et de **Crank-Nicolson** puisque ces derniers ne sont pas explicites. Cependant, on verra que les méthodes d'Euler implicite et de Crank-Nicolson sont inconditionnellement A-stable (autrement dit, on peut choisir n'importe quel pas $h > 0$, contrairement aux méthodes explicites où le pas doit être suffisamment petit pour garantir la stabilité). C'est aussi le cas de nombreuses autres méthodes implicites. Cette propriété rend les méthodes implicites attractives, bien qu'elles soient plus coûteuses que les méthodes explicites.
2. Pour la mise en application d'un schéma, il faut aussi prendre en compte l'influence des erreurs d'arrondi. En effet, afin de minimiser l'erreur globale théorique, on pourrait être tenté d'appliquer une méthode avec un pas très petit, par exemple de l'ordre de 10^{-16}, mais ce faisant, outre que le temps de calcul deviendrait irréaliste, très rapidement les erreurs d'arrondi feraient diverger la solution approchée. En pratique, il faut prendre h assez petit pour que la méthode converge, mais pas trop petit non plus pour que les erreurs d'arrondi ne donnent pas lieu à des résultats incohérents et pour que les calculs puissent être effectués en un temps raisonnable.

5.2. Annexes

5.2.1. Schémas basés sur la formule de quadrature de Simpson

Si on remplace une fonction f par la parabole qui passe par $(a, f(a))$, $(b, f(b))$ et $\left(\frac{a+b}{2}, f\left(\frac{a+b}{2}\right)\right)$ (polynôme qui interpole f en les points $(a, f(a))$, $(b, f(b))$ et $\left(\frac{a+b}{2}, f\left(\frac{a+b}{2}\right)\right)$ et donc de degré 2), dans la base de Lagrange on a

$$\begin{aligned}\tilde{f}(x) = f(a)&\frac{(x-b)\left(x-\frac{a+b}{2}\right)}{(a-b)\left(a-\frac{a+b}{2}\right)}\\ &+ f\left(\tfrac{a+b}{2}\right)\frac{(x-a)(x-b)}{\left(\frac{a+b}{2}-a\right)\left(\frac{a+b}{2}-b\right)}\\ &+ f(b)\frac{(x-a)\left(x-\frac{a+b}{2}\right)}{(b-a)\left(b-\frac{a+b}{2}\right)}\end{aligned}$$

$$\int_a^b f(x)\mathrm{d}x \approx \int_a^b \tilde{f}(x)\mathrm{d}x = \frac{b-a}{6}\left(f(a) + 4f\left(\tfrac{a+b}{2}\right) + f(b)\right).$$

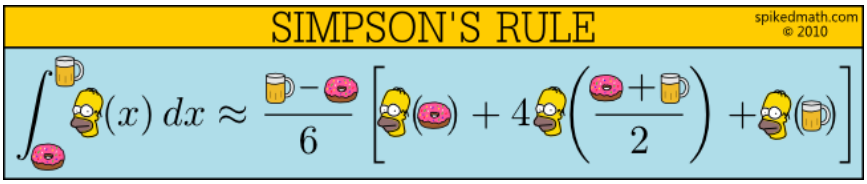


FIGURE 5.6. – Simpson's Rule

Cette formule est dite *formule de Simpson* (voir la Figure Figure 5.6).

5.2.1.1. Schéma de Simpson implicite

En utilisant cette formule pour approcher la fonction $t \mapsto \varphi(t, y(t))$ on a

$$\int_{t_n}^{t_{n+1}} \varphi(t, y(t))\mathrm{d}t \approx \frac{h}{6}\left(\varphi(t_n, y(t_n)) + 4\varphi\left(t_n + \frac{h}{2}, y\left(t_n + \frac{h}{2}\right)\right) + \varphi(t_{n+1}, y(t_{n+1}))\right)$$

et on obtient

$$\begin{cases} u_0 = y(t_0) = y_0, \\ u_{n+1} = u_n + \frac{h}{6}\left(\varphi(t_n, u_n) + 4\varphi\left(t_n + \frac{h}{2}, u_{n+1/2}\right) + \varphi(t_{n+1}, u_{n+1})\right) \quad n = 0, 1, 2, \dots N-1 \end{cases}$$

où $u_{n+1/2}$ est une approximation de $y(t_n + h/2)$.

Puisque $u_{n+1/2}$ est une valeur inconnue, on doit en calculer une approximation $\tilde{u}_{n+1/2}$. Pour cela nous pouvons utiliser soit la moyenne $\tilde{u}_{n+1/2} = \dfrac{u_n + u_{n+1}}{2}$ (comme dans le schéma RK1_M), soit une prédiction d'Euler progressive sur un demi-pas $\tilde{u}_{n+1/2} = u_n + (h/2)\varphi(t_n, u_n)$ (comme dans le schéma EM).

Si par exemple on utilise la moyenne, on obtient le schéma suivant

$$\text{(SI-moy)} \quad \begin{cases} u_0 = y(t_0) = y_0, \\ u_{n+1} = u_n + \frac{h}{6}\left(\varphi(t_n, u_n) + 4\varphi\left(t_n + \frac{h}{2}, \frac{u_n + u_{n+1}}{2}\right) + \varphi(t_{n+1}, u_{n+1})\right) \quad n = 0, 1, 2, \dots N-1 \end{cases}$$

Il s'agit d'un schéma à 1 pas implicite.

Si par exemple on utilise la prédiction d'Euler progressive sur un demi-pas, on obtient le schéma suivant, appelé **schéma de Simpson implicite**, qui s'écrit

$$\text{(SI)} \quad \begin{cases} u_0 = y(t_0) = y_0, \\ \tilde{u}_{n+1/2} = u_n + \frac{h}{2}\varphi(t_n, u_n), \\ u_{n+1} = u_n + \frac{h}{6}\left(\varphi(t_n, u_n) + 4\varphi\left(t_n + \frac{h}{2}, \tilde{u}_{n+1/2}\right) + \varphi(t_{n+1}, u_{n+1})\right) \quad n = 0, 1, 2, \dots N-1 \end{cases}$$

Il s'agit d'un schéma à 1 pas implicite.

5.2.1.2. Schéma de Simpson explicite

Pour éviter le calcul implicite de u_{n+1} dans le schéma de SI, nous pouvons utiliser une prédiction d'Euler progressive et remplacer le u_{n+1} dans le terme $\varphi(t_{n+1}, u_{n+1})$ par $\tilde{u}_{n+1} = u_n + h\varphi(t_n, u_n)$. Nous avons construit ainsi un nouveau schéma qu'on appellera **schéma de Simpson explicite RK** et qui s'écrit

$$\text{(SE-RK)} \quad \begin{cases} u_0 = y(t_0) = y_0, \\ \tilde{u}_{n+1/2} = u_n + \frac{h}{2}\varphi(t_n, u_n), \\ \breve{u}_{n+1} = u_n + h\varphi(t_n, u_n), \\ u_{n+1} = u_n + \frac{h}{6}\left(\varphi(t_n, u_n) + 4\varphi\left(t_n + \frac{h}{2}, \tilde{u}_{n+1/2}\right) + \varphi(t_{n+1}, \breve{u}_{n+1})\right) \quad n = 0, 1, 2, \dots N-1 \end{cases}$$

Il s'agit à nouveau d'un schéma à 1 pas explicite de type Runge-Kutta à deux étages.

5.2.1.3. Schéma de Simpson explicite-variante

Notons qu'on aurait pu remplacer le u_{n+1} dans le terme $\varphi(t_{n+1}, u_{n+1})$ par une approximation utilisant $\tilde{u}_{n+1/2}$ comme une prédiction d'Euler progressive à partir de $t_n + h/2$, ce qui donne $\hat{u}_{n+1} = \tilde{u}_{n+1/2} + \frac{h}{2}\varphi(t_n + h/2, \tilde{u}_{n+1/2})$:

$$\text{(SEV-RK)} \quad \begin{cases} u_0 = y(t_0) = y_0, \\ \tilde{u}_{n+1/2} = u_n + \frac{h}{2}\varphi(t_n, u_n), \\ \hat{u}_{n+1} = \tilde{u}_{n+1/2} + \frac{h}{2}\varphi(t_n + \frac{h}{2}, \tilde{u}_{n+1/2}), \\ u_{n+1} = u_n + \frac{h}{6}\left(\varphi(t_n, u_n) + 4\varphi\left(t_n + \frac{h}{2}, \tilde{u}_{n+1/2}\right) + \varphi(t_{n+1}, \hat{u}_{n+1})\right) \quad n = 0, 1, 2, \dots N-1 \end{cases}$$

5.2.1.4. Schémas de Simpson multipas

Au lieu de travailler sur $[t_n, t_{n+1}]$ on peut répéter la même construction sur $[t_{n-1}, t_{n+1}]$ (et donc intégrer entre t_{n-1} et t_{n+1}). Il faut alors initialiser le schéma avec $u_0 = y_0$ et u_1 calculé par exemple avec le schéma d'Euler explicite.

$$\text{(SMI à deux pas)} \quad \begin{cases} u_0 = y(t_0) = y_0, \\ u_1 = u_0 + h\varphi(t_0, u_0), \\ u_{n+1} = u_{n-1} + \frac{h}{3}\left(\varphi(t_{n-1}, u_{n-1}) + 4\varphi(t_n, u_n) + \varphi(t_{n+1}, u_{n+1})\right) \quad n = 1, 2, 3, \dots N-1 \end{cases}$$

On peut ensuite éviter le calcul implicite de u_{n+1} dans le schéma de SMI en utilisant une prédiction d'Euler progressive (dans l'esprit du schéma de Heun) et remplacer le u_{n+1} dans le terme $\varphi(t_{n+1}, u_{n+1})$ par $\tilde{u}_{n+1} = u_n + h\varphi(t_n, u_n)$. :

$$\text{(SME à deux pas)} \quad \begin{cases} u_0 = y(t_0) = y_0, \\ u_1 = u_0 + h\varphi(t_0, u_0), \\ \tilde{u}_{n+1} = u_n + h\varphi(t_n, u_n), \\ u_{n+1} = u_{n-1} + \frac{h}{3}\left(\varphi(t_{n-1}, u_{n-1}) + 4\varphi(t_n, u_n) + \varphi(t_{n+1}, \tilde{u}_{n+1})\right) \quad n = 1, 2, 3, \dots N-1 \end{cases}$$

5.2.2. Annexe : automatisation de la construction de schémas

Pour vérifier nos calculs d'interpolation puis intégration, nous pouvons utiliser le package de calcul formel `SymPy`.

On notera

- `t_n` : t_n
- `t_nm1` : $t_{n-1} = t_n - h$
- `t_np1` : $t_{n+1} = t_n + h$
- `t_np12` : $t_{n+1/2} = t_n + \frac{h}{2}$

- `phi_n` : $\varphi_n = \varphi(t_n, y(t_n))$
- `phi_nm1` : $\varphi_{n-1} = \varphi(t_{n-1}, y(t_{n-1}))$
- `phi_np1` : $\varphi_{n+1} = \varphi(t_{n+1}, y(t_{n+1}))$
- `phi_np12` : $\varphi_{n+1/2} = \varphi(t_{n+1/2}, y(t_{n+1/2}))$

```
phi_n    = sp.Symbol(r'\varphi_{n}')
phi_np12 = sp.Symbol(r'\varphi_{n+\frac{1}{2}}')
phi_np1  = sp.Symbol(r'\varphi_{n+1}')
phi_nm1  = sp.Symbol(r'\varphi_{n-1}')
sp.var('h, t, t_n')

t_np1  = t_n + h
t_np12 = t_n + h/2
t_nm1  = t_n - h

# Schémas : (Nom, Points pour l'interpolation, Bornes d'intégration, u_départ)
mes_schemas = [
    ("EE",      [(t_n, phi_n)],                          (t_n, t_np1),  "u_n"),
    ("EI",      [(t_np1, phi_np1)],                      (t_n, t_np1),  "u_n"),
    ("EM-init", [(t_np12, phi_np12)],                    (t_n, t_np1),  "u_n"),
    ("CN",      [(t_np1, phi_np1), (t_n, phi_n)],        (t_n, t_np1),  "u_n"),
    ("SI-init", [(t_np1, phi_np1), (t_np12, phi_np12), (t_n, phi_n)], (t_n, t_np1), "u_n"),
    ("SMI",     [(t_np1, phi_np1), (t_n, phi_n), (t_nm1, phi_nm1)],   (t_nm1, t_np1), "u_{n-1}")
]

for nom, points, bornes, u_start in mes_schemas:
    p = sp.interpolate(points, t)
    res = sp.integrate(p, (t, bornes[0], bornes[1])).factor()
    display(Markdown(f"- **{nom}**: $\\displaystyle u_{{n+1}} = {u_start} + {sp.latex(res)}$"))
```

- **EE** : $u_{n+1} = u_n + \varphi_n h$
- **EI** : $u_{n+1} = u_n + \varphi_{n+1} h$
- **EM-init** : $u_{n+1} = u_n + \varphi_{n+\frac{1}{2}} h$
- **CN** : $u_{n+1} = u_n + \dfrac{h\left(\varphi_{n+1} + \varphi_n\right)}{2}$
- **SI-init** : $u_{n+1} = u_n + \dfrac{h\left(\varphi_{n+1} + 4\varphi_{n+\frac{1}{2}} + \varphi_n\right)}{6}$
- **SMI** : $u_{n+1} = u_{n-1} + \dfrac{h\left(\varphi_{n+1} + \varphi_{n-1} + 4\varphi_n\right)}{3}$

Chapitre 6.

Propriétés du problème de Cauchy

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import fsolve
import sympy as sp
from IPython.display import Markdown, display, Math
plt.rcParams.update({'font.size': 12})
```

En général, il ne suffit pas qu'un schéma numérique soit convergent pour qu'il donne de bons résultats sur n'importe quelle équation différentielle. Dès qu'on tente d'approcher une solution, de nombreux facteurs peuvent en fait nous en éloigner : une étude mathématique sophistiquée doit alors être envisagée. En particulier, il faut que le problème de Cauchy soit

- mathématiquement bien posé : sa solution existe et est unique,
- numériquement bien posé : sa solution n'est pas trop perturbée par une erreur initiale ou des perturbations du second membre,
- bien conditionné (= non raide) : les méthodes numériques usuelles peuvent calculer une solution approchée raisonnable en un nombre raisonnable d'itérations.

Ces définitions sont celles adoptés aux pages 235-236 dans le livre de Jean-Pierre DEMAILLY, *Analyse Numérique et équations différentielles*.

6.1. Problème mathématiquement bien posé : existence et unicité de la solution

Définition de problème mathématiquement bien posé
On dit qu'un problème de Cauchy est mathématiquement bien posé s'il existe une et une seule solution.
Si ce n'est pas le cas, on dit qu'il est **mathématiquement mal posé**.

Exemple de problème mathématiquement mal posé
Le problème de Cauchy

$$\begin{cases} y'(t) = \sqrt[3]{y(t)}, & \text{pour } t > 0, \\ y(0) = 0. \end{cases}$$

ne satisfait pas les hypothèses du théorème de Cauchy-Lipschitz, car la fonction $\varphi(t, y) = \sqrt[3]{y}$ n'est pas lipschitzienne par rapport à la deuxième variable en $y = 0$ (on a $\partial_y \varphi(t, y) = \frac{1}{3\sqrt[3]{y^2}} \xrightarrow[y \to 0]{} +\infty$).

On ne peut donc pas garantir l'unicité de la solution et, en effet, il est simple de vérifier que, pour tout $t \geq 0$, les trois fonctions

- $y_1(t) = 0$ pour tout t
- $y_{2,3}(t) = \pm\sqrt{\frac{8}{27}t^3} = \pm\frac{2\sqrt{2}}{3\sqrt{3}}\sqrt{t^3} = \pm\frac{2\sqrt{6}}{9}\sqrt{t^3}$

sont solution du problème de Cauchy donné.

```
# Remarque : si on demande à sympy de calculer toutes les solutions, il ne trouve pas la solution
↪  constante = 0

t = sp.Symbol('t')
y = sp.Function('y')

edo = sp.Eq( sp.diff(y(t),t) , y(t)**(1/sp.S(3))  )
t0, y0 = 0, 0
display(Markdown(f"Equation différentielle : ${sp.latex(edo)}$"))
display(Markdown(f"Condition initiale : $y({t0}) = {y0}$"))

solgenLIST = sp.dsolve(edo,y(t), ics={y(0):0} )
display(Markdown(f"Solution(s) : ${sp.latex(solgenLIST)}$"))
```

Equation différentielle : $\frac{d}{dt}y(t) = \sqrt[3]{y(t)}$

Condition initiale : $y(0) = 0$

Solution(s) : $\left[y(t) = -\frac{2\sqrt{6}t^{\frac{3}{2}}}{9},\ y(t) = \frac{2\sqrt{6}t^{\frac{3}{2}}}{9}\right]$

Que se passe-t-il lorsqu'on utilise un schéma pour un problème mathématiquement MAL posé ? Lorsqu'on utilise une méthode numérique, **on ne peut pas savoir quelle solution ce schéma approche et différents schémas peuvent approcher différentes solutions.**

Dans l'exemple précédent, si on utilise la méthode d'Euler explicite, on trouve que $u_n = 0$ pour tout n, ce qui correspond à la solution y_1.

Exercice : problème MAL posé mathématiquement

- Montrer que le problème de Cauchy suivant admet une infinité de solutions de classe $\mathcal{C}^1(\mathbb{R})$:

$$\begin{cases} y'(t) = 2\sqrt{|y(t)|}, & t > 0 \\ y(0) = 0. \end{cases}$$

- Parmi ces solutions, quelle solution approche-t-on si on utilise la méthode d'Euler explicite ? et la méthode d'Euler implicite ?
- Que se passe-t-il si la donnée initiale est $y(0) = y_0 > 0$?

Correction

Sur $\mathbb{R}^+$ la fonction $\varphi(t, y) = 2\sqrt{y}$ n'est pas lipschitzienne par rapport à y au voisinage de $(t, 0)$ (car $\partial_y\varphi = 1/\sqrt{y} \to \infty$ lorsque $y \to 0$), donc le théorème d'existence et unicité locale n'est pas valable au voisinage de $(0, 0)$. Même raisonnement sur $\mathbb{R}^-$.

L'EDO est à variables séparables, on peut donc calculer explicitement toutes les solutions du problème de Cauchy :

- elle admet une seule solution constante, la fonction $y(t) = 0$ pour tout $t \in \mathbb{R}^+$,
- et des solutions de la forme $y(t) = (t + c)^2$ pour tout $t \geq c$.

En imposant la CI on trouve que, pour tout $b \in \mathbb{R}^+$, les fonctions

$$y_b(t) = \begin{cases} 0, & \text{si } 0 \leq t \leq b, \\ (t-b)^2, & \text{si } t \geq b, \end{cases}$$

sont de classe $\mathcal{C}^1(\mathbb{R}^+)$ et sont solution du problème de Cauchy donné.

```
# Remarque : si on demande à sympy de calculer les solutions, il donne une réponse "enigmatique"
# ================================================================================================

t = sp.Symbol('t')
y = sp.Function('y')

edo = sp.Eq( sp.diff(y(t),t) , 2*sp.sqrt(abs(y(t)))  )
display(Markdown(f"Equation différentielle : ${sp.latex(edo)}$"))

t0, y0 = 0, 0
display(Markdown(f"Condition initiale : $y({t0}) = {y0}$"))

sol = sp.dsolve(edo,y(t), ics={y(t0):y0})
display(Markdown(f"Solution : ${sp.latex(sol)}$"))
```

Equation différentielle : $\frac{d}{dt}y(t) = 2\sqrt{|y(t)|}$

Condition initiale : $y(0) = 0$

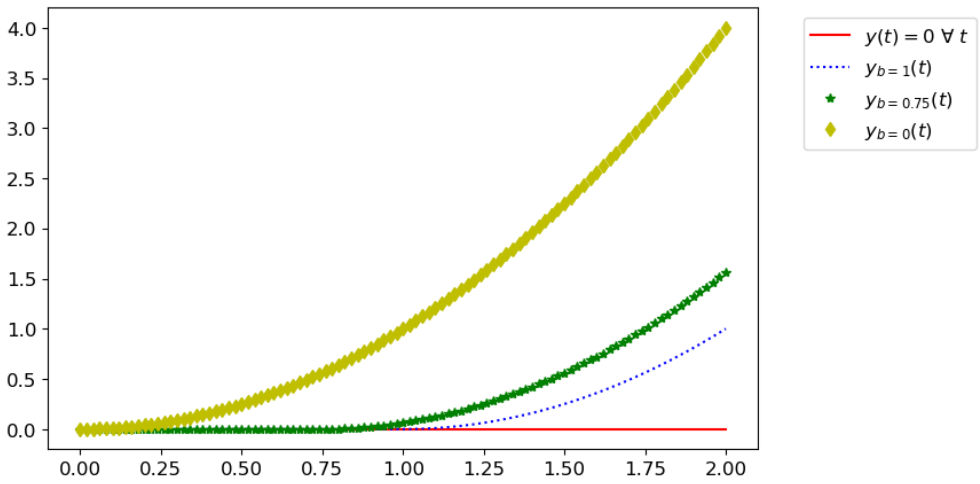
Solution : $\int^{y(t)} \frac{1}{\sqrt{|y|}}\,dy = 2t + \int^{0} \frac{1}{\sqrt{|y|}}\,dy$

Traçons quelques courbes :

```
xx = np.linspace(0,2,101)
yyr = np.zeros_like(xx)

f = lambda x, b : np.where(x<=b, 0, (x-b)**2)
yy1 = f(xx,1)
yy75= f(xx,0.75)
yy0 = f(xx,0)

plt.figure(figsize=(8,5))
plt.plot(xx,yyr  ,'r-', label=r'$y(t)=0$ $\forall$ $t$')
plt.plot(xx,yy1  ,'b:', label=r'$y_{b=1}(t)$')
plt.plot(xx,yy75 ,'g*', label=r'$y_{b=0.75}(t)$')
plt.plot(xx,yy0  ,'yd', label=r'$y_{b=0}(t)$')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left');
```




```
# INTERACTIVE PLOT
# =================
# xx = np.linspace(0, 2, 101)
# fig = plt.figure(figsize=(8, 5))
# plt.xlabel('t')
```

```
# plt.ylabel('y(t)')
# plt.grid(True)
# plt.legend()
# plt.axis([0, 2, 0, 1.5])

# def plot_solution(b,fig):
#     yy = f(xx, b)
#     plt.plot(xx, yy, 'b-', linewidth=2, label=f'$y_{{b={b:.2f}}}(t)$')
#     plt.title(f'Solution pour b = {b:.2f}')
#     display(fig)

# from ipywidgets import interact, FloatSlider
# @interact(b=FloatSlider(min=0, max=1.8, step=0.1, value=0, description='b'))
```

La méthode d'Euler explicite construit la suite

$$\begin{cases} u_0 = y_0 = 0, \\ u_{n+1} = u_n + h\varphi(t_n, u_n) = u_n + hu_n^{1/2}, \quad n = 0, 1, 2, \dots N_h - 1 \end{cases}$$

par conséquent $u_n = 0$ pour tout n : la méthode d'Euler explicite approche la solution constante $y(t) = 0$ pour tout $t \in \mathbb{R}^+$.

La méthode d'Euler implicite construit la suite

$$\begin{cases} u_0 = y_0 = 0, \\ u_{n+1} = u_n + h\varphi(t_{n+1}, u_{n+1}) = u_n + hu_{n+1}^{1/2}, \quad n = 0, 1, 2, \dots N_h - 1. \end{cases}$$

par conséquent $u_0 = 0$ mais u_1 dépend de la méthode de résolution de l'équation implicite $x = 0 + h\sqrt{x}$. Bien-sûr $x = 0$ est une solution mais $x = h^2$ est aussi solution. Si le schéma choisit $u_1 = h^2$, alors $u_n > 0$ pour tout $n \in \mathbb{N}^*$.

Notons que le problème de Cauchy avec une CI $y(0) = y_0 > 0$ admet une et une seule solution, la fonction $y(t) = \frac{1}{4}(t + 2\sqrt{y_0})^2$. Dans ce cas, les deux schémas approchent la même unique solution.

6.2. Problème numériquement bien posé : dépendance continue de la solution par rapport aux données.

Une fois calculée la solution numérique $\{u_n\}_{n=0}^N$ d'un problème de Cauchy mathématiquement bien posé, il est légitime de chercher à savoir dans quelle mesure l'erreur $|y(t_n) - u_n|$ est petite. Cela **dépend** bien sur du schéma choisi, mais aussi **du problème en question**. En effet, étant donné que les erreurs d'arrondi amènent toujours à résoudre un problème perturbé, il est important de savoir si **la solution d'un problème perturbé est proche de la solution du problème non perturbé**.

Définition de problème numériquement bien posé
On dit qu'un problème de Cauchy est **numériquement bien posé** si la solution d'un problème faiblement perturbé (second membre ou condition initiale) possède une solution proche de celle du problème original. Si ce n'est pas le cas, on dit qu'il est **numériquement mal posé**.

Question pratique : *Généralement on utilise un schéma pour approcher la solution d'un problème dont on ne connait pas la solution exacte. Comment peut-on conjecturer que le problème est numériquement mal posé ?*

Idée : on teste un même schéma avec différentes discrétisations (très différentes entre elles) ou on teste différents schémas. Si les solutions obtenues sont très différentes, le problème est probablement numériquement mal posé.

6.2.1. Exemple de problème BIEN posé numériquement

Soit y_0 donné et considérons une petite perturbation ε. Calculons $\lim_{t\to+\infty} y_\varepsilon$ où y_ε est la solution exacte du problème de Cauchy

$$\begin{cases} y'_\varepsilon(t) = -y_\varepsilon(t), & \forall t > 0, \\ y_\varepsilon(0) = y_0 + \varepsilon. \end{cases}$$

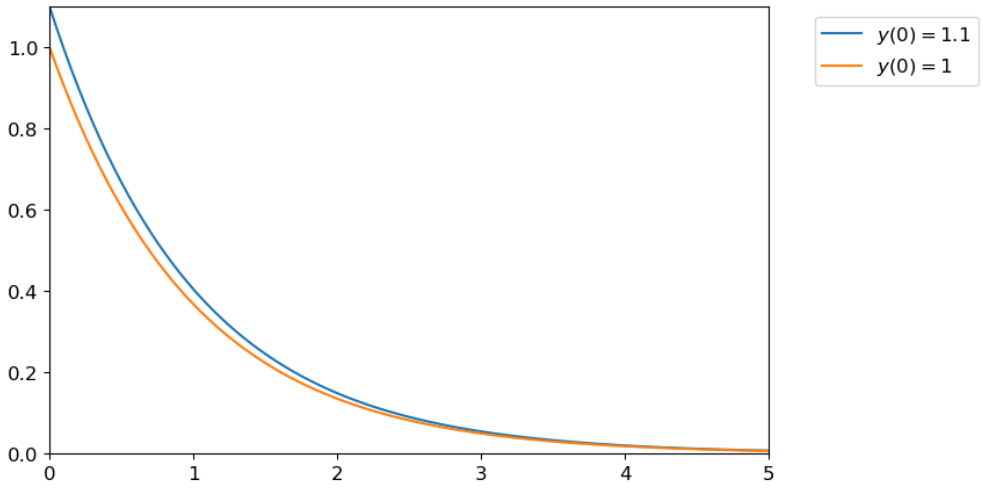
La solution exacte est $y(t) = y_0 e^{-t} + \varepsilon e^{-t}$: l'effet de la perturbation ε s'atténue lorsque $t \to +\infty$ puisque $\varepsilon e^{-t} \xrightarrow[t\to+\infty]{} 0$.
Cela suggère que si une erreur est faite dans une étape d'une méthode d'itération, l'effet de cette erreur s'atténue au cours du temps : **le problème est numériquement bien posé.**

```
y0 = 1

y = lambda t,eps : (y0+eps)*np.exp(-t)

tt  = np.linspace(0,5,101)
yye = y(tt,0.1)
yy0 = y(tt,0)

plt.figure(figsize=(8,5))
plt.plot(tt,yye, label="$y(0)=1.1$")
plt.plot(tt,yy0, label="$y(0)=1$")
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left');
plt.axis([0,5,0,1.1]);
```



6.2.2. Exemple de problème BIEN posé numériquement

Considérons le problème de Cauchy

$$\begin{cases} y'(t) = y(t)\sin(t), & \forall t > 0, \\ y(0) = -1 + \varepsilon. \end{cases}$$

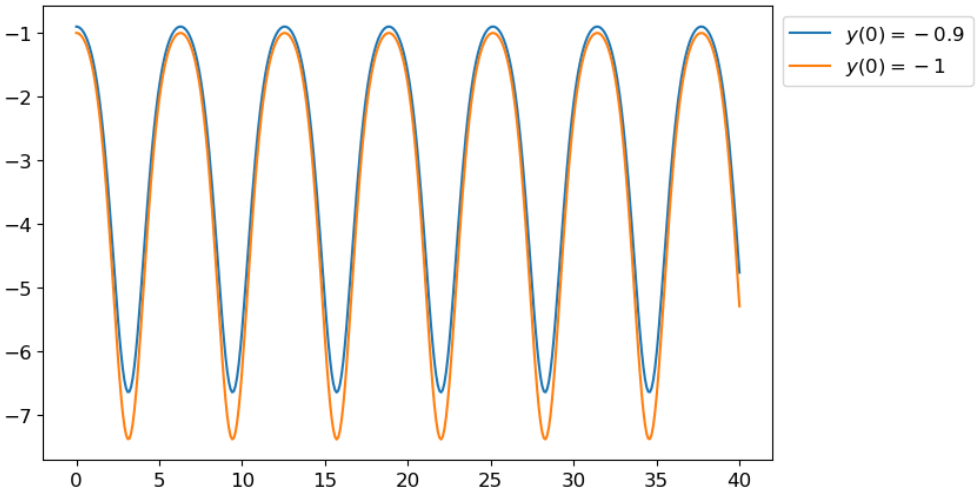
La solution est $y(t) = -e^{1-\cos(t)} + \varepsilon e^{1-\cos(t)}$: l'effet de la perturbation ε reste borné lorsque $t \to +\infty$ puisque $|\varepsilon e^{1-\cos(t)}| \leq \varepsilon e^2$. Cela suggère que si une erreur est faite dans une étape d'une méthode d'itération, l'effet de cette erreur reste borné au cours du temps : **le problème est numériquement bien posé.**

```
y0 = -1
```

```
y = lambda t,eps : (y0+eps)*np.exp(1-np.cos(t))

tt = np.linspace(0,40,401)
yye = y(tt,0.1)
yy0 = y(tt,0)

plt.figure(figsize=(8,5))
plt.plot(tt,yye,label="$y(0)=-0.9$")
plt.plot(tt,yy0,label="$y(0)=-1$")
plt.legend(bbox_to_anchor=(1,1));
```



6.2.3. Exemple de problème MAL posé numériquement

Considérons le problème de Cauchy

$$\begin{cases} y'(t) = y(t), & \forall t > 0, \\ y(0) = y_0 + \varepsilon. \end{cases}$$

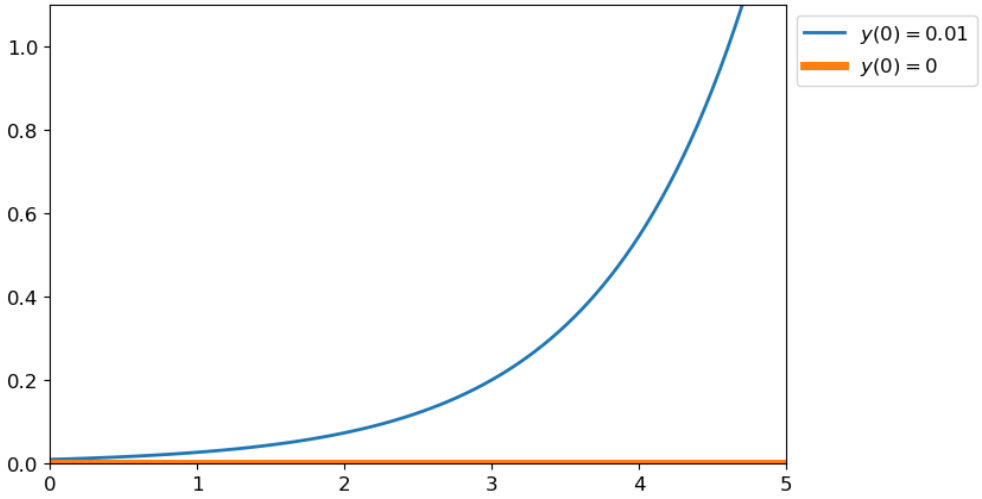
La solution est $y(t) = y_0 e^t + \varepsilon e^t$: l'effet de la perturbation ε **s'amplifie** lorsque $t \to +\infty$ puisque $\varepsilon e^t \xrightarrow[t\to+\infty]{} +\infty$. Cela suggère que si une erreur est faite dans une étape d'une méthode d'itération, l'effet de cette erreur s'amplifie au cours du temps : **le problème est numériquement mal posé.**

```
y0 = 0

y = lambda t,eps : (y0+eps)*np.exp(t)

tt  = np.linspace(0,5,101)
yye = y(tt,0.01)
yy0 = y(tt,0)

plt.figure(figsize=(8,5))
plt.plot(tt, yye, lw='2', label="$y(0)=0.01$")
plt.plot(tt, yy0, lw='5', label="$y(0)=0$")
plt.legend(bbox_to_anchor=(1,1))
plt.axis([0,5,0,1.1]);
```



6.2.4. Exemple de problème MAL posé numériquement

Soit α un nombre quelconque. Étudions la solution exacte y_α du problème de Cauchy

$$\begin{cases} y'_\alpha(t) = 3y_\alpha(t) - 3t, & \forall t \in \mathbb{R}, \\ y_\alpha(0) = \alpha. \end{cases}$$

Comparons ensuite les solutions $y_{1/3}$ et $y_{0.333333}$ en $t = 10$.
La solution, définie sur $\mathbb{R}$, est donnée par

$$y_\alpha(t) = \left(\alpha - \frac{1}{3}\right) e^{3t} + t + \frac{1}{3}.$$

Évaluons y_α en $t = 10$:

- si $\alpha = \frac{1}{3}$ alors $y_\alpha(10) = \frac{31}{3}$,
- si $\alpha = 0.333333$ alors $y_\alpha(10) = (0.333333 - 1/3)e^{30} + 10 + 1/3 = -e^{30}/3000000 + 31/3 \approx 31/3 + 10^7/3$.

Conséquence : si nous faisons le calcul avec l'approximation $\alpha = 0.333333$ au lieu de $1/3$, nous avons $y(10) = 31/3 - e^{30}/3000000$ ce qui représente une différence avec la précédente valeur de $e^{30}/3000000 \approx 10^7/3$. Cet exemple nous apprend qu'**une petite erreur sur la condition initiale (erreur relative d'ordre 10^{-6}) peut provoquer une très grande erreur sur $y(10)$ (erreur relative d'ordre 10^6)**. Ainsi, si le calculateur mis à notre disposition ne calcule qu'avec 6 chiffres significatifs (en virgule flottante), alors $\alpha = 1/3$ devient $\alpha = 0.333333$ et il est inutile d'essayer d'inventer une méthode numérique pour calculer $y(10)$. En effet, la seule erreur sur la condition initiale provoque déjà une erreur inadmissible sur la solution. Nous sommes en présence ici d'un problème **numériquement mal posé**

Vérifions-le avec sympy :

```
t = sp.Symbol('t')
y = sp.Function(r'y_{\alpha}')
alpha = sp.Symbol('alpha')

edo = sp.Eq( sp.diff(y(t),t) , 3*y(t)-3*t  )
t0, y0 = 0, alpha
solpar = sp.dsolve(edo,y(t),ics={y(t0):y0})
display(Markdown("Solution : $\\displaystyle "+sp.latex(solpar)+"$" ))

sol1 = solpar.subs(t,10).subs(alpha,sp.Rational(1,3))
sol2 = solpar.subs(t,10).subs(alpha,0.333333)
display(Markdown("Solution avec $\\alpha=\\frac{1}{3}$ : $\\displaystyle "+sp.latex(sol1)+"$" ))
display(Markdown("Solution avec $\\alpha=0.333333$ : $\\displaystyle "+sp.latex(sol2)+"$" ))
display(Markdown("Différence : $\\displaystyle "+sp.latex((sol1.rhs-sol2.rhs).evalf())+"$"))
```

Solution : $y_\alpha(t) = t + \left(\alpha - \frac{1}{3}\right) e^{3t} + \frac{1}{3}$

Solution avec $\alpha = \frac{1}{3}$: $y_\alpha(10) = \frac{31}{3}$

Solution avec $\alpha = 0.333333$: $y_\alpha(10) = \frac{31}{3} - 3.33333333324415 \cdot 10^{-7} e^{30}$

Différence : 3562158.19374618

6.2.5. Exercices

Exercice : problème numériquement MAL posé à la précision machine
On considère un problème de Cauchy dépendant d'un paramètre ε et on note $y_\varepsilon(t)$ sa solution. Estimer la différence en $t = 50$ entre les solutions y_ε avec $\varepsilon = 10^{-17}$ et $\varepsilon = 0$:

$$\begin{cases} y'_\varepsilon(t) = 5y_\varepsilon(t) - 5, & t \in [0; 50] \\ y_\varepsilon(0) = 1 + \varepsilon. \end{cases}$$

Notons que $\varepsilon = 10^{-17}$ est une valeur très proche de 0 et correspond à l'erreur d'arrondi d'un calculateur qui travaille avec 17 chiffres significatifs.

La solution exacte est donnée par $y_\varepsilon(t) = \varepsilon e^{5t} + 1$ comme on peut le vérifier avec sympy :

```
t = sp.Symbol('t')
epsilon = sp.Symbol(r'\varepsilon')
y = sp.Function(r'y_{\varepsilon}')(t)

edo = sp.Eq(sp.diff(y, t), 5*y - 5)
sol = sp.dsolve(edo, y, ics={y.subs(t, 0): 1 + epsilon})
display(Markdown("Solution : $\\displaystyle "+sp.latex(sol)+"$" ))

# Calcul des solutions pour epsilon=10^(-17) et epsilon=0
sol1 = sol.subs({t: 50, epsilon: sp.Rational(1, 10**17)})
sol0 = sol.subs({t: 50, epsilon: 0})
display(Markdown("Solution avec $\\varepsilon=10^{-17}$ : $\\displaystyle "+sp.latex(sol1)+"$" ))
display(Markdown("Solution avec $\\varepsilon=0$ : $\\displaystyle "+sp.latex(sol0)+"$" ))

# Affichage de la différence entre les solutions
display(Markdown("Différence : $\\displaystyle "+sp.latex((sol1.rhs - sol0.rhs).evalf())+"$"))
```

Solution : $y_\varepsilon(t) = \varepsilon e^{5t} + 1$

Solution avec $\varepsilon = 10^{-17}$: $y_\varepsilon(50) = 1 + \frac{e^{250}}{100000000000000000}$

Solution avec $\varepsilon = 0$: $y_\varepsilon(50) = 1$

Différence : $3.74645461450267 \cdot 10^{91}$

Une très petite erreur sur la condition initiale (de l'ordre de la précision machine) provoque une très grande erreur sur $y(50)$. Ainsi, il est inutile d'essayer d'inventer une méthode numérique pour calculer $y(50)$. En effet, la seule erreur sur la condition initiale provoque déjà une erreur inadmissible sur la solution. Nous sommes en présence ici d'un problème **numériquement mal posé**

Exercice : problème numériquement MAL posé
L'objectif de cet exercice est de bien mettre en évidence l'influence d'une petite perturbation de la condition initiale sur la solution d'un problème numériquement mal posé et d'appliquer la méthode d'Euler explicite pour en approcher la solution exacte.
Considérons le problème de Cauchy

$$\begin{cases} y'(t) = 3\dfrac{y(t)}{t} - \dfrac{5}{t^3}, \\ y(1) = a \end{cases}$$

1. Calculer l'unique solution exacte à ce problème de Cauchy en fonction de a pour $t > 0$. Remarquer qu'avec le changement de variable $u(t) = \frac{y(t)}{t}$ on obtient une EDO linéaire d'ordre 1.
2. Tracer la solution exacte sur l'intervalle $[1;5]$ pour $a = 1$, $a = 1 \pm \frac{1}{1000}$ et $a = 1 \pm \frac{1}{100}$.
3. Pour $a = 1$, sur un autre graphique tracer la solution exacte et les solutions obtenues par la méthode d'Euler explicite avec $N = 20$, $N = 100$, $N = 300$, $N = 5000$. Comment expliquez-vous ce comportement du schéma ?

Correction

Calculons les solutions de l'edo pour $t > 0$. On pose $u(t) = \frac{y(t)}{t}$ ainsi $y(t) = tu(t)$ et $y'(t) = u(t) + tu'(t)$. On obtient l'edo linéaire d'ordre 1 suivante :

$$tu'(t) - 2u(t) = -5t^{-3}.$$

On a $a(t) = t$, $b(t) = -2$ et $g(t) = -5t^{-3}$ donc
$+\ A(t) = \int \frac{b(t)}{a(t)}\mathrm{d}t = \int -\frac{2}{t}\mathrm{d}t = -2\ln(t) = \ln(t^{-2})$, $+\ B(t) = \int \frac{g(t)}{a(t)}e^{A(t)}\mathrm{d}t = \int -5t^{-4}e^{A(t)}\mathrm{d}t = \int -5t^{-6}\mathrm{d}t = t^{-5}$.

On conclut que $u(t) = ce^{-A(t)} + B(t)e^{-A(t)} = ct^2 + t^{-3}$ et $\boxed{y(t) = tu(t) = ct^3 + t^{-2}}$.

En imposant la condition $a = y(1)$ on trouve l'unique solution du problème de Cauchy donné :

$$y(t) = (a-1)t^3 + \frac{1}{t^2}.$$

Vérifions ce calcul avec `sympy` :

```
t = sp.Symbol('t')
y = sp.Function('y')
a = sp.Symbol('a')

edo = sp.Eq( sp.diff(y(t),t) ,  3*y(t)/t-5/t**3  )
t0, y0 = 1, a
solpar = sp.dsolve(edo,y(t),ics={y(t0):y0})
display(Markdown("Solution : $\\displaystyle "+sp.latex(solpar)+"$" ))
```

Solution : $y(t) = \dfrac{t^5\,(a-1)+1}{t^2}$

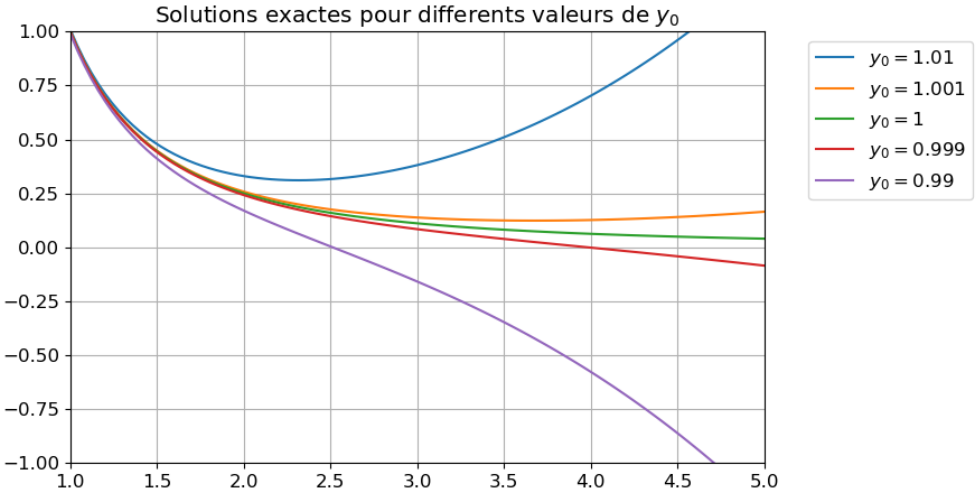
Traçons les solutions exactes pour $a = 1$, $a = 1 \pm \frac{1}{1000}$ et $a = 1 \pm \frac{1}{100}$ sur l'intervalle $[1;5]$:

```
sol_exacte = lambda t,y0 : (y0-1)*t**3+1/t**2

# INITIALISATION
t0     = 1
tfinal = 5
c = 1

tt = np.linspace(t0,tfinal,101)
Y0 = [c+1.e-2,c+1.e-3,c,c-1.e-3,c-1.e-2]

plt.figure(figsize=(8,5))
for y0 in Y0:
    yy = sol_exacte(tt,y0) # [sol_exacte(t,y0) for t in tt]
    plt.plot(tt, yy, label=f'$y_0={y0}$')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.grid()
plt.title('Solutions exactes pour differents valeurs de $y_0$')
plt.axis([t0,tfinal,-c,c]);
```



On constate qu'une petite perturbation de la condition initiale entraîne une grosse perturbation de la solution, en effet

$$\lim_{t\to+\infty} y(t) = \begin{cases} +\infty & \text{si } a > 1, \\ 0^+ & \text{si } a = 1, \\ -\infty & \text{si } a < 1. \end{cases}$$

Si $a = 1$, la solution exacte est $y(t) = t^{-2}$ mais **le problème est numériquement mal posé** car toute (petite) erreur de calcul a le même effet qu'une perturbation de la condition initiale : on "réveille" le terme dormant t^3.

Lorsque l'on essaye d'approcher la solution avec les méthodes classiques à un pas, bien sûr lorsque $N \to +\infty$ on converge vers la solution exacte, cependant en pratique N est fini et on voit qu'inévitablement nous allons approcher une autre solution que celle correspondant à $a = 1$.

```
phi = lambda t, y: 3*y/t - 5/t**3
t0, tfinal, y0 = 1, 5, 1
NN = [20, 100, 300, 5000] # Nombre de points pour les différentes courbes

# --- Définition des "Steps" de chaque schéma ---
def step_EE(phi, t, u, h): return u + h * phi(t, u)
def step_EI(phi, t, u, h): return fsolve(lambda x: u + h * phi(t + h, x) - x, u)[0]
def step_EM(phi, t, u, h): return u + h * phi(t + h/2, u + h/2 * phi(t, u))
def step_HE(phi, t, u, h):
    u_pred = u + h * phi(t, u)
    return u + h/2 * (phi(t, u) + phi(t + h, u_pred))
def step_CN(phi, t, u, h):
    return fsolve(lambda x: u + h/2 * (phi(t, u) + phi(t + h, x)) - x, u)[0]

# --- Solveur Générique ---
def solve_ode(method_step, phi, tt, y0):
    uu = np.zeros_like(tt)
    uu[0] = y0
    h = tt[1] - tt[0]
    for i in range(len(tt) - 1):
        uu[i+1] = method_step(phi, tt[i], uu[i], h)
    return uu

# --- Liste des méthodes à tester ---
# (Nom, Fonction de step, Position dans la figure)
methods = [
    ("EE", step_EE, 1),
    ("EI", step_EI, 3),
    ("EM", step_EM, 4),
    ("HE", step_HE, 5),
    ("CN", step_CN, 6)
]
```

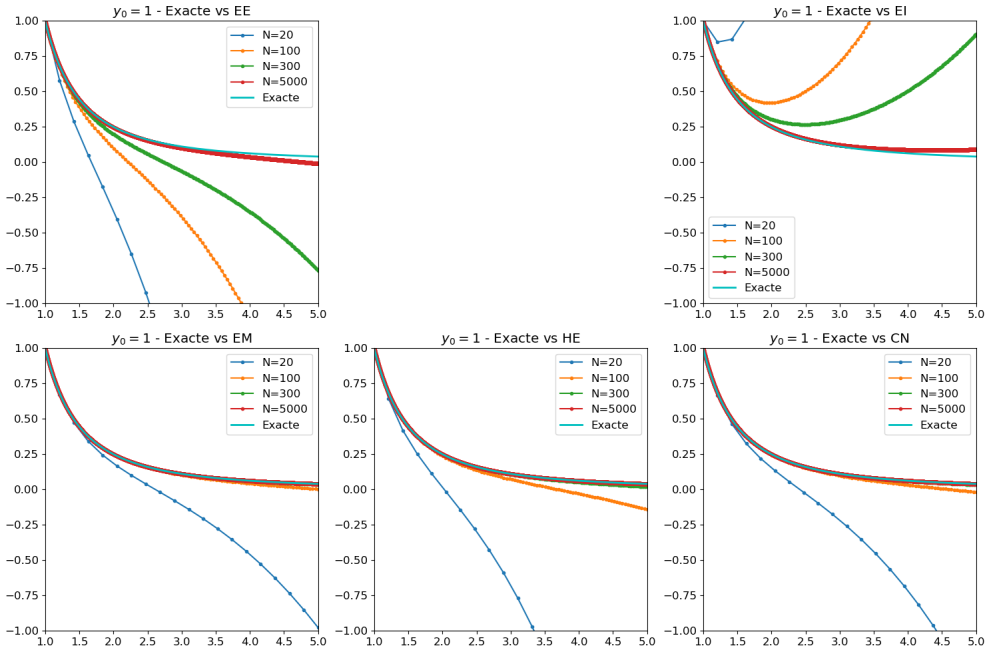
```
# --- Boucle d'affichage ---
plt.figure(figsize=(15, 10))

for name, step_func, pos in methods:
    plt.subplot(2, 3, pos)
    for N in NN:
        tt = np.linspace(t0, tfinal, N)
        uu = solve_ode(step_func, phi, tt, y0)
        plt.plot(tt, uu, '.-', label=f'N={N}')

    # Solution exacte
    t_fine = np.linspace(t0, tfinal, 1000)
    plt.plot(t_fine, sol_exacte(t_fine, y0), 'c-', lw=2, label='Exacte')

    plt.title(f'$y_0={y0}$ - Exacte vs {name}')
    plt.legend()
    plt.axis([t0, tfinal, -c, c])

plt.tight_layout()
plt.show()
```



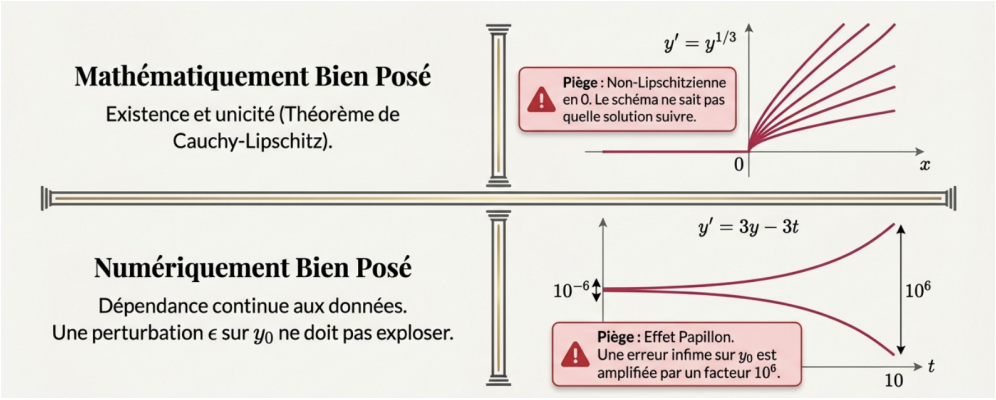


FIGURE 6.1. – Problème de Cauchy bien posé mathématiquement et numériquement

6.3. Problème bien conditionné (= non raide) : solution approchée en un nombre raisonnable d'itérations.

Définition de problème bien conditionné
Une équation différentielle est dite "*stiff*" lorsqu'elle contient des composantes qui évoluent à des vitesses très différentes. Par exemple, considérons une équation différentielle d'ordre 2 (un système des deux équations différentielles). Si l'une varie lentement et l'autre chute très rapidement vers un équilibre, alors on dit que le problème est stiff (ou raide). Ou encore on s'interesse à un problème de Cauchy dont la solution admet une composante rapide qui chute très rapidement vers un équilibre et une composante lente qui évolue lentement. Nous verrons que certains schémas forcent à choisir un pas de temps h minuscule, calé sur la vitesse de la composante rapide, même si celle-ci a déjà atteint son équilibre et ne "sert plus à rien" pour la dynamique globale.
On dit qu'un problème de Cauchy est **bien conditionné** si les méthodes numériques usuelles peuvent donner sa solution en un nombre raisonnable d'opérations.
Dans le cas contraire on parle de **problème raide**.

6.3.1. Exemple de problème MAL conditionné (= raide = *stiff*)

Considérons le problème de Cauchy avec $\beta > 0$:

$$\begin{cases} y'(t) = -\beta y(t), \quad \forall t \in [0;1], \\ y(0) = 1. \end{cases}$$

Ce problème est

- **mathématiquement bien posé** pour tout $\beta \in \mathbb{R}$: il existe une et une seule solution, elle est définie sur $\mathbb{R}$ et est donnée par $y(t) = e^{-\beta t}$;
- **numériquement bien posé** pour tout $\beta > 0$: l'erreur sur la solution sera au pire de l'ordre de l'erreur sur la condition initiale ;
- **mal conditionné** ou **stiff** si $\beta \gg 1$: plus β est grand, plus la solution est raide et seul un pas très petit permet d'approcher la solution exacte dans la partie où la solution chute très rapidement vers son équilibre 0. Certaines méthodes forcent à choisir un pas de temps h calé sur la vitesse de la composante rapide.

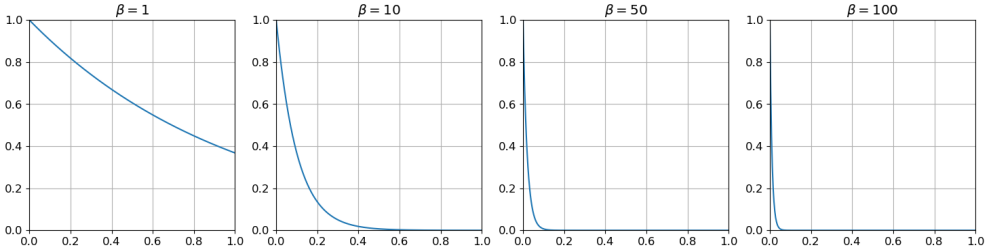
Nous allons tracer la solution exacte pour $\beta = 1$, $\beta = 10$, $\beta = 50$ et $\beta = 100$ sur l'intervalle $[0;1]$ pour illustrer ce phénomène.
Nous allons aussi étudier analytiquement le comportement du schéma d'Euler explicite pour ce problème.

Voici ci-dessous le tracé de la solution exacte pour β allant de 1 à 100 :

```
exacte = lambda t,b : np.exp(-b*t)

t0 = 0
tfinal = 1
tt = np.linspace(t0,tfinal,501)

plt.figure(1, figsize=(18, 4))
for i,b in enumerate([1,10,50,100]):
    plt.subplot(1,4,i+1)
    yy = exacte(tt,b) #[exacte(t,b) for t in tt]
    plt.plot(tt, yy)
    plt.title(f'$\\beta={b}$')
    plt.axis([0,1,0,1])
    plt.grid();
```



La méthode d'Euler explicite pour cette EDO s'écrit

$$u_{n+1} = (1 - \beta h)u_n = \cdots = (1 - \beta h)^{n+1} u_0.$$

La suite obtenue est une suite géométrique de raison $q = 1 - \beta h$. On sait qu'une telle suite

- diverge si $|q| > 1$ ou $q = -1$,
- est stationnaire si $q = 1$,
- converge vers 0 si $|q| < 1$ (de façon monotone vers 0 si $0 < q < 1$).

ainsi la solution numérique converge vers 0 ssi $0 < \beta h < 2$: plus β est grand, plus le pas h doit être petit pour que la solution numérique puisse approcher la solution exacte.

Intuitivement : la solution exacte $y(t) = e^{-\beta t}$ décroit très rapidement lorsque β est grand (le problème devient de plus en plus raide) ; pour que la solution numérique puisse suivre cette décroissance rapide, il faut qu'il y ait des points dans cette zone de décroissance rapide, c'est-à-dire que le pas h soit petit.

On verra plus tard que, si on n'est pas intéressé par cette zone de décroit rapide, mais plutôt par le comportement asymptotique de la solution, on peut utiliser des méthodes numériques spécifiques qui permettent de prendre des pas plus grands. Cela nous amène à la **notion d'*A-stabilité*** que nous allons définir dans les prochains chapitres.

6.3.2. Exercices

Exercie : resolution numérique d'un problème raide
Soient $b, g > 0$ et considérons le problème de Cauchy

$$\begin{cases} y'(t) = -by(t) + g, & t \in [0;1] \\ y(0) = y_0. \end{cases}$$

- **Solution exacte**
  - Donner l'unique solution exacte à ce problème de Cauchy et tracer le graphe $t \mapsto y(t)$ pour $t \geq 0$ en fonction de y_0.
  - Soient $b = g = 10$. Tracer avec python les solutions exactes sur l'intervalle $[0;1]$ pour $y_0 = \frac{g}{b}$ et $y(0) = \frac{g}{b} + 10^{-8}$ pour $b = g = 10$.
  - Sur un graphe à coté tracer les mêmes courbes avec $b = g = 40$. Que peut-on dire lorsque b devient de plus en plus grand ?
- Méthode d'**Euler explicite**
  - Vérifier que la méthode génère une suite arithmético-géométrique et calculer analytiquement pour quelles valeurs de h la suite ainsi construite converge et pour quelles valeurs la convergence est monotone.
  - Soient $b = g = 40$ et $y_0 = \frac{g}{b}$. Tracer les solutions approchées obtenues avec $N = 19$, $N = 23$, $N = 45$, $N = 100$.
  - Répéter pour $y_0 = \frac{g}{b} + 10^{-8}$ et expliquer chaque résultat.
- Méthode d'**Euler implicite**
  - Répéter le même exercice pour la méthode d'Euler implicite. Noter que, étant φ linéaire, la méthode peut être rendu explicite par un calcul élémentaire.

Suggestion : EDO y'(t) = ay(t)+b

Considérons l'équation différentielle ordinaire $y'(t) = ay(t) + b$, où $a \neq 0$ et b sont des constantes réelles et considérons une donnée initiale $y(0) = y_0 \neq 0$.
On cherche à réécrire cette équation sous la forme d'une équation $v'(t) = qv(t)$, où v est une nouvelle fonction à déterminer.
Posons $v(t) = y(t) + \frac{b}{a}$. Alors $v(0) = y_0 + \frac{b}{a}$ et

$$\begin{aligned} v'(t) &= y'(t) \\ &= ay(t) + b \\ &= a\left(v(t) - \frac{b}{a}\right) + b \\ &= av(t). \end{aligned}$$

La solution de cette équation est $v(t) = v(0)e^{at}$.
On conclut alors que $y(t) = v(t) + \frac{b}{a} = v(0)e^{at} + \frac{b}{a} = \left(y_0 + \frac{b}{a}\right)e^{at} - \frac{b}{a}$, autrement dit

$$\left(y(t) + \frac{b}{a}\right) = \left(y_0 + \frac{b}{a}\right)e^{at}.$$

Suggéstion : Suite arithmético-géométrique -> suite géométrique

Une suite (u_n) est dite **arithmético-géométrique** si elle satisfait une relation de la forme $u_{n+1} = au_n + b$ avec $a \neq 1$ et b des réels constants.
On cherche à réécrire cette suite sous la forme d'une **suite géométrique** (v_n) telle que $v_{n+1} = qv_n$ avec q une constante réelle.
On pose $v_n = u_n - \frac{b}{1-a}$. Alors

$$\begin{aligned} v_{n+1} &= u_{n+1} - \frac{b}{1-a} \\ &= au_n + b - \frac{b}{1-a} \\ &= a\left(v_n + \frac{b}{1-a}\right) + b - \frac{b}{1-a} \\ &= av_n + \frac{ab}{1-a} + b - \frac{b}{1-a} \\ &= av_n + \frac{ab + b(1-a) - b}{1-a} \\ &= av_n. \end{aligned}$$

La suite (v_n) est bien géométrique de raison $q = a$ donc $v_n = a^n v_0$.
On conclut alors que $u_n = a^n v_0 + \frac{b}{1-a} = a^n\left(u_0 - \frac{b}{1-a}\right) + \frac{b}{1-a}$, autrement dit

$$\left(u_n - \frac{b}{1-a}\right) = \left(u_0 - \frac{b}{1-a}\right)a^n.$$

Correction : solution exacte

Il s'agit de l'edo linéaire d'ordre 1 $y'(t) = -by(t) + g$, dont la solution est

$$y(t) - \frac{g}{b} = \left(y_0 - \frac{g}{b}\right)e^{-bt}$$

et

$$\lim_{t \to +\infty} y(t) = \frac{g}{b} \text{ pour tout } y_0 \in \mathbb{R}$$

le problème est numériquement bien posé, c'est-à-dire que l'erreur sur la solution sera au pire de l'ordre de l'erreur sur la condition initiale.

Cependant, lorsque b devient de plus en plus grand, la solution devient de plus en plus raide :

```
exacte = lambda t, b, g, y0 : (y0-g/b)*np.exp(-b*t)+g/b

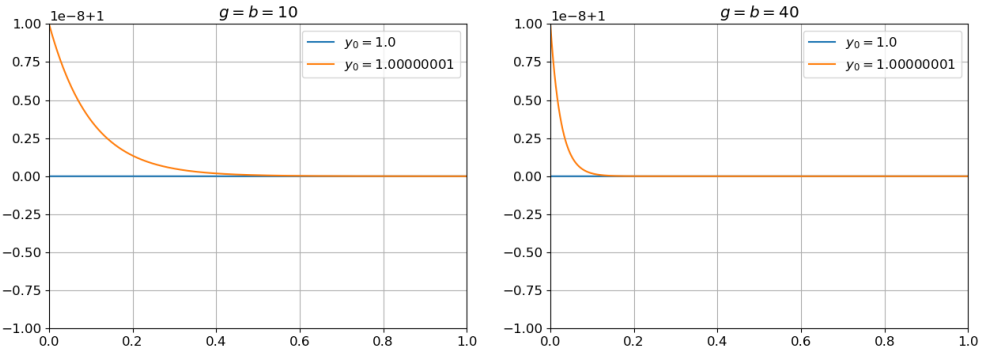
t0 = 0.
tfinal = 1.0
tt = np.linspace(t0,tfinal,10001)

def plot_exacts(tt, b, g):
    Y0 = [g/b, g/b + 1.e-8]
    for y0 in Y0:
        yy = exacte(tt, b, g, y0)
        plt.plot(tt, yy, label=f'$y_0={y0}$')
    plt.title(f"$g=b={g}$")
    plt.axis([0, 1, g/b - 1.e-8, g/b + 1.e-8])
    plt.legend()
    plt.grid()

plt.figure(1, figsize=(15, 5))

plt.subplot(1, 2, 1)
plot_exacts(tt, b=10, g=10)

plt.subplot(1, 2, 2)
plot_exacts(tt, b=40, g=40)
```



Correction : Méthode d'Euler explicite

La méthode d'Euler explicite donne la suite définie par récurrence $u_{n+1} = Au_n + B$ avec $A = 1 - bh$ et $B = gh$, soit encore

$$\left(u_n - \frac{B}{1-A}\right) = \left(u_0 - \frac{B}{1-A}\right) A^n \quad \rightsquigarrow \quad u_n - \frac{g}{b} = (1-hb)^n \left(y_0 - \frac{g}{b}\right).$$

Comparons la solution exacte et la solution approchée :

$$y_n = \frac{g}{b} + \left(y_0 - \frac{g}{b}\right) e^{-bt_n}$$
$$u_n = \frac{g}{b} + \left(y_0 - \frac{g}{b}\right) (1-hb)^n$$

Pour que $u_n \simeq y_n$ il faut que $(1-hb)^n \simeq e^{-bt_n} \in]0, 1]$, donc il faut que $0 \leq (1-hb) < 1$, soit encore $h < \frac{1}{b}$. Comme $h = \frac{T-t_0}{N} = \frac{1}{N}$, cela implique $N \geq b$. Dans cet exemple, pour obtenir une bonne convergence il est nécessaire de prendre un pas h de pus en plus petit que b est grand, c'est-à-dire un coût en calcul plus élevé. On parle de **problème raide** ou mal conditionné.

```
phi = lambda t, y, b, g : -b*y+g

def EE(phi,tt,y0,b,g):
    h = tt[1]-tt[0]
    uu = np.zeros_like(tt)
    uu[0] = y0
    for i in range(len(tt)-1):
        uu[i+1] = uu[i]+h*phi(tt[i],uu[i],b,g)
    return uu

t0 = 0.
tfinal = 1.0

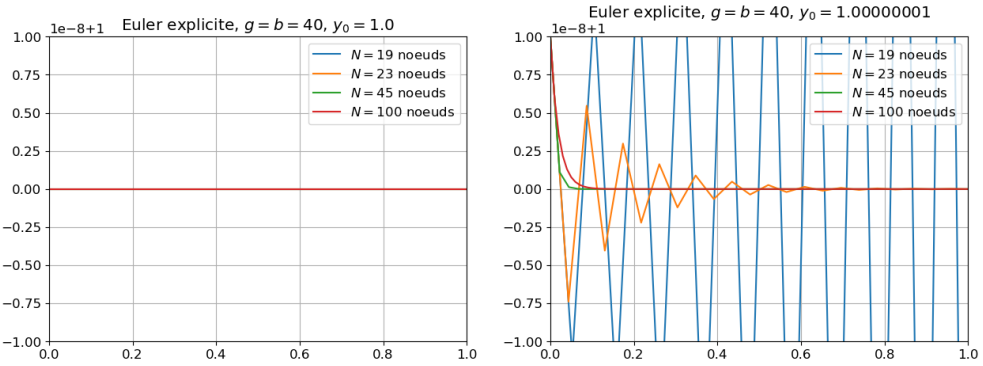
def plot_euler_exp(phi, t0, tfinal, b, g, y0, NN):
    for N in NN:
        tt = np.linspace(t0, tfinal, N+1)
        yy = EE(phi, tt, y0, b, g)
        plt.plot(tt, yy, label=f'$N={N}$ noeuds')
    plt.title(f"Euler explicite, $g=b={g}$, $y_0={y0}$")
    plt.axis([0, 1, g/b - 1.e-8, g/b + 1.e-8])
    plt.legend()
    plt.grid()

g = 40
b = 40
Y0 = [g/b, g/b + 1.e-8]
NN = [19, 23, 45, 100]

plt.figure(figsize=(15, 5))

plt.subplot(1, 2, 1)
plot_euler_exp(phi, t0, tfinal, b, g, y0=Y0[0], NN=NN)

plt.subplot(1, 2, 2)
plot_euler_exp(phi, t0, tfinal, b, g, y0=Y0[1], NN=NN)
```



Correction : Méthode d'Euler implicite

La méthode d'Euler implicite donne la suite définie par récurrence $u_{n+1} = Au_n + B$ avec $A = \frac{1}{1+bh}$ et $B = \frac{gh}{1+bh}$, soit encore

$$\left(u_n - \frac{B}{1-A}\right) = \left(u_0 - \frac{B}{1-A}\right) A^n \quad \rightsquigarrow \quad u_n - \frac{g}{b} = \frac{1}{(1+hb)^n}\left(y_0 - \frac{g}{b}\right).$$

Comparons la solution exacte et la solution approchée :

$$y_n = \frac{g}{b} + \left(y_0 - \frac{g}{b}\right) e^{-bt_n}$$
$$u_n = \frac{g}{b} + \left(y_0 - \frac{g}{b}\right) \left(\frac{1}{1+hb}\right)^n$$

Pour que $u_n \simeq y_n$ il faut que $\frac{1}{(1+hb)^n} \simeq e^{-bt_n} \in]0,1]$, donc il faut que $0 \leq \frac{1}{1+hb} < 1$ c'est-à-dire pour n'importe quelle valeur de h. Dans cet exemple, pour obtenir la convergence il n'est pas nécessaire de limiter le pas h. Cependant, pour une edo quelconque, à chaque pas de temps on doit résoudre une équation non-linéaire.

```
phi = lambda t,y,b,g : -b*y+g

def EI(phi,tt,y0,b,g):
    h = tt[1]-tt[0]
    uu = np.zeros_like(tt)
    uu[0] = y0
    for i in range(len(tt)-1):
        uu[i+1] = g/b + (y0-g/b)/(1+h*b)**i
        uu[i+1] = fsolve(lambda x: x-uu[i]-h*phi(tt[i+1],x,b,g), uu[i])[0]
    return uu

t0 = 0.
tfinal = 1

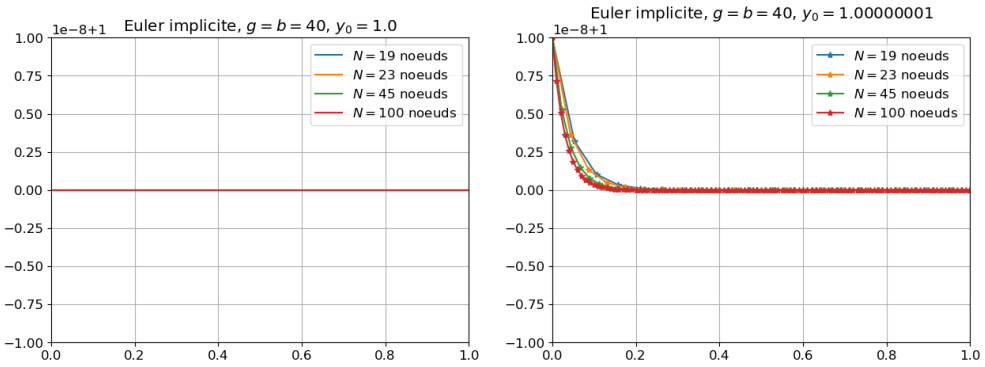
def plot_euler_imp(phi, t0, tfinal, b, g, y0, NN, marker='-'):
    for N in NN:
        tt = np.linspace(t0, tfinal, N+1)
        yy = EI(phi, tt, y0, b, g)
        plt.plot(tt, yy, marker, label=f'$N={N}$ noeuds')
    plt.title(f"Euler implicite, $g=b={g}$, $y_0={y0}$")
    plt.axis([0, 1, g/b - 1.e-8, g/b + 1.e-8])
    plt.legend()
    plt.grid()

g = 40
b = 40
Y0 = [g/b, g/b + 1.e-8]
NN = [19, 23, 45, 100]

plt.figure(figsize=(15, 5))

plt.subplot(1, 2, 1)
plot_euler_imp(phi, t0, tfinal, b, g, y0=Y0[0], NN=NN, marker='-')

plt.subplot(1, 2, 2)
plot_euler_imp(phi, t0, tfinal, b, g, y0=Y0[1], NN=NN, marker='-*')
```



Exercice : problème MAL conditionné (=raide)

$$\begin{cases} y'(t) = -150y(t) + 30 \\ y(0) = \frac{1}{5}. \end{cases}$$

Vérifier qu'il est bien posé mathématiquement et numériquement.
Que se passe-t-il si on utilise la méthode d'Euler explicite ?

Le problème de Cauchy donné admet une et une seule solution, la fonction constante $y(t) = \frac{1}{5}$, il est donc bien posé mathématiquement.

Considérons le problème de Cauchy perturbé

$$\begin{cases} y'_\varepsilon(t) = -150y_\varepsilon(t) + 30 \\ y_\varepsilon(0) = \frac{1}{5} + \varepsilon. \end{cases}$$

La solution exacte y_ε est

$$\left(y_\varepsilon(t) + \frac{30}{-150}\right) = \left(y_\varepsilon(0) + \frac{30}{-150}\right) e^{-150t} \quad \rightsquigarrow \quad \left(y_\varepsilon(t) - \frac{1}{5}\right) = \left(y_\varepsilon(0) - \frac{1}{5}\right) e^{-150t} = \varepsilon e^{-150t}.$$

Si $\varepsilon \to 0$, alors $y_\varepsilon(t) \to \frac{1}{5}$: le problème est numériquement bien posé, car l'erreur sur la solution sera inférieure à l'erreur sur la condition initiale.

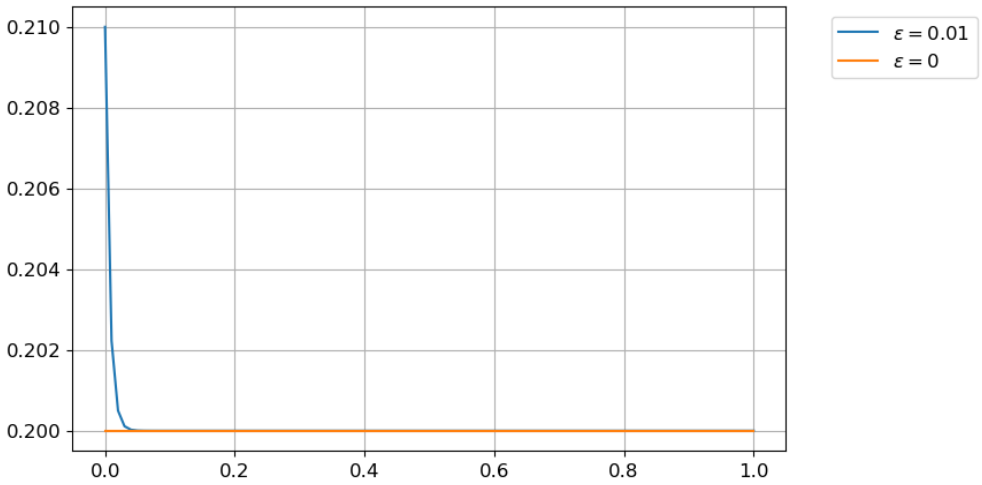
```
# Calcul de la solution exacte
# ============================

t = sp.Symbol('t')
y = sp.Function('y')

eps = sp.Symbol(r'\varepsilon')
t0, y0 = 0, sp.Rational(1,5)+eps
edo = sp.Eq( sp.diff(y(t),t) ,  -150*y(t)+30  )
sol = sp.dsolve(edo,y(t), ics={y(t0):y0})
display(Markdown(f"Solution : ${sp.latex(sol)}$"))

func = sp.lambdify((t,eps),sol.rhs,'numpy')
tt = np.linspace(0,1,101)
yy_eps = func(tt,0.01)
yy_0 = func(tt,0)
plt.figure(figsize=(8,5))
plt.plot(tt,yy_eps,label=r'$\varepsilon=0.01$')
plt.plot(tt,yy_0,label=r'$\varepsilon=0$')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.grid()
```

Solution : $y(t) = \varepsilon e^{-150t} + \frac{1}{5}$



La méthode d'Euler implicite donne la suite définie par récurrence $u_{n+1} = u_n + h(-150u_n + 30) = au_n + b$ avec $a = 1 - 150h$ et $b = 30h$, soit encore

$$\left(u_n - \frac{b}{1-a}\right) = \left(u_0 - \frac{b}{1-a}\right) a^n \quad \rightsquigarrow \quad \left(u_n - \frac{30h}{150h}\right) = \left(u_0 - \frac{30h}{150h}\right) (1-150h)^n$$
$$\rightsquigarrow \quad \left(u_n - \frac{1}{5}\right) = \left(u_0 - \frac{1}{5}\right) (1-150h)^n.$$

Comparons la solution exacte et la solution approchée :

$$y_n - \frac{1}{5} = \varepsilon e^{-150t}$$
$$u_n - \frac{1}{5} = \varepsilon (1-150h)^n$$

Pour que $u_n \simeq y_n$ il faut que $(1-150h)^n \simeq e^{-150(nh)} \in]0,1]$, donc il faut que $|1-150h| < 1$, c'est-à-dire $h < \frac{1}{75}$.

Exercice : problème raide
Il existe des problèmes qui résistent à toutes les méthodes usuelles. On trouve par exemple ces phénomènes en chimie où deux réactions ont des échelles de temps très différentes. Considérons par exemple

$$\begin{cases} y'(t) = -1000(y(t) - e^{-t}) - e^{-t}, & t \in [0;1] \\ y(0) = 0. \end{cases}$$

Vérifier, en calculant la solution exacte, que le problème est bien posé mathématiquement et numériquement et qu'il est raide.

```
# Calcul de la solution exacte
# ============================

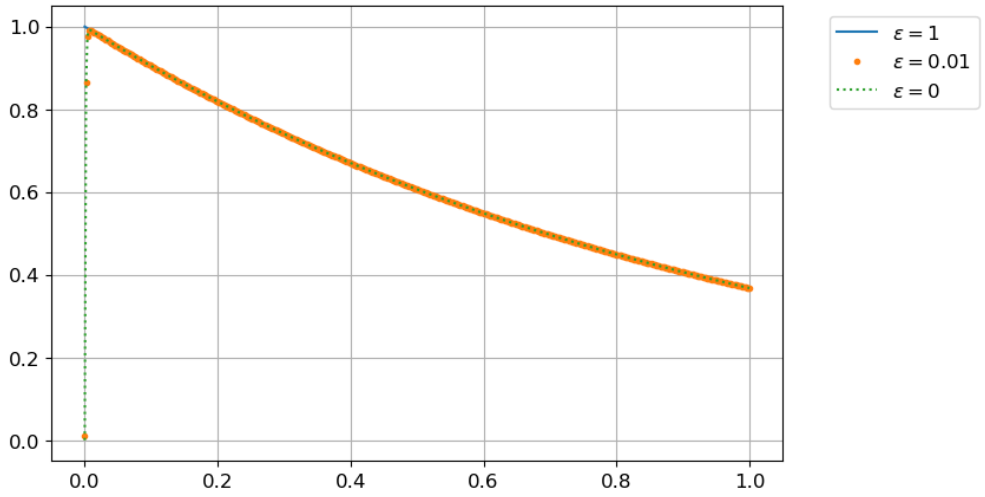
t = sp.Symbol('t')
y = sp.Function('y')

eps = sp.Symbol(r'\varepsilon')
t0, y0 = 0, 0+eps
edo = sp.Eq( sp.diff(y(t),t) ,  -1000*(y(t)-sp.exp(-t))-sp.exp(-t)  )
sol = sp.dsolve(edo,y(t), ics={y(t0):y0})
display(Markdown(f"Solution : ${sp.latex(sol)}$"))

func = sp.lambdify((t,eps),sol.rhs,'numpy')
tt = np.linspace(0,1,501)
yy_1 = func(tt,1)
```

```
yy_eps = func(tt,0.01)
yy_0 = func(tt,0)
plt.figure(figsize=(8,5))
plt.plot(tt,yy_1,label=r'$\varepsilon=1$')
plt.plot(tt,yy_eps,'.',label=r'$\varepsilon=0.01$')
plt.plot(tt,yy_0,':',label=r'$\varepsilon=0$')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.grid()
```

Solution : $y(t) = \left((\varepsilon - 1)\, e^{-999t} + 1\right) e^{-t}$



Le problème est

- bien posé mathématiquement : il admet une et une seule solution, la fonction $y(t) = e^{-t}$, définie sur $\mathbb{R}$,
- bien posé numériquement : l'erreur ε sur la donnée initiale est atténuée au cours du temps,
- raide : la solution exacte $y(t) = e^{-t} - e^{-1000t}$ croit très rapidement ($-e^{-1000t}$) sur un très petit intervalle, puis elle décroit comme e^{-t}. Notons que, si $\varepsilon = 1$, la solution est $y(t) = e^{-t}$.

Propriétés du problème de Cauchy : que retenir ?

- Un problème de Cauchy est **mathématiquement bien posé** s'il existe une et une seule solution. Si ce n'est pas le cas, inutile de chercher à approcher la solution numériquement.
- Un problème de Cauchy est **numériquement bien posé** si la solution d'un problème faiblement perturbé (second membre ou condition initiale) possède une solution proche de celle du problème original. Si ce n'est pas le cas, on peut chercher à approcher la solution numériquement, mais c'est délicat, car une petite erreur peut provoquer une très grande erreur sur la solution. Il faudra alors utiliser des méthodes numériques spécifiques (d'ordre élevé).
- Un problème de Cauchy est **bien conditionné** si les méthodes numériques usuelles peuvent donner sa solution en un nombre raisonnable d'opérations. Dans le cas contraire on parle de problème raide (stiff). Si un problème est raide, il faudra utiliser des méthodes numériques spécifiques (généralement des méthodes A-stable car on verra que c'est grosso modo le même problème que pour les problèmes numériquement mal posés).

Chapitre 7.

Construction de schémas multi-pas linéaires

Considérons le problème de Cauchy dont on suppose l'existence et l'unicité de la solution y sur un intervalle $I = [t_0, T]$:

$$\begin{cases} y'(t) = \varphi(t, y(t)), & \forall t \in I, \\ y(t_0) = y_0, \end{cases}$$

où $\varphi : I \times \mathbb{R} \to \mathbb{R}$ est une fonction donnée et y_0 est une donnée initiale également donnée.

Discrétisation de l'intervalle I : posons $h = \frac{T-t_0}{N} > 0$ (appelé le *pas de discrétisation*) et définissons $t_n \equiv t_0 + nh$ avec $n = 0, 1, 2, \dots, N$ une suite de $N+1$ nœuds de I induisant une discrétisation de l'intervalle I en N sous-intervalles $I_n = [t_n; t_{n+1}]$ chacun de longueur h.

- L'ensemble de $N+1$ valeurs $\{t_0, t_1 = t_0 + h, \dots, t_N = T\}$ représente les points (ou nœud) de la *discrétisation*.

Discrétisation de la solution : pour chaque nœud t_n, on cherche la valeur inconnue u_n qui approche la valeur exacte $y(t_n)$.

- L'ensemble de $N+1$ valeurs $\{y_0, y_1, \dots, y_N\}$ avec $y_n \equiv y(t_n)$ représente la *solution exacte discrète* (en général inconnue).

- L'ensemble de $N+1$ valeurs $\{u_0 = y_0, u_1, \dots, u_N\}$ représente la *solution numérique*. Les schémas que l'on va construire permettent de calculer (explicitement ou implicitement) u_{n+1} à partir de $u_n, u_{n-1}, \dots$ et il est donc possible de calculer successivement u_1, u_2,..., en partant de u_0 par une **formule de récurrence** du type

$$\begin{cases} \left.\begin{array}{l} u_0 = y_0, \text{ connu} \\ u_1 \approx y_1, \\ \vdots \\ u_\kappa \approx y_\kappa, \end{array}\right] \text{ initialisation} \\ u_{n+1} = \Phi(u_{n+1}, u_n, u_{n-1}, \dots, u_{n-\kappa}), \quad \forall n = \kappa, \kappa+1, \dots, N-1. \end{cases}$$

Méthodes explicites VS implicites

- Une méthode est dite *explicite* si la valeur u_{n+1} peut être calculée directement à l'aide des valeurs précédentes u_j, $j \leq n$ (ou d'une partie d'entre elles).
- Une méthode est dite *implicite* si u_{n+1} n'est défini que par une relation implicite faisant intervenir la fonction φ évaluée en t_{n+1} et u_{n+1}.

Méthodes à un pas VS méthodes multi-pas et initialisation(s)

- Une méthode numérique pour l'approximation du problème de Cauchy est dite *à un pas* si pour tout $n \in \mathbb{N}$, u_{n+1} ne dépend que de u_n et éventuellement de lui-même (donc $\kappa = 0$).
- Autrement, on dit que le schéma est une méthode *multi-pas* (ou à pas multiples ou *multistep*). Noter que, pour calculer successivement $u_{\kappa+1}$, $u_{\kappa+2}$, ..., il est nécessaire d'initialiser d'abord $u_0, u_1, \dots, u_\kappa$. Cette initialisation doit être effectuée par des approximations appropriées qui ne dégradent pas l'ordre de la méthode.

Définition d'un schéma multi-pas linéaire à q pas
Soit $p \geq 0$ un entier. Un schéma multi-pas linéaire à $q = p + 1$ pas pour l'approximation du problème de Cauchy consiste à définir une suite de valeurs $\{u_n\}_{n=0}^N$ par la relation de récurrence

$$\begin{cases} \left.\begin{array}{l} u_0 = y_0, \text{ connu} \\ u_1 \approx y_1, \\ \vdots \\ u_p \approx y_p, \end{array}\right] \text{ initialisation} \\ u_{n+1} = \sum_{j=0}^{p} a_j u_{n-j} + h \sum_{j=0}^{p} b_j \varphi(t_{n-j}, u_{n-j}) + h b_{-1} \varphi(t_{n+1}, u_{n+1}), \qquad n = p, p+1, \dots, N-1 \end{cases}$$

où les $\{a_j\}_{j=0}^{p}$ et $\{b_j\}_{j=-1}^{p}$ sont $2p+1$ coefficients donnés déterminant le schéma.

Le terme *linéaire* traduit le fait que le schéma est construit comme une combinaison linéaire des valeurs u_{n-j} et des valeurs de la fonction φ évaluée en ces même points.

Notation : la fonction φ est évaluée seulement en t_{n-j} pour $j = -1, 0, 1, \dots, p$, qui sont des points de la grille de discrétisation, ainsi sans ambiguïté, on peut écrire φ_{n-j} au lieu de $\varphi(t_{n-j}, y_{n-j})$.

Notons qu'on peut réécrire la formule précédente en regroupant les termes dans d'autres manières équivalentes en privilégiant les termes en u_{n-j} pour $j \in \{0, 1, \dots, p\}$ et en u_{n+1} pour bien visualiser si le schéma est explicite ou implicite ou en privilégiant la distinction entre les termes en u_{n-j} et ceux en φ_{n-j}. Par exemple, on peut écrire

$$u_{n+1} = \sum_{j=0}^{p} \left(a_j u_{n-j} + h b_j \varphi_{n-j} \right) + h b_{-1} \varphi_{n+1}$$

ou encore

$$\left(u_{n+1} - \sum_{j=0}^{p} a_j u_{n-j} \right) = h \sum_{j=-1}^{p} b_j \varphi_{n-j}.$$

- **Méthodes explicites et méthodes implicites :** si $b_{-1} = 0$, le schéma est *explicite* ; sinon, il est dit *implicite*.
- **Méthodes à un pas et méthodes multi-pas :** si $p = 0$, le schéma est à un pas ($q = p + 1 = 1$) ; sinon, il est multi-pas.

Cas particuliers :

- Schémas de type Adams : $a_0 = 1$ et $a_j = 0$ pour $j \neq 0$, i.e. de la forme $(u_{n+1} - u_n) = h \sum_{j=-1}^{p} b_j \varphi_{n-j}$.
- Schéma de Nyström et de Milne-Simpson : $a_1 = 1$ et $a_j = 0$ pour $j \neq 1$, i.e. de la forme $(u_{n+1} - u_{n-1}) = h \sum_{j=-1}^{p} b_j \varphi_{n-j}$.
- Schémas de type BDF (*Backward Differentiation Formula*) : $b_{-1} = 1$ et $b_j = 0$ pour $j \neq -1$, i.e. de la forme $\left(u_{n+1} - \sum_{j=0}^{p} a_j u_{n-j} \right) = h b_{-1} \varphi_{n+1}$.

Exemples
Les schémas d'Euler explicite (EE), d'Euler implicite (EI) et de Crank-Nicolson (CN) sont des schémas linéaires à $q = 1$ pas ($p = 0$) car de la forme :

$$\begin{cases} u_0 = y_0, \\ u_{n+1} = a_0 u_n + h b_0 \varphi_n + h b_{-1} \varphi_{n+1}, \quad \forall n = 0, 1, \dots, N-1. \end{cases}$$

En effet, on a les relations suivantes :

	Récurrence	Type	a_0	b_0	b_{-1}
EE	$u_{n+1} = u_n + h\varphi_n$	Explicite	1	1	0
EI	$u_{n+1} = u_n + h\varphi_{n+1}$	Implicite	1	0	1
CN	$u_{n+1} = u_n + \frac{h}{2}(\varphi_n + \varphi_{n+1})$	Implicite	1	$\frac{1}{2}$	$\frac{1}{2}$

NB :

- le schéma d'Euler modifié (EM) et le schéma (RK1_M) ne sont pas des schémas multi-pas linéaires car ils nécéssitent l'évaluation de φ en un point qui n'appartient pas à la grille de discrétisation, à savoir en

$t_n + \frac{h}{2}$. On verra que ces types de schémas sont des schémas de la famille Runge-Kutta (on les verras au dernier chapitre) ;
- le schéma de Heun non plus n'est pas un schéma multi-pas linéaire car il évalue φ non pas en (t_{n+1}, u_{n+1}) mais en $(t_{n+1}, \tilde{u}_{n+1})$ où $\tilde{u}_{n+1} \approx u_{n+1}$. On verra que ce type de schéma peut être considéré soit comme un schéma de type predicteur-corrector (dernière section de ce chapitre) ou encore comme un schéma de la famille Runge-Kutta.

i Plan d'étude

Dans la suite de ce chapitre nous allons nous intéresser à la **construction** de schémas linéaires à q pas bien connus en littérature et concrètement utilisés pour l'approximation numérique de problèmes de Cauchy.
En particulier, nous allons voir **2 stratégies** pour construire ces schémas :

1. **intégration sur un intervalle de temps d'une approximation polynomiale de la fonction** φ :
   - méthodes de Adams : intégration entre $[t_n, t_{n+1}]$ d'une approximation de la fonction φ par un polynôme de degré p (méthodes de type Adams-Bashforth et Adams-Moulton) ; dans ces schémas $a_j = 0$ pour $j \neq 0$;
   - méthodes de Nyström et de Milne-Simpson : intégration entre $[t_{n-1}, t_{n+1}]$ d'une approximation de la fonction φ par un polynôme de degré p ; dans ces schémas $a_j = 0$ pour $j \neq 1$;
2. **dérivation d'une approximation polynomiale de la fonction** y **puis évaluation en** t_{n+1} (méthodes de type BDF) ; dans ces schémas $b_j = 0$ pour $j \neq -1$.

On verra ensuite une approche dite "predictor-corrector" qui combine deux schémas multi-pas linéaires (l'un implicite, l'autre explicite) permettant la construction de schémas d'ordre plus élevé tout en gardant le caractère explicite.
Après avoir introduit les notions de convergence, consistance et stabilité(s) (dans les prochains chapitres), nous pourrons enfin étudier l'ordre de convergence et la stabilité de ces schémas. Dans les TD, nous étudierons des schémas multi-pas linéaires à q pas plus généraux, sans nous intéresser à leur construction, mais seulement à leur analyse de convergence et stabilité.

7.1. Sympy pour la construction de schémas multi-pas linéaires

Dans la suite, nous allons construire explicitement plusieurs schémas multi-pas linéaires à q pas pour l'approximation du problème de Cauchy. Cette construction est basée sur l'interpolation de fonction φ par un polynôme et son intégration sur un intervalle donné. Pour vérifier nos calculs d'interpolation puis intégration, nous allons utiliser le package de calcul formel `SymPy`. Nous allons donc l'importer et définir les variables symboliques nécessaires.

```
from IPython.display import display, Math, Markdown, Latex
import sympy as sp


h = sp.Symbol('h') # pas de discrétisation
t = sp.Symbol('t') # temps continu utilisé pour les polynômes d'interpolation
f = sp.Symbol(r'\tilde f(t)') # polynôme d'interpolation
t_n = sp.Symbol('t_n') # temps discret
t_np1 = t_n + h
t_nm1 = t_n - h
t_nm2 = t_n - 2 * h
t_nm3 = t_n - 3 * h
t_nm4 = t_n - 4 * h
t_nm5 = t_n - 5 * h
t_nm6 = t_n - 6 * h
t_nm7 = t_n - 7 * h
#
y_np1=sp.Symbol('y_{n+1}')
y_n  =sp.Symbol('y_{n}')
y_nm1=sp.Symbol('y_{n-1}')
y_nm2=sp.Symbol('y_{n-2}')
```

```
y_nm3=sp.Symbol('y_{n-3}')
y_nm4=sp.Symbol('y_{n-4}')
y_nm5=sp.Symbol('y_{n-5}')
y_nm6=sp.Symbol('y_{n-6}')
y_nm7=sp.Symbol('y_{n-7}')
y_np1=sp.Symbol('y_{n+1}')
# Pour Adams et N-MS
phi_np1=sp.Symbol(r'\varphi_{n+1}')
phi_n  =sp.Symbol(r'\varphi_{n}')
phi_nm1=sp.Symbol(r'\varphi_{n-1}')
phi_nm2=sp.Symbol(r'\varphi_{n-2}')
phi_nm3=sp.Symbol(r'\varphi_{n-3}')
phi_nm4=sp.Symbol(r'\varphi_{n-4}')
phi_nm5=sp.Symbol(r'\varphi_{n-5}')
phi_nm6=sp.Symbol(r'\varphi_{n-6}')
phi_nm7=sp.Symbol(r'\varphi_{n-7}')
# Pour BDF
p    = sp.Function('p')
u_np1=sp.Symbol('u_{n+1}')
u_n  =sp.Symbol('u_{n}')
u_nm1=sp.Symbol('u_{n-1}')
u_nm2=sp.Symbol('u_{n-2}')
u_nm3=sp.Symbol('u_{n-3}')
u_nm4=sp.Symbol('u_{n-4}')
u_nm5=sp.Symbol('u_{n-5}')
u_nm6=sp.Symbol('u_{n-6}')
u_nm7=sp.Symbol('u_{n-7}')
```

7.2. Schémas de Adams : intégration sur $[t_n, t_{n+1}]$

Idée : intégrer l'EDO $y'(t) = \varphi(t, y(t))$ entre t_n et t_{n+1} puis approcher l'intégrale par une formule de quadrature

$$y_{n+1} - y_n = \int_{t_n}^{t_{n+1}} \varphi(t, y(t))dt \approx \int_{t_n}^{t_{n+1}} \tilde{f}(t)\mathrm{d}t \quad \text{où } \tilde{f}(t) \approx \varphi(t, y(t)).$$

Les **schémas d'Adams** choisissent $\tilde{f}(t)$ comme un polynôme **interpolant φ en des points donnés qui peuvent être à l'extérieur de l'intervalle d'intégration** $[t_n; t_{n+1}]$.

Schémas de Adams à q pas
Soit $p \geq 0$ un entier. Un schéma de Adams à $q = p + 1$ pas (= étapes) est un cas particulier de schéma multi-pas linéaire

$$\begin{cases} \left.\begin{array}{l} u_0 = y_0, \text{ connu} \\ u_1 \approx y_1, \\ \vdots \\ u_p \approx y_p, \end{array}\right] \text{ initialisation} \\ u_{n+1} = u_n + h \sum_{j=-1}^{p} b_j \varphi_{n-j} \qquad n = p, p+1, \ldots, N-1 \end{cases}$$

Autrement dit, un schéma de Adams est un schéma multi-pas linéaire à q pas où
- les coefficients a_j sont tous nuls pour $j \geq 1$,
- les $\{b_j\}_{j=-1}^{p}$ sont $p + 1$ coefficients donnés déterminant le schéma.

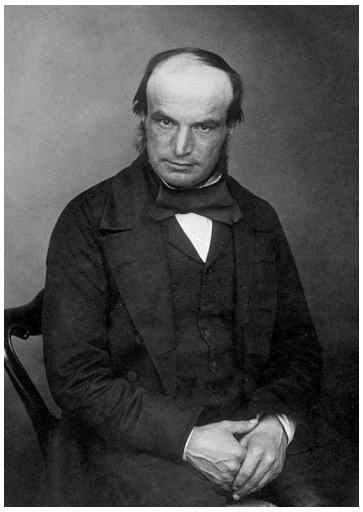


FIGURE 7.1. – John Couch Adams (1819 - 1892)

Sous-classes de schémas de Adams
Ils se divisent en deux familles :
- schémas **explicites** dits d'**Adams-Bashforth** (AB-q) où $b_{-1} = 0$;
- schémas **implicites** dits d'**Adams-Moulton** (AM-q) où $b_{-1} \neq 0$ (avec un abus de notation car AM-0 est toujours un schéma à un pas).

Caveat : la notation AM-k peut varier selon les auteurs. Par exemple, dans les livres de *Quarteroni et al.*, le terme k indique l'ordre de la méthode et non le nombre de pas. Les deux notations coïncident pour les méthodes AB, mais diffèrent pour les méthodes AM.

Remarques

1. Polynômes d'interpolation :
   - dans les schémas **explicites** (AB-q), $\tilde{f}$ est le polynôme $p \in \mathbb{R}^p[t]$ interpolant φ en les $p+1$ points $\{t_n, t_{n-1}, \dots, t_{n-q+1}\}$ avec $q = p+1 \geq 1$;
   - dans les schémas **implicites** (AM-q), $\tilde{f}$ est le polynôme $p \in \mathbb{R}^{p+1}[t]$ interpolant φ en les $p+2$ points $\{t_{n+1}, t_n, t_{n-1}, \dots, t_{n-q+1}\}$ avec $q = p+1 \geq 0$.
2. Noter que, pour calculer successivement u_{p+1}, u_{p+2}, …, il est nécessaire d'initialiser d'abord $u_0, u_1, \dots, u_p$. Cette initialisation doit être effectuée par des approximations appropriées qui ne dégradent pas l'ordre de la méthode.
3. Nous verrons ultérieurement que :
   - la condition de zéro-stabilité est toujours satisfaite ;
   - la condition de A-stabilité devient de plus en plus contraignante avec l'augmentation du nombre de pas. En particulier, une méthode explicitement A-stable est d'ordre au plus 2 (seconde barrière de Dahlquist).

7.2.1. Construction des schémas de Adams-Bashford (AB-q)

7.2.1.1. Construction du schéma de Adams-Bashforth à 1 pas (AB-1)

- $q = 1$ pas donc $p = q - 1 = 0$
- **Points d'interpolation** : (un seul point)
$$\{(t_n, \varphi_n)\}$$
- **Polynôme d'interpolation** : (degré 0)
$$\tilde{f}(t) = \varphi_n$$
- **Approximation** : (coïncide avec la formule du rectangle à gauche)
$$y_{n+1} - y_n \approx \int_{t_n}^{t_{n+1}} \tilde{f}(t)\mathrm{d}t = h\varphi_n$$



FIGURE 7.2. – Francis Bashforth (1819 - 1912)

```
pol = sp.interpolate([(t_n, phi_n)], t)
display(sp.Eq(f, pol))
integrale = sp.integrate(pol, (t, t_n, t_np1)).simplify()
display(sp.Eq(u_np1, u_n + integrale))
```

$\tilde{f}(t) = \varphi_n$

$u_{n+1} = \varphi_n h + u_n$

Ainsi, on obtient le schéma :

$$(\mathrm{AB}_1) \quad \begin{cases} u_0 = y_0, \\ u_{n+1} = u_n + h\varphi_n, \quad n = 0, 1, \dots, N-1. \end{cases}$$

La méthode AB_1 coïncide donc avec la méthode d'Euler explicite.

7.2.1.2. Construction du schéma de Adam-Bashforth à 2 pas (AB-2)

- $q = 2$ pas donc $p = q - 1 = 1$
- **Points d'interpolation** : (deux points)

$$\{(t_n, \varphi_n); (t_{n-1}, \varphi_{n-1})\}$$

- **Polynôme d'interpolation** : (degré 1)

$$\tilde{f}(t) = \varphi_n \frac{t - t_{n-1}}{t_n - t_{n-1}} + \varphi_{n-1} \frac{t - t_n}{t_{n-1} - t_n} = \varphi_n \frac{t - t_{n-1}}{h} + \varphi_{n-1} \frac{t - t_n}{-h}$$

- **Approximation** :

$$y_{n+1} - y_n \approx \int_{t_n}^{t_{n+1}} \tilde{f}(t)\mathrm{d}t = \frac{h}{2}\left(3\varphi_n - \varphi_{n-1}\right)$$

```
pol = sp.interpolate([(t_n, phi_n),
                      (t_nm1, phi_nm1)],
                      t)
display(sp.Eq(f, pol.factor(t)))
integrale = sp.integrate(pol, (t, t_n, t_np1)).simplify()
display(sp.Eq(u_np1, u_n + integrale))
```

$$\tilde{f}(t) = -\frac{-\varphi_{n-1}t_n - \varphi_n h + \varphi_n t_n + t\,(\varphi_{n-1} - \varphi_n)}{h}$$

$$u_{n+1} = \frac{h\,(-\varphi_{n-1} + 3\varphi_n)}{2} + u_n$$

Ainsi, on obtient le schéma :

$$(\mathrm{AB}_2) \quad \begin{cases} u_0 = y(t_0) = y_0, \\ u_1 = u_0 + h\varphi_0 \approx y(t_1), \\ u_{n+1} = u_n + \frac{h}{2}(3\varphi_n - \varphi_{n-1}), \quad n = 1, 2, \dots, N-1. \end{cases}$$

où u_1 est une approximation de $y(t_1)$ obtenue en utilisant ici une prédiction AB_1.

7.2.1.3. Construction du schéma de Adams-Bashforth à 3 pas (AB-3)

- **Points d'interpolation** : (trois points)

$$\{(t_n, \varphi_n); (t_{n-1}, \varphi_{n-1}); (t_{n-2}, \varphi_{n-2})\}$$

- **Polynôme d'interpolation** : (degré 2)

$$\begin{aligned} \tilde{f}(t) &= \varphi_n \frac{(t-t_{n-1})(t-t_{n-2})}{(t_n-t_{n-1})(t_n-t_{n-2})} + \varphi_{n-1}\frac{(t-t_n)(t-t_{n-2})}{(t_{n-1}-t_n)(t_{n-1}-t_{n-2})} + \varphi_{n-2}\frac{(t-t_n)(t-t_{n-1})}{(t_{n-2}-t_n)(t_{n-2}-t_{n-1})} \\ &= \frac{\varphi_{n-2}}{2h^2}(t-t_{n-1})(t-t_n) - \frac{\varphi_{n-1}}{h^2}(t-t_{n-2})(t-t_n) + \frac{\varphi_n}{2h^2}(t-t_{n-2})(t-t_{n-1}) \end{aligned}$$

- **Approximation** :

$$y_{n+1} - y_n \approx \int_{t_n}^{t_{n+1}} \tilde{f}(t)\mathrm{d}t = \frac{h}{12}\,(23\varphi_n - 16\varphi_{n-1} + 5\varphi_{n-2})$$

```
pol = sp.interpolate([(t_n, phi_n),
                      (t_nm1, phi_nm1),
                      (t_nm2, phi_nm2)],
                      t)
display(sp.Eq(f, pol.factor(t)))
integrale = sp.integrate(pol, (t, t_n, t_np1)).simplify()
display(sp.Eq(u_np1, u_n + integrale))
```

$$\tilde{f}(t) = -\frac{-4\varphi_{n-1}ht_n + 2\varphi_{n-1}t_n^2 + \varphi_{n-2}ht_n - \varphi_{n-2}t_n^2 - 2\varphi_n h^2 + 3\varphi_n ht_n - \varphi_n t_n^2 + t^2\cdot(2\varphi_{n-1} - \varphi_{n-2} - \varphi_n) + t\,(4\varphi_{n-1}h - 4\varphi_{n-1}t_n$$

$$u_{n+1} = \frac{h\,(-16\varphi_{n-1} + 5\varphi_{n-2} + 23\varphi_n)}{12} + u_n$$

On obtient le schéma

$$(\mathrm{AB}_3) \quad \begin{cases} u_0 = y(t_0) = y_0, \\ u_1 = u_0 + h\varphi_0 \approx y(t_1), \\ u_2 = u_1 + \frac{h}{2}(3\varphi_1 - \varphi_0) \approx y(t_2), \\ u_{n+1} = u_n + \frac{h}{12}(23\varphi_n - 16\varphi_{n-1} + 5\varphi_{n-2}), \quad n = 2, 3, \dots, N-1. \end{cases}$$

où u_1 est une approximation de $y(t_1)$ obtenue en utilisant une prédiction AB_1 et u_2 est une approximation de $y(t_2)$ obtenue en utilisant la méthode AB_2.

7.2.1.4. Construction du schéma de Adams-Bashforth à 4 pas (AB-4)

- **Points d'interpolation** : (quatre points)

$$\{(t_n, \varphi_n); (t_{n-1}, \varphi_{n-1}); (t_{n-2}, \varphi_{n-2}); (t_{n-3}, \varphi_{n-3})\}$$

- **Polynôme d'interpolation** : (degré 3)

$$\tilde{f}(t) = \sum_{i=n-3}^{n} \left(\varphi_i \prod_{\substack{j=n-2,\dots,n \\ j\neq i}} \frac{t - t_j}{t_i - t_j} \right)$$

- **Approximation** :

$$y_{n+1} - y_n \approx \int_{t_n}^{t_{n+1}} \tilde{f}(t)\mathrm{d}t = \frac{h}{24}\left(55\varphi_n - 59\varphi_{n-1} + 37\varphi_{n-2} - 9\varphi_{n-3}\right)$$

```
pol = sp.interpolate([(t_n, phi_n),
                      (t_nm1, phi_nm1),
                      (t_nm2, phi_nm2),
                      (t_nm3, phi_nm3)],
                     t)
# display(sp.Eq(f, pol.factor(t)))
integrale = sp.integrate(pol, (t, t_n, t_np1)).simplify()
display(sp.Eq(u_np1, u_n + integrale))
```

$$u_{n+1} = \frac{h\left(-59\varphi_{n-1} + 37\varphi_{n-2} - 9\varphi_{n-3} + 55\varphi_n\right)}{24} + u_n$$

On obtient le schéma

$$(\mathrm{AB}_4) \quad \begin{cases} u_0 = y(t_0) = y_0, \\ u_1 = u_0 + h\varphi_0 \approx y(t_1), \\ u_2 = u_1 + \dfrac{h}{2}(3\varphi_1 - \varphi_0) \approx y(t_2), \\ u_3 = u_2 + \dfrac{h}{12}(23\varphi_2 - 16\varphi_1 + 5\varphi_0) \approx y(t_3), \\ u_{n+1} = u_n + \dfrac{h}{24}(55\varphi_n - 59\varphi_{n-1} + 37\varphi_{n-2} - 9\varphi_{n-3}), \quad n = 3, 4, \dots, N-1. \end{cases}$$

où u_1 est une approximation de $y(t_1)$ obtenue en utilisant une prédiction AB_1, u_2 est une approximation de $y(t_2)$ obtenue en utilisant la méthode AB_2 etc.

7.2.2. Construction des schémas de Adams-Moulton (AM-q)

7.2.2.1. Construction du schéma de Adams-Moulton à 0 pas (AM-0) [abus de notation car AM-0 est toujours un schéma à un pas]

- **Points d'interpolation** : (un seul point)

$$\{(t_{n+1}, \varphi_{n+1})\}$$

- **Polynôme d'interpolation** : (degré 0)

$$\tilde{f}(t) = \varphi_{n+1}$$

- **Approximation** : (coïncide avec la formule du rectangle à droite)

$$y_{n+1} - y_n \approx \int_{t_n}^{t_{n+1}} \tilde{f}(t)\mathrm{d}t = h\varphi_{n+1}$$



FIGURE 7.3. – Forest Ray Moulton (1872 - 1952)

```
pol = sp.interpolate([(t_np1, phi_np1)], t)
display(sp.Eq(f, pol.factor(t)))
integrale = sp.integrate(pol, (t, t_n, t_np1)).simplify()
display(sp.Eq(u_np1, u_n + integrale))
```

$\tilde{f}(t) = \varphi_{n+1}$

$u_{n+1} = \varphi_{n+1} h + u_n$

Ainsi, on obtient le schéma

$$(\mathrm{AM}_0) \quad \begin{cases} u_0 = y(t_0) = y_0, \\ u_{n+1} = u_n + h\varphi_{n+1}, \quad n = 0, 1, \dots, N-1. \end{cases}$$

La méthode AM_0 coïncide avec la méthode d'Euler implicite.

7.2.2.2. Construction du schéma de Adam-Multon à 1 pas (AM-1)

- **Points d'interpolation** : (deux points)

$$\{(t_{n+1}, \varphi_{n+1}), (t_n, \varphi_n)\}$$

- **Polynôme d'interpolation** : (degré 1)

$$\tilde{f}(t) = \varphi_{n+1} \frac{t - t_n}{h} + \varphi_n \frac{t - t_{n+1}}{-h}$$

- **Approximation** : (coïncide avec la formule du trapèze)

$$y_{n+1} - y_n \approx \int_{t_n}^{t_{n+1}} \tilde{f}(t) \mathrm{d}t = \frac{h}{2}(\varphi_{n+1} + \varphi_n)$$

```
pol = sp.interpolate([(t_np1, phi_np1),
                      (t_n, phi_n)],
                      t)
display(sp.Eq(f, pol.factor(t)))
integrale = sp.integrate(pol, (t, t_n, t_np1)).simplify()
display(sp.Eq(u_np1, u_n + integrale))
```

$$\tilde{f}(t) = \frac{-\varphi_{n+1}t_n + \varphi_n h + \varphi_n t_n + t\left(\varphi_{n+1} - \varphi_n\right)}{h}$$

$$u_{n+1} = \frac{h\left(\varphi_{n+1} + \varphi_n\right)}{2} + u_n$$

Ainsi, on obtient le schéma

$$(\mathrm{AM}_1) \quad \begin{cases} u_0 = y(t_0) = y_0, \\ u_{n+1} = u_n + \dfrac{h}{2}(\varphi_n + \varphi_{n+1}), \quad n = 0, 1, \ldots, N-1. \end{cases}$$

La méthode AM_1 coïncide avec la méthode de Crank-Nicolson.

7.2.2.3. Construction du schéma de Adam-Multon à 2 pas (AM-2)

- **Points d'interpolation** : (trois points)

$$\{(t_{n+1}, \varphi_{n+1}), (t_n, \varphi_n), (t_{n-1}, \varphi_{n-1})\}$$

- **Polynôme d'interpolation** : (degré 2)

$$\tilde{f}(t) = \varphi_{n+1}\frac{(t-t_n)(t-t_{n-1})}{(t_{n+1}-t_n)(t_{n+1}-t_{n-1})} + \varphi_n\frac{(t-t_{n+1})(t-t_{n-1})}{(t_n-t_{n+1})(t_n-t_{n-1})} + \varphi_{n-1}\frac{(t-t_{n+1})(t-t_n)}{(t_{n-1}-t_{n+1})(t_{n-1}-t_n)}$$

- **Approximation** :

$$y_{n+1} - y_n \approx \int_{t_n}^{t_{n+1}} \tilde{f}(t)\mathrm{d}t = \frac{h}{12}\left(5\varphi_{n+1} + 8\varphi_n - \varphi_{n-1}\right)$$

```
pol = sp.interpolate([(t_np1, phi_np1),
                      (t_n, phi_n),
                      (t_nm1, phi_nm1)],
                      t)
display(sp.Eq(f, pol.factor(t)))
integrale = sp.integrate(pol, (t, t_n, t_np1)).simplify()
display(sp.Eq(u_np1, u_n + integrale))
```

$$\tilde{f}(t) = \frac{-\varphi_{n+1}ht_n + \varphi_{n+1}t_n^2 + \varphi_{n-1}ht_n + \varphi_{n-1}t_n^2 + 2\varphi_n h^2 - 2\varphi_n t_n^2 + t^2\left(\varphi_{n+1} + \varphi_{n-1} - 2\varphi_n\right) + t\left(\varphi_{n+1}h - 2\varphi_{n+1}t_n - \varphi_{n-1}\right.}{2h^2}$$

$$u_{n+1} = \frac{h\left(5\varphi_{n+1} - \varphi_{n-1} + 8\varphi_n\right)}{12} + u_n$$

On obtient le schéma

$$(\mathrm{AM}_2) \quad \begin{cases} u_0 = y(t_0) = y_0, \\ u_1 = u_0 + \dfrac{h}{2}(\varphi_1 + \varphi_0), \\ u_{n+1} = u_n + \dfrac{h}{12}(5\varphi_{n+1} + 8\varphi_n - \varphi_{n-1}), \quad n = 1, 2, \ldots, N-1. \end{cases}$$

où u_1 est une approximation de $y(t_1)$ obtenue en utilisant une prédiction AM_1.

7.2.3. Automatisation de la construction des schémas de Adam

Nous allons maintenant automatiser le calcul des coefficients des schémas de Adam en utilisant la bibliothèque `SymPy`.

```
# ================================================================================
# Pour l'affichage, on veut que les termes soient ordonnés du plus récent au plus ancien
# ================================================================================

# Ordre désiré (du plus récent au plus ancien)
phi_order_AM = [phi_np1,phi_n, phi_nm1, phi_nm2, phi_nm3, phi_nm4, phi_nm5, phi_nm6, phi_nm7]
phi_order_AB = phi_order_AM[1:]

def ordered_expr(u_n, expr, phi_vars, h):
    expr_over_h = sp.expand(sp.simplify(expr / h))

    coeffs, phis_used = [], []
    for phi in phi_vars:
        coeff = expr_over_h.coeff(phi)
        if coeff != 0:
            coeffs.append(coeff)
            phis_used.append(phi)

    if not coeffs:
        return sp.Add(u_n, expr, evaluate=False)

    denoms = [sp.fraction(c)[1] for c in coeffs]
    lcm_denom = denoms[0]
    for d in denoms[1:]:
        lcm_denom = sp.lcm(lcm_denom, d)

    terms = []
    for coeff, phi in zip(coeffs, phis_used):
        num_coeff = sp.simplify(coeff * lcm_denom)
        terms.append(num_coeff * phi)

    inner = sp.Add(*terms, evaluate=False)
    factor = h / lcm_denom
    return sp.Add(u_n, sp.Mul(factor, inner, evaluate=False), evaluate=False)  # u_n en premier




# ================================================================================
# Calcul des formules d'Adam-Bashforth et Adam-Moulton à partir de l'interpolation polynomiale
# ================================================================================

display(Markdown(f"#### Adam-Bashforth (explictes)"))
Points = [(t_n, phi_n), (t_nm1, phi_nm1), (t_nm2, phi_nm2), (t_nm3, phi_nm3),
          (t_nm4, phi_nm4), (t_nm5, phi_nm5), (t_nm6, phi_nm6), (t_nm7, phi_nm7)]

for q in range(1, len(Points) + 1):
    pol = sp.interpolate(Points[:q], t)
    AB = (sp.integrate(pol, (t, t_n, t_np1)).simplify())
    display(Markdown("- "+f"**AB-{q}: explicite, à {q} pas**"))
    # display(sp.Eq(u_np1, u_n + AB))
    schema = sp.Eq(u_np1, ordered_expr(u_n, AB, phi_order_AB, h), evaluate=False)
    display(Markdown(f"$${sp.latex(schema)}$$"))
    #display(Markdown("---"))


display(Markdown(f"#### Adam-Moulton (implicites)"))
Points = [(t_np1, phi_np1), (t_n, phi_n), (t_nm1, phi_nm1), (t_nm2, phi_nm2),
          (t_nm3, phi_nm3), (t_nm4, phi_nm4), (t_nm5, phi_nm5), (t_nm6, phi_nm6), (t_nm7, phi_nm7)]
for q in range(len(Points)):
    pol = sp.interpolate(Points[0:q + 1], t)
    AM = (sp.integrate(pol, (t, t_n, t_np1)).simplify())
    display(Markdown("- "+f"**AM-{q}: implicite, à {q} pas**"))
```

```
schema = sp.Eq(u_np1, ordered_expr(u_n, AM, phi_order_AM, h), evaluate=False)
display(Markdown(f"$${sp.latex(schema)}$$"))
#(Markdown("---"))
```

7.2.3.1. Adam-Bashforth (explictes)

- **AB-1 : explicite, à 1 pas**

$$u_{n+1} = \varphi_n h + u_n$$

- **AB-2 : explicite, à 2 pas**

$$u_{n+1} = \frac{h}{2}\left(-\varphi_{n-1} + 3\varphi_n\right) + u_n$$

- **AB-3 : explicite, à 3 pas**

$$u_{n+1} = \frac{h}{12}\left(-16\varphi_{n-1} + 5\varphi_{n-2} + 23\varphi_n\right) + u_n$$

- **AB-4 : explicite, à 4 pas**

$$u_{n+1} = \frac{h}{24}\left(-59\varphi_{n-1} + 37\varphi_{n-2} - 9\varphi_{n-3} + 55\varphi_n\right) + u_n$$

- **AB-5 : explicite, à 5 pas**

$$u_{n+1} = \frac{h}{720}\left(-2774\varphi_{n-1} + 2616\varphi_{n-2} - 1274\varphi_{n-3} + 251\varphi_{n-4} + 1901\varphi_n\right) + u_n$$

- **AB-6 : explicite, à 6 pas**

$$u_{n+1} = \frac{h}{1440}\left(-7923\varphi_{n-1} + 9982\varphi_{n-2} - 7298\varphi_{n-3} + 2877\varphi_{n-4} - 475\varphi_{n-5} + 4277\varphi_n\right) + u_n$$

- **AB-7 : explicite, à 7 pas**

$$u_{n+1} = \frac{h}{60480}\left(-447288\varphi_{n-1} + 705549\varphi_{n-2} - 688256\varphi_{n-3} + 407139\varphi_{n-4} - 134472\varphi_{n-5} + 19087\varphi_{n-6} + 198721\varphi_n\right) + u_n$$

- **AB-8 : explicite, à 8 pas**

$$u_{n+1} = \frac{h}{120960}\left(-1152169\varphi_{n-1} + 2183877\varphi_{n-2} - 2664477\varphi_{n-3} + 2102243\varphi_{n-4} - 1041723\varphi_{n-5} + 295767\varphi_{n-6} + 434241\varphi_n\right)$$

7.2.3.2. Adam-Moulton (implicites)

- **AM-0 : implicite, à 0 pas**

$$u_{n+1} = \varphi_{n+1} h + u_n$$

- **AM-1 : implicite, à 1 pas**

$$u_{n+1} = \frac{h}{2}\left(\varphi_{n+1} + \varphi_n\right) + u_n$$

- **AM-2 : implicite, à 2 pas**

$$u_{n+1} = \frac{h}{12} \cdot (5\varphi_{n+1} - \varphi_{n-1} + 8\varphi_n) + u_n$$

- **AM-3 : implicite, à 3 pas**

$$u_{n+1} = \frac{h}{24} \cdot (9\varphi_{n+1} - 5\varphi_{n-1} + \varphi_{n-2} + 19\varphi_n) + u_n$$

- **AM-4 : implicite, à 4 pas**

$$u_{n+1} = \frac{h}{720} \cdot (251\varphi_{n+1} - 264\varphi_{n-1} + 106\varphi_{n-2} - 19\varphi_{n-3} + 646\varphi_n) + u_n$$

- **AM-5 : implicite, à 5 pas**

$$u_{n+1} = \frac{h}{1440} \cdot (475\varphi_{n+1} - 798\varphi_{n-1} + 482\varphi_{n-2} - 173\varphi_{n-3} + 27\varphi_{n-4} + 1427\varphi_n) + u_n$$

- **AM-6 : implicite, à 6 pas**

$$u_{n+1} = \frac{h}{60480} \cdot (19087\varphi_{n+1} - 46461\varphi_{n-1} + 37504\varphi_{n-2} - 20211\varphi_{n-3} + 6312\varphi_{n-4} - 863\varphi_{n-5} + 65112\varphi_n) + u_n$$

- **AM-7 : implicite, à 7 pas**

$$u_{n+1} = \frac{h}{120960} \cdot (36799\varphi_{n+1} - 121797\varphi_{n-1} + 123133\varphi_{n-2} - 88547\varphi_{n-3} + 41499\varphi_{n-4} - 11351\varphi_{n-5} + 1375\varphi_{n-6} + 139849\varphi_n) + u_n$$

- **AM-8 : implicite, à 8 pas**

$$u_{n+1} = \frac{h}{3628800} \cdot (1070017\varphi_{n+1} - 4604594\varphi_{n-1} + 5595358\varphi_{n-2} - 5033120\varphi_{n-3} + 3146338\varphi_{n-4} - 1291214\varphi_{n-5} + 312874\varphi_{n-6} - 33$$

7.3. Schémas de Nyström et de Milne-Simpson : intégration sur $[t_{n-1}, t_{n+1}]$

Les méthodes d'Adam peuvent être facilement généralisées en intégrant l'EDO $y'(t) = \varphi(t, y(t))$ entre t_{n-r} et t_{n+1} avec $r \geq 1$.

Idée : choisir $r = 1$; intégrer l'EDO $y'(t) = \varphi(t, y(t))$ entre t_{n-1} et t_{n+1} puis approcher l'intégrale par une formule de quadrature

$$y_{n+1} - y_{n-1} = \int_{t_{n-1}}^{t_{n+1}} \varphi(t, y(t))dt \approx \int_{t_{n-1}}^{t_{n+1}} \tilde{f}(t)\mathrm{d}t \quad \text{où } \tilde{f}(t) \approx \varphi(t, y(t)).$$

À nouveau, $\tilde{f}(t)$ sera un polynôme **interpolant φ en des points donnés qui peuvent être à l'extérieur de l'intervalle d'intégration** $[t_{n-1}; t_{n+1}]$.

Schémas de Nyström et de Milne-Simpson à $q \geq 2$ pas
Soit $p \geq 1$ un entier. Un schéma de Nystrom ou de Milne-Simpson à $q = p + 1 \geq 2$ pas (= étapes) est un cas particulier de schéma multi-pas linéaire

$$\begin{cases} \left.\begin{array}{l} u_0 = y_0, \text{ connu} \\ u_1 \approx y_1, \text{ tj présent} \\ \vdots \\ u_p \approx y_p, \end{array}\right] \text{ initialisation} \\ u_{n+1} = u_{n-1} + h \sum_{j=-1}^{p} b_j \varphi_{n-j} \qquad n = p, p+1, \dots, N-1 \end{cases}$$

Autrement dit, un schéma multi-pas linéaire à q pas où
- les coefficients a_j sont tous nuls pour $j \neq 1$,
- les $\{b_j\}_{j=-1}^{p}$ sont $p+1$ coefficients donnés déterminant le schéma,
- u_1 doit toujours être calculé par une méthode de prédiction.

Sous-classes de schémas
- Schémas **explicites** dits de **Nyström** (N-q) où $b_{-1} = 0$;
- schémas **implicites** dits de **Milne-Simpson** (MS-q) où $b_{-1} \neq 0$.

Remarques

1. Polynômes d'interpolation :
   - dans les schémas **explicites** (N-q), $\tilde{f}$ est le polynôme $p \in \mathbb{R}^p[t]$ interpolant φ en les $p+1$ points $\{t_n, t_{n-1}, \dots, t_{n-q+1}\}$ avec $q = p+1 \geq 1$;
   - dans les schémas **implicites** (MS-q), $\tilde{f}$ est le polynôme $p \in \mathbb{R}^{p+1}[t]$ interpolant φ en les $p+2$ points $\{t_{n+1}, t_n, t_{n-1}, \dots, t_{n-q+1}\}$ avec $q = p+1 \geq 0$.
2. Quel que soit le schéma, il est nécessaire d'initialiser u_0 et u_1. Cette initialisation doit être effectuée par des approximations appropriées qui ne dégradent pas l'ordre de la méthode.

7.3.1. Construction des schémas de Nyström (N-q)

7.3.1.1. Construction du schéma de Nyström à 1 pas (N-1) [abus de notation, car N-1 est en réalité un schéma à 2 pas]

- **Points d'interpolation** : (un seul point)
$$\{(t_n, \varphi_n)\}$$
- **Polynôme** : (degré 0)
$$\tilde{f}(t) = \varphi_n$$
- **Approximation** :
$$y_{n+1} - y_{n-1} \approx \int_{t_{n-1}}^{t_{n+1}} \tilde{f}(t)\,\mathrm{d}t = 2h\varphi_n$$

```
pol = sp.interpolate([(t_n, phi_n)], t).simplify()
display(sp.Eq(f, pol))
integrale = sp.integrate(pol, (t, t_nm1, t_np1)).simplify()
display(sp.Eq(u_np1, u_nm1 + integrale))
```

$\tilde{f}(t) = \varphi_n$

$u_{n+1} = 2\varphi_n h + u_{n-1}$

On obtient le schéma

$$(\mathrm{N}_1) \quad \begin{cases} u_0 = y(t_0) = y_0, \\ u_1 = u_0 + h\varphi_0 \approx y(t_1) \\ u_{n+1} = u_{n-1} + 2h\varphi_n \qquad n = 1, 2, \dots N-1 \end{cases}$$

où u_1 est une approximation de $y(t_1)$ obtenue en utilisant une prédiction d'Euler explicite.

Remarque : on l'appelle N-1 mais c'est un abus de notation car elle est à 2 pas (elle dépend de u_{n-1} directement et non pas à travers φ).

7.3.1.2. Construction du schéma de Nyström à 2 pas (N-2)

- **Points d'interpolation** : (deux points)

$$\{(t_n, \varphi_n); (t_{n-1}, \varphi_{n-1})\}$$

- **Polynôme** : (degré 1)

$$\tilde{f}(t) = \varphi_n \frac{t - t_{n-1}}{h} + \varphi_{n-1} \frac{t - t_n}{-h}$$

- **Approximation** : (coïncide avec la formule d'approximation trouvée précédemment)

$$y_{n+1} - y_{n-1} \approx \int_{t_{n-1}}^{t_{n+1}} \tilde{f}(t)\mathrm{d}t = 2h\varphi_n$$

```
pol = sp.interpolate([(t_n, phi_n),
                      (t_nm1, phi_nm1)],
                     t)
display(sp.Eq(f, pol.factor(t)))
integrale = sp.integrate(pol, (t, t_nm1, t_np1)).simplify()
display(sp.Eq(u_np1, u_nm1 + integrale))
```

$$\tilde{f}(t) = -\frac{-\varphi_{n-1}t_n - \varphi_n h + \varphi_n t_n + t\left(\varphi_{n-1} - \varphi_n\right)}{h}$$

$u_{n+1} = 2\varphi_n h + u_{n-1}$

On obtient le schéma

$$(\mathrm{N}_2) \equiv (\mathrm{N}_1) \quad \begin{cases} u_0 = y(t_0) = y_0, \\ u_1 = u_0 + h\varphi_0 \approx y(t_1) \\ u_{n+1} = u_{n-1} + 2h\varphi_n \qquad n = 1, 2, \ldots N-1 \end{cases}$$

où u_1 est une approximation de $y(t_1)$ obtenue en utilisant une prédiction d'Euler explicite.

Cette méthode coïncide avec la **méthode du point milieu** (aussi appelée **Saute-mouton** ou **Leapfrog**).

7.3.1.3. Construction du schéma de Nyström à 3 pas (N-3)

- **Points d'interpolation** : (trois points)

$$\{(t_n, \varphi_n); (t_{n-1}, \varphi_{n-1}); (t_{n-2}, \varphi_{n-2})\}$$

- **Polynôme** : (degré 2)

$$\begin{aligned} \tilde{f}(t) &= \varphi_n \frac{(t - t_{n-1})(t - t_{n-2})}{(t_n - t_{n-1})(t_n - t_{n-2})} + \varphi_{n-1} \frac{(t - t_n)(t - t_{n-2})}{(t_{n-1} - t_n)(t_{n-1} - t_{n-2})} + \varphi_{n-2} \frac{(t - t_n)(t - t_{n-1})}{(t_{n-2} - t_n)(t_{n-2} - t_{n-1})} \\ &= \frac{\varphi_{n-2}}{2h^2}(t - t_{n-1})(t - t_n) - \frac{\varphi_{n-1}}{h^2}(t - t_{n-2})(t - t_n) + \frac{\varphi_n}{2h^2}(t - t_{n-2})(t - t_{n-1}) \end{aligned}$$

- **Approximation** :

$$y_{n+1} - y_{n-1} \approx \int_{t_{n-1}}^{t_{n+1}} \tilde{f}(t)\,\mathrm{d}t = \frac{h}{3}\left(7\varphi_n - 2\varphi_{n-1} + \varphi_{n-2}\right)$$

```
pol = sp.interpolate([(t_n, phi_n), (t_nm1, phi_nm1), (t_nm2, phi_nm2)],t).simplify()
display(sp.Eq(f, pol))
integrale = sp.integrate(pol, (t, t_nm1, t_np1)).simplify()
display(sp.Eq(u_np1, u_nm1 + integrale))
```

$$\tilde{f}(t) = \frac{-\varphi_{n-1}t^2 + 2\varphi_{n-1}tt_n - \varphi_{n-1}t_n^2 + \frac{\varphi_{n-2}t^2}{2} - \varphi_{n-2}tt_n + \frac{\varphi_{n-2}t_n^2}{2} + \varphi_n h^2 + \frac{\varphi_n t^2}{2} - \varphi_n tt_n + \frac{\varphi_n t_n^2}{2} + \frac{h(-4\varphi_{n-1}t+4\varphi_{n-1}t_n+\varphi_{n-2}}{2}}{h^2}$$

$$u_{n+1} = \frac{h\left(-2\varphi_{n-1} + \varphi_{n-2} + 7\varphi_n\right)}{3} + u_{n-1}$$

On obtient le schéma

$$(\mathrm{N}_3) \quad \begin{cases} u_0 = y(t_0) = y_0, \\ u_1 = u_0 + h\varphi_0 \approx y(t_1) \\ u_2 = u_0 + 2h\varphi_1 \approx y(t_2) \\ u_{n+1} = u_{n-1} + \frac{h}{3}\left(7\varphi_n - 2\varphi_{n-1} + \varphi_{n-2}\right) \quad n = 2, 3, \dots N-1 \end{cases}$$

où u_1 est une approximation de $y(t_1)$ obtenue en utilisant une prédiction d'Euler explicite et u_2 est une approximation de $y(t_2)$ obtenue en utilisant la méthode N-2.

7.3.2. Construction des schémas de Milne-Simpson (MS-q)

7.3.2.1. Construction du schéma de Milne-Simpson à 0 pas (MS-0) [abus de notation car MS-0 est toujours un schéma à 2 pas]

- **Points d'interpolation** : (un seul point)
$$\{(t_{n+1}, \varphi_{n+1})\}$$
- **Polynôme** : (degré 0)
$$\tilde{f}(t) = \varphi_{n+1}$$
- **Approximation** : (coïncide avec la formule du rectangle à droite)
$$y_{n+1} - y_{n-1} \approx \int_{t_{n-1}}^{t_{n+1}} \tilde{f}(t)\,\mathrm{d}t = 2h\varphi_{n+1}$$

```
pol = sp.interpolate([(t_np1, phi_np1)], t).simplify()
display(sp.Eq(f, pol))
integrale = sp.integrate(pol, (t, t_nm1, t_np1)).simplify()
display(sp.Eq(u_np1, u_nm1 + integrale))
```

$\tilde{f}(t) = \varphi_{n+1}$

$u_{n+1} = 2\varphi_{n+1}h + u_{n-1}$

On obtient le schéma

$$(\mathrm{MS}_0) \quad \begin{cases} u_0 = y(t_0) = y_0, \\ u_1 = u_0 + h\varphi_1 \approx y(t_1) \\ u_{n+1} = u_{n-1} + 2h\varphi_{n+1} \quad n = 1, 2, \dots, N-1 \end{cases}$$

où u_1 est une approximation de $y(t_1)$ obtenue en utilisant une prédiction d'Euler implicite.

Remarque : on l'appelle MS-0 mais c'est une schéma à 2 pas, car il dépend de u_{n-1}.

7.3.2.2. Construction du schéma de Milne-Simpson à 1 pas (MS-1) [abus de notation car MS-1 est toujours un schéma à 2 pas]

- **Points d'interpolation** : (deux points)

$$\{(t_{n+1}, \varphi_{n+1}), (t_n, \varphi_n)\}$$

- **Polynôme** : (degré 1)

$$\tilde{f}(t) = \varphi_{n+1}\frac{t - t_n}{h} + \varphi_n\frac{t - t_{n+1}}{-h}$$

- **Approximation** : (coïncide avec les formules N-1 et N-2)

$$y_{n+1} - y_{n-1} \approx \int_{t_{n-1}}^{t_{n+1}} \tilde{f}(t)\,\mathrm{d}t = 2h\varphi(t_n, y_n)$$

```
pol = sp.interpolate([(t_np1, phi_np1),
                      (t_n, phi_n)],
                      t).simplify()
display(sp.Eq(f, pol.factor(t)))
integrale = sp.integrate(pol, (t, t_nm1, t_np1)).simplify()
display(sp.Eq(u_np1, u_nm1 + integrale))
```

$$\tilde{f}(t) = \frac{-\varphi_{n+1}t_n + \varphi_n h + \varphi_n t_n + t\left(\varphi_{n+1} - \varphi_n\right)}{h}$$

$u_{n+1} = 2\varphi_n h + u_{n-1}$

On obtient le schéma

$$(\mathrm{MS}_1) \equiv (\mathrm{N}_1) \equiv (\mathrm{N}_2) \quad \begin{cases} u_0 = y(t_0) = y_0, \\ u_1 = u_0 + h\varphi_1 \approx y(t_1) \\ u_{n+1} = u_{n-1} + 2h\varphi_n \qquad n = 1, 2, \ldots N-1 \end{cases}$$

où u_1 est une approximation de $y(t_1)$ obtenue en utilisant une prédiction d'Euler implicite.

Nota bene : le schéma est en realité explicite.

Remarque : on l'appelle MS-1 mais c'est un schéma à 2 pas, car il dépend de u_{n-1}.

7.3.2.3. Construction du schéma de Milne-Simpson à 2 pas (MS-2)

- **Points d'interpolation** : (trois points)

$$\{(t_{n+1}, \varphi_{n+1}), (t_n, \varphi_n), (t_{n-1}, \varphi_{n-1})\}$$

- **Polynôme** : (degré 2)

$$\tilde{f}(t) = \varphi_{n+1}\frac{(t - t_n)(t - t_{n-1})}{(t_{n+1} - t_n)(t_{n+1} - t_{n-1})} + \varphi_n\frac{(t - t_{n+1})(t - t_{n-1})}{(t_n - t_{n+1})(t_n - t_{n-1})} + \varphi_{n-1}\frac{(t - t_{n+1})(t - t_n)}{(t_{n-1} - t_{n+1})(t_{n+1} - t_n)}$$

- **Approximation** : (coïncide avec la formule de Simpson sur un intervalle de largeur $2h$)

$$y_{n+1} - y_{n-1} \approx \int_{t_{n-1}}^{t_{n+1}} \tilde{f}(t)\,\mathrm{d}t = \frac{h}{3}\left(\varphi_{n+1} + 4\varphi_n + \varphi_{n-1}\right)$$

```
pol = sp.interpolate([(t_np1, phi_np1),
                      (t_n, phi_n),
                      (t_nm1, phi_nm1)],
                      t).simplify()
display(sp.Eq(f, pol.factor(t)))
integrale = sp.integrate(pol, (t, t_nm1, t_np1)).simplify()
display(sp.Eq(u_np1, u_nm1 + integrale))
```

$$\tilde{f}(t) = \frac{-\varphi_{n+1}ht_n + \varphi_{n+1}t_n^2 + \varphi_{n-1}ht_n + \varphi_{n-1}t_n^2 + 2\varphi_n h^2 - 2\varphi_n t_n^2 + t^2\left(\varphi_{n+1} + \varphi_{n-1} - 2\varphi_n\right) + t\left(\varphi_{n+1}h - 2\varphi_{n+1}t_n - \varphi_{n-1}\right.}{2h^2}$$

$$u_{n+1} = \frac{h\left(\varphi_{n+1} + \varphi_{n-1} + 4\varphi_n\right)}{3} + u_{n-1}$$

On obtient le schéma

$$(\mathrm{MS}_2) \equiv (\mathrm{MS}_3) \quad \begin{cases} u_0 = y(t_0) = y_0, \\ u_1 = u_0 + \dfrac{h}{2}(\varphi_1 + \varphi_0)) \\ u_{n+1} = u_{n-1} + \dfrac{h}{3}(\varphi_{n+1} + 4\varphi_n + \varphi_{n-1}) \quad n = 1, 2, \ldots N-1 \end{cases}$$

où u_1 est une approximation de $y(t_1)$ obtenue en utilisant une prédiction AM_1 (= Crank-Nicolson).

7.3.3. Automatisation de la construction des schémas de Nyström et de Milne-Simpson

```
display(Markdown(f"#### Nyström"))
Points = [(t_n, phi_n), (t_nm1, phi_nm1), (t_nm2, phi_nm2), (t_nm3, phi_nm3),
          (t_nm4, phi_nm4), (t_nm5, phi_nm5)]
for q in range(1, len(Points) + 1):
    pol = sp.interpolate(Points[:q], t)
    N = (sp.integrate(pol, (t, t_nm1, t_np1)).simplify())
    display(Markdown("- "+f"**N-{q}: explicite, à {q} pas**"))
    display(sp.Eq(u_np1, u_nm1 + N))
    #display(Markdown("---"))


display(Markdown(f"#### Milne-Simpson"))
Points = [(t_np1, phi_np1), (t_n, phi_n), (t_nm1, phi_nm1), (t_nm2, phi_nm2),
          (t_nm3, phi_nm3), (t_nm4, phi_nm4), (t_nm5, phi_nm5)]
for q in range(len(Points)):
    pol = sp.interpolate(Points[:q + 1], t)
    MS = (sp.integrate(pol, (t, t_nm1, t_np1)).simplify())
    display(Markdown("- "+f"**MS-{q}: implicite, à {q} pas**"))
    display(sp.Eq(u_np1, u_nm1 + MS))
    ##display(Markdown("---"))
```

7.3.3.1. Nyström

- **N-1 : explicite, à 1 pas**

$$u_{n+1} = 2\varphi_n h + u_{n-1}$$

- **N-2 : explicite, à 2 pas**

$$u_{n+1} = 2\varphi_n h + u_{n-1}$$

- **N-3 : explicite, à 3 pas**

$$u_{n+1} = \frac{h\left(-2\varphi_{n-1} + \varphi_{n-2} + 7\varphi_n\right)}{3} + u_{n-1}$$

- **N-4 : explicite, à 4 pas**

$$u_{n+1} = \frac{h\left(-5\varphi_{n-1} + 4\varphi_{n-2} - \varphi_{n-3} + 8\varphi_n\right)}{3} + u_{n-1}$$

- **N-5 : explicite, à 5 pas**

$$u_{n+1} = \frac{h\left(-266\varphi_{n-1} + 294\varphi_{n-2} - 146\varphi_{n-3} + 29\varphi_{n-4} + 269\varphi_n\right)}{90} + u_{n-1}$$



FIGURE 7.4. – Joseph Oakland Hirschfelder (1911 - 1990)

- **N-6 : explicite, à 6 pas**

$$u_{n+1} = \frac{h\left(-406\varphi_{n-1} + 574\varphi_{n-2} - 426\varphi_{n-3} + 169\varphi_{n-4} - 28\varphi_{n-5} + 297\varphi_n\right)}{90} + u_{n-1}$$

7.3.3.2. Milne-Simpson

- **MS-0 : implicite, à 0 pas**

$$u_{n+1} = 2\varphi_{n+1}h + u_{n-1}$$

- **MS-1 : implicite, à 1 pas**

$$u_{n+1} = 2\varphi_n h + u_{n-1}$$

- **MS-2 : implicite, à 2 pas**

$$u_{n+1} = \frac{h\left(\varphi_{n+1} + \varphi_{n-1} + 4\varphi_n\right)}{3} + u_{n-1}$$

- **MS-3 : implicite, à 3 pas**

$$u_{n+1} = \frac{h\left(\varphi_{n+1} + \varphi_{n-1} + 4\varphi_n\right)}{3} + u_{n-1}$$

- **MS-4 : implicite, à 4 pas**

$$u_{n+1} = \frac{h\left(29\varphi_{n+1} + 24\varphi_{n-1} + 4\varphi_{n-2} - \varphi_{n-3} + 124\varphi_n\right)}{90} + u_{n-1}$$

- **MS-5 : implicite, à 5 pas**

$$u_{n+1} = \frac{h\left(28\varphi_{n+1} + 14\varphi_{n-1} + 14\varphi_{n-2} - 6\varphi_{n-3} + \varphi_{n-4} + 129\varphi_n\right)}{90} + u_{n-1}$$

- **MS-6 : implicite, à 6 pas**

$$u_{n+1} = \frac{h\left(1139\varphi_{n+1} + 33\varphi_{n-1} + 1328\varphi_{n-2} - 807\varphi_{n-3} + 264\varphi_{n-4} - 37\varphi_{n-5} + 5640\varphi_n\right)}{3780} + u_{n-1}$$

7.4. Schémas BDF (Backward Differentiation Formulae)

Les méthodes des « différences en arrière » ont été introduites par Charles (« Chuck ») Francis Curtiss (1921 - 2007) et Joseph Oakland Hirschfelder (1911 - 1990) et jouent un rôle particulier dans la résolution des problèmes raides.

Soit $\tilde{y}$ le polynôme qui interpole la fonction inconnue $t \mapsto y(t)$ (supposée de classe $\mathcal{C}^1$) en les points

$$\{t_{n+1}, t_n, \dots, t_{n-q+1}\} \text{ avec } q \geq 1.$$

Idée : remplacer l'évaluation en t_{n+1} de l'EDO $\varphi(t, y) = y'(t)$ par l'évaluation de l'EDO approchée $\varphi(t, y) = \tilde{y}'(t)$, ce qui donne la relation

$$\varphi(t_{n+1}, y_{n+1}) \approx \tilde{y}'(t_{n+1}).$$

Les **schémas BDF** choisissent $\tilde{y}(t)$ comme un polynôme **interpolant** y **en des points donnés de la grille** puis **en dérivant le polynôme interpolant** pour obtenir une approximation de la dérivée de y :

$$u_{n+1} \text{ solution de l'équation } \varphi(t_{n+1}, u_{n+1}) = \tilde{y}'(t_{n+1}) \quad \text{où } \tilde{y}(t) \text{ est un polynôme interpolant } y(t)$$

Schémas BDF-q à q pas
Soit $p \geq 0$ un entier. Un schéma BDF à $q = p + 1$ pas (= étapes) est un cas particulier de schéma multi-pas linéaire

$$\begin{cases} \left.\begin{array}{l} u_0 = y_0, \text{ connu} \\ u_1 \approx y_1, \\ \vdots \\ u_p \approx y_p, \end{array}\right] \text{ initialisation} \\ u_{n+1} = \sum_{j=0}^{n} a_j u_{n-j} + h b_{-1} \varphi_{n+1}, \qquad n = p, p+1, \dots, N-1. \end{cases}$$

Remarques :
- il s'agit de schémas implicites, car u_{n+1} apparaît toujours dans le terme $\varphi_{n+1} = \varphi(t_{n+1}, u_{n+1})$;
- il n'y a qu'une unique évaluation de φ par pas de temps, en t_{n+1}.

7.4.1. Construction des schémas BDF (BDF-q)

7.4.1.1. Construction du schéma BDF-1

- **Points d'interpolation** : (deux points)
$$\{(t_{n+1}, y_{n+1}), (t_n, y_n)\}$$
- **Polynôme** : (degré 1)
$$\tilde{y}(t) = \frac{y_{n+1} - y_n}{h}(t - t_n) + y_n$$
- **Dérivée** : (degré 0)
$$\tilde{y}'(t) = \frac{y_{n+1} - y_n}{h}$$
- **Approximation** :
$$\varphi(t_{n+1}, y_{n+1}) \approx \tilde{y}'(t_{n+1}) = \frac{y_{n+1} - y_n}{h}$$
- **Itération** :
$$u_{n+1} \text{ solution de l'équation } \varphi_{n+1} = \frac{u_{n+1} - u_n}{h}$$

```
p = sp.Function(r'\tilde y') # polynôme d'interpolation

pol = sp.interpolate([(t_np1, u_np1), (t_n, u_n)], t).factor()
display(sp.Eq(p(t),pol))

dp = sp.diff(pol, t).subs(t, t_np1)
display(sp.Eq(sp.diff(p(t),t).subs(t, t_np1) , dp))

rhs = sp.solve(dp - phi_np1, u_np1)[0]
display(sp.Eq(u_np1,rhs))
```

$$\tilde{y}(t) = \frac{hu_n + tu_{n+1} - tu_n - t_n u_{n+1} + t_n u_n}{h}$$

$$\frac{d}{dt}\tilde{y}(t)\Big|_{t=h+t_n} = \frac{u_{n+1} - u_n}{h}$$

$$u_{n+1} = \varphi_{n+1} h + u_n$$

Ainsi, on obtient le schéma

$$(\mathrm{BDF}_1) \quad \begin{cases} u_0 = y(t_0) = y_0, \\ u_{n+1} = u_n + h\varphi_{n+1} \quad n = 0, 1, \dots N-1. \end{cases}$$

La méthode BDF_1 coïncide avec la méthode d'Euler implicite.

7.4.1.2. Construction du schéma BDF-2

- **Points d'interpolation** : (trois points)

$$\{(t_{n+1}, y_{n+1}), (t_n, y_n), (t_{n-1}, y_{n-1})\}$$

- **Polynôme** : (degré 2)

$$\tilde{y}(t) = y_{n+1}\frac{(t-t_n)(t-t_{n-1})}{(t_{n+1}-t_n)(t_{n+1}-t_{n-1})} + y_n\frac{(t-t_{n+1})(t-t_{n-1})}{(t_n-t_{n+1})(t_n-t_{n-1})} + y_{n-1}\frac{(t-t_{n+1})(t-t_n)}{(t_{n-1}-t_{n+1})(t_{n-1}-t_n)}$$

$$= \frac{(t-t_n)(t-t_{n-1})}{2h^2}y_{n+1} + \frac{(t-t_{n+1})(t-t_{n-1})}{-h^2}y_n + \frac{(t-t_{n+1})(t-t_n)}{-2h^2}y_{n-1}$$

- **Dérivée** : (degré 1)

$$\tilde{y}'(t) = \frac{(t-t_n)+(t-t_{n-1})}{2h^2}y_{n+1} + \frac{(t-t_{n+1})+(t-t_{n-1})}{-h^2}y_n + \frac{(t-t_{n+1})+(t-t_n)}{-2h^2}y_{n-1}$$

- **Approximation** :

$$\varphi(t_{n+1}, y_{n+1}) \approx \tilde{y}'(t_{n+1}) = \frac{3}{2h}y_{n+1} - \frac{2}{h}y_n + \frac{1}{2h}y_{n-1}$$

- **Itération** :

$$u_{n+1} \text{ solution de l'équation } \varphi_{n+1} = \frac{3}{2h}u_{n+1} - \frac{2}{h}u_n + \frac{1}{2h}u_{n-1}$$

```
pol = sp.interpolate([(t_np1, u_np1), (t_n, u_n), (t_nm1, u_nm1)],
                     t).simplify()
display(sp.Eq(p(t),pol))

dp = sp.diff(pol, t).subs(t, t_np1)
display(sp.Eq(sp.diff(p(t),t).subs(t, t_np1),dp))

rhs = sp.solve(dp - phi_np1, u_np1)[0]
display(sp.Eq(u_np1,rhs))
```

$$\tilde{y}(t) = \frac{h^2 u_n + \frac{h(tu_{n+1} - tu_{n-1} - t_n u_{n+1} + t_n u_{n-1})}{2} + \frac{t^2 u_{n+1}}{2} + \frac{t^2 u_{n-1}}{2} - t^2 u_n - tt_n u_{n+1} - tt_n u_{n-1} + 2tt_n u_n + \frac{t_n^2 u_{n+1}}{2} + \frac{t_n^2 u_{n-1}}{2} - t_n^2 u_n}{h^2}$$

$$\frac{d}{dt}\tilde{y}(t)\Big|_{t=h+t_n} = \frac{\frac{h(u_{n+1}-u_{n-1})}{2} - t_n u_{n+1} - t_n u_{n-1} + 2t_n u_n + u_{n+1}(h+t_n) + u_{n-1}(h+t_n) - 2u_n(h+t_n)}{h^2}$$

$$u_{n+1} = \frac{2\varphi_{n+1} h}{3} - \frac{u_{n-1}}{3} + \frac{4u_n}{3}$$

On obtient le schéma

$$(\mathrm{BDF}_2) \quad \begin{cases} u_0 = y(t_0) = y_0, \\ u_1 = u_0 + h\varphi_1 \\ u_{n+1} = \frac{4}{3}u_n - \frac{1}{3}u_{n-1} + \frac{2}{3}h\varphi_{n+1} \quad n = 1, 2, 3, \dots, N-1. \end{cases}$$

7.4.2. Automatisation de la construction des schémas BDF

```
Points = [(t_np1, u_np1), (t_n, u_n), (t_nm1, u_nm1), (t_nm2, u_nm2),
          (t_nm3, u_nm3), (t_nm4, u_nm4), (t_nm5, u_nm5)]
for q in range(1, len(Points)):
    pol = sp.interpolate(Points[0:q + 1], t)
    dp = sp.diff(pol, t).subs(t, t_np1)
    sol = sp.solve(dp - phi_np1, u_np1)
    display(Markdown(r"- "+f"**BDF-{q}: implicite, à {q} pas**"))
    display(sp.Eq(u_np1, sol[0]))
    # display(Markdown("---"))
```

- **BDF-1 : implicite, à 1 pas**

$$u_{n+1} = \varphi_{n+1} h + u_n$$

- **BDF-2 : implicite, à 2 pas**

$$u_{n+1} = \frac{2\varphi_{n+1} h}{3} - \frac{u_{n-1}}{3} + \frac{4u_n}{3}$$

- **BDF-3 : implicite, à 3 pas**

$$u_{n+1} = \frac{6\varphi_{n+1} h}{11} - \frac{9u_{n-1}}{11} + \frac{2u_{n-2}}{11} + \frac{18u_n}{11}$$

- **BDF-4 : implicite, à 4 pas**

$$u_{n+1} = \frac{12\varphi_{n+1} h}{25} - \frac{36u_{n-1}}{25} + \frac{16u_{n-2}}{25} - \frac{3u_{n-3}}{25} + \frac{48u_n}{25}$$

- **BDF-5 : implicite, à 5 pas**

$$u_{n+1} = \frac{60\varphi_{n+1} h}{137} - \frac{300u_{n-1}}{137} + \frac{200u_{n-2}}{137} - \frac{75u_{n-3}}{137} + \frac{12u_{n-4}}{137} + \frac{300u_n}{137}$$

- **BDF-6 : implicite, à 6 pas**

$$u_{n+1} = \frac{20\varphi_{n+1} h}{49} - \frac{150u_{n-1}}{49} + \frac{400u_{n-2}}{147} - \frac{75u_{n-3}}{49} + \frac{24u_{n-4}}{49} - \frac{10u_{n-5}}{147} + \frac{120u_n}{49}$$

7.5. Combiner des schéma multi-pas avec une approche *predictor-corrector*

En général, les méthodes multi-pas implicites sont, à nombre de pas équivalent, plus précises et plus stables que les méthodes explicites (nous aborderons ces propriétés dans les prochains cours). Cependant, elles sont aussi plus coûteuses, car le calcul de u_{n+1} nécessite la résolution d'une équation. En général, cette équation est non linéaire et on ne sait pas la résoudre explicitement en toute généralité, on doit alors en chercher une solution approchée, par exemple à l'aide d'une méthode itérative comme la méthode de point fixe (nous utilisons habituellement la méthode `fsolve` de `scipy`).

Une approche différente, qui permet de s'affranchir de cette étape coûteuse, est donnée par les méthodes **predictor-corrector** (PC). L'idée des méthodes PC est de diminuer le coût de chaque itération en remplaçant la résolution de l'équation non-linéaire par une prédiction obtenue en utilisant une méthode explicite, puis de n'effectuer qu'une itération (ou quelques itérations) de la méthode implicite, qui est alors utilisée comme correcteur explicite.

En somme, les méthodes **predictor-corrector permettent de calculer u_{n+1} de façon explicite à partir d'une méthode implicite**, réduisant ainsi le besoin de résoudre des équations non-linéaires à chaque étape.

Principe de la méthode predictor-corrector (PC)
On considère une méthode multi-pas implicite (de coefficients $\{a_j\}$ et $\{b_j\}$)

$$u_{n+1} = \sum_{j=0}^{p} \left(a_j u_{n-j} + h b_j \varphi_{n-j} \right) + h b_{-1} \varphi_{n+1} \qquad \text{pour } n = p, p+1, \dots, N-1.$$

On modifie la méthode implicite comme suit :
- étape de **prédiction** : on calcule $\tilde{u}_{n+1}$ une approximation de u_{n+1} obtenue par une méthode explicite, typiquement une autre méthode multi-pas linéaire (de coefficients $\{\tilde{a}_j\}$ et $\{\tilde{b}_j\}$) ;
- étape de **correction** : on "corrige" la méthode implicite en remplaçant φ_{n+1} par $\varphi(t_{n+1}, \tilde{u}_{n+1})$.

Cela donne une récurrence du type

$$\begin{cases} \tilde{u}_{n+1} = \displaystyle\sum_{j=0}^{\tilde{p}} \left(\tilde{a}_j u_{n-j} + h \tilde{b}_j \varphi_{n-j} \right) \\ u_{n+1} = \displaystyle\sum_{j=0}^{p} \left(a_j u_{n-j} + h b_j \varphi_{n-j} \right) + h b_{-1} \varphi(t_{n+1}, \tilde{u}_{n+1}) \end{cases}$$

Bien-sûr, il reste à définir les valeurs d'initialisation. Comme le schéma final est un schéma explicite, on utlisera des initialisations obtenues avec des méthodes explicites.

Exemple : schéma de Heun
Considérons les deux schémas multi-pas à 1 pas suivants :
- *predictor* : méthode d'Euler explicite (ou AB_1) $\tilde{u}_{n+1} = u_n + h\varphi_n$
- *corrector* : méthode de Crank-Nicolson (ou AM_1) $u_{n+1} = u_n + \frac{h}{2}\left(\varphi_n + \varphi(t_{n+1}, u_{n+1})\right)$

Le schéma PC associé à ces deux schémas n'est rien d'autre que la méthode de Heun

$$\text{(H)} \quad \begin{cases} u_0 = y_0 \\ \text{Pour } n = 0, 1, \dots N-1 : \\ \quad \tilde{u}_{n+1} = u_n + h\varphi_n \\ \quad u_{n+1} = u_n + \frac{h}{2}\left(\varphi(t_{n+1}, \tilde{u}_{n+1}) + \varphi_n\right) \end{cases}$$

Des méthodes de type predictor-corrector sont souvent construites en utilisant une prédiction d'Adams-Bashforth suivie d'une correction d'Adams-Moulton.

Exemple : schéma AB2-AM2
Considérons les deux schémas multi-pas à 2 pas suivants :
- *predictor* : méthode AB_2 $u_{n+1} = u_n + \frac{h}{2}\left(3\varphi_n - \varphi_{n-1}\right)$
- *corrector* : méthode AM_2 $u_{n+1} = u_n + \frac{h}{12}\left(5\varphi(t_{n+1}, u_{n+1}) + 8\varphi_n - \varphi_{n-1}\right)$

Le schéma PC associé à ces deux schémas s'écrit

$$(\mathrm{AB}_2\text{-}\mathrm{AM}_2)\quad \begin{cases} u_0 = y_0 \\ u_1 = u_0 + h\varphi_0, \\ \text{Pour } n = 1, 2, \dots N-1 : \\ \quad \tilde{u}_{n+1} = u_n + \frac{1}{2}h\left(3\varphi_n - \varphi_{n-1}\right) \\ \quad u_{n+1} = u_n + \frac{h}{12}\left(5\varphi(t_{n+1}, \tilde{u}_{n+1}) + 8\varphi_n - \varphi_{n-1}\right) \end{cases}$$

Comme c'est un schéma à 2 pas, on a besoin d'initialiser u_1. Puisqu'au final nous voulons un schéma explicite, on utiliser alors un schéma explicite pour initialiser cette quantité, à savoir le schéma AB_1 qui n'est rien d'autre que le schéma d'Euler explicite.
Astuce de programmation : on affecte les évaluations de φ_j pour les utiliser dans les deux étapes :

$$(\mathrm{AB}_2\text{-}\mathrm{AM}_2)\quad \begin{cases} u_0 = y_0 \\ u_1 = u_0 + h\varphi_0, \\ \text{Pour } n = 1, 2, \dots N-1 : \\ \quad k_0 = \varphi_n, \\ \quad k_1 = \varphi_{n-1} \\ \quad \tilde{u}_{n+1} = u_n + \frac{1}{2}h\left(3k_0 - k_1\right) \\ \quad u_{n+1} = u_n + \frac{h}{12}\left(5\varphi(t_{n+1}, \tilde{u}_{n+1}) + 8k_0 - k_1\right) \end{cases}$$

On peut combiner des schémas multi-pas de différents types, comme une prédiction d'Adam-Bashforth suivie d'une correction BDF.

Exemple : schéma AB2-BDF2
Considérons les deux schémas multi-pas à 2 pas suivants :

- *predictor* : méthode AB_2 $u_{n+1} = u_n + \frac{h}{2}\left(3\varphi_n - \varphi_{n-1}\right)$
- *corrector* : méthode BDF_2 $u_{n+1} = \frac{4}{3}u_n - \frac{1}{3}u_{n-1} + \frac{2}{3}h\varphi(t_{n+1}, u_{n+1})$

Le schéma PC associé à ces deux schémas s'écrit

$$(\mathrm{AB}_2\text{-}\mathrm{BDF}_2)\quad \begin{cases} u_0 = y_0 \\ u_1 = u_0 + h\varphi_0, \\ \text{Pour } n = 1, 2, \dots N-1 : \\ \quad \tilde{u}_{n+1} = u_n + \frac{h}{2}\left(3\varphi_n - \varphi_{n-1}\right) \\ \quad u_{n+1} = \frac{4}{3}u_n - \frac{1}{3}u_{n-1} + \frac{2}{3}h\varphi(t_{n+1}, \tilde{u}_{n+1}) \end{cases}$$

Chapitre 8.

Propriétés d'un schéma sur un intervalle borné

Quelles sont les propriétés d'un schéma numérique qui nous permettent de garantir que les solutions approchées que nous obtenons sont de « bonne qualité » ?

Nous avons déjà vu que les problèmes de Cauchy doivent être mathématiquement bien posés avant d'essayer d'en approcher la solution numériquement. Nous avons aussi vu que certains problèmes de Cauchy sont plus ou moins « difficiles » à résoudre numériquement : cela fait intervenir les notions de

- problème numériquement bien posé, *i.e.* pas trop sensible aux perturbations
- raideur, liée à la présence de plusieurs échelles temporelles dans le problème.

Dans ce chapitre, nous allons introduire les propriétés d'un schéma numérique qui nous permettent de garantir que les solutions approchées que nous obtenons sont de « bonne qualité ». Nous allons notamment introduire les notions de convergence, consistance et stabilité d'un schéma numérique, et montrer comment elles sont liées entre elles.

```
import matplotlib.pyplot as plt
import numpy as np
plt.rcParams.update({'font.size': 12})

from scipy.optimize import fsolve

import sympy as sym

from IPython.display import Markdown
```

8.1. Convergence et ordre de convergence

Définition : schéma convergent et ordre de convergence
Une méthode numérique est **convergente** si

$$|e_n| := |y_n - u_n| \xrightarrow[h\to 0]{} 0 \qquad \forall n = 0, \dots, N$$

Si $|e_n| = \mathcal{O}(h^\omega)$ pour $\omega > 0$, on dit que la convergence de la méthode est d'ordre ω.

Rappel : $f(x) = \mathcal{O}(g(x))$ lorsque $x \to \infty$ signifie que il existe $C > 0$ tel que $|f(x)| \leq C \times |g(x)|$ asymptotiquement.

Pour étudier la convergence d'une méthode numérique, on exprime l'erreur e_{n+1} au pas t_{n+1} sous la forme :

$$\begin{aligned} \boxed{e_{n+1}} &:= y_{n+1} - u_{n+1} \\ &= (y_{n+1} - u^*_{n+1}) + (u^*_{n+1} - u_{n+1}), \\ &= \boxed{h\tau_{n+1}(h)} + \boxed{u^*_{n+1} - u_{n+1}}, \qquad \boxed{\tau_{n+1}(h) := \frac{y_{n+1} - u^*_{n+1}}{h}} \end{aligned}$$

u^*_{n+1} désignant la valeur obtenue en appliquant le schéma numérique à la solution exacte au temps t_{n+1}.

- Le terme $h\tau_{n+1}(h) = y_{n+1} - u^*_{n+1}$ représente l'erreur engendrée par une seule itération du schéma (erreur locale de consistance).
- Le terme $u^*_{n+1} - u_{n+1}$ représente la propagation de t_n à t_{n+1} de l'erreur accumulée au temps précédent t_n.

Si les deux termes $(y_{n+1} - u^*_{n+1})$ et $(u^*_{n+1} - u_{n+1})$ tendent vers zéro lorsque $h \to 0$, alors la méthode est convergente.

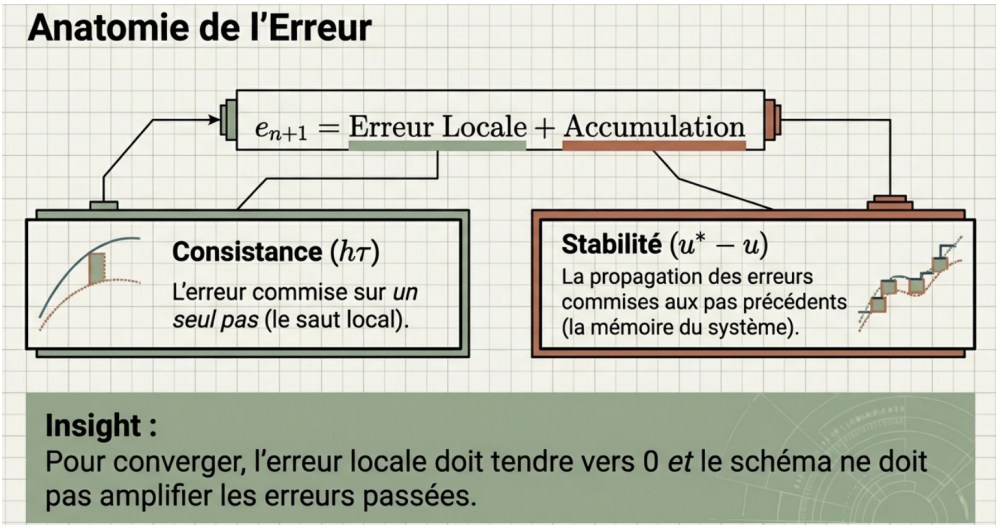


FIGURE 8.1. – Erreur générale

Exemple : convergence de la méthode d'Euler explicite
Considérons le problème de Cauchy $y'(t) = \varphi(t, y(y))$ pour $t \in [t_0; T]$ et $y(t_0) = y_0$. Supposons que la fonction φ soit lipschitzienne de constante L par rapport à sa deuxième variable. Nous allons montrer que la méthode d'Euler explicite

$$\begin{cases} u_0 = y_0, \\ u_{n+1} = u_n + h\varphi(t_n, u_n), \quad n = 0, \dots, N-1 \end{cases}$$

est convergente d'ordre 1.
Pour cela, on étudie séparément l'erreur de consistance et l'accumulation de ces erreurs. On en déduit ensuite la convergence.

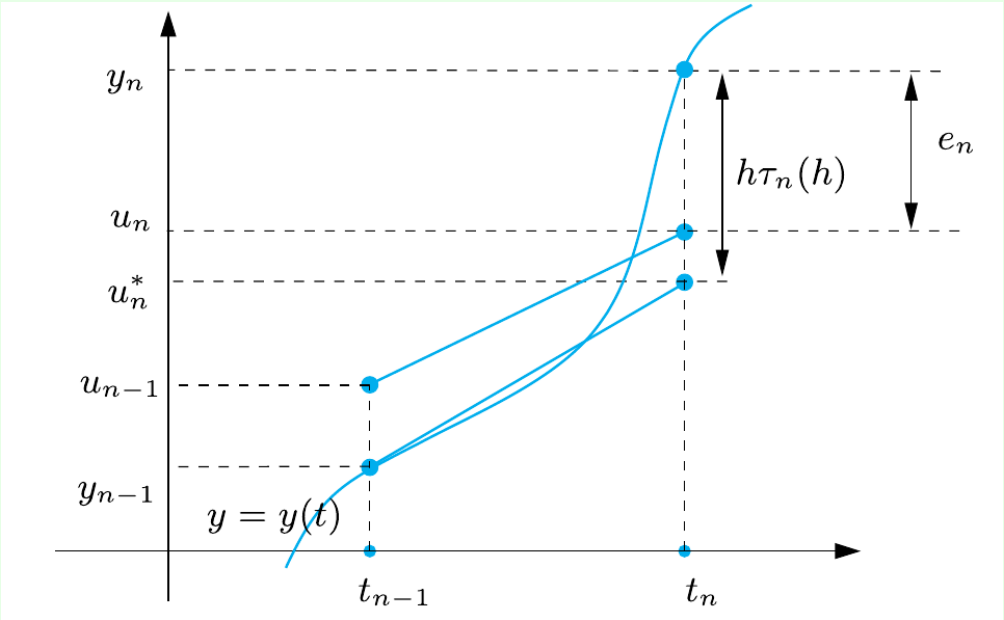


FIGURE 8.2. – Erreur du schéma d'Euler Explicite

Terme $h\tau_{n+1}(h) := y_{n+1} - u^*_{n+1}$ (erreur de consistance).
Il représente l'erreur engendrée par une seule itération de la méthode d'Euler explicite :

$$\begin{aligned} y_{n+1} - u^*_{n+1} &= y_{n+1} - \Big(y_n + h\varphi(t_n, y_n)\Big) \\ &= y(t_{n+1}) - y(t_n) - hy'(t_n). \end{aligned}$$

En supposant que la dérivée seconde de y existe et est continue, on sait qu'il existe un point η_n de l'intervalle $]t_n; t_{n+1}[$ tel que

$$y(t_{n+1}) = y(t_n) + hy'(t_n) + \frac{h^2}{2}y''(\eta_n)$$

(développement de Taylor de y au voisinage de t_n évalué en t_{n+1} avec reste de Lagrange). On a alors

$$y_{n+1} - u^*_{n+1} = y(t_{n+1}) - y(t_n) - hy'(t_n) = \frac{h^2}{2}y''(\eta_n) \leq \frac{M}{2}h^2 \xrightarrow[h\to 0]{} 0$$

où $M \equiv \max_{t\in[t_0,T]} |y''(t)|$.
L'erreur de troncature locale de la méthode d'Euler explicite est donc de la forme

$$\tau_{n+1}(h) = \frac{y_{n+1} - u^*_{n+1}}{h} = \mathcal{O}(h).$$

Terme $u^*_{n+1} - u_{n+1}$ (accumulation des erreurs).
Il représente la propagation de t_n à t_{n+1} de l'erreur accumulée au temps précédent t_n. On a

$$\begin{aligned} u^*_{n+1} - u_{n+1} &= \Big(y_n + h\varphi(t_n, y_n)\Big) - \Big(u_n + h\varphi(t_n, u_n)\Big) \\ &= e_n + h\Big(\varphi(t_n, y_n) - \varphi(t_n, u_n)\Big). \end{aligned}$$

Comme φ est lipschitzienne par rapport à sa deuxième variable, on a

$$|\varphi(t_n, y_n) - \varphi(t_n, u_n)| \leq L|y_n - u_n| = L|e_n|.$$

En factorisant on trouve alors

$$|u^*_{n+1} - u_{n+1}| \leq (1 + hL)|e_n|.$$

Convergence.
Comme $e_0 = 0$, les relations précédentes donnent

$$\begin{aligned} |e_n| &\leq |y_n - u^*_n| + |u^*_n - u_n| \\ &\leq h|\tau_n(h)| + (1 + hL)|e_{n-1}| \\ &\leq h|\tau_n(h)| + (1 + hL)\left(h|\tau_{n-1}(h)| + (1 + hL)|e_{n-2}|\right)| \\ &\leq \left(1 + (1 + hL) + \cdots + (1 + hL)^{n-1}\right) h\tau(h) \qquad \text{avec } \tau(h) = \max_{n=0,\ldots,N} |\tau_n(h)| \\ &= \left(\sum_{i=0}^{n-1}(1 + hL)^i\right) h\tau(h) \\ &= \frac{(1 + hL)^n - 1}{hL}h\tau(h) \\ &\leq \frac{(e^{hL})^n - 1}{hL}h\tau(h) \qquad \text{car } (1 + x) \leq e^x \\ &= \frac{(e^{hL})^{(t_n - t_0)/h} - 1}{L}\tau(h) \qquad \text{car } t_n - t_0 = nh \\ &= \frac{e^{L(t_n - t_0)} - 1}{L}\tau(h) \\ &\leq \frac{e^{L(t_n - t_0)} - 1}{L}\frac{M}{2}h = \mathcal{O}(h) \end{aligned}$$

On peut conclure que la méthode d'Euler explicite est convergente d'ordre 1.

Remarque : l'estimation de convergence est obtenue en supposant seulement φ lipschitzienne. On peut établir une meilleure estimation si $\partial_y\varphi$ existe et est non négative pour tout $t \in [t_0;T]$ et tout $y \in \mathbb{R}$. En effet dans ce cas

$$\begin{aligned} u_n^* - u_n &= (y_{n-1} + h\varphi(t_{n-1}, y_{n-1})) - (u_{n-1} + h\varphi(t_{n-1}, u_{n-1})) \\ &= e_{n-1} + h\left(\varphi(t_{n-1}, y_{n-1}) - \varphi(t_{n-1}, u_{n-1})\right) \\ &= e_{n-1} + h\left(e_{n-1}\partial_y\varphi(t_{n-1}, \eta_n)\right) \\ &= \left(1 + h\partial_y\varphi(t_{n-1}, \eta_n)\right) e_{n-1} \end{aligned}$$

où η_n appartient à l'intervalle dont les extrémités sont y_{n-1} et u_{n-1}. Ainsi, si

$$0 < h < \frac{2}{\max\limits_{t\in[t_0,T]} \partial_y\varphi(t, y(t))}$$

alors

$$|u_n^* - u_n| \leq |e_{n-1}|.$$

On en déduit $|e_n| \leq |y_n - u_n^*| + |e_{n-1}| \leq nh\tau(h) + |e_0|$ et donc

$$|e_n| \leq M\frac{h}{2}(t_n - t_0).$$

La restriction sur le pas de discrétisation h est une condition de stabilité, comme on le verra dans la suite.

Exercice
Calculer l'ordre de consistance du schéma d'**Euler implicite**. Montrer ensuite que la méthode est convergente.

On considère le même problème de Cauchy que précédemment : $y'(t) = \varphi(t, y(y))$ pour $t \in [t_0;T]$ et $y(t_0) = y_0$ où φ est supposée être lipschitzienne de constante L par rapport à sa deuxième variable :

$$|\varphi(t, y_1) - \varphi(t, y_2)| \leq L|y_1 - y_2| \qquad \forall t \in [t_0;T], \forall y_1, y_2 \in \mathbb{R}.$$

On suppose que la dérivée seconde de y existe et est continue.

Le schéma d'Euler implicite est défini par :

$$\begin{cases} u_0 = y_0, \\ u_{n+1} = u_n + h\varphi(t_{n+1}, u_{n+1}), \quad n = 0, \dots, N-1 \end{cases}$$

Soit $u_{n+1}^* = y_n + h\varphi(t_{n+1}, y_{n+1})$. Calculons l'erreur de troncature locale $\tau_{n+1}(h)$:

$$\begin{aligned} \tau_{n+1}(h) &\equiv \frac{y_{n+1} - u_{n+1}^*}{h} \\ &= \frac{y(t_{n+1}) - (y(t_n) + h\varphi(t_{n+1}, y(t_{n+1})))}{h} \\ &= \frac{y(t_{n+1}) - y(t_n)}{h} - \varphi(t_{n+1}, y(t_{n+1})) \end{aligned}$$

Par développement de Taylor de y au voisinage de t_{n+1} évalué en t_n, il existe un point η_n de l'intervalle $]t_n;t_{n+1}[$ tel que

$$\begin{aligned} y(t_n) &= y(t_{n+1}) - hy'(t_{n+1}) + \frac{h^2}{2}y''(\eta_n) \\ &= y(t_{n+1}) - h\varphi(t_{n+1}, y(t_{n+1})) + \frac{h^2}{2}y''(\eta_n) \end{aligned}$$

Comme $y'(t) = \varphi(t, y(t))$, on obtient alors

$$y(t_{n+1}) = y(t_n) + h\varphi(t_{n+1}, y(t_{n+1})) - \frac{h^2}{2}y''(\eta_n).$$

On en déduit que l'erreur de troncature locale de la méthode d'Euler implicite est de la forme

$$\begin{aligned}\tau_{n+1}(h) &= \frac{y(t_n) + h\varphi(t_{n+1}, y(t_{n+1})) - \frac{h^2}{2}y''(\eta_n) - y(t_n)}{h} - \varphi(t_{n+1}, y(t_{n+1})) \\ &= -\frac{h}{2}y''(\eta_n)\end{aligned}$$

On a alors, comme pour le schéma d'Euler explicite,

$$|\tau_{n+1}(h)| \leq \frac{M}{2}h \xrightarrow[h\to 0]{} 0$$

où $M \equiv \max_{t\in[t_0,T]} |y''(t)|$ et le schéma d'Euler implicite est consistant d'ordre 1.

La solution exacte vérifie donc

$$y(t_{n+1}) = y(t_n) + h\varphi(t_{n+1}, y(t_{n+1})) + h\tau_{n+1}(h).$$

En soustrayant l'expression du schéma d'Euler implicite à celle de la solution exacte, on trouve

$$\begin{aligned}e_{n+1} &= y(t_{n+1}) - u_{n+1} \\ &= \Big(y(t_n) + h\varphi(t_{n+1}, y(t_{n+1})) + h\tau_{n+1}(h)\Big) - \Big(u_n + h\varphi(t_{n+1}, u_{n+1})\Big) \\ &= e_n + h\Big(\varphi(t_{n+1}, y(t_{n+1})) - \varphi(t_{n+1}, u_{n+1})\Big) + h\tau_{n+1}(h)\end{aligned}$$

Comme φ est lipschitzienne par rapport à sa deuxième variable,

$$|\varphi(t_{n+1}, y(t_{n+1})) - \varphi(t_{n+1}, u_{n+1})| \leq L|y(t_{n+1}) - u_{n+1}| = L|e_{n+1}|.$$

En factorisant on trouve alors

$$(1 - hL)|e_{n+1}| \leq |e_n| + h|\tau_{n+1}(h)|.$$

Pour h suffisamment petit, on a $1 - hL > 0$ et donc

$$|e_{n+1}| \leq \frac{1}{1-hL}\left(|e_n| + h|\tau_{n+1}(h)|\right).$$

Comme $e_0 = 0$, les relations précédentes donnent

$$\begin{aligned}|e_n| &\leq |y(t_n) - u_n^*| + |u_n^* - u_n| \\ &\leq h|\tau_n(h)| + \frac{1}{1-hL}\left(|e_{n-1}| + h|\tau_n(h)|\right) \\ &\leq \left(1 + \frac{1}{1-hL} + \cdots + \left(\frac{1}{1-hL}\right)^{n-1}\right) h\tau(h) \qquad \text{avec } \tau(h) = \max_{n=0,\ldots,N}|\tau_n(h)| \\ &= \left(\sum_{i=0}^{n-1}\left(\frac{1}{1-hL}\right)^i\right) h\tau(h) \\ &= \frac{\left(\frac{1}{1-hL}\right)^n - 1}{\frac{1}{1-hL} - 1} h\tau(h) \\ &= \frac{(1-hL)^{-n} - 1}{\frac{hL}{1-hL}} h\tau(h) \\ &= \frac{(1-hL)^{-n} - 1}{L} h\tau(h) \qquad \text{car } t_n - t_0 = nh \\ &= \frac{e^{L(t_n - t_0)} - 1}{L} h\tau(h) \\ &= \frac{e^{L(t_n - t_0)} - 1}{L}\frac{M}{2}h = \mathcal{O}(h)\end{aligned}$$

Théorème d'équivalence de Lax-Ritchmyer : consistance + zéro-stabilité = convergence
Un schéma consistant est convergent ssi il est zéro-stable.

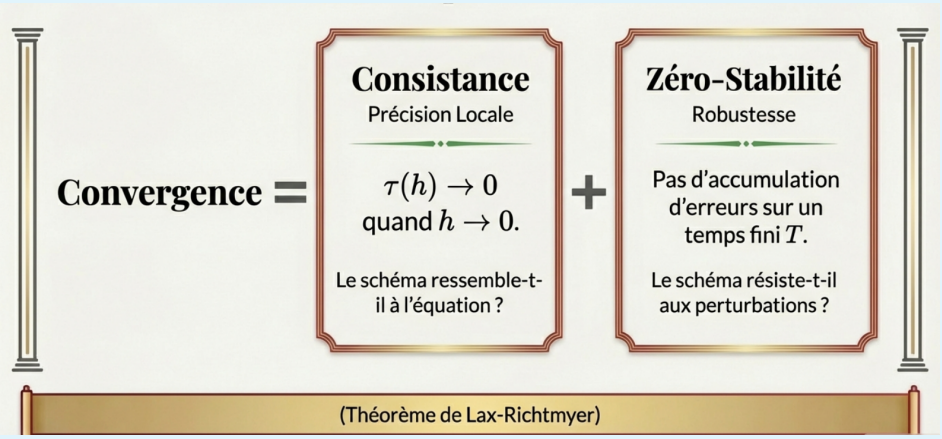


FIGURE 8.3. – Théorème d'équivalence de Lax-Ritchmyer

8.2. Consistance et ordre de consistance

Définition : erreurs de troncature
L'**erreur de troncature locale** est donnée par :

$$\tau_{n+1}(h) \equiv \frac{y_{n+1} - u^*_{n+1}}{h}.$$

Elle représente (à un facteur $1/h$ près) l'erreur introduite en insérant la solution exacte dans le schéma numérique.
L'**erreur de troncature globale** (ou simplement **erreur de troncature**) est définie comme :

$$\tau(h) = \max_{n=0,\dots,N} |\tau_n(h)|.$$

Elle correspond à la borne supérieure de l'erreur de troncature locale sur l'ensemble des pas de temps.

Définition : Consistance et ordre de consistance

- Une méthode numérique est dite **consistante** si $\lim_{h\to 0} \tau(h) = 0$.
- Elle est **consistante d'ordre** ω si l'erreur de troncature vérifie $\tau(h) = \mathcal{O}(h^\omega)$, pour un certain $\omega \geq 1$.

Remarque : La propriété de consistance est une condition nécessaire pour la convergence. En effet, sans consistance, la méthode introduirait à chaque itération une erreur qui ne tendrait pas vers zéro lorsque h diminue. L'accumulation de ces erreurs empêcherait alors l'erreur globale de s'annuler lorsque $h \to 0$.

Remarque : l'ordre de convergence coïncide en général avec l'ordre de son erreur de troncature.

8.3. Zéro-stabilité

Prenons le problème de Cauchy, *supposé mathématiquement et numériquement bien posé* :

$$\begin{cases} y'(t) = \varphi(t, y(t)), & \forall t \in I =]t_0, T[, \\ y(t_0) = y_0. \end{cases}$$

Que se passe-t-il lorsque l'on fixe le temps final T et que l'on fait tendre le pas h vers 0 ?

La **zéro-stabilité** garantit que, sur un intervalle borné, de petites perturbations des données entraînent des perturbations contrôlées de la solution numérique lorsque $h \to 0$. Elle garantit que la méthode numérique est peu sensible aux petites perturbations des données. **Si la méthode numérique n'est pas zéro-stable, les erreurs d'arrondi sur y_0 et le calcul de $\varphi(t_n, y_n)$ rendent la solution calculée inutilisable.**

Plus précisément, une méthode numérique visant à approcher le problème de Cauchy sur un intervalle fini $I = [t_0, T]$ est dite **zéro-stable** si les conditions suivantes sont satisfaites :

$$\exists h_0 > 0,\ \exists C > 0,\ \exists \varepsilon_0 > 0 \text{ tels que } \forall h \in]0, h_0],\ \forall \varepsilon \in]0, \varepsilon_0],\ \text{si } |\varrho_n| \leq \varepsilon,\ 0 \leq n \leq N,$$

alors

$$|z_n - u_n| \leq C\varepsilon,\ 0 \leq n \leq N$$

où :

- C est une constante indépendante de h,
- u_n est la solution obtenue par le schéma numérique sans perturbation,
- z_n est la solution obtenue par le même schéma appliqué au problème perturbé,
- ϱ_n représente la perturbation au n-ième pas,
- ε_0 est la perturbation maximale admissible.

Il est important de noter que ε_0 doit être suffisamment petit pour que le problème perturbé conserve une solution unique sur l'intervalle d'intégration considéré.

Dans le cas d'une méthode **consistante à un pas**, on peut démontrer que la zéro-stabilité découle du fait que φ est lipschitzienne par rapport à sa seconde variable. Dans ce contexte, la constante C dépend de $\exp((T - t_0)L)$, où L est la constante de Lipschitz. Toutefois, cette propriété n'est pas nécessairement vérifiée pour d'autres familles de méthodes numériques.

Exemple : zéro-stabilité de la méthode d'Euler explicite
Considérons u_n comme l'approximation obtenue avec la méthode d'Euler explicite pour le problème initial :

$$\begin{cases} u_0 = y_0, \\ u_{n+1} = u_n + h\varphi(t_n, u_n), \quad n = 0, \dots, N-1 \end{cases}$$

et z_n comme l'approximation obtenue avec la méthode d'Euler explicite pour le problème **perturbé** :

$$\begin{cases} z_0 = y_0 + \varrho_0, \\ z_{n+1} = z_n + h\left(\varphi(t_n, z_n) + \varrho_n\right), \quad n = 0, \dots, N-1 \end{cases}$$

Nous avons alors :

$$
\begin{aligned}
|u_n - z_n| &\leq |u_{n-1} - z_{n-1}| + h|\varphi(t_{n-1}, u_{n-1}) - \varphi(t_{n-1}, z_{n-1})| + h|\varrho_n| \\
&\leq (1 + hL)|u_{n-1} - z_{n-1}| + h\varepsilon \\
&\leq (1 + hL)^2|u_{n-2} - z_{n-2}| + (1 + (1 + hL))h\varepsilon \\
&\vdots \\
&\leq (1 + hL)^n|\varrho_0| + (1 + (1 + hL) + \cdots + (1 + hL)^{n-1})\, h\varepsilon \\
&= (1 + hL)^n|\varrho_0| + \left(\sum_{i=0}^{n-1}(1 + hL)^i\right) h\varepsilon \\
&= (1 + hL)^n|\varrho_0| + \frac{(1 + hL)^n - 1}{L}\varepsilon \\
&\leq (e^{hL})^n|\varrho_0| + \frac{(e^{hL})^n - 1}{hL}h\varepsilon \qquad \text{car } (1 + x) \leq e^x \\
&= e^{nhL}|\varrho_0| + \frac{e^{nhL} - 1}{L}\varepsilon \\
&\leq \left(e^{nhL} + \frac{e^{nhL} - 1}{L}\right)\varepsilon \\
&= \frac{(1 + L)e^{L(T-t_0)} - 1}{L}\varepsilon \qquad \text{car } t_n - t_0 = nh
\end{aligned}
$$

Ainsi, nous avons montré que la méthode d'Euler explicite est **zéro-stable**.

Chapitre 9.

Propriétés d'un schéma sur un intervalle non borné

Revenons sur la question fondamentale de la qualité d'un schéma numérique, déjà abordée dans le chapitre précédent : **quelles propriétés doit satisfaire un schéma numérique pour que les solutions approchées soient de « bonne qualité » ?**

Sur un intervalle de temps borné, nous avons établi que la **consistance** et la **zéro-stabilité** sont des conditions nécessaires et suffisantes pour assurer la convergence de la solution approchée vers la solution exacte du problème de Cauchy. Autrement dit, lorsque le pas de temps h tend vers zéro, la solution numérique converge vers la solution exacte.

Toutefois, ces deux propriétés s'avèrent insuffisantes dès lors que l'on souhaite résoudre le problème de Cauchy sur des intervalles de temps très longs, voire infinis. En effet, que se passe-t-il lorsque le pas $h > 0$ est fixé et que l'horizon T tend vers l'infini ?

Dans les deux situations ($h \to 0$ avec T fixé, ou $T \to +\infty$ avec h fixé), le nombre de nœuds tend vers l'infini. Cependant, les enjeux sont fondamentalement différents :

- dans le premier cas, on cherche à contrôler l'**erreur locale en chaque point** ;
- dans le second, c'est le **comportement asymptotique** de la solution approchée qui est en jeu.

Lorsque $T \to +\infty$ avec h fixé, le nombre N de sous-intervalles tend vers l'infini indépendamment de h, ce qui rend la notion de **zéro-stabilité** inapplicable. On cherche alors des **méthodes capables d'approcher fidèlement la solution sur des intervalles de temps arbitrairement grands**, idéalement sans imposer de restriction trop sévère sur le pas h.

```
import matplotlib.pyplot as plt
import numpy as np
import sympy as sp
from scipy.optimize import fsolve
from IPython.display import Markdown, Math
plt.rcParams.update({'font.size': 12})
```

9.1. Définition de A-stabilité

Pour analyser la stabilité d'un schéma numérique, on étudie son comportement sur un problème test. Dans les études théoriques, on considère généralement le cas où λ est un nombre complexe avec une partie réelle négative ($\Re(\lambda) < 0$). Cependant, **dans les exercices, nous allons nous concentrer sur le cas où λ est un nombre réel négatif et, pour éviter toute ambiguïté, on posera $\beta = -\lambda > 0$.**

Problème test de Dahlquist

$$\begin{cases} y'(t) = \lambda y(t), & t > 0 \\ y(0) = 1 \end{cases}$$

avec $\Re(\lambda) < 0$.
La solution exacte est $y(t) = e^{\lambda t}$ ainsi $|y(t)| \to 0$ lorsque $t \to +\infty$.

Remarque : dans le cas réel, on peut réécrire le problème de Dahlquist sous la forme $y' = -\beta y$ avec $\beta > 0$. La solution exacte est alors $y(t) = e^{-\beta t}$ et $y(t) \to 0$ lorsque $t \to +\infty$.

L'enjeu de la A-stabilité est de déterminer si le schéma numérique reproduit ce comportement de décroissance pour un pas de temps h donné.

Exemple : comportement du schéma d'Euler explicite sur le problème de Dahlquist
Le schéma d'**Euler progressif** $u_{n+1} = u_n + h\varphi(t_n, u_n)$ appliqué au toy-model ($\varphi(t, y) = -\beta y$) s'écrit

$$u_{n+1} = (1 - \beta h)u_n, \qquad n = 0, 1, 2, ...$$

soit encore

$$u_{n+1} = q(h\beta)u_n \qquad \text{avec } q(h\beta) = 1 - \beta h.$$

et par suite

$$u_n = [q(\beta h)]^n u_0, \qquad n = 0, 1, 2, ...$$

Par conséquente, $\lim_{n \to +\infty} u_n = 0$ si et seulement si

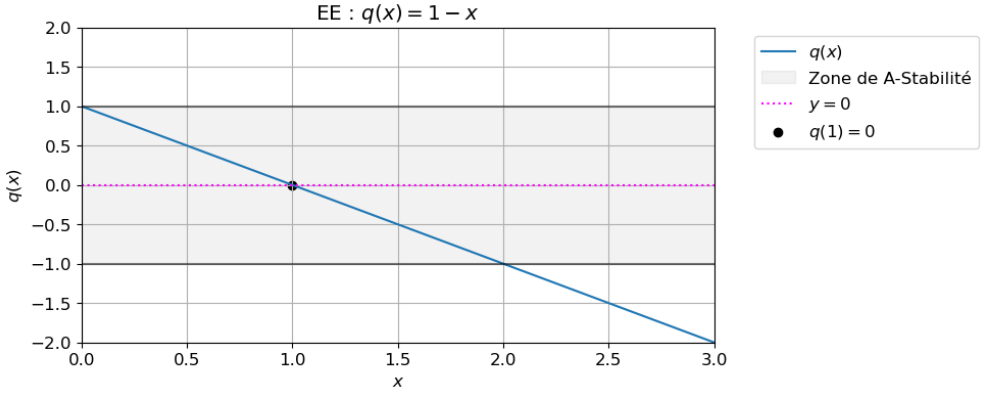
$$|q(\beta h)| < 1.$$

Si on note $x = \beta h > 0$ et on trace la fonction $x \mapsto q(x)$, on cherche les valeurs de x pour lesquelles le graphe de la fonction est compris entre les droites $y = -1$ et $y = 1$.
Le schéma d'Euler progressif a le même comportement que la solution exacte si et seulement si

$$\boxed{h < \frac{2}{\beta}.}$$

Cette condition limite le pas h d'avance en t lorsqu'on utilise le schéma d'Euler progressif.

```
x = np.linspace(0, 3, 101)
q = 1 - x
plt.figure(figsize=(8, 4))
plt.grid()
plt.plot(x, q, label='$q(x)$')
plt.ylim(-2, 2) # On zoome sur la zone intéressante [-1, 1]
plt.xlim(x.min(), x.max())
plt.axhline(y=1, color='black', linestyle='-', alpha=0.5)
plt.axhline(y=-1, color='black', linestyle='-', alpha=0.5)
plt.fill_between(x, -1, 1, color='gray', alpha=0.1, label='Zone de A-Stabilité')
plt.axhline(0, color='magenta', linestyle=':', label='$y=0$')
plt.scatter(1, 0, color='black', label='$q(1) = 0$')
plt.xlabel('$x$')
plt.ylabel('$q(x)$')
plt.title('EE : $q(x) = 1 - x$')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()
```



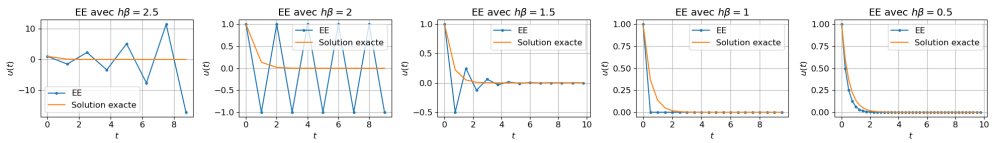
Remarques :

- Si $|1 - \beta h| \geq 1$, alors u_n ne tend pas vers 0 lorsque t tend vers l'infini. Nous dirons dans ces cas que le schéma d'Euler progressif est instable. Plus précisément :
  - si $1 - \beta h > 1$ alors u_n tend vers $+\infty$ lorsque t tend vers l'infini ;
  - si $1 - \beta h < -1$ alors u_n tend vers l'infini en alternant de signe lorsque t tend vers l'infini ;
  - si $1 - \beta h = 1$ alors u_n est constant ;
  - si $1 - \beta h = -1$ alors u_n alterne de signe.

- Pour $h < \frac{2}{\beta}$, la suite u_n tend vers 0 lorsque t tend vers l'infini. Cette convergence est *monotone* ssi $1 - \beta h > 0$, i.e. ssi $h < \frac{1}{\beta}$ (bien plus restrictif que la condition de A-stabilité).

```
def EE(tt,y0,phi):
    uu = np.zeros_like(tt)
    uu[0] = y0
    for n in range(len(tt)-1):
        h = tt[n+1] - tt[n]
        uu[n+1] = uu[n] + h*phi(tt[n], uu[n])
    return uu

BETA = 2
phi = lambda t,y: -BETA*y
y0 = 1
fig, axx = plt.subplots(nrows=1, ncols=5, figsize=(20, 3))
for ax, x in zip(axx, (2.5,2,1.5,1,0.5)):
    tt = np.arange(0, 10, x/BETA)
    uu = EE(tt, y0, phi)
    ax.grid()
    ax.plot(tt, uu, '.-', label='EE')
    ax.plot(tt, np.exp(-BETA*tt), label='Solution exacte')
    ax.set_xlabel('$t$')
    ax.set_ylabel('$u(t)$')
    ax.set_title(f'EE avec $h\\beta={(tt[1]-tt[0])*BETA:g}$ ')
    ax.legend()
plt.tight_layout()
```



Définition : A-stabilité (stabilité absolue) d'un schéma numérique
Soit $h > 0$ un pas de temps **fixé** et $t_n = nh$. On note $u_n \approx y(t_n)$ la solution numérique et y_n la solution exacte du problème de Dahlquist à l'instant t_n.
On introduit la variable de stabilité $z = h\lambda \in \mathbb{C}^-$.
Le schéma est dit

- **inconditionnellement A-stable** lorsque $u_n \to 0$ pour tout $z \in \mathbb{C}^-$, c'est-à-dire pour tout λ avec $\Re(\lambda) < 0$ et pour tout pas de temps $h > 0$ (aucune restriction sur h) ;
- **conditionnellement A-stable** lorsque $u_n \to 0$ seulement si $0 < h < h_{\max}(\lambda)$.

NB : la stabilité numérique dépend uniquement du produit z, et non de h ou λ séparément.

Remarque : dans le cas réel, la variable de stabilité s'écrit $x = h\beta = -h\lambda = -z$ et le schéma est A-stable si $u_n \to 0$ pour tout $x \in \mathbb{R}^+$, c'est-à-dire pour tout $\beta > 0$ et pour tout pas de temps $h > 0$ (aucune restriction sur h). Conditionnellement A-stable si $u_n \to 0$ seulement si $0 < h < h_{\max}(\beta)$. Dans l'exemple du schéma d'Euler explicite, on trouve que $h_{\max}(\beta) = \frac{2}{\beta}$.

Définition : Région et Intervalle de stabilité absolue
La **région de stabilité absolue** $\mathcal{A}$ est l'ensemble des points du plan complexe où le schéma est stable :

$$\mathcal{A} = \{z \in \mathbb{C}^- \mid |u_n| \to 0 \text{ quand } n \to +\infty\}$$

- Si $\mathcal{A} = \mathbb{C}^-$, le schéma est inconditionnellement **A-stable**.
- Sinon, il est **A-stable sous condition**, ce qui impose une contrainte sur le pas : $0 < h < h_{\max}(\lambda)$.

L'**intervalle de stabilité** est la trace de cette région sur l'axe réel :

$$\mathcal{I} = \mathcal{A} \cap \mathbb{R}^- =]-h_{\max}\beta, 0].$$

L'étude de la A-stabilité revient à analyser une **suite de récurrence linéaire** dont les coefficients dépendent de x (ou z).

- **Méthodes à un pas :** la relation est de type $u_{n+1} = q(x)u_n$. La condition de stabilité est directe : il suffit que le facteur d'amplification vérifie $|q(x)| < 1$.
- **Méthodes multi-pas (MPL) :** pour un schéma à q étapes, la récurrence lie u_{n+1} à plusieurs états précédents $(u_n, u_{n-1}, \dots, u_{n-q+1})$. On obtient une récurrence linéaire à coefficients constants dépendant de x (ou z) dont la solution générale s'écrit :

$$u_n = \sum_{j=0}^{p} C_j r_j^n(x)$$

avec C_j des constantes qui dépendent des conditions initiales et les racines $r_j(x)$ qui dépendent de x. L'analyse devient plus délicate : la limite de u_n dépend de la position des racines du **polynôme de stabilité** par rapport au cercle unité. Si toutes les racines sont strictement à l'intérieur du disque unité ($|r_j(x)| < 1$), alors la méthode est absolument stable pour ce paramètre x.

Remarques : à première vue, l'exigence d'une stabilité absolue pour une méthode numérique appliquée à un autre problème que le *toy-model* peut sembler superflue. Cependant, il est possible de démontrer que si une méthode est absolument stable pour le *toy-model*, alors, lorsqu'elle est appliquée à un problème plus général, **l'erreur de perturbation** (c'est-à-dire la différence absolue entre la solution perturbée et la solution non perturbée) reste **uniformément bornée par rapport à** h. En d'autres termes, une méthode absolument stable selon la définition appliquée au *toy-model* garantit un contrôle des perturbations même dans des contextes plus généraux.

Nous allons calculer les conditions de A-stabilité (et de positivité) de différentes méthodes numériques d'approximation à un pas puis à deux pas. On verra dans le chapitre suivant comment généraliser ces résultats à des suites de plus de pas.

Acronyme	Schéma	Type	Ordre	A-stabilité	Positivité
Méthodes à 1 pas					
EE	$u_{n+1} = u_n + h\varphi_n$	Explicite	1	$0 < h < \frac{2}{\beta}$	$0 < h < \frac{1}{\beta}$
EI	$u_{n+1} = u_n + h\varphi_{n+1}$	Implicite	1	$\forall h > 0$	$\forall h > 0$
EM	$u_{n+1} = u_n + h\varphi\left(t_n + \frac{h}{2}, u_n + \frac{h}{2}\varphi_n\right)$	Explicite	2	$0 < h < \frac{2}{\beta}$	$0 < h < \frac{2}{\beta}$
CN	$u_{n+1} = u_n + \frac{h}{2}\left(\varphi_n + \varphi_{n+1}\right)$	Implicite	2	$\forall h > 0$	$0 < h < \frac{2}{\beta}$
H	$u_{n+1} = u_n + \frac{h}{2}\left(\varphi_n + \varphi(t_{n+1}, u_n + h\varphi_n)\right)$	Explicite	2	$0 < h < \frac{2}{\beta}$	$0 < h < \frac{2}{\beta}$
Méthodes à 2 pas					
AB2	$u_{n+1} = u_n + \frac{h}{2}\left(3\varphi_n - \varphi_{n-1}\right)$	Explicite	2	$0 < h < \frac{1}{\beta}$	$0 < h < \frac{1}{\beta}$
AM2	$u_{n+1} = u_n + \frac{h}{12}\left(8\varphi_n - \varphi_{n-1} + 5\varphi_{n+1}\right)$	Implicite	2	$0 < h < \frac{6}{\beta}$	$0 < h < \frac{6}{\beta}$
BDF2	$u_{n+1} = \frac{4}{3}u_n - \frac{1}{3}u_{n-1} + \frac{2}{3}h\varphi_{n+1}$	Implicite	2	$\forall h > 0$	$\forall h > 0$

9.2. Lien entre A-stabilité et Problèmes Stiff

La **A-stabilité**, bien qu'elle soit une propriété définie par le comportement asymptotique ($t \to \infty$), est l'outil fondamental pour approcher la solution des problèmes **raides** sur des intervalles de temps borné. Nous pouvons voir cette équivalence comme un changement d'échelle temporelle.

La raideur comme dilatation du temps : considérons le problème de Cauchy "stiff" sur l'intervalle borné $I = [0, 1]$ avec $\gamma \gg 1$:

$$\begin{cases} y'(t) = -\gamma y(t), \quad t \in [0, 1], \\ y(0) = 1. \end{cases}$$

Pour analyser le comportement d'un schéma numérique à l'échelle de la variation réelle de la solution, effectuons le changement de variable (dilatation du temps) suivant : $\tau = \gamma t$ et posons $\tilde{y}(\tau) = y(t = \frac{\tau}{\gamma})$.

Par dérivation composée, on obtient $\tilde{y}'(\tau) = y'(t) \cdot t'(\tau) = -\gamma y(\frac{\tau}{\gamma}) \cdot \frac{1}{\gamma} = -\tilde{y}(\tau)$. Le problème se reformule alors en termes de $\tilde{y}$ et τ :

$$\begin{cases} \tilde{y}'(\tau) = -\tilde{y}(\tau), \quad \tau \in [0, \gamma], \\ \tilde{y}(0) = 1. \end{cases}$$

Du problème local au comportement asymptotique : ce changement de variable révèle une équivalence qui lie la raideur à la stabilité.

- **Le problème initial** est un problème **stiff** sur un petit intervalle $[0, 1]$. La "pente" initiale est énorme $(-\gamma)$;
- **le problème équivalent** est un problème **non-stiff** (pente $\beta = -1$) mais sur un intervalle "presque infini" $[0, \gamma]$. Par exemple, si $\gamma = 10^6$, intégrer jusqu'à $t = 1$ revient à intégrer jusqu'à $\tau = 10^6$.

Pour intégrer numériquement sur un intervalle aussi grand sans que les erreurs de discrétisation ne s'accumulent ou n'explosent, le schéma doit simuler correctement le comportement de la solution pour $\tau \to \infty$. C'est précisément la définition de la **A-stabilité**.

En résumé : la raideur d'une EDO contraint le schéma numérique à "résoudre" des dynamiques internes extrêmement rapides, même si l'intervalle de temps global est court. Sans A-stabilité inconditionnelle, le pas de temps h est dicté par la stabilité (il doit être inversement proportionnel à γ), ce qui rend le calcul prohibitivement long. Avec la

A-stabilité, on s'affranchit de cette contrainte. Le pas de temps peut être choisi librement en fonction de la précision souhaitée, permettant ainsi de traverser efficacement les zones de forte raideur.

9.3. Étude de la A-stabilité pour un schéma à un pas

Un schéma à un pas est un schéma numérique dans lequel la valeur de u_{n+1} dépend uniquement de u_n (et éventuellement de t_n). Ces méthodes, appliquées au *toy-model*, conduisent à des relations de la forme $u_{n+1} = q(\beta h)u_n$, où q est une fonction de la variable $x = \beta h$. La condition de A-stabilité se traduit alors par l'exigence que $|q(x)| < 1$ pour tout $x > 0$.

Remarques sur les suites à 1 pas

Considérons une relation de récurrence de la forme

$$u_{n+1} = qu_n.$$

Par récurrence, on a

$$u_n = q^n u_0.$$

La suite tend vers 0 si et seulement si $|q| < 1$.

Exercice : étudier l'A-stabilité du schéma d'Euler implicite $u_{n+1} = u_n + h\varphi_{n+1}$.

Le schéma d'**Euler rétrograde** devient dans le cadre de notre exemple

$$(1 + \beta h)u_{n+1} = u_n, \qquad n = 0, 1, 2, ...$$

et par suite

$$u_n = \frac{1}{(1 + \beta h)^n} u_0,$$

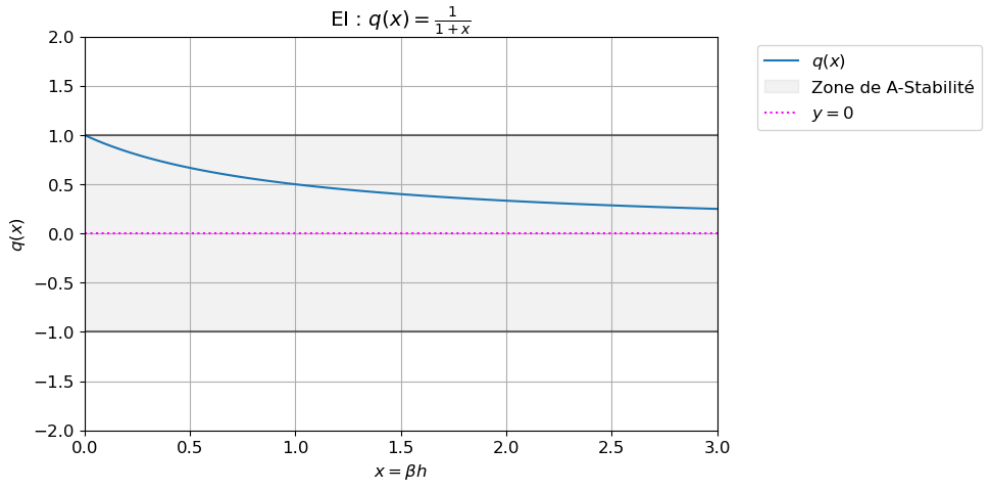
soit encore

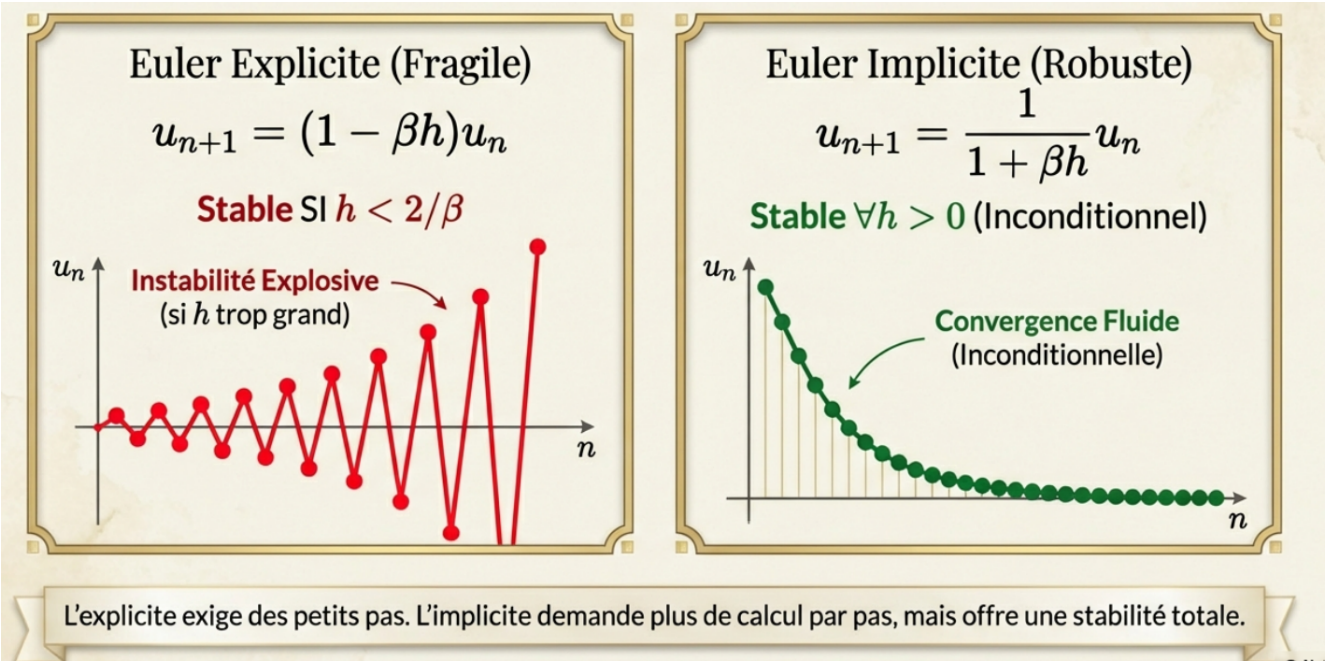
$$u_n = [q(\beta h)]^n u_0 \qquad \text{avec } q(\beta h) = \frac{1}{1 + \beta h}.$$

Dans ce cas $q(\beta h) \in]0, 1[$ **pour tout** $h > 0$, ainsi $\lim_{n\to\infty} u_n = 0$ pour tout $h > 0$: le schéma d'Euler rétrograde est donc toujours stable, sans limitations sur h.

Notons $x = \beta h > 0$ et traçons la fonction $x \mapsto q(x)$. On trouve que $q(x)$ est strictement décroissante, $q(0) = 1$ et tend vers 0 lorsque x tend vers l'infini.

```
x = np.linspace(0, 3, 101)
q = 1/(1 + x)
plt.figure(figsize=(8, 5))
plt.grid()
plt.plot(x, q, label='$q(x)$')
plt.ylim(-2, 2) # On zoome sur la zone intéressante [-1, 1]
plt.xlim(x.min(), x.max())
plt.axhline(y=1, color='black', linestyle='-', alpha=0.5)
plt.axhline(y=-1, color='black', linestyle='-', alpha=0.5)
plt.fill_between(x, -1, 1, color='gray', alpha=0.1, label='Zone de A-Stabilité')
plt.axhline(0, color='magenta', linestyle=':', label='$y=0$')
# plt.scatter(1, 0, color='black', label='$q(1) = 0$')
plt.xlabel(r'$x=\beta h$')
plt.ylabel('$q(x)$')
plt.title('EI : $q(x) = \\frac{1}{1 + x}$')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()
```





Exercice : étudier l'A-stabilité du schéma de Cranck-Nicolson $u_{n+1} = u_n + (\varphi_n + \varphi_{n+1}) \times h/2$.

```
beta = sp.symbols(r'\beta', real=True, positive=True)
h = sp.symbols('h', real=True, positive=True)
tn, un, unp1 = sp.symbols('t_n u_n u_{n+1}', real=True)
phi = lambda t,y: -beta*y
k1 = phi(tn, un)
k2 = phi(tn + h, unp1)
sol = sp.solve(  sp.Eq(unp1, un + h/2*(k1+k2)), unp1)[0]
sp.Eq(unp1, sol)
```

$$u_{n+1} = \frac{u_n\,(-\beta h + 2)}{\beta h + 2}$$

Le schéma de **Crank-Nicolson** appliqué à notre exemple s'écrit

$$\left(1 + \frac{\beta h}{2}\right) u_{n+1} = \left(1 - \frac{\beta h}{2}\right) u_n$$

et par suite

$$u_n = \left(\frac{2 - \beta h}{2 + \beta h}\right)^n u_0,$$

soit encore

$$u_n = [q(\beta h)]^n u_0 \qquad \text{avec } q(\beta h) = \frac{2-\beta h}{2+\beta h}.$$

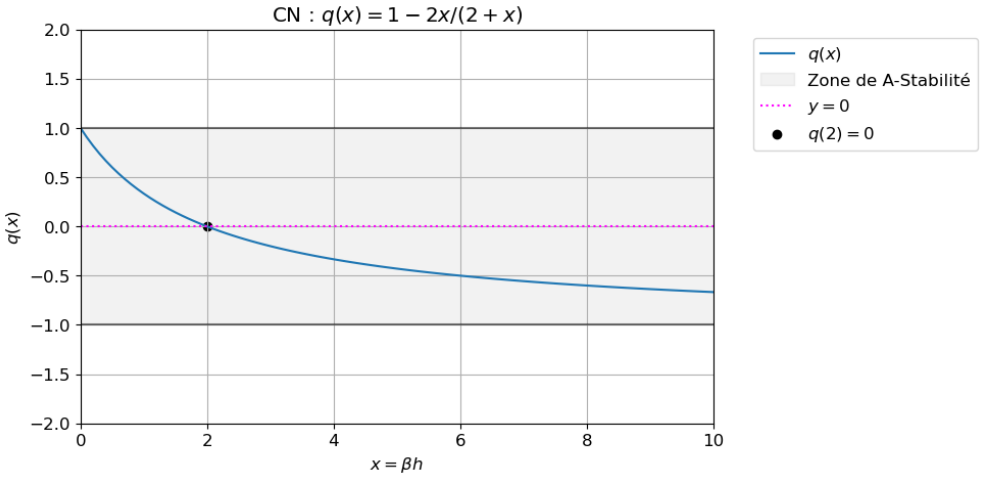
Par conséquent, $\lim_{n\to+\infty} u_n = 0$ si et seulement si

$$|q(\beta h)| < 1.$$

Notons x le produit $\beta h > 0$ et q la fonction $q(x) = \frac{2-x}{2+x} = 1 - 2\frac{x}{2+x} \xrightarrow[x\to+\infty]{} -1^+$. Nous avons $0 < \frac{x}{2+x} < 1$ pour tout $x \in \mathbb{R}_+^*$, donc $|q(x)| < 1$ pour tout $x \in \mathbb{R}_+^*$. La relation $\lim_{n\to+\infty} u_n = 0$ est donc satisfaite pour tout $h > 0$: le schéma de Crank-Nicolson est donc toujours stable, **sans limitations sur h.**
Cependant, si on cherche $0 < q(x) < 1$ on a (pour $x > 0$)

$$1 - 2\frac{x}{2+x} > 0 \iff x < 2.$$

```
x = np.linspace(0, 10, 101)
q = 1-2*x/(2+x)
plt.figure(figsize=(8, 5))
plt.grid()
plt.plot(x, q, label='$q(x)$')
plt.ylim(-2, 2) # On zoome sur la zone intéressante [-1, 1]
plt.xlim(x.min(), x.max())
plt.axhline(y=1, color='black', linestyle='-', alpha=0.5)
plt.axhline(y=-1, color='black', linestyle='-', alpha=0.5)
plt.fill_between(x, -1, 1, color='gray', alpha=0.1, label='Zone de A-Stabilité')
plt.axhline(0, color='magenta', linestyle=':', label='$y=0$')
plt.scatter(2, 0, color='black', label='$q(2) = 0$')
plt.xlabel(r'$x=\beta h$')
plt.ylabel('$q(x)$')
plt.title('CN : $q(x) = 1 - 2x/(2+x)$')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()
```



Exercice : étudier l'A-stabilité du schéma d'Euler modifié $u_{n+1} = u_n + h\varphi\left(t_n + \frac{h}{2}, u_n + \frac{h}{2}\varphi_n\right)$.

```
beta = sp.symbols(r'\beta', real=True, positive=True)
h = sp.symbols('h', real=True, positive=True)
tn, un, unp1 = sp.symbols('t_n u_n u_{n+1}', real=True)
phi = lambda t,y: -beta*y
k1 = phi(tn, un)
k2 = phi(tn + h/2, un+h/2*k1)
sol = sp.solve(  sp.Eq(unp1, un + h*k2), unp1)[0]
sp.Eq(unp1, sol)
```

$$u_{n+1} = \frac{u_n\left(\beta^2h^2 - 2\beta h + 2\right)}{2}$$

Pour $\varphi(t, y) = -\beta y$, on a

$$\varphi(t_n, u_n) = -\beta u_n \qquad \text{et} \qquad \varphi\left(t_n + \frac{h}{2}, u_n + \frac{h}{2}\varphi_n\right) = -\beta\left(u_n + \frac{h}{2}(-\beta u_n)\right).$$

ainsi le schéma d'**Euler modifié** pour notre exemple devient

$$u_{n+1} = u_n + h\left(-\beta\left(u_n + \frac{h}{2}(-\beta u_n)\right)\right) = \left(1 - \beta h + \frac{(\beta h)^2}{2}\right)u_n$$

Par induction on obtient

$$u_n = \left(1 - \beta h + \frac{(\beta h)^2}{2}\right)^n u_0$$

soit encore

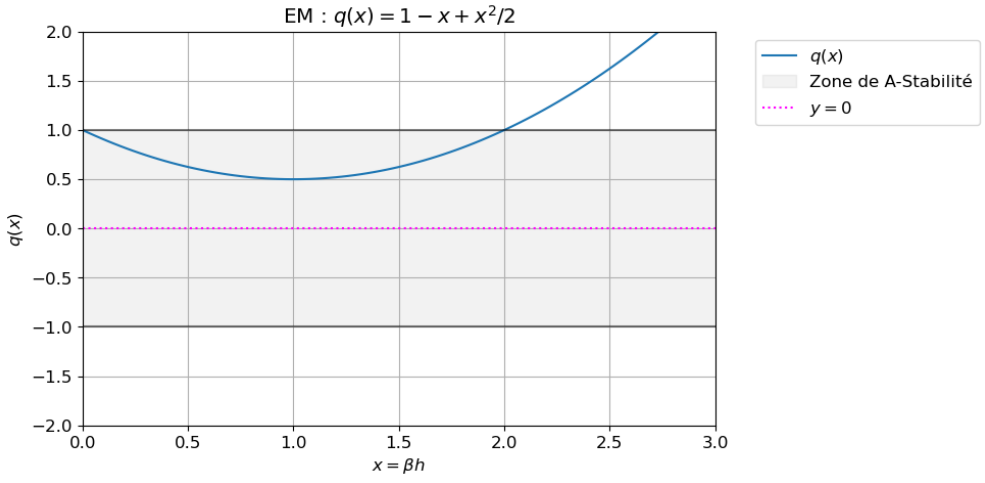
$$u_n = [q(\beta h)]^n u_0 \qquad \text{avec } q(\beta h) = 1 - \beta h + \frac{1}{2}(\beta h)^2.$$

Par conséquent, $\lim\limits_{n\to+\infty} u_n = 0$ si et seulement si

$$|q(\beta h)| < 1.$$

Notons x le produit βh et q le polynôme $q(x) = \frac{1}{2}x^2 - x + 1$ dont le graphe est représenté en figure ci-dessous. :

```
x = np.linspace(0, 3, 101)
q = 1-x+0.5*x**2
plt.figure(figsize=(8, 5))
plt.grid()
plt.plot(x, q, label='$q(x)$')
plt.ylim(-2, 2) # On zoome sur la zone intéressante [-1, 1]
plt.xlim(x.min(), x.max())
plt.axhline(y=1, color='black', linestyle='-', alpha=0.5)
plt.axhline(y=-1, color='black', linestyle='-', alpha=0.5)
plt.fill_between(x, -1, 1, color='gray', alpha=0.1, label='Zone de A-Stabilité')
plt.axhline(0, color='magenta', linestyle=':', label='$y=0$')
# plt.scatter(2, 0, color='black', label='$q(2) = 0$')
plt.xlabel(r'$x=\beta h$')
plt.ylabel('$q(x)$')
plt.title('EM : $q(x) = 1 - x + x^2/2$')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()
```



Nous avons $|q(x)| < 1$ si et seulement si $0 < x < 2$. La relation $\lim_{n\to+\infty} u_n = 0$ est donc satisfaite si et seulement si

$$\boxed{h < \frac{2}{\beta}.}$$

Cette condition de stabilité limite le pas h d'avance en t lorsqu'on utilise le schéma d'Euler modifié.

Notons qu'on a la même condition de A-stabilité que le schéma d'Euler explicite. Cependant, $q(x) > 0$ pour tout x, donc **cette convergence est monotone** (tandis qu'avec le schéma d'Euler explicite la convergence n'est monotone que si $h < \frac{1}{\beta}$).

Exercice : étudier l'A-stabilité du schéma de Heun $u_{n+1} = u_n + (\varphi(t_n, u_n) + \varphi(t_{n+1}, u_n + h\varphi(t_n, u_n))) \times h/2$.

```
beta = sp.symbols(r'\beta', real=True, positive=True)
h = sp.symbols('h', real=True, positive=True)
tn, un, unp1 = sp.symbols('t_n u_n u_{n+1}', real=True)
phi = lambda t,y: -beta*y
k1 = phi(tn, un)
k2 = phi(tn + h/2, un+h*k1)
sol = sp.solve(  sp.Eq(unp1, un + h/2*(k1+k2)), unp1)[0]
sp.Eq(unp1, sol)
```

$$u_{n+1} = \frac{u_n\left(\beta^2h^2 - 2\beta h + 2\right)}{2}$$

Le schéma de **Heun** pour notre exemple devient

$$u_{n+1} = u_n + \frac{h}{2}\left(-\beta u_n - \beta\left(u_n + h(-\beta u_n)\right)\right) = \left(1 - \beta h + \frac{(\beta h)^2}{2}\right) u_n$$

ce qui coïncide avec la méthode d'Euler modifiée. Par conséquent, $\lim_{n\to+\infty} u_n = 0$ si et seulement si

$$\boxed{h < \frac{2}{\beta}}$$

et cette **convergence est monotone**.
Cette condition de stabilité limite le pas h d'avance en t lorsqu'on utilise le schéma de Heun.

Exercice : étudier l'A-stabilité du schéma de Simpson implicite $u_{n+1} = u_n + \left(\varphi(t_n, u_n) + 4\varphi(t_n + \frac{h}{2}, u_n + \frac{h}{2}\varphi_n) + \varphi(t_{n+1}, u_{n+1})\right) \times h/2$.

```
beta = sp.symbols(r'\beta', real=True, positive=True)
h = sp.symbols('h', real=True, positive=True)
tn, un, unp1 = sp.symbols('t_n u_n u_{n+1}', real=True)
phi = lambda t,y: -beta*y
k1 = phi(tn, un)
k2 = phi(tn + h/2, un + h/2*k1)
k3 = phi(tn + h, unp1)
sol = sp.solve(  sp.Eq(unp1, un + h/6*(k1 + 4*k2 + k3)), unp1)[0]
sp.Eq(unp1, sol)
```

$$u_{n+1} = \frac{u_n\left(2\beta^2h^2 - 5\beta h + 6\right)}{\beta h + 6}$$

Le schéma de Simpson implicite appliqué à notre exemple s'écrit

$$\left(1 + \beta\frac{h}{6}\right) u_{n+1} = \left(1 - \frac{5}{6}\beta h + \frac{1}{3}(\beta h)^2\right) u_n$$

et par suite

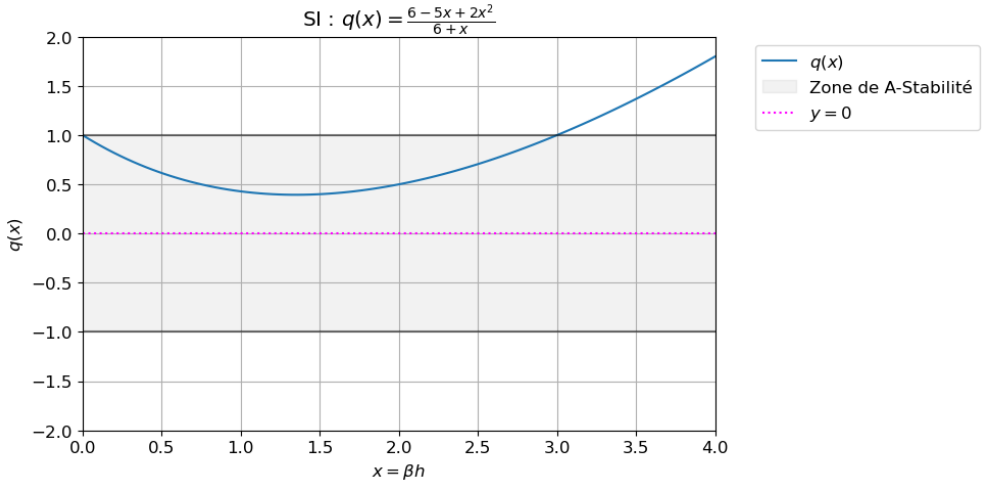
$$u_n = \left(\frac{6 - 5\beta h + 2(\beta h)^2}{6 + \beta h}\right)^n u_0, \qquad n = 0, 1, 2, ...$$

Par conséquent, $\lim_{n\to+\infty} u_n = 0$ si et seulement si

$$\left|\frac{6 - 5\beta h + 2(\beta h)^2}{6 + \beta h}\right| < 1.$$

Notons x le produit $\beta h > 0$ et q la fonction $q(x) = \frac{6-5x+2x^2}{6+x}$ dont le graphe est représenté en figure.

```
x = np.linspace(0, 4, 101)
q = (6-5*x+2*x**2)/(6+x)
plt.figure(figsize=(8, 5))
plt.grid()
plt.plot(x, q, label='$q(x)$')
plt.ylim(-2, 2) # On zoome sur la zone intéressante [-1, 1]
plt.xlim(x.min(), x.max())
plt.axhline(y=1, color='black', linestyle='-', alpha=0.5)
plt.axhline(y=-1, color='black', linestyle='-', alpha=0.5)
plt.fill_between(x, -1, 1, color='gray', alpha=0.1, label='Zone de A-Stabilité')
plt.axhline(0, color='magenta', linestyle=':', label='$y=0$')
# plt.scatter(2, 0, color='black', label='$q(2) = 0$')
plt.xlabel(r'$x=\beta h$')
plt.ylabel('$q(x)$')
plt.title('SI : $q(x) = \\frac{6-5x+2x^2}{6+x}$')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()
```



Nous avons $q(x) > 0$ pour tout $x \in \mathbb{R}_+^*$ et $q(x) < 1$ si et seulement si $x < 3$, donc $|q(x)| < 1$ pour tout $x \in]0; 3[$. Par conséquent, $\lim_{n\to+\infty} u_n = 0$ si et seulement si

$$h < \frac{3}{\beta}.$$

Cette condition de stabilité limite le pas h d'avance en t.

Exercice : étudier l'A-stabilité du schéma de Simpson explicite $u_{n+1} = u_n + \big(\varphi(t_n, u_n) + 4\varphi(t_n + \frac{h}{2}, u_n + \frac{h}{2}\varphi_n) + \varphi(t_{n+1}, u_n + h\varphi(t_n, u_n))\big) \times h/2$.

```
beta = sp.symbols(r'\beta', real=True, positive=True)
h = sp.symbols('h', real=True, positive=True)
tn, un, unp1 = sp.symbols('t_n u_n u_{n+1}', real=True)
```

```
phi = lambda t,y: -beta*y
k1 = phi(tn, un)
k2 = phi(tn + h/2, un + h/2*k1)
k3 = phi(tn + h, un+h*k1)
sol = sp.solve(  sp.Eq(unp1, un + h/6*(k1 + 4*k2 + k3)), unp1)[0]
sp.Eq(unp1, sol)
```

$$u_{n+1} = \frac{u_n\left(\beta^2h^2 - 2\beta h + 2\right)}{2}$$

Le schéma de Simpson explicite pour notre exemple devient

$$u_{n+1} = u_n + \frac{h}{6}\left(-\beta u_n - 4\beta\left(u_n + \frac{h}{2}(-\beta u_n)\right) - \beta\left(u_n + h(-\beta u_n)\right)\right) = \left(1 - \beta h + \frac{3}{2}(\beta h)^2\right)u_n$$

Par induction on obtient

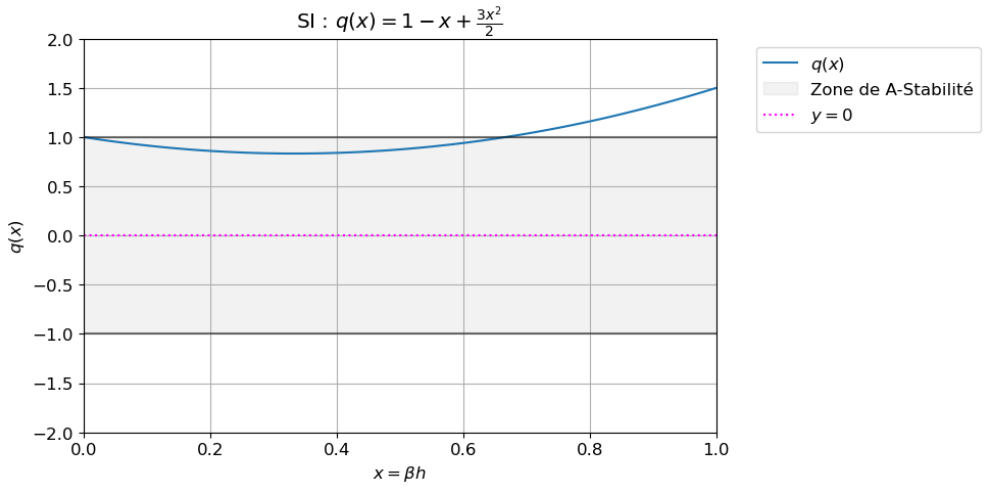
$$u_n = \left(1 - \beta h + \frac{3}{2}(\beta h)^2\right)^n u_0.$$

Par conséquent, $\lim\limits_{n\to+\infty} u_n = 0$ si et seulement si

$$\left|1 - \beta h + \frac{3}{2}(\beta h)^2\right| < 1.$$

Notons x le produit βh et q le polynôme $q(x) = \frac{3}{2}x^2 - x + 1$ dont le graphe est représenté en figure.

```
x = np.linspace(0, 1, 101)
q = 1-x+3*x**2/2
plt.figure(figsize=(8, 5))
plt.grid()
plt.plot(x, q, label='$q(x)$')
plt.ylim(-2, 2) # On zoome sur la zone intéressante [-1, 1]
plt.xlim(x.min(), x.max())
plt.axhline(y=1, color='black', linestyle='-', alpha=0.5)
plt.axhline(y=-1, color='black', linestyle='-', alpha=0.5)
plt.fill_between(x, -1, 1, color='gray', alpha=0.1, label='Zone de A-Stabilité')
plt.axhline(0, color='magenta', linestyle=':', label='$y=0$')
# plt.scatter(2, 0, color='black', label='$q(2) = 0$')
plt.xlabel(r'$x=\beta h$')
plt.ylabel('$q(x)$')
plt.title('SI : $q(x) = 1 - x + \\frac{3x^2}{2}$')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()
```



Nous avons $|q(x)| < 1$ si et seulement si $0 < x < \frac{2}{3}$. La relation $\lim_{n\to+\infty} u_n = 0$ est donc satisfaite si et seulement si

$$h < \frac{2}{3\beta}.$$

Cette condition de stabilité limite le pas h d'avance en t.

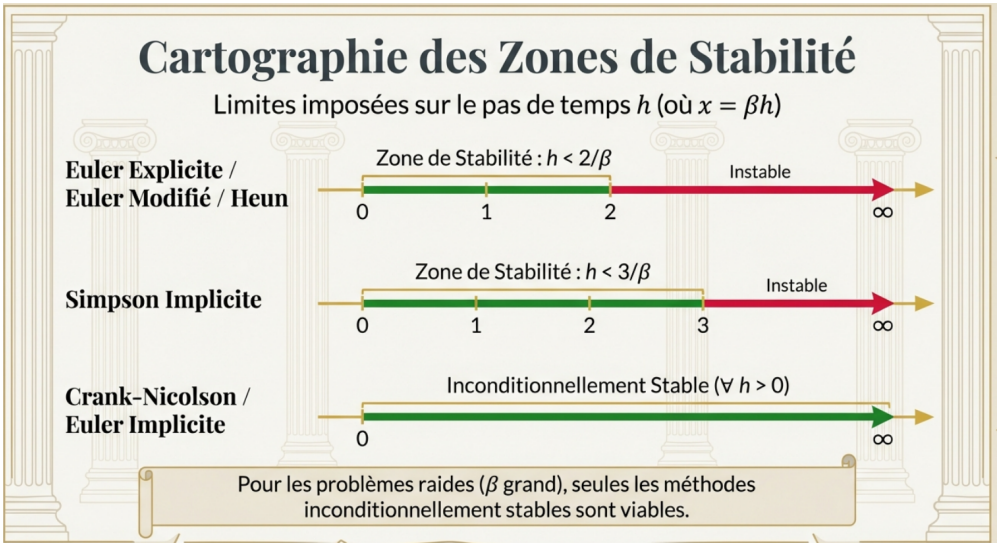


FIGURE 9.1. – Cartographie des zones de stabilité pour les schéma EE, EM, Heun (explicites) et EI, CN, SI (implicites)

9.4. Étude de la A-stabilité pour un schéma à deux pas

Les schémas à deux pas sont des méthodes numériques dans lesquelles la valeur de u_{n+1} dépend à la fois de u_n et de u_{n-1}. Ces méthodes, appliquées au *toy-model*, conduisent à des relations de la forme

$$u_{n+1} = a(\beta h)u_n + b(\beta h)u_{n-1},$$

où a et b sont des fonctions de la variable $x = \beta h$. La condition de A-stabilité se traduit alors par l'exigence que les racines de l'équation caractéristique associée à la relation de récurrence soient de module strictement inférieur à 1.

Remarques sur les suites à 2 pas

Considérons une relation de récurrence de la forme

$$u_{n+1} = \gamma u_n + \eta u_{n-1}.$$

Elle se réécrit sous la forme matricielle

$$\begin{pmatrix} u_{n+1} \\ u_n \end{pmatrix} = \begin{pmatrix} \gamma & \eta \\ 1 & 0 \end{pmatrix} \begin{pmatrix} u_n \\ u_{n-1} \end{pmatrix}$$

ainsi

$$\begin{pmatrix} u_{n+1} \\ u_n \end{pmatrix} = \begin{pmatrix} \gamma & \eta \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} u_1 \\ u_0 \end{pmatrix}$$

La suite u_n tend vers 0 si et seulement si les valeurs propres de la matrice de transition sont de module strictement inférieur à 1.
Les valeurs propres de cette matrice sont les racines du polynôme caractéristique

$$r^2 - \gamma r - \eta = 0$$

et notons $\Delta = \gamma^2 + 4\eta$ son discriminant.
3 cas se présentent alors :

1. Le polynôme a deux racines réelles distinctes $r_{1,2} = \frac{\gamma \pm \sqrt{\Delta}}{2}$. La matrice de transition est diagonalisable ainsi
$$\begin{pmatrix} \gamma & \eta \\ 1 & 0 \end{pmatrix}^n = \mathbb{P} \begin{pmatrix} r_1^n & 0 \\ 0 & r_2^n \end{pmatrix} \mathbb{P}^{-1}.$$
La suite u_n s'écrit $u_n = Ar_1^n + Br_2^n$, où les constantes A et B sont déterminées par les conditions initiales u_0 et u_1. La suite tend vers 0 si et seulement si $|r_1| < 1$ et $|r_2| < 1$.
2. Le polynôme a une racine double $r = \frac{\gamma}{2}$, ce qui correspond au cas où le discriminant Δ est nul. Alors
$$\begin{pmatrix} \gamma & \eta \\ 1 & 0 \end{pmatrix}^n = \mathbb{P} \begin{pmatrix} r^n & nr^{n-1} \\ 0 & r^n \end{pmatrix} \mathbb{P}^{-1}.$$
La suite s'écrit alors $u_n = (An + B)r^n$, où les constantes A et B sont déterminées par les conditions initiales u_0 et u_1. La suite tend vers 0 si et seulement si $|r| < 1$.
3. Le polynôme a deux racines complexes conjugués $r_{1,2} = \frac{\gamma}{2} \pm i\frac{\sqrt{-\Delta}}{2} = \varrho e^{i\theta}$, ce qui correspond au cas où le discriminant Δ est négatif. Alors
$$\begin{pmatrix} \gamma & \eta \\ 1 & 0 \end{pmatrix}^n = \mathbb{P} \begin{pmatrix} \varrho^n e^{i\theta n} & 0 \\ 0 & \varrho^n e^{-i\theta n} \end{pmatrix} \mathbb{P}^{-1} = \mathbb{Q}\varrho^n \begin{pmatrix} \cos(n\theta) & -\sin(n\theta) \\ \sin(n\theta) & \cos(n\theta) \end{pmatrix} \mathbb{Q}^{-1}$$
La suite s'écrit alors $u_n = \varrho^n(C\cos(n\theta) + D\sin(n\theta))$ avec $\varrho = \sqrt{-\eta}$, et $\cos(\theta) = \frac{\gamma}{2\varrho}$, où les constantes C et D sont déterminées par les conditions initiales u_0 et u_1. La suite tend vers 0 si et seulement si $\varrho < 1$.

Dans tous les cas, la suite u_n tend vers 0 si et seulement si les racines du polynôme caractéristique sont de module strictement inférieur à 1.

Exemple : étude de l'A-stabilité du schéma d'Adam-Bashforth à 2 pas.
Le schéma d'Adam-Bashforth à 2 pas s'écrit
$$\begin{cases} u_0 = y_0, \\ u_1 \approx y_1, \\ u_{n+1} = u_n + \frac{h}{2}\left(3\varphi_n - \varphi_{n-1}\right), \quad n = 1, 2, \dots, N-1. \end{cases}$$
Appliqué au problème modèle $y'(t) = -\beta y(t)$ s'écrit
$$u_{n+1} = u_n + \frac{h}{2}\left(-3\beta u_n + \beta u_{n-1}\right) = \left(1 - \frac{3\beta h}{2}\right)u_n + \frac{\beta h}{2}u_{n-1}.$$
C'est bien une suite à 2 pas de la forme
$$u_{n+1} = \gamma u_n + \eta u_{n-1}$$
avec $\gamma = 1 - \frac{3\beta h}{2}$ et $\eta = \frac{\beta h}{2}$.
Dans notre cas, $\Delta = \frac{9}{4}(\beta h)^2 - (\beta h) + 1$ qui est toujours strictement positif ainsi on est dans le premier cas :
$$u_n = Ar_1^n + Br_2^n$$
avec
$$r_{1,2} = \frac{1 - \frac{3\beta h}{2} \pm \sqrt{\left(1 - \frac{3\beta h}{2}\right)^2 - 2\beta h}}{2}$$
Pour que la méthode soit A-stable, il faut que les racines satisfassent $|r_i| < 1$ pour $i = 1, 2$. Si on note $x = \beta h$ et on trace le graphe des fonctions $r_1(x)$ et $r_2(x)$ on trouve que la condition $|r_i| < 1$ est satisfaite si et seulement si $x < 1$, c'est-à-dire $h < \frac{1}{\beta}$.

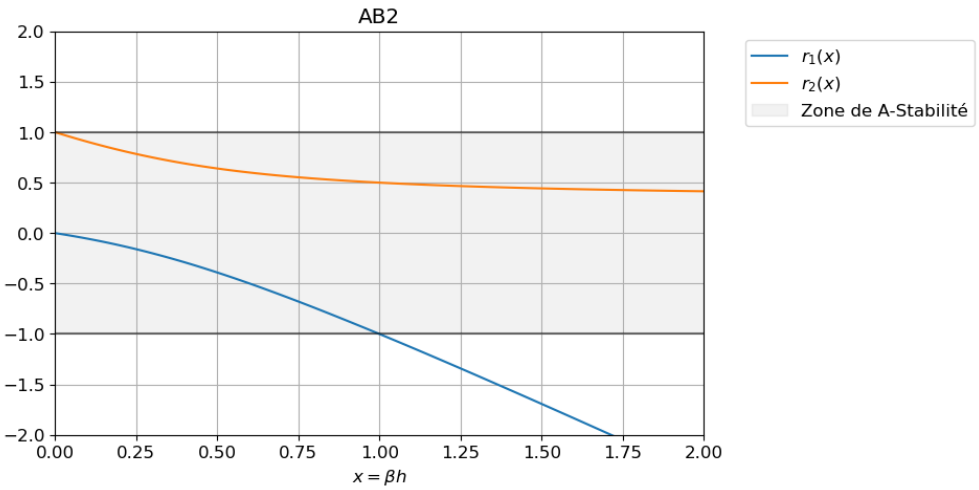
```
beta = sp.symbols(r'\beta', real=True, positive=True)
h = sp.symbols('h', real=True, positive=True)
tn, unm1, un, unp1 = sp.symbols('t_n u_{n-1} u_n u_{n+1}', real=True)
```

```
phi = lambda t,y: -beta*y
k1 = phi(tn, un)
k2 = phi(tn -h, unm1)
sol = sp.solve(  sp.Eq(unp1, un + h/2*(3*k1-k2) ), unp1)[0]
sp.Eq(unp1, sol)
```

$$u_{n+1} = -\frac{3\beta h u_n}{2} + \frac{\beta h u_{n-1}}{2} + u_n$$

```
gamma = lambda x : 1 - 3/2*x
eta   = lambda x : 1/2*x
Delta = lambda x : gamma(x)**2 + 4*eta(x)
r1 = lambda x: (gamma(x) - np.sqrt(Delta(x)))/2
r2 = lambda x: (gamma(x) + np.sqrt(Delta(x)))/2

xx = np.linspace(0,2,101)
plt.figure(figsize=(8, 5))
plt.grid()
yy1 = r1(xx)
yy2 = r2(xx)
plt.plot(xx,yy1, label=r'$r_1(x)$')
plt.plot(xx,yy2,  label=r'$r_2(x)$')
plt.ylim(-2, 2) # On zoome sur la zone intéressante [-1, 1]
plt.xlim(xx.min(), xx.max())
plt.axhline(y=1, color='black', linestyle='-', alpha=0.5)
plt.axhline(y=-1, color='black', linestyle='-', alpha=0.5)
plt.fill_between(xx, -1, 1, color='gray', alpha=0.1, label='Zone de A-Stabilité')
plt.xlabel(r'$x=\beta h$')
plt.title('AB2')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()
```



Exercice : étudier l'A-stabilité du schéma d'Adam-Moulton à 2 pas.

```
beta = sp.symbols(r'\beta', real=True, positive=True)
h = sp.symbols('h', real=True, positive=True)
tn, unm1, un, unp1 = sp.symbols('t_n u_{n-1} u_n u_{n+1}', real=True)
phi = lambda t,y: -beta*y
k1 = phi(tn, un)
k2 = phi(tn-h, unm1)
km1 = phi(tn+h, unp1)
sol = sp.solve(  sp.Eq(unp1, un + h/12*(8*k1-k2+5*km1) ), unp1)[0]
sp.Eq(unp1, sol)
```

$$u_{n+1} = \frac{-8\beta h u_n + \beta h u_{n-1} + 12u_n}{5\beta h + 12}$$

Le schéma d'Adam-Moulton à 2 pas s'écrit

$$\begin{cases} u_0 = y_0, \\ u_1 \approx y_1, \\ u_{n+1} = u_n + \frac{h}{12}\left(8\varphi_n - \varphi_{n-1} + 5\varphi_{n+1}\right), \quad n = 1, 2, \dots, N-1. \end{cases}$$

Appliqué au problème modèle $y'(t) = -\beta y(t)$ s'écrit

$$u_{n+1} = u_n + \frac{h}{12}\left(8(-\beta u_n) - (-\beta u_{n-1}) + 5(-\beta u_{n+1})\right).$$

C'est bien une suite à 2 pas de la forme

$$u_{n+1} = \gamma u_n + \eta u_{n-1}$$

avec $\gamma = \frac{12-8\beta h}{12+5\beta h}$ et $\eta = \frac{\beta h}{12+5\beta h}$.

Dans notre cas, $\Delta = \left(\frac{12-8\beta h}{12+5\beta h}\right)^2 + 4\frac{\beta h}{12+5\beta h}$ qui est toujours strictement positif ainsi on est dans le premier cas :

$$u_n = Ar_1^n + Br_2^n$$

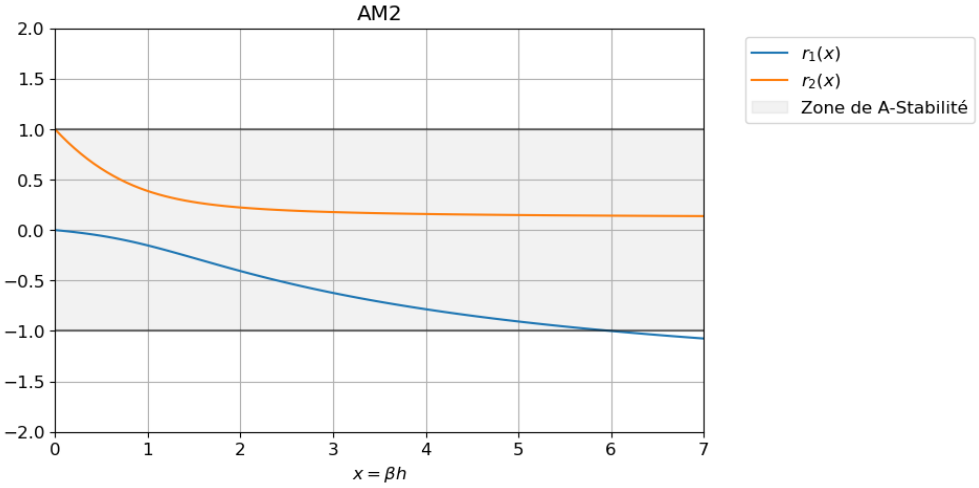
avec

$$r_{1,2} = \frac{\frac{12-8\beta h}{12+5\beta h} \pm \sqrt{\left(\frac{12-8\beta h}{12+5\beta h}\right)^2 + 4\frac{\beta h}{12+5\beta h}}}{2}$$

Pour que la méthode soit A-stable, il faut que les racines satisfassent $|r_i| < 1$ pour $i = 1, 2$. Si on note $x = \beta h$ et on trace le graphe des fonctions $r_1(x)$ et $r_2(x)$ on trouve que la condition $|r_i| < 1$ est satisfaite si et seulement si $x < 6$, c'est-à-dire $h < \frac{6}{\beta}$. On voit ici que les schémas implicites ne sont pas forcément inconditionnellement A-stables.

```
gamma = lambda x : (12-8*x)/(12+5*x)
eta   = lambda x : x/(12+5*x)
Delta = lambda x : gamma(x)**2 + 4*eta(x)
r1 = lambda x: (gamma(x) - np.sqrt(Delta(x)))/2
r2 = lambda x: (gamma(x) + np.sqrt(Delta(x)))/2

xx = np.linspace(0,7,101)
plt.figure(figsize=(8, 5))
plt.grid()
yy1 = r1(xx)
yy2 = r2(xx)
plt.plot(xx,yy1, label=r'$r_1(x)$')
plt.plot(xx,yy2,  label=r'$r_2(x)$')
plt.ylim(-2, 2) # On zoome sur la zone intéressante [-1, 1]
plt.xlim(xx.min(), xx.max())
plt.axhline(y=1, color='black', linestyle='-', alpha=0.5)
plt.axhline(y=-1, color='black', linestyle='-', alpha=0.5)
plt.fill_between(xx, -1, 1, color='gray', alpha=0.1, label='Zone de A-Stabilité')
plt.xlabel(r'$x=\beta h$')
plt.title('AM2')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()
```



Exercice : étudier l'A-stabilité du schéma BDF à 2 pas.

Le schéma BDF à 2 pas s'écrit

$$\begin{cases} u_0 = y_0, \\ u_1 \approx y_1, \\ u_{n+1} = \frac{4}{3}u_n - \frac{1}{3}u_{n-1} + \frac{2}{3}h\varphi_{n+1}, \quad n = 1, 2, \dots, N-1. \end{cases}$$

Appliqué au problème modèle $y'(t) = -\beta y(t)$ s'écrit

$$u_{n+1} = \frac{4}{3}u_n - \frac{1}{3}u_{n-1} + \frac{2}{3}h(-\beta u_{n+1}).$$

C'est bien une suite à 2 pas de la forme

$$u_{n+1} = \gamma u_n + \eta u_{n-1}$$

avec $\gamma = \frac{4}{3+2\beta h}$ et $\eta = -\frac{1}{3+2\beta h}$.

```
beta = sp.symbols(r'\beta', real=True, positive=True)
h = sp.symbols('h', real=True, positive=True)
tn, unm1, un, unp1 = sp.symbols('t_n u_{n-1} u_n u_{n+1}', real=True)
phi = lambda t,y: -beta*y
km1 = phi(tn+h, unp1)
sol = sp.solve(  sp.Eq(unp1, sp.Rational(4,3)*un-sp.Rational(1,3)*unm1+sp.Rational(2,3)*h*km1 ), unp1)[0]
sp.Eq(unp1, sol)
```

$$u_{n+1} = \frac{4u_n - u_{n-1}}{2\beta h + 3}$$

Les racines du polynôme caractéristique sont données par

$$r_{1,2} = \frac{\gamma \pm \sqrt{\gamma^2 + 4\eta}}{2} = \frac{\frac{4}{3+2\beta h} \pm \sqrt{\left(\frac{4}{3+2\beta h}\right)^2 - \frac{4}{3+2\beta h}}}{2}$$

Dans notre cas, $\Delta = \gamma^2 + 4\eta = \frac{4-2\beta h}{(3+2\beta h)^2}$ qui change de signe selon les valeurs de βh.

- Si $\beta h < \frac{1}{2}$, alors $\Delta > 0$ et on est dans le premier cas avec deux racines réelles distinctes et $|r_1| < 1$ et $|r_2| < 1$.
- Si $\beta h = \frac{1}{2}$, alors $\Delta = 0$ et on est dans le deuxième cas avec une racine double et $|r| < 1$.
- Si $\beta h > \frac{1}{2}$, alors $\Delta < 0$ et on est dans le troisième cas avec deux racines complexes conjugées et $\rho < 1$.

En traçant la valeur absolue des racines en fonction de x, on peut visualiser l'intervalle de stabilité absolue. On trouve que la méthode est A-stable pour tout $x > 0$. En effet, les racines sont de module strictement inférieur à 1 pour tout $x > 0$. Par conséquent, la méthode est A-stable pour tout $h > 0$.

```
gamma = lambda x :  4/(3+2*x)
eta   = lambda x : -1/(3+2*x)

r = sp.Symbol('r')
x = sp.Symbol('x', real=True, positive=True)

poly = sp.Poly(r**2-gamma(x)*r-eta(x), r)
display(Math(f"p_x(r) = {sp.latex(poly.as_expr())}"))

# ============================================================================

display(Math(f"$p_x(r) = {sp.latex(poly.as_expr())}$"))

roots = sp.solve(poly, r)

display(Markdown(f"Racines : $r_1(x) = {sp.latex(roots[0])}$, $r_2(x) = {sp.latex(roots[1])}$"))

# On définit une fonction qui calcule le module manuellement pour éviter le sqrt(négatif) dans numpy
def get_abs_expr(root):
    re_part = sp.re(root)
    im_part = sp.im(root)
    return sp.sqrt(re_part**2 + im_part**2)

abs_r1_expr = get_abs_expr(roots[0])
abs_r2_expr = get_abs_expr(roots[1])

r1_func = sp.lambdify(x, abs_r1_expr, modules=['numpy'])
r2_func = sp.lambdify(x, abs_r2_expr, modules=['numpy'])


# Le discriminant s'annule en
x_bifurcation = sp.solve(poly.discriminant(), x)[0]
display(Markdown(f"Point de bifurcation (racines complexes) : $x = {sp.latex(x_bifurcation)}$"))

x_vals = np.linspace(0, 5, 1000)

r_abs_1 = r1_func(x_vals)
r_abs_2 = r2_func(x_vals)


plt.figure(figsize=(10, 6))

plt.axhspan(0, 1, color='gray', alpha=0.1, label=r'Zone stable $|r| \leq 1$')

plt.plot(x_vals, r_abs_1, label=r'$|r_1(x)|$', lw=2)
plt.plot(x_vals, r_abs_2, label=r'$|r_2(x)|$', lw=2, linestyle='--')

# Point où les racines fusionnent (deviennent complexes)
plt.axvline(x_bifurcation, color='orange', linestyle=':', label=f'Bifurcation x = {x_bifurcation}')

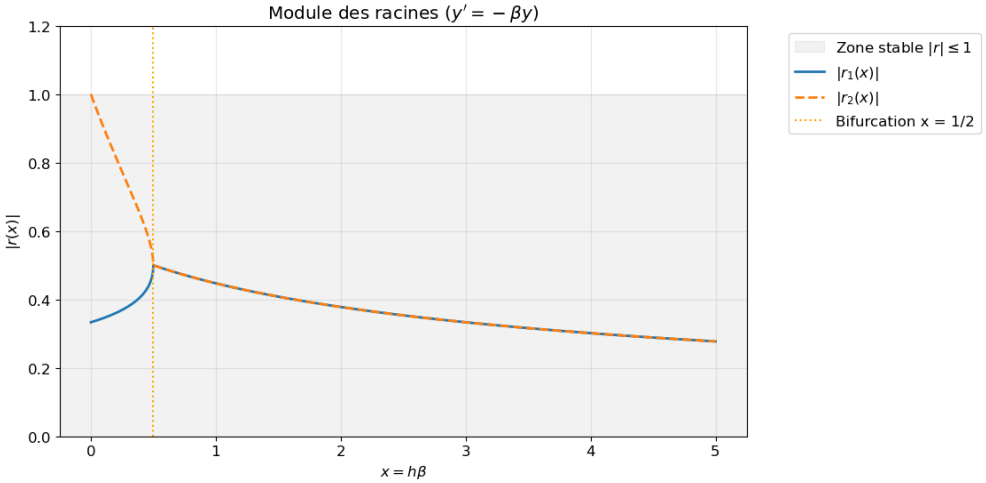
plt.title(r"Module des racines ($y' = -\beta y$)")
plt.xlabel('$x = h\\beta$')
plt.ylabel('$|r(x)|$')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.grid(True, alpha=0.3)
plt.ylim(0, 1.2)
plt.show()
```

$$p_x(r) = r^2 - \frac{4r}{2x+3} + \frac{1}{2x+3}$$

$$p_x(r) = r^2 - \frac{4r}{2x+3} + \frac{1}{2x+3}$$

Racines : $r_1(x) = \frac{2-\sqrt{1-2x}}{2x+3}$, $r_2(x) = \frac{\sqrt{1-2x}+2}{2x+3}$

Point de bifurcation (racines fusionnent) : $x = \frac{1}{2}$



9.5. Vérification empirique des conditions de A-stabilité

Exercice : tester numériquement l'A-stabilité des schémas classiques à un pas
On considère le *toy-model* avec $\beta = 1$ et on teste numériquement l'A-stabilité des schémas d'Euler explicite, d'Euler implicite, d'Euler modifié, de Crank-Nicolson et de Heun.

$$\begin{cases} y'(t) = -y(t), \quad t \in [0; 12] \\ y(0) = 1. \end{cases}$$

Calculer la solution exacte. Calculer ensuite la solution approchée par les schémas pour différentes valeurs de h. Vérifier que les conditions de stabilité sont respectées.

```
def affichage(schema):
    # CALCUL DE LA SOLUTION SUR CHAQUE GRILLE
    U = [ schema(phi,tt,y0) for tt in T ]

    # AFFICHAGE
    plt.figure(1, figsize=(8, 5))
    plt.axis([t0, tfinal, -1, 1])

    plt.plot(T[0],yy,'-',label='$y(t)=e^{-t}$')
    for k in range(5):
        plt.plot(T[k],U[k],'-o',label=f'h = {H[k]}')
    plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
    plt.grid();
```

Solution exacte

L'unique solution du problème de Cauchy est la fonction $y(t) = e^{-t}$ définie pour tout $t \in \mathbb{R}$.

```
phi = lambda t,y : -y
sol_exacte = lambda t : np.exp(-t)

# INITIALISATION
```

```
t0, y0, tfinal = 0 , 1 , 12

# PREPARATIONS DES CAS TEST pour différentes valeurs de h
H = [ 2**(k-2) for k in range(5) ]
T = [ np.arange(t0,tfinal,h) for h in H] # liste des grilles

yy = sol_exacte(T[0]) # on affichera la sol exacte sur la grille la plus fine
```

Solution approchée par la méthode d'Euler explicite

La méthode d'Euler explicite pour cette EDO s'écrit

$$u_{n+1} = (1-h)u_n.$$

En procédant par récurrence sur n, on obtient

$$u_{n+1} = (1-h)^{n+1}.$$

De la formule $u_{n+1} = (1-h)^{n+1}$ on déduit que

- si $0 < h < 1$ alors la solution numérique est stable et convergente,
- si $h = 1$ alors la solution numérique est stationnaire $u_n = 0$ pour tout $n \in \mathbb{N}^*$,
- si $1 < h < 2$ alors la solution numérique oscille mais est encore convergente,
- si $h = 2$ alors la solution numérique oscille, plus précisément on a $u_{2n} = 1$ et $u_{2n+1} = -1$ pour tout $n \in \mathbb{N}^*$,
- si $h > 2$ alors la solution numérique oscille et diverge.

En effet, la méthode est A-stable si et seulement si $|1-h| < 1$.

Remarque : la suite obtenue est une suite géométrique de raison $q = 1-h$. On sait qu'une telle suite

- diverge si $|q| > 1$ ou $q = -1$,
- est stationnaire si $q = 1$,
- converge vers 0 si $|q| < 1$.
- converge de façon monotone vers 0 si $0 < q < 1$.

Voyons graphiquement ce que cela donne avec différentes valeurs de $h > 0$:

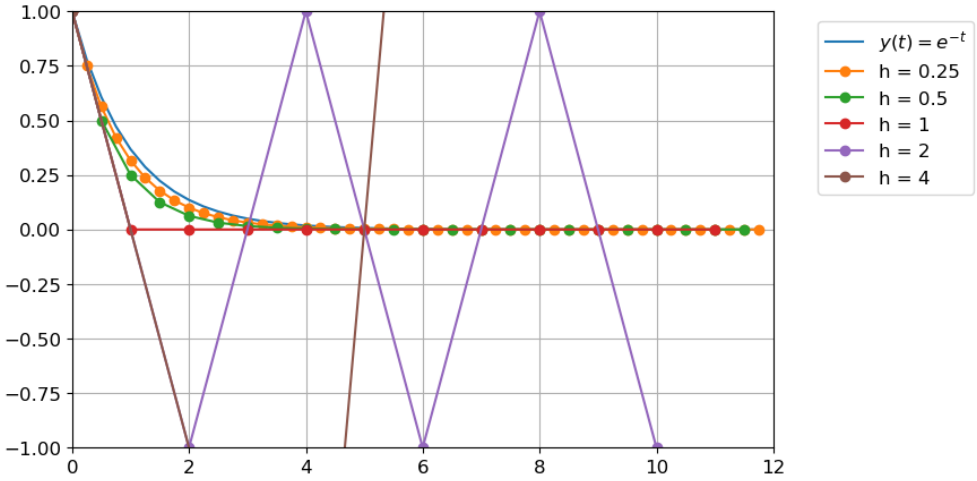
- si $h = 4$ alors $t_n = 4n$ et $u_n = (-4)^n$ tandis que $y(t_n) = e^{-4n}$,
- si $h = 2$ alors $t_n = 2n$ et $u_n = (-1)^n$ tandis que $y(t_n) = e^{-2n}$,
- si $h = 1$ alors $t_n = n$ et $u_n = 0$ tandis que $y(t_n) = e^{-n}$,
- si $h = \frac{1}{2}$ alors $t_n = \frac{n}{2}$ et $u_n = \left(\frac{1}{2}\right)^n$ tandis que $y(t_n) = e^{-n/2}$,
- si $h = \frac{1}{4}$ alors $t_n = \frac{n}{4}$ et $u_n = \left(\frac{3}{4}\right)^n$ tandis que $y(t_n) = e^{-n/4}$.

Ci-dessous sont tracées sur l'intervalle $[0; 10]$, les courbes représentatives de la solution exacte et de la solution calculée par la méthode d'Euler explicite. En faisant varier le pas h nous pouvons constater que si $h > 1$ l'erreur commise entre la solution exacte et la solution calculée est amplifiée d'un pas à l'autre.

Remarque : la première itérée a la même pente quel que soit le pas h (se rappeler la construction géométrique de la méthode d'Euler).

```
# SCHEMA
def EE(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    uu[0] = y0
    for i in range(len(tt)-1):
        uu[i+1] = uu[i]+h*phi(tt[i],uu[i])
    return uu


affichage(EE)
```



Solution approchée par la méthode d'Euler implicite

La méthode d'Euler implicite pour cette EDO s'écrit

$$u_{n+1} = \frac{1}{1+h} u_n.$$

En procédant par récurrence sur n, on obtient

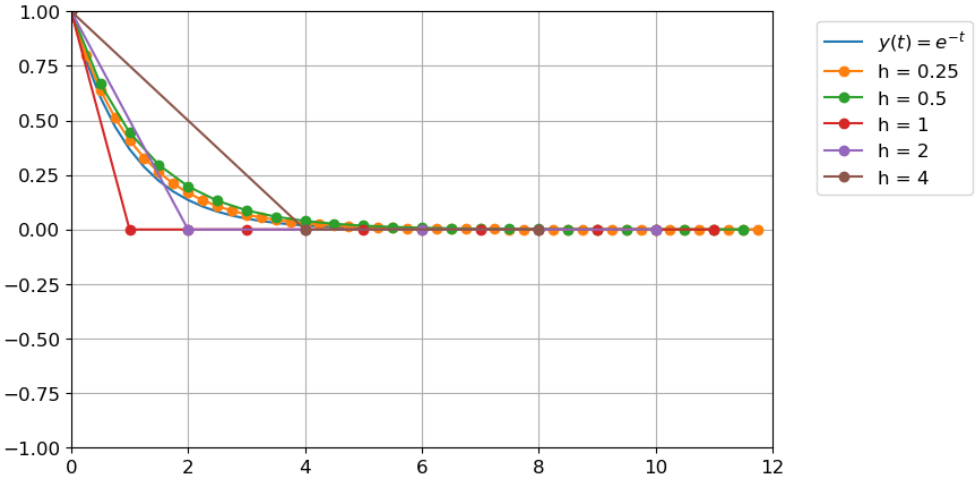
$$u_{n+1} = \frac{1}{(1+h)^{n+1}}.$$

De la formule $u_{n+1} = (1+h)^{-(n+1)}$ on déduit que la solution numérique est stable et convergente pour tout h. En effet, la méthode est inconditionnellement A-stable.

Remarque : la suite obtenue est une suite géométrique de raison $q = \frac{1}{1+h} \in]0;1[$.

```
# SCHEMA
def EI(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    uu[0] = y0
    for i in range(len(tt)-1):
        uu[i+1] = fsolve(lambda x:-x+uu[i]+h*phi(tt[i+1],x),uu[i])[0]
    return uu


affichage(EI)
```



Solution approchée par la méthode d'Euler modifié

La méthode d'Euler modifié pour cette EDO s'écrit

$$u_{n+1} = (1 - h + \frac{1}{2}h^2)u_n.$$

En procédant par récurrence sur n, on obtient

$$u_{n+1} = (1 - h + \frac{1}{2}h^2)^{n+1}.$$

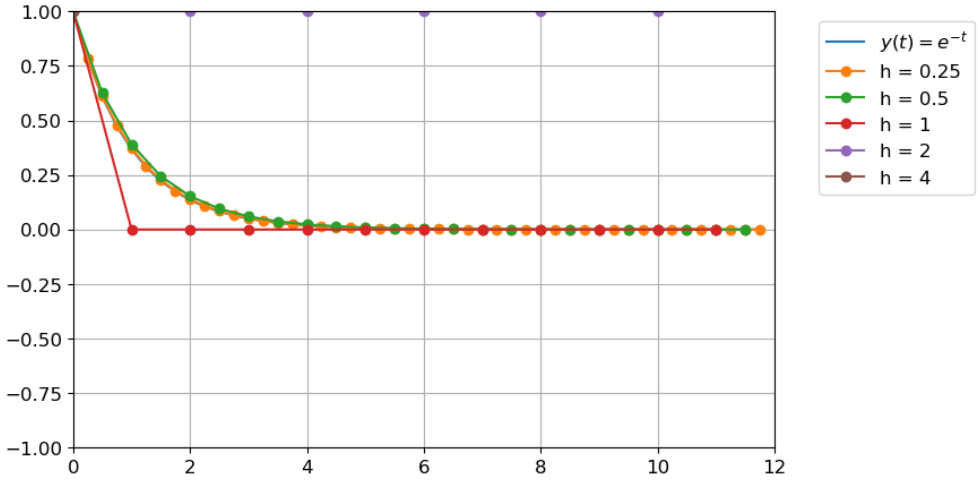
La suite obtenue est une suite géométrique de raison $q = 1 - h + \frac{1}{2}h^2$:

- si $0 < h < 2$ alors la solution numérique est stable et convergente,
- si $h = 2$ alors la solution numérique est stationnaire $u_n = u_0$ pour tout $n \in \mathbb{N}^*$,
- si $h > 2$ alors la solution numérique diverge vers $+\infty$.

En effet, la méthode est A-stable si et seulement si $|1 - h + \frac{1}{2}h^2| < 1$. De plus, comme $1 - h + \frac{1}{2}h^2 > 0$ pour tout $h > 0$, la suite est positive.

```
# SCHEMA
def EM(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    uu[0] = y0
    for i in range(len(tt)-1):
        utilde = uu[i]+h/2*phi(tt[i],uu[i])
        uu[i+1] = uu[i]+h*phi(tt[i]+h/2,utilde)
    return uu


affichage(EM)
```



Solution approchée par la méthode de Crank-Nicolson

La méthode de Crank-Nicolson pour cette EDO s'écrit

$$u_{n+1} = \frac{2-h}{2+h} u_n.$$

En procédant par récurrence sur n, on obtient

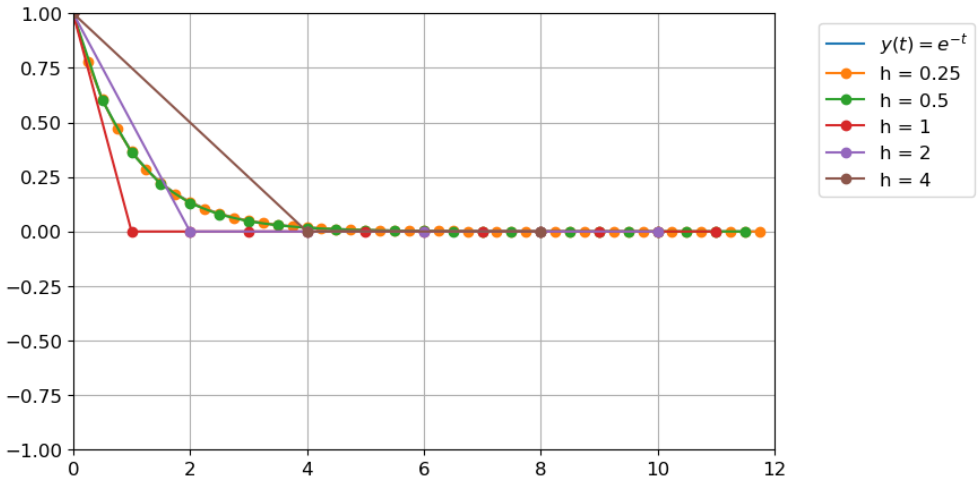
$$u_{n+1} = \left(\frac{2-h}{2+h}\right)^{n+1}.$$

La suite obtenue est une suite géométrique de raison $q = \frac{2-h}{2+h} \in]-1;1[$. On déduit que la solution numérique est stable et convergente pour tout h. En effet, la méthode est inconditionnellement A-stable.

De plus, la suite est positive ssi $0 < q < 1$ donc ssi $h < 2$.

```
# SCHEMA
def CN(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    uu[0] = y0
    for i in range(len(tt)-1):
        uu[i+1] = fsolve(lambda x:-x+uu[i]+h/2*phi(tt[i],uu[i])+h/2*phi(tt[i+1],x),uu[i])[0]
    return uu


affichage(CN)
```



Solution approchée par la méthode de Heun

La méthode de Heun pour cette EDO s'écrit

$$u_{n+1} = \left(1 - h + \frac{1}{2}h^2\right) u_n.$$

On retrouve le schéma d'Euler modifié.

9.6. Annexe : A-stabilité d'un système

Considérons un système de n EDO de la forme

$$\mathbf{y}'(t) = -\mathbb{A}\mathbf{y}(t)$$

où $\mathbb{A}$ est une matrice constante ayant n valeurs propres positives λ_i, $i = 1, 2, \dots, n$.
La solution générale s'écrit

$$\mathbf{y}(t) = \sum_i c_i \mathbf{v}_i e^{-\lambda_i t}$$

où $\mathbf{v}_i$ est le i vecteur propre associé à la valeur propre λ_i. On a donc :

$$\lim_{t \to +\infty} y_i(t) = 0 \quad \forall i = 1, 2, \dots, n.$$

Systèmes raides (*stiff systems*) : un système d'équations différentielles est dit *raide* lorsque les valeurs propres λ_i de la matrice $\mathbb{A}$ présentent des *magnitudes très différentes*. En d'autres termes, si certaines valeurs propres sont beaucoup plus grandes que d'autres, alors le système évolue selon des *échelles de temps très variées*. Cette différence de magnitudes crée un défi pour les méthodes numériques : pour garantir la A-stabilité, il est parfois nécessaire de choisir un pas de temps h suffisamment petit, proportionnel au plus grand λ_i. Cela peut rendre les calculs très coûteux lorsque les valeurs propres sont très disparates. C'est pour cela qu'on préfère des *méthodes inconditionnellement A-stables* pour approcher la solution d'un système raide, car elles permettent d'utiliser des pas de temps plus grands tout en maintenant la A-stabilité, indépendamment du plus grand λ_i.

Chapitre 10.

Convergence et A-stabilité des schémas multi-pas linéaires

```
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import display, Math, Latex, Markdown
plt.rcParams.update({'font.size': 14})
```

10.1. Rappels

Problème de Cauchy de référence

$$\begin{cases} y'(t) = \varphi(t, y(t)), \quad t \in [t_0, T], \\ y(t_0) = y_0, \end{cases}$$

avec solution unique $y(t)$ et approximation $u_n \approx y(t_n)$ sur la grille

$$t_n = t_0 + nh, \quad n = 0, \dots, N, \quad h = \frac{T - t_0}{N}.$$

10.1.1. Schémas multi-pas linéaires (MPL)

Notons $\varphi_{n-j} = \varphi(t_{n-j}, u_{n-j})$.
Soit $p \geq 0$ et posons $q = p + 1$, q étant le nombre de pas du schéma.
Soient les $2q + 1$ coefficients : $\{a_j\}_{j=0}^{p}$, $\{b_j\}_{j=-1}^{p}$. Une méthode multi-pas linéaire à q pas s'écrit

$$\begin{aligned} u_{n+1} &= \sum_{j=0}^{p} a_j u_{n-j} + h \sum_{j=-1}^{p} b_j \varphi_{n-j} \\ &= \sum_{j=0}^{p} \Big(a_j u_{n-j} + b_j \varphi_{n-j} \Big) + h b_{-1} \varphi_{n+1}, \end{aligned} \qquad n = p, \dots, N-1.$$

Initialisation : valeurs nécessaires : $u_0, \dots, u_p$ avec $u_0 = y_0$ connu et $u_1, \dots, u_p$ calculés via une autre méthode.
Elle est **explicite** si $b_{-1} = 0$, et **implicite** sinon.

10.1.2. Généralités : convergence et stabilités

- La convergence se décompose en une propriété locale (consistance) et une propriété de propagation des erreurs (zéro-stabilité).
  Théorème de Lax–Richtmyer : **Consistance + zéro-stabilité $\iff$ convergence**
  La convergence étudie l'évolution des erreurs lorsque $N \to \infty$ sur un intervalle borné (et donc $h \to 0$). Pour des intervalles très grands (voir infinis) la zéro-stabilité ne suffit plus pour garantir un bon comportement numérique. Il faut alors introduire la notion de **A-stabilité**.
- **Stabilité absolue (A-stabilité)** : la solution approchée u_n doit tendre vers zéro pour tout pas $h > 0$

fixé (ou sous certaines conditions sur h) lorsque le schéma est appliqué au *toy-model* $y'(t) = -\beta y(t)$ et $y(0) = 1$ avec $\beta > 0$.

10.2. Polynômes caractéristiques

Dans ce chapitre, nous allons étudier d'abord la convergence (à travers les notions de consistance et de zéro-stabilité) puis la A-stabilité d'une méthode multi-step.

Dans l'analyse de ces méthodes, trois polynômes jouent un rôle clé. Pour cela, commençons par réécrire le schéma multi-step comme suit :

$$u_{n+1} - \sum_{j=0}^{p} a_j u_{n-j} = h \left(b_{-1}\varphi_{n+1} + \sum_{j=0}^{p} b_j \varphi_{n-j} \right).$$

Définition.
Soit une méthode MPL. On lui associe les trois polynômes caractéristiques suivants définis dans $\mathbb{C}$:

- premier polynôme caractéristique

$$\varrho(r) = r^{p+1} - \sum_{j=0}^{p} a_j r^{p-j}$$

- second polynôme caractéristique

$$\sigma(r) = b_{-1} r^{p+1} + \sum_{j=0}^{p} b_j r^{p-j}$$

- polynôme de A-stabilité (une famille de polynômes paramétrés par z)

$$p_z(r) = \varrho(r) - z\sigma(r) = (1 - zb_{-1}) r^{p+1} - \sum_{j=0}^{p} (a_j + zb_j) r^{p-j}$$

Formellement,

- pour construire le premier polynôme caractéristique on ne regarde que le LHS et les u_{n-j} deviennent des r^{p-j},
- pour construire le second polynôme caractéristique on ne regarde que le RHS divisé par h et les φ_{n-j} deviennent des r^{p-j}.

On verra que les racines du polynôme ϱ sont liées à la zéro-stabilité tandis que celles du polynôme p_z sont liées à la stabilité absolue (en association avec un choix particulier de $z = h\lambda$).

Exemple : méthodes classiques à $q = 1$ pas :

Méthode	Formule	p	a_0	b_{-1}	b_0	$\varrho(r)$	$\sigma(r)$
Euler explicite	$u_{n+1} = u_n + h\varphi_n$	0	1	0	1	$r-1$	1
Euler implicite	$u_{n+1} = u_n + h\varphi_{n+1}$	0	1	1	0	$r-1$	r
Crank-Nicolson	$u_{n+1} = u_n + \frac{h}{2}(\varphi_n + \varphi_{n+1})$	0	1	$\frac{1}{2}$	$\frac{1}{2}$	$r-1$	$\frac{1}{2}(r+1)$

Exemple : méthodes à $q > 1$ pas :

Méthode	Formule	p	$\varrho(r)$	$\sigma(r)$
AB_q ou AM_q	$u_{n+1} = u_n + h\sum_{j=-1}^{p} b_j \varphi_{n-j}$	$q-1$	$r^q(r-1)$	$\sum_{j=-1}^{p} b_j r^{p-j}$
N_q ou MS_q	$u_{n+1} = u_{n-1} + h\sum_{j=-1}^{p} b_j \varphi_{n-j}$	$q-1$	$r^{q-1}(r^2-1)$	$\sum_{j=-1}^{p} b_j r^{p-j}$
BDF_2	$u_{n+1} - \frac{4}{3}u_n + \frac{1}{3}u_{n-1} = \frac{2}{3}h\varphi_{n+1}$	$q-1$	$(r-1)(r-\frac{1}{3})$	$\frac{2}{3}r^2$

10.3. Convergence d'un schéma multi-pas linéaire

Théorème (Dahlquist 1956, Lax–Richtmyer)
Une méthode multi-pas linéaire est **convergente** si et seulement si elle est **consistante**, **(zero)-stable** et **les erreurs sur les conditions initiales sont** $O(h)$**.**
La convergence est d'ordre ω si la méthode est convergente, consistante d'ordre ω, et les erreurs sur les conditions initiales sont $O(h^\omega)$.

10.3.1. Consistance

Une méthode est dite consistante (ou consistante d'ordre au moins 1) si l'erreur de troncature $\tau(h) = \max_n |\tau_n(h)|$ tend vers zéro quand h tend vers zéro, où $\tau_n(h)$ est l'erreur de troncature locale de la méthode multi-pas définie par

$$\tau_{n+1}(h) = \frac{1}{h}\left(y_{n+1} - \sum_{j=0}^{p} a_j y_{n-j} - h\sum_{j=0}^{p} b_j \varphi(t_{n-j}, y_{n-j}) - hb_{-1}\varphi(t_{n+1}, y_{n+1})\right).$$

Si, de plus, $\tau(h) = \mathcal{O}(h^\omega)$, on dit que la méthode est consistante d'ordre ω.

Pour évaluer la consistance et son ordre, nous allons introduire une nouvelle quantité $\xi(i)$.

Définition.
Pour $i = 0, 1, ...$ introduisons la quantité

$$\xi(i) = \sum_{j=0}^{p}(-j)^i a_j + i\sum_{j=-1}^{p}(-j)^{i-1} b_j,$$

ayant posé $(-j)^0 = 1$ même pour $j = 0$.

Pour bien comprendre cette définition, on peut écrire les premières valeurs de $\xi(i)$:

$$\begin{aligned}
\xi(0) &= \sum_{j=0}^{p} a_j,\\
\xi(1) &= -\sum_{j=0}^{p} j a_j + \sum_{j=-1}^{p} b_j,\\
\xi(2) &= \sum_{j=0}^{p} j^2 a_j - 2\sum_{j=-1}^{p} j b_j,\\
\xi(3) &= -\sum_{j=0}^{p} j^3 a_j + 3\sum_{j=-1}^{p} j^2 b_j,\\
&\vdots
\end{aligned}$$

Exemple avec un schéma à $q = 4$ pas (et donc $p = 3$) :

- $\xi(0) = \Big(a_0 + a_1 + a_2 + a_3\Big)$
- $\xi(1) = -\Big(a_1 + 2a_2 + 3a_3\Big) + b_{-1} + \Big(b_1 + b_2 + b_3\Big)$
- $\xi(2) = \Big(a_1 + 4a_2 + 9a_3\Big) + 2b_{-1} - 2\Big(b_1 + 2b_2 + 3b_3\Big)$
- $\xi(3) = -\Big(a_1 + 8a_2 + 27a_3\Big) + 3b_{-1} + 3\Big(b_1 + 4b_2 + 9b_3\Big)$

Il n'y a pas de difficulté à calculer ces quantités pour un schéma donné, cependant il faut faire attention à ne pas se tromper dans les calculs, notamment les signes des termes. Pour vous aider, voici un petit script Python qui calcule ces quantités de façon automatique pour un schéma donné. Il suffit de rentrer les valeurs de p et jusqu'à quel ordre vous voulez calculer les $\xi(i)$.

```
## ========================================================================
## Calcul et affichage de xi(i) pour un i donné pour un schema de pas q=p+1
## ========================================================================
def xi(i,p):
    q = p+1
    aa = sp.symbols(f'a_0:{q}')
    bb = sp.symbols(f'b_0:{q}')
    bm1 = sp.Symbol('b_{-1}')
    p = len(aa)
    sa = sum( [ (-j)**i * aa[j] for j in range(p) ] )
    sb = bm1+sum( [(-j)**(i-1) * bb[j] for j in range(1,p)] )
    ## si j=0 et i=0, sympy calcule (-j)**i =1 et on a bien a_0 dans la somme
    ## mais si j=0 et i=1 sympy ne calcule pas b_0 et il faut l'ajouter à la main
    if i==1: sb += bb[0]
    xi_rhs  = (sa).factor()+(i*sb).factor()                  ## calcul de xi(i)
    display(Markdown(f"$\\xi({i}) = {sp.latex(xi_rhs)}$") )  ## affichage de xi(i)
    ## return xi_rhs

## ========================================================================
## Calcul et affichage des xi(i) pour i=0,...,omega
## ========================================================================
def xi_cond(p,omega):
    ## j = 0...p ## coeffs du schema
    ## i = 0, ..., omega ordre du schema
    q = p+1
    display(Markdown(f"**Pour ${p=}$, on a une méthode à $q={p+1}$ pas et on affiche $ (i)$ pour
↪   $i=0,...,{omega}$**"))
    for i in range(omega+1):
        xi(i,p)
```

```
## ========================================================================
## Exemple d'utilisation
## Calcul des xi(i) pour i=0,...,omega
## ========================================================================

xi_cond( p=1, omega=3 )
```

Pour $p = 1$, on a une méthode à $q = 2$ pas et on affiche $\xi(i)$ pour $i = 0, ..., 3$

$\xi(0) = a_0 + a_1$

$\xi(1) = -a_1 + b_0 + b_1 + b_{-1}$

$\xi(2) = a_1 - 2\,(b_1 - b_{-1})$

$\xi(3) = -a_1 + 3\,(b_1 + b_{-1})$

La proposition suivante donne une condition nécessaire et suffisante pour la consistance d'une méthode multi-pas.

Proposition consistance et ordre de consistance
La méthode est **consistante** (ou, plus précisément, consistante d'ordre au moins 1) ssi $\xi(0) = \xi(1) = 1$.
La méthode est **consistante d'ordre** $\omega \geq 2$ ssi $\xi(0) = \xi(1) = \cdots = \xi(\omega) = 1$ et $\xi(\omega + 1) \neq 1$.

Remarques :

- La condition de **consistance d'ordre au moins 1** est une condition nécessaire pour la convergence d'une méthode numérique.

- La condition de **consistance d'ordre** ω signifie que **toutes les valeurs** de $\xi(i)$ sont égales à 1 pour i allant de 0 à ω, et non pas seulement $\xi(\omega)$.

Preuve. On notera $y_{n-j} = y(t_{n-j})$, $\varphi(t_{n-j}, y_{n-j}) = \varphi(t_{n-j}, y(t_{n-j})) = y'(t_{n-j})$ et $u_{n-j} \approx y(t_{n-j})$ pour $j = 0, \dots, p$.

On notera u^*_{n+1} la valeur que prendrait u_{n+1} si on remplaçait u_{n-j} par y_{n-j} et $\varphi(t_{n-j}, u_{n-j})$ par $\varphi(t_{n-j}, y_{n-j})$ dans le schéma numérique, soit

$$u^*_{n+1} = \sum_{j=0}^{p} a_j y_{n-j} + h \sum_{j=-1}^{p} b_j \varphi(t_{n-j}, y_{n-j}).$$

En développant y_{n-j} et $\varphi(t_{n-j}, y_{n-j})$ en série de Taylor autour de t_n, on trouve pour tout $n \geq p$

$$\begin{aligned} y_{n-j} &= y_n - jhy'(t_n) + O(h^2), \\ \varphi(t_{n-j}, y_{n-j}) &= y'(t_{n-j}) = y'(t_n) + O(h). \end{aligned}$$

En remplaçant ces développements dans le schéma numérique

$$\begin{aligned} \sum_{j=0}^{p} a_j y_{n-j} &= \sum_{j=0}^{p} a_j \left(y_n - jhy'(t_n) + O(h^2)\right) = \left(\sum_{j=0}^{p} a_j\right) y_n - h \left(\sum_{j=0}^{p} j a_j\right) y'(t_n) + O(h^2), \\ \sum_{j=-1}^{p} b_j \varphi(t_{n-j}, y_{n-j}) &= \sum_{j=-1}^{p} b_j \left(y'(t_n) + O(h)\right) = \left(\sum_{j=-1}^{p} b_j\right) y'(t_n) + O(h), \end{aligned}$$

on trouve

$$\begin{aligned} u^*_{n+1} &= \sum_{j=0}^{p} a_j y_{n-j} + h \sum_{j=-1}^{p} b_j \varphi(t_{n-j}, y_{n-j}) \\ &= \left(\sum_{j=0}^{p} a_j\right) y_n - h \left(\sum_{j=0}^{p} j a_j\right) y'(t_n) + h \left(\sum_{j=-1}^{p} b_j\right) y'(t_n) + O(h^2) \\ &= \left(\sum_{j=0}^{p} a_j\right) y_n + h \left(\sum_{j=0}^{p} (-j) a_j + \sum_{j=-1}^{p} b_j\right) y'(t_n) + O(h^2) \\ &= \xi(0) y_n + h\xi(1) y'(t_n) + O(h^2). \end{aligned}$$

L'erreur de troncature locale est alors donnée par

$$\begin{aligned} \tau_{n+1}(h) = \frac{y_{n+1} - u^*_{n+1}}{h} &= \frac{y_n + hy'(t_n)}{h} - \frac{\xi(0) y_n + h\xi(1) y'(t_n)}{h} + O(h) \\ &= (1 - \xi(0)) \frac{y_n}{h} + (1 - \xi(1)) y'(t_n) + O(h) \end{aligned}$$

On a $\tau_{n+1}(h) = O(h)$ pour tout $n \geq p$ ssi $\xi(0) = \xi(1) = 1$. On conclut alors que le schéma est consistant (d'ordre au moins 1) ssi $\xi(0) = \xi(1) = 1$.

Ces calculs peuvent être poursuivis pour trouver les conditions de consistance d'ordre $\omega \geq 2$. Par exemple, pour l'ordre 2 :

$$\begin{aligned} y_{n-j} &= y_n - jhy'(t_n) + \frac{j^2 h^2}{2} y''(t_n) + O(h^3), \\ \varphi(t_{n-j}, y_{n-j}) &= y'(t_{n-j}) = y'(t_n) - jhy''(t_n) + O(h^2). \end{aligned}$$

En remplaçant ces développements dans le schéma numérique

$$\begin{aligned} \sum_{j=0}^{p} a_j y_{n-j} &= \sum_{j=0}^{p} a_j \left(y_n - jhy'(t_n) + \frac{j^2 h^2}{2} y''(t_n) + O(h^3)\right) \\ &= \left(\sum_{j=0}^{p} a_j\right) y_n - h \left(\sum_{j=0}^{p} j a_j\right) y'(t_n) + \frac{h^2}{2} \left(\sum_{j=0}^{p} j^2 a_j\right) y''(t_n) + O(h^3), \\ \sum_{j=-1}^{p} b_j \varphi(t_{n-j}, y_{n-j}) &= \sum_{j=-1}^{p} b_j \left(y'(t_{n-j}) = y'(t_n) - jhy''(t_n) + O(h^2)\right) \\ &= \left(\sum_{j=-1}^{p} b_j\right) y'(t_n) - h \left(\sum_{j=-1}^{p} j b_j\right) y''(t_n) + O(h^2), \end{aligned}$$

on trouve

$$\begin{aligned} u^*_{n+1} &= \sum_{j=0}^{p} a_j y_{n-j} + h \sum_{j=-1}^{p} b_j \varphi(t_{n-j}, y_{n-j}) \\ &= \left(\sum_{j=0}^{p} a_j\right) y_n + h \left(\sum_{j=0}^{p} -j a_j\right) y'(t_n) + \frac{h^2}{2} \left(\sum_{j=0}^{p} (-j)^2 a_j\right) y''(t_n) \\ &+ h \left(\sum_{j=-1}^{p} b_j\right) y'(t_n) - h^2 \left(\sum_{j=-1}^{p} (-j) b_j\right) y''(t_n) + O(h^3) \\ &= \xi(0) y_n + \xi(1) h y'(t_n) + \xi(2) \frac{h^2}{2} y''(t_n) + O(h^3). \end{aligned}$$

L'erreur de troncature locale est alors donnée par

$$\begin{aligned} \tau_{n+1}(h) = \frac{y_{n+1} - u^*_{n+1}}{h} &= \frac{y_n + h y'(t_n) + \frac{h^2}{2} y''(t_n)}{h} - \frac{\xi(0) y_n + h \xi(1) y'(t_n) + \frac{h^2}{2} \xi(2) y''(t_n)}{h} + O(h^2) \\ &= (1 - \xi(0)) \frac{y_n}{h} + (1 - \xi(1)) y'(t_n) + (1 - \xi(2)) \frac{h}{2} y''(t_n) + O(h^2) \end{aligned}$$

On conclut que $\tau_{n+1}(h) = O(h^2)$ si et seulement si $\xi(0) = \xi(1) = \xi(2) = 1$. □

Exercice : montrer que

$$\begin{cases} \xi(0) = 1 \\ \xi(1) = 1 \end{cases} \iff \begin{cases} \varrho(1) = 0 \\ \varrho'(1) = \sigma(1) \end{cases}$$

Remarque : il s'agit de démontrer que le premier système est équivalent au second. Cela ne signifie pas que chaque équation du premier système est équivalente à l'équation correspondante du second système, ce qui n'est pas le cas.

Correction

Rappelons les définitions de $\xi(0)$, $\xi(1)$ et des polynômes ϱ et σ. Calculons ensuite ϱ' :

$$\xi(0) = \sum_{j=0}^{p} a_j \qquad \xi(1) = \sum_{j=0}^{p} -j a_j + \sum_{j=-1}^{p} b_j$$

$$\varrho(r) = r^{p+1} - \sum_{j=0}^{p} a_j r^{p-j} \qquad \varrho'(r) = (p+1) r^p - \sum_{j=0}^{p} (p-j) a_j r^{p-j-1}$$

$$\sigma(r) = b_{-1} r^{p+1} + \sum_{j=0}^{p} b_j r^{p-j} = \sum_{j=-1}^{p} b_j r^{p-j}$$

Évaluons ϱ et $\varrho' - \sigma$ en 1 :

$$\varrho(1) = 1 - \sum_{j=0}^{p} a_j = 1 - \xi(0)$$

$$\varrho'(1) - \sigma(1) = (p+1) - \sum_{j=0}^{p} (p-j) a_j - \sum_{j=-1}^{p} b_j = p \left(1 - \sum_{j=0}^{p} a_j\right) + 1 - \sum_{j=0}^{p} -j a_j - \sum_{j=-1}^{p} b_j = p(1 - \xi(0)) + 1 - \xi(1)$$

donc

$$\varrho(1) = 0 \qquad \iff \qquad 1 - \xi(0) = 0$$

et

$$\varrho'(1) - \sigma(1) = 0 \qquad \Longleftrightarrow \qquad p(1 - \xi(0)) + 1 - \xi(1) = 0$$

Bien noter que

- $\xi(0) = 1 \quad \Longleftrightarrow \quad \varrho(1) = 0$: la première équation du premier système est effectivement équivalente à la première équation du second système ;
- $\xi(1) = 1 \quad \not\Longleftrightarrow \quad \varrho'(1) = \sigma(1)$: la deuxième équation du premier système n'est pas équivalente à la deuxième équation du second système ! Ce que l'on peut dire est :
  - $\xi(0) = \xi(1) = 1 \quad \Longrightarrow \quad \varrho'(1) = \sigma(1)$: la deuxième équation du second système est une conséquence de tout le premier système ;
  - $\begin{cases} \varrho'(1) = \sigma(1) \\ \xi(0) = 1 \end{cases} \quad \Longrightarrow \quad \xi(1) = 1$: la deuxième équation du premier système est une conséquence de la deuxième équation du second système et de la première équation du premier système ;
  - $\begin{cases} \varrho'(1) = \sigma(1) \\ \xi(1) = 1 \end{cases} \quad \Longrightarrow \quad \xi(0) = 1$: la première équation du premier système est une conséquence de la première équation du second système et de la deuxième équation du premier système.

Exercice : étudier la consistance et l'ordre de consistance du schéma multi-pas linéaire AB_2

$$u_{n+1} = u_n + \frac{h}{2}\left(3\varphi_n - \varphi_{n-1}\right)$$

Correction

$p = 1 \rightsquigarrow q = 2$: c'est un schéma de la forme

$$u_{n+1} = \underbrace{a_0}_{=1} u_n + \underbrace{a_1}_{=0} u_{n-1} + h\left(\underbrace{b_0}_{=\frac{3}{2}} \varphi_n + \underbrace{b_1}_{=-\frac{1}{2}} \varphi_{n-1}\right) + h\underbrace{b_{-1}}_{=0}\varphi_{n+1}$$

Notons que $b_{-1} = 0 \rightsquigarrow$ le schéma est explicite.

On vérifie la consistance en calculant les premières deux valeurs de $ (i) $. Pour afficher $\xi(0)$ et $\xi(1)$ dans le cas général, on peut utiliser la fonction précédente :

```
xi_cond(p=1,omega=1)
```

Pour $p = 1$, on a une méthode à $q = 2$ pas et on affiche $\xi(i)$ pour $i = 0, ..., 1$

$\xi(0) = a_0 + a_1$

$\xi(1) = -a_1 + b_0 + b_1 + b_{-1}$

Maintenant on remplace les valeurs des coefficients dans les formules de $\xi(i)$ pour obtenir les valeurs de $\xi(0)$ et $\xi(1)$.

$$\xi(0) = 1 + 0 = 1$$
$$\xi(1) = -0 + \frac{3}{2} - \frac{1}{2} + 0 = 1$$

Le schéma est consistante, c'est à dire qu'il est d'ordre au moins 1.

Pour vérifier si le schéma est consistant d'ordre 2, on doit calculer $\xi(2)$ et vérifier si $\xi(2) = 1$. Pour cela, affichons $\xi(i)$ pour $i = 2$ dans le cas général :

```
xi(i=2,p=1) ## xi(2) avec p=1
```

$\xi(2) = a_1 - 2\,(b_1 - b_{-1})$

On évalue alors $\xi(2)$:

$$\xi(2) = 0 - 2\left(-\frac{1}{2} - 0\right) = 1$$

donc le schéma est consistant d'ordre au moins 2.

Pour vérifier si le schéma est consistant d'ordre au moins 3, on doit calculer $\xi(3)$ et vérifier si $\xi(3) = 1$. Pour cela, affichons $\xi(i)$ pour $i = 3$ dans le cas général :

```
xi(i=3,p=1) ## xi(3) avec p=1
```

$\xi(3) = -a_1 + 3\,(b_1 + b_{-1})$

On évalue alors $\xi(3)$:

$$\xi(3) = 0 + 3\left(-\frac{1}{2} + 0\right) \neq 1$$

donc le schéma n'et pas d'ordre 3. On conclut qu'il est consistant d'ordre exactement 2.

10.3.2. Zéro-stabilité

Proposition [zéro-stabilité et condition des racines]
Soient $r_0, r_1, \dots, r_p$ les $p+1$ racines (dans $\mathbb{C}$) du premier polynôme caractéristique $\varrho(r) = r^{p+1} - \sum_{j=0}^{p} a_j r^{p-j}$.
La méthode est **zéro-stable** ssi

$$\begin{cases} |r_j| \leq 1 \quad \text{pour tout } j = 0, \dots, p \\ r_j \text{ a multiplicité} > 1 \end{cases} \quad \Longrightarrow \quad |r_j| < 1.$$

Rappel : si le schéma est consistent, alors $r = 1$ est une racine de ϱ.

Explication :

- La première condition signifie que, dans le plan complexe, toutes les racines doivent être contenues dans le **cercle unitaire**, c'est-à-dire que leur module doit être inférieur ou égal à 1. Si une racine est située à l'extérieur du disque unité, la méthode n'est pas zéro-stable.
- La seconde condition stipule qu'une racine ayant une **multiplicité supérieure à 1** ne peut pas se trouver sur le **bord du disque unitaire**. Si une racine de multiplicité supérieure à 1 est sur le bord du disque unitaire, alors la méthode est dite **semi-stable**, mais pas zéro-stable.

Exemple : étudier la zéro-stabilité des méthodes Euler explicite, Euler implicite, Crank-Nicolson, de Adams à q pas, de Nystrom et Milne-Simpson à q pas et des méthodes BDF.

- **Méthodes à un pas** (Euler explicite, Euler implicite, Crank-Nicolson) :

  Ces méthodes ont pour polynôme caractéristique :

  $$\varrho(r) = r - 1 \quad \rightsquigarrow \quad \text{il n'y a qu'une seule racine : } r = 1$$

  Par conséquent, toutes ces méthodes sont zéro-stables.

- **Méthodes de Adams à q pas** :

  Ces méthodes ont pour polynôme caractéristique :

$$\varrho(r) = r^{p+1} - r^p = r^p(r-1) \quad \rightsquigarrow \quad \text{il y a deux racines : } \begin{cases} r = 1 & \text{a une multiplicité de 1, et } |r| = 1 \\ r = 0 & \text{a une multiplicité de } p, \text{ et } |r| = 0 < 1 \end{cases}$$

  Ces méthodes sont donc zéro-stables.
- **Méthodes de Nystrom et Milne-Simpson à q pas** :

  Pour ces méthodes, le polynôme caractéristique est :

$$\varrho(r) = r^{p+1} - r^{p-1} = r^{p-1}(r-1)(r+1) \quad \rightsquigarrow \quad \text{il y a trois racines : } \begin{cases} r = 1 & \text{a une multiplicité de 1, et } |r| = 1 \\ r = 0 & \text{a une multiplicité de } p-1, \text{ et } |r| = 0 < 1 \\ r = -1 & \text{a une multiplicité de 1, et } |r| = 1 \end{cases}$$

  Ces méthodes sont également zéro-stables.
- **Méthodes BDF** :

  Il est possible de montrer que les méthodes BDF sont zéro-stables **si et seulement si** $p \leq 5$ (ce qui correspond à $q \leq 6$, comme le montre le calcul effectué avec `sympy` ci-dessous).

10.3.3. Ordre de convergence maximale d'une méthode MPL

Il est possible de démontrer que la zero stabilité implique une restriction sur l'ordre maximal atteignable pour une méthode multi-pas. On appelle cette contrainte la première barrière de Dahlquist :

Première barrière de Dahlquist [limitation de l'ordre maximal de convergence]
L'ordre ω d'une méthode MPL à q pas convergente satisfait les inégalités suivantes :
- si $b_{-1} = 0$ (la méthode est explicite) alors $\omega \leq q$
- si $b_{-1} \neq 0$ (la méthode est implicite) on doit considérér deux cas
  - si q est impair alors $\omega \leq q + 1$
  - si q est pair alors $\omega \leq q + 2$

Remarque : la première barrière de Dalquist affirme qu'on peut augmenter arbitrairement l'ordre d'un schéma multi-pas en augmentant le nombre de pas. La convergence dépendra ensuite juste du premier polynôme caractéristique (c'est à dire de la zéro-stabilité).

Exercie : vérifier la première barrière de Dahlquist pour les méthodes Euler explicite, Euler implicite, Crank-Nicolson, de Adams-Bashforth et Adams-Moulton avec $q = 1, 2$ pas et Milne-Simpson avec $q = 2$ pas.

Schéma	Type	Ordre maximal prévu par la barrière	Ordre réel ω	Remarque
EE	Explicite	$q = 1$	1	-
AB_2	Explicite	$q = 2$	2	-
EI	Implicite, q impair	$q + 1 = 2$	1	**Non optimal** : ordre réel 1, inférieur à l'ordre maximal 2
CN	Implicite, q impair	$q + 1 = 2$	2	-
AM_2	Implicite, q pair	$q + 2 = 4$	3	**Non optimal** : ordre réel 3, inférieur à l'ordre maximal 4
MS_2	Implicite, q pair	$q + 2 = 4$	4	-



FIGURE 10.1. – Germund Dahlquist (1925-2005)

Le script suivant permet de vérifier la première barrière de Dahlquist pour une méthode donnée. Il suffit de rentrer la valeur de p et le script affiche l'ordre maximal de convergence prédit par la barrière de Dahlquist ainsi que les équations à vérifier pour chaque cas.

```
## j = 0...p ## coeffs du schema
## q = p+1 ## nb de pas
##   ## ordre de la méthode

p = 3
q = p+1
ordre_max = q+2 if q%2==0 else q+1

display(Markdown(f"**On considère une méthode à q={q} pas (et donc p={p})**"))
display(Markdown(f"**Elle est d'ordre au plus {ordre_max}. Voici les conditions à satisfaire.**"))

aa = sp.symbols(f'a_0:{q}')
bb = sp.symbols(f'b_0:{q}')
bm1 = sp.Symbol('b_{-1}')

i = 1
sa = sum( [-j*aa[j] for j in range(len(aa))] )
sb = bm1+sum( [bb[j] for j in range(len(bb))] )
display(Markdown(f"- Pour que la méthode soit consistante (c'est-à-dire d'ordre au moins {i}) elle doit
↪  vérifier les deux conditions : "))
display(Markdown( r"$\qquad\qquad " + sp.latex(sp.Eq( sum(aa).factor() , 1 ) ) + "$" ))
display(Markdown( r"$\qquad\qquad " + sp.latex(sp.Eq( (sa).factor()+(i*sb).factor() , 1 ) ) + "$" ))

for i in range(2,ordre_max+1):
    sa = sum( [(-j)**i*aa[j] for j in range(len(aa))] )
    sb = bm1+sum( [(-j)**(i-1)*bb[j] for j in range(1,len(bb))] )
    display(Markdown(f"- Pour que la méthode soit d'ordre au moins {i} elle doit vérifier, en plus des
↪  précédentes, la condition : "))
    display(Markdown( r"$\qquad\qquad " + sp.latex(sp.Eq( (sa).factor()+(i*sb).factor() , 1 ) ) + "$" ))
↪
```

On considère une méthode à q=4 pas (et donc p=3)

Elle est d'ordre au plus 6. Voici les conditions à satisfaire.

- Pour que la méthode soit consistante (c'est-à-dire d'ordre au moins 1) elle doit vérifier les deux conditions :

$$a_0 + a_1 + a_2 + a_3 = 1$$

$$-a_1 - 2a_2 - 3a_3 + b_0 + b_1 + b_2 + b_3 + b_{-1} = 1$$

- Pour que la méthode soit d'ordre au moins 2 elle doit vérifier, en plus des précédentes, la condition :

$$a_1 + 4a_2 + 9a_3 - 2\,(b_1 + 2b_2 + 3b_3 - b_{-1}) = 1$$

- Pour que la méthode soit d'ordre au moins 3 elle doit vérifier, en plus des précédentes, la condition :

$$-a_1 - 8a_2 - 27a_3 + 3\,(b_1 + 4b_2 + 9b_3 + b_{-1}) = 1$$

- Pour que la méthode soit d'ordre au moins 4 elle doit vérifier, en plus des précédentes, la condition :

$$a_1 + 16a_2 + 81a_3 - 4\,(b_1 + 8b_2 + 27b_3 - b_{-1}) = 1$$

- Pour que la méthode soit d'ordre au moins 5 elle doit vérifier, en plus des précédentes, la condition :

$$-a_1 - 32a_2 - 243a_3 + 5\,(b_1 + 16b_2 + 81b_3 + b_{-1}) = 1$$

- Pour que la méthode soit d'ordre au moins 6 elle doit vérifier, en plus des précédentes, la condition :

$$a_1 + 64a_2 + 729a_3 - 6\,(b_1 + 32b_2 + 243b_3 - b_{-1}) = 1$$

Récapitulatif des méthodes multi-pas vues jusqu'ici :

```
r = sp.symbols('r')

sc = {
    "EE = AB_1":           {"aa": [1], "bb": [1], "bm1": 0},
    "EI = AM_0 = BDF_1":   {"aa": [1], "bb": [0], "bm1": 1},
    "CN = AM_1":           {"aa": [1], "bb": [sp.Rational(1,2)], "bm1": sp.Rational(1,2)},
    "AB_2":                {"aa": [1, 0], "bb": [sp.Rational(3,2), sp.Rational(-1,2)], "bm1": 0},
    "AB_3":                {"aa": [1, 0, 0], "bb": [sp.Rational(23,12), sp.Rational(-16,12),
     ↪ sp.Rational(5,12)], "bm1": 0},
    "AB_4":                {"aa": [1, 0, 0, 0], "bb": [sp.Rational(55,24), sp.Rational(-59,24),
     ↪ sp.Rational(37,24), sp.Rational(-9,24)], "bm1": 0},
    "AB_5":                {"aa": [1, 0, 0, 0, 0], "bb": [sp.Rational(1901,720), sp.Rational(-2774,720),
     ↪ sp.Rational(2616,720), sp.Rational(-1274,720), sp.Rational(251,720)], "bm1": 0},
    "AM_2":                {"aa": [1, 0], "bb": [sp.Rational(8,12), sp.Rational(-1,12)], "bm1":
     ↪ sp.Rational(5,12)},
    "AM_3":                {"aa": [1, 0, 0], "bb": [sp.Rational(19,24), sp.Rational(-5,24),
     ↪ sp.Rational(1,24)], "bm1": sp.Rational(9,24)},
    "AM_4":                {"aa": [1, 0, 0, 0], "bb": [sp.Rational(646,720), sp.Rational(-264,720),
     ↪ sp.Rational(106,720), sp.Rational(-19,720)], "bm1": sp.Rational(251,720)},
    "BDF_2":               {"aa": [sp.Rational(4,3), sp.Rational(-1,3)], "bb": [0,0], "bm1":
     ↪ sp.Rational(2,3)},
    "BDF_3":               {"aa": [sp.Rational(18,11), sp.Rational(-9,11), sp.Rational(2,11)], "bb":
     ↪ [0,0,0], "bm1": sp.Rational(6,11)},
    "BDF_4":               {"aa": [sp.Rational(48,25), sp.Rational(-36,25), sp.Rational(16,25),
     ↪ sp.Rational(-3,25)], "bb": [0,0,0,0], "bm1": sp.Rational(12,25)},
    "BDF_5":               {"aa": [sp.Rational(300,137), sp.Rational(-300,137), sp.Rational(200,137),
     ↪ sp.Rational(-75,137), sp.Rational(12,137)], "bb": [0,0,0,0,0], "bm1": sp.Rational(60,137)},
    "BDF_6":               {"aa": [sp.Rational(120,49), sp.Rational(-150,49), sp.Rational(400,147),
     ↪ sp.Rational(-75,49), sp.Rational(24,49), sp.Rational(-10,147)], "bb": [0,0,0,0,0,0], "bm1":
     ↪ sp.Rational(20,49)},
    "MS_2":                {"aa": [0, 1], "bb": [sp.Rational(4,3), sp.Rational(1,3)], "bm1":
     ↪ sp.Rational(1,3)}
}

for schema, data in sc.items():
    # display(Markdown(f"---"))
    display(Markdown(f"**{schema}**"))

    aa = data["aa"]
    bb = data["bb"]
    bm1 = data["bm1"]
    q = len(aa)

    # 1. Type et Pas
    etat = "explicite" if bm1 == 0 else f"implicite"
```

```
    display(Markdown(f"- Schéma à $q = {q}$ pas, {etat}."))

    # 2. Calcul de l'Ordre
    def check_order(p):
        if p == 0: return sum(aa) == 1
        term_a = sum([((-j)**p * aa[j]) for j in range(q)])
        term_b = p * sum([((-j)**(p-1) * bb[j]) for j in range(q)])
        term_bm1 = p * bm1
        return sp.simplify(term_a + term_b + term_bm1) == 1

    ordre = 0
    if check_order(0):
        ordre = 1
        while check_order(ordre + 1):
            ordre += 1

    display(Markdown(f"- **Consistance** : {'Oui' if ordre >= 1 else 'Non'}"))
    display(Markdown(f"- **Ordre** : {ordre}"))

    # 3. Polynôme rho et racines
    rho_sp = r**q - sum(aa[j] * r**(q-1-j) for j in range(q))
    display(Markdown(f"- $\\varrho(r) = {sp.latex(sp.expand(rho_sp))}$"))

    # Calcul des modules des racines
    racines = sp.solve(rho_sp, r)
    # On définit une fonction qui calcule le module manuellement
    # pour éviter les problèmes d'évaluation de sp.Abs sur des
    # expressions symboliques complexes
    def get_abs_expr(root):
        re_part = sp.re(root)
        im_part = sp.im(root)
        return sp.sqrt(re_part**2 + im_part**2)
    # moduli = [sp.Abs(rac).evalf(4) for rac in racines]
    moduli = [get_abs_expr(rac).evalf(4) for rac in racines]
    display(Markdown(f"- Module de ses racines ${{|r_j|}}$ : ${sp.latex(moduli)}$"))
```

EE = AB_1

- Schéma à $q = 1$ pas, explicite.
- **Consistance** : Oui
- **Ordre** : 1
- $\varrho(r) = r - 1$
- Module de ses racines $|r_j|$: [1.0]

EI = AM_0 = BDF_1

- Schéma à $q = 1$ pas, implicite.
- **Consistance** : Oui
- **Ordre** : 1
- $\varrho(r) = r - 1$
- Module de ses racines $|r_j|$: [1.0]

CN = AM_1

- Schéma à $q = 1$ pas, implicite.
- **Consistance** : Oui
- **Ordre** : 2

- $\varrho(r) = r - 1$
- Module de ses racines $|r_j|$: [1.0]

AB_2

- Schéma à $q = 2$ pas, explicite.
- **Consistance** : Oui
- **Ordre** : 2
- $\varrho(r) = r^2 - r$
- Module de ses racines $|r_j|$: $[0,\ 1.0]$

AB_3

- Schéma à $q = 3$ pas, explicite.
- **Consistance** : Oui
- **Ordre** : 3
- $\varrho(r) = r^3 - r^2$
- Module de ses racines $|r_j|$: $[0,\ 1.0]$

AB_4

- Schéma à $q = 4$ pas, explicite.
- **Consistance** : Oui
- **Ordre** : 4
- $\varrho(r) = r^4 - r^3$
- Module de ses racines $|r_j|$: $[0,\ 1.0]$

AB_5

- Schéma à $q = 5$ pas, explicite.
- **Consistance** : Oui
- **Ordre** : 5
- $\varrho(r) = r^5 - r^4$
- Module de ses racines $|r_j|$: $[0,\ 1.0]$

AM_2

- Schéma à $q = 2$ pas, implicite.
- **Consistance** : Oui
- **Ordre** : 3
- $\varrho(r) = r^2 - r$
- Module de ses racines $|r_j|$: $[0,\ 1.0]$

AM_3

- Schéma à $q = 3$ pas, implicite.
- **Consistance** : Oui
- **Ordre** : 4
- $\varrho(r) = r^3 - r^2$
- Module de ses racines $|r_j|$: $[0,\ 1.0]$

AM_4

- Schéma à $q = 4$ pas, implicite.
- **Consistance** : Oui
- **Ordre** : 5
- $\varrho(r) = r^4 - r^3$
- Module de ses racines $|r_j|$: $[0,\ 1.0]$

BDF_2

- Schéma à $q = 2$ pas, implicite.
- **Consistance** : Oui
- **Ordre** : 2
- $\varrho(r) = r^2 - \frac{4r}{3} + \frac{1}{3}$
- Module de ses racines $|r_j|$: $[0.3333,\ 1.0]$

BDF_3

- Schéma à $q = 3$ pas, implicite.
- **Consistance** : Oui
- **Ordre** : 3
- $\varrho(r) = r^3 - \frac{18r^2}{11} + \frac{9r}{11} - \frac{2}{11}$
- Module de ses racines $|r_j|$: $[1.0,\ 0.4264,\ 0.4264]$

BDF_4

- Schéma à $q = 4$ pas, implicite.
- **Consistance** : Oui
- **Ordre** : 4
- $\varrho(r) = r^4 - \frac{48r^3}{25} + \frac{36r^2}{25} - \frac{16r}{25} + \frac{3}{25}$
- Module de ses racines $|r_j|$: $[1.0,\ 0.5609,\ 0.5609,\ 0.3815]$

BDF_5

- Schéma à $q = 5$ pas, implicite.
- **Consistance** : Oui
- **Ordre** : 5
- $\varrho(r) = r^5 - \frac{300r^4}{137} + \frac{300r^3}{137} - \frac{200r^2}{137} + \frac{75r}{137} - \frac{12}{137}$
- Module de ses racines $|r_j|$: $[1.0,\ 0.7087 - 5.0 \cdot 10^{-26}i,\ 0.4176 + 1.0 \cdot 10^{-25}i,\ 0.7087 - 5.0 \cdot 10^{-26}i,\ 0.4176 + 1.0 \cdot 10^{-25}i$

BDF_6

- Schéma à $q = 6$ pas, implicite.
- **Consistance** : Oui
- **Ordre** : 6
- $\varrho(r) = r^6 - \frac{120r^5}{49} + \frac{150r^4}{49} - \frac{400r^3}{147} + \frac{75r^2}{49} - \frac{24r}{49} + \frac{10}{147}$
- Module de ses racines $|r_j|$: $[1.0,\ 0.4061,\ 0.474,\ 0.474,\ 0.8634,\ 0.8634]$

MS_2

- Schéma à $q = 2$ pas, implicite.
- **Consistance** : Oui
- **Ordre** : 4

- $\varrho(r) = r^2 - 1$
- Module de ses racines $|r_j|$: [1.0, 1.0]

10.4. A-stabilité

Rappels
- **Problème de Dalquist** : $y'(t) = \lambda y(t)$, $y(0) = 1$ avec $\lambda \in \mathbb{C}$ tel que $\Re(\lambda) < 0$. Sa solution exacte est donnée par $y(t) = e^{\lambda t}$, et on a $\lim_{t\to+\infty} |y(t)| = 0$.
- **Schéma A-stable** : pour h fixé, un schéma numérique est A-stable si, pour tout λ vérifiant $\Re(\lambda) < 0$, la suite numérique u_n satisfait $\lim_{n\to+\infty} |u_n| = 0$. Si cette propriété est vérifiée pour tout pas de temps $h > 0$, le schéma est dit **inconditionnellement A-stable** (ou absolument stable). Si elle n'est vérifiée que sous certaines restrictions sur h, on dit que le schéma est **A-stable sous condition**.

Lorsqu'on applique une méthode MPL au problème de Dahlquist, la suite numérique u_n vérifie une relation de récurrence linéaire du type

$$(1 - (h\lambda)b_{-1})\, u_{n+1} = \sum_{j=0}^{p} (a_j + (h\lambda)b_j) u_{n-j}$$

pour $n \geq p$.

En notant z le produit $h\lambda$, cette relation peut s'écrire sous la forme :

$$(1 - zb_{-1})u_{n+1} = \sum_{j=0}^{p} (a_j + zb_j)\, u_{n-j}.$$

- Si $q = 1$ on a un schéma à un pas et l'équation récurrente s'écrit

$$u_{n+1} = q(z)u_n, \qquad q(z) = \frac{a_0 + zb_0}{1 - zb_{-1}}.$$

  Ainsi, si $|q(z)| < 1$ alors $\lim_{n\to+\infty} |u_n| = 0$.
- Si $q > 1$ on a un schéma à plusieurs pas. Pour étudier la stabilité de cette équation récurrente on verra ci-dessous qu'il faut étudier les racines du **polynôme de stabilité** (ou plutôt de la famille de polynômes paramétrée par z), défini par

$$p_z(r) = (1 - zb_{-1})r^{p+1} - \sum_{j=0}^{p} (a_j + zb_j)r^{p-j} = \varrho(r) - z\sigma(r).$$

Proposition (condition stricte des racines) :
Une méthode MPL est absolument stable si et seulement si son polynôme de A-stabilité satisfait (éventuellement sous certaines conditions sur z) la **condition stricte des racines**. Autrement dit, les racines du polynôme

$$p_z(r) = \varrho(r) - z\sigma(r)$$

doivent être strictement à l'intérieur du disque unité dans $\mathbb{C}$.

CAS RÉEL : lorsqu'on étudie la A-stabilité seulement dans $\mathbb{R}$, on peut se contenter d'étudier les racines du polynôme de stabilité pour $z = -x$ avec $x \in \mathbb{R}^+$. Cela revient à étudier les fonctions $x \mapsto r_j(x)$ racines du polynôme de stabilité. L'intervalle de A-stabilité est alors l'ensemble des valeurs de x pour lesquelles $|r_j(x)| < 1$ pour tout $j = 0, \dots, p$.

NB : contrairement à la zéro-stabilité (où $|r| = 1$ est toléré pour les racines simples), la A-stabilité exige une contraction stricte pour compenser l'amortissement de la solution exacte.

Deuxième barrière de Dahlquist : limitation sur h
- **Aucune** méthode multi-pas **explicite** n'est **inconditionnellement A-stable**.

- **Aucune** méthode multi-pas **implicite** d'ordre **supérieur à 2** n'est **inconditionnellement A-stable** dans $\mathbb{C}$.

En d'autres termes, le n'existe pas :

- **Seules** les méthodes multi-pas **implicites d'ordre 1 et 2** peuvent être **inconditionnellement A-stables** dans $\mathbb{C}$.
- **Méthodes multi-pas explicites** ou **Méthodes multi-pas implicites d'ordre supérieur à 2** $\rightsquigarrow$ toujours **conditionnellement A-stables**, c'est-à-dire qu'il existe une **valeur maximale de** h pour laquelle elles restent A-stables dans $\mathbb{C}$.

Conséquences : augmenter le nombre de pas q...

- **améliore l'ordre de consistance**
- **mais impose une restriction sur** h pour garantir l'A-stabilité
- **peut affecter la convergence** : la zéro-stabilité devient délicate pour assurer la convergence de la méthode.

Exemples : analyse de l'A-stabilité des schémas d'Adams

- **Les schémas AB$_q$** (Adams-Bashforth) **sont toujours conditionnellement A-stables** car ils sont explicites.
  - Par exemple, pour le schéma AB$_1$ (Euler explicite), on a :

$$p_z(r) = (r-1) - z$$

La seule racine est $r = 1 + z$. Pour assurer la stabilité, il faut que $|1 + z| < 1$, ce qui définit un disque ouvert de centre $(-1, 0)$ et de rayon 1 dans le plan complexe. **Dans le cas réel**, où $\varphi(t, y) = -\beta y$ avec $\beta = -\Re(\lambda)$, la condition se réécrit $0 < \beta h < 1$, ce qui correspond au résultat vu au chapitre précédent.

- **Les schémas AM$_q$** (Adams-Moulton) **avec** $q \leq 1$ **sont inconditionnellement A-stables**, car ils sont implicites et d'ordre ≤ 2.
  - **AM$_0$: Euler implicite**

$$p_z(r) = (r-1) - zr$$

La seule racine est $r = \frac{1}{1+z}$. La condition de stabilité est donc :

$$\left|\frac{1}{1+z}\right| < 1$$

Cette condition définit l'extérieur d'un disque de centre $(1, 0)$ et de rayon 1.
Comme le demi-plan $\Re(z) < 0$ est inclus dans cette région, la méthode est inconditionnellement A-stable.

  - **AM$_1$: Crank-Nicolson**

$$p_z(r) = (r-1) - z\frac{1}{2}(r+1)$$

Sa seule racine est :

$$r = \frac{2+z}{2-z}$$

Pour la stabilité, on impose :

$$\left|\frac{2+z}{2-z}\right| < 1$$

Ce qui revient à exiger $\Re(z) < 0$.
Comme $\mathbb{C}^-$ est entièrement inclus dans cette région, la méthode est inconditionnellement A-stable.

- **Les schémas AM$_q$** (Adams-Moulton) **avec** $q > 1$ **sont conditionnellement A-stables**, car ils sont implicites et d'ordre > 2.

Les images affichées ci-dessous montrent les régions de A-stabilité des quelques méthodes vues en précédence et sont issues du livre
A. Quarteroni, F. Saleri, P. Gervasio *Calcul Scientifique : Cours, exercices corrigés et illustrations en Matlab et Octave.*

Régions de A-stabilité des méthodes d'Adams-Bashforth et d'Adams-Moulton

Les figures ci-dessous illustrent les régions de stabilité absolue des méthodes :

- **À gauche** : Adams-Bashforth (AB), qui sont des schémas explicites.
- **À droite** : Adams-Moulton (AM), qui sont des schémas implicites.

On observe que :

- Ces régions **sont bornées** et **diminuent à mesure que l'ordre augmente**.
- De plus, **toutes les méthodes explicites et les méthodes implicites d'ordre supérieur à 2 ne sont pas inconditionnellement A-stables**.
- Elles ne conviennent donc **pas aux problèmes nécessitant une intégration sur de très longs intervalles de temps**, car elles imposent une contrainte sur la taille du pas de temps h.

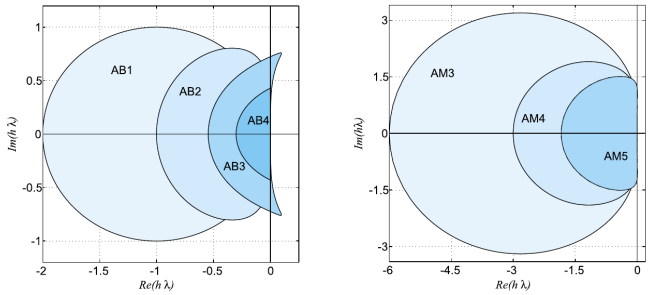


FIGURE 10.2. – A-stabilité Adams

Régions de A-stabilité des méthodes BDF

L'illustration ci-dessous montre les régions de stabilité absolue des méthodes **Backward Differentiation Formula (BDF)** pour $q \leq 4$ qui sont des schémas implicites.

Contrairement aux méthodes d'Adams :

- **Les régions de stabilité des BDF sont conditionnellement A-stable sur $\mathbb{C}$ mais incluent toujours l'axe réel négatif**, ce qui leur confère une **inconditionnelle A-stabilité sur $\mathbb{R}$**.
- Ces méthodes sont donc **particulièrement adaptées aux problèmes nécessitant une intégration sur des intervalles de temps très longs**, voire infinis.
- Elles sont aussi **privilégiées pour la résolution des problèmes raides**, car, avec un changement de variable, cela revient à résoudre un problème de Cauchy sur un intervalle de temps grand.

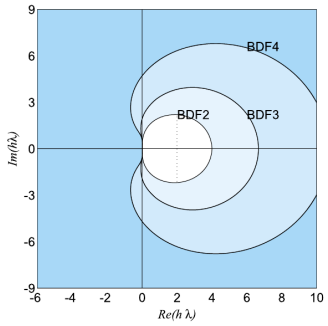


FIGURE 10.3. – A-stabilité BDF

10.5. Exercice de synthèse : consistance, zéro-stabilité et A-stabilité des schémas EE, EI, CN

Pour les méthodes d'Euler explicite, Euler implicite et Cranck-Nicolson :

1. Trouver les coefficients $p \geq 0$, $\{a_j\}_{j=0}^{p}$ et $\{b_j\}_{j=-1}^{p}$ tels que

$$u_{n+1} = \sum_{j=0}^{p} (a_j u_{n-j} + h b_j \varphi_{n-j}) + h b_{-1} \varphi_{n+1}, \qquad n = p, p+1, \dots, N-1$$

2. Calculer les $p+1$ racines $\{r_j\}_{j=0}^{p}$ du premier polynôme caractéristique

$$\varrho(r) = r^{p+1} - \sum_{j=0}^{p} a_j r^{p-j}.$$

   En déduire que la méthode est zéro-stable, i.e. que

$$\begin{cases} |r_j| \leq 1 \quad \text{pour tout } j = 0, \dots, p \\ r_j \text{ a multiplicité} > 1 \end{cases} \implies |r_j| < 1.$$

3. Donner l'ordre de consistance maximal théorique puis étudier l'ordre de consistance réel, autrement dit trouver le plus grand $\omega \geq 1$ tel que $\xi(0) = \xi(1) = \cdots = \xi(\omega) = 1$.
4. Conclure sur la convergence de la méthode et son ordre de convergence.
5. Étudier la A-stabilité

Il s'agit de trois méthodes à $q = 1$ pas, donc $p = 0$.

Commençons par l'**étude de la convergence** en étudiant la consistance (et son ordre) et la zéro-stabilité.

Selon la première barrière de Dahlquist, si la méthode est explicite l'ordre de convergence maximal est 2 (car q impair), si la méthode est explicite l'ordre maximal est 1. Pour pouvoir étudier l'ordre de consistance, commençons par afficher les $\xi(i)$ pour $i = 0, 1, 2$:

```
xi_cond(p=0, omega=2)
```

Pour $p = 0$, on a une méthode à $q = 1$ pas et on affiche $\xi(i)$ pour $i = 0, ..., 2$

$\xi(0) = a_0$

$\xi(1) = b_0 + b_{-1}$

$\xi(2) = 2b_{-1}$

Résumons donc les quantités indiquées dans le tableau suivant :

	EE	**EI**	**CN**
Méthode : $u_{n+1} =$	$u_n + h\varphi_n$	$u_n + h\varphi_{n+1}$	$u_n + \frac{h}{2}(\varphi_{n+1} + \varphi_n)$
Coefficients : $p = 0$ et	$a_0 = 1, b_{-1} = 0, b_0 = 1$	$a_0 = 1, b_{-1} = 1, b_0 = 0$	$a_0 = 1, b_{-1} = \frac{1}{2}, b_0 = \frac{1}{2}$
Polynôme caractéristique : ϱ	$\varrho(r) = r - 1$	$\varrho(r) = r - 1$	$\varrho(r) = r - 1$
Racines :	$r_0 = 1$	$r_0 = 1$	$r_0 = 1$
Zéro-stabilité :	oui, $\lvert r_0\rvert = 1$ et mult. 1	oui, $\lvert r_0\rvert = 1$ et mult. 1	oui, $\lvert r_0\rvert = 1$ et mult. 1
Consistance :	oui, $\xi(0) = 1, \xi(1) = 1$	oui, $\xi(0) = 1, \xi(1) = 1$	oui, $\xi(0) = 1, \xi(1) = 1$
Ordre maximal (Dahlquist) :	$\omega \leq 1$	$\omega \leq 2$	$\omega \leq 2$
Ordre réel : ω	$\omega = 1$	$\omega = 1$ car $\xi(2) = 2 \neq 1$	$\omega = 2$ car $\xi(2) = 1$

Passons maintenant à l'**étude de la A-stabilité**. Pour cela, considérons le toy-model : soit $\lambda \in \mathbb{C}$ un nombre complexe tel que $\Re(\lambda) = -\beta < 0$ et considérons le problème de Cauchy

$$\begin{cases} y'(t) = \lambda y(t), & \text{pour } t > 0, \\ y(0) = 1. \end{cases}$$

Posons $z = \lambda h$ et calculons les racines du polynôme caractéristique $p_z(r) = \varrho(r) - z\sigma(r)$:

	EE	**EI**	**CN**
Schéma appliqué au toy-model	$u_n = (1 + z)^n$	$u_n = \left(\frac{1}{1+z}\right)^n$	$u_n = \left(\frac{2+z}{2-z}\right)^n$
Condition pour $\lvert u_n\rvert \xrightarrow[n\to\infty]{} 0$	$\lvert 1 + z\rvert < 1$	$\left\lvert\frac{1}{1+z}\right\rvert < 1$	$\left\lvert\frac{2+z}{2-z}\right\rvert < 1$
A-stabilité ?	**Uniquement** si $-2 < \lvert z\rvert < 0$	**Oui** (car $\Re(\lambda) < 0$)	**Oui** (car $\Re(\lambda) < 0$)

Conclusion :

- **Euler explicite** : la méthode est consistante d'ordre $\omega = 1$ qui est l'ordre maximal prévu par la barrière de Dahlquist (schéma explicite). Le schéma étant aussi zéro-stable, il est donc convergent d'ordre 1. Concerant la A-stabilité, il est nécessaire que $-2 < \lvert z\rvert < 0$ pour que la méthode soit A-stable. Dans $\mathbb{R}$, cela implique $0 < \beta h < 2$.
- **Euler implicite** : la méthode est consistante d'ordre $\omega = 1$ qui est inférieur à l'ordre maximal prévu par la barrière de Dahlquist (schéma implicite). Le schéma étant aussi zéro-stable, il est donc convergent d'ordre 1. Concernant la A-stabilité, la méthode est inconditionellement A-stable.

- **Cranck-Nicolson** : la méthode est consistante d'ordre $\omega = 2$ qui coïncide avec l'ordre maximal prévu par la barrière de Dahlquist (schéma implicite, nombre de pas impair). Le schéma étant aussi zéro-stable, il est donc convergent d'ordre 2. Concernant la A-stabilité, la méthode est inconditionellement A-stable.

10.6. Propriétés des schémas Prédicteur-Correcteur basés sur les méthodes multi-pas

Proposition : ordre de convergence d'un schéma PC
Si on note
- $\omega_{[P]}$ l'ordre du predictor (schéma explicite)
- $\omega_{[C]}$ l'ordre du corrector (schéma implicite)

alors l'ordre du schéma *predictor-corrector* est donné par

$$\omega_{[PC]} = \min\Big\{\omega_{[C]}, \omega_{[P]} + 1\Big\}$$

Ainsi, pour obtenir une méthode d'ordre ω, on peut choisir un prédicteur d'ordre $\omega - 1$ et un correcteur d'ordre ω (par exemple, deux méthodes d'Adam ayant le même nombre de pas).

Zéro-stabilité : on peut démontrer que la zéro-stabilité du prédicteur n'influe pas sur la zéro-stabilité de la méthode prédicteur-correcteur. Il est même possible d'utiliser un prédicteur instable.

A-stabilité : les méthodes predictor-corrector sont soumises à une condition de A-stabilité, qui est typiquement celle du prédicteur. Elles ne sont donc pas adaptées à la résolution de problèmes de Cauchy sur des intervalles non bornés ou de problèmes stiff.

10.7. Méthodes cycliques composites : comment surmonter les limites de Dahlquist

On peut surmonter les limites imposées par les barrières de Dahlquist en combinant plusieurs méthodes multi-pas. Par exemple, les deux méthodes suivantes à trois pas

$$\begin{aligned} u_{n+1} &= -\frac{8}{11}u_n + \frac{19}{11}u_{n-1} + \frac{h}{30}(30\varphi_{n+1} + 57\varphi_n + 24\varphi_{n-1} - \varphi_{n-2}) \\ u_{n+1} &= \frac{449}{240}u_n - \frac{19}{30}u_{n-1} - \frac{361}{240}u_{n-2} + \frac{h}{720}(251\varphi_{n+1} + 456\varphi_n - 1347\varphi_{n-1} - 350\varphi_{n-2}) \end{aligned}$$

sont d'ordre cinq, mais sont instables. Cependant, en les combinant comme suit

$$u_{n+1} = \begin{cases} -\frac{8}{11}u_n + \frac{19}{11}u_{n-1} + \frac{h}{30}(30\varphi_{n+1} + 57\varphi_n + 24\varphi_{n-1} - \varphi_{n-2}) & \text{si } n \text{ est pair} \\ \frac{449}{240}u_n - \frac{19}{30}u_{n-1} - \frac{361}{240}u_{n-2} + \frac{h}{720}(251\varphi_{n+1} + 456\varphi_n - 1347\varphi_{n-1} - 350\varphi_{n-2}) & \text{sinon.} \end{cases}$$

on obtient une méthode A-stable à trois pas d'ordre cinq.

10.8. Annexe : comment tracer les régions de A-stabilité ?

Soit $z = h\lambda \in \mathbb{C}^-$. La **région de stabilité absolue** $\mathcal{A}$ est l'ensemble des points z pour lesquels toutes les racines r_j du polynôme caractéristique $p_z(r) = \varrho(r) - z\,\sigma(r)$ vérifient $|r_j| < 1$. L'**intervalle de $\mathcal{A}$-stabilité réelle** est l'intersection $\mathcal{A} \cap \mathbb{R}^-$.

Il est en général difficile de déterminer directement $\mathcal{A}$ car il faut tester, pour chaque z, si le critère $|r_j| < 1$ est satisfait pour toutes les racines. Il est plus aisé de déterminer la **frontière** $\partial\mathcal{A}$. Sur cette frontière, au moins une

racine a un module exactement égal à 1 : on peut l'écrire $r = e^{i\theta}$, $\theta \in [0, 2\pi]$. En substituant dans $p_z(r) = 0$, on obtient :

$$\varrho(e^{i\theta}) - z\,\sigma(e^{i\theta}) = 0 \quad \rightsquigarrow \quad z(\theta) = \frac{\varrho(e^{i\theta})}{\sigma(e^{i\theta})}.$$

Cette relation définit $\partial\mathcal{A}$ comme une **courbe paramétrée** dans le plan complexe. Pour en obtenir une approximation numérique, il suffit d'évaluer ce quotient sur une discrétisation de $[0, 2\pi]$ (en vérifiant au préalable que $\sigma(e^{i\theta}) \neq 0$). La courbe ainsi tracée localise les points où *une* racine est de module 1, mais ne garantit pas que les autres racines restent dans le disque unité. C'est pourquoi la frontière est complétée par un remplissage coloré (test sur grille) afin d'identifier la véritable zone de stabilité.

```
#   Helpers symboliques

def _build_rho_sigma(schema):
    """Construit  et  symboliquement (SymPy)."""
    r = sp.Symbol('r')
    aa, bb, bm1 = schema["aa"], schema["bb"], schema["bm1"]
    p = len(aa) - 1
    rho   = r**(p+1) - sum(aa[j] * r**(p-j) for j in range(p+1))
    sigma = bm1 * r**(p+1) + sum(bb[j] * r**(p-j) for j in range(p+1))
    return rho, sigma


def polyCar(schema):
    """Affiche  et  (méthode explicite : bm1 = 0)."""
    rho, sigma = _build_rho_sigma(schema)
    display(Math(r'\varrho(r) = ' + sp.latex(rho)))
    display(Math(r'\sigma(r)  = ' + sp.latex(sigma)))


def polyCarIMPL(schema):
    """
    Affiche ,  et p_z(r) =  (r) - z· (r) avec z symbolique.
    Les termes sont regroupés en puissances de r pour plus de lisibilité.
    """
    r, z = sp.symbols('r z')
    rho, sigma = _build_rho_sigma(schema)
    pz = sp.collect(sp.expand(rho - z * sigma), r)   # ← collecte en r
    display(Math(r'\varrho(r) = '                          + sp.latex(rho)))
    display(Math(r'\sigma(r)  = '                          + sp.latex(sigma)))
    display(Math(r'p_z(r) = \varrho(r)-z\,\sigma(r) = '   + sp.latex(pz)))


#   Fonction de stabilité

def stabilityFunction(z, schema):
    """
    Retourne max|r_j| pour le polynôme caractéristique en z.
    Valeur > 1    z est hors de la région de stabilité absolue.
    Ne modifie pas schema.
    """
    aa  = np.array(schema["aa"],  dtype=float)
    bb  = np.array(schema["bb"],  dtype=float)
    bm1 = float(schema["bm1"])
    # Coefficients de p_z : [1-z·bm1, -(aa+z·bb)]  (degré décroissant)
    coeffs = np.concatenate([[1 - z * bm1], -(aa + z * bb)])
    return float(np.max(np.abs(np.roots(coeffs))))


#   Analyse de stabilité absolue

def Astabilite(schema):
    aa  = np.array(schema["aa"],  dtype=float)
    bb  = np.array(schema["bb"],  dtype=float)
```

```
    bm1 = float(schema["bm1"])
    p   = len(aa) - 1

    #   Frontière paramétrée z( ) =  (e^i ) /  (e^i )
    theta   = np.linspace(0, 2 * np.pi, 201)
    c       = np.exp(1j * theta)
    rho_c   = c**(p+1) - np.polyval(aa, c)
    sigma_c = bm1 * c**(p+1) + np.polyval(bb, c)
    mask    = np.abs(sigma_c) > 1e-14
    zz      = np.where(mask, rho_c / sigma_c, np.nan + 1j * np.nan)
    zz_fin  = zz[np.isfinite(zz)]

    #   Limites de la grille
    x_min = min(zz_fin.real.min(), -0.5)
    # ↓ On inclut Re(z) > 0 pour capturer les frontières qui y résident
    #   (méthodes A-stables :  A est entièrement dans le demi-plan droit)
    x_max = max(zz_fin.real.max() * 1.1, 0.5)
    y_ext = max(np.abs(zz_fin.imag).max(), 0.5) * 1.1
    Lx, Ly = (x_min, x_max), (-y_ext, y_ext)

    x = np.linspace(Lx[0], Lx[1], 301)
    y = np.linspace(Ly[0], Ly[1], 301)
    X, Y = np.meshgrid(x, y)
    _stab = np.vectorize(lambda xv, yv: stabilityFunction(xv + 1j * yv, schema))
    Z = _stab(X, Y)

    #   Tracé
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(11, 4.5))

    # --- Frontière paramétrée ---
    ax1.plot(zz.real, zz.imag, 'r', lw=1.5)
    ax1.axvline(0, color='k', lw=0.8)
    ax1.axhline(0, color='k', lw=0.8)
    ax1.set_aspect('equal')
    ax1.grid(True, alpha=0.4)
    ax1.set_xlabel(r'$\Re(h\lambda)$')
    ax1.set_ylabel(r'$\Im(h\lambda)$')
    ax1.set_title(f"Frontière $\\partial\\mathcal{{A}}$ - {schema['Nom']}")

    # --- Zone de stabilité remplie ---
    ax2.contourf(X, Y, Z, levels=[0, 1], colors=[[1, 0.5, 0.8]])

    # ↓ Correction : on ne trace le contour que si le niveau 1 est dans la plage
    if Z.min() <= 1.0 <= Z.max():
        ax2.contour(X, Y, Z, levels=[1], colors='k', linewidths=1.2)
    else:
        # La frontière est hors de la vue → méthode A-stable sur cette région
        ax2.text(0.97, 0.97, r"$\mathcal{A}$-stable : $\partial\mathcal{A} \notin \mathbb{C}^-$",
                 transform=ax2.transAxes, ha='right', va='top',
                 fontsize=8, color='gray',
                 bbox=dict(boxstyle='round,pad=0.3', fc='white', alpha=0.7))

    ax2.axvline(0, color='k', lw=2)
    ax2.grid(True, alpha=0.4)
    ax2.set_xlabel(r'$\Re(h\lambda)$')
    ax2.set_ylabel(r'$\Im(h\lambda)$')
    ax2.set_title(fr"Zone de stabilité absolue - {schema['Nom']}")

    # --- Intervalle de A-stabilité sur   ---
    # Recherche sur un domaine large, indépendant de la grille graphique
    X_SEARCH = 7.0
    x_real = np.linspace(-X_SEARCH, 0, 401)
    stab_r = np.array([stabilityFunction(xv + 0j, schema) for xv in x_real])
```

```
    stable = x_real[stab_r <= 1.0 + 1e-10]

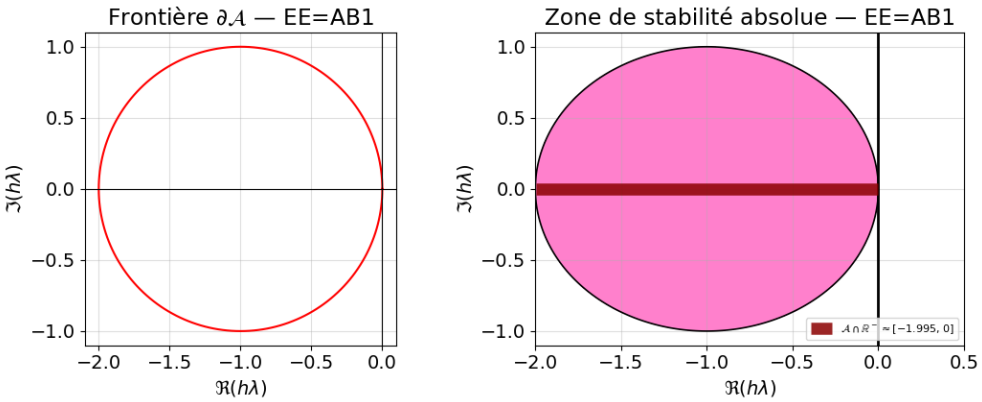
    if stable.size:
        x_lo = stable.min()
        # Non borné ssi le segment atteint la borne gauche de la recherche
        unbounded = x_lo <= -X_SEARCH + (X_SEARCH / len(x_real))
        label = (r"$\mathcal{A}\cap\mathbb{R}^-=(-\infty,\,0]$"
                 if unbounded
                 else fr"$\mathcal{{A}}\cap\mathbb{{R}}^-\approx[{x_lo:.3f},\,0]$")
        # On ne trace que la partie visible sur le graphique
        visible = stable[stable >= Lx[0]]
        if visible.size:
            ax2.plot(visible, np.zeros_like(visible),
                     color='darkred', lw=9, solid_capstyle='butt',
                     alpha=0.85, label=label)
        ax2.legend(fontsize=8, loc='lower right')

    plt.tight_layout()
    plt.show()
```

```
EE = { "Nom":"EE=AB1", "aa":[1], "bb":[1], "bm1":0  }
polyCar(EE)
Astabilite(EE)
```

$\varrho(r) = r - 1$

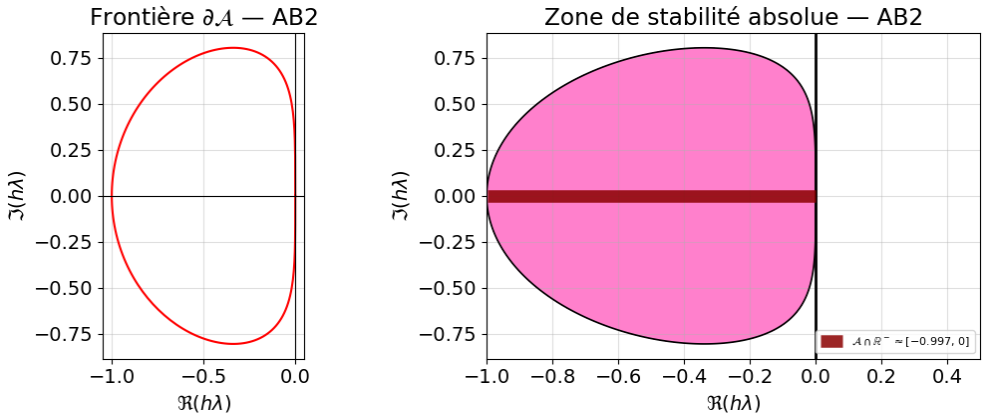
$\sigma(r) = 1$




```
AB2 = { "Nom":"AB2", "aa":[1,0], "bb":[sp.Rational(3,2),-sp.Rational(1,2)], "bm1":0  }
polyCar(AB2)
Astabilite(AB2)
```

$\varrho(r) = r^2 - r$

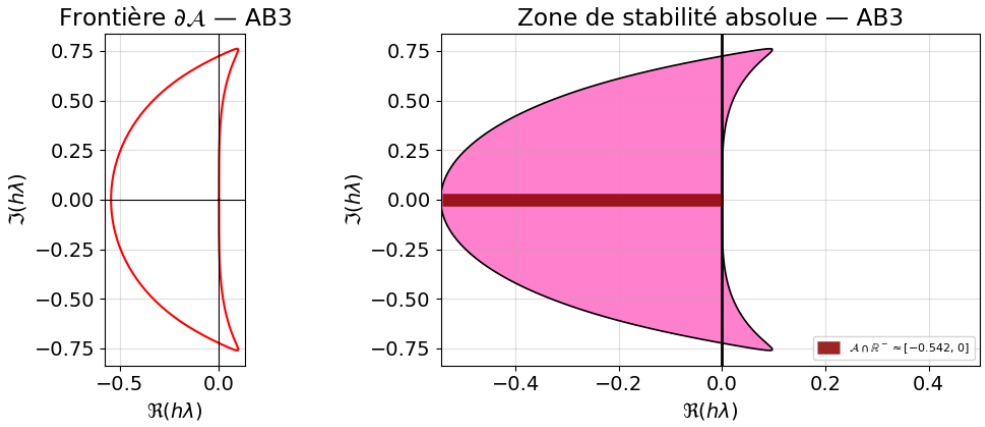
$\sigma(r) = \dfrac{3r}{2} - \dfrac{1}{2}$




```
AB3 = { "Nom":"AB3", "aa":[1,0,0], "bb":[sp.Rational(23,12),-sp.Rational(16,12), sp.Rational(5,12)],
 ↪  "bm1":0  }
polyCar(AB3)
Astabilite(AB3)
```

$$\varrho(r) = r^3 - r^2$$

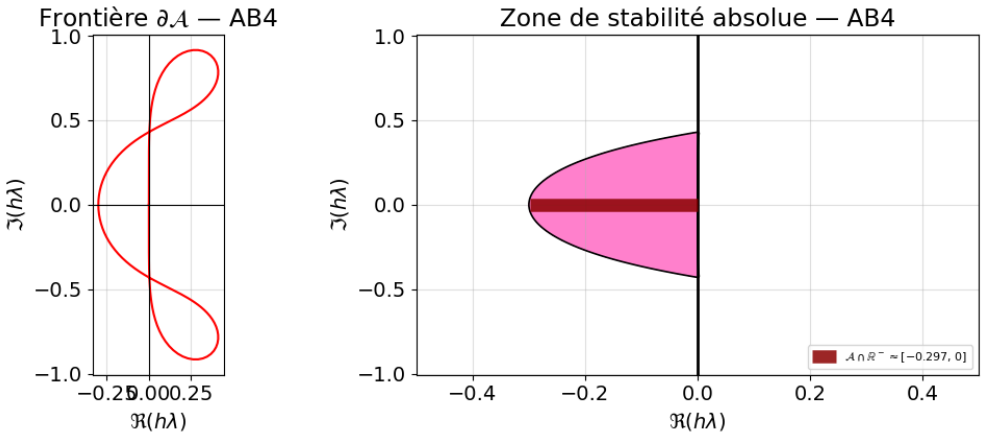
$$\sigma(r) = \frac{23r^2}{12} - \frac{4r}{3} + \frac{5}{12}$$




```
AB4 = { "Nom":"AB4", "aa":[1,0,0,0], "bb":[sp.Rational(55,24),-sp.Rational(59,24), sp.Rational(37,24),
 ↪  -sp.Rational(9,24)], "bm1":0  }
polyCar(AB4)
Astabilite(AB4)
```

$$\varrho(r) = r^4 - r^3$$

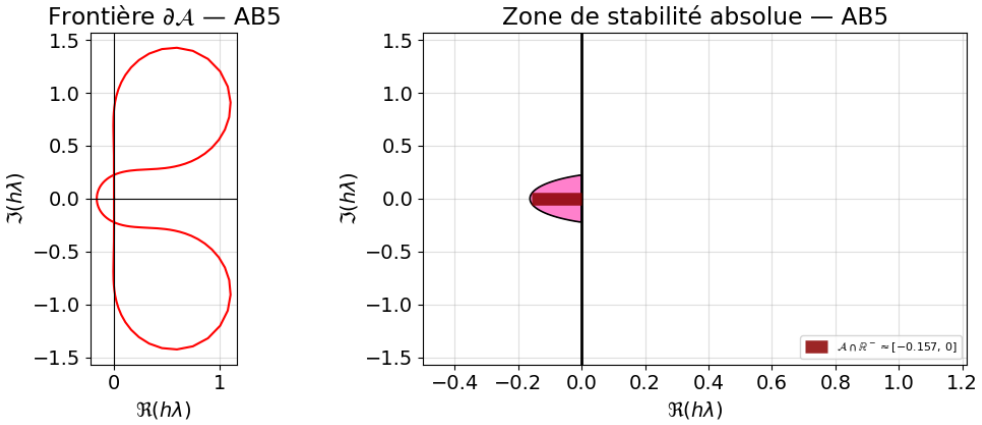
$$\sigma(r) = \frac{55r^3}{24} - \frac{59r^2}{24} + \frac{37r}{24} - \frac{3}{8}$$




```
AB5 = { "Nom":"AB5", "aa":[1,0,0,0,0], "bb":[sp.Rational(1901,720), -sp.Rational(1387,360),
 ↪  sp.Rational(109,30), -sp.Rational(637,360), sp.Rational(251,720)], "bm1":0  }
polyCar(AB5)
Astabilite(AB5)
```

$$\varrho(r) = r^5 - r^4$$

$$\sigma(r) = \frac{1901r^4}{720} - \frac{1387r^3}{360} + \frac{109r^2}{30} - \frac{637r}{360} + \frac{251}{720}$$

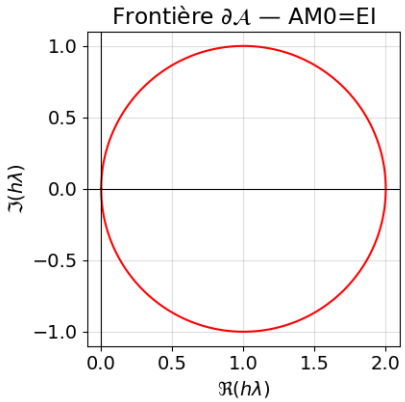


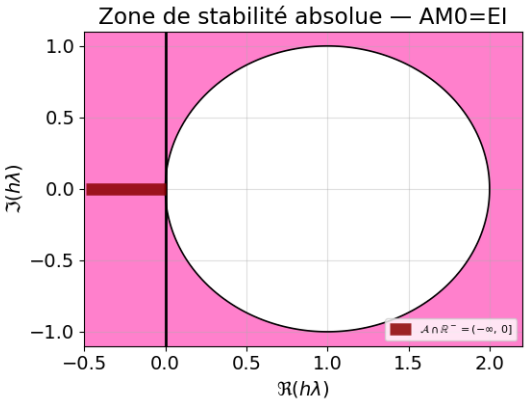

```
AM0 = { "Nom":"AM0=EI", "aa":[1], "bb":[0], "bm1": 1  }
polyCarIMPL(AM0)
Astabilite(AM0)
```

$$\varrho(r) = r - 1$$

$$\sigma(r) = r$$

$$p_z(r) = \varrho(r) - z\,\sigma(r) = r\,(1 - z) - 1$$



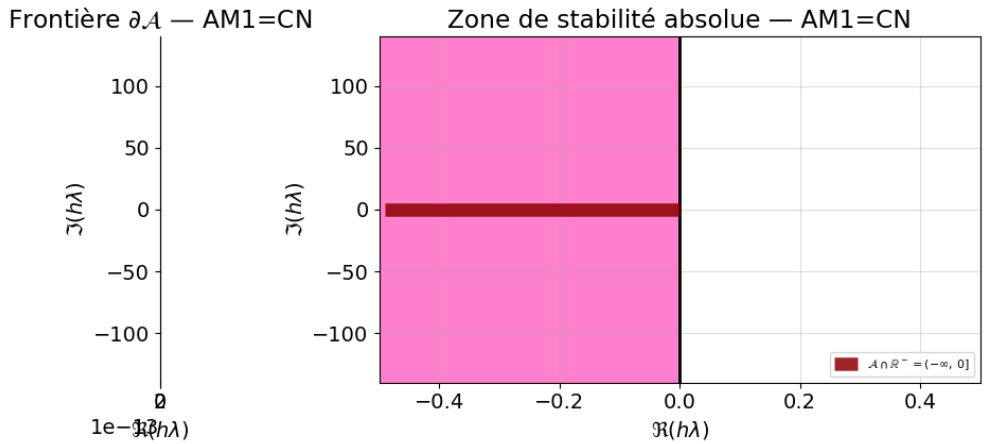



```
AM1 = { "Nom":"AM1=CN",  "aa":[1], "bb":[sp.Rational(1,2)], "bm1": sp.Rational(1,2)  }
polyCarIMPL(AM1)
Astabilite(AM1)
```

$$\varrho(r) = r - 1$$

$$\sigma(r) = \frac{r}{2} + \frac{1}{2}$$

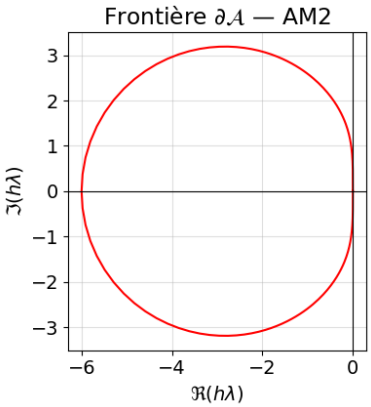
$$p_z(r) = \varrho(r) - z\,\sigma(r) = r\left(1 - \frac{z}{2}\right) - \frac{z}{2} - 1$$

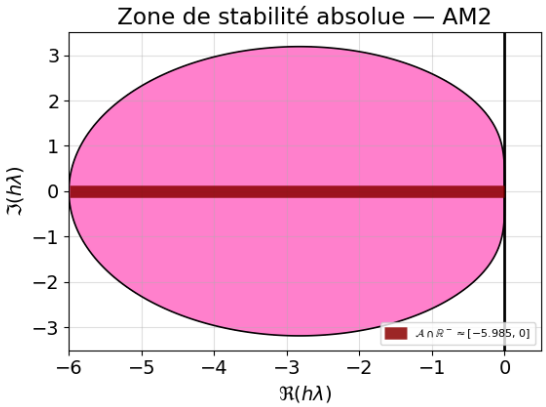



```
AM2 = { "Nom":"AM2", "aa":[1,0], "bb":[sp.Rational(8,12),-sp.Rational(1,12)], "bm1":sp.Rational(5,12) }
polyCar(AM2)
Astabilite(AM2)
```

$$\varrho(r) = r^2 - r$$

$$\sigma(r) = \frac{5r^2}{12} + \frac{2r}{3} - \frac{1}{12}$$

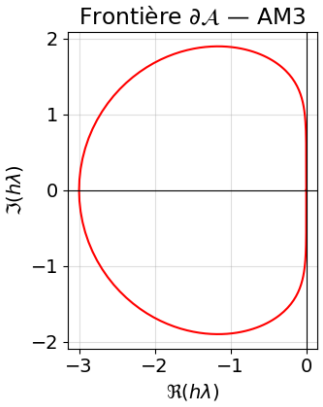


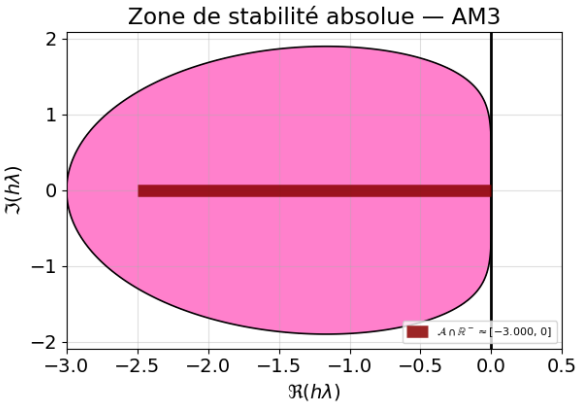



```
AM3 = { "Nom":"AM3", "aa":[1,0,0], "bb":[sp.Rational(19,24),-sp.Rational(5,24),sp.Rational(1,24)],
 ↪  "bm1":sp.Rational(9,24)  }
polyCar(AM3)
Astabilite(AM3)
```

$$\varrho(r) = r^3 - r^2$$

$$\sigma(r) = \frac{3r^3}{8} + \frac{19r^2}{24} - \frac{5r}{24} + \frac{1}{24}$$

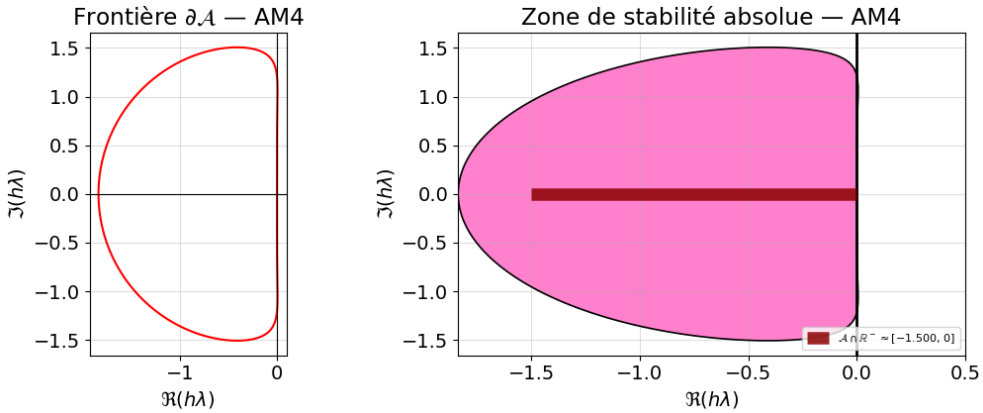





```
AM4 = { "Nom":"AM4", "aa":[1,0,0,0], "bb":[sp.Rational(646,720), sp.Rational(-264,720),
 ↪  sp.Rational(106,720), sp.Rational(-19,720)], "bm1":sp.Rational(251,720)  }
polyCar(AM4)
Astabilite(AM4)
```

$$\varrho(r) = r^4 - r^3$$

$$\sigma(r) = \frac{251r^4}{720} + \frac{323r^3}{360} - \frac{11r^2}{30} + \frac{53r}{360} - \frac{19}{720}$$




```
MS2 = { "Nom":"MS2", "aa":[0,1], "bb":[sp.Rational(4,3),sp.Rational(1,3)], "bm1":sp.Rational(1,3) }
polyCar(MS2)
##Astabilite(MS2)
# Jamais A-stable (voir le TD sur les méthodes de multi-step)
```

$$\varrho(r) = r^2 - 1$$

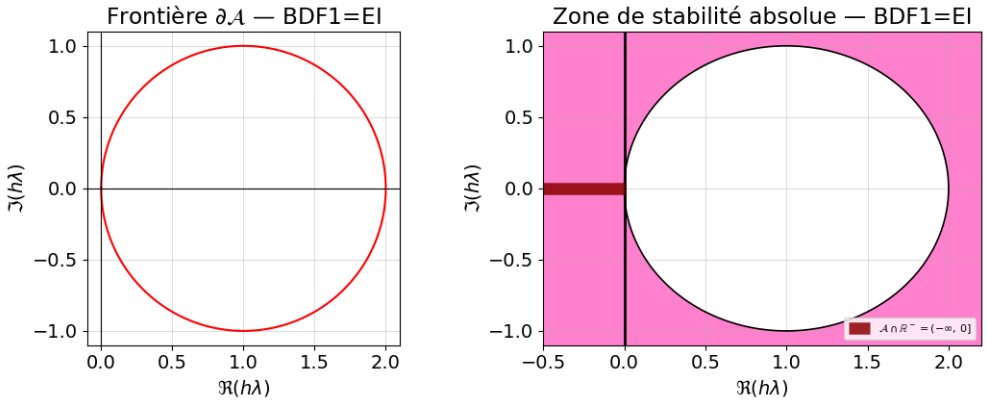
$$\sigma(r) = \frac{r^2}{3} + \frac{4r}{3} + \frac{1}{3}$$

```
BDF1 = { "Nom":"BDF1=EI", "aa":[1], "bb":[0], "bm1":1  }
polyCarIMPL(BDF1)
Astabilite(BDF1)
```

$$\varrho(r) = r - 1$$

$$\sigma(r) = r$$

$$p_z(r) = \varrho(r) - z\,\sigma(r) = r\,(1 - z) - 1$$

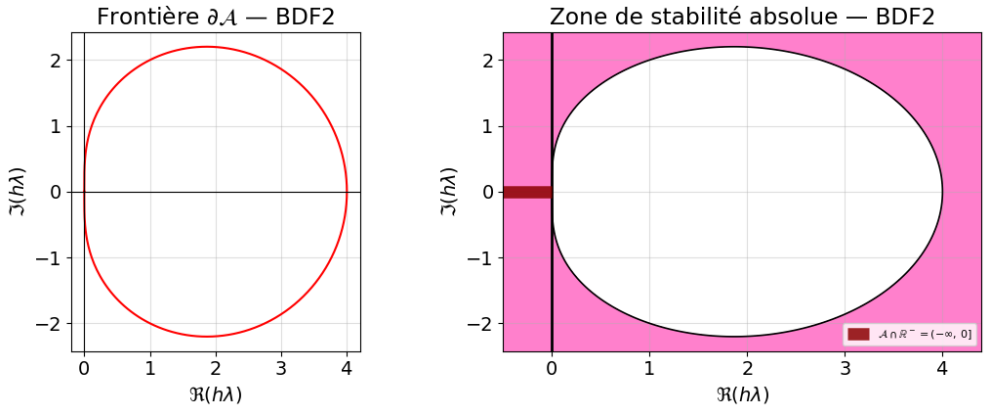



```
BDF2 = { "Nom":"BDF2", "aa":[sp.Rational(4,3),-sp.Rational(1,3)],
                       "bb":[0,0], "bm1":sp.Rational(2,3) }
polyCarIMPL(BDF2)
Astabilite(BDF2)
```

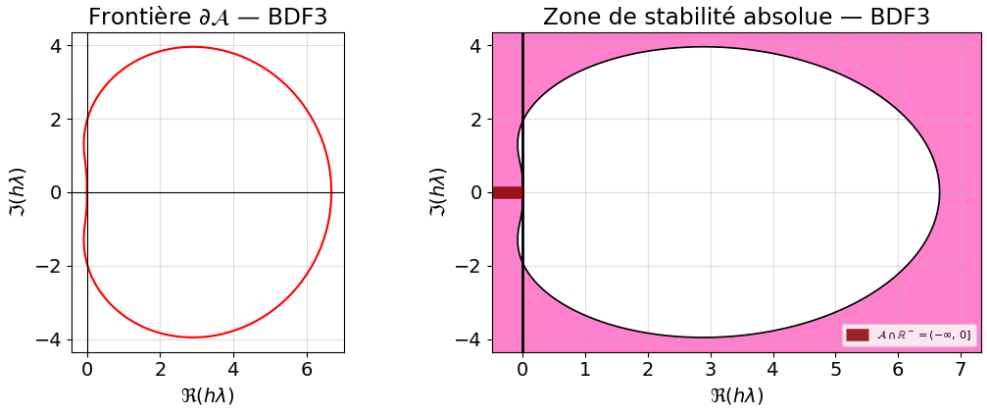
$$\varrho(r) = r^2 - \frac{4r}{3} + \frac{1}{3}$$

$$\sigma(r) = \frac{2r^2}{3}$$

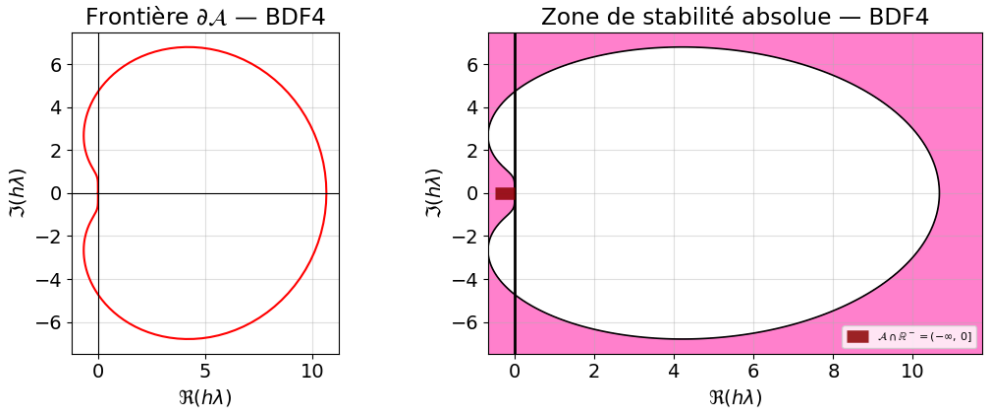
$$p_z(r) = \varrho(r) - z\,\sigma(r) = r^2 \cdot \left(1 - \frac{2z}{3}\right) - \frac{4r}{3} + \frac{1}{3}$$




```
BDF3 = { "Nom":"BDF3", "aa":[sp.Rational(18,11),-sp.Rational(9,11),sp.Rational(2,11)],
                       "bb":[0,0,0], "bm1":sp.Rational(6,11)  }
##polyCarIMPL(BDF3)
Astabilite(BDF3)
```

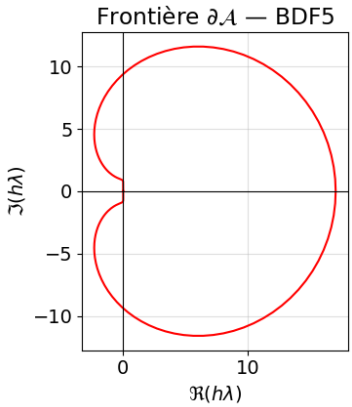


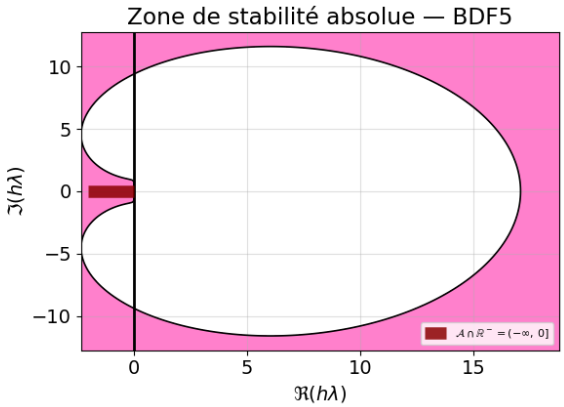

```
BDF4 = { "Nom":"BDF4",
 ↪  "aa":[sp.Rational(48,25),-sp.Rational(36,25),sp.Rational(16,25),-sp.Rational(3,25)],
                       "bb":[0,0,0,0], "bm1":sp.Rational(12,25)  }
## polyCarIMPL(BDF4)
Astabilite(BDF4)
```



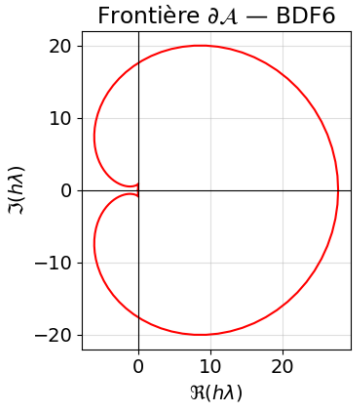

```
BDF5 = { "Nom":"BDF5",
 ↪  "aa":[sp.Rational(300,137),-sp.Rational(300,137),sp.Rational(200,137),-sp.Rational(75,137),sp.Rational(12,137)],
 ↪
                       "bb":[0,0,0,0,0], "bm1":sp.Rational(60,137) }
```

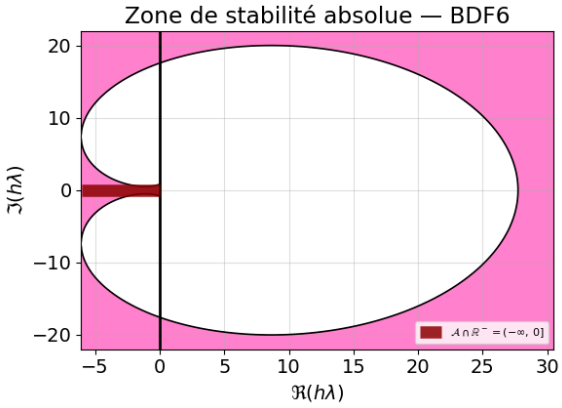
```
## polyCar(BDF5)
Astabilite(BDF5)
```






```
BDF6 = { "Nom":"BDF6",
 ↪  "aa":[sp.Rational(120,49),-sp.Rational(150,49),sp.Rational(400,147),-sp.Rational(75,49),sp.Rational(24,49),-s
 ↪
                          "bb":[0,0,0,0,0,0], "bm1":sp.Rational(20,49)  }
## polyCar(BDF6)
Astabilite(BDF6)
```





Chapitre 11.

Schémas de Runge-Kutta

Comme nous l'avons vu au chapitre précédent, les barrières sur l'ordre de convergence des méthodes multi-pas linéaires sont de plus en plus sévères. Nous allons alors étudier une autre famille de méthodes, à savoir les méthodes de Runge-Kutta. La stratégie est nouvelle par rapport à celle des méthodes multi-pas : en effet, **on construit des méthodes à un seul pas, mais nous passons à une stratégie multi-étages** s'appuyant sur l'information en quelques points supplémentaires, situés à l'intérieur de chaque sous-intervalle de la discrétisation du domaine.

```
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp
from scipy.optimize import fsolve
from IPython.display import display, Math, Markdown
```

11.1. Schémas

Les schémas de Runge-Kutta sont des méthodes à un pas permettant d'approcher l'intégrale

$$\int_{t_n}^{t_{n+1}} \varphi(t, y(t))\mathrm{d}t$$

par une formule de quadrature en des points situés à l'intérieur de l'intervalle $[t_n, t_{n+1}]$:

$$t_{n,i} \overset{\text{déf}}{=} t_n + c_i h, \quad \text{avec } c_i \in [0,1],$$

$$\int_{t_n}^{t_{n+1}} \varphi(t, y(t))\mathrm{d}t \approx h \sum_{i=1}^{s} b_i \varphi(t_{n,i}, y(t_{n,i})).$$

Cependant, si $c_i \neq 0$ et $c_i \neq 1$, alors $t_{n,i}$ ne correspond pas à un point de discrétisation, ce qui signifie que $y(t_{n,i})$ est inconnu.

Pour contourner ce problème, l'évaluation de $\varphi(t_{n,i}, y(t_{n,i}))$ en ces points intermédiaires est approchée à l'aide d'une autre formule de quadrature :

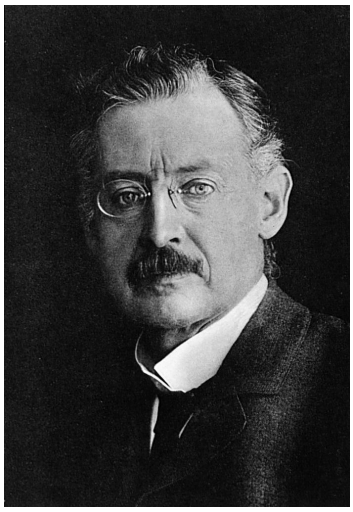
$$y(t_{n,i}) \approx y_n + h \sum_{j=1}^{s} a_{ij} \varphi(t_{n,j}, y(t_{n,j})).$$

On en déduit alors l'approximation suivante :

$$y_{n+1} \approx y_n + h \sum_{i=1}^{s} b_i \varphi \left(t_{n,i}, y_n + h \sum_{j=1}^{s} a_{ij} \varphi(t_{n,j}, y_{n,j}) \right).$$

Remarque 1 : si $c_s = 1$ et $a_{sj} = b_j$ pour tout $j = 1, \dots, s$, alors $K_s = \varphi(t_{n+1}, u_{n+1})$.

Remarque 2 : on verra des exemples où $c_i < 0$.



(a) Carl David Tolmé Runge (1856-1927)



(b) Martin Wilhelm Kutta (1867-1944)

FIGURE 11.1. – Runge et Kutta



FIGURE 11.2. – John Butcher (1933-)

Une méthode de Runge-Kutta à $s \geq 1$ étages contient $s^2 + 2s$ paramètres et s'écrit :

$$\begin{cases} u_0 = y(t_0) = y_0, \\ K_i = \varphi\left(t_n + hc_i, u_n + h\sum_{j=1}^{s} a_{ij}K_j\right), \quad i = 1, \dots s \\ u_{n+1} = u_n + h\sum_{i=1}^{s} b_i K_i \end{cases} \qquad n = 0, 1, \dots N-1.$$

NB C'est un schéma à un pas, car u_{n+1} ne dépend que de u_n et pas de $u_{n-1}, u_{n-2}, \dots$

11.2. Matrice de Butcher

Les coefficients sont généralement organisés en deux vecteurs $\mathbf{b} = (b_1, \dots, b_s)^T$, $\mathbf{c} = (c_1, \dots, c_s)^T$ et une matrice $\mathbb{A} = (a_{ij})_{1 \leq i,j \leq s}$.

Le tableau

$$\begin{array}{c|c} \mathbf{c} & \mathbb{A} \\ \hline & \mathbf{b}^T \end{array}$$

est appelé *matrice de Butcher* de la méthode de Runge-Kutta considérée.

- **Méthode explicite.** Si $a_{ij} = 0$ pour $j \geq i$ (i.e. la matrice $\mathbb{A}$ est **triangulaire inférieure stricte**), alors chaque K_i peut être calculé explicitement en fonction des $i-1$ coefficients $K_1, \dots, K_{i-1}$ déjà connus.
- **Méthode semi-implicite (ou diagonalement implicite, DIRK).** Si $a_{ij} = 0$ pour $j > i$ (i.e. la matrice $\mathbb{A}$ est **triangulaire inférieure**), alors chaque K_i est solution de l'équation non linéaire

$$\boxed{K_i} = \varphi\left(t_n + c_i h, u_n + h a_{ii} \boxed{K_i} + h \sum_{j=1}^{i-1} a_{ij} K_j\right)$$

  Un schéma semi-implicite implique donc la résolution de s équations non linéaires indépendantes.
  Si, de plus, tous les termes diagonaux sont égaux ($a_{ii} = \gamma$ pour $i = 1, \dots, s$), on parle de *singly diagonal implicit method* (SDIRK).
- **Méthode implicite.** Dans tous les autres cas, la méthode est **implicite**, ce qui signifie qu'il faut **résoudre un système non linéaire de dimension** s pour obtenir les K_i.
  Bien noter que la méthode est implicite non pas parce que u_{n+1} dépend de lui-même, mais parce qu'au moins un K_i dépend de lui-même.

L'augmentation du coût de calcul pour les schémas implicites peut rendre leur utilisation onéreuse. Un compromis souvent adopté consiste à utiliser des méthodes **semi-implicites**.

11.3. Convergence et ordre de convergence

L'étude de l'ordre des méthodes de Runge-Kutta utilise la théorie de Butcher. Elle repose sur des outils qui combinent l'analyse numérique, la théorie des graphes et la différenciation des champs vectoriels. Plus tard, des relations avec la théorie des groupes et la théorie quantique des champs ont été découvertes. Une monographie de référence sur la théorie de l'ordre de Butcher et les problèmes connexes est rédigée par John C. Butcher lui-même :

Butcher, J. C. (2016). *Numerical methods for ordinary differential equations.* John Wiley & Sons.

L'outil de base qui relie tous les domaines susmentionnés est la notion d'arbres racinés. Nous n'énoncerons ici que les résultats.

Soit ω l'ordre de la méthode (la méthode est dite **consistante** si $\omega \geq 1$).

Théorème de Lax-Richtmyer (consistance + zéro-stabilité $\iff$ convergence)
Le schéma est **convergent (d'ordre $\omega \geq 1$)** ssi il est **zéro-stable** et **consistante (d'ordre ω)**.

Proposition (Zéro-stabilité)
Les méthodes de Runge-Kutta sont **zéro-stables**.

Proposition (Consistance)
La méthode de Runge-Kutta est **consistante** (i.e. $\omega \geq 1$) ssi

$$\begin{cases} \displaystyle\sum_{j=1}^{s} b_j = 1 \\ \displaystyle c_i = \sum_{j=1}^{s} a_{ij}, \quad i = 1, \dots, s \end{cases}$$

Remarque 1 : une méthode à s étages consistante a alors $s^2 + 2s - (s+1) = s^2 + s - 1$ degrés de liberté.

Remarque 2 : une méthode *explicite* consistante vérifie

- $c_1 = a_{1,j} = 0$ ainsi $K_1 = \varphi(t_n, u_n)$
- $c_2 = a_{21}$ ainsi $K_2 = \varphi(t_n + c_2 h, u_n + h c_2 K_1)$.

Interprétation : un bon schéma doit au moins être exact pour les équations très simples. Imposons donc à ce que le schéma reproduise correctement l'EDO la plus simple, $y'(t) = 1$, dont la solution est $y(t) = t + C$. Comme $\varphi(t, y) = 1$, alors pour tout $i = 1, \dots, s$, on a $K_i = 1$. Le schéma devient alors

$$u_{n+1} = u_n + h \sum_{i=1}^{s} b_i.$$

Or la solution exacte vérifie $y(t_{n+1}) = y(t_n) + h$. Pour que le schéma coïncide avec cette évolution pour tout h, il faut nécessairement

$$\sum_{i=1}^{s} b_i = 1.$$

Par ailleurs,

$$K_i = \varphi\left(t_n + h c_i, \text{approx de } y(t_n + h c_i)\right),$$

autrement dit

$$u_n + h \sum_{j=1}^{s} a_{ij} K_j \approx y(t_n + h c_i).$$

Comme $K_i = 1$ et $y(t_n + h c_i) = y(t_n) + h c_i$, la cohérence au premier ordre impose

$$c_i = \sum_{j=1}^{s} a_{ij}, \quad i = 1, \dots, s.$$

Preuve. On notera $y_n = y(t_n)$ la solution exacte au temps t_n, $\varphi_n = \varphi(t_n, y(t_n))$ et $y'_n = y'(t_n)$ pour alléger l'écriture.

On note K_i^* les valeurs de stages obtenues en remplaçant u_n par y_n dans les équations de stage, c'est-à-dire les solutions de

$$K_i^* = \varphi\left(t_n + h c_i, y_n + h \sum_{j=1}^{s} a_{ij} K_j^*\right), \quad i = 1, \dots, s,$$

et on pose

$$u_{n+1}^* = y_n + h \sum_{i=1}^{s} b_i K_i^*.$$

Développement de Taylor des stages. Lorsque $h \to 0$, le deuxième argument de φ dans la définition de K_i^* tend vers y_n. Par continuité de φ et par le théorème du point fixe de Banach, les K_i^* convergent tous vers $\varphi(t_n, y_n) = \varphi_n$. On peut donc écrire

$$K_i^* = \varphi_n + O(h), \quad i = 1, \dots, s.$$

En substituant $K_j^* = \varphi_n + O(h)$ dans l'argument interne de φ, on obtient plus précisément

$$y_n + h \sum_{j=1}^{s} a_{ij} K_j^* = y_n + h \left(\sum_{j=1}^{s} a_{ij}\right) \varphi_n + O(h^2).$$

D'autre part, par développement de Taylor,

$$y(t_n + h c_i) = y_n + h c_i \varphi_n + O(h^2).$$

Substitution dans u_{n+1}^*. On utilise $K_i^* = \varphi_n + O(h)$:

$$\sum_{i=1}^{s} b_i K_i^* = \left(\sum_{i=1}^{s} b_i\right) \varphi_n + O(h),$$

d'où

$$u_{n+1}^* = y_n + h \left(\sum_{i=1}^{s} b_i\right) \varphi_n + O(h^2).$$

Par développement de Taylor, on a par ailleurs

$$y_{n+1} = y_n + h\varphi_n + O(h^2).$$

Erreur de troncature locale. En substituant les expressions obtenues :

$$\tau_{n+1}(h) = \frac{y_{n+1} - u^*_{n+1}}{h} = \frac{(y_n + h\varphi_n + O(h^2)) - \left(y_n + h\left(\sum_{i=1}^{s} b_i\right)\varphi_n + O(h^2)\right)}{h}$$

$$= \left(1 - \sum_{i=1}^{s} b_i\right)\varphi_n + O(h).$$

On a $\tau_{n+1}(h) = O(h)$ pour tout n et toute EDO si et seulement si le coefficient de φ_n s'annule, ce qui donne la **première condition de consistance** :

$$\sum_{i=1}^{s} b_i = 1.$$

Cohérence interne des stages. Pour que le schéma soit consistant, il faut de plus que le deuxième argument de φ dans K_i^* soit une approximation consistante de $y(t_n + hc_i)$ à l'ordre $O(h)$. En comparant terme à terme les deux développements de Taylor en h :

$$y_n + h\left(\sum_{j=1}^{s} a_{ij}\right)\varphi_n + O(h^2) \quad \text{et} \quad y(t_n + hc_i) = y_n + hc_i\varphi_n + O(h^2),$$

on identifie les termes en $h\varphi_n$ (valable pour tout φ_n et tout i), ce qui donne la **deuxième condition de consistance** :

$$c_i = \sum_{j=1}^{s} a_{ij}, \quad i = 1, \dots, s.$$

On conclut que le schéma de Runge–Kutta implicite à s étages est consistant (d'ordre au moins 1) si et seulement si

$$\sum_{i=1}^{s} b_i = 1 \qquad \text{et} \qquad c_i = \sum_{j=1}^{s} a_{ij}, \quad i = 1, \dots, s.$$

□

Proposition (Ordre de consistance)
Une méthode de Runge-Kutta est **consistante d'ordre** $\omega \geq 1$ si elle satisfait les conditions suivantes :

Ordre minimum	**Conditions à satisfaire**
$\omega \geq 1$	$\begin{cases} 1 = \sum_{j=1}^{s} b_j \\ c_i = \sum_{j=1}^{s} a_{ij}, \quad i = 1, \dots, s \end{cases}$
$\omega \geq 2$	Les conditions pour $\omega \geq 1$ **et** $\sum_{j=1}^{s} b_j c_j = \frac{1}{2}$
$\omega \geq 3$	Les conditions pour $\omega \geq 2$ **et** $\begin{cases} \sum_{j=1}^{s} b_j c_j^2 = \frac{1}{3} \\ \sum_{i=1}^{s}\sum_{j=1}^{s} b_i a_{ij} c_j = \frac{1}{6} \end{cases}$

$\omega \geq 4$ Les conditions pour $\omega \geq 3$ **et**

$$\begin{cases} \sum_{j=1}^{s} b_j c_j^3 = \frac{1}{4} \\ \sum_{i=1}^{s}\sum_{j=1}^{s} b_i c_i a_{ij} c_j = \frac{1}{8} \\ \sum_{i=1}^{s}\sum_{j=1}^{s} b_i a_{ij} c_j^2 = \frac{1}{12} \\ \sum_{i=1}^{s}\sum_{j=1}^{s}\sum_{k=1}^{s} b_i a_{ij} a_{jk} c_k = \frac{1}{24} \end{cases}$$

$\omega \geq 5$ Les conditions pour $\omega \geq 4$ **et**

$$\begin{cases} \sum_{j=1}^{s} b_j c_j^4 = \frac{1}{5} \\ \sum_{i,j=1}^{s} b_i c_i^2 a_{ij} c_j = \frac{1}{10} \\ \sum_{i=1}^{s} b_i \left(\sum_{j=1}^{s} a_{ij} c_j \right)^2 = \frac{1}{20} \\ \sum_{i,j=1}^{s} b_i a_{ij} c_j^3 = \frac{1}{20} \\ \sum_{i,j=1}^{s} b_i c_i a_{ij} c_j^2 = \frac{1}{15} \\ \sum_{i,j,k=1}^{s} b_i c_i a_{ij} a_{jk} c_k = \frac{1}{30} \\ \sum_{i,j,k=1}^{s} b_i a_{ij} c_j a_{jk} c_k = \frac{1}{40} \\ \sum_{i,j,k=1}^{s} b_i a_{ij} a_{jk} c_k^2 = \frac{1}{60} \\ \sum_{i,j,k,\ell=1}^{s} b_i a_{ij} a_{jk} a_{k\ell} c_\ell = \frac{1}{120} \end{cases}$$

Barrière de Butcher
L'ordre $\omega \geq 1$ d'une méthode de Runge-Kutta à $s \geq 1$ étapes convergente satisfait les inégalités suivantes :

$$\omega \leq \begin{cases} 2s, & \text{si la méthode est implicite,} \\ s, & \text{si la méthode est explicite et } s < 5, \\ s-1, & \text{si la méthode est explicite et } s \geq 5. \end{cases}$$

Exemples de méthodes
On verra plus bas que les méthodes classiques sont des exemples de méthodes de Runge-Kutta. On peut alors vérifier que les ordres de convergence de ces méthodes respectent les barrières de Butcher.

Schéma	Type de méthode RK	Nombre d'étages s	Ordre ω	Remarque
Schéma d'Euler explicite	Explicite	1	1	$s < 5$ et $\omega = s$
Schéma d'Euler implicite	Implicite	1	1	$\omega < 2s$ donc sub-optimale
Schéma d'Euler modifié	Explicite	2	2	$s < 5$ et $\omega = s$
Schéma de Heun	Explicite	2	2	$s < 5$ et $\omega = s$
Schéma de Crank-Nicholson	Implicite	2	2	$\omega < 2s$ donc sub-optimale

Les résultats ci-dessus permettent de comprendre la relation entre s le nombre d'étages d'une méthode de Runge-Kutta et son ordre de convergence ω.

1. **Méthode implicite** : l'ordre maximal est $2s$, ce qui permet d'obtenir un meilleur ordre de convergence avec moins d'étages, mais au coût d'une complexité supplémentaire (résolution d'un système non-linéaire).
2. **Méthode explicite** : si $s < 5$, l'ordre maximal est s ; si $s \geq 5$, l'ordre est limité à $s - 1$ (strictement inférieur au nombre d'étages donc). Autrement dit, un schéma RK explicite à s étages est au mieux d'ordre s, mais cela seulement si $s < 5$. Si $s \geq 5$, alors il sera au mieux d'ordre $s - 1$, mais ce n'est même pas garanti. En effet, les travaux de Butcher démontrent que pour des méthodes d'ordre 7 et 8, un minimum de 9 et 11 étages, respectivement, est requis. En général, le nombre minimum précis d'étages pour qu'une méthode explicite de Runge-Kutta ait l'ordre ω reste un problème ouvert. Certaines valeurs sont connues :

ω	1	2	3	4	5	6	7	8
s minimal	1	2	3	4	6	7	9	11

Remarque (le cas des systèmes) Une méthode de Runge-Kutta peut être aisément étendue aux systèmes d'équations différentielles ordinaires. Néanmoins, l'ordre d'une méthode RK dans le cas scalaire ne coïncide pas nécessairement avec celui de la méthode analogue dans le cas vectoriel. En particulier, pour $p \geq 4$, une méthode d'ordre p dans le cas du système autonome $\mathbf{y}'(t) = \varphi(t, \mathbf{y})$, avec $\varphi \colon \mathbb{R}^m \to \mathbb{R}^n$ est encore d'ordre p quand on l'applique à l'équation scalaire autonome $\mathbf{y}'(t) = \varphi(\mathbf{y})$, mais la réciproque n'est pas vraie.

11.3.1. Calcul explicite de l'ordre de consistance avec Sympy

La fonction `ordre_RK` sert à analyser l'ordre de consistance d'une méthode de Runge-Kutta donnée. Elle prend en entrée :

- `s` : le nombre d'étages de la méthode Runge-Kutta (RK),

- `type` : le type de la méthode RK, soit "explicite", soit "implicite" (optionnel)
- `A`, `b`, `c` (optionnels) : les coefficients de la matrice de Butcher. Si ces coefficients ne sont pas fournis, ils sont remplacés par des variables symboliques
- `ordre_max` (optionnel) : l'ordre maximum à vérifier. Par défaut, il est égal à s pour les méthodes explicites et à $2s$ pour les méthodes implicites.

La fonction affiche successivement : la matrice de Butcher, qui résume les coefficients de la méthode RK, et les conditions nécessaires pour que la méthode soit d'un certain ordre (de 1 à 4).

- Exemple d'utilisation ou tous les coefficients sont des variables symboliques

```
ordre_RK(s=2, type="explicite", ordre_max = 3)
```

- Exemple d'utilisation avec les coefficients de la matrice de Butcher donnés

```
A = sp.Matrix([[0, 0], [sp.Rational(1, 2), 0]])
b = sp.Matrix([sp.Rational(1, 2), sp.Rational(1, 2)])
c = sp.Matrix([0, 1])

s = len(c)
ordre_RK(s, A, b, c, ordre_max=3)
```

```
# ============================================================================
# Fonctions pour calculer les conditions de consistance pour un schéma RK.
#
# La fonction ordre_RK calcule les conditions pour les ordres 1, 2, 3 et 4 (même si théoriquement cela
 ↪ n'est pas possible)
# - affiche la matrice de Butcher
# - affiche les conditions de consistance obtenues pour chaque ordre par la fonction ordre_i
# - renvoie une liste d'équations pour une éventuelle résolution
#
```

```
# Chaque fonction ordre_i calcule les conditions pour être d'ordre i
# - peut afficher les conditions de consistance [actuellement en commentaire]
# - renvoie la liste d'équations
#+++
# ============================================================================

# ============================================================================
# Calcul des conditions de consistance pour un schéma RK
# On doit indiquer
# le nombre d'étages s
# le type de schéma parmi implicite, semi-implicite ou explicite
# éventuellement les coefficients A, b et c
# ============================================================================

def ordre_RK(s, type="implicite", A=None, b=None, c=None, ordre_max = None):

    j = sp.symbols('j')

    if ordre_max is None :
        if type == "explicite" and s<5: ordre_max = s
        else : ordre_max = 2*s

    if c is None: c = sp.symbols(f'c_0:{s}')
    if b is None: b = sp.symbols(f'b_0:{s}')

    if A is None:
        if type == "implicite": A = sp.MatrixSymbol('a', s, s)
        elif type=="semi-implicite": A = sp.Matrix(s, s, lambda i,j: sp.Symbol('a{}{}'.format(i, j)) if i
         ↪  >= j else 0)
        elif type=="explicite": A = sp.Matrix(s, s, lambda i,j: sp.Symbol('a{}{}'.format(i, j)) if i > j
         ↪  else 0)
    else:
        A = sp.Matrix(A)

    display(Markdown("**Matrice de Butcher**"))
    But = matrice_Butcher(s, A, b, c)

    Eqs = []

    display(Markdown(f"**On a {s+1} conditions pour avoir consistance (= pour être d'ordre 1)**"))
    Eq_1 = ordre_1(s, A, b, c, j)
    Eqs.append(Eq_1)
    display(*Eq_1)

    if ordre_max >=2:
        display(Markdown("**On doit ajouter 1 condition pour être d'ordre 2**"))
        Eq_2 = ordre_2(s, A, b, c, j)
        Eqs.append([Eq_2])
        display(Eq_2)

    if ordre_max >=3:
        display(Markdown("**On doit ajouter 2 conditions pour être d'ordre 3**"))
        Eq_3 = ordre_3(s, A, b, c, j)
        Eqs.append(Eq_3)
        display(*Eq_3)

    if ordre_max >=4:
        display(Markdown("**On doit ajouter 4 conditions pour être d'ordre 4**"))
        Eq_4 = ordre_4(s, A, b, c, j)
        Eqs.append(Eq_4)
        display(*Eq_4)

    return Eqs
```

```
def matrice_Butcher(s, A, b, c):
    But = sp.Matrix(A)
    But = But.col_insert(0, sp.Matrix(c))
    last = [sp.Symbol(" ")]
    last.extend(b)
    But = But.row_insert(s,sp.Matrix(last).T)
    display(Math(sp.latex(sp.Matrix(But))))
    return But


def ordre_1(s, A, b, c, j):
    texte = r"\sum_{j=1}^s b_j =" + f"{sum(b).simplify()}"
    texte += r"\text{ doit être égale à }1"
    # display(Math(texte))
    Eq_1 = [sp.Eq( sum(b).simplify(), 1 )]
    for i in range(s):
        somma = sp.summation(A[i,j],(j,0,s-1)).simplify()
        texte = r'\sum_{j=1}^s a_{'+str(i)+r'j}=' + sp.latex( somma )
        texte += r"\text{ doit être égale à }"+sp.latex(c[i])
        # display(Math( texte ))
        Eq_1.append(sp.Eq( somma, c[i] ))
    return Eq_1


def ordre_2(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j=' +sp.latex(sum([b[i]*c[i] for i in range(s)]).simplify())
    texte += r"\text{ doit être égale à }\frac{1}{2}"
    # display(Math(texte))
    Eq_2 = sp.Eq( sum([b[i]*c[i] for i in range(s)]).simplify(), sp.Rational(1,2) )
    return Eq_2


def ordre_3(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j^2='
    texte += sp.latex( sum([b[i]*c[i]**2 for i in range(s)]).simplify() )
    texte += r"\text{ doit être égale à }\frac{1}{3}"
    # display(Math(texte))
    Eq_3_1 = sp.Eq( sum([b[i]*c[i]**2 for i in range(s)]).simplify(), sp.Rational(1,3) )
    texte = r'\sum_{i,j=1}^s b_i a_{ij} c_j='
    somma = sum([b[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{6}"
    # display(Math(texte))
    Eq_3_2 = sp.Eq( sum([b[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,6) )
    return [Eq_3_1, Eq_3_2]


def ordre_4(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j^3='
    texte += sp.latex( sum([b[i]*c[i]**3 for i in range(s)]).simplify() )
    texte += r"\text{ doit être égale à }\frac{1}{4}"
    # display(Math(texte))
    Eq_4_1 = sp.Eq( sum([b[i]*c[i]**3 for i in range(s)]).simplify(), sp.Rational(1,4) )
    texte = r'\sum_{i,j=1}^s b_i c_i a_{ij} c_j='
    somma = sum([b[i]*c[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{8}"
    # display(Math(texte))
    Eq_4_2 = sp.Eq( sum([b[i]*c[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,8) )
```

```
    texte = r'\sum_{i,j=1}^s b_i a_{ij} c_j^2='
    somma = sum([b[i]*A[i,j]*c[j]**2 for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{12}"
    # display(Math(texte))
    Eq_4_3 = sp.Eq( sum([b[i]*A[i,j]*c[j]**2 for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,12) )
    texte = r'\sum_{i,j,k=1}^s b_i a_{ij} a_{jk} c_k='
    somma = sum([b[i]*A[i,j]*A[j,k]*c[k] for k in range(s) for j in range(s) for i in
↪   range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{24}"
    # display(Math(texte))
    Eq_4_4 = sp.Eq( sum([b[i]*A[i,j]*A[j,k]*c[k] for k in range(s) for j in range(s) for i in
↪   range(s)]).simplify(), sp.Rational(1,24) )
    return [Eq_4_1, Eq_4_2, Eq_4_3, Eq_4_4]
```

```
# Exemple sans paramètres
s = 2
type = "implicite"
display(Markdown(f"<mark>Conditions pour un schéma {type} avec s = {s} étages</mark>"))
ordre_RK( s = s, type = type );
```

Conditions pour un schéma implicite avec s = 2 étages

Matrice de Butcher

$$\begin{bmatrix} c_0 & a_{0,0} & a_{0,1} \\ c_1 & a_{1,0} & a_{1,1} \\ & b_0 & b_1 \end{bmatrix}$$

On a 3 conditions pour avoir consistance (= pour être d'ordre 1)

$$b_0 + b_1 = 1$$

$$a_{0,0} + a_{0,1} = c_0$$

$$a_{1,0} + a_{1,1} = c_1$$

On doit ajouter 1 condition pour être d'ordre 2

$$b_0 c_0 + b_1 c_1 = \frac{1}{2}$$

On doit ajouter 2 conditions pour être d'ordre 3

$$b_0 c_0^2 + b_1 c_1^2 = \frac{1}{3}$$

$$b_0 c_0 a_{0,0} + b_0 c_1 a_{0,1} + b_1 c_0 a_{1,0} + b_1 c_1 a_{1,1} = \frac{1}{6}$$

On doit ajouter 4 conditions pour être d'ordre 4

$$b_0 c_0^3 + b_1 c_1^3 = \frac{1}{4}$$

$$b_0 c_0^2 a_{0,0} + b_0 c_0 c_1 a_{0,1} + b_1 c_0 c_1 a_{1,0} + b_1 c_1^2 a_{1,1} = \frac{1}{8}$$

$$b_0 c_0^2 a_{0,0} + b_0 c_1^2 a_{0,1} + b_1 c_0^2 a_{1,0} + b_1 c_1^2 a_{1,1} = \frac{1}{12}$$

$$b_0 c_0 a_{0,0}^2 + b_0 c_0 a_{0,1} a_{1,0} + b_0 c_1 a_{0,0} a_{0,1} + b_0 c_1 a_{0,1} a_{1,1} + b_1 c_0 a_{0,0} a_{1,0} + b_1 c_0 a_{1,0} a_{1,1} + b_1 c_1 a_{0,1} a_{1,0} + b_1 c_1 a_{1,1}^2 = \frac{1}{24}$$

```
# Exemple : méthode à 2 étages, correspondant à Crank-Nicolson

A = [ [sp.S(0)           , sp.S(0)],
      [sp.Rational(1,2)  , sp.Rational(1,2)]
      ]
b = [sp.Rational(1,2), sp.Rational(1,2)]
c = [sp.S(0), sp.S(1)]

s = len(c)

display(Markdown(f"<mark>Exemple de méthode de Runge-Kutta à {s} étages (Crank-Nicolson)</mark>"))
ordre_RK(s, A=A, b=b, c=c);
```

Exemple de méthode de Runge-Kutta à 2 étages (Crank-Nicolson)

Matrice de Butcher

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & \frac{1}{2} & \frac{1}{2} \\ & \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

On a 3 conditions pour avoir consistance (= pour être d'ordre 1)

True

True

True

On doit ajouter 1 condition pour être d'ordre 2

True

On doit ajouter 2 conditions pour être d'ordre 3

False

False

On doit ajouter 4 conditions pour être d'ordre 4

False

False

False

False

11.4. A-stabilité

Rappel : définition de A-stabilité dans $\mathbb{C}$

Soit $\lambda \in \mathbb{C}$ avec $\Re(\lambda) = -\beta$ où $\beta \geq 0$. Considérons l'équation test $y'(t) = \lambda y(t)$. Sa solution générale est $y(t) = y_0 e^{\lambda t}$, ce qui implique $\lim_{t\to+\infty} |y(t)| = 0$.
Soit $h > 0$ le pas de temps et $u_n \approx y(t_n)$ l'approximation numérique. On dit que **le schéma est A-stable** si, sous d'éventuelles conditions sur h, on a $\lim_{n\to+\infty} |u_n| = 0$.

Les méthodes de Runge-Kutta étant des méthodes à un pas, lorsqu'on les applique à l'équation test $y'(t) = \lambda y(t)$, on obtient une relation de récurrence de la forme $u_{n+1} = q(h\lambda)u_n$ pour une certaine fonction q. Par conséquent, la condition de A-stabilité se traduit par $|q(h\lambda)| < 1$.

Proposition (A-stabilité sur l'axe réel négatif pour un schéma RK).
Considérons l'équation test

$$y'(t) = -\beta y(t), \qquad \beta \in \mathbb{R}_*^+$$

et un schéma de Runge-Kutta à s étages, de matrice de Butcher $\mathbb{A}$ et de poids $\mathbf{b}$. Posons $x = \beta h > 0$ et $\mathbf{1}_s = (1, \dots, 1)^T$ le vecteur de dimension s dont tous les éléments sont égaux à 1. Notons

$$q(x) = 1 - x\,\mathbf{b}^T(\mathbb{I}_s + x\mathbb{A})^{-1}\mathbf{1}_s,$$

pour tout x tel que $\mathbb{I}_s + x\mathbb{A}$ soit inversible (i.e. $-1/x \notin \sigma(\mathbb{A})$).
Alors la suite numérique vérifie $u_{n+1} = q(x)u_n$ et, par conséquent, le schéma est A-stable (sous des eventuelles conditions sur h) ssi $|q(x)| < 1$.

Preuve. La méthode de Runge-Kutta à s étages pour l'équation test s'écrit :

$$\begin{cases} K_i = -\beta\left(u_n + h\sum_{j=1}^s a_{ij}K_j\right), \quad i = 1, \dots, s. \\ u_{n+1} = u_n + h\sum_{i=1}^s b_i K_i, \end{cases}$$

En introduisant les notations vectorielles $\mathbf{K} = (K_1, \dots, K_s)^T$, le système se réécrit de manière compacte :

$$\mathbf{K} = -\beta u_n \mathbf{1} - \beta h\,\mathbb{A}\mathbf{K}, \qquad u_{n+1} = u_n + h\,\mathbf{b}^T\mathbf{K}.$$

En isolant $\mathbf{K}$ dans la première équation :

$$(\mathbb{I}_s + \beta h\mathbb{A})\mathbf{K} = -\beta u_n \mathbf{1}_s \rightsquigarrow \mathbf{K} = (\mathbb{I}_s + \beta h\mathbb{A})^{-1}(-\beta u_n \mathbf{1}_s).$$

Si la matrice $\mathbb{I}_s + \beta h\mathbb{A}$ est inversible (c'est-à-dire si $-1/(\beta h)$ n'est pas une valeur propre de la matrice de Butcher $\mathbb{A}$), alors

$$\mathbf{K} = (\mathbb{I}_s + \beta h\mathbb{A})^{-1}(-\beta u_n \mathbf{1}_s).$$

En substituant ce résultat dans l'expression de u_{n+1}, nous obtenons la relation de récurrence :

$$u_{n+1} = u_n - \beta h\,\mathbf{b}^T(\mathbb{I}_s + \beta h\mathbb{A})^{-1}\mathbf{1}_s\,u_n = u_n\,\underbrace{(1 - \beta h\mathbf{b}^T(\mathbb{I}_s + \beta h\mathbb{A})^{-1}\mathbf{1}_s)}_{q(h\beta)} = q(\beta h)\,u_n.$$

Avec $x = \beta h$, on obtient bien $u_{n+1} = q(x)u_n$, donc par récurrence $u_n = q^n(x)u_0$. Ainsi, $u_n \to 0$ ssi $q^n(x) \to 0$ ssi $|q(x)| < 1$. □

```python
def calculer_q_rk_general(s, A=None, b=None):
    x = sp.symbols('x')
    # beta, h = sp.symbols('beta h')
    # x = beta * h

    if A is None:  A = sp.Matrix(s, s, lambda i, j: sp.symbols(f'a_{i+1}{j+1}'))
    if b is None:  b = sp.Matrix(s, 1, lambda i, j: sp.symbols(f'b_{i+1}'))

    ones = sp.Matrix.ones(s, 1)
    I = sp.eye(s)
    mat_res = x * b.T * (I + x * A).inv() * ones

    # On extrait le contenu avec [0, 0] pour obtenir un scalaire SymPy
    produit_scalaire = mat_res[0, 0]
    q = 1 - produit_scalaire
    return sp.simplify(q)
```

```python
# test de la fonction pour s = 2
q = calculer_q_rk_general(2)
display(Markdown(f"$$q(x) = {sp.latex(q)}$$"))
```

$$q(x) = \frac{a_{11}a_{22}x^2 + a_{11}x - a_{12}a_{21}x^2 + a_{12}b_1x^2 + a_{21}b_2x^2 + a_{22}x - b_1x\,(a_{22}x + 1) - b_2x\,(a_{11}x + 1) + 1}{a_{11}a_{22}x^2 + a_{11}x - a_{12}a_{21}x^2 + a_{22}x + 1}$$

```
# test de la fonction pour s = 2 et des coefficients donnés
s = 2
A = sp.Matrix(s, s, lambda i, j: sp.symbols(f'a_{i+1}{j+1}'))
b = sp.Matrix(s, 1, lambda i, j: sp.symbols(f'b_{i+1}'))
A[0, 0] = A[0, 1] = A[1,1] = sp.S(0)
b[0] = 1-b[1]
display(Markdown(f"$$A = {sp.latex(A)},\\qquad b^T = {sp.latex(b.T)}$$"))
q = calculer_q_rk_general(s, A, b)
display(Markdown(f"$$q(x) = {sp.latex(q)}$$"))
```

$$A = \begin{bmatrix} 0 & 0 \\ a_{21} & 0 \end{bmatrix}, \qquad b^T = \begin{bmatrix} 1-b_2 & b_2 \end{bmatrix}$$

$$q(x) = a_{21} b_2 x^2 - x + 1$$

Le facteur de stabilité q est une fonction rationnelle de $x = \beta h$ dont les coefficients dépendent de la matrice de Butcher. En particulier, $q(0) = 1$ et $q'(0) = -1$.

Proposition (Formule de Butcher pour le facteur de stabilité)
Le facteur de stabilité q défini par

$$q(x) = 1 - x\,\mathbf{b}^T(\mathbb{I}_s + x\mathbb{A})^{-1}\mathbf{1}_s$$

se réécrit aussi sous la forme

$$q(x) = \frac{\det(\mathbb{I}_s + x\mathbb{A} - x\mathbf{1}_s\mathbf{b}^T)}{\det(\mathbb{I}_s + x\mathbb{A})}.$$

Autrement dit, $q(x)$ est une fraction rationnelle.

Preuve. Le lemme du déterminant de Sylvester affirme que, pour toute matrice $\mathbb{M}$ inversible et tous vecteurs $\mathbf{u}$ et $\mathbf{v}$ de $\mathbb{R}^s$, on a

$$\det(\mathbb{M} - \mathbf{u}\mathbf{v}^T) = \det(\mathbb{M})\left(\mathbb{I}_s - \mathbf{v}^T\mathbb{M}^{-1}\mathbf{u}\right).$$

Soit $\mathbb{M} = \mathbb{I}_s + x\mathbb{A}$, $\mathbf{v} = x\mathbf{b}$ et $\mathbf{u} = \mathbf{1}_s$. Alors, on trouve que

$$\begin{aligned}
\det(\mathbb{I}_s + x\mathbb{A} - x\mathbf{1}_s\mathbf{b}^T) &= \det(\mathbb{M} - \mathbf{u}\mathbf{v}^T) \\
&= \det(\mathbb{M})\left(\mathbb{I}_s - \mathbf{v}^T\mathbb{M}^{-1}\mathbf{u}\right) \\
&= \det(\mathbb{I}_s + x\mathbb{A})\left(\mathbb{I}_s - x\mathbf{b}^T(\mathbb{I}_s + x\mathbb{A})^{-1}\mathbf{1}_s\right) \\
&= \det(\mathbb{I}_s + x\mathbb{A})q(x).
\end{aligned}$$

□

```
def calculer_q_rk_general_v2(s, A=None, b=None):
    x = sp.symbols('x')
    # beta, h = sp.symbols('beta h')
    # x = beta * h

    if A is None:  A = sp.Matrix(s, s, lambda i, j: sp.symbols(f'a_{i+1}{j+1}'))
    if b is None:  b = sp.Matrix(s, 1, lambda i, j: sp.symbols(f'b_{i+1}'))

    ones = sp.Matrix.ones(s, 1)
    I = sp.eye(s)
    q = sp.det(I + x * A - x * ones * b.T) / sp.det(I + x * A)
    return sp.simplify(q)
```

```
# test de la fonction pour s = 2
# q = calculer_q_rk_general(2)
# display(Markdown(f"$$q(x) = {sp.latex(q)}$$"))
q = calculer_q_rk_general_v2(2)
display(Markdown(f"$$q(x) = {sp.latex(q)}$$"))
```

$$q(x) = \frac{a_{11}a_{22}x^2 - a_{11}b_2x^2 + a_{11}x - a_{12}a_{21}x^2 + a_{12}b_1x^2 + a_{21}b_2x^2 - a_{22}b_1x^2 + a_{22}x - b_1x - b_2x + 1}{a_{11}a_{22}x^2 + a_{11}x - a_{12}a_{21}x^2 + a_{22}x + 1}$$

```
# test de la fonction pour s = 2 et des coefficients donnés
s = 2
A = sp.Matrix(s, s, lambda i, j: sp.symbols(f'a_{i+1}{j+1}'))
b = sp.Matrix(s, 1, lambda i, j: sp.symbols(f'b_{i+1}'))
A[0, 0] = A[0, 1] = A[1,1] = sp.S(0)
b[0] = 1-b[1]
display(Markdown(f"$$A = {sp.latex(A)},\\qquad b^T = {sp.latex(b.T)}$$"))
# q = calculer_q_rk_general(s, A, b)
# display(Markdown(f"$$q(x) = {sp.latex(q)}$$"))
q = calculer_q_rk_general_v2(s, A, b)
display(Markdown(f"$$q(x) = {sp.latex(q)}$$"))
```

$$A = \begin{bmatrix} 0 & 0 \\ a_{21} & 0 \end{bmatrix}, \qquad b^T = \begin{bmatrix} 1 - b_2 & b_2 \end{bmatrix}$$

$$q(x) = a_{21}b_2x^2 - x + 1$$

Proposition (Inexistence de la A-stabilité inconditionnelle pour les schémas explicites)
Toute méthode de Runge-Kutta explicite à s étages possède un facteur de stabilité $q(x)$ polynomial. Par conséquent, aucune de ces méthodes ne peut être inconditionnellement A-stable.

Preuve. Soit une méthode de Runge-Kutta explicite à s étages définie par sa matrice de Butcher $\mathbb{A} \in \mathcal{M}_s(\mathbb{R})$ et son vecteur de poids $\mathbf{b} \in \mathbb{R}^s$. On considère la variable réelle $x = \beta h \geq 0$.

- Méthode 1 : Dans un schéma explicite, $\mathbb{A}$ est une matrice triangulaire inférieure stricte ($a_{ij} = 0$ pour $j \geq i$). Une propriété fondamentale de ces matrices est qu'elles sont **nilpotentes** d'indice s, *i.e.* $\mathbb{A}^s = \mathbb{O}$, ce qui implique que les puissances de $\mathbb{A}$ au-delà de $s - 1$ sont nulles.

  Le facteur de stabilité est donné par l'expression matricielle $q(x) = 1 - x\mathbf{b}^T(\mathbb{I}_s + x\mathbb{A})^{-1}\mathbf{1}_s$. Puisque la matrice $x\mathbb{A}$ est nilpotente, son inverse s'exprime par une somme finie (série de Neumann tronquée) :

  $$(\mathbb{I}_s + x\mathbb{A})^{-1} = \mathbb{I}_s - x\mathbb{A} + x^2\mathbb{A}^2 - \cdots + (-x)^{s-1}\mathbb{A}^{s-1} = \sum_{k=0}^{s-1}(-x\mathbb{A})^k.$$

  En substituant ce développement dans l'expression de $q(x)$, nous obtenons :

  $$q(x) = 1 - x\mathbf{b}^T\left(\sum_{k=0}^{s-1}(-x\mathbb{A})^k\right)\mathbf{1}_s.$$

  En distribuant le produit, on identifie un polynôme en x de degré au plus s :

  $$q(x) = 1 + \sum_{k=1}^{s}(-x)^k(\mathbf{b}^T\mathbb{A}^{k-1}\mathbf{1}_s).$$

  Pour tout polynôme $q(x)$ de degré non nul, on a

  $$\lim_{x\to+\infty} |q(x)| = +\infty.$$

  Il existe donc nécessairement une valeur critique x^* telle que pour tout $x > x^*$, $|q(x)| > 1$. La région de stabilité est donc bornée sur l'axe réel, ce qui exclut la A-stabilité inconditionnelle.

- Méthode 2 : En utilisant la formule de Butcher pour le facteur de stabilité, on trouve que

$$q(x) = \frac{\det(\mathbb{I}_s + x\mathbb{A} - x\mathbf{1}_s\mathbf{b}^T)}{\det(\mathbb{I}_s + x\mathbb{A})}.$$

  Puisque $\mathbb{A}$ est une matrice triangulaire inférieure stricte, $\det(\mathbb{I}_s + x\mathbb{A}) = 1$ pour tout x. Par conséquent, $q(x)$ est égal au déterminant du numérateur, qui est un polynôme de degré au plus s en x.
  Comme dans la méthode 1, cela implique que $\lim_{x\to+\infty} |q(x)| = +\infty$, et donc que la région de stabilité est bornée sur l'axe réel, excluant ainsi la A-stabilité inconditionnelle.

□

Remarque. Si le schéma **explicite** est d'**ordre maximal avec** $s < 5$ (c'est-à-dire $\omega = s$), alors les conditions d'ordre imposent que :

$$\mathbf{b}^T\mathbb{A}^{k-1}\mathbf{1}_s = \frac{1}{k!}, \quad \text{pour } k = 1, \dots, s.$$

Le facteur de stabilité $q(x)$ s'écrit alors comme le polynôme de Taylor de l'exponentielle :

$$q(x) = 1 + \sum_{k=1}^{s} \frac{(-x)^k}{k!} = \sum_{k=0}^{s} \frac{(-x)^k}{k!}.$$

On retrouve bien que pour $s \to \infty$, $q(x)$ converge vers e^{-x} pour tout x. réel.

```
# Polynômes de A-stabilité pour s = 1 à 4 (supposés d'ordre maximal omega=s)
# q(x) = \sum_{k=0}^s \frac{(-x)^k}{k!}

s_max = 4

# 1. Pré-calcul des factorielles
# On stocke 1/k! pour transformer la division en multiplication (plus rapide)
from math import factorial
inv_fact = [1 / factorial(k) for k in range(s_max+1)]

# 2. Fonction optimisée utilisant les coefficients pré-calculés
def q_omega_max_opti(x, s):
    # On calcule la somme : 1 - x/1! + x^2/2! - x^3/3! ...
    # polyval de numpy est aussi une option, mais une compréhension de liste suffit ici
    return sum([inv_fact[k] * (-x)**k for k in range(s+1)])


plt.figure(figsize=(10, 4))
xx = np.linspace(0, 4, 500)

linestyles = ['-', '--', '-.', ':']

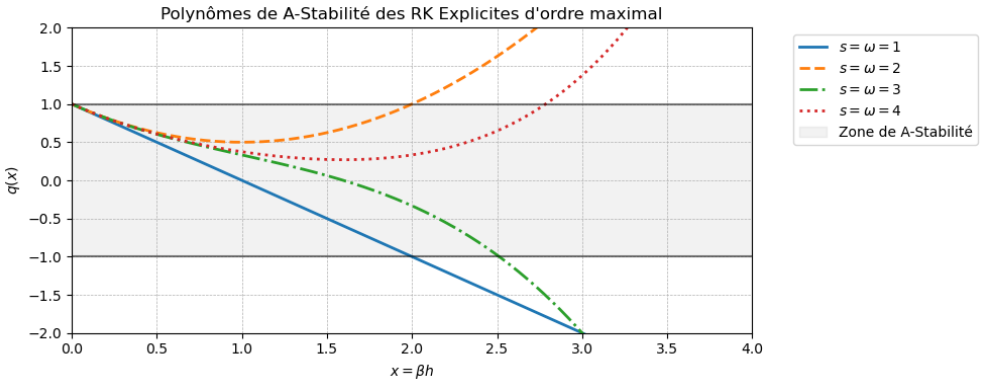
# Tracé des courbes avec changement de style et de couleur
for s in range(1, s_max+1):
    plt.plot(xx, q_omega_max_opti(xx, s),
             label=f'$s = \\omega = {s}$',
             linestyle=linestyles[(s-1)%len(linestyles)],
             linewidth=2)

# Tracé des limites de stabilité (|q(x)| = 1)
plt.axhline(y=1, color='black', linestyle='-', alpha=0.5)
plt.axhline(y=-1, color='black', linestyle='-', alpha=0.5)
plt.fill_between(xx, -1, 1, color='gray', alpha=0.1, label='Zone de A-Stabilité')

# Configuration des axes
plt.title("Polynômes de A-Stabilité des RK Explicites d'ordre maximal")
plt.xlabel(r'$x = \beta h$')
plt.ylabel(r'$q(x)$')
```

```
plt.ylim(-2, 2) # On zoome sur la zone intéressante [-1, 1]
plt.xlim(xx.min(), xx.max())
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left');


plt.tight_layout()
plt.show()
```



On constate que, **contrairement aux méthodes multi-pas, la taille des régions de stabilité absolue des méthodes RK augmente quand l'ordre augmente.**

11.5. Exercice : schémas RK à 1 étage

Étudier les méthodes RK avec $s = 1$ **consistants** :

1. écrire le schéma **implicite** générique RK à un étage
2. étudier son ordre de convergence
3. écrire le schéma **explicite** générique RK à un étage
4. étudier son ordre de convergence
5. étudier la A-stabilité

Un schéma de Runge-Kutta à un étage est défini par les coefficients a_{11}, b_1 et c_1. En ajoutant les conditions de consistance, on trouve que $c_1 = a_{11}$ et $b_1 = 1$.

Ces méthodes ont pour matrice de Butcher

$$\begin{array}{c|c} c_1 & a_{11} \\ \hline & b_1 \end{array} \text{ avec } \begin{cases} c_1 = a_{11} \\ b_1 = 1 \end{cases} \quad \rightsquigarrow \boxed{\begin{array}{c|c} c_1 & c_1 \\ \hline & 1 \end{array} \quad \rightsquigarrow \begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n + hc_1, u_n + hc_1 K_1\right) \\ u_{n+1} = u_n + hK_1 \end{cases} \quad n = 0, 1, \dots N-1}$$

Exemple

Si on choisit $c_1 = 1$ on trouve le schéma implicite

$$\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array} \quad \Longrightarrow \begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_{n+1}, u_n + hK_1\right) \\ u_{n+1} = u_n + hK_1 \end{cases} \quad n = 0, 1, \dots N-1$$

Ce schéma n'est rien d'autre que le schéma d'Euler implicite. En effet,

$$K_1 = \varphi\left(t_{n+1}, u_n + hK_1\right) = \varphi\left(t_{n+1}, u_{n+1}\right) \quad \rightsquigarrow \quad u_{n+1} = u_n + hK_1 = u_n + h\varphi\left(t_{n+1}, u_{n+1}\right)$$

Plus généralement, si $c_s = 1$ et $a_{sj} = b_j$ pour tout $j = 1, \dots, s$ alors $K_s = \varphi(t_{n+1}, u_{n+1})$.

11.5.1. Ordre maximal de convergence

Selon le théorème de Butcher :

- si le schéma est implicite, il est au mieux d'ordre $\omega \leq 2s = 2$;
- si le schéma est explicite, il est au mieux d'ordre $\omega \leq s = 1$.

Les conditions sont les suivantes :

```
ordre_RK(s=1);
```

Matrice de Butcher

$$\begin{bmatrix} c_0 & a_{0,0} \\ & b_0 \end{bmatrix}$$

On a 2 conditions pour avoir consistance (= pour être d'ordre 1)

$b_0 = 1$

$a_{0,0} = c_0$

On doit ajouter 1 condition pour être d'ordre 2

$b_0 c_0 = \dfrac{1}{2}$

Pour que la méthode soit d'ordre 2 il faut que $1 \times c_1 = \frac{1}{2}$: il existe une seule méthode d'ordre 2, elle est implicite et sa matrice de Butcher est

$$\begin{array}{c|c} \frac{1}{2} & \frac{1}{2} \\ \hline & 1 \end{array} \quad \Longrightarrow \begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n + \frac{h}{2}, u_n + \frac{h}{2}K_1\right) \\ u_{n+1} = u_n + hK_1 \qquad\qquad n = 0, 1, \dots N-1 \end{cases}$$

Notons que $u_{n+1} = u_n + hK_1$, donc $hK_1 = u_{n+1} - u_n$. Si on remplace le hK_1 dans la définition de K_1 ou trouve $K_1 = \varphi\left(t_n + \frac{h}{2}, u_n + \frac{h}{2}K_1\right) = \varphi\left(t_n + \frac{h}{2}, u_n + \frac{h}{2}u_{n+1} - \frac{h}{2}u_n\right) = \varphi\left(t_n + \frac{h}{2}, h\frac{(u_{n+1}+u_n)}{2}\right)$: on obtient le schéma RK1_M

$$K_1 = \varphi\left(t_n + \frac{h}{2}, \frac{u_n + u_{n+1}}{2}\right) \quad \Longrightarrow \quad u_{n+1} = u_n + hK_1 = u_n + h\varphi\left(\frac{t_n + t_{n+1}}{2}, \frac{u_n + u_{n+1}}{2}\right)$$

11.5.2. Schémas explicites

Il existe un seul schéma explicite RK à un étage obtenu avec $c_1 = 0$:

$$\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array} \quad \Longrightarrow \begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n, u_n\right) \\ u_{n+1} = u_n + hK_1 \quad n = 0, 1, \dots N-1 \end{cases}$$

Ce schéma n'est rien d'autre que le schéma d'Euler explicite.

Si le schéma est explicite, alors il est forcément d'ordre 1.

11.5.3. Étude de la A-stabilité

Une méthode de Runge-Kutta à 1 étage pour $\varphi(t,y) = -\beta y$ s'écrit :

$$\begin{cases} u_0 = y_0, \\ u_{n+1} = u_n + hK_1 & n = 0, 1, \dots N-1 \\ K_1 = -\beta\left(u_n + hc_1K_1\right). \end{cases}$$

On trouve donc $(1+\beta h c_1)K_1 = -\beta u_n$ ainsi

$$\begin{cases} u_0 = y_0 \\ u_{n+1} = \left(1 - \frac{(\beta h)}{1+c_1(\beta h)}\right) u_n & n = 0, 1, \dots N-1 \end{cases}$$

Remarque : si on utilise directement la relation de récurrence

$$q(h\beta) = 1 - \beta h \mathbf{b}^T(\mathbb{I} + \beta h \mathbb{A})^{-1}\mathbf{1} = 1 - \beta h b_1(1+\beta h a_{11})^{-1} = 1 - \beta h(1+\beta h c_1)^{-1} = 1 - \frac{\beta h}{1+c_1\beta h},$$

ce qui correspond au résultat trouvé précédemment.

Par conséquent, $\lim\limits_{n\to+\infty} u_n = 0$ si et seulement si

$$\left|1 - \frac{\beta h}{1+c_1\beta h}\right| < 1.$$

Notons x le produit $\beta h > 0$ (donc $x > 0$) et q la fonction $q(x) = 1 - \frac{x}{1+c_1 x}$ (avec $c_1 \in [0;1]$).

$$\begin{aligned} |q(x)| < 1 \quad &\iff \quad -1 < 1 - \frac{x}{1+c_1x} < 1 \quad \iff \quad \begin{cases} \frac{x}{1+c_1x} < 2 \\ \frac{x}{1+c_1x} > 0 \end{cases} \\ &\overset{\substack{x>0\\ c_1\geq 0}}{\iff} \quad \frac{x}{1+c_1x} < 2 \quad \overset{c_1x\geq 0}{\iff} \quad x < 2(1+c_1x) \\ &\iff \quad (1-2c_1)x < 2 \quad \iff \quad \begin{cases} x < \frac{2}{1-2c_1} & \text{si } 1-2c_1 > 0 \\ \forall x & \text{sinon.} \end{cases} \end{aligned}$$

Conclusion :

- si $c_1 \geq \frac{1}{2}$ le schéma est inconditionnellement A-stable,
- si $c_1 < \frac{1}{2}$ le schéma est A-stable ssi $\beta h < \frac{2}{1-2c_1}$.

Pour nos exemples, on obtient :

- si $c_1 = 1$ (Euler Implicite) alors le schéma est inconditionnellement A-stable et d'ordre 1,
- si $c_1 = \frac{1}{2}$ (RK1_M) alors le schéma est inconditionnellement A-stable et d'ordre 2,
- si $c_1 = 0$ (Euler Explicite) alors le schéma est A-stable ssi $h < \frac{2}{\beta}$ et il est d'ordre 1.

```
# Vérifions que pour s=1, on retrouve bien la formule de q
q = calculer_q_rk_general(1) # implicite
display(Markdown(f"$$q(x) = {sp.latex(q)}$$"))
```

$$q(x) = \frac{a_{11}x - b_1x + 1}{a_{11}x + 1}$$

11.6. Exercice : schémas RK à 2 étages

Étudier les méthodes RK avec $s = 2$ consistantes :

1. écrire le schéma **implicite** générique RK à deux étages

2. écrire le schéma **semi-implicite** générique RK à deux étages
3. écrire le schéma **explicite** générique RK à deux étages
4. étudier l'ordre de convergences des schémas explicites
5. étudier la A-stabilité des schémas explicites

Les méthodes avec $s = 2$ consistantes ont pour matrice de Butcher

$$\begin{array}{c|cc} c_1 & a_{11} & a_{12} \\ c_2 & a_{21} & a_{22} \\ \hline & b_1 & b_2 \end{array} \quad \text{avec} \quad \begin{cases} c_1 = a_{11} + a_{12} \\ c_2 = a_{21} + a_{22} \\ 1 = b_1 + b_2 \end{cases}$$

ce qui nous donne les schémas suivants :

$$\begin{array}{c|cc} c_1 & c_1 - a_{12} & a_{12} \\ c_2 & c_2 - a_{22} & a_{22} \\ \hline & 1 - b_2 & b_2 \end{array} \quad \rightsquigarrow \quad \begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n + hc_1, u_n + h((c_1 - a_{12})a_{11}K_1 + a_{12}K_2)\right) \\ K_2 = \varphi\left(t_n + hc_2, u_n + h(a_{21}K_1 + (c_2 - a_{22})K_2)\right) \\ u_{n+1} = u_n + h\left((1 - b_2)K_1 + b_2K_2\right) \qquad n = 0, 1, \dots N-1 \end{cases}$$

Donc, on a bien $s^2 + 2s - (s + 1) = 5$ degrés de liberté.

Pour l'implémentation, il s'agit de résoudre un système non linéaire de dimension 2 pour trouver les K_i. Les inconnues étant des évaluations de φ, lorsqu'on utilise une méthode de point fixe pour trouver les K_i il faut indiquer comme point de départ une évaluation de φ en des points connus, par exemple $\varphi(t_n, u_n)$.

```
def RK(tt,phi,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    uu[0] = y0
    for n in range(len(tt)-1):
        # X = [K1,K2]
        sys = lambda X : [ -X[0]+phi( tt[n]+h*c1 , uu[n]+h*(a11*X[0]+a12*X[1]) ) ,
                           -X[1]+phi( tt[n]+h*c2 , uu[n]+h*(a21*X[0]+a22*X[1]) )
                           ]
        K_start = phi(tt[n],uu[n])
        K1, K2 = fsolve( sys , [K_start,K_start ] )
        uu[n+1] = uu[n]+h*(b1*K1+b2*K2)
    return uu
```

Voici un exemple, appelé méthode de Gauss :

$$\begin{array}{c|cc} \frac{1}{2} - \gamma & \frac{1}{4} & \frac{1}{4} - \gamma \\ \frac{1}{2} + \gamma & \frac{1}{4} + \gamma & \frac{1}{4} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array} \qquad \text{avec } \gamma = \frac{\sqrt{3}}{6} = \frac{1}{2\sqrt{3}}.$$

Cette méthode est d'ordre 4 et on peut monter que c'est la seule méthode d'ordre 4 à deux étages.

```
gamma = sp.sqrt(3)/6
c = [ sp.Rational(1,2)-gamma, sp.Rational(1,2)+gamma ]
b = [ sp.Rational(1,2), sp.Rational(1,2) ]
A = [ [ sp.Rational(1,4), sp.Rational(1,4)-gamma ] ,
      [ sp.Rational(1,4)+gamma, sp.Rational(1,4) ] ]

s = len(c)
# ordre_RK(s=s)
ordre_RK(s=s,type="implicite",A=A,b=b,c=c);
```

Matrice de Butcher

$$\begin{bmatrix} \frac{1}{2}-\frac{\sqrt{3}}{6} & \frac{1}{4} & \frac{1}{4}-\frac{\sqrt{3}}{6} \\ \frac{\sqrt{3}}{6}+\frac{1}{2} & \frac{1}{4}+\frac{\sqrt{3}}{6} & \frac{1}{4} \\ & \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

On a 3 conditions pour avoir consistance (= pour être d'ordre 1)

True

True

True

On doit ajouter 1 condition pour être d'ordre 2

True

On doit ajouter 2 conditions pour être d'ordre 3

True

True

On doit ajouter 4 conditions pour être d'ordre 4

True

True

True

True

11.6.1. Schémas semi-implicites

Les méthodes semi-implicites vérifient, de plus, $a_{12} = 0$ ainsi la matrice de Butcher est

$$\begin{array}{c|cc} c_1 & c_1 & 0 \\ c_2 & c_2 - a_{22} & a_{22} \\ \hline & 1-b_2 & b_2 \end{array} \quad \Longrightarrow \quad \begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n + hc_1, u_n + hc_1 K_1\right) \\ K_2 = \varphi\left(t_n + hc_2, u_n + h((c_2 - a_{22})K_1 + a_{22}K_2)\right) \\ u_{n+1} = u_n + h\left((1-b_2)K_1 + b_2 K_2\right) \end{cases} \qquad n = 0, 1, \ldots N-1$$

Pour l'implémentation, il s'agit de résoudre des équations non linéaires séparément pour trouver les K_i. Les inconnues étant des évaluations de φ, lorsqu'on utilise une méthode de point fixe pour trouver les K_i il faut indiquer comme point de départ une évaluation de φ en des points connus, par exemple $\varphi(t_n, u_n)$.

```
def RK(tt,phi,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    uu[0] = y0
    for n in range(len(tt)-1):
        # X = [K1,K2]
        K_start = phi(tt[n],uu[n])
        K1 = fsolve( lambda x : -x+phi( tt[n]+h*c1 , uu[n]+h*(a11*x)        ) , K_start ) [0]
        K2 = fsolve( lambda x : -x+phi( tt[n]+h*c2 , uu[n]+h*(a21*K1+a22*x) ) , K_start ) [0]
        uu[n+1] = uu[n]+h*(b1*K1+b2*K2)
    return uu
```

Par exemple, si on choisit $c_1 = 0$, $c_2 = 1$ et $a_{22} = b_2 = \frac{1}{2}$ on trouve le schéma semi-implicite

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & \frac{1}{2} & \frac{1}{2} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array} \quad \Longrightarrow \quad \begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n, u_n\right) \\ K_2 = \varphi\left(t_{n+1}, u_n + \frac{h}{2}(K_1 + K_2)\right) \\ u_{n+1} = u_n + \frac{h}{2}(K_1 + K_2) \qquad\qquad n = 0, 1, \dots N-1 \end{cases}$$

Notons que ce schéma n'est rien d'autre que le schéma de Crank-Nicolson car $u_{n+1} = u_n + \frac{h}{2}(K_1 + K_2)$, donc si on remplace le $u_n + \frac{h}{2}(K_1 + K_2)$ dans la définition de K_2 par u_{n+1} on obtient

$$K_2 = \varphi\left(t_{n+1}, u_n + \frac{h}{2}(K_1 + K_2)\right) = \varphi\left(t_{n+1}, u_{n+1}\right)$$

ainsi

$$u_{n+1} = u_n + \frac{h}{2}(K_1 + K_2) = u_n + \frac{h}{2}(\varphi\left(t_n, u_n\right) + \varphi\left(t_{n+1}, u_{n+1}\right))$$

Il s'agit donc d'un schéma d'ordre 2.

```
c = [0, 1]
b = [sp.Rational(1,2), sp.Rational(1,2)]
A = [[0, 0], [sp.Rational(1,2), sp.Rational(1,2)]]

s = len(c)
# ordre_RK(s=s)
ordre_RK( s=s, type="implicite", A=A, b=b, c=c );
```

Matrice de Butcher

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & \frac{1}{2} & \frac{1}{2} \\ & \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

On a 3 conditions pour avoir consistance (= pour être d'ordre 1)

True

True

True

On doit ajouter 1 condition pour être d'ordre 2

True

On doit ajouter 2 conditions pour être d'ordre 3

False

False

On doit ajouter 4 conditions pour être d'ordre 4

False

False

False

False

Un autre exemple est la méthode dite DIRK (Diagonally Implicit Runge-Kutta) qui est d'ordre 3 :

$$\begin{array}{c|cc} \frac{1}{3} & \frac{1}{3} & 0 \\ 1 & 1 & 0 \\ \hline & \frac{3}{4} & \frac{1}{4} \end{array} \quad \Longrightarrow \quad \begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n + \frac{1}{3}h, u_n + \frac{1}{3}hK_1\right) \\ K_2 = \varphi\left(t_{n+1}, u_n + hK_1\right) \\ u_{n+1} = u_n + \frac{h}{4}(3K_1 + K_2) \qquad n = 0, 1, \dots N-1 \end{cases}$$

```
c = [sp.Rational(1,3), 1]
b = [sp.Rational(3,4), sp.Rational(1,4)]
A = [[sp.Rational(1,3),0],[1,0]]

ss = len(c)
# ordre_RK(s=s)
ordre_RK( s=s, A=A, b=b, c=c );
```

Matrice de Butcher

$$\begin{bmatrix} \frac{1}{3} & \frac{1}{3} & 0 \\ 1 & 1 & 0 \\ & \frac{3}{4} & \frac{1}{4} \end{bmatrix}$$

On a 3 conditions pour avoir consistance (= pour être d'ordre 1)

True

True

True

On doit ajouter 1 condition pour être d'ordre 2

True

On doit ajouter 2 conditions pour être d'ordre 3

True

True

On doit ajouter 4 conditions pour être d'ordre 4

False

False

False

False

11.6.2. Schémas explicites

Les méthodes avec $s = 2$ explicites ont pour matrice de Butcher

$$\begin{array}{c|cc} 0 & 0 & 0 \\ c_2 & c_2 & 0 \\ \hline & 1-b_2 & b_2 \end{array} \quad \Longrightarrow \quad \begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n, u_n\right) \\ K_2 = \varphi\left(t_n + hc_2, u_n + hc_2K_1\right) \\ u_{n+1} = u_n + h\left((1-b_2)K_1 + b_2K_2\right) \quad n = 0, 1, \dots N-1 \end{cases}$$

```
def RK(tt,phi,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    uu[0] = y0
    for n in range(len(tt)-1):
        K1 = phi( tt[n], uu[n] )
        K2 = phi( tt[n]+h*c2 , uu[n]+h*(c2*K1) )
        uu[n+1] = uu[n]+h*(b1*K1+b2*K2)
    return uu
```

Un schéma RK à deux étage explicite est au mieux d'ordre 2. Pour avoir l'ordre 2 il faut que $b_2c_2 = \frac{1}{2}$ ainsi il s'écrit

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \frac{1}{2\alpha} & \frac{1}{2\alpha} & 0 \\ \hline & 1-\alpha & \alpha \end{array} \quad \Longrightarrow \quad \begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n, u_n\right) \\ K_2 = \varphi\left(t_n + \frac{h}{2}\alpha, u_n + \frac{h}{2}\alpha K_1\right) \\ u_{n+1} = u_n + h\left((1-\alpha)K_1 + \alpha K_2\right) \quad n = 0, 1, \dots N-1 \end{cases}$$

Voici quelques cas particuliers :

- $\alpha = \frac{1}{2}$: schéma de Heun

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array} \quad \Longrightarrow \quad \begin{cases} u_0 = y_0 \\ K_1 = \varphi(t_n, u_n) \\ K_2 = \varphi(t_{n+1}, u_n + hK_1) \\ u_{n+1} = u_n + h\left(\frac{1}{2}K_1 + \frac{1}{2}K_2\right) \quad n = 0, 1, \dots N-1 \end{cases}$$

- $\alpha = 1$: schéma d'Euler Modifié (ou du point milieu)

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \hline & 0 & 1 \end{array} \quad \Longrightarrow \quad \begin{cases} u_0 = y_0 \\ K_1 = \varphi(t_n, u_n) \\ K_2 = \varphi(t_n + \frac{1}{2}h, u_n + \frac{1}{2}hK_1) \\ u_{n+1} = u_n + hK_2 \qquad\qquad n = 0, 1, \dots N-1 \end{cases}$$

- $\alpha = \frac{2}{3}$: schéma de Ralston

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \frac{3}{4} & \frac{3}{4} & 0 \\ \hline & \frac{1}{3} & \frac{2}{3} \end{array} \quad \Longrightarrow \quad \begin{cases} u_0 = y_0 \\ K_1 = \varphi(t_n, u_n) \\ K_2 = \varphi(t_n + \frac{3}{4}h, u_n + \frac{3}{4}hK_1) \\ u_{n+1} = u_n + \frac{h}{3}(K_1 + 2K_2) \qquad n = 0, 1, \dots N-1 \end{cases}$$

11.6.3. Étude de la A-stabilité des schémas explicites

Notons x le produit $\beta h > 0$ (donc $x > 0$).

Le schéma RK explicite à 2 étages appliqué à cette équation devient

$$\begin{cases} u_0 = y_0 \\ K_1 = -\beta u_n \\ K_2 = -\beta(u_n + hc_2K_1) \\ u_{n+1} = u_n + h\left((1-b_2)K_1 + b_2K_2\right) \quad n = 0, 1, \dots N-1 \end{cases}$$

ainsi

$$K_2 = -\beta u_n - \beta h c_2 K_1 = K_1 - \beta h c_2 K_1$$

et

$$\begin{aligned} u_{n+1} &= u_n + h(1-b_2)K_1 + hb_2K_2 \\ &= u_n + h(1-b_2)K_1 + hb_2(K_1 - \beta h c_2 K_1) \\ &= u_n + hK_1 - \beta h^2 b_2 c_2 K_1 \\ &= u_n + (1 - \beta h b_2 c_2) h K_1 \\ &= u_n + (1 - \beta h b_2 c_2) h(-\beta u_n) \\ &= (1 - \beta h + (\beta h)^2 b_2 c_2) u_n \end{aligned}$$

et enfin

$$\begin{cases} u_0 = y_0 \\ u_{n+1} = \left(1 - (\beta h) + c_2 b_2 (\beta h)^2\right) u_n \quad n = 0, 1, \dots N-1 \end{cases}$$

Par conséquent, $\lim_{n\to+\infty} u_n = 0$ si et seulement si

$$\left|1 - (\beta h) + c_2 b_2 (\beta h)^2\right| < 1.$$

Remarque : pour trouver cette relation de récurrence, on peut aussi utiliser directement la formule générale $q(x) = 1 - x\mathbf{b}^T(\mathbb{I} + x\mathbb{A})^{-1}\mathbf{1}$ ayant posé $x = \beta h$. La matrice $(\mathbb{I} + x\mathbb{A})$ est

$$\mathbb{I} + x\mathbb{A} = \begin{pmatrix} 1 & 0 \\ xc_2 & 1 \end{pmatrix}.$$

Puisque c'est une matrice triangulaire avec des 1 sur la diagonale, son déterminant est 1 et son inverse est immédiat :

$$(\mathbb{I} + x\mathbb{A})^{-1} = \begin{pmatrix} 1 & 0 \\ -xc_2 & 1 \end{pmatrix}.$$

On développe le produit matriciel dans la formule $q(x) = 1 - x\mathbf{b}^T(\mathbb{I} + x\mathbb{A})^{-1}\mathbf{1}$:

$$\begin{aligned} q(x) &= 1 - x \begin{pmatrix} b_1 & b_2 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ -xc_2 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\ &= 1 - x \begin{pmatrix} b_1 & b_2 \end{pmatrix} \begin{pmatrix} 1 \\ 1 - xc_2 \end{pmatrix} \\ &= 1 - x\left(b_1 + b_2(1 - xc_2)\right) \\ &= 1 - x(b_1 + b_2) + x^2(b_2 c_2) \\ &= 1 - x + x^2(b_2 c_2). \end{aligned}$$

Notons q le polynôme $q(x) = 1 - x + c_2 b_2 x^2$.

- Si $c_2 b_2 = 0$, le schéma se réduit au schéma d'Euler explicite.
- Si $c_2 b_2 > 0$, c'est une parabole convexe et le sommet est situé en $\left(\frac{1}{2c_2 b_2} > 0, 1 - \frac{1}{4c_2 b_2}\right)$.

$$|q(x)| < 1 \quad \iff \quad x < \frac{1}{c_2 b_2}.$$

  En particulier, lorsque le schéma est d'ordre 2 on a $b_2 c_2 = \frac{1}{2}$ et le schéma est A-stable ssi $h < \frac{2}{\beta}$.
- Si $c_2 b_2 < 0$, c'est une parabole concave et le sommet est situé en $\left(\frac{1}{2c_2 b_2} < 0, 1 - \frac{1}{4c_2 b_2}\right)$.

$$|q(x)| < 1 \quad \iff \quad x < \frac{1 - \sqrt{1 - 8b_2 c_2}}{2b_2 c_2}.$$

```
# q = calculer_q_rk_general(2) # implicite

# explicite
c_2, b_2 = sp.symbols('c_2 b_2')
A = sp.Matrix([[0, 0], [c_2, 0]])
b = sp.Matrix([1 - b_2, b_2])
display(Markdown(f"$$A = {sp.latex(A)},\\qquad b^T = {sp.latex(b.T)}$$"))
q = calculer_q_rk_general(2, A=A, b=b)
display(Markdown(f"$$q(x) = {sp.latex(q)}$$"))
```

$$A = \begin{bmatrix} 0 & 0 \\ c_2 & 0 \end{bmatrix}, \qquad b^T = \begin{bmatrix} 1 - b_2 & b_2 \end{bmatrix}$$

$$q(x) = b_2 c_2 x^2 - x + 1$$

11.7. Quelques schémas RK à 3 étages explicites

Un schéma RK à 3 étages explicite a pour matrice de Butcher

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ c_2 & a_{21} & 0 & 0 \\ c_3 & a_{31} & a_{32} & 0 \\ \hline & b_1 & b_2 & b_3 \end{array} \quad \text{avec} \quad \begin{cases} c_2 = a_{21} \\ c_3 = a_{31} + a_{32} \\ b_1 + b_2 + b_3 = 1 \end{cases} \quad \Longrightarrow \quad \begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n, u_n\right) \\ K_2 = \varphi\left(t_n + hc_2, u_n + hc_2 K_1\right) \\ K_3 = \varphi\left(t_n + hc_3, u_n + h(a_{31}K_1 + a_{32}K_2)\right) \\ u_{n+1} = u_n + h\left(b_1 K_1 + b_2 K_2 + b_3 K_3\right) \qquad n = 0, 1, \ldots N-1 \end{cases}$$

Voici trois exemples :

- schéma de Heun à trois étages

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & 0 \\ \frac{2}{3} & 0 & \frac{2}{3} & 0 \\ \hline & \frac{1}{4} & 0 & \frac{3}{4} \end{array} \quad \Longrightarrow \quad \begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n, u_n\right) \\ K_2 = \varphi\left(t_n + \frac{h}{3}, u_n + \frac{h}{3}K_1\right) \\ K_3 = \varphi\left(t_n + \frac{2}{3}h, u_n + \frac{2}{3}hK_2\right) \\ u_{n+1} = u_n + \frac{h}{4}\left(K_1 + 3K_3\right) \qquad n = 0, 1, \ldots N-1 \end{cases}$$

  Étant une méthode explicite à 3 étapes, elle est au mieux d'ordre 3. Vérifions qu'elle est effectivement d'ordre 3 :

```
c = [0, sp.Rational(1,3), sp.Rational(2,3)]
b = [sp.Rational(1,4), 0, sp.Rational(3,4)]
A = [[0, 0, 0],
     [sp.Rational(1,3), 0, 0],
     [0, sp.Rational(2,3), 0]]

s = len(c)
# ordre_RK(s=s)
ordre_RK(s=s,type="explicite",A=A,b=b,c=c);
```

Matrice de Butcher

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & 0 \\ \frac{2}{3} & 0 & \frac{2}{3} & 0 \\ & \frac{1}{4} & 0 & \frac{3}{4} \end{bmatrix}$$

On a 4 conditions pour avoir consistance (= pour être d'ordre 1)

True

True

True

True

On doit ajouter 1 condition pour être d'ordre 2

True

On doit ajouter 2 conditions pour être d'ordre 3

True

True

```
# Calculons le facteur q(beta h)
b = sp.Matrix(b)
A = sp.Matrix(A)
q = calculer_q_rk_general(s=s,A=A,b=b)
display(Markdown(f"$$q(x) = {sp.latex(q)}$$"))
```

$$q(x) = -\frac{x^3}{6} + \frac{x^2}{2} - x + 1$$

- schéma de Kutta

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 1 & -1 & 2 & 0 \\ \hline & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{array} \quad \Longrightarrow \quad \begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n, u_n\right) \\ K_2 = \varphi\left(t_n + \frac{h}{2}, u_n + \frac{h}{2}K_1\right) \\ K_3 = \varphi\left(t_{n+1}, u_n + h(-K_1 + 2K_2)\right) \\ u_{n+1} = u_n + \frac{h}{6}\left(K_1 + 4K_2 + K_3\right) \qquad n = 0, 1, \ldots N-1 \end{cases}$$

Étant une méthode explicite à 3 étages, elle a au mieux d'ordre 3. On vérifie ci-dessous qu'elle est d'ordre 3 :

```
c = [0, sp.S(1)/2, 1]
b = [sp.S(1)/6, sp.S(2)/3, sp.S(1)/6]
A = [[0, 0, 0],
     [sp.S(1)/2, 0, 0],
     [-1, 2, 0]]

s = len(c)
# ordre_RK(s=s)
ordre_RK(s=s,A=A,b=b,c=c);
```

Matrice de Butcher

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 1 & -1 & 2 & 0 \\ & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{bmatrix}$$

On a 4 conditions pour avoir consistance (= pour être d'ordre 1)

True

True

True

True

On doit ajouter 1 condition pour être d'ordre 2

True

On doit ajouter 2 conditions pour être d'ordre 3

True

True

On doit ajouter 4 conditions pour être d'ordre 4

True

False

True

False

```
# Calculons le facteur q(beta h)
b = sp.Matrix(b)
A = sp.Matrix(A)
q = calculer_q_rk_general(s=s,A=A,b=b)
display(Markdown(f"$$q(x) = {sp.latex(q)}$$"))
```

$$q(x) = -\frac{x^3}{6} + \frac{x^2}{2} - x + 1$$

- schéma

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 1 & -1 & 2 & 0 \\ \hline & -\frac{1}{6} & \frac{4}{3} & -\frac{1}{6} \end{array} \quad \Longrightarrow \quad \begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n, u_n\right) \\ K_2 = \varphi\left(t_n + \frac{h}{2}, u_n + \frac{h}{2} K_1\right) \\ K_3 = \varphi\left(t_{n+1}, u_n + h(-K_1 + 2K_2)\right) \\ u_{n+1} = u_n + \frac{h}{6}\left(-K_1 + 8K_2 - K_3\right) \qquad n = 0, 1, \ldots N-1 \end{cases}$$

  Étant une méthode explicite à 3 étages, elle a au mieux d'ordre 3. On vérifie ci-dessous qu'elle n'est que d'ordre 2 :

```
c = [0, sp.Rational(1,2), 1]
b = [-sp.Rational(1,6), sp.Rational(4,3), -sp.Rational(1,6)]
A = [[0, 0, 0], [sp.Rational(1,2), 0, 0], [-1, 2, 0]]

s = len(c)
# ordre_RK(s=s)
ordre_RK(s=s,A=A,b=b,c=c);
```

Matrice de Butcher

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 1 & -1 & 2 & 0 \\ & -\frac{1}{6} & \frac{4}{3} & -\frac{1}{6} \end{bmatrix}$$

On a 4 conditions pour avoir consistance (= pour être d'ordre 1)

True

True

True

True

On doit ajouter 1 condition pour être d'ordre 2

True

On doit ajouter 2 conditions pour être d'ordre 3

False

False

On doit ajouter 4 conditions pour être d'ordre 4

False

False

False

False

```
# Calculons le facteur q(beta h)
b = sp.Matrix(b)
A = sp.Matrix(A)
q = calculer_q_rk_general(s=s,A=A,b=b)
display(Markdown(f"$$q(x) = {sp.latex(q)}$$"))
```

$$q(x) = \frac{x^3}{6} + \frac{x^2}{2} - x + 1$$

11.8. Quelques schémas RK à 4 étages explicites

Un schéma RK à 4 étages explicite a pour matrice de Butcher

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ c_2 & a_{21} & 0 & 0 & 0 \\ c_3 & a_{31} & a_{32} & 0 & 0 \\ c_4 & a_{41} & a_{42} & a_{43} & 0 \\ \hline & b_1 & b_2 & b_3 & b_4 \end{array} \quad \text{avec} \quad \begin{cases} c_2 = a_{21} \\ c_3 = a_{31} + a_{32} \\ c_4 = a_{41} + a_{42} + a_{43} \\ b_1 + b_2 + b_3 + b_4 = 1 \end{cases} \quad \Longrightarrow \quad \begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n, u_n\right) \\ K_2 = \varphi\left(t_n + hc_2, u_n + h(a_{21}K_1)\right) \\ K_3 = \varphi\left(t_n + hc_3, u_n + h(a_{31}K_1 + a_{32}K_2)\right) \\ K_4 = \varphi\left(t_n + hc_4, u_n + h(a_{41}K_1 + a_{42}K_2 + a_{43}K_3)\right) \\ u_{n+1} = u_n + h\left(b_1K_1 + b_2K_2 + b_3K_3 + b_4K_4\right) \end{cases}$$

Voici deux exemples (le premier est le plus connus) :

- schéma RK4-1 (“La” méthode de Runge-Kutta)

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array} \quad \Longrightarrow \quad \begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n, u_n\right) \\ K_2 = \varphi\left(t_n + \frac{h}{2}, u_n + \frac{h}{2}K_1\right)\right) \\ K_3 = \varphi\left(t_n + \frac{h}{2}, u_n + \frac{h}{2}K_2\right) \\ K_4 = \varphi\left(t_{n+1}, u_n + hK_3\right) \\ u_{n+1} = u_n + \frac{h}{6}\left(K_1 + 2K_2 + 2K_3 + K_4\right) \quad n = 0, 1, \ldots N-1 \end{cases}$$

Il est bien d’ordre 4 :

```
c = [0, sp.Rational(1,2), sp.Rational(1,2), 1]
b = [sp.Rational(1,6), sp.Rational(1,3), sp.Rational(1,3), sp.Rational(1,6)]
A = [[0, 0, 0, 0],
    [sp.Rational(1,2), 0, 0, 0],
    [0, sp.Rational(1,2), 0, 0],
    [0, 0, 1, 0]]

s = len(c)
# ordre_RK(s=s)
ordre_RK(s=s,A=A,b=b,c=c);
```

Matrice de Butcher

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{bmatrix}$$

On a 5 conditions pour avoir consistance (= pour être d’ordre 1)

True

True

True

True

True

On doit ajouter 1 condition pour être d'ordre 2

True

On doit ajouter 2 conditions pour être d'ordre 3

True

True

On doit ajouter 4 conditions pour être d'ordre 4

True

True

True

True

```
# Calculons le facteur q(beta h)
b = sp.Matrix(b)
A = sp.Matrix(A)
q = calculer_q_rk_general(s=s,A=A,b=b)
display(Markdown(f"$$q(x) = {sp.latex(q)}$$"))
```

$$q(x) = \frac{x^4}{24} - \frac{x^3}{6} + \frac{x^2}{2} - x + 1$$

- schéma RK4-2

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 1 & 1 & -2 & 2 & 0 \\ \hline & \frac{1}{6} & 0 & \frac{2}{3} & \frac{1}{6} \end{array} \quad \Longrightarrow \quad \begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n, u_n\right) \\ K_2 = \varphi\left(t_n + \frac{h}{4}, u_n + \frac{h}{4}K_1\right)\right) \\ K_3 = \varphi\left(t_n + \frac{h}{2}, u_n + \frac{h}{2}K_2\right) \\ K_4 = \varphi\left(t_{n+1}, u_n + h(K_1 - 2K_2 + 2K_3)\right) \\ u_{n+1} = u_n + \frac{h}{6}\left(K_1 + 4K_3 + K_4\right) \qquad n = 0, 1, \dots N-1 \end{cases}$$

Il est bien d'ordre 4 :

```
c = [0, sp.Rational(1,4), sp.Rational(1,2), 1]
b = [sp.Rational(1,6), 0, sp.Rational(2,3), sp.Rational(1,6)]
A = [ [0, 0, 0, 0],
      [sp.Rational(1,4), 0, 0, 0],
      [0, sp.Rational(1,2), 0, 0],
      [1, -2, 2, 0] ]

s = len(c)
# ordre_RK(s=s)
ordre_RK(s=s,A=A,b=b,c=c);
```

Matrice de Butcher

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 1 & 1 & -2 & 2 & 0 \\ & \frac{1}{6} & 0 & \frac{2}{3} & \frac{1}{6} \end{bmatrix}$$

On a 5 conditions pour avoir consistance (= pour être d'ordre 1)

True

True

True

True

True

On doit ajouter 1 condition pour être d'ordre 2

True

On doit ajouter 2 conditions pour être d'ordre 3

True

True

On doit ajouter 4 conditions pour être d'ordre 4

True

True

True

True

```
# Calculons le facteur q(beta h)
b = sp.Matrix(b)
A = sp.Matrix(A)
q = calculer_q_rk_general(s=s,A=A,b=b)
display(Markdown(f"$$q(x) = {sp.latex(q)}$$"))
```

$$q(x) = \frac{x^4}{24} - \frac{x^3}{6} + \frac{x^2}{2} - x + 1$$

- schéma RK4-3 (règle 3/8)

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 \\ \frac{2}{3} & -\frac{1}{3} & 1 & 0 & 0 \\ 1 & 1 & -1 & 1 & 0 \\ \hline & \frac{1}{8} & \frac{3}{8} & \frac{3}{8} & \frac{1}{8} \end{array} \quad \Longrightarrow \quad \begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n, u_n\right) \\ K_2 = \varphi\left(t_n + \frac{h}{3}, u_n + \frac{h}{3}K_1\right) \\ K_3 = \varphi\left(t_n + \frac{2}{3}h, u_n + \frac{h}{3}(-K_1 + 3K_2)\right) \\ K_4 = \varphi\left(t_{n+1}, u_n + h(K_1 - K_2 + K_3)\right) \\ u_{n+1} = u_n + \frac{h}{8}\left(K_1 + 3K_2 + 3K_3 + K_4\right) \quad n = 0, 1, \dots N-1 \end{cases}$$

Il est bien d'ordre 4 :

```
c = [0, sp.Rational(1,3), sp.Rational(2,3), 1]
b = [sp.Rational(1,8), sp.Rational(3,8), sp.Rational(3,8), sp.Rational(1,8)]
A = [ [0, 0, 0, 0],
      [sp.Rational(1,3), 0, 0, 0],
      [-sp.Rational(1,3), 1, 0, 0],
      [1, -1, 1, 0] ]

s = len(c)
# ordre_RK(s=s)
ordre_RK(s=s,A=A,b=b,c=c);
```

Matrice de Butcher

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 \\ \frac{2}{3} & -\frac{1}{3} & 1 & 0 & 0 \\ 1 & 1 & -1 & 1 & 0 \\ & \frac{1}{8} & \frac{3}{8} & \frac{3}{8} & \frac{1}{8} \end{bmatrix}$$

On a 5 conditions pour avoir consistance (= pour être d'ordre 1)

True

True

True

True

True

On doit ajouter 1 condition pour être d'ordre 2

True

On doit ajouter 2 conditions pour être d'ordre 3

True

True

On doit ajouter 4 conditions pour être d'ordre 4

True

True

True

True

```
# Calculons le facteur q(beta h)
b = sp.Matrix(b)
A = sp.Matrix(A)
q = calculer_q_rk_general(s=s,A=A,b=b)
display(display(Markdown(f"$$q(x) = {sp.latex(q)}$$")))
```

$$q(x) = \frac{x^4}{24} - \frac{x^3}{6} + \frac{x^2}{2} - x + 1$$

```
None
```

11.9. Quelques schémas RK à 5 étages explicites

Un schéma RK à 5 étages explicite a pour matrice de Butcher

$$\begin{array}{c|ccccc} 0 & 0 & 0 & 0 & 0 & 0 \\ c_2 & a_{21} & 0 & 0 & 0 & 0 \\ c_3 & a_{31} & a_{32} & 0 & 0 & 0 \\ c_4 & a_{41} & a_{42} & a_{43} & 0 & 0 \\ c_5 & a_{51} & a_{52} & a_{53} & a_{54} & 0 \\ \hline & b_1 & b_2 & b_3 & b_4 & b_5 \end{array} \implies \begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n, u_n\right) \\ K_2 = \varphi\left(t_n + hc_2, u_n + h(a_{21}K_1)\right) \\ K_3 = \varphi\left(t_n + hc_3, u_n + h(a_{31}K_1 + a_{32}K_2)\right. \\ K_4 = \varphi\left(t_n + hc_4, u_n + h(a_{41}K_1 + a_{42}K_2 + a_{43}K_3)\right) \\ K_5 = \varphi\left(t_n + hc_5, u_n + h(a_{51}K_1 + a_{52}K_2 + a_{53}K_3 + a_{54}K_4)\right) \\ u_{n+1} = u_n + h\left(b_1K_1 + b_2K_2 + b_3K_3 + b_4K_4 + b_5K_5\right) \qquad n = 0, 1, \dots N-1 \end{cases}$$

avec

$$\begin{cases} c_2 = a_{21} \\ c_3 = a_{31} + a_{32} \\ c_4 = a_{41} + a_{42} + a_{43} \\ c_5 = a_{51} + a_{52} + a_{53} + a_{54} \\ b_1 + b_2 + b_3 + b_4 + b_5 = 1 \end{cases}$$

Voici un exemple :

- schéma de Merson

$$\begin{array}{c|ccccc} 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{6} & \frac{1}{6} & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{8} & 0 & \frac{3}{8} & 0 & 0 \\ 1 & \frac{1}{2} & 0 & -\frac{3}{2} & 2 & 0 \\ \hline & \frac{1}{6} & 0 & 0 & \frac{2}{3} & \frac{1}{6} \end{array} \Longrightarrow \begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n, u_n\right) \\ K_2 = \varphi\left(t_n + \frac{h}{3}, u_n + \frac{h}{3} K_1\right) \\ K_3 = \varphi\left(t_n + \frac{h}{3}, u_n + \frac{h}{6}(K_1 + K_2)\right) \\ K_4 = \varphi\left(t_n + \frac{h}{2}, u_n + \frac{h}{8}(K_1 + 3K_3)\right) \\ K_5 = \varphi\left(t_n + h, u_n + \frac{h}{2}(K_1 - 3K_3 + 4K_4)\right) \\ u_{n+1} = u_n + \frac{h}{6}\left(K_1 + 4K_4 + K_5\right) \end{cases} \qquad n = 0, 1, \ldots N-1$$

```
c = [0, sp.Rational(1,3), sp.Rational(1,3), sp.Rational(1,2), 1]
b = [sp.Rational(1,6), 0, 0, sp.Rational(2,3), sp.Rational(1,6)]
A = [[0, 0, 0, 0, 0], [sp.Rational(1,3), 0, 0, 0, 0], [sp.Rational(1,6), sp.Rational(1,6), 0, 0, 0],
 ↪  [sp.Rational(1,8), 0, sp.Rational(3,8), 0, 0], [sp.Rational(1,2), 0, -sp.Rational(3,2), 2, 0]]

s = len(c)
# ordre_RK(s=s)
ordre_RK(s=s,A=A,b=b,c=c);
```

Matrice de Butcher

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{6} & \frac{1}{6} & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{8} & 0 & \frac{3}{8} & 0 & 0 \\ 1 & \frac{1}{2} & 0 & -\frac{3}{2} & 2 & 0 \\ & \frac{1}{6} & 0 & 0 & \frac{2}{3} & \frac{1}{6} \end{bmatrix}$$

On a 6 conditions pour avoir consistance (= pour être d'ordre 1)

True

True

True

True

True

True

On doit ajouter 1 condition pour être d'ordre 2

True

On doit ajouter 2 conditions pour être d'ordre 3

True

True

On doit ajouter 4 conditions pour être d'ordre 4

True

True

True

True

```
# Calculons le facteur q(beta h)
b = sp.Matrix(b)
A = sp.Matrix(A)
q = calculer_q_rk_general(s=s,A=A,b=b)
display(Markdown(f"$$q(x) = {sp.latex(q)}$$"))
```

$$q(x) = -\frac{x^5}{144} + \frac{x^4}{24} - \frac{x^3}{6} + \frac{x^2}{2} - x + 1$$

11.10. Quelques schémas RK à 6 étages explicites

Un schéma RK à 6 étages explicite a pour matrice de Butcher

$$\begin{array}{c|cccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ c_2 & a_{21} & 0 & 0 & 0 & 0 & 0 \\ c_3 & a_{31} & a_{32} & 0 & 0 & 0 & 0 \\ c_4 & a_{41} & a_{42} & a_{43} & 0 & 0 & 0 \\ c_5 & a_{51} & a_{52} & a_{53} & a_{54} & 0 & 0 \\ c_6 & a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & 0 \\ \hline & b_1 & b_2 & b_3 & b_4 & b_5 & b_6 \end{array} \implies \begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n, u_n\right) \\ K_2 = \varphi\left(t_n + hc_2, u_n + h(a_{21}K_1)\right) \\ K_3 = \varphi\left(t_n + hc_3, u_n + h(a_{31}K_1 + a_{32}K_2\right) \\ K_4 = \varphi\left(t_n + hc_4, u_n + h(a_{41}K_1 + a_{42}K_2 + a_{43}K_3)\right) \\ K_5 = \varphi\left(t_n + hc_5, u_n + h(a_{51}K_1 + a_{52}K_2 + a_{53}K_3 + a_{54}K_4)\right) \\ K_6 = \varphi\left(t_n + hc_6, u_n + h(a_{61}K_1 + a_{62}K_2 + a_{63}K_3 + a_{64}K_4 + a_{65}K_5)\right) \\ u_{n+1} = u_n + h\left(b_1K_1 + b_2K_2 + b_3K_3 + b_4K_4 + b_5K_5 + b_6K_6\right) \end{cases} \quad n = 0, 1,$$

avec

$$\begin{cases} c_2 = a_{21} \\ c_3 = a_{31} + a_{32} \\ c_4 = a_{41} + a_{42} + a_{43} \\ c_5 = a_{51} + a_{52} + a_{53} + a_{54} \\ c_6 = a_{61} + a_{62} + a_{63} + a_{64} + a_{65} \\ b_1 + b_2 + b_3 + b_4 + b_5 + b_6 = 1 \end{cases}$$

Voici un exemple :

- schéma de Butcher (d'ordre 5)

$$\begin{array}{c|cccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{4} & \frac{1}{8} & \frac{1}{8} & 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & -\frac{1}{2} & 1 & 0 & 0 & 0 \\ \frac{3}{4} & \frac{3}{16} & 0 & 0 & \frac{9}{16} & 0 & 0 \\ 1 & \frac{-3}{7} & \frac{2}{7} & \frac{12}{7} & \frac{-12}{7} & \frac{8}{7} & 0 \\ \hline & \frac{7}{90} & 0 & \frac{32}{90} & \frac{12}{90} & \frac{32}{90} & \frac{7}{90} \end{array} \implies \begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n, u_n\right) \\ K_2 = \varphi\left(t_n + \frac{h}{4}, u_n + \frac{h}{4}K_1\right) \\ K_3 = \varphi\left(t_n + \frac{h}{4}, u_n + \frac{h}{8}(K_1 + K_2)\right) \\ K_4 = \varphi\left(t_n + \frac{h}{2}, u_n + \frac{h}{2}(-K_2 + 2K_3)\right) \\ K_5 = \varphi\left(t_n + \frac{3h}{4}, u_n + \frac{h}{16}(3K_1 + 9K_4)\right) \\ K_6 = \varphi\left(t_{n+1}, u_n + \frac{h}{7}(-3K_1 + 2K_2 + 12K_3 - 12K_4 + 8K_5)\right) \\ u_{n+1} = u_n + \frac{h}{90}\left(7K_1 + 32K_3 + 12K_4 + 32K_5 + 7K_6\right) \end{cases} \quad n = 0, 1, \ldots N - 1$$

```
c = [0, sp.Rational(1,4), sp.Rational(1,4), sp.Rational(1,2), sp.Rational(3,4), 1]
b = [sp.Rational(7,90), 0, sp.Rational(32,90), sp.Rational(12,90), sp.Rational(32,90), sp.Rational(7,90)]
A = [[0, 0, 0, 0, 0, 0],
     [sp.Rational(1,4), 0, 0, 0, 0, 0],
     [sp.Rational(1,8), sp.Rational(1,8), 0, 0, 0, 0],
```

```
     [0, sp.Rational(-1,2), 1, 0, 0, 0],
     [sp.Rational(3,16), 0, 0, sp.Rational(9,16), 0, 0],
     [sp.Rational(-3,7), sp.Rational(2,7), sp.Rational(12,7), sp.Rational(-12,7), sp.Rational(8,7), 0]]

s = len(c)
# ordre_RK(s=s)
ordre_RK(s=s,A=A,b=b,c=c);
```

Matrice de Butcher

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{4} & \frac{1}{8} & \frac{1}{8} & 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & -\frac{1}{2} & 1 & 0 & 0 & 0 \\ \frac{3}{4} & \frac{3}{16} & 0 & 0 & \frac{9}{16} & 0 & 0 \\ 1 & -\frac{3}{7} & \frac{2}{7} & \frac{12}{7} & -\frac{12}{7} & \frac{8}{7} & 0 \\ & \frac{7}{90} & 0 & \frac{16}{45} & \frac{2}{15} & \frac{16}{45} & \frac{7}{90} \end{bmatrix}$$

On a 7 conditions pour avoir consistance (= pour être d'ordre 1)

True

True

True

True

True

True

True

On doit ajouter 1 condition pour être d'ordre 2

True

On doit ajouter 2 conditions pour être d'ordre 3

True

True

On doit ajouter 4 conditions pour être d'ordre 4

True

True

True

True

```
# Calculons le facteur q(beta h)
b = sp.Matrix(b)
A = sp.Matrix(A)
q = calculer_q_rk_general(s=s,A=A,b=b)
display(Markdown(f"$$q(x) = {sp.latex(q)}$$"))
```

$$q(x) = \frac{x^6}{640} - \frac{x^5}{120} + \frac{x^4}{24} - \frac{x^3}{6} + \frac{x^2}{2} - x + 1$$

11.11. Méthodes adapatatives

La manière la plus simple pour appliquer une méthode de résolution numérique consiste à utiliser un pas h constant. La principale difficulté est alors de choisir *a priori* la valeur optimale de ce pas h : il faut qu'il soit suffisamment petit pour garantir une précision suffisante, mais pas trop petit pour éviter de surcharger inutilement le calcul. Par ailleurs, la solution que l'on cherche à approcher peut nécessiter un pas h petit à certains endroits et plus grand à d'autres. Ainsi, si l'on accepte l'idée d'utiliser une discrétisation non uniforme, on peut décider de construire une méthode qui choisit à chaque itération n la taille du pas d'intégration h, potentiellement différents à chaque itération. Pour ce faire, on a besoin d'estimer l'erreur ε_n que l'on commet lorsque l'on effectue une itération du schéma, afin de le comparer à une erreur « acceptable » "tol" : si $\varepsilon_n \leq \text{tol}$, on pourra augmenter la taille du pas d'intégration (si possible), et si au contraire $\varepsilon_n > \text{tol}$, la dernière étape effectuée n'est pas acceptable, la taille du pas est diminuée et on répète les deux calculs.

Ce procédé, appelé **adaptation du pas de discrétisation**, est efficace mais nécessite d'avoir un bon estimateur d'erreur locale. En général, il s'agit d'un estimateur d'erreur *a posteriori*, car les estimateurs d'erreur *a priori* sont trop compliqués en pratique. On peut construire l'estimateur d'erreur de deux manières :

1. en utilisant un unique schéma, mais avec deux pas de discrétisation différents ;
2. en utilisant deux schémas d'ordre différent.

Choix et ajustement du pas : dans la pratique, on impose un encadrement du pas de temps $h_{\min} \leq h_n \leq h_{\max}$, où $h_{\min}$ est fixé par les contraintes de temps de calcul et par la croissance des erreurs d'arrondi lorsque le pas devient trop petit, et $h_{\max}$ par des considérations de stabilité et de précision globale. À chaque itération, l'erreur locale est estimée par

$$\varepsilon_{n+1} = \frac{|u_{n+1} - \hat{u}_{n+1}|}{h_n}.$$

Étant donnée une tolérance prescrite $\text{tol} > 0$, le pas est ajusté selon les règles suivantes : - si $\varepsilon_{n+1} > \text{tol}$, le pas est jugé trop grand et l'étape est rejetée ; on diminue alors le pas, par exemple en posant

$$h_{n+1} = 0.8 h_n;$$

- si $\varepsilon_{n+1} \leq \text{tol}$, l'étape est acceptée ; si de plus ε_{n+1} est nettement plus petite que tol, on peut augmenter le pas pour la suite, par exemple en posant
$$h_{n+1} = 1.25 h_n;$$
- sinon, le pas est conservé inchangé.

11.11.1. CAS 1 : un unique schéma, mais avec deux pas de discrétisation différents

Une méthode courante pour garantir la précision dans la résolution d'un problème de Cauchy est de résoudre le problème deux fois en utilisant des pas h et $h/2$, puis de comparer les approximations obtenues aux points de maillage correspondant à la grille la plus grossière. Autrement dit, à chaque étape, deux approximations différentes de la solution sont calculées

- on calcule u_{n+1} avec un pas h donné,
- on calcule également $\hat{u}_{n+1}$ avec un pas $h/2$ de la même méthode (ce qui nécessite deux étapes pour que cela corresponde à l'approximation au même point $t_n + h$) ;

L'erreur est donc estimée par la différence entre ces deux approximations :

$$\varepsilon_{n+1} = \frac{|u_{n+1} - \hat{u}_{n+1}|}{h}.$$

Selon la valeur de ε_{n+1}, on peut alors décider d'augmenter h ou de le diminuer pour la prochaine itération.

11.11.1.1. Exemple : méthode d'Euler explicite à pas adaptatif

La méthode d'Euler explicite est bien adaptée à un calcul dynamique du pas de discrétisation h tenant compte des variations de l'inconnue sur l'intervalle d'intégration. Ici, l'estimateur d'erreur peut être construit à l'aide de deux pas de discrétisation (typiquement h et $h/2$).

Voyons un exemple d'implémentation de la méthode d'Euler explicite à pas adaptatif.

```
def EE( phi, tt, y0 ):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    uu[0] = y0
    for n in range(len(tt)-1):
        uu[n+1] = uu[n]+h*phi(tt[n],uu[n])
    return uu

def Adaptatif_EE(phi, t0, tfinal, y0, tol, hinit, hmin, hmax):
    tt = [t0]
    uu = [y0]
    h = hinit

    while tt[-1] < tfinal:
        t = tt[-1]
        u = uu[-1]

        # Un pas d'Euler (h)
        u_gr = u + h * phi(t, u)

        # Deux pas d'Euler (h/2)
        v = u + (h/2) * phi(t, u)
        u_fi = v + (h/2) * phi(t + h/2, v)

        # Estimation de l'erreur
        err = abs(u_gr - u_fi) / h

        if err > tol:
            # Solution refusée et adaptation du pas
            h = max(0.8 * h, hmin)
        else:
            # Solution acceptée
            tt.append(t + h)
            uu.append(u_fi)
            # Adaptation du pas
            if err < tol / 3:
                h = min(1.25 * h, hmax)
            # sinon h inchangé

    return np.array(tt), np.array(uu)


# ================================
# MAIN
# ================================

t0, tfinal = 0, 2
y0 = 0
phi = lambda t,y : 1+y
sol_exacte = lambda t : np.exp(t)-1


# ================================
# Comparaison des méthodes
# ================================
```

```
plt.figure(figsize=(15,5))

# exacte
tt_ex = np.linspace(t0,tfinal,101)
yy = sol_exacte(tt_ex)
plt.plot(tt_ex, yy, 'r',label=r'Exacte')
plt.xlabel('t')
plt.ylabel('y(t)')


# Pas fixe h
tt = np.linspace(t0,tfinal,11)
h_init = tt[1]-tt[0]

uu_EE = EE(phi, tt, y0)
plt.plot(tt,uu_EE, 'x', label='EE avec h')

# Pas fixe h/2
tt2 = np.linspace(t0,tfinal,21)
uu_EE2 = EE(phi, tt2, y0)
plt.plot(tt2,uu_EE2, 'd', label='EE avec h/2')

# Pas adaptatif
tol = h_init/2
tt_adapt, uu_adapt = Adaptatif_EE(phi, t0, tfinal, y0, tol=tol, hinit=h_init, hmin=h_init/10,
 ↪  hmax=4*h_init)
plt.plot(tt_adapt,uu_adapt, 'o', label='Adaptatif_EE')

plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
plt.grid();


# ================================
# Graphe de comparaison des erreurs
# ================================


err_EE   = np.abs(uu_EE  - sol_exacte(tt))
err_EE2  = np.abs(uu_EE2 - sol_exacte(tt2))
err_adapt = np.abs(uu_adapt - sol_exacte(tt_adapt))


plt.figure(figsize=(15,5))

plt.plot(tt, err_EE, 'x-', label='Erreur EE (h)')
plt.plot(tt2, err_EE2, 'd-', label='Erreur EE (h/2)')
plt.plot(tt_adapt, err_adapt, 'o-', label='Erreur Adaptatif')

plt.xlabel("t_n")
plt.ylabel(r"$|u(t_n)-y(t_n)|$")
plt.title("Comparaison des erreurs en fonction du temps")
plt.grid(True, which="both")
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()


# ================================
# Comparaison des grilles
# ================================
plt.figure(figsize=(15,5))
plt.plot(tt,np.ones_like(tt)*3,'x',label='Grille grossière')
plt.plot(tt2,np.ones_like(tt2)*2,'d',label='Grille fine')
plt.plot(tt_adapt,np.ones_like(tt_adapt),'o',label='Grille adaptative')
```

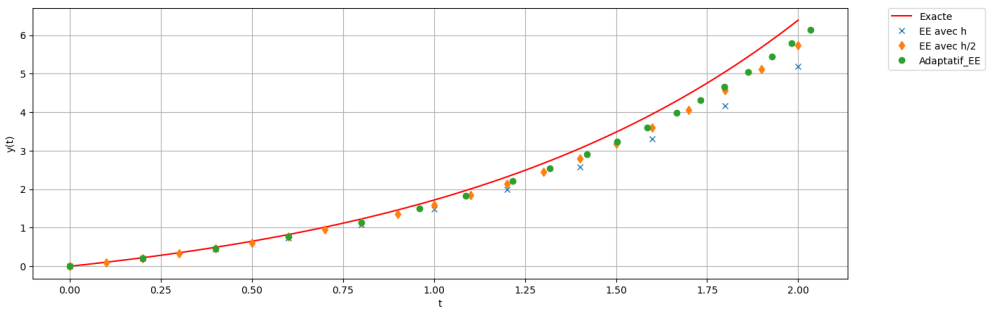
```
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
plt.ylim(0,4)
plt.xlabel('t')
plt.xticks(tt2)
plt.yticks([1,2,3], ['Adaptative','Fine','Grossière'])
plt.grid()
plt.show()

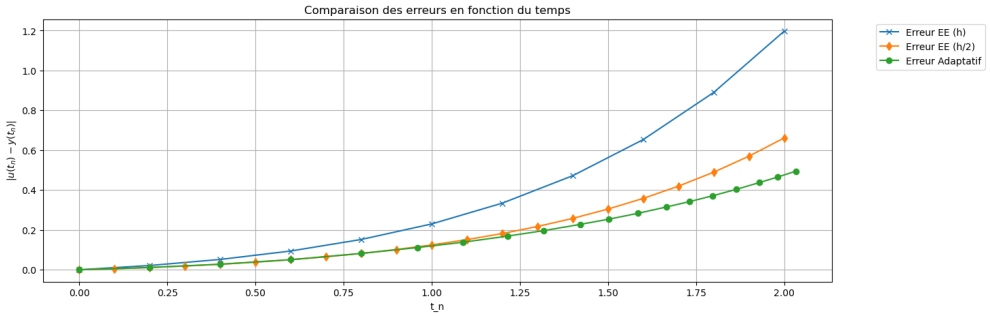

# ================================================================
# Évolution du pas adaptatif
# ================================================================
plt.figure(figsize=(8,5))
plt.plot(tt_adapt[1:], np.diff(tt_adapt), 'o-')
plt.xlabel("$t$")
plt.ylabel("$h$")
plt.title("Évolution du pas adaptatif")
plt.grid(True)
plt.show()

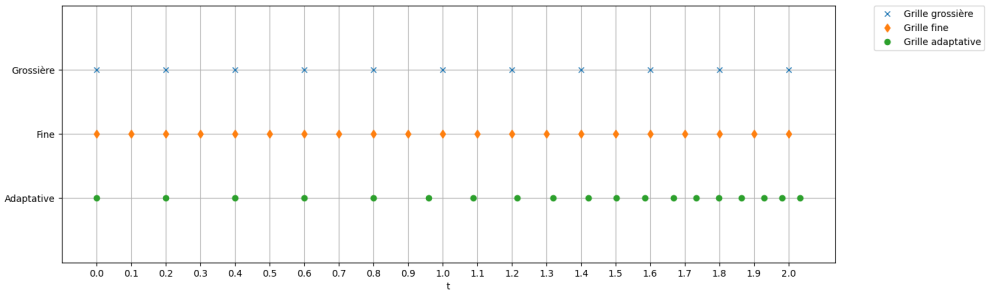
# ================================================================
# Comparaison des approximations en t=tfinal
# ================================================================

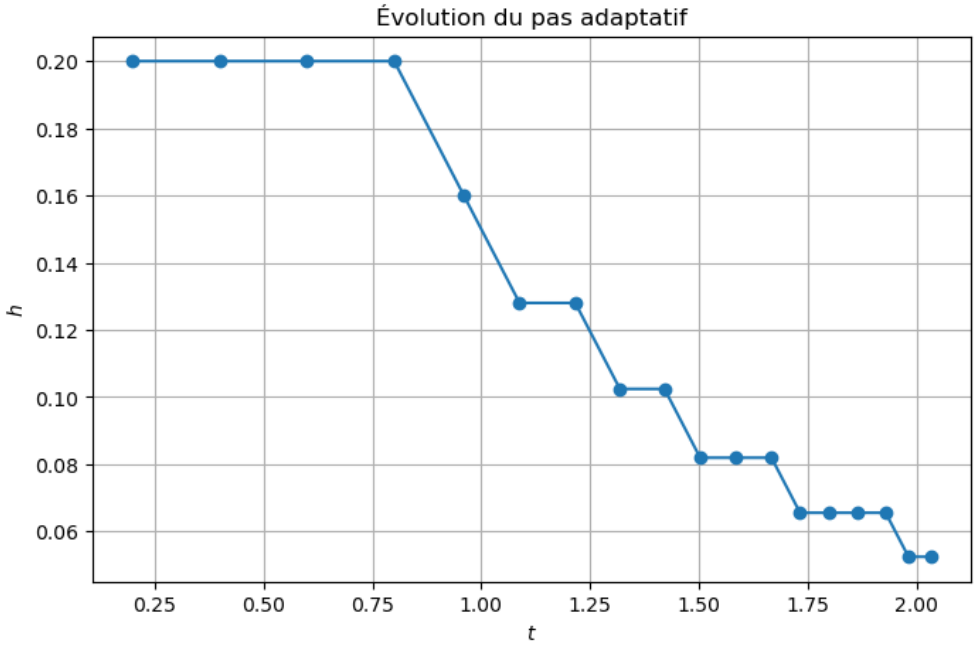
from tabulate import tabulate
tab = []
headers = ["Méthode", "t final", "|u(t final) - sol_exacte(tfinal)|", "Points de grille"]
tab.append(["EE avec h", tt[-1], abs(uu_EE[-1]-sol_exacte(tt[-1])), len(tt)])
tab.append(["EE avec h/2", tt2[-1], abs(uu_EE2[-1]-sol_exacte(tt2[-1])), len(tt2)])

idx = np.argmax(tt_adapt>tfinal)-1
tab.append(["Adaptatif_EE", tt_adapt[idx], abs(uu_adapt[idx]-sol_exacte(tt_adapt[idx])), len(tt_adapt)])
# tab.append(["Exacte", tfinal, sol_exacte(tfinal), ""])
display(tabulate(tab, headers=headers, tablefmt="html", floatfmt=".2e"))
```









Méthode	t final	|u(t final) - sol_exacte(tfinal)|	Points de grille
EE avec h	2.00e+00	1.20e+00	11
EE avec h/2	2.00e+00	6.62e-01	21
Adaptatif_EE	1.98e+00	4.65e-01	19

11.11.2. CAS 2 : deux schémas d'ordre différent (méthode de Runge-Kutta emboîtée)

Tout comme la méthode d'Euler explicite, les méthodes RK étant à un pas se prêtent bien aux techniques d'adaptation. On peut construire leur estimateur d'erreur de deux manières :

- en utilisant un schéma RK du même ordre, mais avec deux pas de discrétisation différents (comme pour la méthode d'Euler ci-dessus) ;
- en utilisant deux schémas RK d'ordre différent, mais avec le même nombre s d'étages et les mêmes coefficients a_{ij} et c_i (seules les coefficients b_i changent entre les deux méthodes).

La méthode de Runge-Kutta emboîtée utilise cette deuxième approche. L'intérêt de ces deux formules emboîtées est de permettre une estimation de l'erreur locale facilement calculable :

$$\varepsilon_{n+1} = |u_{n+1} - \hat{u}_{n+1}| = h \left| \sum_{i=1}^{s} (b_i - \hat{b}_i) K_i \right|$$

11.11.2.1. Méthode de Runge-Kutta adaptative Euler Explicite/Heun (RK12)

La méthode de Runge-Kutta adaptative la plus simple implique de combiner la méthode de Heun, d'ordre 2, avec la méthode d'Euler explicite, d'ordre 1. Son tableau de Butcher étendu est le suivant :

0	0	0
1	1	0
(Heun)	$\frac{1}{2}$	$\frac{1}{2}$
(EE)	1	0

```
def EE( phi, tt, y0 ):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    uu[0] = y0
    for n in range(len(tt)-1):
        k0 = phi(tt[n],uu[n])
        k1 = phi(tt[n]+h,uu[n]+h*k0)
        uu[n+1] = uu[n]+h*(1*k0+0*k1)
    return uu

def HEUN( phi, tt, y0 ):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    uu[0] = y0
    for n in range(len(tt)-1):
        k0 = phi(tt[n],uu[n])
        k1 = phi(tt[n]+h,uu[n]+h*k0)
        uu[n+1] = uu[n]+h*(1/2*k0+1/2*k1)
    return uu

def RK12( phi, t0, tfinal, y0, tol, hinit, hmin, hmax ):
    tt = np.array([t0])
    uu = np.array([y0])
    uu[0] = y0
    h = hinit
    n = 0

    while tt[n]<tfinal:
        k0 = phi(tt[n],uu[n])
        k1 = phi(tt[n]+h,uu[n]+h*k0)

        # Un pas de EE
        # uu_om   = uu[n]+h*(1*k0+0*k1)

        # Un pas de HEUN
        uu_om_p1 =  uu[n]+h*(1/2*k0+1/2*k1)

        # choix
        err = (-1/2*k0+1/2*k1)*h

        if err > tol:
            # Solution refusée et adaptation du pas
            h = max( 0.8*h, hmin )
        else:
            # Solution acceptée
            tt = np.append(tt,tt[n]+h)
            uu = np.append(uu,uu_om_p1)
            n = n+1
            # Adaptation du pas
            if err < tol / 3:
                h = min( 1.25*h, hmax )
            # sinon h inchangé
```

```
        if tt[n]+h>tfinal: h = tfinal-tt[n]

    return tt,uu

# ================================
# MAIN
# ================================

t0, tfinal = 0, np.pi/2-0.1
y0 = 0
phi = lambda t,y : 1+y**2
sol_exacte = lambda t : np.tan(t)


# ================================
# Comparaison des méthodes
# ================================
plt.figure(figsize=(15,5))

# exacte
tt_ex = np.linspace(t0,tfinal,101)
yy = sol_exacte(tt_ex)
plt.plot(tt_ex, yy, 'r',label=r'$\tan(t)$')

# Pas fixe h
tt = np.linspace(t0,tfinal,11)
h_init = tt[1]-tt[0]

# EE
uu_EE = EE(phi, tt, y0)
plt.plot(tt,uu_EE, 'x', label='EE')

#  HEUN
uu_HEUN = HEUN(phi, tt, y0)
plt.plot(tt,uu_HEUN, 'd', label='HEUN')

# Pas adaptatif
toll = 1.e-1
h_min = h_init/10
h_max = 4*h_init
tt_adapt, uu_adapt = RK12(phi, t0, tfinal, y0, tol=toll, hinit=h_init, hmin=h_min, hmax=h_max)
plt.plot(tt_adapt,uu_adapt, 'o', label='Adaptatif_EE')

plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
plt.xlabel('$t$')
plt.ylabel('$y(t)$')
plt.grid();


# ================================
# Graphe de comparaison des erreurs
# ================================

err_EE    = np.abs(uu_EE    - sol_exacte(tt))
err_HEUN  = np.abs(uu_HEUN  - sol_exacte(tt))
err_adapt = np.abs(uu_adapt - sol_exacte(tt_adapt))
plt.figure(figsize=(15,5))
plt.plot(tt, err_EE, 'x-', label='Erreur EE')
plt.plot(tt, err_HEUN, 'd-', label='Erreur HEUN')
plt.plot(tt_adapt, err_adapt, 'o-', label='Erreur Adaptatif')
plt.xlabel("$t_n$")
plt.ylabel(r"$|u(t_n)-y(t_n)|$")
plt.title("Comparaison des erreurs en fonction du temps")
```

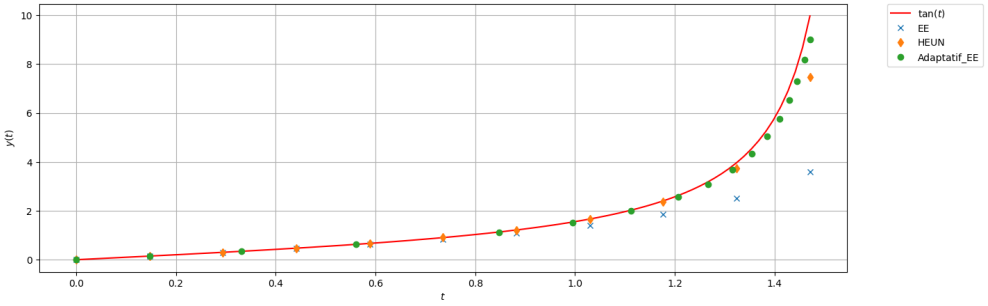
```
plt.grid(True, which="both")
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()

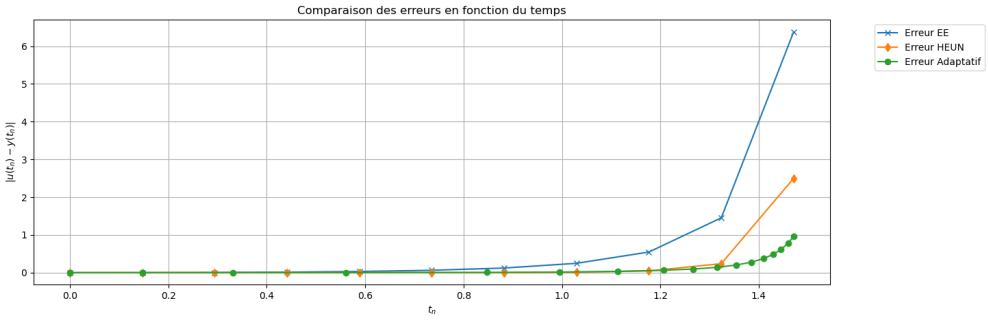

# ================================
# Comparaison des grilles
# ================================
plt.figure(figsize=(15,5))
plt.plot(tt,np.ones_like(tt)*2,'x',label='Grille fixe')
plt.plot(tt_adapt,np.ones_like(tt_adapt),'o',label='Grille adaptative')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
plt.ylim(0,3)
plt.xticks(tt)
plt.xlabel('$t$')
plt.yticks([1,2], ['Adaptative','Fixe'])
plt.grid()
plt.show()

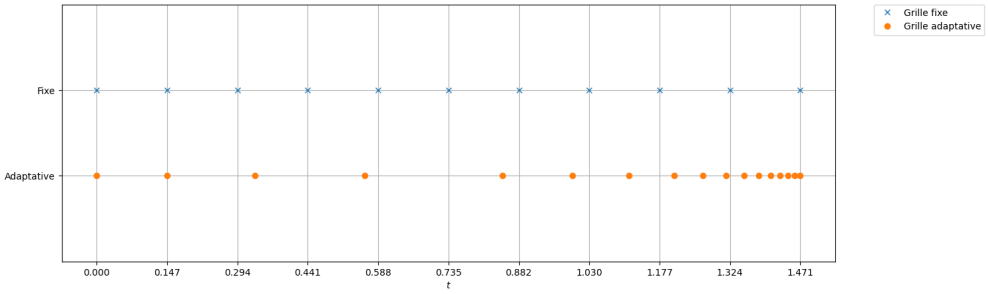

# ================================================================
# Évolution du pas adaptatif
# ================================================================
plt.figure(figsize=(8,5))
plt.plot(tt_adapt[1:], np.diff(tt_adapt), 'o-')
plt.xlabel("$t$")
plt.ylabel("$h$")
plt.title("Évolution du pas adaptatif")
plt.grid(True)
plt.show()

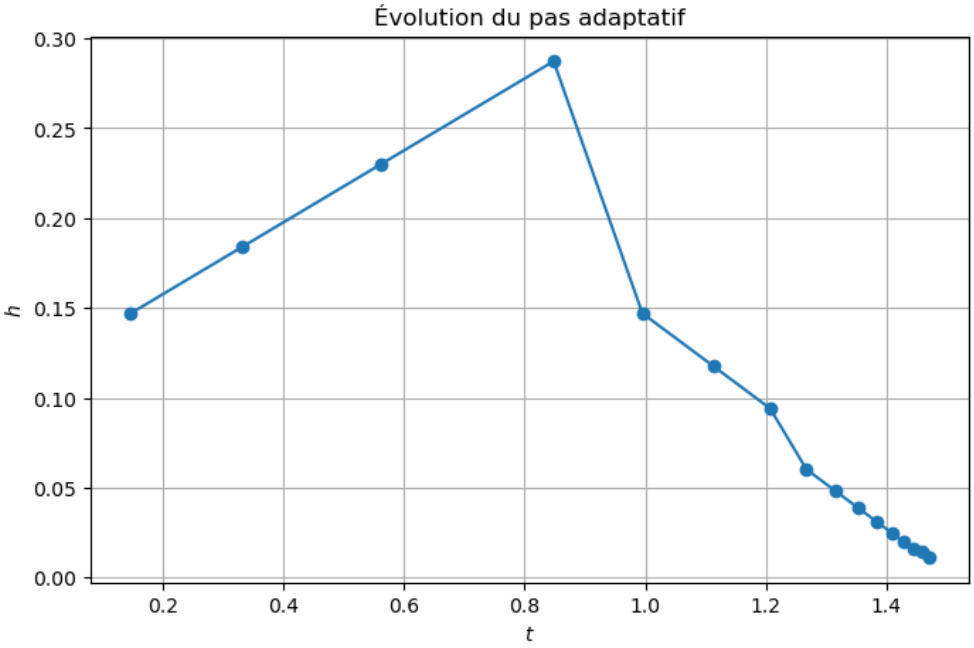
# ================================================================
# Comparaison des approximations en t=tfinal
# ================================================================

from tabulate import tabulate
tab = []
headers = ["Méthode", "t final", "|u(t final)-exacte(tfinal)|", "Points de grille"]
tab.append(["EE", tt[-1], abs(uu_EE[-1]-sol_exacte(tfinal)), len(tt)])
tab.append(["HEUN", tt[-1], abs(uu_HEUN[-1]-sol_exacte(tfinal)), len(tt)])
idx = np.argmax(tt_adapt>tfinal)-1
tab.append(["RK12", tt_adapt[idx], abs(uu_adapt[idx]-sol_exacte(tfinal)), len(tt_adapt)])
display(tabulate(tab, headers=headers, tablefmt="html", floatfmt=".2e"))
```









Méthode	t final	|u(t final)-exacte(tfinal)|	Points de grille
EE	1.47e+00	6.38e+00	11
HEUN	1.47e+00	2.50e+00	11
RK12	1.47e+00	9.59e-01	17

11.11.2.2. Méthode de Runge-Kutta adaptative Heun/Simpson (RK23)

Une méthode de Runge-Kutta adaptative simple combine la méthode de Heun, d'ordre 2, avec la méthode de Simpso, d'ordre 3. Son tableau de Butcher étendu est le suivant :

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{4} & \frac{1}{4} & 0 \\ \hline (\text{Heun}) & \frac{1}{2} & \frac{1}{2} & 0 \\ (\text{Simpson}) & \frac{1}{6} & \frac{1}{6} & \frac{4}{6} \end{array}$$

```
def coeff_K(phi, h, t, u):
    k0 = phi(t,u)
    k1 = phi(t+h,u+h*k0)
    k2 = phi(t+h/2,u+h/4*k0+h/4*k1)
    return k0, k1, k2

def HEUN( phi, tt, y0 ):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    uu[0] = y0
    for n in range(len(tt)-1):
        k0, k1, k2 = coeff_K(phi, h, tt[n], uu[n])
        uu[n+1] = uu[n]+h*(1/2*k0+1/2*k1)
    return uu

def SIMPSON( phi, tt, y0 ):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    uu[0] = y0
    for n in range(len(tt)-1):
        k0, k1, k2 = coeff_K(phi, h, tt[n], uu[n])
        uu[n+1] = uu[n]+h*(1/6*k0+4/6*k2+1/6*k1)
    return uu

def RK23( phi, t0, tfinal, y0, tol, hinit, hmin, hmax ):
    tt = np.array([t0])
    uu = np.array([y0])
    uu[0] = y0
    h = hinit
    n = 0
    while tt[n]<tfinal :
        k0, k1, k2 = coeff_K(phi, h, tt[n], uu[n])

        # Un pas de Heun
        # uu_om   = uu[n]+h*(1/2*k0+1/2*k1)

        # Un pas de Simpson
        uu_om_p1 =  uu[n]+h*(1/6*k0+1/6*k1+4/6*k2)

        # choix
        err = h * ( (1/6-1/2)*k0 + (1/6-1/2)*k1 + (4/6-0)*k2 )

        if err > tol:
            # Solution refusée et adaptation du pas
            h = max( 0.8*h, hmin )
        else:
            # Solution acceptée
            tt = np.append(tt,tt[n]+h)
            uu = np.append(uu,uu_om_p1)
            n = n+1
            # Adaptation du pas
            if err < tol / 3:
                h = min( 1.25*h, hmax )
            # sinon h inchangé

        if tt[n]+h>tfinal: h = tfinal-tt[n]

    return tt,uu

# ================================
# MAIN
# ================================
```

```
t0, tfinal = 0, np.pi/2-0.1
y0 = 0
phi = lambda t,y : 1+y**2
sol_exacte = lambda t : np.tan(t)


# ================================
# Comparaison des méthodes
# ================================
plt.figure(figsize=(15,5))

# exacte
tt_ex = np.linspace(t0,tfinal,101)
yy = sol_exacte(tt_ex)
plt.plot(tt_ex, yy, 'r',label=r'$\tan(t)$')

# Pas fixe h
tt = np.linspace(t0,tfinal,11)
h_init = tt[1]-tt[0]

#  HEUN
uu_HEUN = HEUN(phi, tt, y0)
plt.plot(tt,uu_HEUN, 'x', label='HEUN')

# SIMPSON
uu_SIMPSON = SIMPSON(phi, tt, y0)
plt.plot(tt,uu_SIMPSON, 'd', label='SIMPSON')

# Pas adaptatif
tt_adapt, uu_adapt = RK23(phi, t0, tfinal, y0, tol=1.e-1, hinit=h_init, hmin=1e-3, hmax=4*h_init)
plt.plot(tt_adapt,uu_adapt, 'o', label='Adaptatif')

plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
plt.grid();

# ================================
# Graphe de comparaison des erreurs
# ================================
err_HEUN   = np.abs(uu_HEUN   - sol_exacte(tt))
err_SIMPSON  = np.abs(uu_SIMPSON  - sol_exacte(tt))
err_adapt = np.abs(uu_adapt - sol_exacte(tt_adapt))
plt.figure(figsize=(15,5))
plt.plot(tt, err_HEUN, 'x-', label='Erreur HEUN')
plt.plot(tt, err_SIMPSON, 'd-', label='Erreur SIMPSON')
plt.plot(tt_adapt, err_adapt, 'o-', label='Erreur Adaptatif')
plt.xlabel("$t_n$")
plt.ylabel(r"$|u(t_n)-y(t_n)|$")
plt.title("Comparaison des erreurs en fonction du temps")
plt.grid(True, which="both")
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()

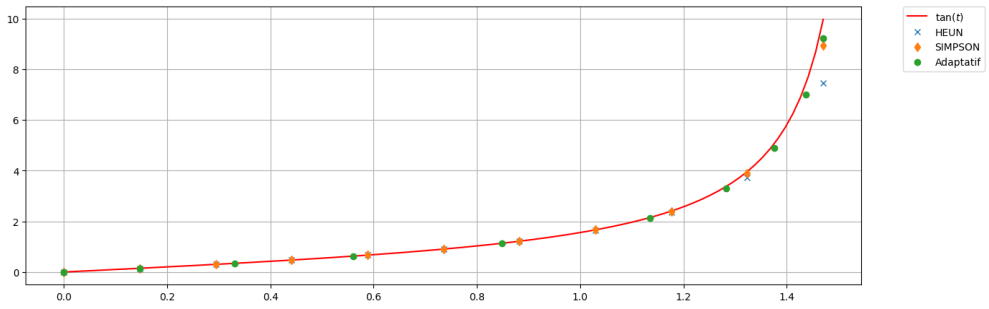

# ================================
# Comparaison des grilles
# ================================
plt.figure(figsize=(15,5))
plt.plot(tt,np.ones_like(tt)*2,'x',label='Grille fixe')
plt.plot(tt_adapt,np.ones_like(tt_adapt),'o',label='Grille adaptative')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
plt.ylim(0,3);
plt.xticks(tt)
plt.grid()
```

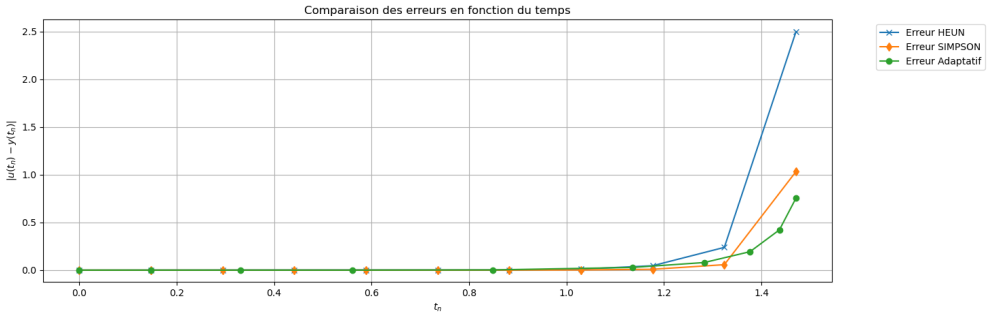
```
plt.show()

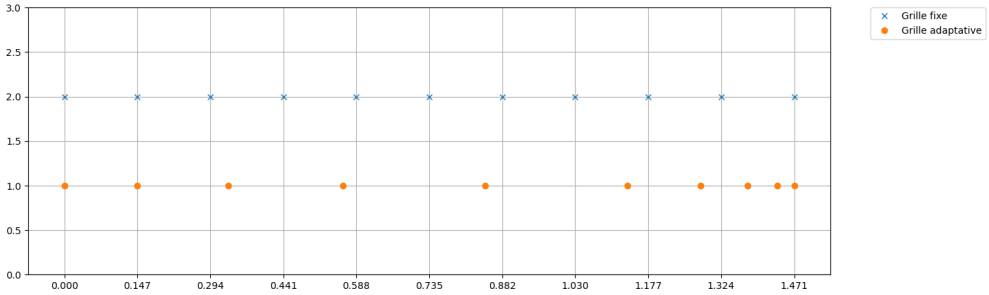

# ================================================================
# Évolution du pas adaptatif
# ================================================================
plt.figure(figsize=(8,5))
plt.plot(tt_adapt[1:], np.diff(tt_adapt), 'o-')
plt.xlabel("$t$")
plt.ylabel("$h$")
plt.title("Évolution du pas adaptatif")
plt.grid(True)
plt.show()

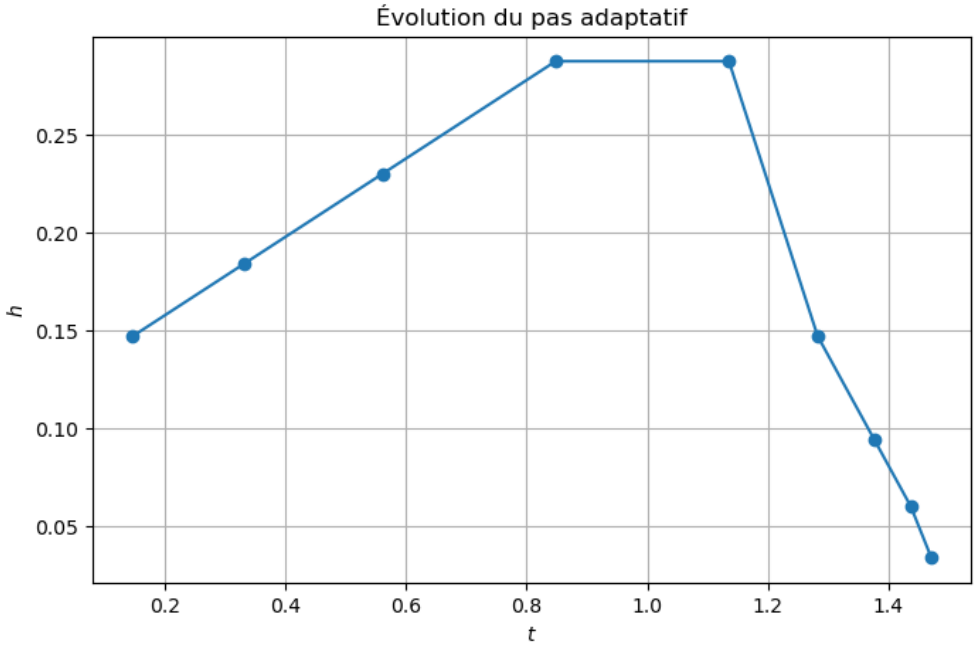
# ================================================================
# Comparaison des approximations en t=tfinal
# ================================================================

from tabulate import tabulate
tab = []
headers = ["Méthode", "t final", "|u(t final)-exacte(tfinal)|", "Points de grille"]
tab.append(["HEUN", tt[-1], abs(uu_HEUN[-1]-sol_exacte(tfinal)), len(tt)])
tab.append(["SIMPSON", tt[-1], abs(uu_SIMPSON[-1]-sol_exacte(tfinal)), len(tt)])
idx = np.argmax(tt_adapt>tfinal)-1
tab.append(["RK23", tt_adapt[idx], abs(uu_adapt[idx]-sol_exacte(tt_adapt[idx])), len(tt_adapt)])
display(tabulate(tab, headers=headers, tablefmt="html", floatfmt=".2e"))
```









Méthode	t final	|u(t final)-exacte(tfinal)|	Points de grille
HEUN	1.47e+00	2.50e+00	11
SIMPSON	1.47e+00	1.03e+00	11
RK23	1.47e+00	7.55e-01	10

11.11.2.3. Méthode de Runge-Kutta adaptative Bogacki/Shampine (ode23)

Une méthode de Runge-Kutta adaptative souvent utilisée combine la méthode de Bogacki avec celle de Shampine. Son tableau de Butcher étendu est le suivant :

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{3}{4} & 0 & \frac{3}{4} & 0 & 0 \\ 1 & \frac{2}{9} & \frac{1}{3} & \frac{4}{9} & 0 \\ \hline & \frac{2}{9} & \frac{1}{3} & \frac{4}{9} & 0 \\ & \frac{7}{24} & \frac{1}{4} & \frac{1}{3} & \frac{1}{8} \end{array}$$

```
def coeff_K(phi, h, t, u):
    k0 = phi(t,u)
    k1 = phi(t+h/2,u+h/2*k0)
    k2 = phi(t+h*3/4,u+h*3/4*k1)
    k3 = phi(t+h,u+h*2/9*k0+h/3*k1+h*4/9*k2)
    return k0, k1, k2, k3

def Bogacki( phi, tt, y0 ):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    uu[0] = y0
    for n in range(len(tt)-1):
        k0, k1, k2, k3 = coeff_K(phi, h, tt[n], uu[n])
        uu[n+1] = uu[n]+h*(2/9*k0+1/3*k1+4/9*k2)
    return uu

def Shampine( phi, tt, y0 ):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    uu[0] = y0
```

```
    for n in range(len(tt)-1):
        k0, k1, k2, k3 = coeff_K(phi, h, tt[n], uu[n])
        uu[n+1] = uu[n]+h*(7/24*k0+1/4*k1+1/3*k2+1/8*k3)
    return uu

def edo23( phi, t0, tfinal, y0, tol, hinit, hmin, hmax ):
    tt = np.array([t0])
    uu = np.array([y0])
    uu[0] = y0
    h = hinit
    n = 0
    while tt[n]<tfinal :
        k0, k1, k2, k3 = coeff_K(phi, h, tt[n], uu[n])

        # Un pas de Bogacki
        # uu_om   = uu[n]+h*(2/9*k0+1/3*k1+4/9*k2)

        # Un pas de Shamphine
        uu_om_p1 =  uu[n]+h*(7/24*k0+1/4*k1+1/3*k2+1/8*k3)

        # choix
        err = h * ( (7/24-2/9)*k0 + (1/4-1/3)*k1 + (1/3-4/9)*k2 + (1/8-0)*k3 )

        if err > tol:
            # Solution refusée et adaptation du pas
            h = max( 0.7*h, hmin )
        else:
            # Solution acceptée
            tt = np.append(tt,tt[n]+h)
            uu = np.append(uu,uu_om_p1)
            n = n+1
            # Adaptation du pas
            if err < tol/4:
                h = min( 1.2*h, hmax )
            # sinon h inchangé

        if tt[n]+h>tfinal: h = tfinal-tt[n]

    return tt,uu

# ================================
# MAIN
# ================================

t0, tfinal = 0, np.pi/2-0.1
y0 = 0
phi = lambda t,y : 1+y**2
sol_exacte = lambda t : np.tan(t)


# ================================
# Comparaison des méthodes
# ================================
plt.figure(figsize=(15,5))

# exacte
tt_ex = np.linspace(t0,tfinal,21)
yy = sol_exacte(tt_ex)
plt.plot(tt_ex, yy, 'r',label=r'$\tan(t)$')

# Pas fixe h
tt = np.linspace(t0,tfinal,21)
h_init = tt[1]-tt[0]
```

```
#  Bogacki
uu_Bogacki = Bogacki(phi, tt, y0)
plt.plot(tt,uu_Bogacki, 'x', label='Bogacki')

# Shampine
uu_Shampine = Shampine(phi, tt, y0)
plt.plot(tt,uu_Shampine, 'd', label='Shampine')

# Pas adaptatif
tt_adapt, uu_adapt = edo23(phi, t0, tfinal, y0, tol=1.e-3, hinit=h_init, hmin=1e-3, hmax=4*h_init)
plt.plot(tt_adapt,uu_adapt, 'o', label='Adaptatif')

plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
plt.grid();

# ================================
# Graphe de comparaison des erreurs
# ================================
err_Bogacki   = np.abs(uu_Bogacki   - sol_exacte(tt))
err_Shampine  = np.abs(uu_Shampine  - sol_exacte(tt))
err_adapt = np.abs(uu_adapt - sol_exacte(tt_adapt))
plt.figure(figsize=(15,5))
plt.plot(tt, err_Bogacki, 'x-', label='Erreur Bogacki')
plt.plot(tt, err_Shampine, 'd-', label='Erreur Shampine')
plt.plot(tt_adapt, err_adapt, 'o-', label='Erreur Adaptatif')
plt.xlabel("$t_n$")
plt.ylabel(r"$|u(t_n)-y(t_n)|$")
plt.title("Comparaison des erreurs en fonction du temps")
plt.grid(True, which="both")
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()

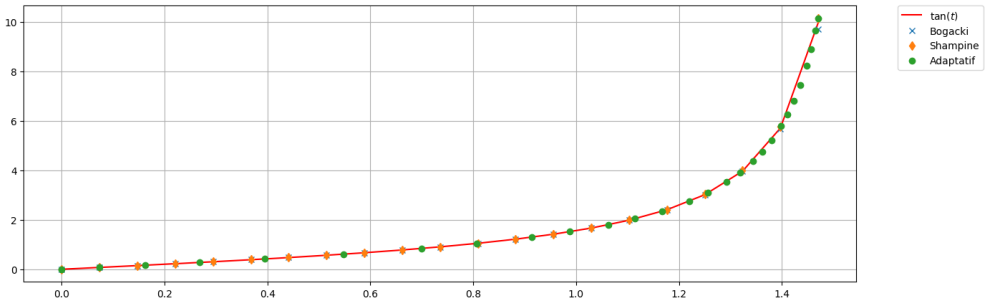
# ================================
# Comparaison des grilles
# ================================
plt.figure(figsize=(15,5))
plt.plot(tt,np.ones_like(tt)*2,'x',label='Grille fixe')
plt.plot(tt_adapt,np.ones_like(tt_adapt),'o',label='Grille adaptative')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
plt.ylim(0,3);
plt.xticks(tt)
plt.grid()
plt.show()

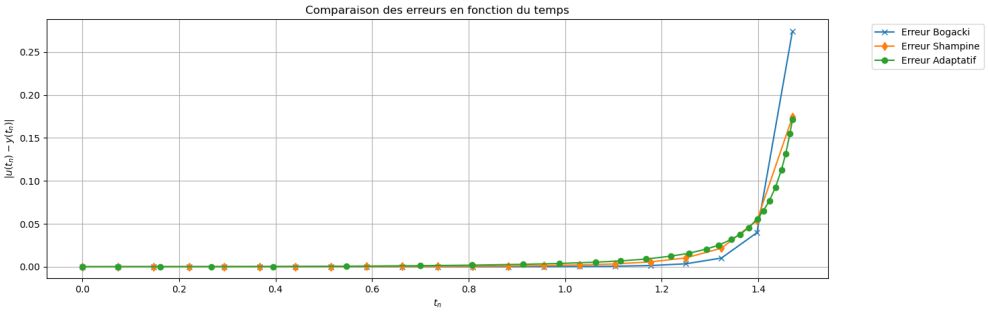

# ================================================================
# Évolution du pas adaptatif
# ================================================================
plt.figure(figsize=(8,5))
plt.plot(tt_adapt[1:], np.diff(tt_adapt), 'o-')
plt.xlabel("$t$")
plt.ylabel("$h$")
plt.title("Évolution du pas adaptatif")
plt.grid(True)
plt.show()

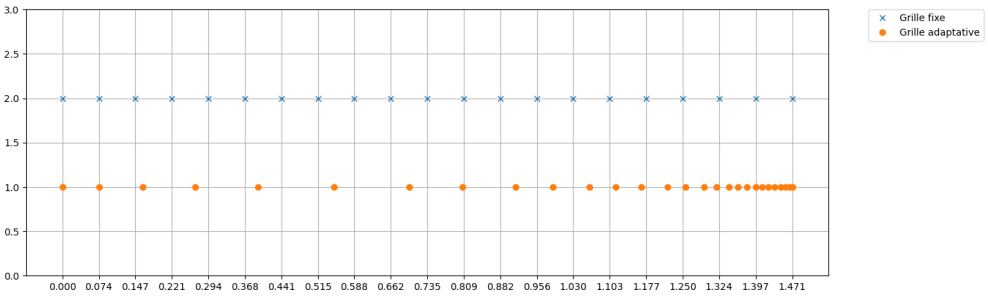
# ================================================================
# Comparaison des approximations en t=tfinal
# ================================================================

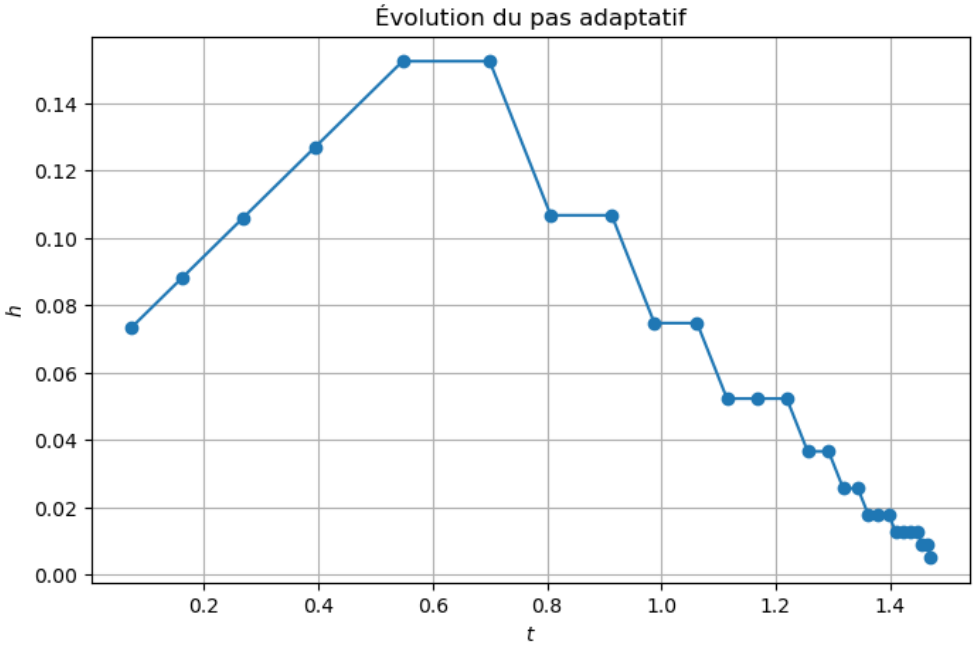
from tabulate import tabulate
tab = []
```

```
headers = ["Méthode", "t final", "|u(t final)-exacte(tfinal)|", "Points de grille"]
tab.append(["Bogacki", tt[-1], abs(uu_Bogacki[-1]-sol_exacte(tfinal)), len(tt)])
tab.append(["Shampine", tt[-1], abs(uu_Shampine[-1]-sol_exacte(tfinal)), len(tt)])
idx = np.argmax(tt_adapt>tfinal)-1
tab.append(["edo23", tt_adapt[idx], abs(uu_adapt[idx]-sol_exacte(tt_adapt[idx])), len(tt_adapt)])
display(tabulate(tab, headers=headers, tablefmt="html", floatfmt=".2e"))
```









Méthode	t final	|u(t final)-exacte(tfinal)|	Points de grille
Bogacki	1.47e+00	2.74e-01	21
Shampine	1.47e+00	1.74e-01	21
edo23	1.47e+00	1.72e-01	28

11.11.2.4. Runge-Kutta-Fehlberg (RKF45) à pas adaptatif

Voici la matrice de Butcher associée à la méthode de Runge-Kutta-Fehlberg (RKF45). Ce tableau représente les coefficients des deux méthodes Runge-Kutta utilisées dans la méthode RK45 pour calculer les coefficients k_0 à k_5, ainsi que les coefficients b_0 à b_5 et $\hat{b}_0$ à $\hat{b}_5$ utilisés pour combiner les K_i afin d'obtenir une approximation à l'étape suivante.

$$
\begin{array}{c|cccccc}
0 & & & & & & \\
\frac{1}{4} & \frac{1}{4} & & & & & \\
\frac{3}{8} & \frac{3}{32} & \frac{9}{32} & & & & \\
\frac{12}{13} & \frac{1932}{2197} & -\frac{7200}{2197} & \frac{7296}{2197} & & & \\
1 & \frac{439}{216} & -\frac{8}{27} & \frac{3680}{513} & -\frac{845}{4104} & & \\
\frac{1}{2} & -\frac{8}{27} & \frac{2}{9} & -\frac{3544}{2565} & \frac{1859}{4104} & -\frac{11}{40} & \\
\hline
 & \frac{16}{135} & 0 & \frac{6656}{12825} & \frac{28561}{56430} & -\frac{9}{50} & \frac{2}{55} \\
 & \frac{25}{216} & 0 & \frac{1408}{2565} & \frac{2197}{4104} & -\frac{1}{5} & 0
\end{array}
$$

Voyons un exemple d'implémentation de la méthode de Runge-Kutta-Fehlberg (RKF45) à pas adaptatif.

```
def coeff_K(phi, h, t, u):
    k = np.zeros(6)
    k[0] = phi(t,u)
    k[1] = phi(t+h/4,u+h/4*k[0])
    k[2] = phi(t+3*h/8,u+3*h/32*k[0]+9*h/32*k[1])
    k[3] = phi(t+12*h/13,u+1932*h/2197*k[0]-7200*h/2197*k[1]+7296*h/2197*k[2])
    k[4] = phi(t+h,u+439*h/216*k[0]-8*h*k[1]+3680*h/513*k[2]-845*h/4104*k[3])
    k[5] = phi(t+h/2,u-8*h/27*k[0]+2*h*k[1]-3544*h/2565*k[2]+1859*h/4104*k[3]-11*h/40*k[4])
    return k

def RK4( phi, tt, y0 ):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    uu[0] = y0
    for n in range(len(tt)-1):
        kk = coeff_K(phi, h, tt[n], uu[n])
        uu[n+1] = uu[n] + h*( 25/216*kk[0] + 1408/2565*kk[2] + 2197/4104*kk[3] - 1/5*kk[4] )
    return uu

def RK5( phi, tt, y0 ):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    uu[0] = y0
    for n in range(len(tt)-1):
        kk = coeff_K(phi, h, tt[n], uu[n])
        uu[n+1] = uu[n] + h* ( 16/135*kk[0] + 6656/12825*kk[2] + 28561/56430*kk[3] -9/50*kk[4] +
 ↪  2/55*kk[5] )
    return uu

def RK45( phi, t0, tfinal, y0, tol, hinit, hmin, hmax ):
    tt = np.array([t0])
    uu = np.array([y0])
    uu[0] = y0
    h = hinit
    n = 0
    while tt[n]<tfinal:
        kk = coeff_K(phi, h, tt[n], uu[n])
```

```
        # Un pas de RK5
        uu_om_p1 = uu[n] + h* ( 16/135*kk[0] + 6656/12825*kk[2] + 28561/56430*kk[3] -9/50*kk[4] +
↪  2/55*kk[5] )


        # choix
        err = h*( (25/216-16/135)*kk[0] +
                    (1408/2565-6656/12825)*kk[2] +
                    (2197/4104-28561/56430)*kk[3] +
                    (-1/5+9/50)*kk[4] +(0-2/55)*kk[5]  )

        if err > tol:
            # Solution refusée et adaptation du pas
            h = max( 0.8*h, hmin )
        else:
            # Solution acceptée
            tt = np.append(tt,tt[n]+h)
            uu = np.append(uu,uu_om_p1)
            n = n+1
            # Adaptation du pas
            if err < tol / 3:
                h = min( 1.25*h, hmax )
            # sinon h inchangé

        if tt[n]+h>tfinal: h = tfinal-tt[n]

    return tt,uu

# ================================
# MAIN
# ================================

t0, tfinal = 0, 4
y0 = 1
phi = lambda t,y : y*(2-y)

import sympy as sp
t = sp.symbols('t')
y = sp.Function('y')
sol_par = sp.dsolve( sp.Eq(y(t).diff(t), phi(t,y(t)) ) , ics={y(t0): y0} )
display(sol_par)
sol_exacte = sp.lambdify(t,sol_par.rhs,'numpy')


# ================================
# Comparaison des méthodes
# ================================
plt.figure(figsize=(15,5))

# exacte
tt_ex = np.linspace(t0,tfinal,101)
yy = sol_exacte(tt_ex)
plt.plot(tt_ex, yy, 'r',label=r'Exacte')


# Pas fixe h
tt = np.linspace(t0,tfinal,6)
h_init = tt[1]-tt[0]

#  RK4
uu_RK4 = RK4(phi, tt, y0)
plt.plot(tt,uu_RK4, 'x', label='RK4')
```

```
# RK5
uu_RK5 = RK5(phi, tt, y0)
plt.plot(tt,uu_RK5, 'd', label='RK5')

# Pas adaptatif
tt_adapt, uu_adapt = RK45(phi, t0, tfinal, y0, tol=1.e-5, hinit=h_init, hmin=5e-3, hmax=4*h_init)
plt.plot(tt_adapt,uu_adapt, 'o', label='Adaptatif')

plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
plt.grid();

# ================================
# Graphe de comparaison des erreurs
# ================================
err_RK4   = np.abs(uu_RK4   - sol_exacte(tt))
err_RK5   = np.abs(uu_RK5   - sol_exacte(tt))
err_adapt = np.abs(uu_adapt - sol_exacte(tt_adapt))
plt.figure(figsize=(15,5))
plt.plot(tt, err_RK4, 'x-', label='Erreur RK4')
plt.plot(tt, err_RK5, 'd-', label='Erreur RK5')
plt.plot(tt_adapt, err_adapt, 'o-', label='Erreur Adaptatif')
plt.xlabel("$t_n$")
plt.ylabel(r"$|u(t_n)-y(t_n)|$")
plt.title("Comparaison des erreurs en fonction du temps")
plt.grid(True, which="both")
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()


# ================================
# Comparaison des grilles
# ================================
plt.figure(figsize=(15,5))
plt.plot(tt,np.ones_like(tt)*2,'x',label='Grille fixe')
plt.plot(tt_adapt,np.ones_like(tt_adapt),'o',label='Grille adaptative')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
plt.ylim(0,3);
plt.xticks(tt)
plt.grid()
plt.show()

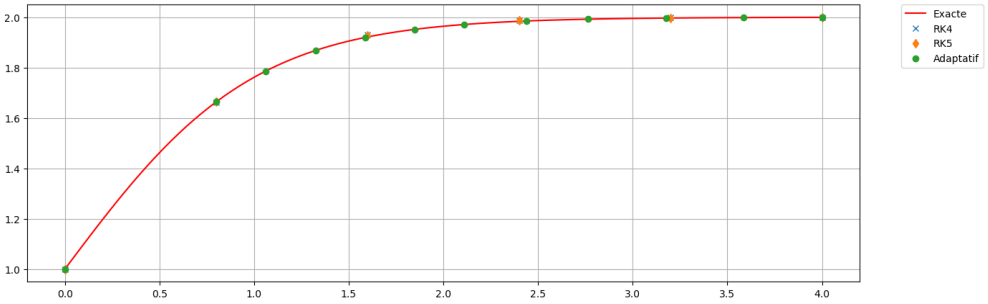

# ================================================================
# Évolution du pas adaptatif
# ================================================================
plt.figure(figsize=(8,5))
plt.plot(tt_adapt[1:], np.diff(tt_adapt), 'o-')
plt.xlabel("$t$")
plt.ylabel("$h$")
plt.title("Évolution du pas adaptatif")
plt.grid(True)
plt.show()

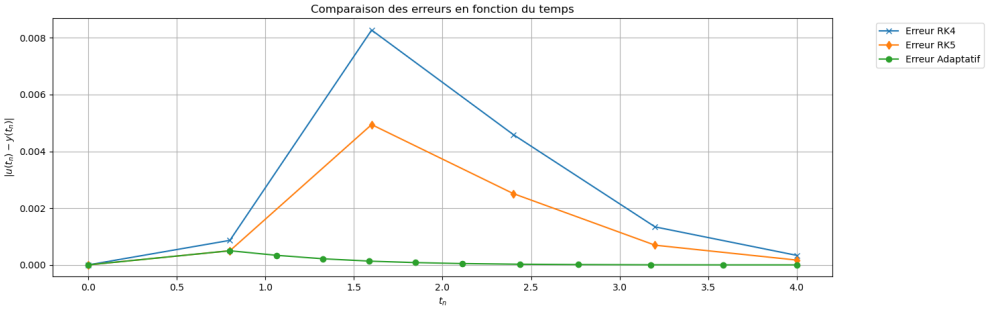
# ================================================================
# Comparaison des approximations en t=tfinal
# ================================================================

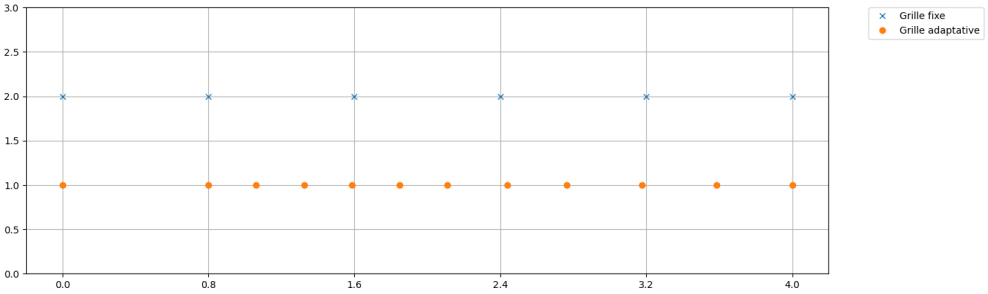
from tabulate import tabulate
tab = []
headers = ["Méthode", "t final", "|u(t final)-exacte(tfinal)|", "Points de grille"]
tab.append(["RK4", tt[-1], abs(uu_RK4[-1]-sol_exacte(tfinal)), len(tt)])
tab.append(["RK5", tt[-1], abs(uu_RK5[-1]-sol_exacte(tfinal)), len(tt)])
idx = np.argmax(tt_adapt>tfinal)-1
```

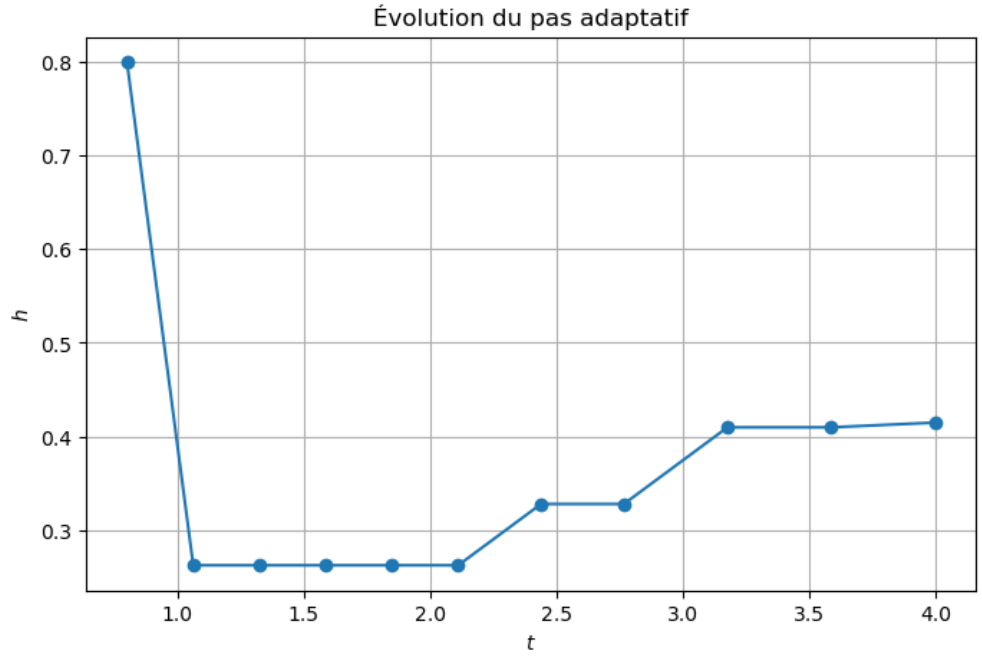
```
tab.append(["RK45", tt_adapt[idx], abs(uu_adapt[idx]-sol_exacte(tt_adapt[idx])), len(tt_adapt)])
display(tabulate(tab, headers=headers, tablefmt="html", floatfmt=".2e"))
```

$$y(t) = \frac{2}{1 + e^{-2t}}$$









Méthode	t final	|u(t final)-exacte(tfinal)|	Points de grille
RK4	4.00e+00	3.36e-04	6
RK5	4.00e+00	1.70e-04	6
RK45	4.00e+00	1.31e-07	12

11.11.2.5. Dormand-Prince (DP45) à pas adaptatif

Voici la matrice de Butcher associée à la méthode de Dormand-Prince (DP45). Ce tableau représente les coefficients des deux méthodes Runge-Kutta utilisées dans la méthode RK45 pour calculer les coefficients k_0 à k_5, ainsi que les coefficients b_0 à b_5 et $\widehat{b}_0$ à $\widehat{b}_5$ utilisés pour combiner les K_i afin d'obtenir une approximation à l'étape suivante.

$$
\begin{array}{c|ccccccc}
0 & & & & & & & \\
\frac{1}{5} & \frac{1}{5} & & & & & & \\
\frac{3}{10} & \frac{3}{40} & \frac{9}{40} & & & & & \\
\frac{4}{5} & \frac{44}{45} & -\frac{56}{15} & \frac{32}{9} & & & & \\
\frac{8}{9} & \frac{19372}{6561} & -\frac{25360}{2187} & \frac{64448}{6561} & -\frac{212}{729} & & & \\
1 & \frac{9017}{3168} & -\frac{355}{33} & \frac{46732}{5247} & \frac{49}{176} & -\frac{5103}{18656} & & \\
1 & \frac{35}{384} & 0 & \frac{500}{1113} & \frac{125}{192} & -\frac{2187}{6784} & \frac{11}{84} & \\
\hline
 & \frac{35}{384} & 0 & \frac{500}{1113} & \frac{125}{192} & -\frac{2187}{6784} & \frac{11}{84} & 0 \\
 & \frac{5179}{57600} & 0 & \frac{7571}{16695} & \frac{393}{640} & -\frac{92097}{339200} & \frac{187}{2100} & \frac{1}{40}
\end{array}
$$

Voyons un exemple d'implémentation de la méthode de Runge-Kutta-Fehlberg (RKF45) à pas adaptatif.

```
def coeff_K(phi, h, t, u):
    k = np.zeros(7)
    k[0] = phi(t,u)
    k[1] = phi(t+h/5,u+h/5*k[0])
    k[2] = phi(t+3*h/10,u+3*h/40*k[0]+9*h/40*k[1])
    k[3] = phi(t+4*h/5,u+44*h/45*k[0]-56*h/15*k[1]+32*h/9*k[2])
    k[4] = phi(t+8*h/9,u+19372*h/6561*k[0]-25360*h/2187*k[1]+64448*h/6561*k[2]-212*h/729*k[3])
    k[5] = phi(t+h,u+9017*h/3168*k[0]-355*h/33*k[1]+46732*h/5247*k[2]+49*h/176*k[3]-5103*h/18656*k[4])
    k[6] = phi(t+h,u+35*h/384*k[0]+0*k[1]+500*h/1113*k[2]+125*h/192*k[3]-2187*h/6784*k[4]+11*h/84*k[5])
    return k

def Dor( phi, tt, y0 ):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    uu[0] = y0
    for n in range(len(tt)-1):
        kk = coeff_K(phi, h, tt[n], uu[n])
        uu[n+1] = uu[n] + h*( 35/384*kk[0] + 0*kk[1] + 500/1113*kk[2] + 125/192*kk[3] - 2187/6784*kk[4] +
 ↪  11/84*kk[5] )
    return uu

def Pri( phi, tt, y0 ):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    uu[0] = y0
    for n in range(len(tt)-1):
        kk = coeff_K(phi, h, tt[n], uu[n])
        uu[n+1] = uu[n] + h* ( 5179/57600*kk[0] + 0*kk[1] + 7571/16695*kk[2] + 393/640*kk[3] -
 ↪  92097/339200*kk[4] + 187/2100*kk[5] + 1/40*kk[6] )
    return uu

def DP45( phi, t0, tfinal, y0, tol, hinit, hmin, hmax ):
    tt = np.array([t0])
    uu = np.array([y0])
    uu[0] = y0
    h = hinit
```

```
    n = 0
    while tt[n]<tfinal:
        kk = coeff_K(phi, h, tt[n], uu[n])

        # Un pas de Pri
        # uu_om_p1 = uu[n] + h* ( 5179/57600*k0 + 0*k1 + 7571/16695*k2 + 393/640*k3 - 92097/339200*k4 +
        ↪  187/2100*k5 + 1/40*k6 )
        uu_om_p1 =  uu[n] + h*( 35/384*kk[0] + 0*kk[1] + 500/1113*kk[2] + 125/192*kk[3] - 2187/6784*kk[4]
↪  + 11/84*kk[5] )

        # choix
        err = h*( (35/384-5179/57600)*kk[0] +
                  (0-0)*kk[1] +
                  (500/1113-7571/16695)*kk[2] +
                  (125/192-393/640)*kk[3] +
                  (-2187/6784-92097/339200)*kk[4] +
                  (11/84-187/2100)*kk[5] +
                  (0-1/40)*kk[6] )

        if err > tol:
            # Solution refusée et adaptation du pas
            h = max( 0.8*h, hmin )
        else:
            # Solution acceptée
            tt = np.append(tt,tt[n]+h)
            uu = np.append(uu,uu_om_p1)
            n = n+1
            # Adaptation du pas
            if err < tol / 3:
                h = min( 1.25*h, hmax )
            # sinon h inchangé

        if tt[n]+h>tfinal: h = tfinal-tt[n]

    return tt,uu


# ================================
# MAIN
# ================================

t0, tfinal = 0, 8
y0 = 1
phi = lambda t,y : y*(2-y)

import sympy as sp
t = sp.symbols('t')
y = sp.Function('y')
sol_par = sp.dsolve( sp.Eq(y(t).diff(t), phi(t,y(t)) ) , ics={y(t0): y0} )
display(sol_par)
sol_exacte = sp.lambdify(t,sol_par.rhs,'numpy')

# ================================
# Comparaison des méthodes
# ================================
plt.figure(figsize=(15,5))

# exacte
tt_ex = np.linspace(t0,tfinal,101)
yy = sol_exacte(tt_ex)
plt.plot(tt_ex, yy, 'r',label=r'Exacte')
```

```
# Pas fixe h
tt = np.linspace(t0,tfinal,11)
h_init = tt[1]-tt[0]

# Dorman
uu_Dor = Dor(phi, tt, y0)
plt.plot(tt,uu_Dor, 'v', label='Dorman')

# Prince
uu_Pri = Pri(phi, tt, y0)
plt.plot(tt,uu_Pri, 'd', label='Prince')

# DP45 à pas adaptatif
tt_adapt, uu_adapt = DP45(phi, t0, tfinal, y0, tol=1.e-3, hinit=h_init, hmin=h_init/10, hmax=3*h_init)
plt.plot(tt_adapt,uu_adapt, 'o', label='Adaptatif_EE')

plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
plt.grid();


# ================================
# Graphe de comparaison des erreurs
# ================================
err_Dor   = np.abs(uu_Dor   - sol_exacte(tt))
err_Pri   = np.abs(uu_Pri   - sol_exacte(tt))
err_adapt = np.abs(uu_adapt - sol_exacte(tt_adapt))
plt.figure(figsize=(15,5))
plt.plot(tt, err_Dor, 'v-', label='Erreur Dorman')
plt.plot(tt, err_Pri, 'd-', label='Erreur Prince')
plt.plot(tt_adapt, err_adapt, 'o-', label='Erreur Adaptatif')
plt.xlabel("$t_n$")
plt.ylabel(r"$|u(t_n)-y(t_n)|$")
plt.title("Comparaison des erreurs en fonction du temps")
plt.grid(True, which="both")
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()


# ================================
# Comparaison des grilles
# ================================
plt.figure(figsize=(15,5))
plt.plot(tt,np.ones_like(tt)*2,'v',label='Grille grossière')
plt.plot(tt_adapt,np.ones_like(tt_adapt),'o',label='Grille adaptative')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
plt.ylim(0,3);
plt.xticks(tt)
plt.yticks([1,2], ['Adaptative','Fixe'])
plt.grid()
plt.show()



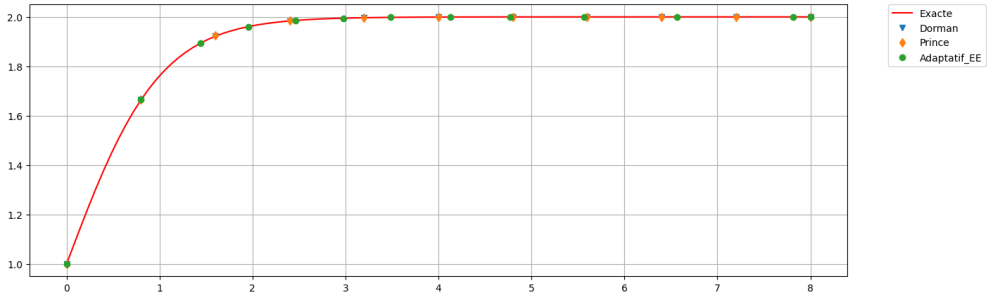
# ================================================================
# Évolution du pas adaptatif
# ================================================================
plt.figure(figsize=(8,5))
plt.plot(tt_adapt[1:], np.diff(tt_adapt), 'o-')
plt.xlabel("$t$")
plt.ylabel("$h$")
plt.title("Évolution du pas adaptatif")
plt.grid(True)
plt.show()
```

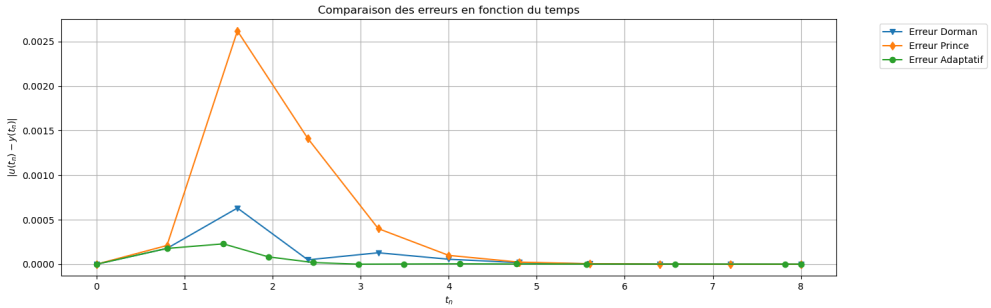
```
# ================================================================
# Comparaison des approximations en t=tfinal
# ================================================================

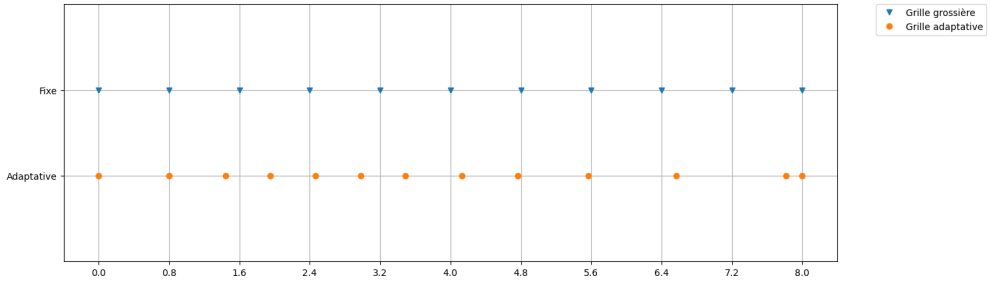
from tabulate import tabulate
tab = []
headers = ["Méthode", "t final", "|u(t final)-exacte(tfinal)|", "Points de grille"]
tab.append(["Dorman", tt[-1], abs(uu_Dor[-1]-sol_exacte(tfinal)), len(tt)])
tab.append(["Prince", tt[-1], abs(uu_Pri[-1]-sol_exacte(tfinal)), len(tt)])

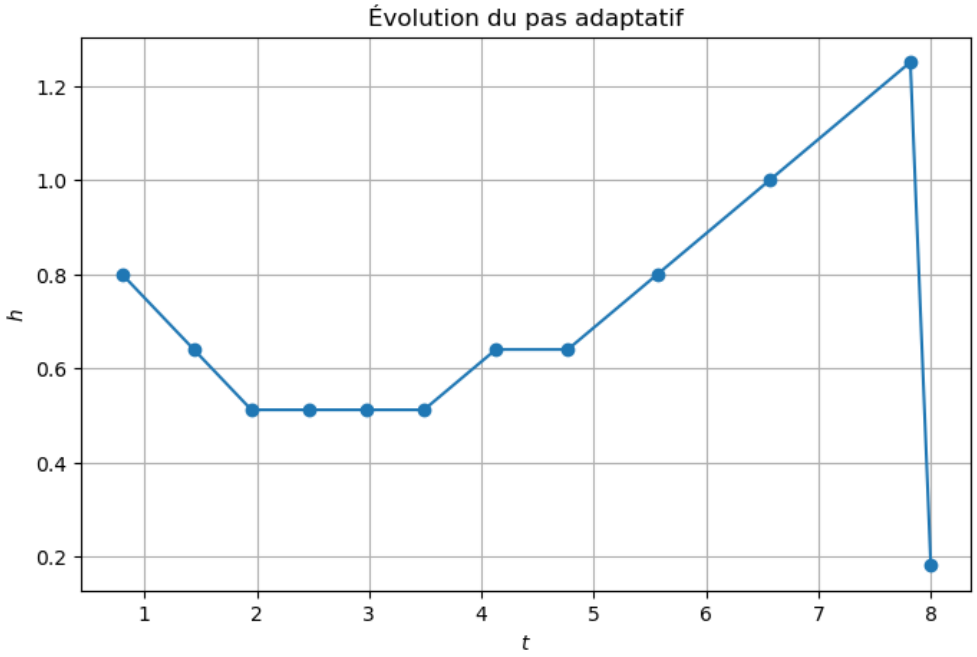
idx = np.argmax(tt_adapt>tfinal)-1
tab.append(["DP45", tt_adapt[idx], abs(uu_adapt[idx]-sol_exacte(tt_adapt[idx])), len(tt_adapt)])
display(tabulate(tab, headers=headers, tablefmt="html", floatfmt=".2e"))
```

$$y(t) = \frac{2}{1 + e^{-2t}}$$









Méthode	t final	|u(t final)-exacte(tfinal)|	Points de grille
Dorman	8.00e+00	7.86e-08	11
Prince	8.00e+00	6.01e-08	11
DP45	8.00e+00	6.76e-07	13

11.11.3. Méthode d'Euler explicite adaptative

Lorsqu'on utilise une des méthodes présentées jusqu'ici, une question ouverte est le choix du pas de discrétisation h pour obtenir une solution précise sans effectuer un nombre excessif de calculs. Une idée naturelle consiste à ajuster dynamiquement le pas de discrétisation en fonction de l'erreur commise à chaque itération. Si l'erreur est inférieure à une tolérance donnée, on peut augmenter le pas de discrétisation pour accélérer le calcul. Inversement, si l'erreur est trop grande, on peut diminuer le pas de discrétisation pour améliorer la précision. Dans ce cas, la discrétisation h_n n'est plus uniforme mais dépend de l'itération n : $t_{n+1} = t_n + h_n$.
Pour estimer l'erreur de discrétisation locale, nous comparons deux approximations : la première calcule u_{n+1} avec un pas h, la seconde calcule u_{n+1} avec un pas $h/2$. L'erreur de discrétisation locale est alors estimée par la différence entre ces deux approximations.
Cette approche est particulièrement adaptée au cas des équations raides, où la solution varie rapidement sur certaines parties de l'intervalle de temps et lentement sur d'autres. Un pas de discrétisation uniforme peut être trop grand pour les parties lentes et trop petit pour les parties rapides. En ajustant le pas de discrétisation en fonction de l'erreur locale, on peut obtenir une solution précise sans effectuer un nombre excessif de calculs.
Étape I : Implémentation de la fonction `euler_adaptatif`
La méthode d'Euler explicite classique utilise un pas de temps fixe h : $u_{n+1} = u_n + h\varphi(t_n, u_n)$. Cependant, pour améliorer la précision tout en minimisant le nombre de calculs, on peut utiliser une version **adaptative** de la méthode d'Euler. Dans cette version, le pas de temps h_n est ajusté dynamiquement en fonction d'une estimation de l'erreur locale.
Créez une fonction `euler_adaptatif(phi, t0, y0, h0, tol, tfin)` qui prend en entrée :

- `phi` : fonction définissant l'équation différentielle $y'(t) = \varphi(t, y(t))$
- `t0` : temps initial
- `y0` : condition initiale $y(t_0)$
- `h0` : pas de temps initial
- `tol` : tolérance pour l'estimation de l'erreur locale
- `tfin` : temps maximal jusqu'où intégrer l'équation.

La fonction doit renvoyer :

- un tableau des temps $[t_0, t_1, \dots, t_N]$.
- un tableau des valeurs $[u_0, u_1, \dots, u_N]$.

L'algorithme d'adaptation du pas de temps suit les étapes suivantes :
Tant que $t_n \leq t_{\text{fin}}$:

1. **Calcul des solutions avec deux pas différents :**
   - Utiliser la méthode d'Euler explicite avec un pas h : $v_{n+1} = u_n + h\varphi(t_n, u_n)$.
   - Utiliser la méthode d'Euler explicite deux fois avec un pas $h/2$:
     - Calcul intermédiaire : $\tilde{u} = u_n + \frac{h}{2}\varphi(t_n, u_n)$
     - Calcul final : $w_{n+1} = \tilde{u} + \frac{h}{2}\varphi(t_n + h/2, \tilde{u})$.
2. **Ajustement du pas de temps :**
   - Définir l'erreur locale relative : erreur $= \frac{|w_{n+1} - v_{n+1}|}{\max\limits_{k \leq n} |u_k|}$
   - Si l'erreur est **inférieure** à la tolérance tol, ajouter $t_{n+1} = t_n + h$ et u_{n+1} aux tableaux de résultats, puis doubler h.
   - Si l'erreur est **supérieure** à la tolérance tol, diviser h par 2 (si h est inférieur à $h_{\min}$, continuer avec $h_{\min}$) et refaire le calcul des solutions avec ce nouveau pas de temps.

Étape 2 : Test de la méthode
Considérons le problème de Cauchy

$$\begin{cases} y'(t) = -y(t), \\ y(0) = 1. \end{cases} \qquad t \in [0, 20].$$

Nous voulons comparer les solutions obtenues avec la méthode d'Euler explicite et avec la méthode d'Euler adaptative. Nous comparerons ensuite le nombre de points utilisés avec les deux méthodes.

- Utiliser Euler explicite avec un pas de temps fixe $h = 1$ (valeur maximale qui garantit la monotonie de la solution).
- Utiliser Euler adaptatif avec $h_0 = 1$ et une tolérance tol $= 10^{-3}$.
- Comparer les solutions obtenues en traçant les 2 courbes (et la solution exacte).
- Comparer le nombre de pas effectués par chaque méthode et leurs erreurs respectives.

Correction

On commence par calculer une solution approchée avec la méthode d'Euler explicite. On peut ensuite calculer une solution approchée avec la méthode d'Euler adaptatif. On compare ensuite le nombre de points calculés avec les deux méthodes.

```
def EE(phi,tt,y0):
    N = len(tt)-1
    h = tt[1]-tt[0]
    uu = np.zeros(N+1)
    uu[0] = y0
    for n in range(N):
        uu[n+1] = uu[n] + h*phi(tt[n],uu[n])
    return uu

def EA(phi,t0,y0,hinit,tol,tfin):
    hmin = 1.e-12
    tt = np.array([t0])
    uu = np.array([y0])
    h = hinit
    while tt[-1] < tfin:
        k1 = phi(tt[-1],uu[-1])
        v = uu[-1] + h*k1
        u_demi = uu[-1] + h/2*k1
        w = u_demi + h/2*phi(tt[-1]+h/2,u_demi)
        erreur = np.abs(w - v)/np.linalg.norm(uu,ord=np.inf)
        if erreur < tol or h <= hmin:
            tt = np.append(tt,tt[-1]+h)
            uu = np.append(uu,w)
            h = 2*h
        else :
            h = max(h/2,hmin)
```

```
    return tt,uu
```

```
# TEST
# ========================

phi = lambda t, y : -y
t0, tfin = 0, 40
y0 = 1

# Pour la A-stabilité, il faut que h<2, pour la monotonie h<=1
# h = 1 pour N = (tfin-t0)/h = tfin-t0

N = int(tfin-t0)
tt = np.linspace(t0,tfin,N+1)

plt.figure(figsize=(8,5))

# Solution exacte
sol_exacte = lambda t : y0 * np.exp(-t)
plt.plot(tt,sol_exacte(tt),'-',label='exacte', color='green', lw=2)

# EE
uu_EE = EE(phi,tt,y0)
plt.plot(tt,uu_EE,'o-',label='EE', color='blue')

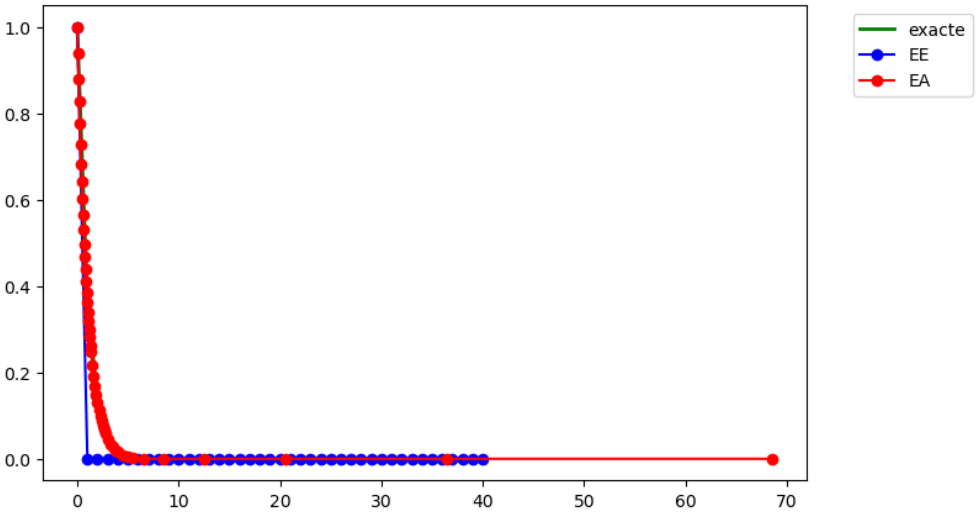
# EA
hinit = tt[1]-tt[0]
tol = 1.e-3
tt_EA, uu_EA = EA(phi,t0,y0,hinit,tol,tfin)
plt.plot(tt_EA,uu_EA,'o-',label='EA', color='red')

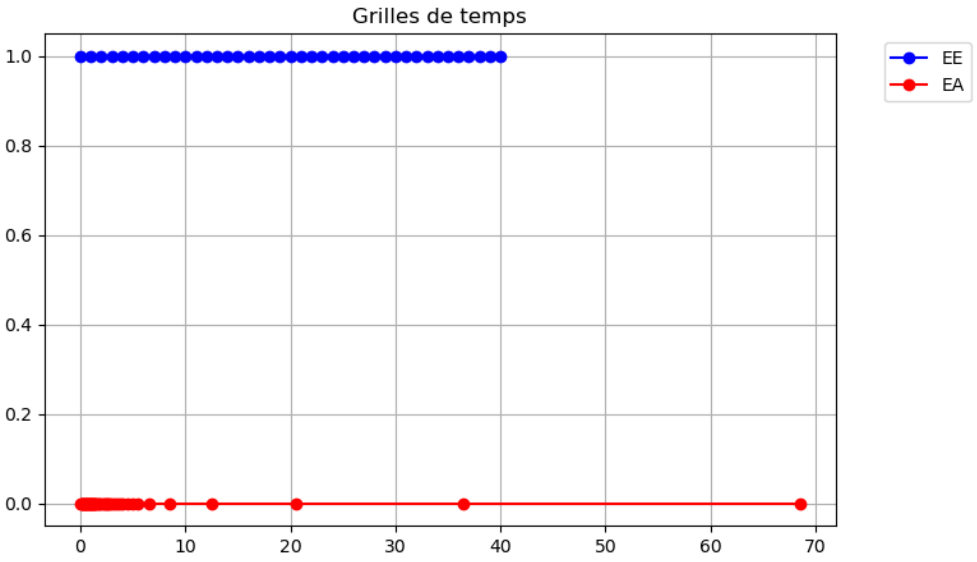
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left');

print(f"Erreur EE : {np.linalg.norm(uu_EE-sol_exacte(tt))}")
print(f"Erreur EA : {np.linalg.norm(uu_EA-sol_exacte(tt_EA))}")
print(f"Nombre de pas EE : {len(tt)-1}")
print(f"Nombre de pas EA : {len(tt_EA)-1}")

# Affichons juste les deux grilles de temps sur une droite horizontale
plt.figure(figsize=(8,5))
plt.plot(tt,np.ones_like(tt),'o-',label='EE',color='blue')
plt.plot(tt_EA,1-np.ones_like(tt_EA),'o-',label='EA',color='red')
plt.grid()
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.title('Grilles de temps');
```

```
Erreur EE : 0.3956231069460752
Erreur EA : 0.03012966353642783
Nombre de pas EE : 40
Nombre de pas EA : 47
```





11.11.4. Méthode adaptative avec deux schémas distincts

Lorsqu'on utilise une des méthodes présentées jusqu'ici, une question ouverte est le choix du pas de discrétisation h pour obtenir une solution précise sans effectuer un nombre excessif de calculs. Une idée naturelle consiste à ajuster dynamiquement le pas de discrétisation en fonction de l'erreur commise à chaque itération. Si l'erreur est inférieure à une tolérance donnée, on peut augmenter le pas de discrétisation pour accélérer le calcul. Inversement, si l'erreur est trop grande, on peut diminuer le pas de discrétisation pour améliorer la précision. Dans ce cas, la discrétisation h_n n'est plus uniforme mais dépend de l'itération n : $t_{n+1} = t_n + h_n$.
Pour estimer l'erreur de discrétisation locale, nous comparons deux approximations : pour un même pas h, la première calcule u_{n+1} avec un schéma, la seconde calcule u_{n+1} avec un autre schéma. L'erreur de discrétisation locale est alors estimée par la différence entre ces deux approximations.
Cette approche est particulièrement adaptée au cas des équations raides, où la solution varie rapidement sur certaines parties de l'intervalle de temps et lentement sur d'autres. Un pas de discrétisation uniforme peut être trop grand pour les parties lentes et trop petit pour les parties rapides. En ajustant le pas de discrétisation en fonction de l'erreur locale, on peut obtenir une solution précise sans effectuer un nombre excessif de calculs.
Étape I : Implémentation de la fonction `euler_heun_adaptatif`
Créez une fonction `euler_heun_adaptatif(phi, t0, y0, h0, tol, tfin)` qui prend en entrée :

- `phi` : fonction définissant l'équation différentielle $y'(t) = \varphi(t, y(t))$
- `t0` : temps initial
- `y0` : condition initiale $y(t_0)$
- `h0` : pas de temps initial
- `tol` : tolérance pour l'estimation de l'erreur locale

- `tfin` : temps maximal jusqu'où intégrer l'équation.

La fonction doit renvoyer :

- un tableau des temps $[t_0, t_1, \dots, t_N]$.
- un tableau des valeurs $[u_0, u_1, \dots, u_N]$.

L'algorithme d'adaptation du pas de temps suit les étapes suivantes :
Tant que $t_n \geq t_{\text{fin}}$:

1. **Calcul des solutions avec deux pas différents :**
   - Utiliser la méthode d'Euler explicite avec un pas h : $v_{n+1} = u_n + h\varphi(t_n, u_n)$.
   - Utiliser la méthode de Heun avec le même pas h : $w_{n+1} = u_n + \frac{h}{2}\Big(\varphi(t_n, u_n) + \varphi(t_{n+1}, v_{n+1})\Big)$.
2. **Ajustement du pas de temps :**
   - Définir l'erreur locale relative : erreur $= \frac{|w_{n+1} - v_{n+1}|}{\max\limits_{k \leq n} |u_k|}$
   - Si l'erreur est **inférieure** à la tolérance tol, ajouter $t_{n+1} = t_n + h$ et u_{n+1} aux tableaux de résultats, puis doubler h.
   - Si l'erreur est **supérieure** à la tolérance tol, diviser h par 2 (si h est inférieur à $h_{\min}$, continuer avec $h_{\min}$) et refaire le calcul des solutions avec ce nouveau pas de temps.

Étape 2 : Test de la méthode
Considérons le problème de Cauchy

$$\begin{cases} y'(t) = -y(t), \\ y(0) = 1. \end{cases} \qquad t \in [0, 20].$$

Nous voulons comparer les solutions obtenues avec la méthode d'Euler explicite, la méthode de Heun et avec la méthode d'Euler-Heun adaptative. Nous comparerons ensuite le nombre de points utilisés avec les deux méthodes.

- Utiliser Euler explicite avec un pas de temps fixe $h = 1$ (valeur maximale qui garantit la monotonie de la solution). Même exercice avec Heun.
- Utiliser Euler-Heun adaptatif avec $h_0 = 1$ et une tolérance tol $= 5 \times 10^{-3}$.
- Comparer les solutions obtenues en traçant les 3 courbes (et la solution exacte).
- Comparer le nombre de pas effectués par chaque méthode et leurs erreurs respectives.

Correction

On commence par calculer une solution approchée avec la méthode d'Euler explicite. On peut ensuite calculer une solution approchée avec la méthode d'Euler adaptatif. On compare ensuite le nombre de points calculés avec les deux méthodes.

```
def EE(phi,tt,y0):
    N = len(tt)-1
    h = tt[1]-tt[0]
    uu = np.zeros(N+1)
    uu[0] = y0
    for n in range(N):
        k_0 = phi(tt[n],uu[n])
        uu[n+1] = uu[n] + h*k_0
    return uu

def Heun(phi,tt,y0):
    N = len(tt)-1
    h = tt[1]-tt[0]
    uu = np.zeros(N+1)
    uu[0] = y0
    for n in range(N):
        k_0 = phi(tt[n],uu[n])
        k_1 = phi(tt[n]+h,uu[n]+h*k_0)
        uu[n+1] = uu[n] + h/2*(k_0+k_1)
    return uu

def EHA(phi,t0,y0,hinit,tol,tfin):
    hmin = 1.e-12
```

```
    tt = np.array([t0])
    uu = np.array([y0])
    h = hinit
    while tt[-1] < tfin:
        k_0 = phi(tt[-1],uu[-1])
        v = uu[-1] + h*k_0
        k_1 = phi(tt[-1]+h,v)
        w = uu[-1] + h/2*(k_0 + k_1)
        erreur = np.abs(w - v)/np.linalg.norm(uu,ord=np.inf)
        if erreur < tol or h <= hmin:
            tt = np.append(tt,tt[-1]+h)
            uu = np.append(uu,w)
            h = 2*h
        else :
            h = max(h/2,hmin)
    return tt,uu
```

```
# TEST
# ========================

phi = lambda t, y : -y
t0, tfin = 0, 40
y0 = 1

# Pour la A-stabilité, il faut que h<2, pour la monotonie h<=1
# h = 1 pour N = (tfin-t0)/h = tfin-t0

N = int(tfin-t0)
tt = np.linspace(t0,tfin,N+1)


plt.figure(figsize=(8,5))

# Solution exacte
sol_exacte = lambda t : y0 * np.exp(-t)
plt.plot(tt,sol_exacte(tt),'-',label='exacte', color='green', lw=2)

# EE
uu_EE = EE(phi,tt,y0)
plt.plot(tt,uu_EE,'o-',label='EE', color='blue')

# Heun
uu_Heun = Heun(phi,tt,y0)
plt.plot(tt,uu_Heun,'o-',label='Heun', color='orange')

# EHA
hinit = tt[1]-tt[0]
tol = 5.e-3
tt_EHA, uu_EHA = EHA(phi,t0,y0,hinit,tol,tfin)
plt.plot(tt_EHA,uu_EHA,'o-',label='EA', color='red')

plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left');


# Affichons juste les deux grilles de temps sur une droite horizontale
plt.figure(figsize=(8,5))
plt.plot(tt,np.ones_like(tt),'o-',label='EE',color='blue')
plt.plot(tt,0.5-np.empty_like(tt),'o-',label='Heun',color='orange')
plt.plot(tt_EHA,1-np.ones_like(tt_EHA),'o-',label='EHA',color='red')
plt.grid()
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.title('Grilles de temps');
```

```
# print(f"Erreur EE : {np.linalg.norm(uu_EE-sol_exacte(tt))}")
# print(f"Erreur Heun : {np.linalg.norm(uu_Heun-sol_exacte(tt))}")
# print(f"Erreur EHA : {np.linalg.norm(uu_EHA-sol_exacte(tt_EHA))}")
# print(f"Nombre de pas EE : {len(tt)-1}")
# print(f"Nombre de pas Heun : {len(tt)-1}")
# print(f"Nombre de pas EHA : {len(tt_EHA)-1}")

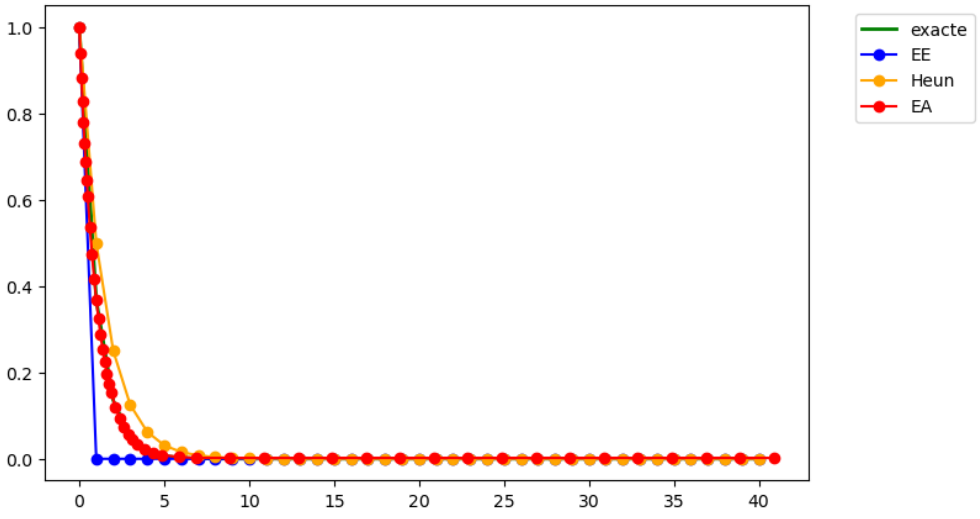

# Affichage des résultats avec pandas
# ===================================

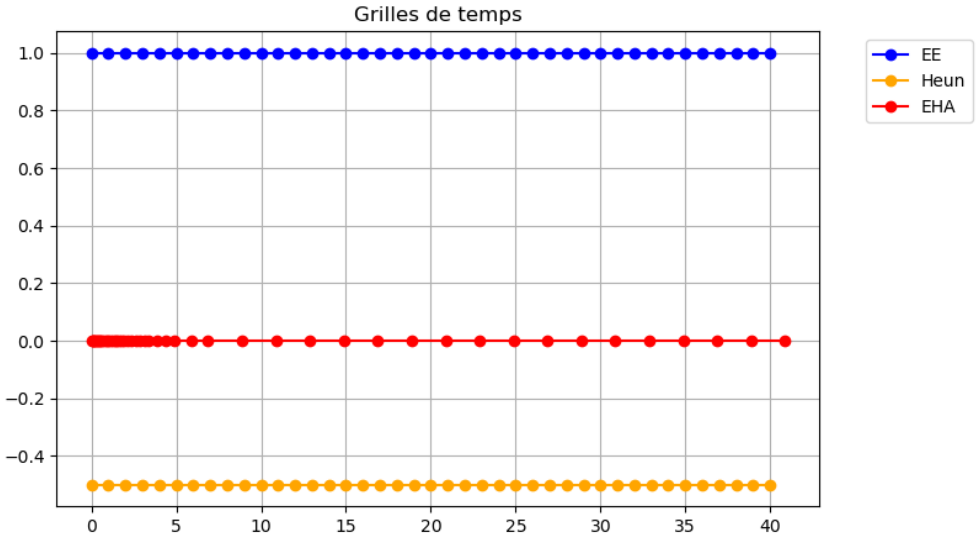
import pandas as pd

# Création du DataFrame contenant les résultats
data = {
    "Méthode": ["Euler Explicite", "Heun", "Euler-Heun Adaptatif"],
    "Erreur": [
        np.linalg.norm(uu_EE - sol_exacte(tt)),
        np.linalg.norm(uu_Heun - sol_exacte(tt)),
        np.linalg.norm(uu_EHA - sol_exacte(tt_EHA))
    ],
    "Nombre de pas": [
        len(tt) - 1,
        len(tt) - 1,
        len(tt_EHA) - 1
    ]
}

pd.DataFrame(data)
```

	Méthode	Erreur	Nombre de pas
0	Euler Explicite	0.395623	40
1	Heun	0.197615	40
2	Euler-Heun Adaptatif	0.009670	47




Grilles de temps
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
0
5
10
15
20
25
30
35
40
EE
Heun
EHA

Troisième partie

TP

Chapitre 12.

Calculs analytiques avec le module sympy

En L1 vous avez appris comment calculer la solution générale de quelques équations différentielles ordinaires (EDO). Nous allons maintenant voir comment automatiser ces calculs avec le module `sympy` conçu pour du calcul symbolique/formel/exact.

Cf. https://docs.sympy.org/latest/modules/solvers/ode.html

> **i** Plan de la séance de TP (partie 1)
>
> Bien lire et comprendre les deux exemples pour résoudre des EDO du premier ordre et des systèmes d'EDO du premier ordre, puis résoudre les deux exercices proposés.
> - Exemples :
> - EDO du premier ordre
> - système d'EDO du premier ordre
> - EDO d'ordre 2
> - Exercices
> - EDO du premier ordre
> - Système d'EDO du premier ordre

```
import sympy as sp
from IPython.display import display, Markdown # For pretty printing

import matplotlib.pyplot as plt
import numpy as np
# plt.rcdefaults()
# plt.rcParams.update({'font.size': 16})
```

12.1. Exemples

12.1.1. Exemple de résolution analytique d'une équation différentielle d'ordre 1

Considérons le problème de Cauchy

$$\begin{cases} u'(x) = -3x^2u(x) + 6x^2, \\ u(0) = 4. \end{cases}$$

On peut résoudre ce problème de Cauchy de manière analytique en utilisant le module `sympy` de la manière suivante.

Définition du problème de Cauchy :

```
# Variables
x = sp.Symbol('x')
u = sp.Function('u')
```

```
# Equation différentielle
left  = sp.diff(u(x),x) # la dérivée de u par rapport à x
right = 6*x**2-3*x**2*u(x) # l'expression de l'équation différentielle
edo = sp.Eq( left , right ) # l'équation différentielle
display(edo)

# Conditions initiales
x0 = 0
u0 = 4
display(sp.Eq(u(x0),u0))
```

$$\frac{d}{dx}u(x) = -3x^2u(x) + 6x^2$$

$$u(0) = 4$$

Calcul de la solution exacte (2 méthodes : une méthode directe et une méthode pas à pas) :

```
# Méthode 1 : méthode directe
# ============================

display(Markdown("- Méthode 1 : méthode directe"))
solpar = sp.dsolve(edo,u(x), ics={u(x0):u0}) # l'edo, la fonction u(x) inconnue, les conditions initiales
 ↪  comme dictionnaire
display(solpar)
```

- Méthode 1 : méthode directe

$$u(x) = 2 + 2e^{-x^3}$$

```
# Méthode 2 : méthode pas-à-pas
# ==============================
# On va calculer la solution générale, puis la solution particulière en prenant en compte
# la condition initiale

display(Markdown("- Méthode 2 : méthode pas-à-pas"))

# Étape 1 : calcul de la solution générale (qui contient la constante d'intégration) :
solgen = sp.dsolve(edo,u(x))
display(solgen)

# Étape 2 : calcul de la constante d'intégration en prenant en compte la condition initiale :
cond_init = sp.Eq( u0, solgen.rhs.subs(x,x0) )
display(cond_init)
consts = sp.solve( cond_init , dict=True)[0]
display(consts)

# Étape 3 : calcul de la solution particulière :
solpar = solgen.subs(consts)
display(solpar)
```

- Méthode 2 : méthode pas-à-pas

$$u(x) = C_1e^{-x^3} + 2$$

$$4 = C_1 + 2$$

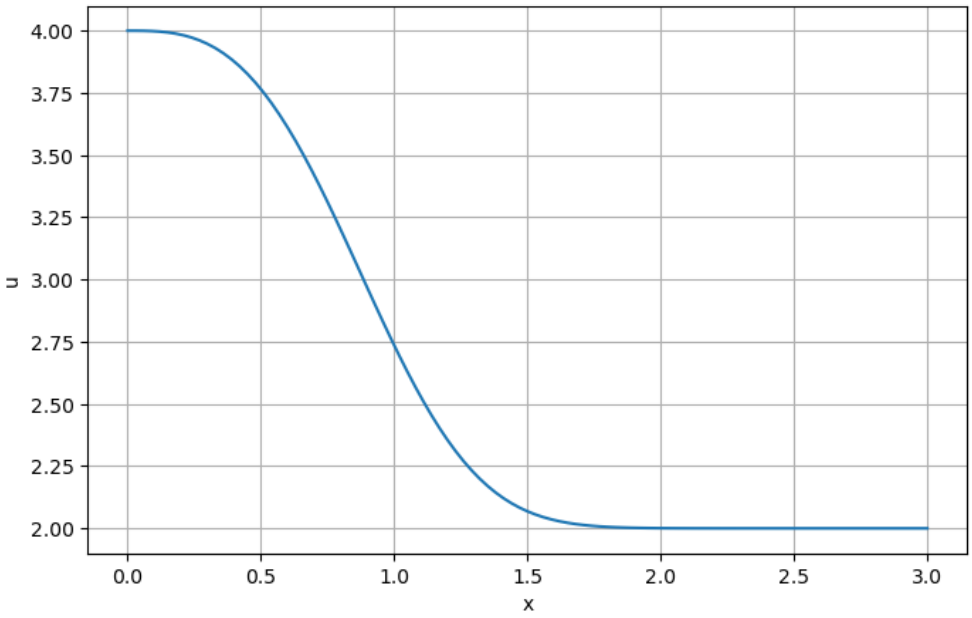
```
{C1: 2}
```

$$u(x) = 2 + 2e^{-x^3}$$

Conversion en une fonction de `numpy` et affichage :

```
# Transformation en fonction numpy
func = sp.lambdify(x,solpar.rhs,'numpy')

# Tracé de la solution
plt.figure(figsize=(8,5))
xx = np.linspace(0,3,101)
yy = func(xx)
plt.plot(xx,yy)
plt.xlabel('x')
plt.ylabel('u')
plt.grid();
```



12.1.2. Exemple de résolution analytique d'un système d'équations différentielles d'ordre 1

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix}' (t) = \begin{pmatrix} \varphi_1(t, y_1(t), y_2(t)) \\ \varphi_2(t, y_1(t), y_2(t)) \end{pmatrix} = \begin{pmatrix} y_1(t) - y_2(t) \\ y_2(t) - y_1(t) \end{pmatrix}$$

avec $t \in [0; 10]$ et en prenant comme condition initiale le vecteur $\mathbf{y}_0 = (2, 1)$.

```
# Variables
t  = sp.Symbol('t')

# Fonctions inconnues
y1 = sp.Function(r'y_1')
y2 = sp.Function(r'y_2')

# Termes de droite
right1 = y1(t)-y2(t)
right2 = y2(t)-y1(t)

# Équations différentielles
edo1 = sp.Eq( sp.diff(y1(t),t) , right1 )
edo2 = sp.Eq( sp.diff(y2(t),t) , right2 )
display(edo1)
display(edo2)
```

```
# Conditions initiales
t0   = 0
y1_0 = 2
y2_0 = 1
display(sp.Eq(y1(t0),y1_0))
display(sp.Eq(y2(t0),y2_0))
```

$$\frac{d}{dt}y_1(t) = y_1(t) - y_2(t)$$

$$\frac{d}{dt}y_2(t) = -y_1(t) + y_2(t)$$

$$y_1(0) = 2$$

$$y_2(0) = 1$$

```
# Méthode 1 : méthode directe
# ============================

solpar_1, solpar_2 = sp.dsolve( [ edo1, edo2 ] ,
                                ics = { y1(t0):y1_0, y2(t0):y2_0 } )
display(solpar_1)
display(solpar_2)


# Méthode 2 : méthode pas-à-pas
# ==============================
# solgen = sp.dsolve([edo1,edo2],[y1(t),y2(t)])
# display(solgen)
# consts = sp.solve( [sp.Eq( y1_0, solgen[0].rhs.subs(t,t0)) ,
#                     sp.Eq( y2_0, solgen[1].rhs.subs(t,t0)) ] , dict=True)[0]
# display(consts)
# solpar_1 = solgen[0].subs(consts)
# solpar_2 = solgen[1].subs(consts)
# display(solpar_1)
# display(solpar_2)
```

$$y_1(t) = \frac{e^{2t}}{2} + \frac{3}{2}$$

$$y_2(t) = \frac{3}{2} - \frac{e^{2t}}{2}$$

```
# Affichage
# =========

# Transformation des solutions en fonctions numpy
func_1 = sp.lambdify(t,solpar_1.rhs,'numpy')
func_2 = sp.lambdify(t,solpar_2.rhs,'numpy')

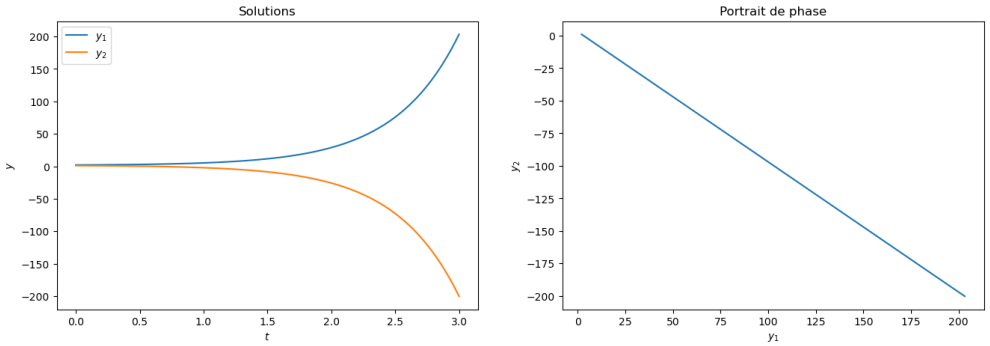

plt.figure(figsize=(16,5))

tt = np.linspace(0,3,101)
yy_1 = func_1(tt)
yy_2 = func_2(tt)

plt.subplot(1,2,1)
plt.plot(tt,yy_1,tt,yy_2)
plt.legend([r'$y_1$',r'$y_2$'])
plt.xlabel(r'$t$')
plt.ylabel(r'$y$')
```

```
plt.title("Solutions")

plt.subplot(1,2,2)
plt.plot(yy_1,yy_2)
plt.xlabel(r'$y_1$')
plt.ylabel(r'$y_2$')
plt.title("Portrait de phase");
```



12.1.3. Exemple de résolution analytique d'une équation différentielle d'ordre 2

Calculons l'unique solution du problème de Cauchy

$$\begin{cases} y''(t) + 2y'(t) + 2y(t) = 2\sin(t), & t > 0 \\ y(0) = 1, \\ y'(0) = 0. \end{cases}$$

- D'abord avec la méthode directe en définissant l'équation différentielle d'ordre 2 et les conditions initiales.
- Ensuite avec la méthode pas à pas en définissant l'équation différentielle d'ordre 2 et les conditions initiales.
- Enfin, en transformant l'équation différentielle d'ordre 2 en un système d'équations différentielles d'ordre 1 (voir l'exercice précédent) et en utilisant la méthode pas à pas.

Méthode directe sur l'équation différentielle d'ordre 2 :

```
# Variables
t = sp.Symbol('t')

# Fonction inconnue
y = sp.Function('y')

# Équation différentielle
left  = sp.diff(y(t),t,2) + 2*sp.diff(y(t),t) + 2*y(t)
right =  2*sp.sin(t)
edo = sp.Eq( left , right ) # l'équation différentielle
display(edo)

# Conditions initiales
t0 = 0
y0 = 1
yp0 = 0

display(sp.Eq(y(t0),y0))
display(sp.Eq(sp.diff(y(t),t).subs(t,t0),yp0))

# Méthode 1 : méthode directe
# ===========================
solpar = sp.dsolve(edo,y(t), ics={y(t0):y0, sp.diff(y(t),t).subs(t,t0):yp0})
display(solpar)
```

$$2y(t) + 2\frac{d}{dt}y(t) + \frac{d^2}{dt^2}y(t) = 2\sin(t)$$

$$y(0) = 1$$

$$\left.\frac{d}{dt}y(t)\right|_{t=0} = 0$$

$$y(t) = \left(\frac{7\sin(t)}{5} + \frac{9\cos(t)}{5}\right)e^{-t} + \frac{2\sin(t)}{5} - \frac{4\cos(t)}{5}$$

Vérifions si elle satisfait bien notre problème de Cauchy :

```
# Qua donne la solution trouvée lorsqu'on la remplace dans l'équation différentielle ?
display( (sp.diff(solpar.rhs,t,2) + 2*sp.diff(solpar.rhs,t) + 2*solpar.rhs   -2*sp.sin(t)   ).simplify()
 ↪  )
# Que vaut la solution trouvée lorsqu'on la remplace dans les conditions initiales ?
display( solpar.rhs.subs(t,t0) )
display( sp.diff(solpar.rhs,t).subs(t,t0) )
```

0

1

0

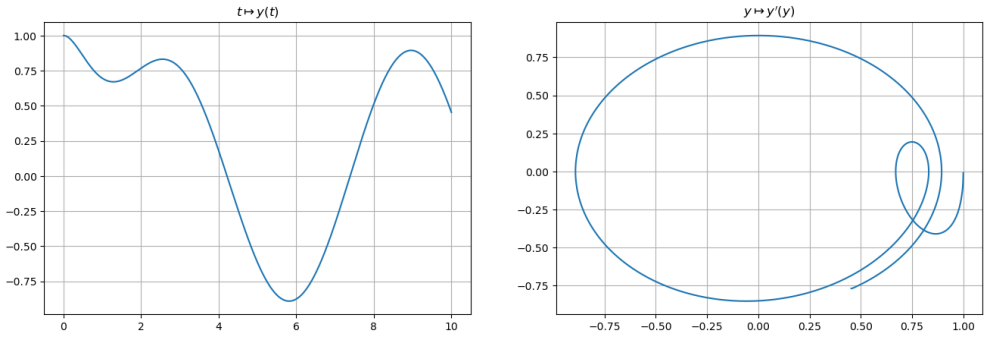
```
# Affichage de la solution
func = sp.lambdify(t,solpar.rhs,'numpy')

tt = np.linspace(0,10,1001)
yy = func(tt)


plt.figure(figsize=(16,5))

# Solution
# ========
plt.subplot(1,2,1)
plt.plot(tt,yy)
plt.title(r"$t\mapsto y(t)$")
plt.grid();

# Portraits de phase
# ==================
plt.subplot(1,2,2)
plt.plot(yy,np.gradient(yy,tt))
plt.title(r"$y\mapsto y'(y)$")
plt.grid();
```



Méthode pas à pas sur l'équation différentielle d'ordre 2 :

```
# Bonus : résolution de l'équation différentielle avec la méthode pas-à-pas
# ========================================================================

# Étape 1 : calcul de la solution générale (qui contient la constante d'intégration) :
solgen = sp.dsolve(edo,y(t))
display(solgen)

# Étape 2 : calcul de la constante d'intégration en prenant en compte les conditions initiales :
cond_init1 = sp.Eq( y0, solgen.rhs.subs(t,t0) )
cond_init2 = sp.Eq( yp0, sp.diff(solgen.rhs,t).subs(t,t0) )
display(cond_init1)
display(cond_init2)

consts = sp.solve( [cond_init1,cond_init2] , dict=True)[0]
display(consts)

# Étape 3 : calcul de la solution particulière :
solpar = solgen.subs(consts)
display(solpar)
```

$$y(t) = \left(C_1 \sin\left(t\right) + C_2 \cos\left(t\right)\right) e^{-t} + \frac{2 \sin\left(t\right)}{5} - \frac{4 \cos\left(t\right)}{5}$$

$$1 = C_2 - \frac{4}{5}$$

$$0 = C_1 - C_2 + \frac{2}{5}$$

```
{C1: 7/5, C2: 9/5}
```

$$y(t) = \left(\frac{7 \sin\left(t\right)}{5} + \frac{9 \cos\left(t\right)}{5}\right) e^{-t} + \frac{2 \sin\left(t\right)}{5} - \frac{4 \cos\left(t\right)}{5}$$

Méthode directe ou pas à pas après transformation en un système d'équations différentielles d'ordre 1 :

```
# Bonus : transformation en un système d'équations différentielles du premier ordre
# ================================================================================

# Variables
t = sp.Symbol('t')
y = sp.Function('y')
z = sp.Function('z')

# Équations
edo1 = sp.Eq( sp.diff(y(t),t) , z(t) )
edo2 = sp.Eq( sp.diff(z(t),t) , 2*sp.sin(t) - 2*z(t) - 2*y(t) )
display(edo1)
display(edo2)

# Conditions initiales
t0 = 0
y0 = 1
yp0 = 0

display(sp.Eq(y(t0),y0))
display(sp.Eq(z(t0),yp0))

# Méthode 1 : méthode directe
# ===========================
```

```
display(Markdown("- Méthode 1 : méthode directe"))
solpar = sp.dsolve( [edo1,edo2], [y(t),z(t)], ics={y(t0):y0, z(t0):yp0} )
display(solpar)

# Méthode 2 : méthode pas-à-pas
# ==============================
display(Markdown("- Méthode 2 : méthode pas-à-pas"))
# On va calculer la solution générale, puis la solution particulière en prenant en compte les conditions
↪ initiales
solgen = sp.dsolve( [edo1,edo2], [y(t),z(t)] )
display(solgen)

# Étape 2 : calcul de la constante d'intégration en prenant en compte les conditions initiales :
cond_init1 = sp.Eq( y0, solgen[0].rhs.subs(t,t0) )
cond_init2 = sp.Eq( yp0, solgen[1].rhs.subs(t,t0) )
display(cond_init1)
display(cond_init2)

consts = sp.solve( [cond_init1,cond_init2] , dict=True)[0]
display(consts)

# Étape 3 : calcul de la solution particulière :
solpar = [sol.subs(consts) for sol in solgen]
display(solpar)
```

$$\frac{d}{dt}y(t) = z(t)$$

$$\frac{d}{dt}z(t) = -2y(t) - 2z(t) + 2\sin(t)$$

$$y(0) = 1$$

$$z(0) = 0$$

- Méthode 1 : méthode directe

```
[Eq(y(t), 2*sin(t)**3/5 - 4*sin(t)**2*cos(t)/5 + 2*sin(t)*cos(t)**2/5 - 4*cos(t)**3/5 + 7*exp(-
t)*sin(t)/5 + 9*exp(-t)*cos(t)/5),
 Eq(z(t), 4*sin(t)**3/5 + 2*sin(t)**2*cos(t)/5 + 4*sin(t)*cos(t)**2/5 + 2*cos(t)**3/5 - 16*exp(-
t)*sin(t)/5 - 2*exp(-t)*cos(t)/5)]
```

- Méthode 2 : méthode pas-à-pas

```
[Eq(y(t), -(C1/2 - C2/2)*exp(-t)*cos(t) + (C1/2 + C2/2)*exp(-t)*sin(t) + 2*sin(t)**3/5 - 4*sin(t)**2*
 Eq(z(t), C1*exp(-t)*cos(t) - C2*exp(-t)*sin(t) + 4*sin(t)**3/5 + 2*sin(t)**2*cos(t)/5 + 4*sin(t)*cos
```

$$1 = -\frac{C_1}{2} + \frac{C_2}{2} - \frac{4}{5}$$

$$0 = C_1 + \frac{2}{5}$$

```
{C1: -2/5, C2: 16/5}
```

```
[Eq(y(t), 2*sin(t)**3/5 - 4*sin(t)**2*cos(t)/5 + 2*sin(t)*cos(t)**2/5 - 4*cos(t)**3/5 + 7*exp(-
t)*sin(t)/5 + 9*exp(-t)*cos(t)/5),
 Eq(z(t), 4*sin(t)**3/5 + 2*sin(t)**2*cos(t)/5 + 4*sin(t)*cos(t)**2/5 + 2*cos(t)**3/5 - 16*exp(-
t)*sin(t)/5 - 2*exp(-t)*cos(t)/5)]
```

12.2. Exercices

12.2.1. Exercice (EDO d'ordre 1)

Calculer la solution **exacte** du problème de Cauchy

$$\begin{cases} y'(t) = \dfrac{3}{2} y(t) \left(1 - \dfrac{y(t)}{6}\right), \quad t > 0 \\ y(0) = 1. \end{cases}$$

- Utiliser la méthode directe pour calculer puis afficher l'unique solution.
- Que vaut $\lim_{t \to \infty} y(t)$?
- Bonus : si on utilise la méthode pas à pas, que remarquez-vous sur la solution générale ? Comment retrouver la solution directe ?

Remarque : ne pas écrire `3/2` mais `sympy.Rational(3,2)` pour éviter que ce terme soit évalué comme un flottant.

```
t  = sp.Symbol('t')
y  = sp.Function('y')
right = sp.Rational(3,2)*y(t)*(1-y(t)/6) # Attention à ne pas écrire 3/2 qui donne 1.5
edo = sp.Eq( sp.diff(y(t),t) , right )
display(edo)

t0 = 0
y0 = 1
display(sp.Eq(y(t0),y0))
```

$$\frac{d}{dt} y(t) = \frac{3 \cdot \left(1 - \frac{y(t)}{6}\right) y(t)}{2}$$

$$y(0) = 1$$

```
# Méthode directe
# ===============
solpar = sp.dsolve(edo,y(t), ics={y(t0):y0})
display(solpar)
```

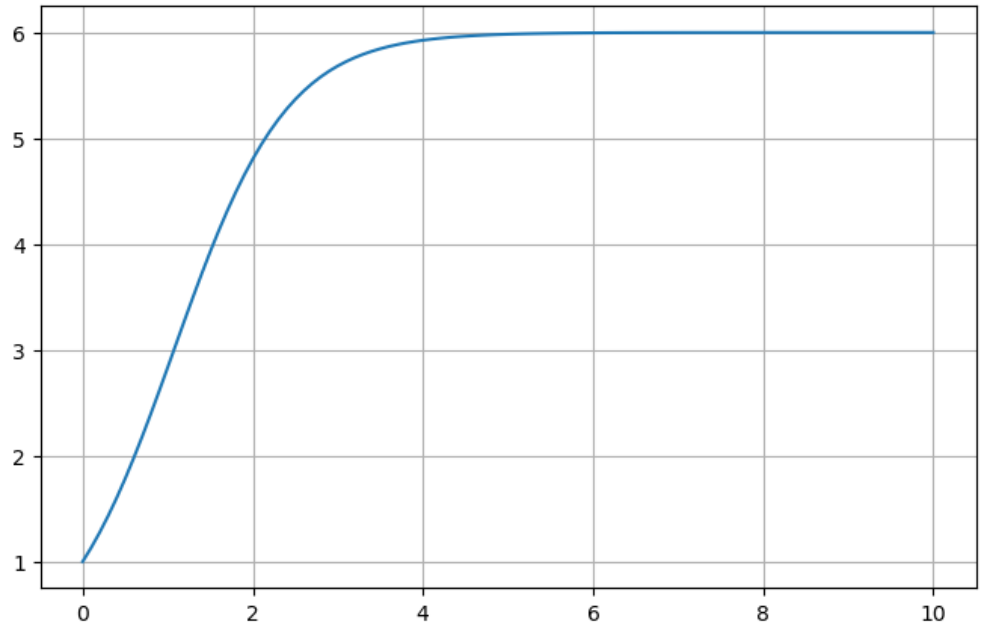
$$y(t) = \frac{6 \left(-e^{3t} + 5\sqrt{e^{3t}}\right)}{25 - e^{3t}}$$

```
# Affichage
# =========


# Étape 1 : conversion de la solution en fonction numpy
func = sp.lambdify(t,solpar.rhs,'numpy')

# Étape 2 : plot

plt.figure(figsize=(8,5))
tt = np.linspace(0,10,101)
yy = func(tt)
plt.plot(tt,yy)
plt.grid();
```




```
# Calcul de la limite
# ===================

limite = sp.limit(solpar.rhs,t,sp.oo)
display( sp.Eq( sp.Limit(y(t),t,sp.oo) , limite ) )
```

$$\lim_{t\to\infty} y(t) = 6$$

Bonus : dans cet exemple, si on veut d'abord trouver la solution générale, on s'aperçoit qu'il y a deux expressions possibles. On doit alors choisir celle qui convient à notre condition initiale. Comme la solution du problème de Cauchy est unique, une seule des deux expressions est correcte.

```
# Méthode pas-à-pas
# =================

# solgenLIST = sp.dsolve(edo,y(t))
# display(solgenLIST)

# On teste avec la première expression, si ça ne marche pas, on essaie avec la seconde
# ====================================================================================
# solgen = solgenLIST[0]
# display(solgen)

# equation = sp.Eq( y0, solgen.rhs.subs(t,t0))
# display(equation)
# consts = sp.solve( equation , dict=True)[0]
# display(consts)

# solpar = solgen.subs(consts)
# display(solpar)
```

12.2.2. Exercice (système et EDO d'ordre 2)

- Calculer la solution **exacte** du système

$$\begin{pmatrix} x(t) \\ y(t) \end{pmatrix}' = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} x(t) \\ y(t) \end{pmatrix}$$

  pour $x(0) = -1$ et $y(0) = 1$.

- Afficher $t \mapsto x$ et $t \mapsto y$ dans un même graphe.
- Afficher ensuite $x \mapsto y$.
- Calculer analytiquement $E'(t)$ où $t \mapsto E(t)$ est l'énergie mécanique du système définie par

$$E(t) = \frac{x^2(t)}{2} + \frac{y^2(t)}{2}.$$

- Bonus : transformer le système en une équation différentielle d'ordre 2 et résoudre le problème de Cauchy associé.

La première équation s'écrit

$$x'(t) = y(t)$$

et la deuxième

$$y'(t) = -x(t).$$

```
t = sp.Symbol('t')
x = sp.Function('x')
y = sp.Function('y')

edo1 = sp.Eq( sp.diff(x(t),t) ,  y(t) )
edo2 = sp.Eq( sp.diff(y(t),t) , -x(t) )
display(edo1)
display(edo2)

t_0 = 0
x_0 = -1
y_0 = 1
display(sp.Eq(x(t_0),x_0))
display(sp.Eq(y(t_0),y_0))
```

$$\frac{d}{dt}x(t) = y(t)$$

$$\frac{d}{dt}y(t) = -x(t)$$

$$x(0) = -1$$

$$y(0) = 1$$

```
# Méthode directe
# ===============

solpar_1, solpar_2 = sp.dsolve( [edo1,edo2],
                                [x(t),y(t)],
                                ics={x(t_0):x_0, y(t_0):y_0} )
display(solpar_1)
display(solpar_2)


# Méthode pas-à-pas
# =================

# solgen = sp.dsolve([edo1,edo2],[x(t),y(t)])
# display(solgen)

# consts = sp.solve( [ sp.Eq( x_0, solgen[0].rhs.subs(t,t_0)) , sp.Eq( y_0, solgen[1].rhs.subs(t,t_0)) ]
 ↪  , dict=True)[0]
# display(consts)
```

```
# solpar_1 = solgen[0].subs(consts)
# solpar_2 = solgen[1].subs(consts)
# display(solpar_1)
# display(solpar_2)
```

$x(t) = \sin(t) - \cos(t)$

$y(t) = \sin(t) + \cos(t)$

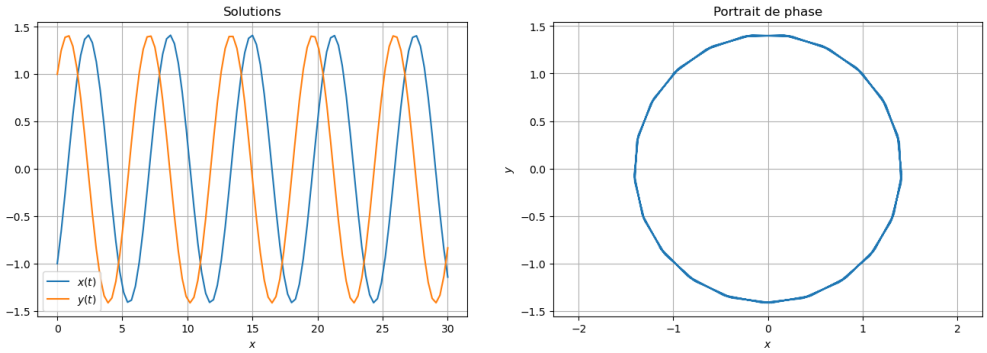
```
# Conversion en fonctions numpy
# =============================
func_1 = sp.lambdify(t,solpar_1.rhs,'numpy')
func_2 = sp.lambdify(t,solpar_2.rhs,'numpy')

# Affichage
# =========

plt.figure(figsize=(16,5))
tt = np.linspace(0,30,101)
xx = func_1(tt)
yy = func_2(tt)

plt.subplot(1,2,1)
plt.plot(tt,xx,tt,yy)
plt.legend([r'$x(t)$',r'$y(t)$'])
plt.xlabel(r'$x$')
plt.grid()
plt.title('Solutions')

plt.subplot(1,2,2)
plt.plot(xx,yy)
# plt.axis('square')
plt.axis('equal')
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
plt.grid()
plt.title('Portrait de phase');
```



La fonction $t \mapsto E(t) = \frac{x^2(t)}{2} + \frac{y^2(t)}{2}$ vérifie

$$\begin{aligned} E'(t) &= x(t)x'(t) + y(t)y'(t) \\ &= x(t)\varphi_1(t, x(t), y(t)) + y(t)(t)\varphi_2(t, x(t), y(t)) \\ &= x(t)y(t) + y(t)(-x(t)) \\ &= 0. \end{aligned}$$

Vérifions qu'effectivement E est une constante en remplaçant x et y par leurs expressions analytiques.

```
# On calcule E et E' analytiquement
# ================================

# La fonction E(t) = x(t)^2/2 + y(t)^2/2
E = (solpar_1.rhs)**2/2+(solpar_2.rhs)**2/2
E = E.simplify()
display(sp.Eq(  sp.Symbol('E(t)') , E  ))

# La dérivée de E par rapport à t
Ep = sp.diff(E,t).simplify()
display(sp.Eq(  sp.Symbol("E'(t)") , Ep  ))
```

$E(t) = 1$

$E'(t) = 0$

Transformation et résolution du système en une seule équation différentielle du deuxième ordre

On a $x'(t) = y(t)$ et $y'(t) = -x(t)$, on obtient alors $x''(t) = y'(t) = -x(t)$.
De plus, $x(0) = -1$ et $x'(0) = y(0) = 1$.

```
# Variables
t = sp.Symbol('t')

# Fonction inconnue
x = sp.Function('x')

# Équation différentielle
left  = sp.diff(x(t),t,2)
right =  -x(t)
edo = sp.Eq( left , right ) # l'équation différentielle
display(edo)

# Conditions initiales
t0 = 0
x0 = -1
xp0 = 1

display(sp.Eq(x(t0),x0))
display(sp.Eq(sp.diff(x(t),t).subs(t,t0),xp0))

# Méthode 1 : méthode directe
# ===========================
solpar_x = sp.dsolve(edo,x(t), ics={x(t0):x0, sp.diff(x(t),t).subs(t,t0):xp0})
display(solpar_x)
solpar_y = sp.diff(solpar_x.rhs,t)
display( sp.Eq( y(t) , solpar_y ) )
```

$\frac{d^2}{dt^2}x(t) = -x(t)$

$x(0) = -1$

$\left.\frac{d}{dt}x(t)\right|_{t=0} = 1$

$x(t) = \sin(t) - \cos(t)$

$y(t) = \sin(t) + \cos(t)$

Chapitre 13.

Calcul approché avec la fonction `odeint` du module `SciPy`

> **i** Plan de la séance de TP (partie 2)
>
> Bien lire et comprendre les trois exemples qui montrent comment approcher avec la fonction `odeint` la solution des EDO du premier ordre, des systèmes d'EDO du premier ordre ou d'une EDO d'ordre supérieur à 1 après transformation en un système d'EDO du premier ordre.
> Apprendre à tracer le champ de vecteurs associé à une EDO du premier ordre ou le portrait de phase associé à un système d'EDO du premier ordre avec la fonction `quiver`.
> Puis résoudre les deux premiers exercices proposés.
>
> - Exemples :
>   - EDO du premier ordre + champ de vecteurs
>   - Système d'EDO du premier ordre + portrait de phase
>   - EDO d'ordre 2 $\rightsquigarrow$ système d'EDO du premier ordre + portrait de phase
> - Exercices :
>   - EDO du premier ordre
>   - Système d'EDO du premier ordre
>   - Bonus : étude de la stabilité d'un système d'EDO du premier ordre

```
# from matplotlib.pyplot import * # importe aussi numpy sans alias
import matplotlib.pyplot as plt
import numpy as np

# plt.rcdefaults()
# plt. rcParams.update({'font.size': 12})
from scipy.integrate import odeint, solve_ivp
```

Ce cours est une introduction aux méthodes numériques d'approximation d'Équations Différentielles Ordinaires (EDO) en utilisant Python comme langage de programmation. Au cours de ce module, nous explorerons, implémenterons et testerons plusieurs méthodes numériques.

Le module `SciPy` (et d'autres modules plus spécifiques) offre déjà des méthodes numériques pour résoudre des EDO, y compris celles que nous allons étudier dans ce module. Bien que ces méthodes soient déjà implémentées dans des fonctions prêtes à l'emploi, il est important de comprendre les principes de base de ces méthodes pour les utiliser correctement.

Pour illustrer l'utilisation pratique de ce module, nous allons voir sur trois exemples comment utiliser la fonction `odeint` du module `SciPy` pour approcher la solution d'abord d'une EDO ensuite d'un système d'EDO (ce qui inclut les équations différentielles d'ordre 2 ou plus).

Voici la signature de la fonction `odeint` pour référence :

```
scipy.integrate.odeint(func, y0, t, args=(), Dfun=None, col_deriv=0, full_output=0, ml=None,
mu=None, rtol=None, atol=None, tcrit=None, h0=0.0, hmax=0.0, hmin=0.0, ixpr=0, mxstep=0,
mxhnil=0, mxordn=12, mxords=5, printmessg=0, tfirst=False)
```

Cette fonction résout un problème de Cauchy d'ordre 1, c'est-à-dire une ou plusieurs EDO d'ordre 1, définies par la fonction `func` et une condition initiale `y0`. La fonction `func` doit être définie par l'utilisateur et doit renvoyer la dérivée de `y` par rapport à `t` à un point donné `t`.

La fonction `odeint` renvoie un tableau de valeurs de `y` pour chaque valeur de `t` spécifiée dans le tableau `t` :
`dy/dt = func(y, t, ...)  [or func(t, y, ...)]`

Les arguments principaux sont :

- `func` la fonction qui définit le problème de Cauchy,
- `y0` la condition initiale,
- `t` le tableau des temps où on veut approcher la solution.

La méthode d'intégration utilisée est `LSODA` (méthode Adam/BDF adaptative).

Vous trouverez des informations détaillées dans la documentation officielle de `SciPy` : https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.html

13.1. Principe d'utilisation de `odeint`

Le principe d'utilisation de `odeint` (pour intégrer numériquement des équations différentielles) est le suivant : pour avoir une estimation numérique de la solution du problème de Cauchy

$$\begin{cases} \mathbf{y}'(t) = \varphi(t, \mathbf{y}(t)), \\ \mathbf{y}(t_0) = \mathbf{y}_0 \end{cases}$$

avec $\mathbf{y}(t) = (y_1(t), y_2(t), \dots, y_n(t))$ le vecteur des fonctions recherchées dépendant de la variable t et $\varphi = (\varphi_1, \varphi_2, \dots, \varphi_n)$ une fonction de $\mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$, on donne comme argument la fonction φ (qui doit avoir deux paramètres, même dans le cas autonome), la condition initiale $\mathbf{y}_0$ et les points de l'interval de temps où approcher la solution (qui commence à t_0). Elle retourne un tableau `Numpy` (même si t était une liste).

Par défaut, l'ordre requis des deux premiers arguments de `func` est dans l'ordre inverse de celui classique. Pour utiliser une fonction avec la signature `func(t, y, ...)`, c'est-à-dire d'abord t puis y, l'argument `tfirst` doit être défini à `True`.

Remarque : la résolution de $y''(t) = F(t, y(t), y'(t))$ passera par celle du système différentiel

$$\begin{cases} y_1'(t) = y_2(t), \\ y_2'(t) = F(t, y_1(t), y_2(t)) \end{cases}$$

avec $y(t) = y_1(t)$, $y'(t) = y_1'(t) = y_2(t)$ et $\varphi_1(t, y_1(t), y_2(t)) = y_2(t)$, $\varphi_2(t, y_1(t), y_2(t)) = F(t, y_1(t), y_2(t))$.

Et la fonction `solve_ivp` ?

SciPy propose une interface plus récente et plus personnalisable pour résoudre un problème de Cauchy, il s'agit de `solve_ivp`. L'avantage principal est que `solve_ivp` offre plusieurs méthodes pour résoudre les équations différentielles alors qu'`odeint` est limité à une seule. Cependant, la prise en main de `odeint` est plus simple et suffisante pour les objectifs de ce cours.
Cf. https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html
Voici la signature de la fonction `solve_ivp` pour référence :
`scipy.integrate.solve_ivp(fun, t_span, y0, method='RK45', t_eval=None, dense_output=False, events=None, vectorized=False, args=None, **options)`
Les arguments principaux sont :

- `fun` la fonction qui définit le problème de Cauchy,
- `t_span` un tuple qui contient l'intervalle de temps,
- `y0` la condition initiale,
- `method` la méthode d'intégration,
- `t_eval` le tableau des temps où on veut approcher la solution.

Parmi les méthodes d'intégration, on trouve `RK45` (méthode explicite de Runge-Kutta d'ordre 5(4)), `RK23` (méthode explicite de Runge-Kutta d'ordre 3(2)), `DOP853` (méthode de Dormand-Prince d'ordre 8(5)), `Radau`

(méthode implicite de Runge-Kutta d'ordre 5), `BDF` (méthode implicite multipas d'ordre variable) et `LSODA` (méthode Adam/BDF adaptative).
Les méthodes explicites de Runge-Kutta (RK23, RK45, DOP853) doivent être utilisées pour les problèmes non stiff et les méthodes implicites ('Radau', 'BDF') pour les problèmes stiff (on verra ce concept au CM-5). Parmi les méthodes de Runge-Kutta, DOP853 est recommandée pour la résolution avec une grande précision. En cas de doute, essayez d'abord d'exécuter RK45. S'il effectue un nombre anormalement élevé d'itérations, diverge ou échoue, il est probable que le problème soit stiff et que l'on doit utiliser Radau ou BDF. LSODA peut également être un bon choix universel, mais il peut être un peu moins pratique à utiliser car il intègre un ancien code Fortran.

13.2. Champ de vecteurs et lignes de courant pour une EDO du premier ordre

Bien qu'il soit trés rare que l'on puisse résoudre explicitement une équation différentielle donnée, on peut souvent avoir une idée de l'allure des graphes des solutions en observant le champs de vecteurs associé. En effet le graphe d'une solution de l'équation $y'(t) = \varphi(t, y(t))$ est par définition tangent à son vecteur vitesse $(1, y'(t))$ et donc au vecteur $(1, \varphi(t, y(t))$. La connaissance de la fonction φ en chaque point (t, y) permet donc de représenter facilement ces vecteurs tangents même si l'on ne connait pas les solutions. Et si l'on en trace un grand nombre, uniformément répartis dans le plan (t, y), on obtient une représentation du champ de vecteurs associé à l'équation différentielle qui permet souvent de deviner les graphes des solutions puisqu'il s'agit des courbes qui sont tangentes en tous leurs points aux vecteurs de ce champs de vecteurs.
Il est alors intéressant d'y superposer la solution numérique obtenue avec `odeint`.

- La fonction `quiver` du module `matplt.plotlib` permet de tracer un champ de vecteurs : `quiver(x_pos, y_pos, x_dir, y_dir, color)`
- La fonction `streamplt.plot` permet de tracer des lignes de courant représentant un champ vectoriel.

Utilisons-les pour obtenir les tracés associés à notre équation différentielle. La courbe rouge est la solution déterminée par `odeint`.

Cf. https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.quiver.html et https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.streamplot.html

```
# Pour aller plus loin, décommenter un des deux appels suivants
# ==============================================================================
# help(plt.quiver)
# help(plt.streamplot)
```

13.3. Exemples

13.3.1. Exemple de résolution approchée d'une équation différentielle d'ordre 1

En guise d'exemple, considérons le problème de Cauchy $y(0) = 1$ et une équation logistique simple de la forme

$$y'(t) = \frac{3}{2} y(t) \left(1 - \frac{y(t)}{6}\right).$$

On cherche à afficher la solution approchée sur l'intervalle $[0, 5]$.
On ajoutera ensuite le champ de vecteurs associé à cette équation différentielle.

On crée alors la fonction φ

La fonction `odeint` peut être appelée avec au minimum trois arguments :

- `func` : la fonction φ,
- `y0` : la valeur $y(t_0)$,

- `t` : le vecteur (qui commence à t_0) où la fonction y sera évaluée.

Elle renvoi un vecteur `sol` contenant l'évaluation de la solution en les points du vecteur t.

Nota bene : par défaut la fonction φ doit avoir t comme deuxième variable, mais on retrouve l'ordre usuel avec le paramètre `tfirst=True`.

```
# def phi(t,y):
#     return 1.5*y*(1-y/6)
# ou

phi = lambda t,y : 1.5*y*(1-y/6) # Attention, pas de calcul symbolique ici, on utlise des Float !
```

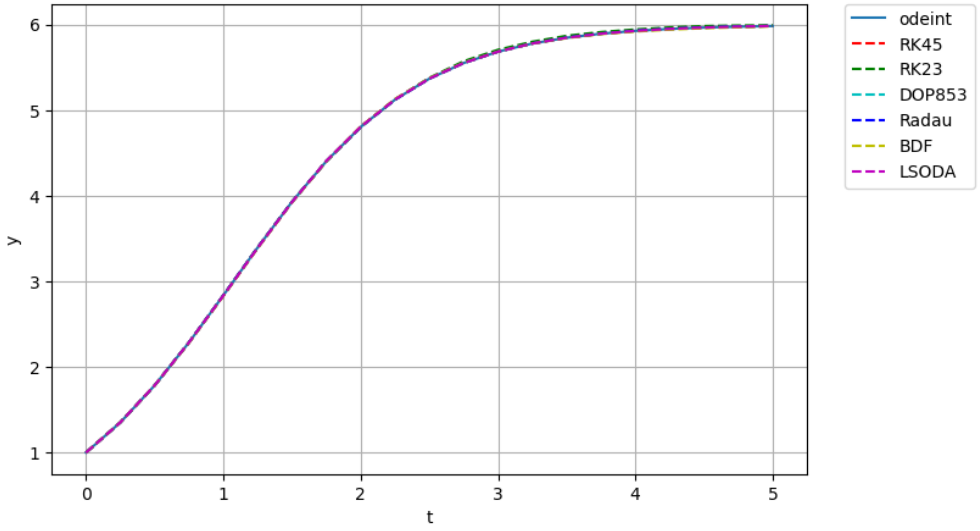
```
t0  = 0
y0  = 1
tt  = np.linspace(t0,5,21)
# sol = odeint(func=phi, y0=y0, t=tt, tfirst=True) # Appel avec arguments nommés: 4 paramètres, passés
 ↪  dans n'importe quel ordre
sol = odeint(phi, y0, tt, tfirst=True) # Appel classique: 4 paramètres, 3 passés selon l'ordre, le
 ↪  dernier nommé

# ============================================================================
# Bonus : on ajoute la comparaison avec les calculs de solve_ivp
# ============================================================================
from scipy.integrate import odeint, solve_ivp
sol_RK45   = solve_ivp(fun=phi, t_span=(t0,5), y0=[y0], t_eval=tt, method='RK45')
sol_RK23   = solve_ivp(fun=phi, t_span=(t0,5), y0=[y0], t_eval=tt, method='RK23')
sol_DOP853 = solve_ivp(fun=phi, t_span=(t0,5), y0=[y0], t_eval=tt, method='DOP853')
sol_Radau  = solve_ivp(fun=phi, t_span=(t0,5), y0=[y0], t_eval=tt, method='Radau')
sol_BDF    = solve_ivp(fun=phi, t_span=(t0,5), y0=[y0], t_eval=tt, method='BDF')
sol_LSODA  = solve_ivp(fun=phi, t_span=(t0,5), y0=[y0], t_eval=tt, method='LSODA')
```

La solution peut alors être tracée simplement

```
plt.figure(figsize=(8,5))
plt.plot(tt,sol,label='odeint')
plt.xlabel('t')
plt.ylabel('y')
plt.grid();


# ============================================================================
# Bonus
# ============================================================================
plt.plot(tt, sol_RK45.y[0],   'r--', label='RK45')
plt.plot(tt, sol_RK23.y[0],   'g--', label='RK23')
plt.plot(tt, sol_DOP853.y[0], 'c--', label='DOP853')
plt.plot(tt, sol_Radau.y[0],  'b--', label='Radau')
plt.plot(tt, sol_BDF.y[0],    'y--', label='BDF')
plt.plot(tt, sol_LSODA.y[0],  'm--', label='LSODA')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.);
```



NB : le nombre de points en lesquels les résultats sont évalués n'est pas (du moins directement) relié à la précision des calculs internes (ne pas imaginer que cela fixe le pas de la méthode, en particulier).

Superposons enfin le champ de vecteurs associé à cette équation différentielle à la solution approchée.

```
# define a plt.grid
g1  = np.linspace(0,5,21)
g2  = np.linspace(0,8,21)
T,Y = np.meshgrid(g1,g2)          # create a plt.grid

# compute direction at each point
DT, DY = 1, phi(T,Y)           # compute growth rate on the plt.grid
# M = np.sqrt(DT**2+DY**2)          # norm growth rate
M = np.hypot(DT,DY)

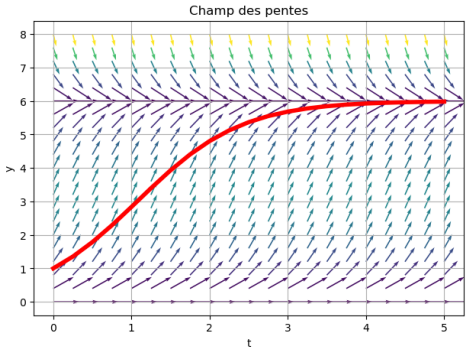

# Deux types de représentations : champ des pentes et lignes de courant
# ============================================================================

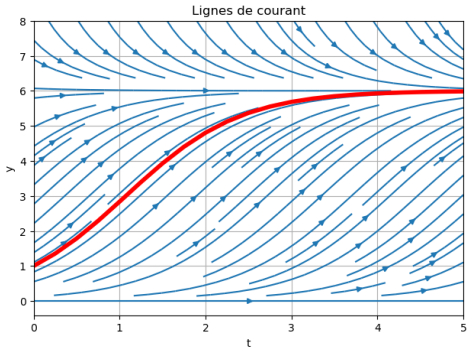
plt.figure(figsize=(16,5))

# Champ des pentes
# ============================================================================
plt.subplot(1,2,1)
# plt.quiver(T,Y, DT/M, DY/M, M, pivot='mid')
plt.quiver(T,Y, DT/M, DY/M,
           # options de personnalisation
           M, cmap = plt.cm.viridis, scale_units='xy',scale=3.
           )
plt.plot(tt,sol,'r', lw=4)
plt.grid()
plt.xlabel('t')
plt.ylabel('y')
plt.title('Champ des pentes');

# Lignes de courant
# ============================================================================
plt.subplot(1,2,2)
plt.streamplot(T,Y, DT/M, DY/M,
               # options de personnalisation
               # color=M, linewidth=1, cmap=plt.cm.viridis, density=2, arrowstyle='->', arrowsize=1.5
               )
plt.plot(tt,sol,'r', lw=4)
plt.grid()
```

```
plt.xlabel('t')
plt.ylabel('y')
plt.title('Lignes de courant');
```





13.3.2. Exemple de résolution approchée d'un système d'équations différentielles d'ordre 1

Considérons deux espèces : une proie (des lièvres par exemple) et un prédateur (des lynx par exemple). Ces deux populations sont représentées par $y_1(t)$ et $y_2(t)$ des fonctions continues du temps t. Si on suppose qu'il n'y a aucune autre intervention extérieure, une modélisation possible pour ce genre de système a été proposée indépendamment par Alfred James Lotka en 1925 et Vito Volterra en 1926 :

$$\begin{cases} y_1'(t) = y_1(t)(a - by_2(t)) & [\overset{\text{déf}}{=} \varphi_1(t, y_1(t), y_2(t))] \quad \text{équation des proies} \\ y_2'(t) = -y_2(t)(c - dy_1(t)) & [\overset{\text{déf}}{=} \varphi_2(t, y_1(t), y_2(t))] \quad \text{équation des prédateurs} \end{cases}$$

soit encore, en notation matricielle,

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix}' (t) = \begin{pmatrix} \varphi_1(t, y_1(t), y_2(t)) \\ \varphi_2(t, y_1(t), y_2(t)) \end{pmatrix}.$$

On suppose qu'à l'instant initiale $t = 0$ il y a

- $y_1(0) = 2$ unités de proies (une unités = 1000 animaux) et
- $y_2(0) = 1$ unités de prédateurs.

On se demande comment vont évoluer les populations de ces deux espèces au cours du temps. Est-ce que les deux espèces vont coexister ou bien l'une des deux va disparaître ? Y a-t-il des cycles de croissance et de décroissance des populations ?
Pour les simulations on prendra $a = 2$, $b = 1$, $c = 1$ et $d = 0.3$.

Pour une introduction à la modélisation de la dynamique de populations, voir par exemple cette page de vulgarisation Le modèle de Lotka-Volterra - Le lynx prédateur du lièvre

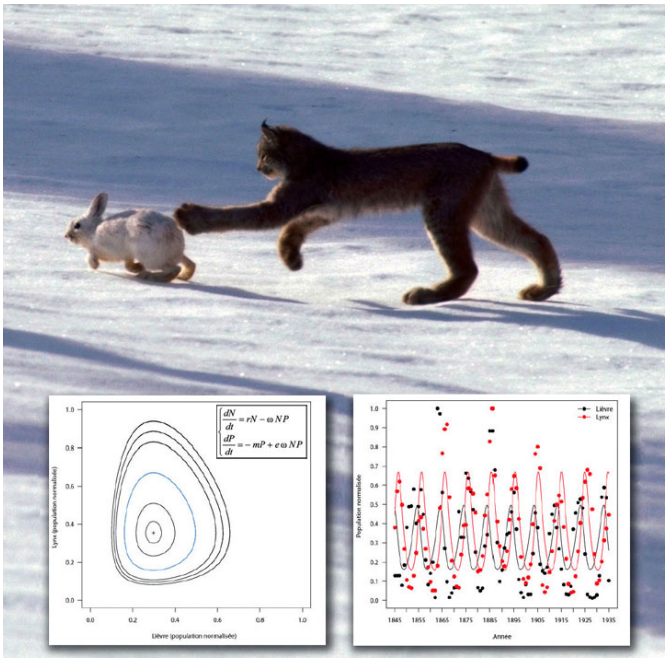


FIGURE 13.1. – Lotka Volterra

Soit $\mathbf{y}(t) \overset{\text{déf}}{=} (y_1(t), y_2(t))$ le vecteur des deux fonctions inconnues. On commence par créer la fonction vectorielle

$$\varphi(t, \mathbf{y}) = (\varphi_1(t, \mathbf{y}), \varphi_2(t, \mathbf{y}))$$

```
# yy est une liste a deux composantes
pphi = lambda t,yy : [ yy[0]*(2-yy[1]) , -yy[1]*(1-0.3*yy[0]) ]
```

En faisant varier le temps sur l'intervalle $[0; 20]$ et en prenant comme condition initiale le vecteur $\mathbf{y}_0 = (2, 1)$ on écrit

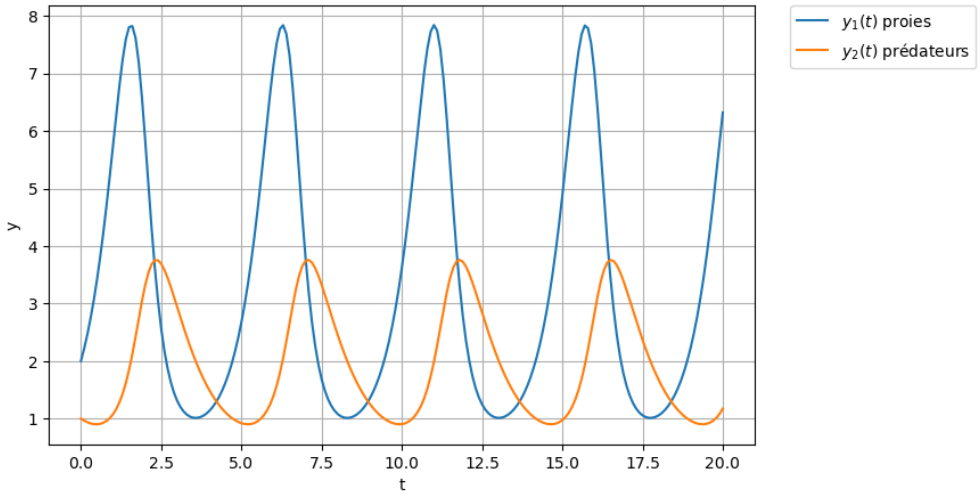
```
t0  = 0
yy0 = [2,1]
tt  = np.linspace(t0,20,201)

# V1
# odeint revoie une matrice, la colonne j contient l'approximation de la j-ème composante de la fonction
 ↪ inconnue
# sol = odeint(pphi,yy0,tt,tfirst=True)
# sol_1 = sol[:,0] # [y_1(t) for t in tt]
# sol_2 = sol[:,1] # [y_2(t) for t in tt]

# V2
# Pour que odeint renvoit séparément les valeurs de la solution, il faut rajouter .T à la fin
sol_1,sol_2 = odeint(pphi,yy0,tt,tfirst=True).T
```

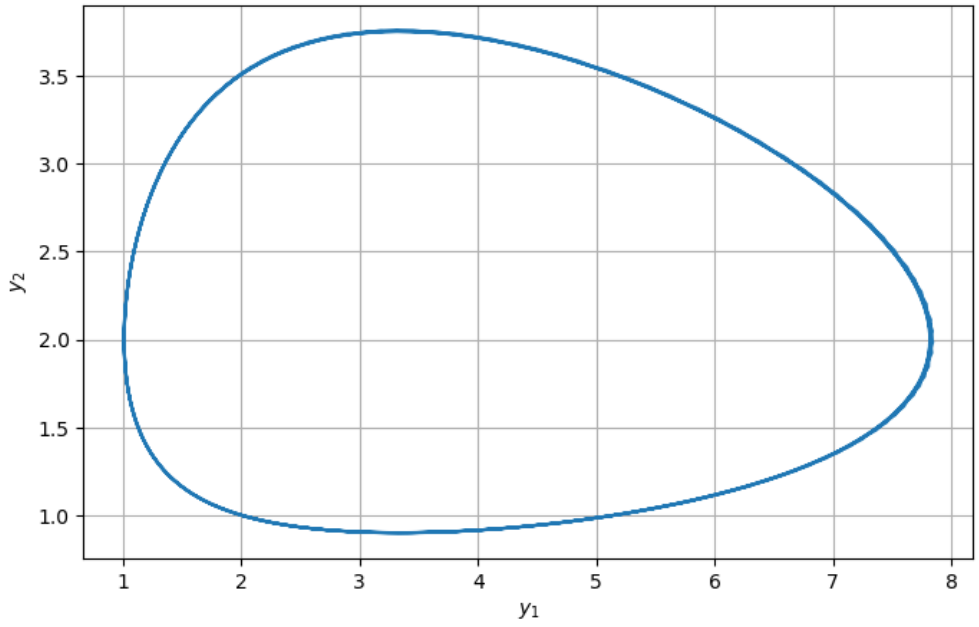
Le tracé des évolutions de y_1 et y_2 en fonction du temps t peut être obtenu par

```
plt.figure(figsize=(8,5))
plt.plot(tt,sol_1, label=r"$y_1(t)$ proies")
plt.plot(tt,sol_2, label=r"$y_2(t)$ prédateurs")
plt.grid()
plt.xlabel('t')
plt.ylabel('y')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.); # astuce pour déplacer la
 ↪ légende à l'extérieur du plot
```



Le tracé de l'évolution de y_2 en fonction de y_1 peut être obtenu par

```
plt.figure(figsize=(8,5))
plt.plot(sol_1,sol_2)
plt.grid()
plt.xlabel(r'$y_1$')
plt.ylabel(r'$y_2$');
```



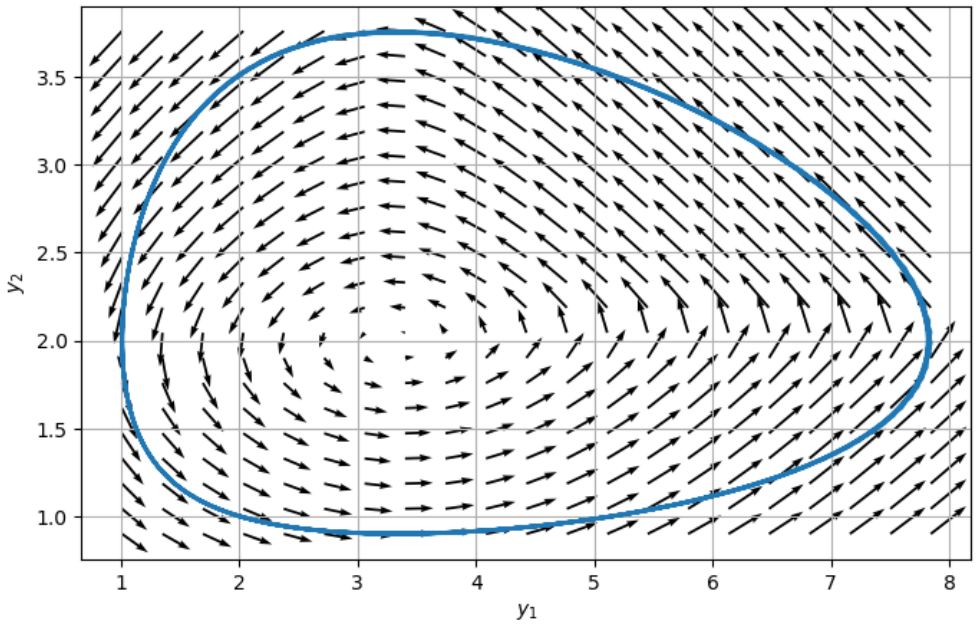
On peut superposer les deux graphes sur le plan des phases pour voir comment les solutions se comportent.

Attention, cette fois-ci les solutions sont des courbes paramétrées par t et non des fonctions de t.

```
Y1,Y2 = np.meshgrid(np.linspace(min(sol_1),max(sol_1),21),np.linspace(min(sol_2),max(sol_2),21))
V1,V2 = pphi(tt,[Y1,Y2])
r1 = np.sqrt(1+V1**2)
r2 = np.sqrt(1+V2**2)

plt.figure(figsize=(8,5))
plt.quiver(Y1, Y2, V1/r1, V2/r2, cmap=plt.cm.viridis, scale_units='xy',scale=4.)
plt.plot(sol_1,sol_2,lw=2)
plt.grid()
```

```
plt.xlabel(r'$y_1$')
plt.ylabel(r'$y_2$');
```



Considérons la fonction

$$E(t) = dy_1(t) + by_2(t) - c\ln(y_1(t)) - a\ln(y_2(t))$$

Vérifions analytiquement que la fonction E est une intégrale première du système, c'est-à-dire que si (y_1, y_2) est une solution alors l'application E est constante :

$$\begin{aligned}
E'(t) &= dy_1'(t) + by_2'(t) - \frac{c}{y_1(t)}y_1'(t) - \frac{a}{y_2(t)}y_2'(t) \\
&= \left(d - \frac{c}{y_1(t)}\right)y_1'(t) + \left(b - \frac{a}{y_2(t)}\right)y_2'(t) \\
&= \left(d - \frac{c}{y_1(t)}\right)y_1(t)(a - by_2(t)) - \left(b - \frac{a}{y_2(t)}\right)y_2(t)(c - dy_1(t)) \\
&= (dy_1(t) - c)(a - by_2(t)) - (by_2(t) - a)(c - dy_1(t)) \\
&= (dy_1(t) - c)(a - by_2(t)) - (a - by_2(t))(dy_1(t) - c) = 0.
\end{aligned}$$

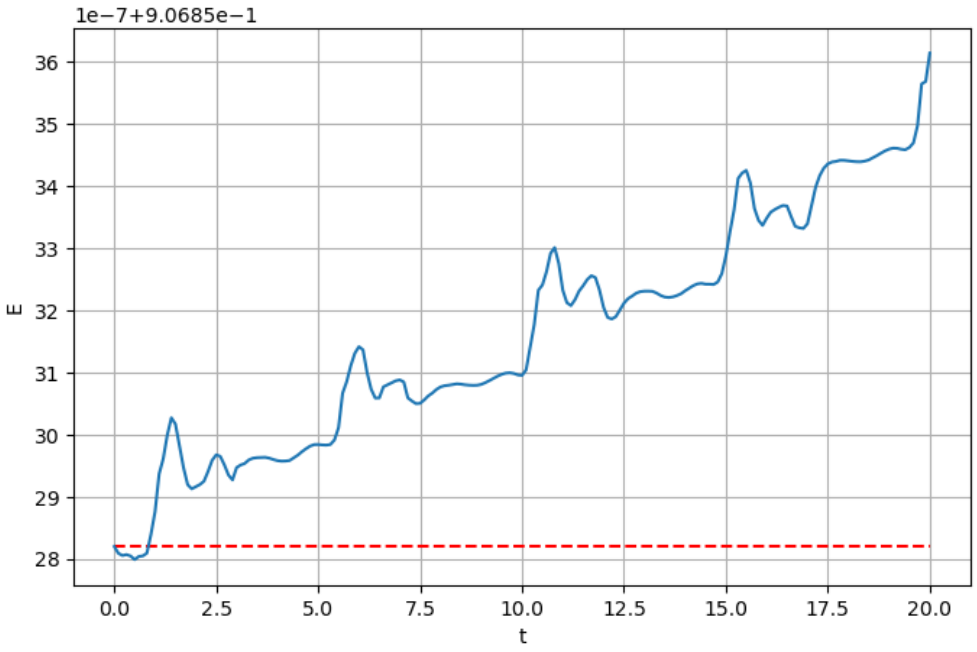
Vérifions cette propriété numériquement :

```
E = lambda yy : 0.3*yy[0]+yy[1]-np.log(yy[0])-2*np.log(yy[1])

plt.figure(figsize=(8,5))
E0 = E(yy0)
EE  = E([sol_1,sol_2])


print(max(abs(EE-E0)))
plt.hlines(E0,tt[0],tt[-1],colors='r',linestyles='dashed') # idem que
 ↪  plt.plot([tt[0],tt[-1]],[E0,E0],colors='r',linestyles='dashed') # valeur pour t=0
plt.plot(tt,EE)
plt.xlabel('t')
plt.ylabel('E')
plt.grid();
```

```
7.942278004158254e-07
```



13.3.3. Exemple de résolution approchée d'une équation différentielle d'ordre 2

Considérons l'EDO $y''(t) = -\sin(y(t))$ qui décrit le mouvement d'un pendule non amorti, équivalente au système différentiel

$$\begin{cases} y_1'(t) = y_2(t), \\ y_2'(t) = -\sin(y_1(t)) \end{cases}$$

avec $y(0) = 0$ et $y'(0) = 1$.
On cherche à approcher la solution sur l'intervalle $[0, 10]$.

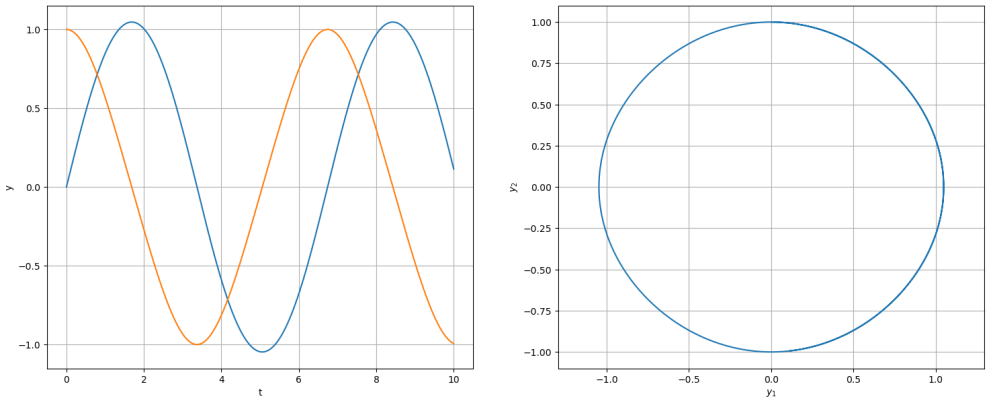
```
pphi = lambda t,yy : [ yy[1], -np.sin(yy[0])  ]

t0  = 0
yy0  = [0,1]
tt  = np.linspace(t0,10,1001)
sol_1, sol_2 = odeint(pphi,yy0,tt,tfirst=True).T

plt.figure(figsize=(18,7))

plt.subplot(1,2,1)
plt.plot(tt,sol_1,tt,sol_2)
plt.xlabel('t')
plt.ylabel('y')
plt.grid()

plt.subplot(1,2,2)
plt.plot(sol_1,sol_2)
plt.axis('square')
plt.xlabel(r'$y_1$')
plt.ylabel(r'$y_2$')
plt.grid()
plt.axis('equal');
```



Considérons l'énergie suivante (somme de l'énergie potentielle et de l'énergie cinétique) :

$$E(t) = -\cos(y_1(t)) + \frac{1}{2}(y_2(t))^2$$

Vérifions analytiquement que la fonction E est conservée au cours du temps :

$$\begin{aligned} E'(t) &= \sin(y_1(t))y_1'(t) + y_2(t)y_2'(t) \\ &= -y_2'(t)y_1'(t) + y_1'(t)y_2'(t) = 0. \end{aligned}$$

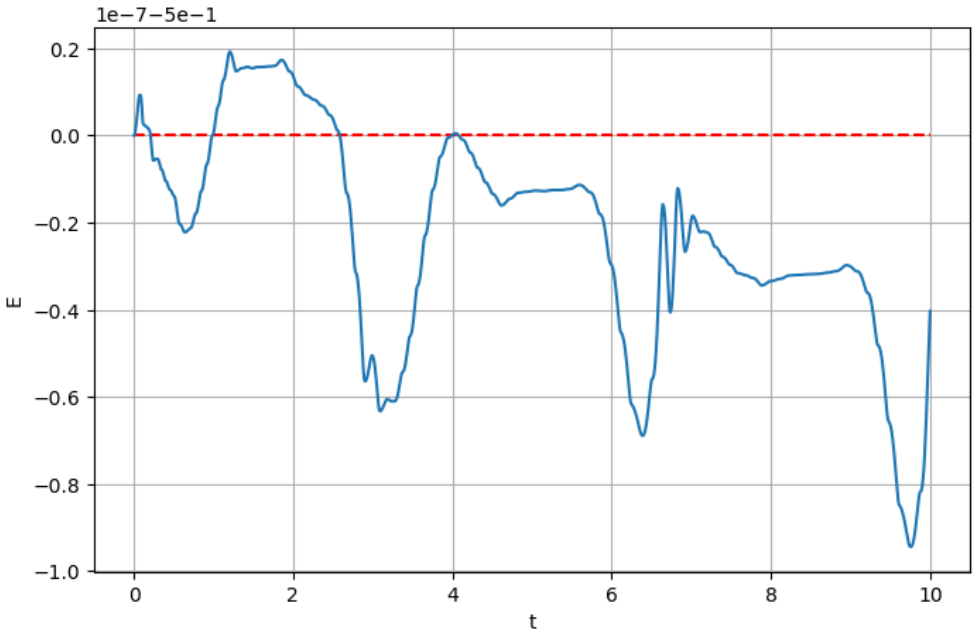
Vérifions cette propriété numériquement :

```
E = lambda yy : -np.cos(yy[0])+yy[1]**2/2

plt.figure(figsize=(8,5))
E0 = E(yy0)
EE  = E([sol_1,sol_2])

print(max(abs(EE-E0)))
plt.hlines(E0,tt[0],tt[-1],colors='r',linestyles='dashed') # idem que
 ↪  plt.plot([tt[0],tt[-1]],[E0,E0],colors='r',linestyles='dashed') # valeur pour t=0
plt.plot(tt,EE)
plt.xlabel('t')
plt.ylabel('E')
plt.grid();
```

```
9.448590565508397e-08
```



13.4. Exercices

13.4.1. Exercice (EDO d'ordre 1)

Pour $a = -2, -1, -\frac{1}{2}, -\frac{1}{10}, 0, \frac{1}{10}, \frac{1}{2}, 1, 2$, afficher la solution **approchée** et le champ de vecteurs associé pour le problème de Cauchy

$$\begin{cases} y'(t) = \sin\left(ty(t)\right), & t > 0 \\ y(0) = a \end{cases}$$

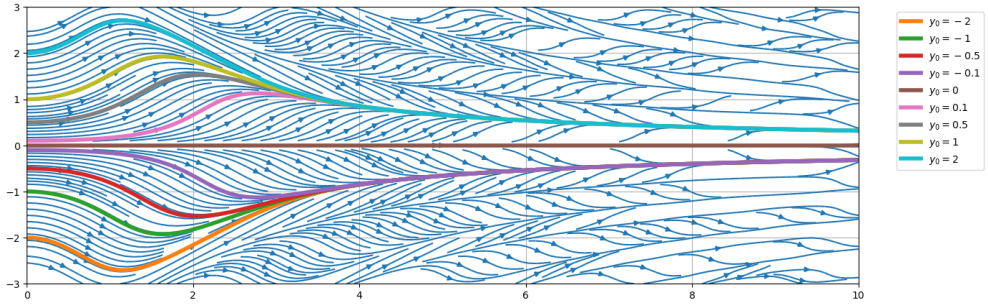
Bonus : savez-vous calculer la solution exacte en fonction de a ? Qu'en pense sympy ?

```
phi = lambda t,y : np.sin(t*y)
t_final = 10
tt  = np.linspace(0,t_final,201)

plt.figure(figsize=(15,5))

# Champ des pentes / Lignes de courant
# À décommenter pour voir le champ des pentes au-dessus des solutions approchées
# ================================================================================
T,Y = np.meshgrid(np.linspace(0,t_final,61),np.linspace(-3,3,31))
DT, DY = np.ones_like(T), phi(T,Y)
M = np.sqrt(DT**2+DY**2)   # norm growth rate
# plt.quiver(T,Y, DT/M, DY/M, M, cmap=plt.cm.viridis, scale_units='xy',scale=4.)
plt.streamplot(T,Y, DT, DY,density=3)

plt.grid()
for y0 in [-2, -1, -0.5, -0.1, 0, 0.1, 0.5, 1, 2]:
    sol = odeint(phi, y0, tt, tfirst=True)
    plt.plot(tt, sol, lw=4, label=f'$y_0={y0}$');
plt.legend(bbox_to_anchor=(1.04,1),loc='upper left'); # Astuce pour déplacer la légende à l'extérieur du
 ↪ plot
```




```
# Affichage des deux graphes côte à côte
# ======================================

# T,Y = np.meshgrid(np.linspace(0,t_final,31),np.linspace(-3,3,31))
# DT, DY = np.ones_like(T), phi(T,Y)
# M = np.sqrt(DT**2+DY**2)   # norm growth rate

# fig, aax = plt.subplots(1,2,figsize=(15,5))


# aax[0].quiver(T,Y, DT/M, DY/M, M, pivot='mid')
# aax[0].grid()

# aax[1].streamplot(T,Y, DT/M, DY/M, color=M,density=2)
# aax[1].grid()

# for y0 in [-2, -1, -0.5, -0.1, 0, 0.1, 0.5, 1, 2]:
#     sol = odeint(phi,y0,tt,tfirst=True)
#     for ax in aax:
#         ax.plot(tt,sol,lw=4,label=f'$y_0={y0}$');
# plt.legend(bbox_to_anchor=(1.04,1),loc='upper left');
```

Malheureusement sympy ne peut pas nous fournir une expression analytique de la solution exacte en fonction de a.

```
# import sympy as sp

# # Variables
# A = sp.symbols('A')
# t = sp.symbols('t', positive=True)
# y = sp.Function('y')

# # EDO
# edo = sp.Eq( sp.diff( y(t),t) , sp.sin(t*y(t)))
# display(edo)

# # Condition initiale
# t0 = 0
# y0 = A
# display(sp.Eq(y(t0), y0))

# # Méthode 1 : méthode directe
# # ===========================
# solpar = sp.dsolve(edo,y(t), ics={y(t0):y0})
# display(solpar)
```

13.4.2. Exercice (système)

Calculer la solution particulière **approchée** du système

$$\begin{cases} x'(t) = -y(t) + y^2(t)x(t) \\ y'(t) = x(t) - x^2(t)y(t) \end{cases}$$

avec $x(0) = 0$ et $y(0) = 1$.

1. Afficher $t \mapsto x$ et $t \mapsto y$ dans un même graphe pour $t \in [0, 14]$.

2. Afficher dans un autre répère $x \mapsto y$.
3. Soit une fonction $t \mapsto E(t)$. Calculer analytiquement $E'(t)$. Afficher ensuite $E(t)$ calculé à partir des solutions approchées. La fonction E est définie par

$$t \mapsto E(t) = \frac{x^2(t)}{2} + \frac{y^2(t)}{2}.$$

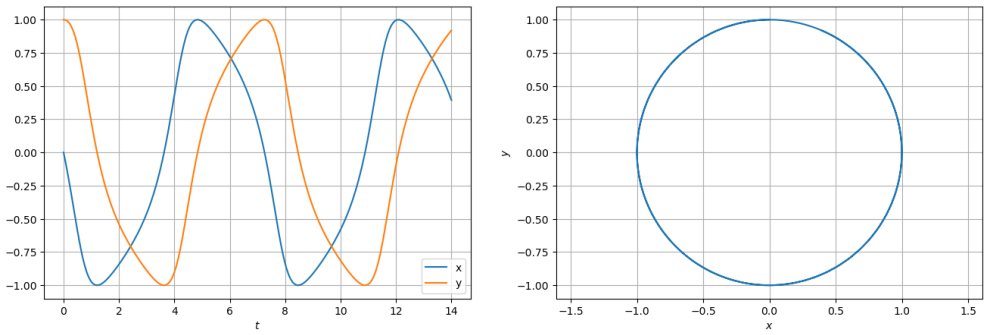
```
# phi : R x R^2 -> R^2
pphi = lambda t,yy : [ -yy[1]+(yy[1])**2*yy[0] , yy[0]-(yy[0])**2*yy[1] ]

yy0  = [0,1]
tt   = np.linspace(0, 14, 201)
solx, soly = odeint(pphi, yy0, tt, tfirst=True).T

plt.figure(figsize=(16,5))

plt.subplot(1,2,1)
plt.plot(tt,solx,tt,soly)
plt.grid()
plt.xlabel(r'$t$')
plt.legend(["x","y"]);

plt.subplot(1,2,2)
plt.plot(solx,soly)
plt.grid()
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
plt.axis('equal');
```



On calcule la dérivée de E (analytiquement à la main ou, si on veut, avec sympy) pour vérifier que c'est une intégrale première du système :

$$E'(t) = x'(t)x(t) + y'(t)y(t) = \left(-y(t) + y^2(t)x(t)\right)x(t) + \left(x(t) - x^2(t)y(t)\right)y(t) = 0.$$

On vérifie ensuite numériquement que E est conservée (à une précision près) au cours du temps. Pour cela, on calcule la valeur exacte de E en t_0, puis on trace l'évolution de E en fonction du temps. On constate que E est conservée à une 10^{-7} près.

```
# Formule qui définie l'énergie
E = lambda yy : (yy[0]**2+yy[1]**2)/2
```

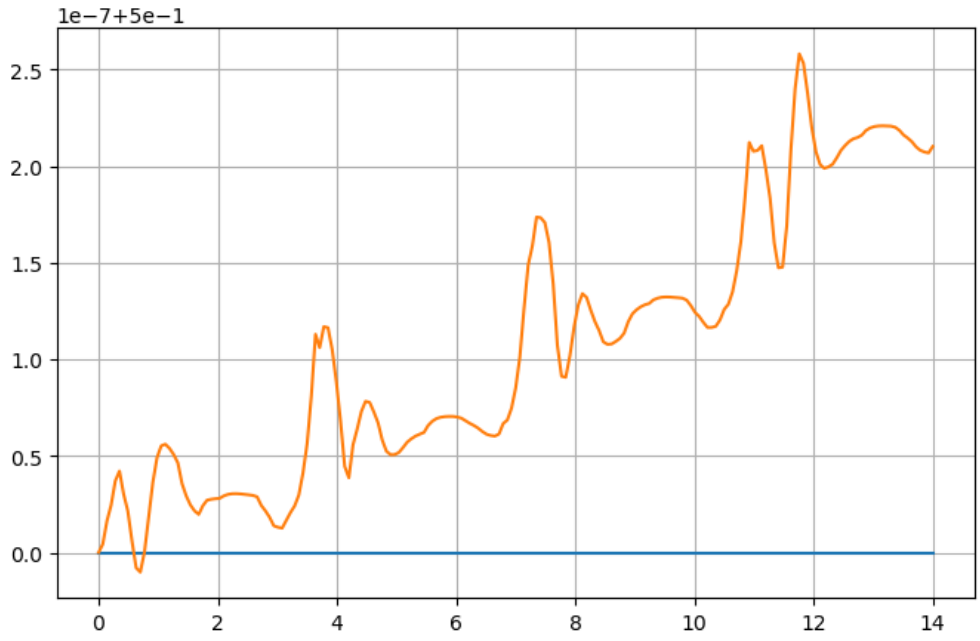
```
# E initial exact
E0 = E(yy0)

# On evalue l'énergie en chaque point de la discrétisation en utlisant la solution approchée
EE = E([solx,soly])

# On affiche le plus grand écart entre l'énergie initiale et l'énergie évaluée en chaque point
# Si le calcul était exact, on devrait avoir max(abs(EE-E0)) = 0, norme infinie de l'erreur absolue
print(f"{min(EE-E0) = :e}, {max(EE-E0) = :e}")

# Affichage
# ==============================================================================
plt.figure(figsize=(8,5))
# On affiche une droite qui correspond à l'énergie initiale
plt.plot([tt[0],tt[-1]],[E0,E0]) # pour tracer le segment reliant (x0,y0) et (x1,y1) on peut écrire
 ↪  plt.plot([x0,x1],[y0,y1])
# On affiche l'énergie évaluée en chaque point de la discrétisation
plt.plot(tt,EE)
plt.grid();
```

```
min(EE-E0) = -1.018167e-08, max(EE-E0) = 2.582059e-07
```



13.4.3. Exercice (quiver - système)

Considérons le système

$$\begin{cases} x'(t) = x(t)(y(t) - x(t)^2) \\ y'(t) = y(t)(x(t) - y(t)^2) \end{cases}$$

Afficher le champ des vecteurs associé à ce système. Que remarquez-vous ?

```
pphi = lambda t,yy : [ yy[0]*(yy[1]-yy[0]**2) , yy[1]*(yy[0]-yy[1]**2) ]


# Preparation de la grille
# ==============================================================================
t0  = 0
```

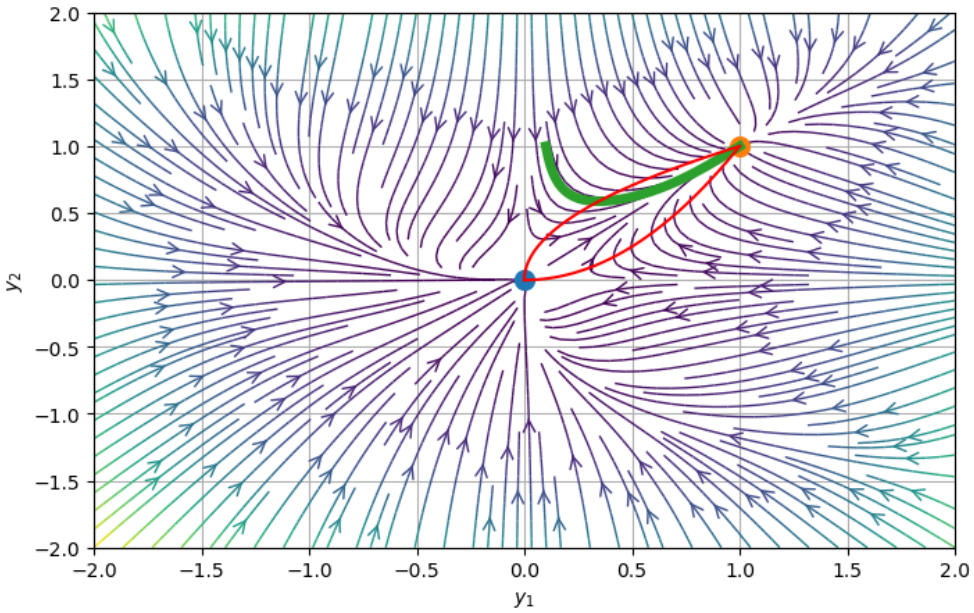
```
tt  = np.linspace(t0,20,201)
Y1,Y2 = np.meshgrid(np.linspace(-2,2,51),np.linspace(-2,2,51))
V1,V2 = pphi(tt,[Y1,Y2])
M = np.hypot(V1,V2)
V1 = V1/np.sqrt(V1**2+V2**2+1)
V2 = V2/np.sqrt(V1**2+V2**2+1)

# Affichage
# ============================================================================
plt.figure(figsize=(8,5))
# plt.quiver(Y1, Y2, V1, V2,
#         #    M, cmap = plt.cm.viridis, scale_units='xy',scale=10.
#         )
plt.streamplot(Y1, Y2, V1, V2,
               # options de personnalisation
               color=M, linewidth=1, cmap=plt.cm.viridis, density=2, arrowstyle='->', arrowsize=1.5
               )
plt.grid()
plt.xlabel(r'$y_1$')
plt.ylabel(r'$y_2$')

# Points stationnaires
# ============================================================================
plt.plot(0,0,'o',ms=10)
plt.plot(1,1,'o',ms=10);

# Une solution approchée
# ============================================================================
yy0 = [ 0.1,1]
# yy0 = [-0.1,1]
sol_1,sol_2 = odeint(pphi,yy0,tt,tfirst=True).T
plt.plot(sol_1,sol_2,lw=5)

# Courbes
# ============================================================================
x_tmp = np.linspace(0,1,101)
plt.plot(x_tmp,x_tmp**2, 'r')
plt.plot(x_tmp**2,x_tmp, 'r');
```



13.4.4. Exercice (système) : étude de la stabilité

Considérons le système

$$\begin{cases} x'(t) = x^2(t) + y^2(t) - 25 \\ y'(t) = x(t)y(t) - 12 \end{cases}$$

1. Afficher le champ des vecteurs et les solutions stationnaires.
2. Pour chaque point stationnaire, considérer 4 données initiales proches de ce point et calculer la solution particulière **approchée** associée à telle donnée. Que peut-on déduire sur la stabilité de ces points ?

L'expression du second membre est une fonction polynômiale en x, y, le théormème de Cauchy-Lipschitz s'applique donc et il existe une unique solution maximale pour chaque donnée de Cauchy.

Pour les données de Cauchy telles que

$$\begin{cases} x^2 + y^2 = 25 \\ xy = 12 \end{cases}$$

la solution est globale et stationnaire. On cherche les couples (x, y) qui satisfont ce système.

- On peut par exemple retravailler le système comme suit :

$$\begin{cases} x^2 + y^2 = 25 \\ xy = 12 \end{cases} \rightsquigarrow \begin{cases} (x+y)^2 - 2xy = 25 \\ xy = 12 \end{cases} \rightsquigarrow \begin{cases} (x+y)^2 = 49 \\ xy = 12 \end{cases} \rightsquigarrow \begin{cases} x + y = \pm 7 \\ xy = 12 \end{cases} \rightsquigarrow \begin{cases} x + y = 7 \\ xy = 12 \end{cases} \rightsquigarrow (3,4), (4,3)$$
$$\begin{cases} x + y = -7 \\ xy = 12 \end{cases} \rightsquigarrow (-3,-4), (-4,$$

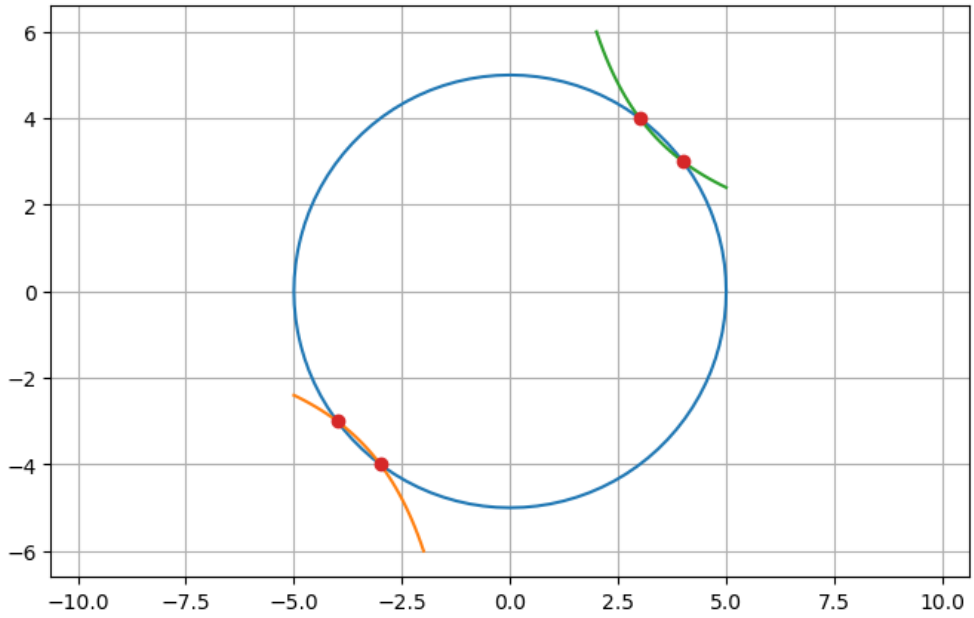
- soit résoludre graphiquemt en affichant les deux courbes du système. De telles données de Cauchy sont l'intersection du cercle centré en 0 et de rayon 5 et de l'hyperbole d'équation $y = 12/x$. On obtient alors 4 points : $(3, 4)$, $(4, 3)$, $(-3, -4)$ et $(-4, -3)$.

```
plt.figure(figsize=(8,5))

# x^2+y^2=25 circle de centre (0,0) et rayon 5 donc d'équation x(t)=5cos(t) et y(t)=5sin(t)
t = np.linspace(0,2*np.pi,101)
plt.plot(5*np.cos(t),5*np.sin(t))

# pour afficher xy=12 on doit sauter 0 donc on utilise deux listes separées
t = np.linspace(-5, -2, 101)
plt.plot(t,12/t);
t = np.linspace(2,5,101)
plt.plot(t,12/t)

plt.grid()
plt.plot([3,4,-3,-4],[4,3,-4,-3],'o')
plt.axis('equal');
```



- soit résoudre

$$\begin{cases} x^2+y^2=25 \\ xy=12 \end{cases} \overset{x\neq 0}{\rightsquigarrow} \begin{cases} x^2+y^2=25 \\ y=\frac{12}{x} \end{cases} \rightsquigarrow \begin{cases} x^2+\frac{12^2}{x^2}=25 \\ y=\frac{12}{x} \end{cases} \rightsquigarrow \begin{cases} x^4-25x^2+12^2=0 \\ y=\frac{12}{x} \end{cases}$$

On peut ainsi calculer les valeurs de x qui satisfont la première équation et en déduire les valeurs de y associées.

```
# numpy calcule les racines d'un polynome
x_sol = np.roots([1, 0, -25, 0, 12**2])
y_sol = 12/x_sol
print(x_sol,y_sol)
```

```
[-4. -3.  4.  3.] [-3. -4.  3.  4.]
```

En conclusion, on a 4 points stationnaires :

$$\{(3,4),(-4,-3),(4,3),(-3,-4)\}$$

```
pphi = lambda t,y : [ (y[0])**2+(y[1])**2-25 , y[0]*y[1]-12 ]

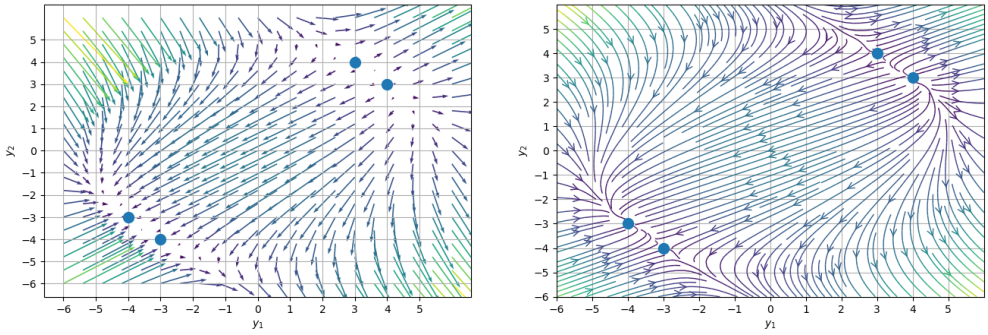
Y1,Y2 = np.meshgrid(np.linspace(-6,6,21),np.linspace(-6,6,21))
tt    = np.linspace(0,0.2,201)
V1,V2 = pphi(tt,[Y1,Y2])
r = np.sqrt(V1**2+V2**2)
# r[ r==0 ] = 1

plt.figure(figsize=(16,5))

plt.subplot(1,2,1)
# plt.quiver(Y1, Y2, V1/r, V2/r, r, cmap=plt.cm.viridis)
plt.quiver(Y1, Y2, V1, V2, r, cmap=plt.cm.viridis, scale_units='xy',scale=20.)
plt.plot(x_sol,y_sol,'o',ms=10)
plt.xticks(range(-6,6))
plt.yticks(range(-6,6))
plt.grid()
plt.xlabel(r'$y_1$')
plt.ylabel(r'$y_2$');

plt.subplot(1,2,2)
```

```
# plt.streamplot(Y1, Y2, V1/r, V2/r)
plt.streamplot(Y1, Y2, V1/r, V2/r, color=r, linewidth=1, cmap=plt.cm.viridis, density=2, arrowstyle='->',
 ↪  arrowsize=1.5)
plt.plot(x_sol,y_sol,'o',ms=10)
plt.xticks(range(-6,6))
plt.yticks(range(-6,6))
plt.grid()
plt.xlabel(r'$y_1$')
plt.ylabel(r'$y_2$');
```



```
plt.figure(figsize=(16,10))

plt.subplot(2,2,1)
plt.title('Voisinage de (3,4)')
plt.plot(3,4,'rD',ms=20)
plt.grid()
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
eps = 1
for y0,tfin in [ ((3+0.1,4+0.1),0.2), ((3-0.1,4+0.1),0.5) , ((3+0.1,4-0.1),0.4), ((3-0.1,4-0.1),0.2)]:
    tt  = np.linspace(0,tfin,201)
    solx1, soly1 = odeint(pphi,y0,tt,tfirst=True).T
    plt.plot(solx1,soly1,"r-",lw=2)
    plt.plot(solx1[0],soly1[0],'ro',ms=10) # point de départ proche de  (3,4)
    Y1,Y2 = np.meshgrid(np.linspace(3-eps,3+eps,11),np.linspace(4-eps,4+eps,11))
    V1,V2 = pphi(tt,[Y1,Y2])
    plt.streamplot(Y1, Y2, V1, V2, color='b',linewidth=0.1)

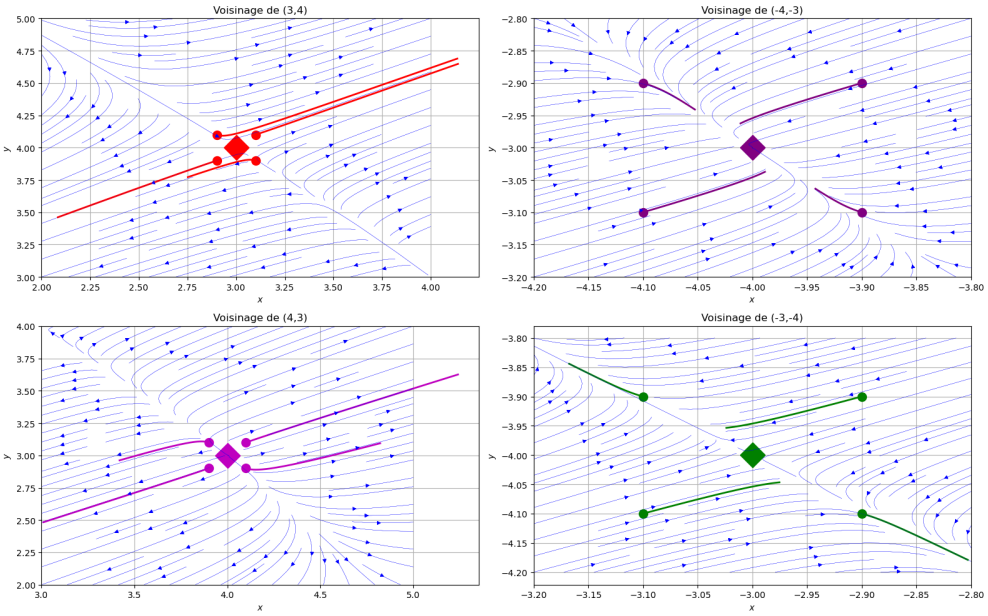

plt.subplot(2,2,2)
plt.title('Voisinage de (-4,-3)')
plt.plot(-4,-3,'D',c='purple',ms=20)
plt.grid()
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
eps = 0.2
for y0,tfin in [ ((-4+0.1,-3+0.1),0.2), ((-4-0.1,-3+0.1),0.4) , ((-4+0.1,-3-0.1),0.4),
 ↪  ((-4-0.1,-3-0.1),0.2)]:
    tt  = np.linspace(0,tfin,201)
    solx1,soly1 = odeint(pphi,y0,tt,tfirst=True).T
    plt.plot(solx1,soly1,"-",c='purple',lw=2)
    plt.plot(solx1[0],soly1[0],'o',c='purple',ms=10) # point de départ proche de  (-4,-3)
    Y1,Y2 = np.meshgrid(np.linspace(-4-eps,-4+eps,11),np.linspace(-3-eps,-3+eps,11))
    V1,V2 = pphi(tt,[Y1,Y2])
    plt.streamplot(Y1, Y2, V1, V2, color='b',linewidth=0.1)


plt.subplot(2,2,3)
plt.title('Voisinage de (4,3)')
plt.plot(4,3,'mD',ms=20)
```

```
plt.grid()
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
eps = 1
for y0,tfin in [ ((4+0.1,3+0.1),0.2), ((4-0.1,3+0.1),0.4) , ((4+0.1,3-0.1),0.4), ((4-0.1,3-0.1),0.2)]:
    tt  = np.linspace(0,tfin,201)
    solx1, soly1 = odeint(pphi,y0,tt,tfirst=True).T
    plt.plot(solx1,soly1,"m-",lw=2)
    plt.plot(solx1[0],soly1[0],'mo',ms=10) # point de départ proche de   (4,3)
    Y1,Y2 = np.meshgrid(np.linspace(4-eps,4+eps,11),np.linspace(3-eps,3+eps,11))
    V1,V2 = pphi(tt,[Y1,Y2])
    plt.streamplot(Y1, Y2, V1, V2, color='b',linewidth=0.1)


plt.subplot(2,2,4)
plt.title('Voisinage de (-3,-4)')
plt.plot(-3,-4,'gD',ms=20)
plt.grid()
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
eps = 0.2
for y0,tfin in [ ((-3+0.1,-4+0.1),0.2), ((-3-0.1,-4+0.1),0.4) , ((-3+0.1,-4-0.1),0.4),
 ↪  ((-3-0.1,-4-0.1),0.2)]:
    tt  = np.linspace(0,tfin,201)
    solx1, soly1 = odeint(pphi,y0,tt,tfirst=True).T
    plt.plot(solx1,soly1,"g-",lw=2)
    plt.plot(solx1[0],soly1[0],'go',ms=10); # point de départ proche de (-3,-4)
    Y1,Y2 = np.meshgrid(np.linspace(-3-eps,-3+eps,11),np.linspace(-4-eps,-4+eps,11))
    V1,V2 = pphi(tt,[Y1,Y2])
    plt.streamplot(Y1, Y2, V1, V2, color='b',linewidth=0.1)

plt.tight_layout()
```



Chapitre 14.

TP 2 - Implémentation des schémas “classiques” et étude de la convergence

Plan de la séance de TP

Cette séance de TP comporte 4 exercices :

- **Exercice 1** [45 minutes] : Implémentation et test des schémas d'Euler explicite et implicite sur un exemple avec solution exacte. Votre travail consiste à ajouter les autres schémas classiques.
- **Exercice 2** [30 minutes] : Étude de la convergence empirique des schémas d'Euler explicite et implicite sur le même exemple. Votre travail consiste à ajouter l'étude de la convergence des autres schémas implémentés précédemment.
- **Exercice 3** [30 minutes] : Application de la méthode d'Euler explicite pour approcher une intégrale elliptique (aucune solution exacte n'est disponible dans ce cas).
- **Exercice 4** [1h15] : Implémentation des schémas classiques appliqués à un système de deux EDO d'ordre 1. Bien que la solution exacte soit disponible pour ce problème, l'accent sera mis sur la conservation d'un invariant, souvent lié à un principe de conservation physique tel que l'énergie totale du système.

```
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import fsolve # pour les schémas implicites
from scipy.integrate import odeint, solve_ivp # pour les schémas déja implémentés dans scipy
from IPython.display import display, Markdown
plt.rcParams.update({'font.size': 12}) # plt.rcParams.update(rcParamsDefault)
```

14.1. Exercice (à compléter) : Implémentation des Schémas Classiques

Considérons le problème de Cauchy :

$$\begin{cases} y'(t) = -4y(t) + t^2, & \forall t \in I = [0, 1], \\ y(0) = 1. \end{cases}$$

Ci-dessous, nous avons calculé la solution exacte en utilisant le module `sympy`. Ensuite, nous avons implémenté les schémas d'**Euler explicite (EE)** et d'**Euler implicite (EI)**.
Nous avons comparé la solution exacte avec les solutions approchées obtenues avec ces deux schémas en utilisant un pas de $h = \frac{1}{N}$ avec $N = 8$.
Cette grille grossière permet de bien visualiser les erreurs.

Travail demandé Ajouter l'implémentation des méthodes suivantes :

- **Euler modifié**
- **RK1_M**
- **Crank-Nicolson**
- **Heun**

Les endroits où il faut écrire du code sont indiqués par les commentaires commençant par : `# !!!!!!`

Calculons la solution exacte en utilisant le module `sympy` :

```
# Variables
t = sp.Symbol('t')
y = sp.Function('y')

# Equation différentielle
edo = sp.Eq( sp.diff(y(t),t) , -4*y(t)+t**2 )
display(edo)

# Conditions initiales
t0 = 0
y0 = 1
display(sp.Eq(y(t0),y0))

# Méthode directe
solpar = sp.dsolve(edo, y(t), ics={y(t0):y0})
display(solpar)
```

$$\frac{d}{dt}y(t) = t^2 - 4y(t)$$

$$y(0) = 1$$

$$y(t) = \frac{t^2}{4} - \frac{t}{8} + \frac{1}{32} + \frac{31e^{-4t}}{32}$$

On définit la solution exacte à utiliser pour estimer les erreurs :

```
sol_exacte = sp.lambdify(t,solpar.rhs,'numpy')
```

On commence par importer les modules `numpy` et `matplotlib` et la fonction `fsolve` du module `scipy.optimize` pour résoudre l'équation implicite présente dans le schéma implicite :

On initialise le problème de Cauchy

```
# variables globales
t0     = 0
tfinal = 1
y0     = 1
```

On définit l'équation différentielle : `phi` est une lambda function qui contient la fonction mathématique $\varphi(t, y) = -4y + t^2$ dépendant des variables t et y.

```
phi = lambda t,y : -4*y+t**2
```

On écrit les schémas numériques : les valeurs $[u_0, u_1, \dots, u_N]$ pour chaque méthode sont contenues dans le vecteur uu.

Schéma d'Euler explicite :

$$\begin{cases} u_0 = y_0, \\ u_{n+1} = u_n + h\varphi(t_n, u_n) \quad n = 0, 1, 2, \dots N-1 \end{cases}$$

```
def EE(phi,tt,y0):
    h  = tt[1]-tt[0]
    uu = np.empty_like(tt)
    N = len(tt)-1
    uu[0] = y0
    for n in range(N):
        uu[n+1] = uu[n]+h*phi(tt[n],uu[n])
    return uu
```

```
# !!!!!! Ajouter ici le schéma d'Euler modifié : en markdown ET en python
```

Schéma d'Euler modifié :

$$\begin{cases} u_0 = y_0, \\ \tilde{u} = u_n + \frac{h}{2}\varphi(t_n, u_n), \\ u_{n+1} = u_n + h\varphi\left(t_n + \frac{h}{2}, \tilde{u}\right) \end{cases} \quad n = 0, 1, 2, \dots N-1$$

qu'on peut réécrire sous la forme

$$\begin{cases} u_0 = y_0, \\ k_1 = \varphi(t_n, u_n), \\ u_{n+1} = u_n + h\varphi\left(t_n + \frac{h}{2}, u_n + h\frac{k_1}{2}\right) \end{cases} \quad n = 0, 1, 2, \dots N-1$$

```
def EM(phi,tt,y0):
    h  = tt[1]-tt[0]
    uu = np.empty_like(tt)
    N = len(tt)-1
    uu[0] = y0
    for n in range(N):
        u_tilde = uu[n]+0.5*h*phi(tt[n],uu[n])
        uu[n+1] = uu[n]+h*phi(tt[n]+h/2,u_tilde)
    return uu
```

```
# !!!!!! Ajouter ici le schéma de Heun : en markdown ET en python
```

Schéma de Heun :

$$\begin{cases} u_0 = y_0, \\ \tilde{u} = u_n + h\varphi(t_n, u_n) \\ u_{n+1} = u_n + \frac{h}{2}\Big(\varphi(t_n, u_n) + \varphi(t_{n+1}, \tilde{u})\Big) \end{cases} \quad n = 0, 1, 2, \dots N-1$$

qu'on peut réécrire sous la forme

$$\begin{cases} u_0 = y_0, \\ k_1 = \varphi(t_n, u_n), \\ k_2 = \varphi(t_{n+1}, u_n + hk_1), \\ u_{n+1} = u_n + h\frac{k_1+k_2}{2} \end{cases} \quad n = 0, 1, 2, \dots N-1$$

```
def heun(phi,tt,y0):
    h  = tt[1]-tt[0]
    uu = np.empty_like(tt)
    N = len(tt)-1
    uu[0] = y0
    for n in range(N):
        k0 = phi( tt[n], uu[n] )
        k1 = phi( tt[n+1], uu[n] + h*k0  )
        uu[n+1] = uu[n] + 0.5*h*(k0+k1)
    return uu
```

Schéma d’Euler implicite :

$$\begin{cases} u_0 = y_0, \\ u_{n+1} = u_n + h\varphi(t_{n+1}, u_{n+1}) \quad n = 0, 1, 2, \dots N-1 \end{cases}$$

Attention : u_{n+1} est solution de l’équation $x = u_n + h\varphi(t_{n+1}, x)$, c’est-à-dire un zéro de la fonction (en générale non linéaire)

$$x \mapsto -x + u_n + h\varphi(t_{n+1}, x)$$

```
def EI(phi,tt,y0):
    h  = tt[1]-tt[0]
    uu = np.empty_like(tt)
    N = len(tt)-1
    uu[0] = y0
    for n in range(N):
        fct_implicite = lambda x: -x+uu[n]+h*phi(tt[n+1],x)
        tmp = fsolve(fct_implicite, uu[n])
        uu[n+1] = tmp[0] # on récupère la valeur scalaire
    return uu
```

```
# !!!!!! Ajouter ici le schéma de  Crank-Nicolson : en markdown ET en python
```

Schéma de Crank-Nicolson :

$$\begin{cases} u_0 = y_0, \\ u_{n+1} = u_n + \frac{h}{2}\Big(\varphi(t_n, u_n) + \varphi(t_{n+1}, u_{n+1})\Big) \quad n = 0, 1, 2, \dots N-1 \end{cases}$$

avec u_{n+1} zéro de la fonction $x \mapsto -x + u_n + \frac{h}{2}(\varphi(t_n, u_n) + \varphi(t_{n+1}, x))$.

```
def CN(phi,tt,y0):
    h  = tt[1]-tt[0]
    uu = np.empty_like(tt)
    N = len(tt)-1
    uu[0] = y0
    for n in range(N):
        fct_implicite = lambda x: -x+uu[n]+0.5*h*( phi(tt[n],uu[n])+phi(tt[n+1],x) )
        tmp = fsolve(fct_implicite, uu[n])
        uu[n+1] = tmp[0]
    return uu
```

```
# !!!!!! Ajouter ici le schéma RK1_M : en markdown ET en python
```

Schéma RK1_M :

$$\begin{cases} u_0 = y_0, \\ u_{n+1} = u_n + h\varphi\left(\frac{t_n + t_{n+1}}{2}, \frac{u_n + u_{n+1}}{2}\right) \quad n = 0, 1, 2, \dots N-1 \end{cases}$$

avec u_{n+1} zéro de la fonction $x \mapsto -x + u_n + h\varphi\left(\frac{t_n + t_{n+1}}{2}, \frac{u_n + x}{2}\right)$.

```
def RK1_M(phi,tt,y0):
    h  = tt[1]-tt[0]
    uu = np.empty_like(tt)
    N = len(tt)-1
    uu[0] = y0
    for n in range(N):
        fct_implicite = lambda x: -x+uu[n]+h*phi( (tt[n]+tt[n+1])/2,(uu[n]+x)/2 )
        tmp = fsolve(fct_implicite, uu[n])
        uu[n+1] = tmp[0]
    return uu
```

On introduit la discrétisation : les nœuds d'intégration $[t_0, t_1, \dots, t_N]$ sont contenus dans le vecteur `tt` qui contient $N+1$ points espacés de $h = \frac{t_N - t_0}{N}$.

On construit la solution exacte discrète et les solutions approchées en ces points :

```
N = 8
tt = np.linspace(t0,tfinal,N+1)

# yy      = np.array([sol_exacte(t) for t in tt], dtype=object)
yy      = sol_exacte(tt)  # si vectorisée
uu_ee   = EE(phi, tt, y0)
uu_ei   = EI(phi, tt, y0)
uu_em   = EM(phi, tt, y0)
uu_rk1m = RK1_M(phi, tt, y0)
uu_cn   = CN(phi, tt, y0)
uu_heun = heun(phi, tt, y0)
```

On compare les graphes de la solution exacte discrète et des solutions approchées et on affiche le maximum de l'erreur :

$$e = \max_{n \in [0,N]} |y(t_n) - u_n| = ||\mathbf{y} - \mathbf{u}||_\infty$$

```
plt.figure(figsize=(24, 14))

rows, cols, pos = 2, 3, 0

pos +=1
plt.subplot(rows, cols, pos)
plt.plot(tt, yy, 'b-')
plt.plot(tt, uu_ee, 'r-D')
# err_ee = max( [abs(uu_ee[i] - yy[i]) for n in range(N)] )
err_ee = np.linalg.norm( uu_ee-yy, np.inf)
plt.title(f'Euler explicite\nmax(|erreur|)= {err_ee:1.10f}')
plt.grid()

pos +=1
plt.subplot(rows, cols, pos)
plt.plot(tt, yy, 'b-', tt, uu_ei, 'r-D')
# err_ei = max( [abs(uu_ei[i] - yy[i]) for n in range(N)] )
err_ei = np.linalg.norm(uu_ei-yy, np.inf)
plt.title(f'Euler implicite\nmax(|erreur|)={err_ei:1.2f}')
plt.grid();

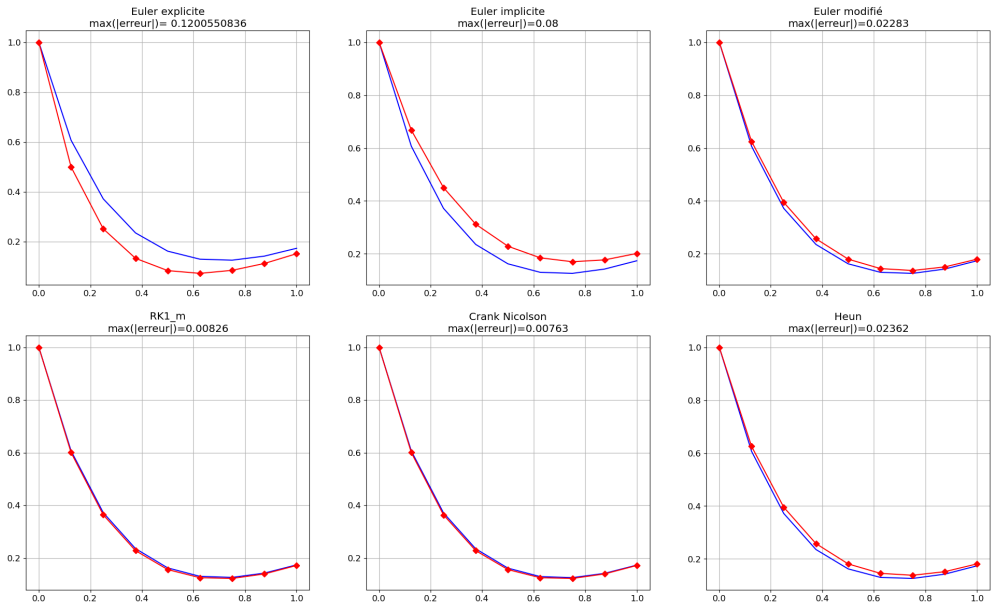

pos +=1
plt.subplot(rows, cols, pos)
plt.plot(tt, yy, 'b-', tt, uu_em, 'r-D')
# err_em = max( [abs(uu_em[i] - yy[i]) for n in range(N)] )
err_em = np.linalg.norm(uu_em-yy, np.inf)
```

```
plt.title(f'Euler modifié\nmax(|erreur|)={err_em:1.5f}')
plt.grid()

pos +=1
plt.subplot(rows, cols, pos)
plt.plot(tt, yy, 'b-', tt, uu_rk1m, 'r-D')
# err_rk1m = max( [abs(uu_rk1m[i] - yy[i]) for n in range(N)] )
err_rk1m = np.linalg.norm(uu_rk1m-yy, np.inf)
plt.title(f'RK1_m\nmax(|erreur|)={err_rk1m:1.5f}')
plt.grid()

pos +=1
plt.subplot(rows, cols, pos)
plt.plot(tt, yy, 'b-', tt, uu_cn, 'r-D')
# err_cn = max( [abs(uu_cn[i] - yy[i]) for n in range(N)] )
err_cn = np.linalg.norm(uu_cn-yy,np.inf)
plt.title(f'Crank Nicolson\nmax(|erreur|)={err_cn:1.5f}')
plt.grid()

pos +=1
plt.subplot(rows, cols, pos)
plt.plot(tt, yy, 'b-', tt, uu_heun, 'r-D')
# err_heun = max( [abs(uu_heun[i] - yy[i]) for n in range(N)] )
err_heun = np.linalg.norm(uu_heun-yy, np.inf)
plt.title(f'Heun\nmax(|erreur|)={err_heun:1.5f}')
plt.grid();
```



14.2. Exercice (à compléter) : Estimation de l’Ordre de Convergence

Considérons le même problème de Cauchy.
Ci-dessous, nous avons déjà implémenté et étudié la convergence des méthodes d’Euler explicite (EE) et d’Euler implicite (EI).

- Nous avons calculé la solution approchée pour différentes valeurs de $h_k = \frac{1}{N_k}$.
- Pour chaque valeur de h_k, nous avons déterminé l’erreur maximale et l’avons sauvegardée dans des vecteurs `err_ee` (pour EE) et `err_ei` (pour EI).
- Nous avons ensuite affiché les points $(h[k], \text{err}[k])$ en échelle logarithmique et estimé l’ordre de convergence à l’aide de la fonction `polyfit` pour chaque méthode.

Travail demandé Ajouter l’étude de convergence des méthodes suivantes :

- **Euler modifié**

- **RK1_M**
- **Crank-Nicolson**
- **Heun**

Les endroits où il faut écrire du code sont indiqués par les commentaires commençant par : `# !!!!!!`

> **Remarque**
>
> **Choix de la norme pour mesurer l'erreur.**
> Le choix de la norme doit être cohérent avec la norme continue que l'on cherche à approcher.
> Pour la norme infinie continue $\|e\|_{L^\infty} = \sup_{t\in[0,1]} |e(t)|$, la norme discrète naturelle est $\max_i |e_i|$, qui ne dépend pas du nombre de points du maillage.
> En revanche, pour approcher la norme L^2 continue $\|e\|_{L^2} = \left(\int_0^1 e(t)^2\,dt\right)^{1/2}$, la norme euclidienne discrète $\|e\|_2 = \sqrt{\sum_i e_i^2}$ doit être multipliée par $\sqrt{h}$ (où h est le pas de temps), car l'intégrale est approchée par une somme de Riemann : $\int_0^1 e(t)^2 dt \approx h\sum_i e_i^2$. On obtient donc $\|e\|_{L^2} \approx \sqrt{h}\,\|e\|_2$. Sans ce facteur $\sqrt{h}$, la norme dépend artificiellement du nombre de points et l'ordre de convergence mesuré est incorrect.

Pour chaque schéma, on calcule la solution approchée avec différentes valeurs de $h_k = 1/N_k$ et on sauvegarde les valeurs de h_k dans le vecteur `H`.

Pour chaque valeur de h_k, on calcule le maximum de la valeur absolue de l'erreur et on sauvegarde toutes ces erreurs dans le vecteur `err_schema` de sort que `err_schema[k]` contient $e_k = \max_{i=0,\dots,N_k} |y(t_i) - u_i|$.

```
H = []
err_ee = []
err_ei = []
# !!!!!! Ajouter ici les initialisations des nouveaux schémas
err_em = []
err_rk1m = []
err_cn = []
err_heun = []

for k in range(7):

    N = 2**(k + 3)

    tt = np.linspace(t0, tfinal, N + 1)
    h = tt[1] - tt[0]
    yy = sol_exacte(tt)  # si vectorisée

    uu_ee   = EE(phi, tt, y0)
    uu_ei   = EI(phi, tt, y0)
    # !!!!!! Ajouter ici l'appel aux nouveaux schémas
    uu_em   = EM(phi, tt, y0)
    uu_rk1m = RK1_M(phi, tt, y0)
    uu_cn   = CN(phi, tt, y0)
    uu_heun = heun(phi, tt, y0)

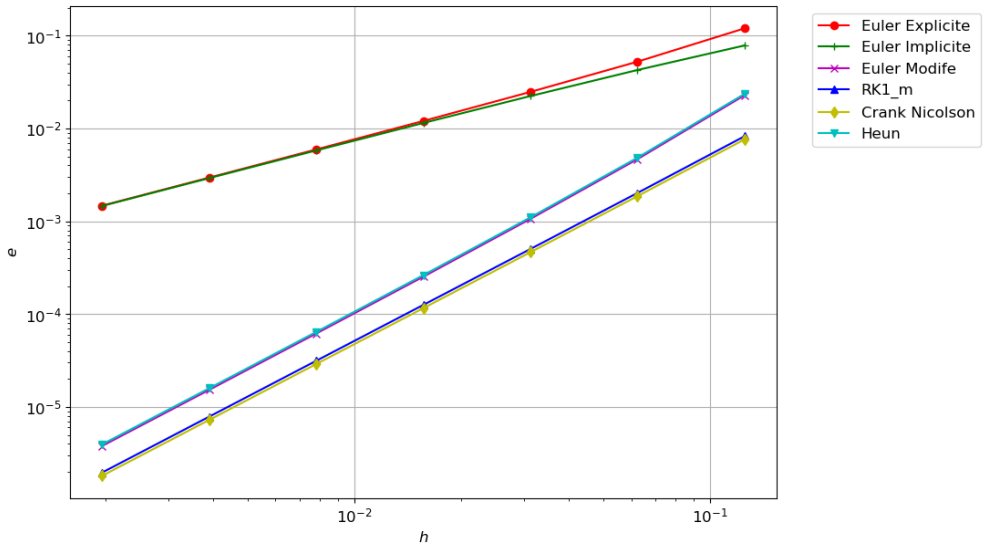
    H.append(h)

    err_ee.append(np.linalg.norm(uu_ee-yy,np.inf))
    err_ei.append(np.linalg.norm(uu_ei-yy,np.inf))
    # !!!!!! Ajouter ici le calcul des erreurs
    err_em.append(np.linalg.norm(uu_em-yy,np.inf))
    err_rk1m.append(np.linalg.norm(uu_rk1m-yy,np.inf))
    err_cn.append(np.linalg.norm(uu_cn-yy,np.inf))
    err_heun.append(np.linalg.norm(uu_heun-yy,np.inf))
```

Pour afficher l'ordre de convergence on utilise une échelle logarithmique, i.e. on représente $\ln(h)$ sur l'axe des abscisses et $\ln(\text{err})$ sur l'axe des ordonnées.

En effet, si $\text{err} = Ch^{\omega}$ alors $\ln(\text{err}) = \ln(C) + \omega \ln(h)$.
En échelle logarithmique, ω représente donc la pente de la ligne droite ln(err).

```
plt.figure(figsize=(10,7))
# plt.plot(np.log(H), np.log(err_ee), 'r-o', label='Euler Explicite')
# plt.plot(np.log(H), np.log(err_ei), 'g-+', label='Euler Implicite')
plt.loglog(H, err_ee, 'r-o', label='Euler Explicite')
plt.loglog(H, err_ei, 'g-+', label='Euler Implicite')
# !!!!!! Ajouter ici les graphes des nouveaux schémas
plt.loglog(H, err_em,   'm-x', label='Euler Modife')
plt.loglog(H, err_rk1m, 'b-^', label='RK1_m')
plt.loglog(H, err_cn,   'y-d', label='Crank Nicolson')
plt.loglog(H, err_heun, 'c-v', label='Heun')
plt.xlabel(r'$h$')
plt.ylabel(r'$e$')
plt.legend(bbox_to_anchor=(1.04, 1), loc='upper left')
plt.grid(True)
```



Pour estimer l'ordre de convergence on doit estimer la pente de la droite qui relie l'erreur au pas k à l'erreur au pas $k+1$ en echelle logarithmique. Pour estimer la pente globale de cette droite (par des moindres carrés) on peut utiliser la fonction `polyfit` (du module `numpy`).

```
# Calcul des ordres de convergence = pente des droites de regression
# ==================================================================
order_ee = np.polyfit(np.log(H), np.log(err_ee), 1)[0]
order_ei = np.polyfit(np.log(H), np.log(err_ei), 1)[0]
# !!!!!! Ajouter ici les pentes des nouveaux schémas
order_em   = np.polyfit(np.log(H), np.log(err_em), 1)[0]
order_rk1m = np.polyfit(np.log(H), np.log(err_rk1m), 1)[0]
order_cn   = np.polyfit(np.log(H), np.log(err_cn), 1)[0]
order_heun = np.polyfit(np.log(H), np.log(err_heun), 1)[0]


# Print des ordres de convergence
# ===============================

# Version avec print
# print (f'Euler progressif : ordre = {order_ee:1.2f}')
# print (f'Euler regressif  : ordre = {order_ei:1.2f}')
# print (f'Euler modifié  : ordre = {order_em:1.2f}')
# print (f'RK1_M  : ordre = {order_rk1m:1.2f}')
# print (f'Cranck Nicolson  : ordre = {order_cn:1.2f}')
```

```
# print (f'Heun  : ordre = {order_heun:1.2f}')

# Version avec tabulate pour une affichage ameliore
from tabulate import tabulate
T = []
T.append( ["Schéma", "Ordre"] )
T.append( ["Euler progressif", order_ee] )
T.append( ["Euler regressif", order_ei] )
T.append( ["Euler modifié", order_em] )
T.append( ["RK1_M", order_rk1m] )
T.append( ["Cranck Nicolson", order_cn] )
T.append( ["Heun", order_heun] )
display(tabulate(T,headers="firstrow",tablefmt="html",floatfmt=".2f"))

# Version avec pandas pour une affichage ameliore
# import pandas as pd
# schemes = [
#     ("Euler progressif", order_ee),
#     ("Euler regressif", order_ei),
#     ("Euler modifié", order_em),
#     ("RK1_M", order_rk1m),
#     ("Cranck Nicolson", order_cn),
#     ("Heun", order_heun)
# ]
# T = pd.DataFrame(schemes, columns=["Schéma", "Ordre"])
# display(T)
```

Schéma	Ordre
Euler progressif	1.05
Euler regressif	0.96
Euler modifié	2.08
RK1_M	2.00
Cranck Nicolson	2.01
Heun	2.08

14.3. Exercice : Fonction intégrale

Il existe des fonctions définies par des intégrales qu'on ne peut pas calculer explicitement. Il est pourtant possible de calculer des valeurs approchées de ces fonctions.
Pour $x \in [0; \frac{\pi}{2}]$ calculer, afficher la table des valeurs et tracer le graphe de la fonction

$$x \mapsto f(x) = \int_0^x \sqrt{1 - \frac{1}{4}\sin^2(t)}\mathrm{d}t$$

en approchant numériquement un problème de Cauchy (lequel ?) avec la méthode d'**Euler explicite**.
Vérifier que $f\left(\frac{\pi}{6}\right) \simeq 0.517881934859938$ et $f\left(\frac{\pi}{2}\right) \simeq 1.46746220933943$.
Remarque : f est une intégrale elliptique de seconde espèce, cf. https://fr.wikipedia.org/wiki/Int%C3%A9grale_elliptique

Rappel sur la dérivée d'une fonction définie par une intégrale :

$$F(x) = \int_{u(x)}^{v(x)} g(x,t)\mathrm{d}t \qquad \rightsquigarrow \qquad F'(x) = g\left(x, v(x)\right) v'(x) - g\left(x, u(x)\right) u'(x) + \int_{u(x)}^{v(x)} \frac{\partial}{\partial x} g(x,t)\mathrm{d}t.$$

La fonction f est solution du problème de Cauchy

$$\begin{cases} y'(t) = \sqrt{1 - \frac{1}{4}\sin^2(t)}, \\ y(0) = 0. \end{cases}$$

En effet, $f(0) = 0$ et si on intègre l'EDO entre 0 et x on a

$$f(x) = \int_0^x \sqrt{1 - \frac{1}{4}\sin^2(t)}\mathrm{d}t = \int_0^x y'(t)\mathrm{d}t = y(x) - y(0) = y(x).$$

Pour estimer la fonction f en $\frac{\pi}{6}$ et $\frac{\pi}{2}$, nous devons nous assurer que ces valeurs fassent partie des nœuds de discrétisation. Nous avons posé

- $t_0 = 0$ le point initial,
- $t_N = \frac{\pi}{2}$ le point final,
- t_i les nœuds de discrétisation définis par $t_i = t_0 + ih$ avec le pas de discrétisation $h = \frac{t_N - t_0}{N} = \frac{\pi}{2N}$.

Nous voulons que $\frac{\pi}{6}$ soit l'un des nœuds t_i, ce qui revient à chercher un indice i tel que $t_i = \frac{\pi}{6}$.

En remplaçant t_i par sa définition, nous avons $ih = \frac{\pi}{6}$, soit encore $i\frac{\pi}{2N} = \frac{\pi}{6}$. On obtient $i = \frac{2N}{6} = \frac{N}{3}$.

Cela montre que $\frac{\pi}{6}$ sera un nœud de discrétisation si $i = \frac{N}{3}$ est un entier. Autrement dit, N doit être un multiple de 3 pour garantir que i soit entier. Par ailleurs, $t_N = \frac{\pi}{2}$ est automatiquement inclus comme nœud terminal. Ainsi, **pour inclure $\frac{\pi}{6}$ et $\frac{\pi}{2}$ parmi les nœuds de discrétisation, il suffit de choisir N comme un multiple de 3.**

On calcule enfin la solution approchée et on vérifie que $f(\pi/6) \simeq 0.517881934859938$ et $f(\pi/2) \simeq 1.46746220933943$ et on affiche le graphe de la fonction approchée.

```
def EE(phi, tt, y0):
    h  = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    uu[0] = y0
    for n in range(len(tt) - 1):
        uu[n+1] = uu[n] + h * phi(tt[n], uu[n])
    return uu

# EDO
phi = lambda t,y : np.sqrt(1-0.25*(np.sin(t))**2)

# CI
t0, y0 = 0, 0

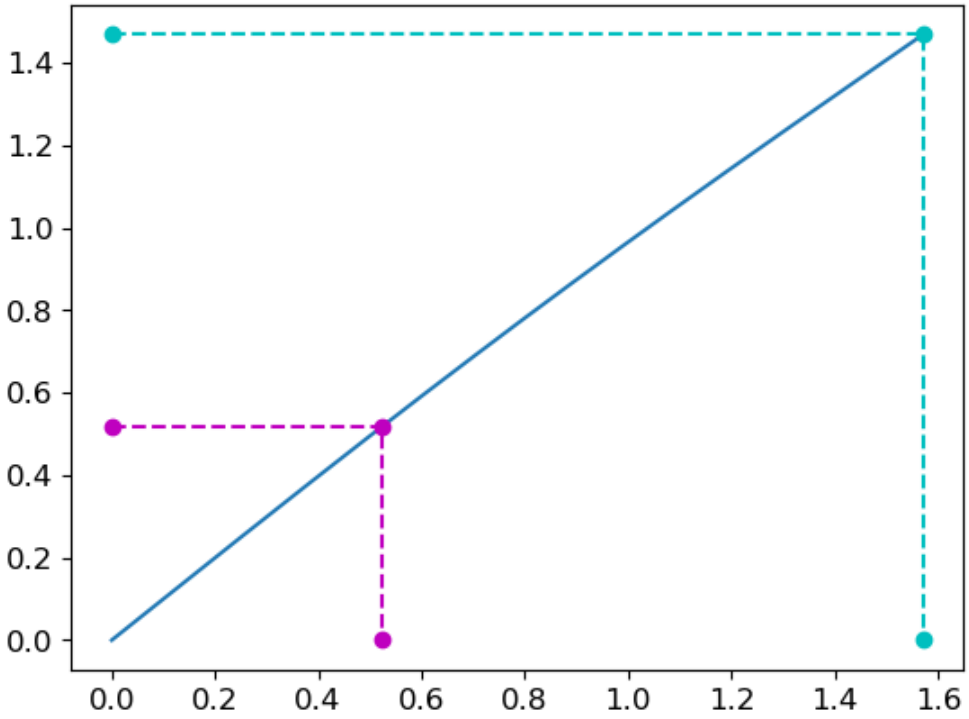
# DISCRETISATION
tfinal = np.pi/2
k = 100
N = 3*k # ainsi t_N=pi/2=3pi/6 et t_k=pi/6
tt = np.linspace(t0,tfinal,N+1)

# CALCUL DE LA SOLUTION APPROCHÉE
uu = EE(phi,tt, y0)

print(f"f(pi/{np.pi/tt[k]}) = {uu[k]:g}")
print(f"f(pi/{np.pi/tt[-1]}) = {uu[-1]:g}")

# AFFICHAGE, NON DEMANDÉ
plt.plot(tt,uu)
plt.plot([tt[0],tt[k],tt[k]],[uu[k],uu[k],0],'m--o');
plt.plot([tt[0],tt[-1],tt[-1]],[uu[-1],uu[-1],0],'c--o');
```

```
f(pi/6.0) = 0.517965
f(pi/2.0) = 1.46781
```



Et `sympy` ? Il dit bien qu'il ne peut pas calculer cette intégrale et qu'il s'agit d'une intégrale elliptique de seconde espèce.

```
t   = sp.Symbol('t')
y   = sp.Function('y')
edo = sp.Eq( sp.diff(y(t),t) , sp.sqrt(1-sp.Rational(1,4)*(sp.sin(t))**2) )
display(edo)
# solgen = sp.dsolve(edo)
# display(solgen)

t  = sp.Symbol('t')
f = sp.Integral(sp.sqrt(1-sp.Rational(1,4)*(sp.sin(t))**2))
display(f)
f.doit()
```

$$\frac{d}{dt}y(t) = \sqrt{1 - \frac{\sin^2(t)}{4}}$$

$$\int \sqrt{1 - \frac{\sin^2(t)}{4}}\, dt$$

$$E\left(t\middle|\frac{1}{4}\right)$$

Il est cependant possible d'afficher la valeur "exacte" de l'intégrale elliptique de seconde espèce en utilisant la fonction `elliptic_e` du module `sympy`.

```
display( sp.elliptic_e(np.pi/6,1/4) ) # si on utilise sp.pi, il ne donne pas le résultat float
display( sp.elliptic_e(np.pi/2,1/4) )

# help(sp.elliptic_e)
```

0.517881934859938

1.46746220933943

14.4. Exercice : système de 2 EDO avec Invariant

Les résultats numériques doivent être interprétés avec prudence. Les outils numériques ne fournissent que des approximations et n'offrent aucune garantie quant à la validité des résultats obtenus.
Lorsque la solution exacte est inconnue, il faut définir des critères permettant d'évaluer la qualité des approximations. Par exemple, si on connait un invariant théorique, il est souhaitable que l'invariant soit préservé aussi précisément que possible lors d'une simulation numérique.

Problème de Cauchy
Considérons le problème de Cauchy

$$\begin{cases} x'(t) = -y(t), \\ y'(t) = x(t), \qquad \text{pour } t \in [0, 4\pi] \\ x(0) = 1, \\ y(0) = 0. \end{cases}$$

Partie 1 : Invariant
Dans la majorité des cas, on ne peut pas calculer explicitement la solution d'un système d'EDO. En revanche, il est souvent possible de trouver des fonctions invariantes du système. Une fonction $t \mapsto I(t)$ est un invariant du système si, pour toute valeur de t, la valeur $I(t)$ est égale à la valeur initiale $I(0)$. Trouver un invariant calculable sans expliciter la solution est très utile pour évaluer la qualité d'une méthode numérique. En effet, la qualité d'une méthode numérique pour approcher la solution de ce problème peut être mesurée en calculant dans quelle mesure cet invariant calculé avec la solution approchée s'éloigne de la valeur initiale au cours du temps.
Travail demandé : vérifier que la fonction $t \mapsto I(t) = x^2(t) + y^2(t)$ est un invariant pour le système.
Partie 2 : Approximations Numériques
Choix des paramètres et notations :

- Nombre de points de discrétisation : $N = 193$
- Pas de temps : $h = t_{n+1} - t_n = \frac{\pi}{48}$
- $\mathbf{z}_n = (u_n, w_n)$ avec $u_n \approx x_n = x(t_n)$ et $w_n \approx y_n = y(t_n)$
- $J_n \approx I_n = I(t_n)$

Travail demandé : implémenter les schémas ci-dessous et compléter l'analyse indiquée.

1. Méthode d'**Euler explicite**
   - Tracer (t_n, u_n) et (t_n, w_n) sur un même graphique.
   - Tracer (t_n, J_n) sur un autre graphique.
   - Tracer (u_n, w_n) (portrait de phase) et vérifier que la solution numérique diverge vers l'extérieur.
   - Montrer analytiquement que $J_n = (1 + h^2)^n$ et interpréter les résultats.
2. Méthode d'**Euler implicite**
   - Répéter les mêmes analyses que pour Euler explicite.
   - *Remarque :* la linéarité du second membre permet de rendre ce schéma explicite.
   - *À faire :* commenter les résultats et écrire l'expression analytique de J_n.
3. Méthode de **Crank-Nicolson**
   - Répéter les mêmes étapes que précédemment.
   - *Remarque :* la linéarité du second membre permet également ici de rendre ce schéma explicite.
   - *À faire :* commenter les résultats et écrire l'expression analytique de J_n.

4. Méthode avec `odeint` du module `scipy.integrate`
   - Appliquer la fonction `odeint` pour résoudre le système.
   - Analyser le comportement de l'invariant J_n et commenter les résultats.

Partie 3 : Solution exacte

Il est possible de calculer analytiquement la solution exacte dans ce cas particulier.

Travail demandé : utiliser le module `sympy` pour déterminer la solution exacte et comparer avec les solutions approchées.

Partie 1 : Invariant

Pour tout t on a

$$I'(t) = \left(x^2(t) + y^2(t)\right)' = 2x'(t)x(t) + 2y'(t)y(t) = -2x(t)y(t) + 2x(t)y(t) = 0$$

donc $I(t) = I(0) = 0^2 + 1^2 = 1$.

Partie 2 : Approximations

```
t0 = 0
tfinal = 4*np.pi


N = 192 #*100
tt = np.linspace(0, tfinal, N+1)  # np.arange(t0, tfinal+h/2, h)


# Notation vectorielle
# ====================

# CI (2 conditions)
zz0 = np.array([1,0])

# EDO (2 equations)
# phi : RxR^2 -> R^2
phi = lambda t,zz : np.array( [ -zz[1], zz[0] ] )
```

Pour simplifier les affichages, on définit une fonction `affichage` qui affiche les graphes de u_n et w_n en fonction de t_n, le graphe de J_n et enfin le portrait de phase (u_n, w_n). Il suffira alors d'appeler cette fonction pour chaque schéma.

```
def affichage(zz, JJ, tt, zz0, phi, schema):

    # zz[:, 0] = x(t) [vecteur uu]
    # zz[:, 1] = y(t) [vecteur ww]

    plt.figure(figsize=(24, 7))

    # Graphe de x(t) et y(t)
    plt.subplot(1, 3, 1)
    plt.plot(tt, zz[:, 0], label='x(t)')
    plt.plot(tt, zz[:, 1], label='y(t)')
    plt.grid()
    plt.xlabel('t')
    plt.legend()
    plt.title(f'{schema} - x(t) et y(t)')

    # Graphe de l'invariant JJ
    plt.subplot(1, 3, 2)
    plt.plot(tt, JJ)
```

```
    plt.grid()
    plt.xlabel('t')
    plt.title(f'{schema} - Invariant')

    # Diagramme de phase y(x)
    plt.subplot(1, 3, 3)
    u_min, u_max = min(zz[:, 0]), max(zz[:, 0])
    w_min, w_max = min(zz[:, 1]), max(zz[:, 1])
    Y1, Y2 = np.meshgrid(np.linspace(u_min, u_max, 20), np.linspace(w_min, w_max, 20))

    # Calcul du champ de vecteurs
    V = np.array([phi(0, [y1, y2]) for y1, y2 in zip(Y1.flatten(), Y2.flatten())])
    V1 = V[:, 0].reshape(Y1.shape)
    V2 = V[:, 1].reshape(Y2.shape)
    r = np.sqrt(V1**2 + V2**2)

    # Affichage du champ et des trajectoires
    #plt.quiver(Y1, Y2, V1 / r, V2 / r)
    plt.streamplot(Y1, Y2, V1, V2, color='lightgray', density=1.5)  # lignes de courant
    plt.plot(zz[:, 0], zz[:, 1], 'r')
    plt.scatter(zz[0, 0], zz[0, 1], c='g', s=100)  # point initial
    plt.xlabel('x(t)')
    plt.ylabel('y(t)')
    plt.title(f'{schema} - y(x)')
    plt.axis('equal')
    plt.show()
```

Euler explicite

$$\begin{cases} u_0 = 1, \\ w_0 = 0, \\ u_{n+1} = u_n + h\varphi_1(t_n, u_n, w_n), \\ w_{n+1} = w_n + h\varphi_2(t_n, u_n, w_n) \end{cases}$$

```
def EE(phi,tt,zz0):
    nb_eq = len(zz0) # nombre d'équations
    N = len(tt)-1 # N+1 points, donc N intervalles
    zz = np.zeros((N+1,nb_eq)) # colonnes = variables, lignes = temps
    zz[0] = zz0
    h = tt[1]-tt[0]
    for n in range(N):
        zz[n+1] = zz[n]+h*phi(tt[n],zz[n])
    return zz


# Version qui retourne un tableau 1D si zz0 est scalaire
# def EE(phi, tt, zz0):
#     zz0 = np.atleast_1d(zz0)  # Convertir zz0 en tableau (même si scalaire)
#     nb_eq = len(zz0)  # Nombre d'équations
#     N = len(tt) - 1  # N+1 points, donc N intervalles
#     zz = np.zeros((N + 1, nb_eq))  # Colonnes = variables, lignes = temps
#     zz[0] = zz0
#     h = tt[1] - tt[0]
#     for n in range(N):
#         zz[n + 1] = zz[n] + h * phi(tt[n], zz[n])
#     if nb_eq == 1:
#         return zz.flatten()  # Retourner un tableau 1D si scalaire
#     return zz
```

On remarque que

$$\begin{cases} u_0 = 1, \\ w_0 = 0, \\ u_{n+1} = u_n - hw_n, \\ w_{n+1} = w_n + hu_n \end{cases}$$

On a

$$J_{n+1} = u_{n+1}^2 + w_{n+1}^2 = (u_n - hw_n)^2 + (w_n + hu_n)^2 = (1+h^2)(u_n^2 + w_n^2) = (1+h^2)J_n$$

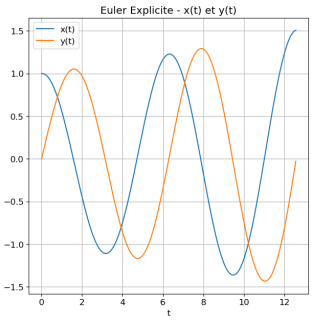
soit encore

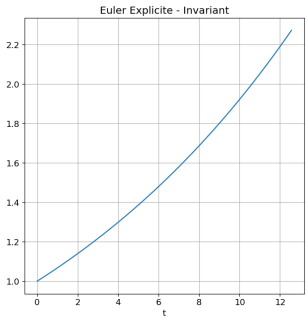
$$J_n = (1+h^2)^n J_0 = (1+h^2)^n.$$

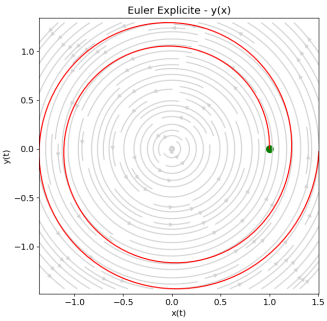
On voit que

- l'invariant n'est conservé que si $h = 0$, ce qui est impossible,
- pour h fixé, $\lim_{n \to +\infty} J_n = +\infty$ (comportement pour $t \to \infty$),
- pour n fixé, l'erreur $|J_n - I_n|$ diminue comme h^2 (convergence sur un intervalle fixé).

```
zz_ep = EE(phi,tt,zz0) # N+1 lignes, 2 colonnes
uu_ep, ww_ep = zz_ep.T
JJ_ep = uu_ep**2+ww_ep**2
affichage(zz_ep, JJ_ep, tt, zz0, phi, 'Euler Explicite')
```







Euler implicite

$$\begin{cases} u_0 = 1, \\ w_0 = 0, \\ u_{n+1} = u_n + h\varphi_1(t_{n+1}, u_{n+1}, w_{n+1}) = u_n - hw_{n+1}, \\ w_{n+1} = w_n + h\varphi_2(t_{n+1}, u_{n+1}, w_{n+1}) = w_n + hu_{n+1} \end{cases}$$

En résolvant le système linéaire on obtient une écriture explicite

$$\begin{cases} u_0 = 1, \\ w_0 = 0, \\ u_{n+1} = \dfrac{u_n - hw_n}{1+h^2}, \\ w_{n+1} = \dfrac{w_n + hu_n}{1+h^2}. \end{cases}$$

```
def EI(phi,tt,zz0):
    nb_eq = len(zz0) # nombre d'équations
    N = len(tt)-1 # N+1 points, donc N intervalles
    zz = np.zeros((N+1,nb_eq))
    zz[0] = zz0
    h = tt[1]-tt[0]
    for n in range(N):
        zz[n+1,0] = (zz[n,0]-h*zz[n,1])/(1+h**2)
        zz[n+1,1] = (zz[n,1]+h*zz[n,0])/(1+h**2)
    return zz
```

```
#  VERSION AVEC fsolve (non demandé)
from scipy.optimize import fsolve

def EI_bis(phi,tt,zz0):
    nb_eq = len(zz0) # nombre d'équations
    N = len(tt)-1 # N+1 points, donc N intervalles
    zz = np.zeros((N+1,nb_eq))
    zz[0] = zz0
    h = tt[1]-tt[0]
    for n in range(N):
        system = lambda z : z - zz[n] - h*phi(tt[n+1],z)
        zz[n+1] = fsolve( system , zz[n] )
    return zz


# Version qui retourne un tableau 1D si zz0 est scalaire
# def EI_bis(phi, tt, zz0):
#     zz0 = np.atleast_1d(zz0)  # Convertir zz0 en tableau (même si scalaire)
#     nb_eq = len(zz0)  # Nombre d'équations
#     N = len(tt) - 1  # N+1 points, donc N intervalles
#     zz = np.zeros((N + 1, nb_eq))
#     zz[0] = zz0
#     h = tt[1] - tt[0]
#     for n in range(N):
#         system = lambda z: z - zz[n] - h * phi(tt[n + 1], z)
#         zz[n + 1] = fsolve(system, zz[n])
#     if nb_eq == 1:
#         return zz.flatten()  # Retourner un tableau 1D si scalaire
#     return zz
```

On a

$$J_{n+1} = u_{n+1}^2 + w_{n+1}^2 = \frac{(u_n - hw_n)^2 + (w_n + hu_n)^2}{(1+h^2)^2} = (u_n^2 + w_n^2)\frac{1+h^2}{(1+h^2)^2} = \frac{1+h^2}{(1+h^2)^2} J_n = \frac{1}{1+h^2} J_n$$

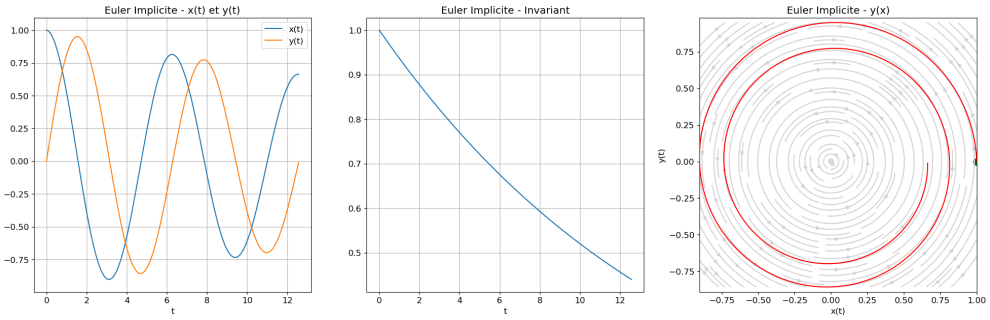
soit encore

$$J_n = \left(\frac{1}{1+h^2}\right)^n J_0 = \left(\frac{1}{1+h^2}\right)^n.$$

On voit que

- l’invariant n’est conservé que si $h = 0$, ce qui est impossible,
- pour h fixé, $\lim_{n\to+\infty} J_n = 0^+$,
- pour n fixé, l’erreur $|J_n - I_n|$ diminue comme h^2.

```
# zz_er = EI(phi,tt,zz0)
zz_er = EI_bis(phi,tt,zz0)
uu_er, ww_er = zz_er.T
JJ_er = uu_er**2+ww_er**2
affichage(zz_er, JJ_er, tt, zz0, phi, 'Euler Implicite')
```



Crank Nicolson

$$\begin{cases} u_0 = 1, \\ w_0 = 0, \\ u_{n+1} = u_n + \frac{h}{2}\Big(\varphi_1(t_n, u_n, w_n) + \varphi_1(t_{n+1}, u_{n+1}, w_{n+1})\Big) = u_n - \frac{h}{2}w_n - \frac{h}{2}w_{n+1}, \\ w_{n+1} = w_n + \frac{h}{2}\Big(\varphi_2(t_n, u_n, w_n) + \varphi_2(t_{n+1}, u_{n+1}, w_{n+1})\Big) = w_n + \frac{h}{2}u_n + \frac{h}{2}u_{n+1}. \end{cases}$$

En résolvant le système linéaire on obtient une écriture explicite :

$$\begin{cases} u_0 = 1, \\ w_0 = 0, \\ u_{n+1} = \dfrac{(4-h^2)u_n - 4hw_n}{4+h^2}, \\ w_{n+1} = \dfrac{(4-h^2)w_n + 4hu_n}{4+h^2}. \end{cases}$$

De plus, on a

$$\begin{aligned} J_{n+1} &= u_{n+1}^2 + w_{n+1}^2 \\ &= \frac{((4-h^2)u_n - 4hw_n)^2 + ((4-h^2)w_n + 4hu_n)^2}{(4+h^2)^2} \\ &= \frac{(4-h^2)^2u_n^2 + 16h^2w_n^2 - 8(4-h^2)hu_nw_n + (4-h^2)^2w_n^2 + 16h^2u_n^2 + 8(4-h^2)hu_nw_n}{(4+h^2)^2} \\ &= \frac{((4-h^2)^2 + 16h^2)(u_n^2 + w_n^2)}{(4+h^2)^2} \\ &= \frac{(4+h^2)^2(u_n^2 + w_n^2)}{(4+h^2)^2} \\ &= (u_n^2 + w_n^2) = J_n \end{aligned}$$

On voit que l'invariant est conservé pour tout h

Vérifions ces calculs avec `sympy` :

```
dt = sp.Symbol('h') # pour ne pas redefinir h comme un symbole, on écrit dt mais on affiche h
u_n = sp.Symbol('u_n')
u_np1 = sp.Symbol('u_{n+1}')
w_n = sp.Symbol('w_n')
w_np1 = sp.Symbol('w_{n+1}')
```

```
eq1 = sp.Eq( u_np1 , u_n-dt/2*w_n-dt/2*w_np1 )
eq2 = sp.Eq( w_np1 , w_n+dt/2*u_n+dt/2*u_np1 )
sol = sp.solve([eq1,eq2],[u_np1,w_np1],dict=True)[0]
display(sol)

J_np1 = u_np1**2+w_np1**2
J_np1.subs(sol).simplify()
```

```
{u_{n+1}: (-h**2*u_n - 4*h*w_n + 4*u_n)/(h**2 + 4),
 w_{n+1}: (-h**2*w_n + 4*h*u_n + 4*w_n)/(h**2 + 4)}
```

$u_n^2 + w_n^2$

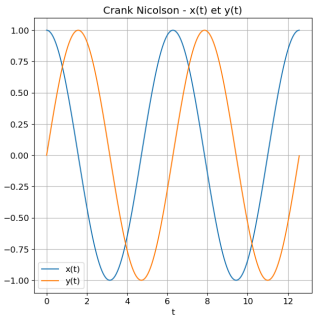
```
def cn(phi,tt,zz0):
    nb_eq = len(zz0) # nombre d'équations
    N = len(tt)-1 # N+1 points, donc N intervalles
    zz = np.zeros((N+1,nb_eq))
    zz[0] = zz0
    h = tt[1]-tt[0]
    for n in range(N):
        zz[n+1,0] = ((4-h**2)*zz[n,0]-4*h*zz[n,1])/(4+h**2)
        zz[n+1,1] = ((4-h**2)*zz[n,1]+4*h*zz[n,0])/(4+h**2)
    return zz
```

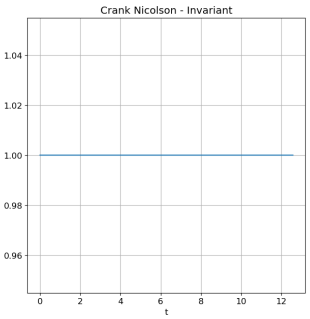
```
#  VERSION AVEC fsolve (non demandé)

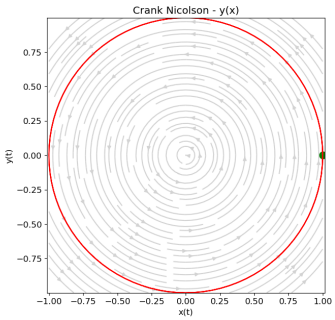
def cn_bis(phi,tt,zz0):
    nb_eq = len(zz0) # nombre d'équations
    N = len(tt)-1 # N+1 points, donc N intervalles
    zz = np.zeros((N+1,nb_eq))
    zz[0] = zz0
    h = tt[1]-tt[0]
    for n in range(N):
        system = lambda z : z - zz[n] - h/2*( phi(tt[n],zz[n]) + phi(tt[n+1],z) )
        zz[n+1] = fsolve( system , zz[n] )
    return zz

# Version qui retourne un tableau 1D si zz0 est scalaire
# def cn_bis(phi, tt, zz0):
#     zz0 = np.atleast_1d(zz0)  # Convertir zz0 en tableau (même si scalaire)
#     nb_eq = len(zz0)  # Nombre d'équations
#     N = len(tt) - 1  # N+1 points, donc N intervalles
#     zz = np.zeros((N + 1, nb_eq))
#     zz[0] = zz0
#     h = tt[1] - tt[0]
#     for n in range(N):
#         system = lambda z: z - zz[n] - h / 2 * (phi(tt[n], zz[n]) + phi(tt[n + 1], z))
#         zz[n + 1] = fsolve(system, zz[n])
#     if nb_eq == 1:
#         return zz.flatten()  # Retourner un tableau 1D si scalaire
#     return zz
```

```
# zz_cn = cn(phi,tt,zz0)
zz_cn = cn_bis(phi,tt,zz0)
uu_cn, ww_cn = zz_cn.T
JJ_cn = uu_cn**2+ww_cn**2
affichage(zz_cn, JJ_cn, tt, zz0, phi, 'Crank Nicolson')
```



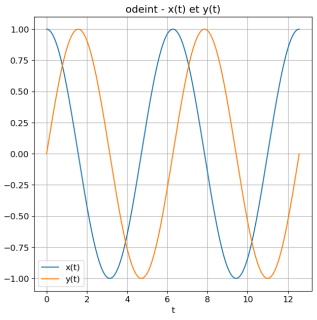


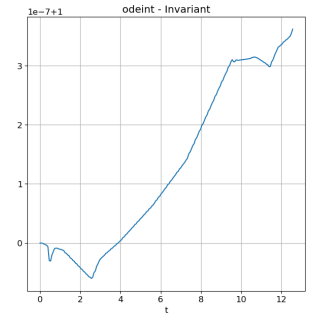


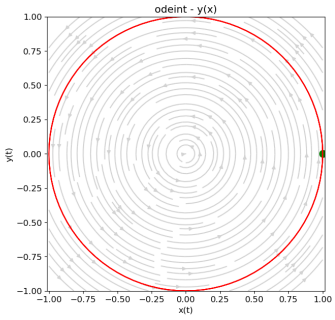
Avec `odeint` du module `scipy.integrate` :

```
zz_odeint = odeint(phi, zz0, tt, tfirst=True)
uu_odeint, ww_odeint = zz_odeint.T

JJ_odeint = uu_odeint**2+ww_odeint**2
affichage(zz_odeint, JJ_odeint, tt, zz0, phi, 'odeint')
```





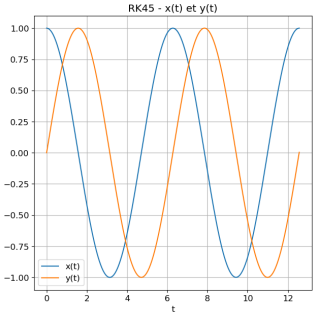


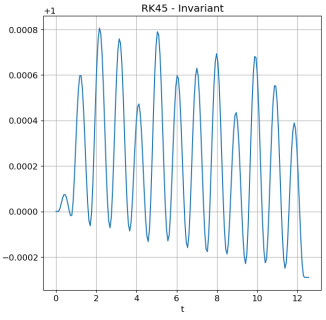
Bonus : avec `solve_ivp` du module `scipy.integrate` :

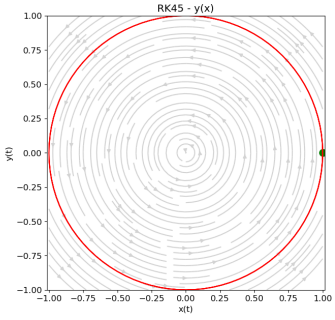
```
sol_RK45 = solve_ivp(phi,[t0,tfinal],zz0,t_eval=tt, method='RK45')
uu_RK45, ww_RK45 = sol_RK45.y

JJ_RK45 = uu_RK45**2+ww_RK45**2

affichage(sol_RK45.y.T, JJ_RK45, tt, zz0, phi, 'RK45')
```



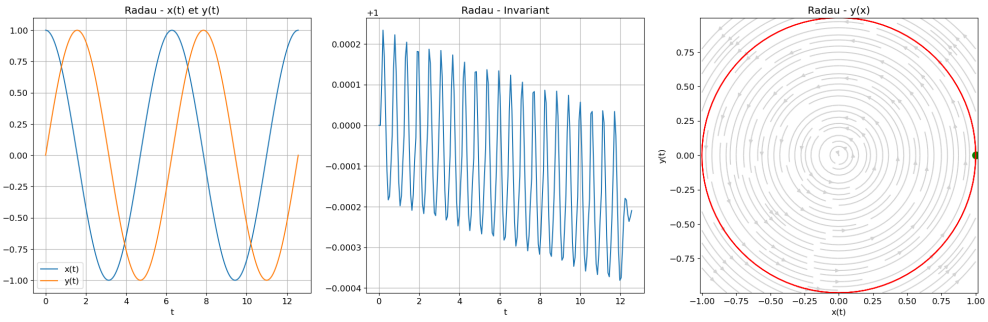





```
sol_Radau = solve_ivp(phi,[t0,tfinal],zz0,t_eval=tt, method='Radau')
uu_Radau, ww_Radau = sol_Radau.y

JJ_Radau = uu_Radau**2+ww_Radau**2

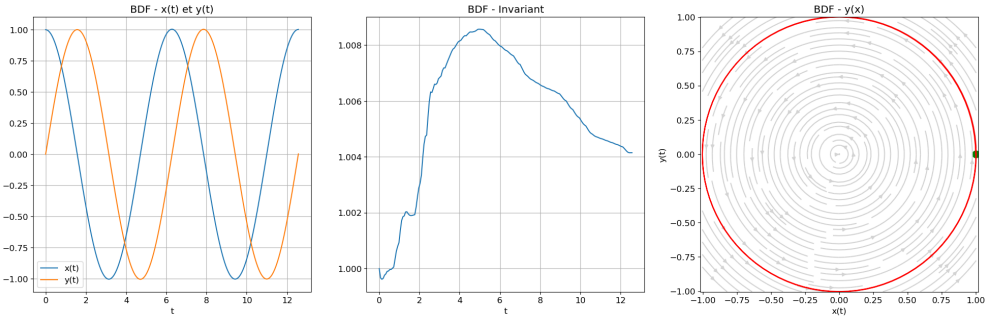
affichage(sol_Radau.y.T, JJ_Radau, tt, zz0, phi, 'Radau')
```




```
sol_BDF = solve_ivp(phi,[t0,tfinal],zz0,t_eval=tt, method='BDF')
uu_BDF, ww_BDF = sol_BDF.y

JJ_BDF = uu_BDF**2+ww_BDF**2

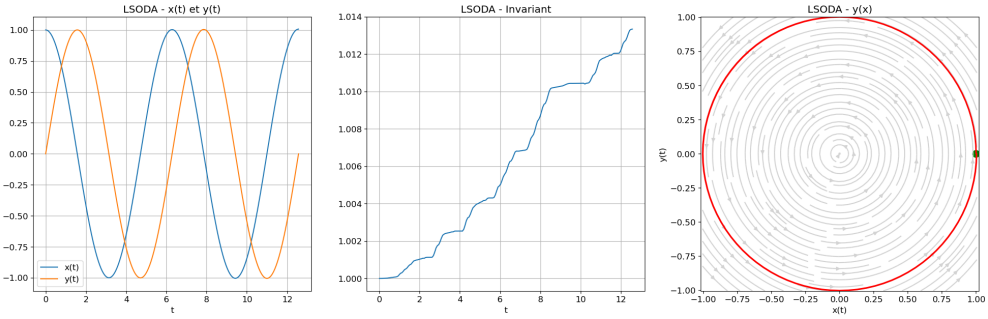
affichage(sol_BDF.y.T, JJ_BDF, tt, zz0, phi, 'BDF')
```




```
sol_LSODA = solve_ivp(phi,[t0,tfinal],zz0,t_eval=tt, method='LSODA')
uu_LSODA, ww_LSODA = sol_LSODA.y

JJ_LSODA = uu_LSODA**2+ww_LSODA**2

affichage(sol_LSODA.y.T, JJ_LSODA, tt, zz0, phi, 'LSODA')
```



Partie 3 : Solution exacte

La solution exacte est

$$x(t) = \cos(t),$$
$$y(t) = \sin(t).$$

Cette solution est périodique de période 2π et on a bien

$$I(t) = x^2(t) + y^2(t) = 1 \quad \forall t.$$

Vérifions-le avec le module `sympy` :

```
t = sp.Symbol('t')
x = sp.Function('x')
y = sp.Function('y')

t_0 = 0
x_0 = zz0[0]
y_0 = zz0[1]


edo1 = sp.Eq( sp.diff(x(t),t) , -y(t) )
edo2 = sp.Eq( sp.diff(y(t),t) , x(t) )
display(edo1)
display(edo2)


solpar_1, solpar_2 = sp.dsolve([edo1,edo2],[x(t),y(t)], ics={x(t_0): x_0, y(t_0): y_0})
display(solpar_1)
display(solpar_2)

# AFFICHAGE
func_1 = sp.lambdify(t,solpar_1.rhs,'numpy')
func_2 = sp.lambdify(t,solpar_2.rhs,'numpy')

tt = np.linspace(0,4*np.pi,193)
xx = func_1(tt)
yy = func_2(tt)
JJ = xx**2+yy**2

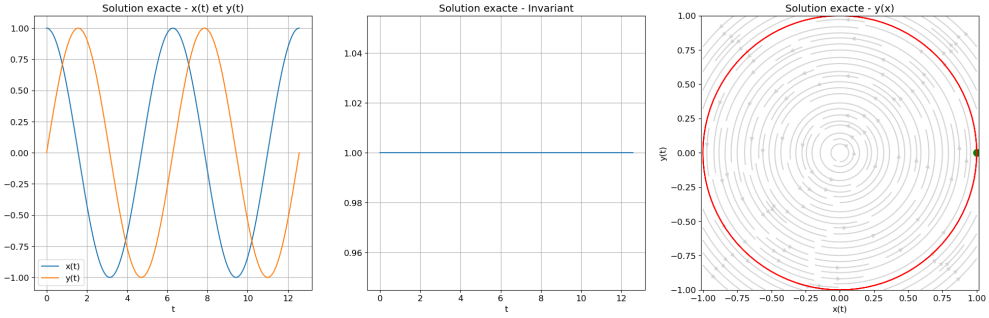
affichage(np.array([xx,yy]).T, JJ, tt, zz0, phi, 'Solution exacte')
```

$$\frac{d}{dt}x(t) = -y(t)$$

$$\frac{d}{dt}y(t) = x(t)$$

$$x(t) = \cos(t)$$

$$y(t) = \sin(t)$$



Chapitre 15.

TP 3 - Implémentation de schémas

15.1. Exercice (à compléter) : Implémentation et Comparaison de Schémas

Objectif : implémenter les méthodes indiquées et les **tester progressivement** sur le problème de Cauchy :

$$\begin{cases} y'(t) = \sin(t) + y(t), \quad \forall t \in I = [0,1], \\ y(0) = 0 \end{cases}$$

Travail à réaliser :

1. **Implémenter les schémas numériques** indiqués, classés par catégories : schémas explicites, schémas implicites, schémas predictor-corrector. Comme d'habitude, on notera $\varphi_j = \varphi(t_j, u_j)$.
2. **Calculer la solution exacte** du problème de Cauchy en utilisant le module `sympy`.
3. Pour chaque schéma, **calculer la solution approchée** avec un pas $h = \frac{1}{N}$ et $N = 8$ (une grille grossière permet de mieux visualiser les erreurs).
4. **Analyse et comparaison des résultats :**
   - Pour chaque schéma, tracer la **solution exacte** et la **solution approchée**.
   - Calculer et afficher l'**erreur maximale** définie par : $e = \max_{i=0,\dots,N} |y(t_i) - u_i|$

Démarche recommandée :

- Calculer la solution exacte et tester les deux schémas déjà implémentés (**Étape 0**). Pour cela on s'appuiera sur le TP2.
- Implémenter un seul schéma (**Étape 1**).
- Tester immédiatement ce schéma sur le problème de Cauchy donné (**Étape 2**).
- Répéter cette démarche pour chaque méthode demandée (commencer par une méthode explicite, puis une méthode implicite, puis une méthode predictor-corrector et recommencer).

Les sections où vous devez ajouter du code sont signalées par les commentaires commençant par : `# !!!!!!`

Remarques sur les boucles pour l'implémentation des suites définies par récurrence :

Rappels :

1. En python,
   - L'instruction `range(5)` produit 0,1,2,3,4
   - L'instruction `range(1,5)` produit 1,2,3,4
2. Selon notre notation,
   - la suite $\{t_0, t_1, \dots, t_N\}$ contient les $N+1$ points de *discrétisation* ;
   - la suite $\{u_0, u_1, \dots, u_N\}$ contient les $N+1$ valeurs à calculer.

$\rightsquigarrow$ `len(tt)` $= N + 1$.
Itérations pour calculer u_N :

- pour un schéma à un pas, $u_{n+1} = F(u_n)$; on initialise u_0 et on calcule $F(u_n)$ pour n de 0 jusqu'à $N-1$, autrement dit `n in range(N)` soit encore `n in range(len(tt)-1)` ;
- pour un schéma à deux pas, $u_{n+1} = F(u_n, u_{n-1})$; on initialise u_0 et u_1 et on calcule $F(u_n, u_{n-1})$ pour n de 1 jusqu'à $N-1$, autrement dit `n in range(1,N)` soit encore `n in range(1,len(tt)-1)` ;
- pour un schéma à trois pas, $u_{n+1} = F(u_n, u_{n-1}, u_{n-2})$; on initialise u_0, u_1 et u_2 et on calcule u_{n+1} pour n de 2 jusqu'à $N-1$, autrement dit `n in range(2,N)` soit encore `n in range(2,len(tt)-1)`
- ...
- pour un schéma à $q = p+1$ pas, $u_{n+1} = F(u_n, u_{n-1}, u_{n-2}, \dots, u_{n-p})$; on initialise u_0, u_1 et u_p et on calcule u_{n+1} pour n de p jusqu'à $N-1$, autrement dit `n in range(p,N)` soit encore `n in range(p,len(tt)-1)`

```
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import fsolve
plt.rcParams.update({'font.size': 12})
from IPython.display import display, Markdown
```

15.2. Schémas explicites

15.2.1. Schéma AB-1

Schéma d'Euler progressif = d'Adam-Bashforth à 1 pas

$$\begin{cases} u_0 = y_0, \\ u_{n+1} = u_n + h\varphi_n \quad n = 0, 1, 2, \dots N-1 \end{cases}$$

```
def EE(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initial condition
    uu[0] = y0
    # time-stepping loop
    for n in range(len(tt)-1):
        k1 = phi( tt[n], uu[n] )
        uu[n+1] = uu[n]+k1*h
    return uu
```

15.2.2. Schéma AB-2

Schéma d'Adam-Bashforth à 2 pas

$$\begin{cases} u_0 = y_0, \\ u_1 = u_0 + h\varphi_0, \\ u_{n+1} = u_n + \dfrac{h}{2}\Big(3\varphi_n - \varphi_{n-1}\Big) \quad n = 1, 2, 3, 4, 5, \dots N-1 \end{cases}$$

```
def AB2(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initial conditions
    uu[0] = y0
    k1 = phi( tt[0], uu[0] )
    uu[1] = uu[0]+h*k1
    # time-stepping loop
    for n in range(1,len(tt)-1):
```

```
        k1 = phi( tt[n], uu[n] )
        k2 = phi( tt[n-1], uu[n-1] )
        uu[n+1] = uu[n] + (3*k1-k2)*h/2
    return uu
```

15.2.3. Schéma AB-3

Schéma d'Adam-Bashforth à 3 pas

$$\begin{cases} u_0 = y_0, \\ u_1 = u_0 + h\varphi_0, \\ u_2 = u_1 + \dfrac{h}{2}\Big(3\varphi_1 - \varphi_0\Big), \\ u_{n+1} = u_n + \dfrac{h}{12}\Big(23\varphi_n - 16\varphi_{n-1} + 5\varphi_{n-2}\Big) \quad n = 2, 3, 4, 5, \dots N-1 \end{cases}$$

```
def AB3(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initial conditions
    uu[0] = y0
    k1 = phi( tt[0], uu[0] )
    uu[1] = uu[0]+h*k1
    k2 = phi( tt[1], uu[1] )
    uu[2] = uu[1]+h*(3*k2-k1)/2
    # time-stepping loop
    for n in range(2,len(tt)-1):
        k1 = phi( tt[n], uu[n] )
        k2 = phi( tt[n-1], uu[n-1] )
        k3 = phi( tt[n-2], uu[n-2] )
        uu[n+1] = uu[n] + (23*k1-16*k2+5*k3)*h/12
    return uu
```

15.2.4. Schéma AB-4

Schéma d'Adam-Bashforth à 4 pas

$$\begin{cases} u_0 = y_0, \\ u_1 = u_0 + h\varphi_0, \\ u_2 = u_1 + \dfrac{h}{2}\Big(3\varphi_1 - \varphi_0\Big), \\ u_3 = u_2 + \dfrac{h}{12}\Big(23\varphi_2 - 16\varphi_1 + 5\varphi_0\Big), \\ u_{n+1} = u_n + \dfrac{h}{24}\Big(55\varphi_n - 59\varphi_{n-1} + 37\varphi_{n-2} - 9\varphi_{n-3}\Big) \quad n = 3, 4, 5, \dots N-1 \end{cases}$$

```
def AB4(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initial conditions
    uu[0] = y0
    k1 = phi( tt[0], uu[0] )
    uu[1] = uu[0]+h*k1
    k2 = phi( tt[1], uu[1] )
    uu[2] = uu[1]+h*(3*k2-k1)/2
    k3 = phi( tt[2], uu[2] )
    uu[3] = uu[2]+h*(23*k3-16*k2+5*k1)/12
    # time-stepping loop
    for n in range(3,len(tt)-1):
```

```
        k1 = phi( tt[n], uu[n] )
        k2 = phi( tt[n-1], uu[n-1] )
        k3 = phi( tt[n-2], uu[n-2] )
        k4 = phi( tt[n-3], uu[n-3] )
        uu[n+1] = uu[n] + (55*k1-59*k2+37*k3-9*k4)*h/24
    return uu
```

15.2.5. Schéma AB-5

Schéma d'Adam-Bashforth à 5 pas

$$\begin{cases} u_0 = y_0, \\ u_1 = u_0 + h\varphi_0, \\ u_2 = u_1 + \dfrac{h}{2}\Big(3\varphi_1 - \varphi_0\Big), \\ u_3 = u_2 + \dfrac{h}{12}\Big(23\varphi_2 - 16\varphi_1 + 5\varphi_0\Big), \\ u_4 = u_3 + \dfrac{h}{24}\Big(55\varphi_3 - 59\varphi_2 + 37\varphi_1 - 9\varphi_0\Big), \\ u_{n+1} = u_n + \dfrac{h}{720}\Big(1901\varphi_n - 2774\varphi_{n-1} + 2616\varphi_{n-2} - 1274\varphi_{n-3} + 251\varphi_{n-4}\Big) \quad n = 4, 5, \dots N-1 \end{cases}$$

```
def AB5(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initial conditions
    uu[0] = y0
    k1 = phi( tt[0], uu[0] )
    uu[1] = uu[0]+h*k1
    k2 = phi( tt[1], uu[1] )
    uu[2] = uu[1]+h*(3*k2-k1)/2
    k3 = phi( tt[2], uu[2] )
    uu[3] = uu[2]+h*(23*k3-16*k2+5*k1)/12
    k4 = phi( tt[3], uu[3] )
    uu[4] = uu[3]+h*(55*k4-59*k3+37*k2-9*k1)/24
    # time-stepping loop
    for n in range(4,len(tt)-1):
        k1 = phi( tt[n], uu[n] )
        k2 = phi( tt[n-1], uu[n-1] )
        k3 = phi( tt[n-2], uu[n-2] )
        k4 = phi( tt[n-3], uu[n-3] )
        k5 = phi( tt[n-4], uu[n-4] )
        uu[n+1] = uu[n] + (1901*k1-2774*k2+2616*k3-1274*k4+251*k5)*h/720
    return uu
```

15.2.6. Schéma N-2

Schéma de Nylström à 2 pas

$$\begin{cases} u_0 = y_0, \\ u_1 = u_0 + h\varphi_0, \\ u_{n+1} = u_{n-1} + 2h\varphi_n \quad n = 1, 2, 3, 4, 5, \dots N-1 \end{cases}$$

```
def N2(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initial conditions
    uu[0] = y0
```

```
    k1 = phi( tt[0], uu[0] )
    uu[1] = uu[0]+h*k1
    # time-stepping loop
    for n in range(1,len(tt)-1):
        k1 = phi( tt[n], uu[n] )
        uu[n+1] = uu[n-1] + 2*h*k1
    return uu
```

15.2.7. Schéma N-3

Schéma de Nylström à 3 pas

$$\begin{cases} u_0 = y_0, \\ u_1 = u_0 + h\varphi_0, \\ u_2 = u_0 + 2h\varphi_1, \\ u_{n+1} = u_{n-1} + \frac{h}{3}\Big(7\varphi_n - 2\varphi_{n-1} + \varphi_{n-2}\Big) \quad n = 2, 3, 4, 5, \dots N-1 \end{cases}$$

```
def N3(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initial conditions
    uu[0] = y0
    k1 = phi( tt[0], uu[0] )
    uu[1] = uu[0]+h*k1
    k2 = phi( tt[1], uu[1] )
    uu[2] = uu[0]+2*h*k2
    # time-stepping loop
    for n in range(2,len(tt)-1):
        k1 = phi( tt[n], uu[n] )
        k2 = phi( tt[n-1], uu[n-1] )
        k3 = phi( tt[n-2], uu[n-2] )
        uu[n+1] = uu[n-1] + (7*k1-2*k2+k3)*h/3
    return uu
```

15.2.8. Schéma N-4

Schéma de Nylström à 4 pas

$$\begin{cases} u_0 = y_0, \\ u_1 = u_0 + h\varphi_0, \\ u_2 = u_0 + 2h\varphi_1, \\ u_3 = u_1 + \frac{h}{3}\Big(7\varphi_2 - 2\varphi_1 + \varphi_0\Big), \\ u_{n+1} = u_{n-1} + \frac{h}{3}\Big(8\varphi_n - 5\varphi_{n-1} + 4\varphi_{n-2} - \varphi_{n-3}\Big) \quad n = 3, 4, 5, \dots N-1 \end{cases}$$

```
def N4(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initial conditions
    uu[0] = y0
    k1 = phi( tt[0], uu[0] )
    uu[1] = uu[0]+h*k1
    k2 = phi( tt[1], uu[1] )
    uu[2] = uu[0]+2*h*k2
    k3 = phi( tt[2], uu[2] )
    uu[3] = uu[1]+(7*k3-2*k2+k1)*h/3
    # time-stepping loop
```

```
    for n in range(3,len(tt)-1):
        k1 = phi( tt[n], uu[n] )
        k2 = phi( tt[n-1], uu[n-1] )
        k3 = phi( tt[n-2], uu[n-2] )
        k4 = phi( tt[n-3], uu[n-3] )
        uu[n+1] = uu[n-1] + (8*k1-5*k2+4*k3-k4)*h/3
    return uu
```

15.2.9. Schéma EM

Schéma d'Euler modifié (qui est un schéma RK à 2 étages d'ordre 2)

$$\begin{cases} u_0 = y_0, \\ \tilde{u} = u_n + \frac{h}{2}\varphi_n, \\ u_{n+1} = u_n + h\varphi\left(t_n + \frac{h}{2}, \tilde{u}\right) \quad n = 0, 1, 2, \dots N-1 \end{cases}$$

```
def EM(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initial condition
    uu[0] = y0
    # time-stepping loop
    for n in range(len(tt)-1):
        k1 = phi( tt[n], uu[n] )
        k2 = phi( tt[n]+h/2, uu[n]+h/2*k1 )
        uu[n+1] = uu[n]+h*k2
    return uu
```

15.2.10. Schéma RK-4.1

$$\begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n, u_n\right) \\ K_2 = \varphi\left(t_n + \frac{h}{2}, u_n + \frac{h}{2}K_1\right)) \\ K_3 = \varphi\left(t_n + \frac{h}{2}, u_n + \frac{h}{2}K_2\right) \\ K_4 = \varphi\left(t_{n+1}, u_n + hK_3\right) \\ u_{n+1} = u_n + \frac{h}{6}\left(K_1 + 2K_2 + 2K_3 + K_4\right) \quad n = 0, 1, \dots N-1 \end{cases}$$

```
def RK4(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initial condition
    uu[0] = y0
    # time-stepping loop
    for n in range(len(tt)-1):
        k1 = phi( tt[n], uu[n] )
        k2 = phi( tt[n]+h/2 , uu[n]+k1*h/2 )
        k3 = phi( tt[n]+h/2 , uu[n]+h*k2/2 )
        k4 = phi( tt[n+1]   , uu[n]+h*k3 )
        uu[n+1] = uu[n] + (k1+2*k2+2*k3+k4)*h/6
    return uu
```

15.2.11. Schéma RK-6.5

Schéma de Runge-Kutta RK6_5 (Butcher à 6 étages d'ordre 5)

$$\begin{array}{c|cccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{4} & \frac{1}{8} & \frac{1}{8} & 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & -\frac{1}{2} & 1 & 0 & 0 & 0 \\ \frac{3}{4} & \frac{3}{16} & 0 & 0 & \frac{9}{16} & 0 & 0 \\ 1 & \frac{-3}{7} & \frac{2}{7} & \frac{12}{7} & \frac{-12}{7} & \frac{8}{7} & 0 \\ \hline & \frac{7}{90} & 0 & \frac{32}{90} & \frac{12}{90} & \frac{32}{90} & \frac{7}{90} \end{array}$$

```
def RK6_5(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    uu[0] = y0
    # recurrence
    for n in range(len(tt)-1):
        k1 = phi( tt[n]      , uu[n] )
        k2 = phi( tt[n]+h/4  , uu[n]+h*k1/4 )
        k3 = phi( tt[n]+h/4  , uu[n]+h*(k1+k2)/8 )
        k4 = phi( tt[n]+h/2  , uu[n]+h*(-k2+2*k3)/2 )
        k5 = phi( tt[n]+h*3/4, uu[n]+h*(3*k1+9*k4)/16 )
        k6 = phi( tt[n+1]    , uu[n]+h*(-3*k1+2*k2+12*k3-12*k4+8*k5)/7 )
        uu[n+1] = uu[n] + (7*k1+32*k3+12*k4+32*k5+7*k6)*h/90
    return uu
```

15.2.12. Schéma RK-7.6

Schéma de Runge-Kutta RK7_6 (Butcher à 7 étages d'ordre 6)

$$\begin{array}{c|ccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{2}{3} & \frac{2}{9} & \frac{4}{9} & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{3} & \frac{7}{36} & \frac{2}{9} & -\frac{1}{12} & 0 & 0 & 0 & 0 \\ \frac{5}{6} & -\frac{35}{144} & -\frac{55}{36} & \frac{35}{48} & \frac{15}{8} & 0 & 0 & 0 \\ \frac{1}{6} & -\frac{1}{360} & -\frac{11}{36} & -\frac{1}{8} & \frac{1}{2} & \frac{1}{10} & 0 & 0 \\ 1 & \frac{-41}{260} & \frac{22}{13} & \frac{43}{156} & -\frac{118}{39} & \frac{32}{195} & \frac{80}{39} & 0 \\ \hline & \frac{13}{200} & 0 & \frac{11}{40} & \frac{11}{40} & \frac{4}{25} & \frac{4}{25} & \frac{13}{200} \end{array}$$

```
def RK7_6(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    uu[0] = y0
    # recurrence
    for n in range(len(tt)-1):
        k1 = phi( tt[n]      , uu[n] )
        k2 = phi( tt[n]+h/2  , uu[n]+h*(        k1
 ↪  )/2 )
        k3 = phi( tt[n]+h*2/3, uu[n]+h*(      2*k1     +4*k2
 ↪  )/9 )
        k4 = phi( tt[n]+h/3  , uu[n]+h*(      7*k1     +8*k2      -3*k3
 ↪  )/36 )
        k5 = phi( tt[n]+h*5/6, uu[n]+h*(    -35*k1   -220*k2    +105*k3    +270*k4
 ↪  )/144 )
        k6 = phi( tt[n]+h/6  , uu[n]+h*(       -k1   -110*k2     -45*k3    +180*k4     +36*k5
 ↪  )/360 )
        k7 = phi( tt[n+1]    , uu[n]+h*(-41/260*k1 +22/13*k2 +43/156*k3 -118/39*k4 +32/195*k5 +80/39*k6)
 ↪  )
```

```
        uu[n+1] = uu[n] + (13/200*k1+11/40*k3+11/40*k4+4/25*k5+4/25*k6+13/200*k7)*h
    return uu
```

15.3. Schémas implicites

15.3.1. Schéma EI

Schéma d'Euler régressif (AM-0)

$$\begin{cases} u_0 = y_0, \\ u_{n+1} = u_n + h\varphi(t_{n+1}, u_{n+1}) \quad n = 0, 1, 2, \dots N-1 \end{cases}$$

avec u_{n+1} zéro de la fonction

$$x \mapsto -x + u_n + h\varphi(t_{n+1}, x)$$

```
def EI(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initial condition
    uu[0] = y0
    # time-stepping loop
    for n in range(len(tt)-1):
        f = lambda x: -x+uu[n]+h*phi(tt[n+1],x)
        temp = fsolve(f, uu[n])
        uu[n+1] = temp[0]
    return uu
```

15.3.2. Schéma CN = AM-1

Schéma de Crank-Nicolson (AM-1 = BDF-1)

$$\begin{cases} u_0 = y_0, \\ u_{n+1} = u_n + \frac{h}{2}\Big(\varphi_n + \varphi(t_{n+1}, u_{n+1})\Big) \quad n = 0, 1, 2, \dots N-1 \end{cases}$$

avec u_{n+1} zéro de la fonction

$$x \mapsto -x + u_n + \frac{h}{2}(\varphi_n + \varphi(t_{n+1}, x))$$

```
def CN(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initial condition
    uu[0] = y0
    # time-stepping loop
    for n in range(len(tt)-1):
        f = lambda x: -x+uu[n]+0.5*h*( phi(tt[n+1],x)+phi(tt[n],uu[n]) )
        temp = fsolve(f, uu[n])
        uu[n+1] = temp[0]
    return uu
```

15.3.3. Schéma AM-2

Schéma d'Adam-Moulton à 2 pas

$$\begin{cases} u_0 = y_0, \\ u_1 = u_0 + \frac{h}{2}\Big(\varphi_1 + \varphi_0\Big), \\ u_{n+1} = u_n + \frac{h}{12}\Big(5\varphi(t_{n+1}, u_{n+1}) + 8\varphi_n - \varphi_{n-1}\Big) \quad n = 1, 2, \dots N-1 \end{cases}$$

avec u_{n+1} un zéro de la fonction

$$x \mapsto -x + u_n + \frac{h}{12}\Big(5\varphi(t_{n+1}, x) + 8\varphi_n - \varphi_{n-1}\Big).$$

```
def AM2(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)

    # initial condition
    uu[0] = y0

    k0 = phi( tt[0], uu[0] )
    temp = fsolve(lambda x: -x+uu[0]+0.5*h*( phi(tt[1],x)+k0 ), uu[0])
    uu[1] = temp[0]

    # time-stepping loop
    for n in range(1,len(tt)-1):
        f = lambda x: -x+uu[n]+h*( 3*phi(tt[n+1],x)-phi(tt[n],uu[n]) )/2
        temp = fsolve(f, uu[n])
        uu[n+1] = temp[0]
    return uu
```

15.3.4. Schéma AM-3

Schéma d'Adam-Moulton à 3 pas

$$\begin{cases} u_0 = y_0, \\ u_1 = u_0 + \frac{h}{2}\Big(\varphi_1 + \varphi_0\Big), \\ u_2 = u_1 + \frac{h}{12}\Big(5\varphi_2 + 8\varphi_1 - \varphi_0\Big), \\ u_{n+1} = u_n + \frac{h}{24}\Big(9\varphi(t_{n+1}, u_{n+1}) + 19\varphi_n - 5\varphi_{n-1} + \varphi_{n-2}\Big) \quad n = 2, 3, \dots N-1 \end{cases}$$

avec u_{n+1} un zéro de la fonction

$$x \mapsto -x + u_n + \frac{h}{24}\Big(9\varphi(t_{n+1}, x) + 19\varphi_n - 5\varphi_{n-1} + \varphi_{n-2}\Big).$$

```
def AM3(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)

    # initial condition
    uu[0] = y0

    k0 = phi( tt[0], uu[0] )
    temp = fsolve(lambda x: -x+uu[0]+0.5*h*( phi(tt[1],x)+k0 ), uu[0])
    uu[1] = temp[0]
```

```
    k1 = phi( tt[1], uu[1] )
    temp = fsolve(lambda x: -x+uu[1]+h*( 5*phi(tt[2],x)+8*k1-phi(tt[0],uu[0]) )/12, uu[1])
    uu[2] = temp[0]

    # time-stepping loop
    for n in range(2,len(tt)-1):
        k0 = phi( tt[n], uu[n] )
        k1 = phi( tt[n-1], uu[n-1] )
        k2 = phi( tt[n-2], uu[n-2] )
        f = lambda x: -x + uu[n] + h*( 9*phi(tt[n+1],x) +19*k0 -5*k1 +k2 )/24
        temp = fsolve(f, uu[n])
        uu[n+1] = temp[0]
    return uu
```

15.3.5. Schéma AM-4

Schéma d'Adam-Moulton à 4 pas

$$\begin{cases} u_0 = y_0, \\ u_1 = u_0 + \frac{h}{2}\Big(\varphi_1 + \varphi_0\Big), \\ u_2 = u_1 + \frac{h}{12}\Big(5\varphi_2 + 8\varphi_1 - \varphi_0\Big), \\ u_3 = u_2 + \frac{h}{24}\Big(9\varphi_3 + 19\varphi_2 - 5\varphi_1 + \varphi_0\Big), \\ u_{n+1} = u_n + \frac{h}{720}\Big(251\varphi(t_{n+1}, u_{n+1}) + 646\varphi_n - 264\varphi_{n-1} + 106\varphi_{n-2} - 19\varphi_{n-3}\Big) \quad n = 3, 4, \dots N-1 \end{cases}$$

avec u_{n+1} un zéro de la fonction

$$x \mapsto -x + u_n + \frac{h}{720}\Big(251\varphi(t_{n+1}, x) + 646\varphi_n - 264\varphi_{n-1} + 106\varphi_{n-2} - 19\varphi_{n-3}\Big).$$

```
def AM4(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)

    # initial condition
    uu[0] = y0

    k0 = phi( tt[0], uu[0] )
    temp = fsolve(lambda x: -x+uu[0]+0.5*h*( phi(tt[1],x)+k0 ), uu[0])
    uu[1] = temp[0]

    k1 = phi( tt[1], uu[1] )
    temp = fsolve(lambda x: -x+uu[1]+h*( 5*phi(tt[2],x)+8*k1-phi(tt[0],uu[0]) )/12, uu[1])
    uu[2] = temp[0]

    k2 = phi( tt[2], uu[2] )
    temp = fsolve(lambda x: -x+uu[2]+h*( 9*phi(tt[3],x)+19*k2-5*k1+phi(tt[0],uu[0]) )/24, uu[2])
    uu[3] = temp[0]

    # time-stepping loop
    for n in range(3,len(tt)-1):
        k0 = phi( tt[n], uu[n] )
        k1 = phi( tt[n-1], uu[n-1] )
        k2 = phi( tt[n-2], uu[n-2] )
        k3 = phi( tt[n-3], uu[n-3] )
        f = lambda x: -x + uu[n] + h*( 251*phi(tt[n+1],x)+646*k0-264*k1+106*k2-19*k3 )/720
        temp = fsolve(f, uu[n])
        uu[n+1] = temp[0]
    return uu
```

15.3.6. Schéma AM-5

Schéma d'Adam-Moulton à 5 pas

$$\begin{cases} u_0 = y_0, \\ u_1 = u_0 + \frac{h}{2}\Big(\varphi_1 + \varphi_0\Big), \\ u_2 = u_1 + \frac{h}{12}\Big(5\varphi_2 + 8\varphi_1 - \varphi_0\Big), \\ u_3 = u_2 + \frac{h}{24}\Big(9\varphi_3 + 19\varphi_2 - 5\varphi_1 + \varphi_0\Big), \\ u_4 = u_3 + \frac{h}{720}\Big(251\varphi_4 + 646\varphi_3 - 264\varphi_2 + 106\varphi_1 - 19\varphi_0\Big), \\ u_{n+1} = u_n + \frac{h}{1440}\Big(475\varphi(t_{n+1}, u_{+1}) + 1427\varphi_n - 798\varphi_{n-1} + 482\varphi_{n-2} - 173\varphi_{n-3} + 27\varphi_{n-4}\Big), \quad n = 4, 5, \dots N-1 \end{cases}$$

avec u_{n+1} un zéro de la fonction

$$x \mapsto -x + u_n + \frac{h}{1440}\Big(475\varphi(t_{n+1}, x) + 1427\varphi_n - 798\varphi_{n-1} + 482\varphi_{n-2} - 173\varphi_{n-3} + 27\varphi_{n-4}\Big).$$

```
def AM5(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)

    # initial condition
    uu[0] = y0

    k0 = phi( tt[0], uu[0] )
    temp = fsolve(lambda x: -x+uu[0]+0.5*h*( phi(tt[1],x)+k0 ), uu[0])
    uu[1] = temp[0]

    k1 = phi( tt[1], uu[1] )
    temp = fsolve(lambda x: -x+uu[1]+h*( 5*phi(tt[2],x)+8*k1-phi(tt[0],uu[0]) )/12, uu[1])
    uu[2] = temp[0]

    k2 = phi( tt[2], uu[2] )
    temp = fsolve(lambda x: -x+uu[2]+h*( 9*phi(tt[3],x)+19*k2-5*k1+phi(tt[0],uu[0]) )/24, uu[2])
    uu[3] = temp[0]

    k3 = phi( tt[3], uu[3] )
    temp = fsolve(lambda x: -x+uu[3]+h*( 251*phi(tt[4],x)+646*k3-264*k2+106*k1-19*k0 )/720, uu[3])
    uu[4] = temp[0]

    # time-stepping loop
    for n in range(4,len(tt)-1):
        k0 = phi( tt[n], uu[n] )
        k1 = phi( tt[n-1], uu[n-1] )
        k2 = phi( tt[n-2], uu[n-2] )
        k3 = phi( tt[n-3], uu[n-3] )
        k4 = phi( tt[n-4], uu[n-4] )
        f = lambda x: -x + uu[n] + h*( 475*phi(tt[n+1],x)+1427*k0-798*k1+482*k2-173*k3+27*k4 )/1440
        temp = fsolve(f, uu[n])
        uu[n+1] = temp[0]
    return uu
```

15.3.7. Schéma BDF-2

Schéma Backward Differentiation Formula à 2 pas

$$\begin{cases} u_0 = y_0, \\ u_1 = u_0 + h\varphi_1, \\ u_{n+1} = \frac{4}{3}u_n - \frac{1}{3}u_{n-1} + \frac{2}{3}h\varphi(t_{n+1}, u_{n+1}) \quad n = 1, 2, \dots N-1 \end{cases}$$

avec u_{n+1} un zéro de la fonction

$$x \mapsto -x + \frac{4}{3}u_n - \frac{1}{3}u_{n-1} + \frac{2}{3}h\varphi(t_{n+1}, x).$$

```
def BDF2(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initial condition
    uu[0] = y0
    temp = fsolve(lambda x: -x+uu[0]+h*phi(tt[1],x), uu[0])
    uu[1] = temp[0]
    # time-stepping loop
    for n in range(1,len(tt)-1):
        temp = fsolve(lambda x: -x+4/3*uu[n]-1/3*uu[n-1] + 2/3*h*phi(tt[n+1],x) , uu[n])
        uu[n+1] = temp[0]
    return uu
```

15.3.8. Schéma BDF-3

Schéma Backward Differentiation Formula à 3 pas

$$\begin{cases} u_0 = y_0, \\ u_1 = u_0 + h\varphi_1, \\ u_2 = \frac{4}{3}u_1 - \frac{1}{3}u_0 + \frac{2}{3}h\varphi_2, \\ u_{n+1} = \frac{18}{11}u_n - \frac{9}{11}u_{n-1} + \frac{2}{11}u_{n-2} + \frac{6}{11}h\varphi(t_{n+1}, u_{n+1}) \quad n = 2, 3, \dots N-1 \end{cases}$$

avec u_{n+1} un zéro de la fonction

$$x \mapsto -x + \frac{18}{11}u_n - \frac{9}{11}u_{n-1} + \frac{2}{11}u_{n-2} + \frac{6}{11}h\varphi(t_{n+1}, x).$$

```
def BDF3(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initial condition
    uu[0] = y0
    temp = fsolve(lambda x: -x+uu[0]+h*phi(tt[1],x), uu[0])
    uu[1] = temp[0]
    temp = fsolve(lambda x: -x+4/3*uu[1]-1/3*uu[0] + 2/3*h*phi(tt[2],x), uu[1])
    uu[2] = temp[0]
    # time-stepping loop
    for n in range(2,len(tt)-1):
        temp = fsolve(lambda x: -x+18/11*uu[n]-9/11*uu[n-1] + 2/11*uu[n-2]+6/11*h*phi(tt[n+1],x) , uu[n])
        uu[n+1] = temp[0]
    return uu
```

15.3.9. Schéma RK1_M

$$\begin{cases} u_0 = y_0, \\ u_{n+1} = u_n + h\varphi\left(\frac{t_n+t_{n+1}}{2}, \frac{u_n+u_{n+1}}{2}\right) \quad n = 0, 1, 2, \dots N-1 \end{cases}$$

avec u_{n+1} un zéro de la fonction

$$x \mapsto -x + u_n + h\varphi\left(\frac{t_n + t_{n+1}}{2}, \frac{u_n + x}{2}\right).$$

Rappelons-nous que ce schéma est bien un schéma de Runge-Kutta :

$$\begin{array}{c|c|c} \frac{1}{2} & \frac{1}{2} \\ \hline & 1 \end{array} \quad \Longrightarrow \quad \begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n + \frac{h}{2}, u_n + \frac{h}{2}K_1\right) \\ u_{n+1} = u_n + hK_1 \qquad\qquad n = 0, 1, \dots N-1 \end{cases}$$

avec K_1 un zéro de la fonction

$$x \mapsto -x + \varphi\left(t_n + \frac{h}{2}, u_n + \frac{h}{2}x\right).$$

```
def RK1_M(phi,tt,y0):
    h  = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initial condition
    uu[0] = y0
    # time-stepping loop
    for n in range(len(tt)-1):
        uu[n+1] = fsolve(lambda x: -x+uu[n]+h*phi( (tt[n]+tt[n+1])/2,(uu[n]+x)/2 ), uu[n])[0]
    return uu

def RK1_M_bis(phi,tt,sol_exacte):
    h  = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initial condition
    uu[0] = y0
    # time-stepping loop
    for n in range(len(tt)-1):
        k1 = fsolve( lambda x : -x + phi(tt[n]+h/2, uu[n]+h*x/2), phi(tt[n],uu[n]) )[0]
        uu[n+1] = uu[n]+h*k1
    return uu
```

15.3.10. Schéma DIRK-2

Schéma de Runge-Kutta semi-implicite à 2 étages

$$\begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n + \frac{h}{3}, u_n + \frac{h}{3}K_1\right) \\ K_2 = \varphi\left(t_{n+1}, u_n + hK_1\right) \\ u_{n+1} = u_n + \frac{h}{4}\left(3K_1 + K_2\right) \quad n = 0, 1, \dots N-1 \end{cases}$$

avec K_1 zéro de la fonction

$$x \mapsto -x + \varphi\left(t_n + \frac{h}{3}, u_n + \frac{h}{3}x\right).$$

Le point de départ du `fsolve` n'est donc pas u_n mais φ_n.

```
def DIRK2(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initial condition
    uu[0] = y0
    # time-stepping loop
    for n in range(len(tt)-1):
        k1 = fsolve(lambda kappa : -kappa+phi(tt[n]+h/3,uu[n]+h/3*kappa), phi(tt[n],uu[n])  )[0]
        k2 = phi( tt[n+1], uu[n]+h*k1 )
        uu[n+1] = uu[n]+h/4*(3*k1+k2)
    return uu
```

15.3.11. Schéma Randau-IIa-2

Schéma de Runge-Kutta implicite à 2 étages

$$\begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n + \frac{h}{3}, u_n + \frac{h}{12}(5K_1 - K_2)\right) \\ K_2 = \varphi\left(t_{n+1}, u_n + \frac{h}{4}(3K_1 + K_2)\right) \\ u_{n+1} = u_n + \frac{h}{4}\left(3K_1 + K_2\right) \qquad\qquad n = 0, 1, \dots N-1 \end{cases}$$

avec K_1 et K_2 solutions des équations

$$\begin{cases} K_1 = \varphi\left(t_n + \frac{h}{3}, u_n + \frac{h}{12}(5K_1 - K_2)\right) \\ K_2 = \varphi\left(t_{n+1}, u_n + \frac{h}{4}(3K_1 + K_2)\right) \end{cases}$$

soit encore zéro de la fonction de $\mathbb{R}^2$ dans $\mathbb{R}^2$

$$(x, y) \mapsto \begin{pmatrix} x - \varphi\left(t_n + \frac{h}{3}, u_n + \frac{h}{12}(5x - y)\right) \\ y - \varphi\left(t_{n+1}, u_n + \frac{h}{4}(3x + y)\right) \end{pmatrix}.$$

Le point de départ du `fsolve` n'est donc pas (u_n, u_n) mais (φ_n, φ_n).

```
def Randau(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initial condition
    uu[0] = y0
    # time-stepping loop
    for n in range(len(tt)-1):
          systeme = lambda K: [  K[0] - phi(tt[n] + h / 3, uu[n] + h / 12 * (5 * K[0] - K[1])),
                                 K[1] - phi(tt[n + 1], uu[n] + h / 4 * (3 * K[0] + K[1]))
                                 ]
          K_start = phi(tt[n], uu[n])
          k1, k2 = fsolve(systeme, [K_start, K_start])
          uu[n+1] = uu[n]+h/4*(3*k1+k2)
    return uu
```

15.4. Schémas predicteur-correcteur

15.4.1. Schéma de Heun

$$\begin{cases} u_0 = y_0, \\ \tilde{u} = u_n + h\varphi_n \\ u_{n+1} = u_n + \frac{h}{2}\Big(\varphi_n + \varphi(t_{n+1}, \tilde{u})\Big) \end{cases} \quad n = 0, 1, 2, \dots N-1$$

```
def heun(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initial condition
    uu[0] = y0
    # time-stepping loop
    for n in range(len(tt)-1):
        k1 = phi( tt[n], uu[n] )
        k2 = phi( tt[n+1], uu[n] + h*k1  )
        uu[n+1] = uu[n] + (k1+k2)*h/2
    return uu
```

15.4.2. Schéma AM-2 AB-1

$$\begin{cases} u_0 = y_0, \\ u_1 = u_0 + h\varphi_0, \\ \tilde{u} = u_n + h\varphi_n, \\ u_{n+1} = u_n + \frac{h}{12}\Big(5\varphi(t_{n+1}, \tilde{u}) + 8\varphi_n - \varphi_{n-1}\Big) \end{cases} \quad n = 1, 2, \dots N-1$$

```
def AM2AB1(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initial condition
    uu[0] = y0
    uu[1] = uu[0]+h*phi(tt[0],uu[0])
    # time-stepping loop
    for n in range(1,len(tt)-1):
        k1 = phi( tt[n], uu[n] )
        k2 = phi( tt[n-1], uu[n-1] )
        pred = uu[n] + h*k1
        uu[n+1] = uu[n]+h*(5*phi(tt[n+1],pred)+8*k1-k2)/12
    return uu
```

15.4.3. Schéma AM-3 AB-2

$$\begin{cases} u_0 = y_0, \\ u_1 = u_0 + h\varphi_0, \\ u_2 = u_1 + \frac{h}{2}(3\varphi_1 - \varphi_0), \\ \tilde{u} = u_n + \frac{h}{2}(3\varphi_n - \varphi_{n-1}), \\ u_{n+1} = u_n + \frac{h}{24}\Big(9\varphi(t_{n+1}, \tilde{u}) + 19\varphi_n - 5\varphi_{n-1} + \varphi_{n-2}\Big) \end{cases} \quad n = 2, 3, \dots N-1$$

```
def AM3AB2(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initial condition
    uu[0] = y0
    uu[1] = uu[0]+h*phi(tt[0],uu[0])
    uu[2] = uu[1]+0.5*h*(3*phi(tt[1],uu[1])-phi(tt[0],uu[0]))
    # time-stepping loop
    for n in range(2,len(tt)-1):
        k1 = phi( tt[n], uu[n] )
        k2 = phi( tt[n-1], uu[n-1] )
        k3 = phi( tt[n-2], uu[n-2] )
        pred = uu[n] + (3*k1-k2)*h/2
        uu[n+1] = uu[n]+h*(9*phi(tt[n+1],pred)+19*k1-5*k2+k3)/24
    return uu
```

15.5. Solution exacte

Nous allons calculer la solution exacte en utilisant le module `sympy`.

```
t  = sp.Symbol('t')
y  = sp.Function('y')

edo = sp.Eq( sp.diff(y(t),t) , sp.sin(t)+y(t) ) # Attention : c'est le sinus de sympy
display(edo)

t0 = 0
y0 = 0
display(sp.Eq(y(t0),y0))

solpar = sp.dsolve(edo, y(t), ics={y(t0):y0})
display(solpar)
```

$$\frac{d}{dt}y(t) = y(t) + \sin(t)$$

$$y(0) = 0$$

$$y(t) = \frac{e^t}{2} - \frac{\sin(t)}{2} - \frac{\cos(t)}{2}$$

On définit la solution exacte à utiliser pour estimer les erreurs :

```
sol_exacte = sp.lambdify(t, solpar.rhs, 'numpy')
```

15.6. Tests

On initialise le problème de Cauchy et on définit l'équation différentielle : `phi` est une fonction python qui contient la fonction mathématique $\varphi\colon [t_0;T]\times\mathbb{R}\to\mathbb{R}$ définie per $\varphi(t,y)=\sin(t)+y$ dépendant des variables t et y.

On introduit la discrétisation : les $N+1$ nœuds d'intégration $[t_0,t_1,\dots,t_N]$ sont contenus dans le vecteur `tt` et sont espacés de $h=\frac{t_N-t_0}{N}$.

```
t0     = 0
tfinal = 1
y0 = 0

phi = lambda t,y : np.sin(t) + y # Attention : c'est le sinus de numpy
```

```
N = 8
tt = np.linspace(t0, tfinal, N+1)
```

On calcule les solutions exacte et approchées et on les compare.

Nous pouvons utiliser deux méthodes différentes pour calculer et afficher les solutions.

15.6.1. Méthode 1

La première méthode est celle utilisée lors des deux premiers TP : - on crée autant de liste que de solutions approchée - on compare les graphes des solutions exacte et approchées - on affiche le maximum de l'erreur

```
yy = sol_exacte(tt)

uu_EE     = EE(phi,tt,y0)
uu_AB2    = AB2(phi,tt,y0)
uu_AB3    = AB3(phi,tt,y0)
uu_AB4    = AB4(phi,tt,y0)
uu_AB5    = AB5(phi,tt,y0)
uu_N2     = N2(phi,tt,y0)
uu_N3     = N3(phi,tt,y0)
uu_N4     = N4(phi,tt,y0)
uu_EM     = EM(phi,tt,y0)
uu_RK4    = RK4(phi,tt,y0)
uu_RK6_5  = RK6_5(phi,tt,y0)
uu_RK7_6  = RK7_6(phi,tt,y0)

uu_EI   = EI(phi,tt,y0)
uu_CN   = CN(phi,tt,y0)
uu_AM2  = AM2(phi,tt,y0)
uu_AM3  = AM3(phi,tt,y0)
uu_AM4  = AM4(phi,tt,y0)
uu_AM5  = AM5(phi,tt,y0)
uu_BDF2 = BDF2(phi,tt,y0)
uu_BDF3 = BDF3(phi,tt,y0)
uu_RK1_M  = RK1_M(phi,tt,y0)
uu_RK1_M_bis = RK1_M_bis(phi,tt,y0)
uu_DIRK2  = DIRK2(phi,tt,y0)
uu_Randau = Randau(phi,tt,y0)

uu_heun   = heun(phi,tt,y0)
uu_AM2AB1 = AM2AB1(phi,tt,y0)
uu_AM3AB2 = AM3AB2(phi,tt,y0)
```

```
fig, ax = plt.subplots(nrows=9, ncols=3, figsize=(18,32))
ax = ax.reshape(-1)


i = 0
ax[i].plot(tt,yy,'b-',tt,uu_EE,'r-D')
err = np.linalg.norm(uu_EE-yy, np.inf)
ax[i].set_title(f'AB1=EE\nmax(|err|)={err:g}')

i += 1
ax[i].plot(tt,yy,'b-',tt,uu_AB2,'r-D')
err = np.linalg.norm(uu_AB2-yy, np.inf)
ax[i].set_title(f'AB2\nmax(|err|)={err:g}')

i += 1
```

```
ax[i].plot(tt,yy,'b-',tt,uu_AB3,'r-D')
err = np.linalg.norm(uu_AB3-yy, np.inf)
ax[i].set_title(f'AB3\nmax(|err|)={err:g}')

i += 1
ax[i].plot(tt,yy,'b-',tt,uu_AB4,'r-D')
err = np.linalg.norm(uu_AB4-yy, np.inf)
ax[i].set_title(f'AB4\nmax(|err|)={err:g}')

i += 1
ax[i].plot(tt,yy,'b-',tt,uu_AB5,'r-D')
err = np.linalg.norm(uu_AB5-yy, np.inf)
ax[i].set_title(f'AB5\nmax(|err|)={err:g}')

i += 1
ax[i].plot(tt,yy,'b-',tt,uu_N2,'r-D')
err = np.linalg.norm(uu_N2-yy, np.inf)
ax[i].set_title(f'N2\nmax(|err|)={err:g}')

i += 1
ax[i].plot(tt,yy,'b-',tt,uu_N3,'r-D')
err = np.linalg.norm(uu_N3-yy, np.inf)
ax[i].set_title(f'N3\nmax(|err|)={err:g}')

i += 1
ax[i].plot(tt,yy,'b-',tt,uu_N4,'r-D')
err = np.linalg.norm(uu_N4-yy, np.inf)
ax[i].set_title(f'N4\nmax(|err|)={err:g}')

i += 1
ax[i].plot(tt,yy,'b-',tt,uu_EM,'r-D')
err = np.linalg.norm(uu_EM-yy, np.inf)
ax[i].set_title(f'EM\nmax(|err|)={err:g}')

i += 1
ax[i].plot(tt,yy,'b-',tt,uu_RK4,'r-D')
err = np.linalg.norm(uu_RK4-yy, np.inf)
ax[i].set_title(f'RK4\nmax(|err|)={err:g}')

i += 1
ax[i].plot(tt,yy,'b-',tt,uu_RK6_5,'r-D')
err = np.linalg.norm(uu_RK6_5-yy, np.inf)
ax[i].set_title(f'RK6_5\nmax(|err|)={err:g}')

i += 1
ax[i].plot(tt,yy,'b-',tt,uu_RK7_6,'r-D')
err = np.linalg.norm(uu_RK7_6-yy, np.inf)
ax[i].set_title(f'RK7_6\nmax(|err|)={err:g}')

i += 1
ax[i].plot(tt,yy,'b-',tt,uu_EI,'r-D')
err = np.linalg.norm(uu_EI-yy, np.inf)
ax[i].set_title(f'AM0=EI\nmax(|err|)={err:g}')

i += 1
ax[i].plot(tt,yy,'b-',tt,uu_CN,'r-D')
err = np.linalg.norm(uu_CN-yy, np.inf)
ax[i].set_title(f'AM1=CN\nmax(|err|)={err:g}')

i += 1
ax[i].plot(tt,yy,'b-',tt,uu_AM2,'r-D')
err = np.linalg.norm(uu_AM2-yy, np.inf)
ax[i].set_title(f'AM2\nmax(|err|)={err:g}')
```

```
i += 1
ax[i].plot(tt,yy,'b-',tt,uu_AM3,'r-D')
err = np.linalg.norm(uu_AM3-yy, np.inf)
ax[i].set_title(f'AM3\nmax(|err|)={err:g}')

i += 1
ax[i].plot(tt,yy,'b-',tt,uu_AM4,'r-D')
err = np.linalg.norm(uu_AM4-yy, np.inf)
ax[i].set_title(f'AM4\nmax(|err|)={err:g}')

i += 1
ax[i].plot(tt,yy,'b-',tt,uu_AM5,'r-D')
err = np.linalg.norm(uu_AM5-yy, np.inf)
ax[i].set_title(f'AM5\nmax(|err|)={err:g}')

i += 1
ax[i].plot(tt,yy,'b-',tt,uu_BDF2,'r-D')
err = np.linalg.norm(uu_BDF2-yy, np.inf)
ax[i].set_title(f'BDF2\nmax(|err|)={err:g}')

i += 1
ax[i].plot(tt,yy,'b-',tt,uu_BDF3,'r-D')
err = np.linalg.norm(uu_BDF3-yy, np.inf)
ax[i].set_title(f'BDF3\nmax(|err|)={err:g}')

i += 1
ax[i].plot(tt,yy,'b-',tt,uu_RK1_M,'r-D')
err = np.linalg.norm(uu_RK1_M-yy, np.inf)
ax[i].set_title(f'RK1_M\nmax(|err|)={err:g}')

i += 1
ax[i].plot(tt,yy,'b-',tt,uu_RK1_M_bis,'r-D')
err = np.linalg.norm(uu_RK1_M_bis-yy, np.inf)
ax[i].set_title(f'RK1_M_bis\nmax(|err|)={err:g}')

i += 1
ax[i].plot(tt,yy,'b-',tt,uu_DIRK2,'r-D')
err = np.linalg.norm(uu_DIRK2-yy, np.inf)
ax[i].set_title(f'DIRK2\nmax(|err|)={err:g}')

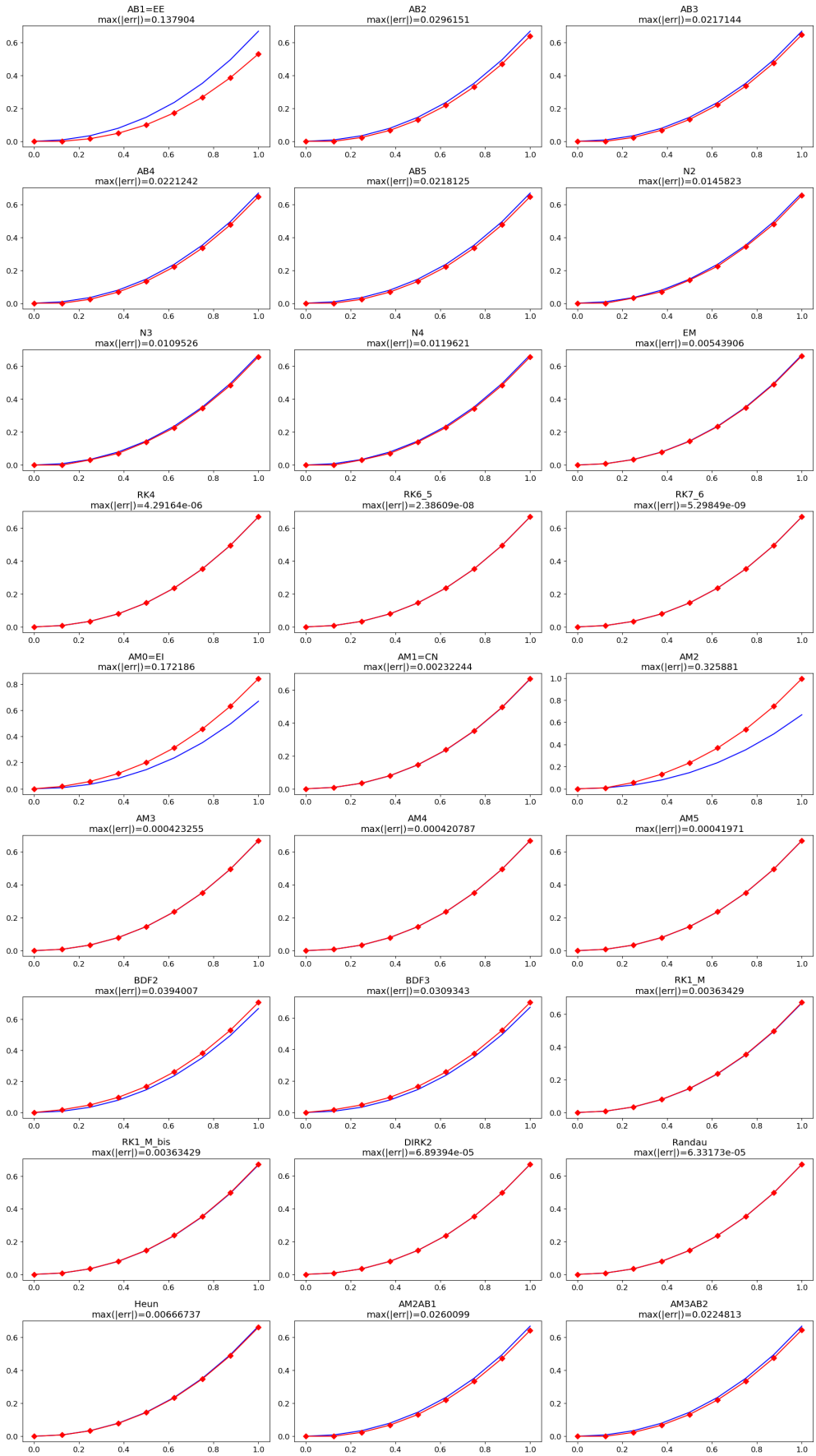
i += 1
ax[i].plot(tt,yy,'b-',tt,uu_Randau,'r-D')
err = np.linalg.norm(uu_Randau-yy, np.inf)
ax[i].set_title(f'Randau\nmax(|err|)={err:g}')

i += 1
ax[i].plot(tt,yy,'b-',tt,uu_heun,'r-D')
err = np.linalg.norm(uu_heun-yy, np.inf)
ax[i].set_title(f'Heun\nmax(|err|)={err:g}')

i += 1
ax[i].plot(tt,yy,'b-',tt,uu_AM2AB1,'r-D')
err = np.linalg.norm(uu_AM2AB1-yy, np.inf)
ax[i].set_title(f'AM2AB1\nmax(|err|)={err:g}')

i += 1
ax[i].plot(tt,yy,'b-',tt,uu_AM3AB2,'r-D')
err = np.linalg.norm(uu_AM3AB2-yy, np.inf)
ax[i].set_title(f'AM3AB2\nmax(|err|)={err:g}')


fig.tight_layout() ;
```


AB1=EE
max(|err|)=0.137904
AB2
max(|err|)=0.0296151
AB3
max(|err|)=0.0217144
AB4
max(|err|)=0.0221242
AB5
max(|err|)=0.0218125
N2
max(|err|)=0.0145823
N3
max(|err|)=0.0109526
N4
max(|err|)=0.0119621
EM
max(|err|)=0.00543906
RK4
max(|err|)=4.29164e-06
RK6_5
max(|err|)=2.38609e-08
RK7_6
max(|err|)=5.29849e-09
AM0=EI
max(|err|)=0.172186
AM1=CN
max(|err|)=0.00232244
AM2
max(|err|)=0.325881
AM3
max(|err|)=0.000423255
AM4
max(|err|)=0.000420787
AM5
max(|err|)=0.00041971
BDF2
max(|err|)=0.0394007
BDF3
max(|err|)=0.0309343
RK1_M
max(|err|)=0.00363429
RK1_M_bis
max(|err|)=0.00363429
DIRK2
max(|err|)=6.89394e-05
Randau
max(|err|)=6.33173e-05
Heun
max(|err|)=0.00666737
AM2AB1
max(|err|)=0.0260099
AM3AB2
max(|err|)=0.0224813

15.6.2. Méthode 2

La deuxième méthode fait appelle à la notion de dictionnaire, elle est compacte mais peut-être plus difficile à comprendre. On crée

- une liste avec les noms des schémas (sous forme de chaînes de caractères),
- un dictionnaire avec comme clé les noms des schémas et comme valeur l'array solution approchée
- un dictionnaire avec comme clé les noms des schémas et comme valeur le maximum de la valeur absolue de l'erreur.

Pour appeler les fonctions, on utilise `eval` qui permet d'évaluer une chaîne de caractères comme une commande python.

```
yy = sol_exacte(tt)

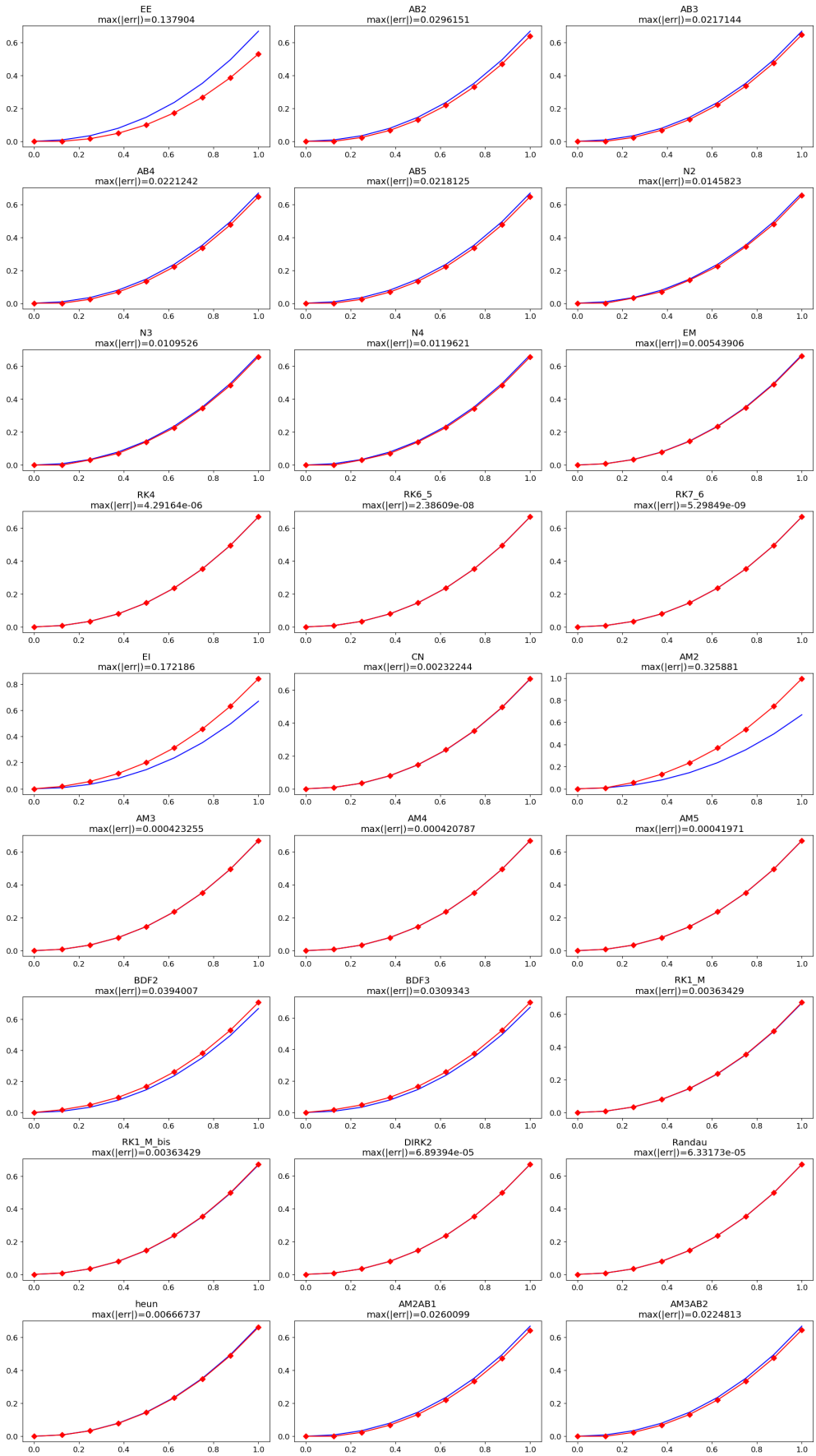
schemas = ['EE','AB2','AB3','AB4','AB5','N2','N3','N4', 'EM', 'RK4', 'RK6_5', 'RK7_6',
           'EI', 'CN', 'AM2', 'AM3', 'AM4', 'AM5', 'BDF2', 'BDF3', 'RK1_M', 'RK1_M_bis', 'DIRK2',
           ↪  'Randau',
           'heun', 'AM2AB1', 'AM3AB2']

uu = { s : eval(s)(phi,tt,y0) for s in schemas }
err= { s : np.linalg.norm(uu[s]-yy, np.inf) for s in schemas }

fig, ax = plt.subplots(nrows=9, ncols=3, figsize=(18, 32), constrained_layout=False)
ax = ax.reshape(-1)

idx = 0
for key in uu:
    ax[idx].plot(tt,yy,'b-',tt,uu[key],'r-D')
    ax[idx].set_title( f'{key}\nmax(|err|)={err[key]:g}' )
    idx += 1

fig.tight_layout()
```


EE
max(|err|)=0.137904
AB2
max(|err|)=0.0296151
AB3
max(|err|)=0.0217144
AB4
max(|err|)=0.0221242
AB5
max(|err|)=0.0218125
N2
max(|err|)=0.0145823
N3
max(|err|)=0.0109526
N4
max(|err|)=0.0119621
EM
max(|err|)=0.00543906
RK4
max(|err|)=4.29164e-06
RK6_5
max(|err|)=2.38609e-08
RK7_6
max(|err|)=5.29849e-09
EI
max(|err|)=0.172186
CN
max(|err|)=0.00232244
AM2
max(|err|)=0.325881
AM3
max(|err|)=0.000423255
AM4
max(|err|)=0.000420787
AM5
max(|err|)=0.00041971
BDF2
max(|err|)=0.0394007
BDF3
max(|err|)=0.0309343
RK1_M
max(|err|)=0.00363429
RK1_M_bis
max(|err|)=0.00363429
DIRK2
max(|err|)=6.89394e-05
Randau
max(|err|)=6.33173e-05
heun
max(|err|)=0.00666737
AM2AB1
max(|err|)=0.0260099
AM3AB2
max(|err|)=0.0224813

15.6.3. Méthode 2bis

La troisième méthode ressemble à la précédente sans utiliser `eval()`. On crée

- une liste avec les noms des schémas (il s'agit d'une liste de fonctions cette fois-ci),
- un dictionnaire avec comme clé les noms des schémas et comme valeur l'array solution approchée
- un dictionnaire avec comme clé les noms des schémas et comme valeur le maximum de la valeur absolue de l'erreur.

Pour récupérer le nom des fonctions, on utilise `.__name__`.

```
yy = sol_exacte(tt)

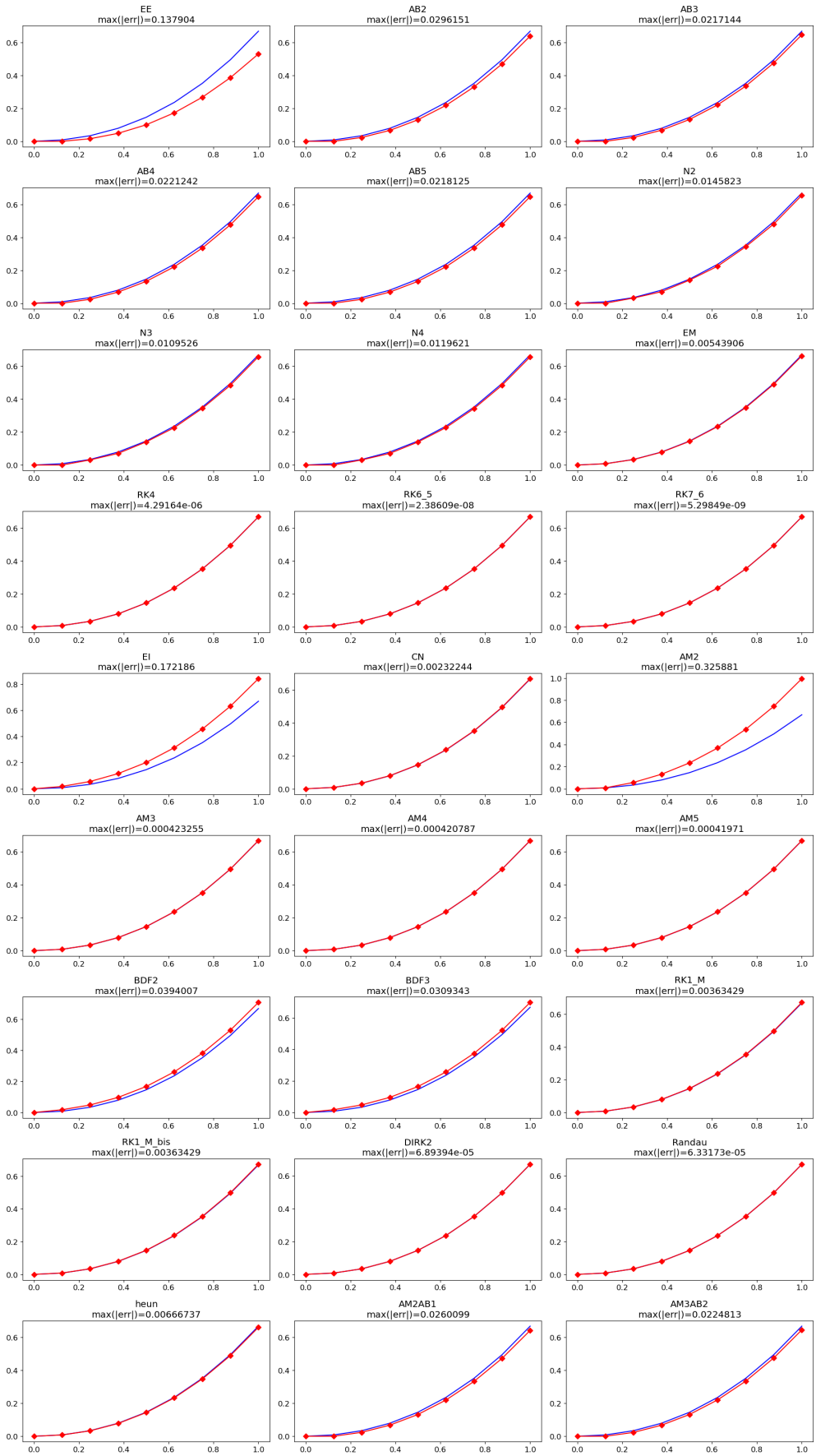

schemas = [EE, AB2, AB3, AB4, AB5, N2, N3, N4, EM, RK4, RK6_5, RK7_6,
           EI, CN, AM2, AM3, AM4, AM5, BDF2, BDF3, RK1_M, RK1_M_bis, DIRK2, Randau,
           heun, AM2AB1, AM3AB2]

uu = { s.__name__ : s(phi,tt,y0) for s in schemas }
err= { s.__name__ : np.linalg.norm(uu[s.__name__]-yy, np.inf) for s in schemas }

fig, ax = plt.subplots(nrows=9, ncols=3, figsize=(18, 32), constrained_layout=False)
ax = ax.reshape(-1)

idx = 0
for key in uu:
    ax[idx].plot(tt,yy,'b-',tt,uu[key],'r-D')
    ax[idx].set_title( f'{key}\nmax(|err|)={err[key]:g}' )
    idx += 1

fig.tight_layout()
```


EE
max(|err|)=0.137904
AB2
max(|err|)=0.0296151
AB3
max(|err|)=0.0217144
AB4
max(|err|)=0.0221242
AB5
max(|err|)=0.0218125
N2
max(|err|)=0.0145823
N3
max(|err|)=0.0109526
N4
max(|err|)=0.0119621
EM
max(|err|)=0.00543906
RK4
max(|err|)=4.29164e-06
RK6_5
max(|err|)=2.38609e-08
RK7_6
max(|err|)=5.29849e-09
EI
max(|err|)=0.172186
CN
max(|err|)=0.00232244
AM2
max(|err|)=0.325881
AM3
max(|err|)=0.000423255
AM4
max(|err|)=0.000420787
AM5
max(|err|)=0.00041971
BDF2
max(|err|)=0.0394007
BDF3
max(|err|)=0.0309343
RK1_M
max(|err|)=0.00363429
RK1_M_bis
max(|err|)=0.00363429
DIRK2
max(|err|)=6.89394e-05
Randau
max(|err|)=6.33173e-05
heun
max(|err|)=0.00666737
AM2AB1
max(|err|)=0.0260099
AM3AB2
max(|err|)=0.0224813

Chapitre 16.

TP 4 - Étude de la convergence

```
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import fsolve
plt.rcParams.update({'font.size': 12})
from IPython.display import display, Markdown
```

16.1. Exercice (à compléter) : Estimation de l'Ordre de Convergence des Schémas Numériques

Objectif : compléter l'implémentation des schémas numériques indiqués et estimer **l'ordre de convergence** de chaque schéma sur le problème de Cauchy

$$\begin{cases} y'(t) = y(t), \quad \forall t \in I = [0,1], \\ y(0) = 1. \end{cases}$$

dont la solution exacte est $y(t) = e^t$.

Résultats attendus :

1. Méthodes AB_q et N_q (explicites à q pas) : convergence à l'ordre q.
2. Méthodes AM_q et MS_q (implicites à q pas) : convergence à l'ordre $q+1$ (donc sub-optimale lorsque q est pair). Noter que $\mathrm{MS}_2 \equiv \mathrm{MS}_3$.
3. Méthodes BDF_q (implicites à q pas) : convergence à l'ordre q (donc sub-optimale quelle que soit la parité de q). Rappelle : elles sont zéro-stables pour $q \leq 6$ et instables pour $q \geq 7$.

De plus, on vérifiera que les autres méthodes ont les ordres de convergence suivants :

1. Méthodes predictor-corrector : convergence à l'ordre $\min(\omega_E + 1, \omega_I)$.
   - ω_E : ordre du schéma predictor (explicite),
   - ω_I : ordre du schéma corrector (implicite).
2. Méthodes de Runge-Kutta à s étages : convergence à l'ordre …
   - $\leq s$ si elles sont explicites,
   - $\leq 2s$ si elles sont implicites.

Remarque :

1. Puisque la fonction $\varphi(t, y) = y$ est *linéaire*, toute méthode *implicite* peut être rendue *explicite* par un calcul élémentaire.
   Cela consiste à expliciter directement, pour chaque schéma, l'expression de u_{n+1}. Cependant, nous pouvons utiliser le module `SciPy` pour résoudre ces équations *sans modifier l'implémentation des schémas*.
   Attention : l'utilisation de `fsolve` peut réduire l'ordre de convergence global au même ordre que celui de `fsolve` lui-même. Dans notre exemple ce n'est pas le cas car la fonction φ est linéaire, mais il est important de le garder à l'esprit.
2. **Initialisation des schémas multi-step :** les *schémas multi-step* nécessitent l'initialisation de plusieurs pas pour pouvoir démarrer. Plutôt que d'utiliser un schéma d'ordre inférieur pour l'initialisation (ce qui dégraderait l'ordre de précision global), **nous utiliserons la solution exacte**.

3. Comme d'habitude, on notera $\varphi_j = \varphi(t_j, u_j)$.
4. Selon la barrière de Dahlquist, les MPL à q pas sont au mieux
   - d'ordre q si explicite,
   - d'ordre $q+1$ si implicite et q impair,
   - d'ordre $q+2$ si implicite et q pair. C'est pour cela que les méthodes AM_q et MS_q sont sub-optimales lorsque q est pair, tandis que les méthodes BDF_q sont sub-optimales pour tout q.

Rappels de la démarche pour le calcul de l'orde de convergence.

- On écrit les schémas numériques :
  - les nœuds d'intégration $[t_0, t_1, \dots, t_N]$ sont contenus dans le vecteur `tt` (qui change en fonction de `h`)
  - les valeurs $[u_0, u_1, \dots, u_N]$ pour chaque méthode sont contenues dans le vecteur `uu`.
- Pour chaque schéma, on calcule la solution approchée avec différentes valeurs de $h_k = 1/N_k$. On sauvegarde les valeurs de h_k dans le vecteur `H`.
- Pour chaque valeur de h_k, on calcule le maximum de la valeur absolue de l'erreur et on sauvegarde toutes ces erreurs dans le vecteur `err_schema` de sort que `err_schema[k]` contient $e_k = \max_{i=0,\dots,N_k} |y(t_i) - u_i|$.
- Pour afficher l'ordre de convergence on utilise une échelle logarithmique, i.e. on représente $\ln(h)$ sur l'axe des abscisses et $\ln(\text{err})$ sur l'axe des ordonnées.
  En effet, si $\text{err} = Ch^p$ alors $\ln(\text{err}) = \ln(C) + p\ln(h)$.
  En échelle logarithmique, p représente donc la pente de la ligne droite $\ln(\text{err})$.

Nota Bene Le choix de la norme doit être cohérent avec la norme continue que l'on cherche à approcher.

Pour la norme infinie continue $\|e\|_{L^\infty} = \sup_{t\in[0,1]} |e(t)|$, la norme discrète naturelle est $\max_i |e_i|$, qui ne dépend pas du nombre de points du maillage.

En revanche, pour approcher la norme L^2 continue $\|e\|_{L^2} = \left(\int_0^1 e(t)^2\, dt\right)^{1/2}$, la norme euclidienne discrète $\|e\|_2 = \sqrt{\sum_i e_i^2}$ doit être multipliée par $\sqrt{h}$ (où h est le pas de temps), car l'intégrale est approchée par une somme de Riemann : $\int_0^1 e(t)^2 dt \approx h\sum_i e_i^2$. On obtient donc $\|e\|_{L^2} \approx \sqrt{h}\,\|e\|_2$. Sans ce facteur $\sqrt{h}$, la norme dépend artificiellement du nombre de points et l'ordre de convergence mesuré est incorrect.

16.2. Schémas explicites

16.2.1. Schéma EE

Schéma d'Adam-Bashforth à 1 pas = schéma d'Euler progressif

$$\begin{cases} u_0 = y_0, \\ u_{n+1} = u_n + h\varphi_n \quad n = 0, 1, 2, \dots N-1 \end{cases}$$

```
def EE(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    uu[0] = sol_exacte(tt[0])
    # récurrence
    for n in range(len(tt)-1):
        k1 = phi(tt[n],uu[n])
        uu[n+1] = uu[n]+h*k1
    return uu
```

16.2.2. Schéma AB-2

Schéma d'Adam-Bashforth à 2 pas

$$\begin{cases} u_0 = y_0, \\ u_1 = y_1, \\ u_{n+1} = u_n + \frac{h}{2}\Big(3\varphi_n - \varphi_{n-1}\Big) \quad n = 1, 2, 3, 4, 5, \dots N-1 \end{cases}$$

```
def AB2(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    q = 2
    uu[:q] = sol_exacte(tt[:q]) # on initialise uu[0] et uu[1]
    # récurrence
    for n in range(q-1,len(tt)-1):
        k1 = phi( tt[n], uu[n] )
        k2 = phi( tt[n-1], uu[n-1] )
        uu[n+1] = uu[n] + (3*k1-k2)*h/2
    return uu
```

16.2.3. Schéma AB-3

Schéma d'Adam-Bashforth à 3 pas

$$\begin{cases} u_0 = y_0, \\ u_1 = y_1, \\ u_2 = y_2, \\ u_{n+1} = u_n + \frac{h}{12}\Big(23\varphi_n - 16\varphi_{n-1} + 5\varphi_{n-2}\Big) \quad n = 2, 3, 4, 5, \dots N-1 \end{cases}$$

```
def AB3(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    q = 3
    uu[:q] = sol_exacte(tt[:q])
    # récurrence
    for n in range(q-1,len(tt)-1):
        k1 = phi( tt[n], uu[n] )
        k2 = phi( tt[n-1], uu[n-1] )
        k3 = phi( tt[n-2], uu[n-2] )
        uu[n+1] = uu[n] + (23*k1-16*k2+5*k3)*h/12
    return uu
```

16.2.4. Schéma AB-4

Schéma d'Adam-Bashforth à 4 pas

$$\begin{cases} u_0 = y_0, \\ u_1 = y_1, \\ u_2 = y_2, \\ u_3 = y_3, \\ u_{n+1} = u_n + \frac{h}{24}\Big(55\varphi_n - 59\varphi_{n-1} + 37\varphi_{n-2} - 9\varphi_{n-3}\Big) \quad n = 3, 4, 5, \dots N-1 \end{cases}$$

```
def AB4(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    q = 4
    uu[:q] = sol_exacte(tt[:q])
    # récurrence
    for n in range(q-1,len(tt)-1):
        k1 = phi( tt[n], uu[n] )
        k2 = phi( tt[n-1], uu[n-1] )
        k3 = phi( tt[n-2], uu[n-2] )
        k4 = phi( tt[n-3], uu[n-3] )
        uu[n+1] = uu[n] + (55*k1-59*k2+37*k3-9*k4)*h/24
    return uu
```

16.2.5. Schéma AB-5

Schéma d'Adam-Bashforth à 5 pas

$$\begin{cases} u_0 = y_0, \\ u_1 = y_1, \\ u_2 = y_2, \\ u_3 = y_3, \\ u_4 = y_4, \\ u_{n+1} = u_n + \frac{h}{720}\left(1901\varphi_n - 2774\varphi_{n-1} + 2616\varphi_{n-2} - 1274\varphi_{n-3} + 251\varphi_{n-4}\right) \quad n = 4, 5, \dots N-1 \end{cases}$$

```
def AB5(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    q = 5
    uu[:q] = sol_exacte(tt[:q])
    # récurrence
    for n in range(q-1,len(tt)-1):
        k1 = phi( tt[n], uu[n] )
        k2 = phi( tt[n-1], uu[n-1] )
        k3 = phi( tt[n-2], uu[n-2] )
        k4 = phi( tt[n-3], uu[n-3] )
        k5 = phi( tt[n-4], uu[n-4] )
        uu[n+1] = uu[n] + (1901*k1-2774*k2+2616*k3-1274*k4+251*k5)*h/720
    return uu
```

16.2.6. Schéma N-2

Schéma de Nylström à 2 pas

$$\begin{cases} u_0 = y_0, \\ u_1 = y_1, \\ u_{n+1} = u_{n-1} + 2h\varphi_n \quad n = 1, 2, 3, 4, 5, \dots N-1 \end{cases}$$

```
def N2(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    q = 2
```

```
    uu[:q] = sol_exacte(tt[:q])
    # récurrence
    for n in range(q-1,len(tt)-1):
        k1 = phi( tt[n], uu[n] )
        uu[n+1] = uu[n-1] + 2*h*k1
    return uu
```

16.2.7. Schéma N-3

Schéma de Nylström à 3 pas

$$\begin{cases} u_0 = y_0, \\ u_1 = y_1, \\ u_2 = y_2, \\ u_{n+1} = u_{n-1} + \frac{h}{3}\Big(7\varphi_n - 2\varphi_{n-1} + \varphi_{n-2}\Big) \quad n = 2, 3, 4, 5, \dots N-1 \end{cases}$$

```
def N3(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    q = 3
    uu[:q] = sol_exacte(tt[:q])
    # récurrence
    for n in range(q-1,len(tt)-1):
        k1 = phi( tt[n], uu[n] )
        k2 = phi( tt[n-1], uu[n-1] )
        k3 = phi( tt[n-2], uu[n-2] )
        uu[n+1] = uu[n-1] + (7*k1-2*k2+k3)*h/3
    return uu
```

16.2.8. Schéma N-4

Schéma de Nylström à 4 pas

$$\begin{cases} u_0 = y_0, \\ u_1 = y_1, \\ u_2 = y_2, \\ u_3 = y_3, \\ u_{n+1} = u_{n-1} + \frac{h}{3}\Big(8\varphi_n - 5\varphi_{n-1} + 4\varphi_{n-2} - \varphi_{n-3}\Big) \quad n = 3, 4, 5, \dots N-1 \end{cases}$$

```
def N4(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    q = 4
    uu[:q] = sol_exacte(tt[:q])
    # récurrence
    for n in range(q-1,len(tt)-1):
        k1 = phi( tt[n], uu[n] )
        k2 = phi( tt[n-1], uu[n-1] )
        k3 = phi( tt[n-2], uu[n-2] )
        k4 = phi( tt[n-3], uu[n-3] )
        uu[n+1] = uu[n-1] + (8*k1-5*k2+4*k3-k4)*h/3
    return uu
```

16.2.9. Schéma EM

Schéma d'Euler modifié (RK explicite à 2 pas)

$$\begin{cases} u_0 = y_0, \\ \tilde{u} = u_n + \frac{h}{2}\varphi_n, \\ u_{n+1} = u_n + h\varphi\left(t_n + \frac{h}{2}, \tilde{u}\right) \end{cases} \quad n = 0, 1, 2, \dots N-1$$

```
def EM(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    uu[0] = sol_exacte(tt[0])
    # récurrence
    for n in range(len(tt)-1):
        k1 = phi( tt[n], uu[n] )
        uu[n+1] = uu[n]+h*phi(tt[n]+h/2,uu[n]+k1*h/2)
    return uu
```

16.2.10. Schéma RK4-1

Schéma de Runge-Kutta RK4-1

$$\begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n, u_n\right) \\ K_2 = \varphi\left(t_n + \frac{h}{2}, u_n + \frac{h}{2}K_1\right)\right) \\ K_3 = \varphi\left(t_n + \frac{h}{2}, u_n + \frac{h}{2}K_2\right) \\ K_4 = \varphi\left(t_{n+1}, u_n + hK_3\right) \\ u_{n+1} = u_n + \frac{h}{6}\left(K_1 + 2K_2 + 2K_3 + K_4\right) \end{cases} \quad n = 0, 1, \dots N-1$$

```
def RK4(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    uu[0] = sol_exacte(tt[0])
    # récurrence
    for n in range(len(tt)-1):
        k1 = phi( tt[n], uu[n] )
        k2 = phi( tt[n]+h/2 , uu[n]+h*k1/2 )
        k3 = phi( tt[n]+h/2 , uu[n]+h*k2/2 )
        k4 = phi( tt[n+1]    , uu[n]+h*k3 )
        uu[n+1] = uu[n] + (k1+2*k2+2*k3+k4)*h/6
    return uu
```

16.2.11. Schéma RK6_5

Schéma de Runge-Kutta RK6_5 (Butcher à 6 étages d'ordre 5)

$$\begin{array}{c|cccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{4} & \frac{1}{8} & \frac{1}{8} & 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & -\frac{1}{2} & 1 & 0 & 0 & 0 \\ \frac{3}{4} & \frac{3}{16} & 0 & 0 & \frac{9}{16} & 0 & 0 \\ 1 & \frac{-3}{7} & \frac{2}{7} & \frac{12}{7} & \frac{-12}{7} & \frac{8}{7} & 0 \\ \hline & \frac{7}{90} & 0 & \frac{32}{90} & \frac{12}{90} & \frac{32}{90} & \frac{7}{90} \end{array}$$

```
def RK6_5(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    uu[0] = sol_exacte(tt[0])
    # récurrence
    for n in range(len(tt)-1):
        k1 = phi( tt[n]      , uu[n] )
        k2 = phi( tt[n]+h/4  , uu[n]+h*k1/4 )
        k3 = phi( tt[n]+h/4  , uu[n]+h*(k1+k2)/8 )
        k4 = phi( tt[n]+h/2  , uu[n]+h*(-k2+2*k3)/2 )
        k5 = phi( tt[n]+h*3/4, uu[n]+h*(3*k1+9*k4)/16 )
        k6 = phi( tt[n+1]    , uu[n]+h*(-3*k1+2*k2+12*k3-12*k4+8*k5)/7 )
        uu[n+1] = uu[n] + (7*k1+32*k3+12*k4+32*k5+7*k6)*h/90
    return uu
```

16.2.12. Schéma RK7_6

Schéma de Runge-Kutta RK7_6 (Butcher à 7 étages d'ordre 6)

0	0	0	0	0	0	0	0
$\frac{1}{2}$	$\frac{1}{2}$	0	0	0	0	0	0
$\frac{2}{3}$	$\frac{2}{9}$	$\frac{4}{9}$	0	0	0	0	0
$\frac{1}{3}$	$\frac{7}{36}$	$\frac{2}{9}$	$-\frac{1}{12}$	0	0	0	0
$\frac{5}{6}$	$-\frac{35}{144}$	$-\frac{55}{36}$	$\frac{35}{48}$	$\frac{15}{8}$	0	0	0
$\frac{1}{6}$	$-\frac{1}{360}$	$-\frac{11}{36}$	$-\frac{1}{8}$	$\frac{1}{2}$	$\frac{1}{10}$	0	0
1	$\frac{-41}{260}$	$\frac{22}{13}$	$\frac{43}{156}$	$-\frac{118}{39}$	$\frac{32}{195}$	$\frac{80}{39}$	0
	$\frac{13}{200}$	0	$\frac{11}{40}$	$\frac{11}{40}$	$\frac{4}{25}$	$\frac{4}{25}$	$\frac{13}{200}$

```
def RK7_6(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    uu[0] = sol_exacte(tt[0])
    # récurrence
    for n in range(len(tt)-1):
        k1 = phi( tt[n]      , uu[n] )
        k2 = phi( tt[n]+h/2  , uu[n]+h*(        k1
 ↪  )/2 )
        k3 = phi( tt[n]+h*2/3, uu[n]+h*(      2*k1     +4*k2
 ↪  )/9 )
        k4 = phi( tt[n]+h/3  , uu[n]+h*(      7*k1     +8*k2      -3*k3
 ↪  )/36 )
        k5 = phi( tt[n]+h*5/6, uu[n]+h*(    -35*k1   -220*k2    +105*k3    +270*k4
 ↪  )/144 )
        k6 = phi( tt[n]+h/6  , uu[n]+h*(       -k1   -110*k2     -45*k3    +180*k4     +36*k5
 ↪  )/360 )
        k7 = phi( tt[n+1]    , uu[n]+h*(-41/260*k1 +22/13*k2 +43/156*k3 -118/39*k4 +32/195*k5 +80/39*k6)
 ↪  )
        uu[n+1] = uu[n] + (13/200*k1+11/40*k3+11/40*k4+4/25*k5+4/25*k6+13/200*k7)*h
    return uu
```

16.3. Schémas implicites

16.3.1. Schéma EI

Schéma d'Euler régressif

$$\begin{cases} u_0 = y_0, \\ u_{n+1} = u_n + h\varphi(t_{n+1}, u_{n+1}) \quad n = 0, 1, 2, \dots N-1 \end{cases}$$

avec u_{n+1} zéro de la fonction

$$x \mapsto -x + u_n + h\varphi(t_{n+1}, x).$$

```
def EI(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    uu[0] = sol_exacte(tt[0])
    # récurrence
    for n in range(len(tt)-1):
        temp = fsolve(lambda x: -x+uu[n]+h*phi(tt[n+1],x), uu[n])
        uu[n+1] = temp[0]
    return uu
```

16.3.2. Schéma CN

Schéma de Crank-Nicolson

$$\begin{cases} u_0 = y_0, \\ u_{n+1} = u_n + \frac{h}{2}\Big(\varphi_n + \varphi(t_{n+1}, u_{n+1})\Big) \quad n = 0, 1, 2, \dots N-1 \end{cases}$$

avec u_{n+1} un zéro de la fonction

$$x \mapsto -x + u_n + \frac{h}{2}(\varphi_n + \varphi(t_{n+1}, x)).$$

```
def CN(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    uu[0] = sol_exacte(tt[0])
    # récurrence
    for n in range(len(tt)-1):
        temp = fsolve(lambda x: -x+uu[n]+0.5*h*( phi(tt[n+1],x)+phi(tt[n],uu[n]) ), uu[n])
        uu[n+1] = temp[0]
    return uu
```

16.3.3. Schéma AM-2

$$\begin{cases} u_0 = y_0, \\ u_1 = y_1, \\ u_{n+1} = u_n + \frac{h}{12}\Big(5\varphi(t_{n+1}, u_{n+1}) + 8\varphi_n - \varphi_{n-1}\Big) \quad n = 1, 2, \dots N-1 \end{cases}$$

avec u_{n+1} un zéro de la fonction

$$x \mapsto -x + u_n + \frac{h}{12}\Big(5\varphi(t_{n+1}, x) + 8\varphi_n - \varphi_{n-1}\Big).$$

```
def AM2(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    q = 2
    uu[:q] = sol_exacte(tt[:q]) # on initialise uu[0] et uu[1]
    # récurrence
    for n in range(q-1,len(tt)-1):
        temp = fsolve(lambda x: -x+uu[n]+h*( 5*phi(tt[n+1],x)+8*phi(tt[n],uu[n])-phi(tt[n-1],uu[n-1])
↪   )/12, uu[n])
        uu[n+1] = temp[0]
    return uu
```

16.3.4. Schéma AM-3

$$\begin{cases} u_0 = y_0, \\ u_1 = y_1, \\ u_2 = y_2, \\ u_{n+1} = u_n + \frac{h}{24}\Big(9\varphi(t_{n+1}, u_{n+1}) + 19\varphi_n - 5\varphi_{n-1} + \varphi_{n-2}\Big) \quad n = 2, 3, \dots N-1 \end{cases}$$

avec u_{n+1} un zéro de la fonction

$$x \mapsto -x + u_n + \frac{h}{24}\Big(9\varphi(t_{n+1}, x) + 19\varphi_n - 5\varphi_{n-1} + \varphi_{n-2}\Big).$$

```
def AM3(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    q = 3
    uu[:q] = sol_exacte(tt[:q])
    # récurrence
    for n in range(q-1,len(tt)-1):
        temp = fsolve(lambda x: -x+uu[n]+h*(
↪   9*phi(tt[n+1],x)+19*phi(tt[n],uu[n])-5*phi(tt[n-1],uu[n-1])+phi(tt[n-2],uu[n-2]) )/24, uu[n])
        uu[n+1] = temp[0]
    return uu
```

16.3.5. Schéma AM-4

$$\begin{cases} u_0 = y_0, \\ u_1 = y_1, \\ u_2 = y_2, \\ u_3 = y_3, \\ u_{n+1} = u_n + \frac{h}{720}\Big(251\varphi(t_{n+1}, u_{n+1}) + 646\varphi_n - 264\varphi_{n-1} + 106\varphi_{n-2} - 19\varphi_{n-3}\Big) \quad n = 3, 4, \dots N-1 \end{cases}$$

avec u_{n+1} un zéro de la fonction

$$x \mapsto -x + u_n + \frac{h}{720}\Big(251\varphi(t_{n+1}, x) + 646\varphi_n - 264\varphi_{n-1} + 106\varphi_{n-2} - 19\varphi_{n-3}\Big).$$

```
def AM4(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    q = 4
    uu[:q] = sol_exacte(tt[:q])
    # récurrence
    for n in range(q-1,len(tt)-1):
        temp = fsolve(lambda x: -x+uu[n]+h*(
↪   251*phi(tt[n+1],x)+646*phi(tt[n],uu[n])-264*phi(tt[n-1],uu[n-1])+106*phi(tt[n-2],uu[n-2])-19*phi(tt[n-3],uu[n
↪   )/720, uu[n])
        uu[n+1] = temp[0]
    return uu
```

16.3.6. Schéma AM-5

$$\begin{cases} u_0 = y_0, \\ u_1 = y_1, \\ u_2 = y_2, \\ u_3 = y_3, \\ u_4 = y_4, \\ u_{n+1} = u_n + \frac{h}{1440}\Big(475\varphi(t_{n+1}, u_{+1}) + 1427\varphi_n - 798\varphi_{n-1} + 482\varphi_{n-2} - 173\varphi_{n-3} + 27\varphi_{n-4}\Big), \quad n = 4, 5, \dots N-1 \end{cases}$$

avec u_{n+1} un zéro de la fonction

$$x \mapsto -x + u_n + \frac{h}{1440}\Big(475\varphi(t_{n+1}, x) + 1427\varphi_n - 798\varphi_{n-1} + 482\varphi_{n-2} - 173\varphi_{n-3} + 27\varphi_{n-4}\Big).$$

```
def AM5(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    q = 5
    uu[:q] = sol_exacte(tt[:q])
    # récurrence
    for n in range(q-1,len(tt)-1):
        temp = fsolve(lambda x: -x+uu[n]+h*(
↪   475*phi(tt[n+1],x)+1427*phi(tt[n],uu[n])-798*phi(tt[n-1],uu[n-1])+482*phi(tt[n-2],uu[n-2])-173*phi(tt[n-3],uu
↪   uu[n])
        uu[n+1] = temp[0]
    return uu
```

16.3.7. Schéma MS-2 = MS-3

$$\begin{cases} u_0 = y_0, \\ u_1 = y_1, \\ u_{n+1} = u_{n-1} + \frac{h}{3}\Big(\varphi(t_{n+1}, u_{n+1}) + 4\varphi_n + \varphi_{n-1}\Big) \quad n = 1, 2, \dots N-1 \end{cases}$$

avec u_{n+1} un zéro de la fonction

$$x \mapsto -x + u_{n-1} + \frac{h}{3}\Big(\varphi(t_{n+1}, x) + 4\varphi_n + \varphi_{n-1}\Big).$$

```
def MS2(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    q = 2
    uu[:q] = sol_exacte(tt[:q])
    # récurrence
    for n in range(q-1,len(tt)-1):
        k0 = phi( tt[n], uu[n] )
        k1 = phi( tt[n-1], uu[n-1] )
        temp = fsolve(lambda x: -x+uu[n-1]+h*(phi(tt[n+1],x)+4*k0+k1)/3, uu[n])
        uu[n+1] = temp[0]
    return uu
```

16.3.8. Schéma MS-4

$$\begin{cases} u_0 = y_0, \\ u_1 = y_1, \\ u_2 = y_2, \\ u_3 = y_3, \\ u_{n+1} = u_{n-1} + \frac{h}{90}\Big(29\varphi(t_{n+1}, u_{n+1}) + 124\varphi_n + 24\varphi_{n-1} + 4\varphi_{n-2} - \varphi_{n-3}\Big) \quad n = 3, 4, \dots N-1 \end{cases}$$

avec u_{n+1} un zéro de la fonction

$$x \mapsto -x + u_{n-1} + \frac{h}{90}\Big(29\varphi(t_{n+1}, x) + 124\varphi_n + 24\varphi_{n-1} + 4\varphi_{n-2} - \varphi_{n-3}\Big).$$

```
def MS4(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    q = 4
    uu[:q] = sol_exacte(tt[:q])
    # récurrence
    for n in range(q-1,len(tt)-1):
        k0 = phi( tt[n], uu[n] )
        k1 = phi( tt[n-1], uu[n-1] )
        k2 = phi( tt[n-2], uu[n-2] )
        k3 = phi( tt[n-3], uu[n-3] )
        temp = fsolve(lambda x: -x+uu[n-1]+h*(29*phi(tt[n+1],x)+124*k0+24*k1+4*k2-k3)/90, uu[n])
        uu[n+1] = temp[0]
    return uu
```

16.3.9. Schéma MS-5

$$\begin{cases} u_0 = y_0, \\ u_1 = y_1, \\ u_2 = y_2, \\ u_3 = y_3, \\ u_4 = y_4, \\ u_{n+1} = u_{n-1} + \frac{h}{90}\Big(28\varphi(t_{n+1}, u_{n+1}) + 129\varphi_n + 14\varphi_{n-1} + 14\varphi_{n-2} - 6\varphi_{n-3} + \varphi_{n-4}\Big) \quad n = 4, 5, \dots N-1 \end{cases}$$

avec u_{n+1} un zéro de la fonction

$$x \mapsto -x + u_{n-1} + \frac{h}{90}\Big(28\varphi(t_{n+1},x) + 129\varphi_n + 14\varphi_{n-1} + 14\varphi_{n-2} - 6\varphi_{n-3} + \varphi_{n-4}\Big).$$

```
def MS5(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    q = 5
    uu[:q] = sol_exacte(tt[:q])
    # récurrence
    for n in range(q-1,len(tt)-1):
        k0 = phi( tt[n], uu[n] )
        k1 = phi( tt[n-1], uu[n-1] )
        k2 = phi( tt[n-2], uu[n-2] )
        k3 = phi( tt[n-3], uu[n-3] )
        k4 = phi( tt[n-4], uu[n-4] )
        temp = fsolve(lambda x: -x+uu[n-1]+h*(28*phi(tt[n+1],x)+129*k0+14*k1+14*k2-6*k3+k4)/90, uu[n])
        uu[n+1] = temp[0]
    return uu
```

16.3.10. Schéma BDF2

$$\begin{cases} u_0 = y_0, \\ u_1 = y_1, \\ u_{n+1} = \frac{4}{3}u_n - \frac{1}{3}u_{n-1} + \frac{2}{3}h\varphi(t_{n+1},u_{n+1}) \quad n = 1,2,\dots N-1 \end{cases}$$

avec u_{n+1} un zéro de la fonction

$$x \mapsto -x + \frac{4}{3}u_n - \frac{1}{3}u_{n-1} + \frac{2}{3}h\varphi(t_{n+1},x).$$

```
def BDF2(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    q = 2
    uu[:q] = sol_exacte(tt[:q])
    # récurrence
    for n in range(q-1,len(tt)-1):
        temp = fsolve(lambda x: -x+4/3*uu[n]-1/3*uu[n-1] + 2/3*h*phi(tt[n+1],x) , uu[n])
        uu[n+1] = temp[0]
    return uu
```

16.3.11. Schéma BDF3

$$\begin{cases} u_0 = y_0, \\ u_1 = y_1, \\ u_2 = y_2, \\ u_{n+1} = \frac{18}{11}u_n - \frac{9}{11}u_{n-1} + \frac{2}{11}u_{n-2} + \frac{6}{11}h\varphi(t_{n+1},u_{n+1}) \quad n = 2,3,\dots N-1 \end{cases}$$

avec u_{n+1} un zéro de la fonction

$$x \mapsto -x + \frac{18}{11}u_n - \frac{9}{11}u_{n-1} + \frac{2}{11}u_{n-2} + \frac{6}{11}h\varphi(t_{n+1},x).$$

```
def BDF3(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    q = 3
    uu[:q] = sol_exacte(tt[:q])
    # récurrence
    for n in range(q-1,len(tt)-1):
        temp = fsolve(lambda x: -x+18/11*uu[n]-9/11*uu[n-1] + 2/11*uu[n-2]+6/11*h*phi(tt[n+1],x) , uu[n])
        uu[n+1] = temp[0]
    return uu
```

16.3.12. Schéma BDF4

$$\begin{cases} u_0 = y_0, \\ u_1 = y_1, \\ u_2 = y_2, \\ u_3 = y_3, \\ u_{n+1} = \frac{48}{25}u_n - \frac{36}{25}u_{n-1} + \frac{16}{25}u_{n-2} - \frac{3}{25}u_{n-3} + \frac{12}{25}h\varphi(t_{n+1}, u_{n+1}) \quad n = 3, 4, \dots N-1 \end{cases}$$

avec u_{n+1} un zéro de la fonction

$$x \mapsto -x + \frac{48}{25}u_n - \frac{36}{25}u_{n-1} + \frac{16}{25}u_{n-2} - \frac{3}{25}u_{n-3} + \frac{12}{25}h\varphi(t_{n+1}, x).$$

```
def BDF4(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    q = 4
    uu[:q] = sol_exacte(tt[:q])
    # récurrence
    for n in range(q-1,len(tt)-1):
        temp = fsolve(lambda x: -x+48/25*uu[n]-36/25*uu[n-1] +
↪   16/25*uu[n-2]-3/25*uu[n-3]+12/25*h*phi(tt[n+1],x) , uu[n])
        uu[n+1] = temp[0]
    return uu
```

16.3.13. Schéma BDF5

$$\begin{cases} u_0 = y_0, \\ u_1 = y_1, \\ u_2 = y_2, \\ u_3 = y_3, \\ u_4 = y_4, \\ u_5 = y_5, \\ u_{n+1} = \frac{300}{137}u_n - \frac{300}{137}u_{n-1} + \frac{200}{137}u_{n-2} - \frac{75}{137}u_{n-3} + \frac{12}{137}u_{n-4} + \frac{60}{137}h\varphi(t_{n+1}, u_{n+1}) \quad n = 5, 6, \dots N-1 \end{cases}$$

avec u_{n+1} un zéro de la fonction

$$x \mapsto -x + \frac{300}{137}u_n - \frac{300}{137}u_{n-1} + \frac{200}{137}u_{n-2} - \frac{75}{137}u_{n-3} + \frac{12}{137}u_{n-4} + \frac{60}{137}h\varphi(t_{n+1}, x).$$

```
def BDF5(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    q = 5
    uu[:q] = sol_exacte(tt[:q])
    # récurrence
    for n in range(q-1,len(tt)-1):
        temp = fsolve(lambda x: -x+300/137*uu[n]-300/137*uu[n-1] +
↪   200/137*uu[n-2]-75/137*uu[n-3]+12/137*uu[n-4]+60/137*h*phi(tt[n+1],x) , uu[n])
        uu[n+1] = temp[0]
    return uu
```

16.3.14. Schéma RK1_M

$$\begin{cases} u_0 = y_0, \\ u_{n+1} = u_n + h\varphi\left(t_n + \frac{h}{2}, \frac{u_n+u_{n+1}}{2}\right) \quad n = 0, 1, 2, \dots N-1 \end{cases}$$

avec u_{n+1} un zéro de la fonction

$$x \mapsto -x + u_n + h\varphi\left(t_n + \frac{h}{2}, \frac{u_n + x}{2}\right).$$

Rappelons-nous que ce schéma est bien un schéma de Runge-Kutta :

$$\begin{array}{c|c} \frac{1}{2} & \frac{1}{2} \\ \hline & 1 \end{array} \quad \Longrightarrow \quad \begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n + \frac{h}{2}, u_n + \frac{h}{2}K_1\right) \\ u_{n+1} = u_n + hK_1 \end{cases} \qquad n = 0, 1, \dots N-1$$

avec K_1 un zéro de la fonction

$$x \mapsto -x + \varphi\left(t_n + \frac{h}{2}, u_n + \frac{h}{2}x\right).$$

```
# Version "classique"
# def RK1_M(phi,tt,sol_exacte):
#     h  = tt[1]-tt[0]
#     uu = np.empty_like(tt)
#     # initialisation
#     uu[0] = sol_exacte(tt[0])
#     # récurrence
#     for n in range(len(tt)-1):
#         uu[n+1] = fsolve(lambda x: -x+uu[n]+h*phi( (tt[n]+tt[n+1])/2,(uu[n]+x)/2 ), uu[n])[0]
#     return uu

# Version "Runge-Kutta"
def RK1_M(phi,tt,sol_exacte):
    h  = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    uu[0] = sol_exacte(tt[0])
    # récurrence
    for n in range(len(tt)-1):
        k1 = fsolve( lambda x : -x + phi(tt[n]+h/2, uu[n]+h*x/2), phi(tt[n],uu[n]) )[0]
        uu[n+1] = uu[n]+h*k1
    return uu
```

16.3.15. Schéma Randau-IIa-2

$$\begin{array}{c|cc} \frac{1}{3} & \frac{5}{12} & -\frac{1}{12} \\ 1 & \frac{3}{4} & \frac{1}{4} \\ \hline & \frac{3}{4} & \frac{1}{4} \end{array} \quad \Longrightarrow \quad \begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n + \frac{1}{3}h, u_n + h(\frac{5}{12}K_1 - \frac{1}{12}K_2)\right) \\ K_2 = \varphi\left(t_{n+1}, u_n + h(\frac{3}{4}K_1 + \frac{1}{4}K_2)\right) \\ u_{n+1} = u_n + \frac{h}{4}(3K_1 + K_2) \end{cases} \qquad n = 0, 1, \dots N-1$$

avec $[K_1, K_2]$ un zéro de la fonction vectorielle

$$(x_1, x_2) \mapsto \begin{pmatrix} -x_1 + \varphi\left(t_n + \frac{h}{3}, u_n + h(\frac{5}{12}x_1 - \frac{1}{12}x_2)\right) \\ -x_2 + \varphi\left(t_n + \frac{h}{3}, u_n + h(\frac{3}{4}x_1 - \frac{1}{4}x_2)\right) \end{pmatrix}$$

.

```
def Randau_IIa_2(phi,tt,sol_exacte):
    h  = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    uu[0] = sol_exacte(tt[0])
    # récurrence
    for n in range(len(tt)-1):
        systeme = lambda KK : [ -KK[0] + phi( tt[n]+h/3, uu[n]+h/12*(5*KK[0]-KK[1]) ),
                                -KK[1] + phi( tt[n+1]   , uu[n]+ h/4*(3*KK[0]+KK[1]) ) ]
        k_init = phi(tt[n],uu[n])
        k1,k2 = fsolve( systeme, (k_init,k_init) )
        uu[n+1] = uu[n]+h/4*(3*k1+k2)
    return uu
```

16.4. Schémas predicteur-correcteur

16.4.1. Schéma de Heun (AM-1 AB-1)

Le schéma de Heun est à la fois un schéma prédicteur-correcteur et un schéma de Runge-Kutta.

Comme predictor-corrector, il utilise le schéma d'Euler explicite (AB-1) pour prédire la valeur de u_{n+1} et le schéma de Crank-Nicolson (AM-1) pour corriger cette valeur :

$$\begin{cases} u_0 = y_0, \\ \tilde{u} = u_n + h\varphi_n \\ u_{n+1} = u_n + \frac{h}{2}\Big(\varphi_n + \varphi(t_{n+1}, \tilde{u})\Big) \end{cases} \quad n = 0, 1, 2, \dots N-1$$

Comme schéma de Runge-Kutta, il s'écrit comme

$$\begin{cases} u_0 = y_0, \\ k_1 = \varphi_n, \\ k_2 = \varphi(t_{n+1}, u_n + hk_1) \\ u_{n+1} = u_n + \frac{h}{2}\Big(k_1 + k_2\Big) \end{cases} \quad n = 0, 1, 2, \dots N-1$$

dont la matrice de Buthcher est

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

```
def heun(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    uu[0] = sol_exacte(tt[0])
    # récurrence
    for n in range(len(tt)-1):
        k1 = phi( tt[n], uu[n] )
        k2 = phi( tt[n+1], uu[n] + k1*h  )
        uu[n+1] = uu[n] + (k1+k2)*h/2
    return uu
```

16.4.2. Schéma AM-4 AB-2/3/4/5

- Predictor : Adam-Bashforth à 2, 3, 4 et 5 pas respectivement
- Corrector : Adam-Moulton à 4 pas

```
def AM4AB2(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    uu[:4] = sol_exacte(tt[:4])
    # récurrence
    for n in range(3,len(tt)-1):
        k1 = phi( tt[n], uu[n] )
        k2 = phi( tt[n-1], uu[n-1] )
        pred = uu[n] + (3*k1-k2)*h/2
        uu[n+1] = uu[n]+h*(
↪   251*phi(tt[n+1],pred)+646*phi(tt[n],uu[n])-264*phi(tt[n-1],uu[n-1])+106*phi(tt[n-2],uu[n-2])-19*phi(tt[n-3],u
↪   )/720
    return uu
```

```
def AM4AB3(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    uu[:4] = sol_exacte(tt[:4])
    # récurrence
    for n in range(3,len(tt)-1):
        k1 = phi( tt[n], uu[n] )
        k2 = phi( tt[n-1], uu[n-1] )
        k3 = phi( tt[n-2], uu[n-2] )
        pred = uu[n] + (23*k1-16*k2+5*k3)*h/12
        uu[n+1] = uu[n]+h*(
↪   251*phi(tt[n+1],pred)+646*phi(tt[n],uu[n])-264*phi(tt[n-1],uu[n-1])+106*phi(tt[n-2],uu[n-2])-19*phi(tt[n-3],u
↪   )/720
    return uu
```

```
def AM4AB4(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    uu[:4] = sol_exacte(tt[:4])
    # récurrence
    for n in range(3,len(tt)-1):
        k1 = phi( tt[n], uu[n] )
        k2 = phi( tt[n-1], uu[n-1] )
        k3 = phi( tt[n-2], uu[n-2] )
        k4 = phi( tt[n-3], uu[n-3] )
        pred = uu[n] + (55*k1-59*k2+37*k3-9*k4)*h/24
```

```
        uu[n+1] = uu[n]+h*(
↪   251*phi(tt[n+1],pred)+646*phi(tt[n],uu[n])-264*phi(tt[n-1],uu[n-1])+106*phi(tt[n-2],uu[n-2])-19*phi(tt[n-3],uu[n-3])
↪   )/720
    return uu
```

```
def AM4AB5(phi,tt,sol_exacte):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    uu[:5] = sol_exacte(tt[:5])
    # récurrence
    for n in range(4,len(tt)-1):
        k1 = phi( tt[n], uu[n] )
        k2 = phi( tt[n-1], uu[n-1] )
        k3 = phi( tt[n-2], uu[n-2] )
        k4 = phi( tt[n-3], uu[n-3] )
        k5 = phi( tt[n-4], uu[n-4] )
        pred = uu[n] + (1901*k1-2774*k2+2616*k3-1274*k4+251*k5)*h/720
        uu[n+1] = uu[n]+h*(
↪   251*phi(tt[n+1],pred)+646*phi(tt[n],uu[n])-264*phi(tt[n-1],uu[n-1])+106*phi(tt[n-2],uu[n-2])-19*phi(tt[n-3],uu[n-3])
↪   )/720
    return uu
```

16.5. Convergence

On initialise le problème de Cauchy, on définit l'équation différentielle (`phi` est une fonction python qui contient la fonction mathématique $\varphi(t, y)$ dépendant des variables t et y) et enfin on définit la solution exacte :

```
t0, y0, tfinal = 0, 1, 3

# CHOIX DU CAS TEST

sol_exacte = lambda t: np.exp(t)
phi = lambda t,y: y

# Autres cas tests
# ========================================
# sol_exacte = lambda t: np.exp(-t)
# phi = lambda t,y: -y

# sol_exacte = lambda t: np.sqrt(2*t+1)
# phi = lambda t,y: 1/y

# sol_exacte = lambda t: np.sqrt(t**2+1)
# phi = lambda t,y: t/y

# sol_exacte = lambda t: 1/np.sqrt(1-2*t)
# phi = lambda t,y: y**3
```

Pour chaque schéma, on calcule la solution approchée avec différentes valeurs de $h_k = 1/N_k$; on sauvegarde les valeurs de h_k dans le vecteur `H`.

Pour chaque valeur de h_k, on calcule le maximum de la valeur absolue de l'erreur et on sauvegarde toutes ces erreurs dans le vecteur `err_schema` de sort que `err_schema[k]` contient $e_k = \max_{i=0,\dots,N_k} |y(t_i) - u_i|$.

Nous pouvons utiliser deux méthodes différentes pour stocker ces informations.

16.5.1. Méthode 1

La première méthode est celle utilisée lors des deux premiers TP : on crée autant de listes que de solutions approchées.

```
H = []

err_ep   = []
err_AB2  = []
err_AB3  = []
err_AB4  = []
err_AB5  = []
err_N2   = []
err_N3   = []
err_N4   = []
err_RK4  = []
err_RK6_5  = []
err_RK7_6  = []

err_er   = []
err_CN   = []
err_AM2  = []
err_AM3  = []
err_AM4  = []
err_AM5  = []
err_BDF2 = []
err_BDF3 = []
err_BDF4 = []
err_BDF5 = []
err_MS2=[]
err_MS4=[]
err_MS5=[]
err_RK1_M = []
err_Randau_IIa_2 = []

err_em   = []
err_heun = []
err_AM4AB2  = []
err_AM4AB3  = []
err_AM4AB4  = []
err_AM4AB5  = []

N = 10
for k in range(6):
    N += 50 #2**(k+3)
    tt = np.linspace(t0,tfinal,N+1)
    h = tt[1]-tt[0]
    H.append(h)
    yy = np.array([sol_exacte(t) for t in tt])

    # schemas explicites
    uu_ep   = EE(phi,tt,sol_exacte)
    uu_AB2  = AB2(phi,tt,sol_exacte)
    uu_AB3  = AB3(phi,tt,sol_exacte)
    uu_AB4  = AB4(phi,tt,sol_exacte)
    uu_AB5  = AB5(phi,tt,sol_exacte)
    uu_N2   = N2(phi,tt,sol_exacte)
    uu_N3   = N3(phi,tt,sol_exacte)
    uu_N4   = N4(phi,tt,sol_exacte)
    uu_em   = EM(phi,tt,sol_exacte)
    uu_RK4  = RK4(phi,tt,sol_exacte)
    uu_RK6_5  = RK6_5(phi,tt,sol_exacte)
    uu_RK7_6  = RK7_6(phi,tt,sol_exacte)
```

```
    # schemas implicites
    uu_er   = EI(phi,tt,sol_exacte)
    uu_CN   = CN(phi,tt,sol_exacte)
    uu_AM2  = AM2(phi,tt,sol_exacte)
    uu_AM3  = AM3(phi,tt,sol_exacte)
    uu_AM4  = AM4(phi,tt,sol_exacte)
    uu_AM5  = AM5(phi,tt,sol_exacte)
    uu_BDF2 = BDF2(phi,tt,sol_exacte)
    uu_BDF3 = BDF3(phi,tt,sol_exacte)
    uu_BDF4 = BDF4(phi,tt,sol_exacte)
    uu_BDF5 = BDF5(phi,tt,sol_exacte)
    uu_MS2  = MS2(phi,tt,sol_exacte)
    uu_MS4  = MS4(phi,tt,sol_exacte)
    uu_MS5  = MS5(phi,tt,sol_exacte)
    uu_RK1_M = RK1_M(phi,tt,sol_exacte)
    uu_Randau_IIa_2 = Randau_IIa_2(phi,tt,sol_exacte)
    # schemas predictor-corrector
    uu_heun   = heun(phi,tt,sol_exacte)
    uu_AM4AB2 = AM4AB2(phi,tt,sol_exacte)
    uu_AM4AB3 = AM4AB3(phi,tt,sol_exacte)
    uu_AM4AB4 = AM4AB4(phi,tt,sol_exacte)
    uu_AM4AB5 = AM4AB5(phi,tt,sol_exacte)

    # erreurs
    err_ep.append(np.linalg.norm(uu_ep-yy,np.inf))
    err_AB2.append(np.linalg.norm(uu_AB2-yy,np.inf))
    err_AB3.append(np.linalg.norm(uu_AB3-yy,np.inf))
    err_AB4.append(np.linalg.norm(uu_AB4-yy,np.inf))
    err_AB5.append(np.linalg.norm(uu_AB5-yy,np.inf))
    err_N2.append(np.linalg.norm(uu_N2-yy,np.inf))
    err_N3.append(np.linalg.norm(uu_N3-yy,np.inf))
    err_N4.append(np.linalg.norm(uu_N4-yy,np.inf))
    err_em.append(np.linalg.norm(uu_em-yy,np.inf))
    err_RK4.append(np.linalg.norm(uu_RK4-yy,np.inf))
    err_RK6_5.append(np.linalg.norm(uu_RK6_5-yy,np.inf))
    err_RK7_6.append(np.linalg.norm(uu_RK7_6-yy,np.inf))

    err_er.append(np.linalg.norm(uu_er-yy,np.inf))
    err_CN.append(np.linalg.norm(uu_CN-yy,np.inf))
    err_AM2.append(np.linalg.norm(uu_AM2-yy,np.inf))
    err_AM3.append(np.linalg.norm(uu_AM3-yy,np.inf))
    err_AM4.append(np.linalg.norm(uu_AM4-yy,np.inf))
    err_AM5.append(np.linalg.norm(uu_AM5-yy,np.inf))
    err_BDF2.append(np.linalg.norm(uu_BDF2-yy,np.inf))
    err_BDF3.append(np.linalg.norm(uu_BDF3-yy,np.inf))
    err_BDF4.append(np.linalg.norm(uu_BDF4-yy,np.inf))
    err_BDF5.append(np.linalg.norm(uu_BDF5-yy,np.inf))
    err_MS2.append(np.linalg.norm(uu_MS2-yy,np.inf))
    err_MS4.append(np.linalg.norm(uu_MS4-yy,np.inf))
    err_MS5.append(np.linalg.norm(uu_MS5-yy,np.inf))
    err_RK1_M.append(np.linalg.norm(uu_RK1_M-yy,np.inf))
    err_Randau_IIa_2.append(np.linalg.norm(uu_Randau_IIa_2-yy,np.inf))

    err_heun.append(np.linalg.norm(uu_heun-yy,np.inf))
    err_AM4AB2.append(np.linalg.norm(uu_AM4AB2-yy,np.inf))
    err_AM4AB3.append(np.linalg.norm(uu_AM4AB3-yy,np.inf))
    err_AM4AB4.append(np.linalg.norm(uu_AM4AB4-yy,np.inf))
    err_AM4AB5.append(np.linalg.norm(uu_AM4AB5-yy,np.inf))
```

Pour estimer l'ordre de convergence on estime la pente de la droite qui relie l'erreur au pas k à l'erreur au pas $k+1$ en echelle logarithmique en utilisant la fonction `polyfit` basée sur la régression linéaire.

```
# print (f'EE    {np.polyfit(np.log(H),np.log(err_ep),    1)[0]:1.2f}' )
# print (f'AB2   {np.polyfit(np.log(H),np.log(err_AB2),   1)[0]:1.2f}'  )
# print (f'AB3   {np.polyfit(np.log(H),np.log(err_AB3),   1)[0]:1.2f}' )
# print (f'AB4   {np.polyfit(np.log(H),np.log(err_AB4),   1)[0]:1.2f}' )
# print (f'AB5   {np.polyfit(np.log(H),np.log(err_AB5),   1)[0]:1.2f}' )
# print (f'N2    {np.polyfit(np.log(H),np.log(err_N2),    1)[0]:1.2f}' )
# print (f'N3    {np.polyfit(np.log(H),np.log(err_N3),    1)[0]:1.2f}' )
# print (f'N4    {np.polyfit(np.log(H),np.log(err_N4),    1)[0]:1.2f}' )
# print (f'EM    {np.polyfit(np.log(H),np.log(err_em),    1)[0]:1.2f}' )
# print (f'RK4   {np.polyfit(np.log(H),np.log(err_RK4),   1)[0]:1.2f}' )
# print (f'RK6_5 {np.polyfit(np.log(H),np.log(err_RK6_5), 1)[0]:1.2f}' )
# print (f'RK7_6 {np.polyfit(np.log(H),np.log(err_RK7_6), 1)[0]:1.2f}' )
# print('\n')
# print (f'EI   {np.polyfit(np.log(H),np.log(err_er),   1)[0]:1.2f}' )
# print (f'CN   {np.polyfit(np.log(H),np.log(err_CN),   1)[0]:1.2f}' )
# print (f'AM2  {np.polyfit(np.log(H),np.log(err_AM2),  1)[0]:1.2f}' )
# print (f'AM3  {np.polyfit(np.log(H),np.log(err_AM3),  1)[0]:1.2f}' )
# print (f'AM4  {np.polyfit(np.log(H),np.log(err_AM4),  1)[0]:1.2f}' )
# print (f'AM5  {np.polyfit(np.log(H),np.log(err_AM5),  1)[0]:1.2f}' )
# print (f'BDF2 {np.polyfit(np.log(H),np.log(err_BDF2), 1)[0]:1.2f}' )
# print (f'BDF3 {np.polyfit(np.log(H),np.log(err_BDF3), 1)[0]:1.2f}' )
# print (f'BDF4 {np.polyfit(np.log(H),np.log(err_BDF4), 1)[0]:1.2f}' )
# print (f'MS2  {np.polyfit(np.log(H),np.log(err_MS2),  1)[0]:1.2f}' )
# print (f'MS4  {np.polyfit(np.log(H),np.log(err_MS4),  1)[0]:1.2f}' )
# print (f'MS5  {np.polyfit(np.log(H),np.log(err_MS5),  1)[0]:1.2f}' )
# print (f'RK1_M {np.polyfit(np.log(H),np.log(err_RK1_M),  1)[0]:1.2f}' )
# print (f'Randau_IIa_2 {np.polyfit(np.log(H),np.log(err_Randau_IIa_2),  1)[0]:1.2f}' )
# print('\n')
# print (f'Heun    {np.polyfit(np.log(H),np.log(err_heun),   1)[0]:1.2f}' )
# print (f'AM4_AB2 {np.polyfit(np.log(H),np.log(err_AM4AB2), 1)[0]:1.2f}' )
# print (f'AM4_AB3 {np.polyfit(np.log(H),np.log(err_AM4AB3), 1)[0]:1.2f}' )
# print (f'AM4_AB4 {np.polyfit(np.log(H),np.log(err_AM4AB4), 1)[0]:1.2f}' )
# print (f'AM4_AB5 {np.polyfit(np.log(H),np.log(err_AM4AB5), 1)[0]:1.2f}' )
```

Pour afficher l'ordre de convergence on utilise une échelle logarithmique : on représente $\ln(h)$ sur l'axe des abscisses et $\ln(\text{err})$ sur l'axe des ordonnées. Le but de cette représentation est clair : si $\text{err} = Ch^p$ alors $\ln(\text{err}) = \ln(C) + p\ln(h)$. En échelle logarithmique, p représente donc la pente de la ligne droite $\ln(\text{err})$.

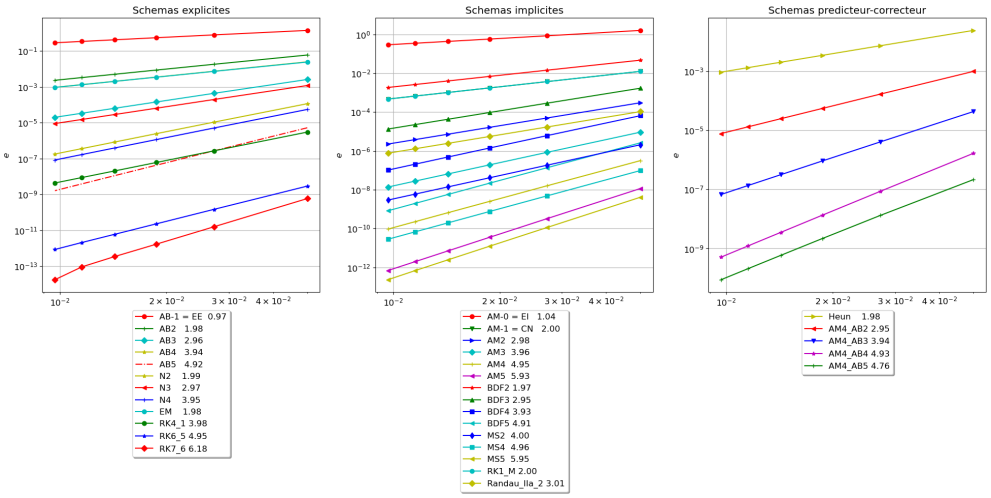
```
plt.figure(figsize=(24,7))

plt.subplot(1,3,1)
plt.loglog(H,err_ep,    'r-o',label=f'AB-1 = EE  {np.polyfit(np.log(H),np.log(err_ep),    1)[0]:1.2f}')
plt.loglog(H,err_AB2,   'g-+',label=f'AB2   {np.polyfit(np.log(H),np.log(err_AB2),   1)[0]:1.2f}')
plt.loglog(H,err_AB3,   'c-D',label=f'AB3   {np.polyfit(np.log(H),np.log(err_AB3),   1)[0]:1.2f}')
plt.loglog(H,err_AB4,   'y-*',label=f'AB4   {np.polyfit(np.log(H),np.log(err_AB4),   1)[0]:1.2f}')
plt.loglog(H,err_AB5,   'r-.',label=f'AB5   {np.polyfit(np.log(H),np.log(err_AB5),   1)[0]:1.2f}')
plt.loglog(H,err_N2,    'y-*',label=f'N2    {np.polyfit(np.log(H),np.log(err_N2),    1)[0]:1.2f}')
plt.loglog(H,err_N3,    'r-<',label=f'N3    {np.polyfit(np.log(H),np.log(err_N3),    1)[0]:1.2f}')
plt.loglog(H,err_N4,    'b-+',label=f'N4    {np.polyfit(np.log(H),np.log(err_N4),    1)[0]:1.2f}')
plt.loglog(H,err_em,    'c-o',label=f'EM    {np.polyfit(np.log(H),np.log(err_em),    1)[0]:1.2f}')
plt.loglog(H,err_RK4,   'g-o',label=f'RK4_1 {np.polyfit(np.log(H),np.log(err_RK4),   1)[0]:1.2f}')
plt.loglog(H,err_RK6_5, 'b-*',label=f'RK6_5 {np.polyfit(np.log(H),np.log(err_RK6_5), 1)[0]:1.2f}')
plt.loglog(H,err_RK7_6, 'r-D',label=f'RK7_6 {np.polyfit(np.log(H),np.log(err_RK7_6), 1)[0]:1.2f}')
plt.xlabel('$h$')
plt.ylabel('$e$')
plt.title("Schemas explicites")
plt.legend(loc='upper center', bbox_to_anchor=(0.5, -0.05), fancybox=True, shadow=True, ncol=1)
plt.grid(True)

plt.subplot(1,3,2)
plt.loglog(H,err_er,  'r-o',label=f'AM-0 = EI   {np.polyfit(np.log(H),np.log(err_er),   1)[0]:1.2f}')
plt.loglog(H,err_CN,  'g-v',label=f'AM-1 = CN   {np.polyfit(np.log(H),np.log(err_CN),   1)[0]:1.2f}')
```

```
plt.loglog(H,err_AM2, 'b->',label=f'AM2  {np.polyfit(np.log(H),np.log(err_AM2),  1)[0]:1.2f}')
plt.loglog(H,err_AM3, 'c-D',label=f'AM3  {np.polyfit(np.log(H),np.log(err_AM3),  1)[0]:1.2f}')
plt.loglog(H,err_AM4, 'y-+',label=f'AM4  {np.polyfit(np.log(H),np.log(err_AM4),  1)[0]:1.2f}')
plt.loglog(H,err_AM5, 'm-<',label=f'AM5  {np.polyfit(np.log(H),np.log(err_AM5),  1)[0]:1.2f}')
plt.loglog(H,err_BDF2,'r-*',label=f'BDF2 {np.polyfit(np.log(H),np.log(err_BDF2), 1)[0]:1.2f}')
plt.loglog(H,err_BDF3,'g-^',label=f'BDF3 {np.polyfit(np.log(H),np.log(err_BDF3), 1)[0]:1.2f}')
plt.loglog(H,err_BDF4,'b-s',label=f'BDF4 {np.polyfit(np.log(H),np.log(err_BDF4), 1)[0]:1.2f}' )
plt.loglog(H,err_BDF5,'c-<',label=f'BDF5 {np.polyfit(np.log(H),np.log(err_BDF5), 1)[0]:1.2f}' )
plt.loglog(H,err_MS2, 'b-d',label=f'MS2  {np.polyfit(np.log(H),np.log(err_MS2),  1)[0]:1.2f}')
plt.loglog(H,err_MS4, 'c-s',label=f'MS4  {np.polyfit(np.log(H),np.log(err_MS4),  1)[0]:1.2f}')
plt.loglog(H,err_MS5, 'y-<',label=f'MS5  {np.polyfit(np.log(H),np.log(err_MS5),  1)[0]:1.2f}')
plt.loglog(H,err_RK1_M, 'c-o',label=f'RK1_M {np.polyfit(np.log(H),np.log(err_RK1_M),  1)[0]:1.2f}')
plt.loglog(H,err_Randau_IIa_2, 'y-D',label=f'Randau_IIa_2 {np.polyfit(np.log(H),np.log(err_Randau_IIa_2),
 ↪  1)[0]:1.2f}')
plt.xlabel('$h$')
plt.ylabel('$e$')
plt.title("Schemas implicites")
plt.legend(loc='upper center', bbox_to_anchor=(0.5, -0.05), fancybox=True, shadow=True, ncol=1)
plt.grid(True)

plt.subplot(1,3,3)
plt.loglog(H,err_heun,   'y->',label=f'Heun    {np.polyfit(np.log(H),np.log(err_heun),   1)[0]:1.2f}')
plt.loglog(H,err_AM4AB2, 'r-<',label=f'AM4_AB2 {np.polyfit(np.log(H),np.log(err_AM4AB2), 1)[0]:1.2f}')
plt.loglog(H,err_AM4AB3, 'b-v',label=f'AM4_AB3 {np.polyfit(np.log(H),np.log(err_AM4AB3), 1)[0]:1.2f}' )
plt.loglog(H,err_AM4AB4, 'm-*',label=f'AM4_AB4 {np.polyfit(np.log(H),np.log(err_AM4AB4), 1)[0]:1.2f}')
plt.loglog(H,err_AM4AB5, 'g-+',label=f'AM4_AB5 {np.polyfit(np.log(H),np.log(err_AM4AB5), 1)[0]:1.2f}' )
plt.xlabel('$h$')
plt.ylabel('$e$')
plt.title("Schemas predicteur-correcteur")
plt.legend(loc='upper center', bbox_to_anchor=(0.5, -0.05), fancybox=True, shadow=True, ncol=1)
plt.grid(True)
```



Si on note

- $\omega_{[C]}$ l'ordre du corrector (schéma implicite)
- $\omega_{[P]}$ l'ordre du predictor (schéma explicite)

alors l'ordre du schéma predictor-corrector est

$$\omega_{[PC]} = \min\{\omega_{[C]}, \omega_{[P]} + 1\}$$

Pour les schémas AM4-ABx, le schéma corrector est d'ordre $p = 5$, pour ne pas perdre en ordre convergence il faut choisir un schéma predictor d'ordre $p - 1$ (ici AB4).

16.5.2. Méthode 2 : version compacte

De manière compacte en utilisant une liste des noms des schémas et des dictionnaires :

```
schemasEXPL = ['EE','AB2','AB3','AB4','AB5','N2','N3','N4', 'EM', 'RK4', 'RK6_5', 'RK7_6']
schemasIMPL = ['EI', 'CN', 'AM2', 'AM3', 'AM4', 'AM5', 'BDF2', 'BDF3', 'BDF4', 'BDF5', 'MS2', 'MS4',
 ↪  'MS5', 'RK1_M', 'Randau_IIa_2']
schemasPRCO = ['heun', 'AM4AB2', 'AM4AB3', 'AM4AB4', 'AM4AB5']

schemas = schemasEXPL + schemasIMPL + schemasPRCO

H = []
err = { schemas[s] : [] for s in range(len(schemas)) }

N = 10
for k in range(6):
    N += 50
    tt = np.linspace(t0, tfinal, N+1)
    h = tt[1]-tt[0]
    H.append(h)
    yy = np.array([sol_exacte(t) for t in tt])
    uu = { s : eval(s)(phi, tt, sol_exacte) for s in schemas }
    for key in uu:
        err[key].append(np.linalg.norm(uu[key]-yy,np.inf))

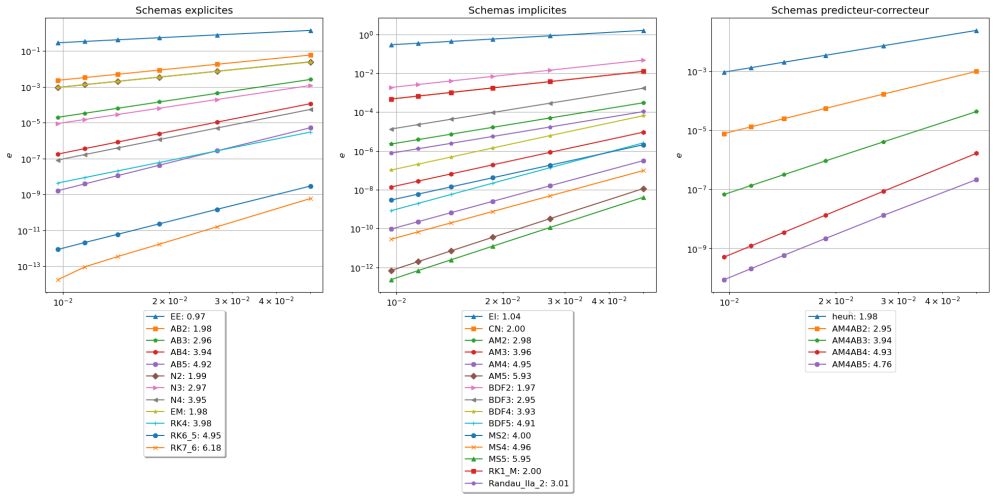
ordre = {  key : np.polyfit(np.log(H),np.log(err[key]),1)[0] for key in err  }


plt.figure(figsize=(24,7))
markers=['^', 's', 'p', 'h', '8', 'D', '>', '<', '*','+','o', 'x']

plt.subplot(1,3,1)
for idx,s in enumerate(schemasEXPL):
    plt.loglog(H,err[s], label=f'{s}: {ordre[s]:1.2f}',marker=markers[idx%len(markers)])
plt.xlabel('$h$')
plt.ylabel('$e$')
plt.title("Schemas explicites")
plt.legend(loc='upper center', bbox_to_anchor=(0.5, -0.05), fancybox=True, shadow=True, ncol=1)
plt.grid(True)

plt.subplot(1,3,2)
for idx,s in enumerate(schemasIMPL):
    plt.loglog(H,err[s], label=f'{s}: {ordre[s]:1.2f}',marker=markers[idx%len(markers)])
plt.xlabel('$h$')
plt.ylabel('$e$')
plt.title("Schemas implicites")
plt.legend(loc='upper center', bbox_to_anchor=(0.5, -0.05), fancybox=True, shadow=True, ncol=1)
plt.grid(True)

plt.subplot(1,3,3)
for idx,s in enumerate(schemasPRCO):
    plt.loglog(H,err[s], label=f'{s}: {ordre[s]:1.2f}',marker=markers[idx%len(markers)])
plt.xlabel('$h$')
plt.ylabel('$e$')
plt.title("Schemas predicteur-correcteur")
plt.legend(loc='upper center', bbox_to_anchor=(0.5, -0.05), fancybox=True, ncol=1)
plt.grid(True);
```



16.5.3. Méthode 2 bis : version compacte

Idem sans utiliser `eval` :

```
schemasEXPL = [EE, AB2, AB3, AB4, AB5, N2, N3, N4, EM, RK4, RK6_5, RK7_6]
schemasIMPL = [EI, CN, AM2, AM3, AM4, AM5, BDF2, BDF3, BDF4, BDF5, MS2, MS4, MS5, RK1_M, Randau_IIa_2]
schemasPRCO = [heun, AM4AB2, AM4AB3, AM4AB4, AM4AB5]

schemas = schemasEXPL + schemasIMPL + schemasPRCO

H = []
err = { sfct.__name__ : [] for sfct in schemas }

N = 10
for k in range(6):
    N += 50
    tt = np.linspace(t0, tfinal, N+1)
    h = tt[1]-tt[0]
    H.append(h)
    yy = np.array([sol_exacte(t) for t in tt])
    uu = { sfct.__name__ : sfct(phi, tt, sol_exacte) for sfct in schemas }
    for key in uu:
        err[key].append(np.linalg.norm(uu[key]-yy,np.inf))

ordre = {  key : np.polyfit(np.log(H),np.log(err[key]),1)[0] for key in err  }

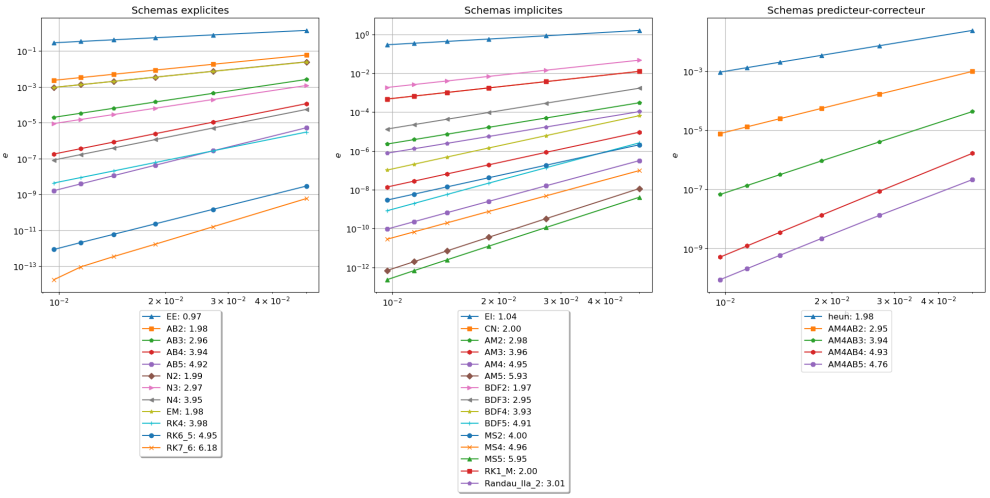

plt.figure(figsize=(24,7))
markers=['^', 's', 'p', 'h', '8', 'D', '>', '<', '*','+','o', 'x']

plt.subplot(1,3,1)
for idx,sfct in enumerate(schemasEXPL):
    s = sfct.__name__
    plt.loglog(H,err[s], label=f'{s}: {ordre[s]:1.2f}',marker=markers[idx%len(markers)])
plt.xlabel('$h$')
plt.ylabel('$e$')
plt.title("Schemas explicites")
plt.legend(loc='upper center', bbox_to_anchor=(0.5, -0.05), fancybox=True, shadow=True, ncol=1)
plt.grid(True)

plt.subplot(1,3,2)
for idx,sfct in enumerate(schemasIMPL):
    s = sfct.__name__
    plt.loglog(H,err[s], label=f'{s}: {ordre[s]:1.2f}',marker=markers[idx%len(markers)])
```

```
plt.xlabel('$h$')
plt.ylabel('$e$')
plt.title("Schemas implicites")
plt.legend(loc='upper center', bbox_to_anchor=(0.5, -0.05), fancybox=True, shadow=True, ncol=1)
plt.grid(True)

plt.subplot(1,3,3)
for idx,sfct in enumerate(schemasPRCO):
    s = sfct.__name__
    plt.loglog(H,err[s], label=f'{s}: {ordre[s]:1.2f}',marker=markers[idx%len(markers)])
plt.xlabel('$h$')
plt.ylabel('$e$')
plt.title("Schemas predicteur-correcteur")
plt.legend(loc='upper center', bbox_to_anchor=(0.5, -0.05), fancybox=True, ncol=1)
plt.grid(True);
```



Quatrième partie

TD

Chapitre 17.

TD1 - Exercices sur les schémas à un pas

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import fsolve
from scipy.integrate import solve_ivp, odeint
import sympy as sp
from IPython.display import display, Math, Markdown
# plt.rcParams.update({'font.size': 12})
plt.rcdefaults() # Réinitialisation des paramètres par défaut de Matplotlib
# Mise à jour des paramètres de police
# plt.rcParams.update({'font.size': 12})
# plt.rcParams.update({'figure.max_open_warning': 0})
```

17.1. Exercice : Modèle de Malthus (schéma d'Euler explicite)

On modélise la croissance d'une population d'effectif $y(t) \geq 0$ par l'équation différentielle

$$y'(t) = ay(t), \qquad a \in \mathbb{R}. \tag{17.1}$$

Dans ce modèle la variation de population est proportionnelle à la population elle-même : le coefficient a est appelé paramètre de Malthus ou potentiel biologique et représente la différence entre le nombre de nouveaux nés par individu en une unité de temps et la fraction d'individus qui meurent en une unité de temps. On s'intéresse à l'unique solution de cette équation vérifiant $y(t_0) = y_0$ où y_0 et t_0 sont des réels donnés.

1. Calculer la solution exacte. Calculer $\lim_{t\to+\infty} y(t)$ en fonction du paramètre a.
2. Soit $(u_n)_n$ la suite obtenue avec la méthode d'Euler explicite sur l'intervalle $[0;2]$ avec $N = 5$. Exprimer u_n en fonction de u_0 (en éliminant la relation de récurrence). Tracer ensuite les suites $(y_n)_n$ (solution exacte discrète) et $(u_n)_n$ (solution approchée par la méthode d'Euler explicite).
3. Soit $u_N(2)$ l'approximation de $y(2)$ obtenue avec la méthode d'Euler explicite et N nœuds. Calculer $u_N(2)$ avec $N = 2^i$ et montrer analytiquement que $\lim_{N\to+\infty} u_N(2) = y(2)$.
4. Tracer le graphe de $\max_{0\leq n\leq N} |y(t_n) - u_n|$ en fonction de N ainsi que le graphe des courbes $1/N$, $1/N^2$ … avec une progression logarithmique en abscisse et en ordonnées.

L'unique solution du problème de Cauchy $\begin{cases} y'(t) = ay(t) \\ y(t_0) = y_0 > 0 \end{cases}$ pour $t \geq t_0$ est la fonction $y(t) = y_0 e^{at}$. On a

$$\lim_{t\to+\infty} y(t) = \begin{cases} +\infty & \text{si } a > 0, \\ y_0 & \text{si } a = 0, \\ 0^+ & \text{si } a < 0, \end{cases}$$

Pour $a = 1$, $t_0 = 0$ et $y_0 = 1$, la solution exacte est $y(t) = e^t$.

On subdivise l'intervalle $[t_0; T]$ en N intervalles $[t_n; t_{n+1}]$ de largeur $h = (T - t_0)/N$ avec $t_n = t_0 + nh$ pour $n = 0, \dots, N$.
Pour chaque nœud t_n, on note $y_n = y(t_n)$; l'ensemble des valeurs $\{y_0, y_1, \dots, y_N\}$ représente la solution exacte discrète.

Pour chaque nœud t_n, on cherche la valeur inconnue u_n qui approche la valeur exacte y_n.
L'ensemble des valeurs $\{u_0 = y_0, u_1, \dots, u_N\}$ représente la solution numérique.

Dans la méthode d'Euler explicite, cette solution approchée est obtenue en construisant une suite récurrente comme suit

$$\begin{cases} u_0 = y_0, \\ u_{n+1} = u_n + h\varphi(t_n, u_n). \end{cases}$$

En utilisant ce schéma pour notre EDO on obtient

$$\begin{cases} u_0 = y_0, \\ u_{n+1} = (1 + ah)u_n \end{cases}$$

soit encore $u_n = (1 + ah)^n u_0 = \left(1 + \frac{2}{5}\right)^n$.

```
def EE(phi, tt, y0):
    h = tt[1] - tt[0]
    uu = np.empty_like(tt)
    uu[0] = y0
    for n in range(len(tt) - 1):
        uu[n + 1] = uu[n] + h * phi(tt[n], uu[n])
    return uu

phi = lambda t, y: a * y
sol_exacte = lambda t: y0 * np.exp(a * t)

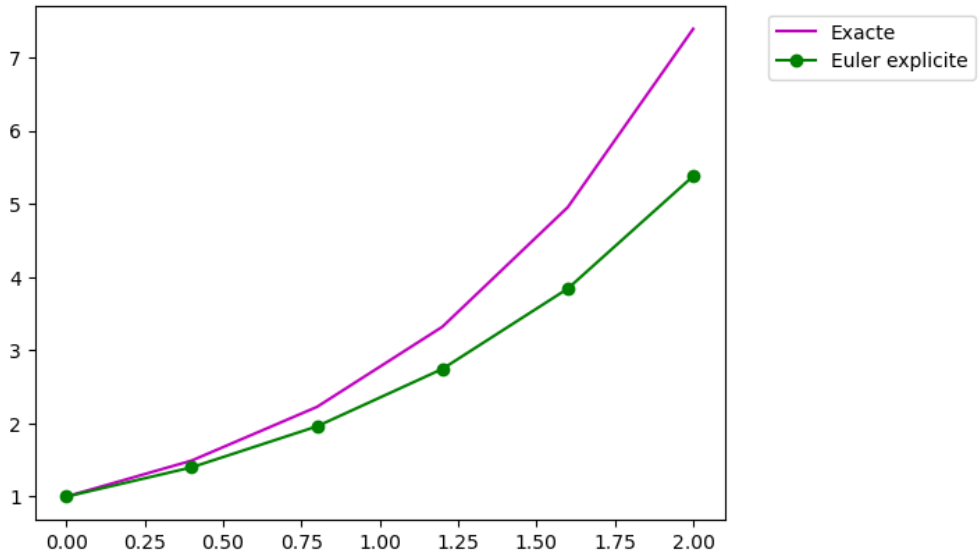
# INITIALISATION
a = 1
N = 5
t0, y0 = 0, 1
tfinal = 2

# CALCUL SOLUTION APPROCHEE
tt = np.linspace(t0, tfinal, N + 1)
uu = EE(phi, tt, y0)

# CALCUL SOLUTION EXACTE POUR PLOT
yy = sol_exacte(tt)

# PLOT
plt.plot(tt, yy, 'm-', label='Exacte')
plt.plot(tt, uu, 'go-', label='Euler explicite')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()

# TABLEAU
from tabulate import tabulate
T = []
T.append(["n", "y_n", "u_n", "(1+h)^n"])
h1 = tt[1] - tt[0]
for n in range(N):
    T.append([n, yy[n], uu[n], (1 + h1) ** n])
display(tabulate(T, headers="firstrow", tablefmt="html"))
```



n	y_n	u_n	(1+h)^n
0	1	1	1
1	1.49182	1.4	1.4
2	2.22554	1.96	1.96
3	3.32012	2.744	2.744
4	4.95303	3.8416	3.8416

On compare les approximations de $y(2)$ avec plusieurs valeurs de N_h :

$$u_{N_h} = (1+ah)^{N_h} u_0 = \left(1 + \frac{2}{N_h}\right)^{N_h} \xrightarrow[N_h \to +\infty]{} e^2 = y(2).$$

```
y_2 = sol_exacte(tfinal)
display(Markdown(f'Solution exacte en $t=2$ : $y(2) = {y_2}$'))
display(Markdown('Solution approchee en $t=2$ : $u(2)$ avec $N$ points:'))
from tabulate import tabulate
table_data = []
table_data.append(["N", "u(2)", "|y(2)-u(2)|"])
for i in range(3, 15):
    N = 2 ** i
    tt = np.linspace(t0, tfinal, N + 1)
    uu = EE(phi, tt, y0)
    error = abs(uu[-1] - y_2)
    table_data.append([N, f'{uu[-1]:1.8f}', f'{error:1.8f}'])
display(tabulate(table_data, headers="firstrow", tablefmt="html"))
```

Solution exacte en $t = 2$: $y(2) = 7.38905609893065$

Solution approchee en $t = 2$: $u(2)$ avec N points :

N	u(2)	|y(2)-u(2)|
8	5.96046	1.42859
16	6.58325	0.805806
32	6.95867	0.430389
64	7.16628	0.22278
128	7.27567	0.113386
256	7.33185	0.0572055
512	7.36032	0.0287325

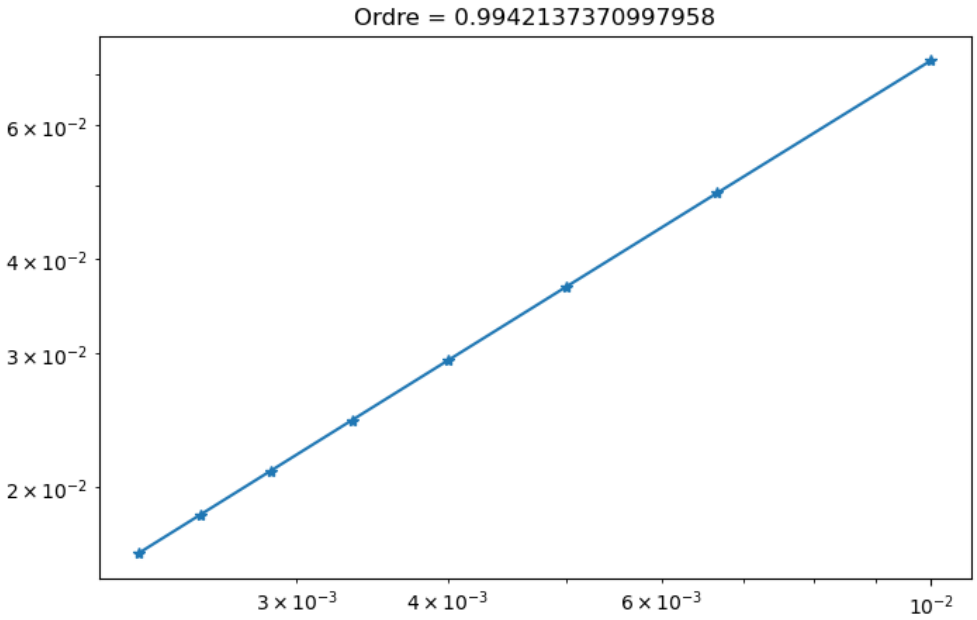
N	u(2)	|y(2)-u(2)|
1024	7.37466	0.0143989
2048	7.38185	0.00720766
4096	7.38545	0.00360588
8192	7.38725	0.00180346
16384	7.38815	0.00090186

Vérifions la convergence linéaire :

```
error = []
H = []
N = np.arange(200, 1000, 100)
for n in N:
    tt = np.linspace(t0, tfinal, n + 1)
    H.append(tt[1] - tt[0])
    yy = sol_exacte(tt)
    uu = EE(phi, tt, y0)
    error.append(np.linalg.norm(uu-yy, np.inf))

# ESTIMATION ORDRES DE CONVERGENCE par regression
pente = np.polyfit(np.log(H), np.log(error), 1)[0]

# PLOT ERREUR
plt.figure(figsize=(8, 5))
plt.title(f'Ordre = {pente}')
plt.loglog(H, error, '*-');
```



17.2. Exercice : Modèle de Verhulst (schéma d'Euler explicite)

Dans le modèle logistique, on suppose que la croissance de la population d'effectif $y(t) \geq 0$ est modélisée par l'équation différentielle :

$$y'(t) = (a - by(t))y(t), \qquad a, b > 0. \tag{17.2}$$

Ce modèle prend en compte une éventuelle surpopulation. En effet, le taux de croissance de la population est désormais de la forme $a - by(t)$: c'est la différence entre un taux de fertilité a et un taux de mortalité de la forme $by(t)$ qui augmente avec l'effectif de la population.

1. Calculer les solutions exactes de l'équation différentielle.
2. Écrire la relation entre u_{n+1} et u_n obtenue en appliquant la méthode d'Euler explicite.
3. Pour $a = b = 0.5$ et $t_0 = 0$, tracer les solutions exactes sur l'intervalle de temps $[0; 20]$. Sur un autre graphique à côté, tracer les solutions correspondantes obtenues par la méthode d'Euler explicite avec $N = 50$. Sur un autre graphique à côté, tracer les solutions correspondantes obtenues par la méthode d'Euler explicite avec $N = 100$. Considérer les valeurs de y_0 suivantes :

$$y_0 = 0.01 \quad y_0 = 0.1 \quad y_0 = 0.4 \quad y_0 = 0.8 \quad y_0 = 1 \quad y_0 = 1.5 \quad y_0 = 2$$

On cherche l'unique solution du problème de Cauchy $\begin{cases} y'(t) = (a - by(t))y(t) \\ y(t_0) = y_0 \geq 0 \end{cases}$ pour $t \geq t_0$.

On écrit l'EDO sous la forme $y'(t) = -by(t)(y(t) - \frac{a}{b})$.

Si $y(t) = A$ pour tout $t > 0$ est solution de l'EDO, alors on doit avoir $A = (a - bA)A$. Donc, soit $A = 0$ soit $A = \frac{a}{b}$. Ainsi la solution constante $y(t) = \frac{a}{b} > 0$ représente l'effectif d'équilibre de la population, directement liée aux contraintes environnementales (densité, ressources, etc.).

De plus, les solutions sont des fonctions strictement croissantes si $y(t) \in]0, a/b[$ et décroissantes si $y(t) > a/b$.

Calculons la solution exacte en remarquant qu'il s'agit d'une EDO à variables séparables. Pour $y(t) \neq 0$ et $y(t) \neq \frac{a}{b}$, on remarque que

$$-b = \frac{y'(t)}{(y(t) - \frac{a}{b})\, y(t)} = \frac{\frac{b}{a}y'(t)}{y(t) - \frac{a}{b}y'(t)} - \frac{\frac{b}{a}}{y(t)}.$$

On doit alors calculer

$$\int \frac{1}{y - \frac{a}{b}} \mathrm{d}y - \int \frac{1}{y} \mathrm{d}y = -a \int 1 \mathrm{d}t.$$

On obtient

$$\ln\left(1 - \frac{b/a}{y}\right) = -at + C$$

et on en déduit

$$y(t) = \frac{a/b}{1 - De^{-at}}.$$

En imposant la condition initiale $y(0) = y_0$ on trouve la constante d'intégration D et on conclut que toutes les solutions du problème de Cauchy pour $t \geq 0$ sont

$$y(t) = \begin{cases} 0 & \text{si } y_0 = 0, \\ \dfrac{a}{b} & \text{si } y_0 = \dfrac{a}{b}, \\ \dfrac{1}{\left(\frac{1}{y_0} - \frac{b}{a}\right) e^{-at} + \frac{b}{a}} & \text{sinon.} \end{cases}$$

Remarquons que $\lim_{t \to +\infty} y(t) = \frac{a}{b}$: une population qui évolue à partir de y_0 individus à l'instant initiale tend à se stabiliser vers un nombre d'individus d'environ a/b, ce qui représente la capacité de l'environnement.

La méthode d'Euler explicite (ou progressive) est une méthode d'intégration numérique d'EDO du premier ordre de la forme $y'(t) = \varphi(t, y(t))$.
On subdivise l'intervalle $[t_0; T]$ en N intervalles $[t_n; t_{n+1}]$ de largeur $h = \dfrac{T - t_0}{N}$ avec $t_n = t_0 + nh$ pour $n = 0, \ldots, N$.
La longueur h est appelé le pas de discrétisation.
Pour chaque nœud t_n, on note $y_n = y(t_n)$; l'ensemble des valeurs $\{y_0, y_1, \ldots, y_N\}$ représente la solution exacte discrète.
Pour chaque nœud t_n, on cherche la valeur inconnue u_n qui approche la valeur exacte y_n.
L'ensemble des valeurs $\{u_0 = y_0, u_1, \ldots, u_{N_h}\}$ représente la solution numérique.

Dans la méthode d'Euler explicite, cette solution approchée est obtenue en construisant une suite récurrente comme suit

$$\begin{cases} u_0 = y_0, \\ u_{n+1} = u_n + h\varphi(t_n, u_n). \end{cases}$$

En utilisant cette construction pour notre EDO on obtient

$$\begin{cases} u_0 = y_0, \\ u_{n+1} = u_n + h(a - bu_n)u_n \end{cases}$$

```
def EE(phi, tt, y0):
    h = tt[1] - tt[0]
    uu = np.empty_like(tt)
    uu[0] = y0
    for n in range(len(tt) - 1):
        uu[n + 1] = uu[n] + h * phi(tt[n], uu[n])
    return uu

phi = lambda t, y: (a - b * y) * y

# def sol_exacte(t, y0):
#     if abs(y0 - 0) <= 1.e-10:  # avec des float on ne peut pas utiliser ==0
#         return 0
#     elif abs(y0 - a / b) <= 1.e-10:
#         return a / b
#     else:
#         return 1 / ((1 / y0 - b / a) * np.exp(-a * t) + b / a)

# # Vectoriser la fonction sol_exacte
# sol_exacte_vectorized = np.vectorize(sol_exacte, excluded=['y0'])

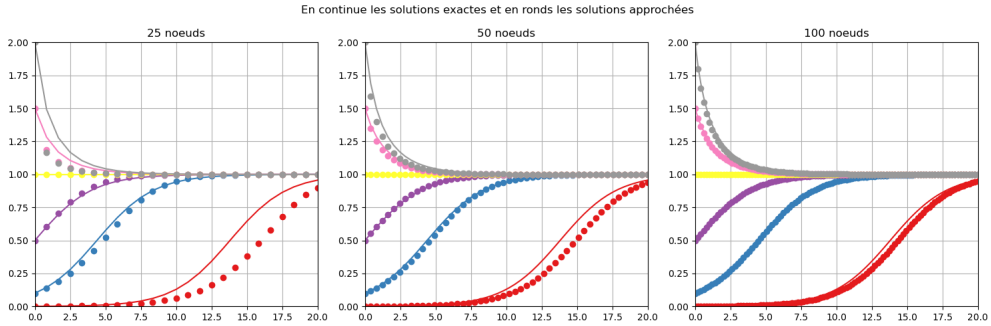
# Version directement vectorisée sans np.vectorize
sol_exacte_vectorized = lambda y, y0 : np.where(  abs(y0-0)<=1.e-10, 0 , np.where( abs(y0 - a / b) <=
 ↪  1.e-10, a / b, 1 / ((1 / y0 - b / a) * np.exp(-a * y) + b / a) )  )

# INITIALISATION
a = 0.5
b = 0.5
t0 = 0
tfinal = 20
yy_0 = [0.001, 0.1, 0.5, 1, 1.5, 2]
NN = [25, 50, 100]


# MAIN
f, axs = plt.subplots(1, 3, figsize=(15, 5))
axs = np.ravel(axs)
colors = plt.cm.Set1(np.linspace(0, 1, len(yy_0)))
for ax,N in zip(axs,NN):
    tt = np.linspace(t0, tfinal, N)
    for i, y0 in enumerate(yy_0):
        color = colors[i]
        uu = EE(phi, tt, y0)
        ax.plot(tt, uu, 'o', label=f'Approchée avec y_0={y0}',color=color)
        yy = sol_exacte_vectorized(tt, y0)
        ax.plot(tt, yy, label=f'Exacte avec y_0={y0}',color=color)
    ax.set_title(f'{N} noeuds')
    ax.set_xlim(t0, tfinal)
    ax.set_ylim(0, 2.)
    ax.grid()
    # ax.legend()

plt.suptitle('En continue les solutions exactes et en ronds les solutions approchées')
```

```
plt.tight_layout()
plt.show()
```



17.3. Exercice : concentration (schéma d'Euler implicite)

L'évolution de la concentration de certaines réactions chimiques au cours du temps peut être décrite par l'équation différentielle

$$y'(t) = -\frac{1}{1+t^2}y(t).$$

Sachant qu'à l'instant $t = 0$ la concentration est $y(0) = 5$, déterminer la concentration à $t = 2$ à l'aide de la méthode d'**Euler implicite** avec un pas $h = 0.5$.

Dans ce cas, le schéma d'Euler implicite s'écrit

$$u_{n+1} = u_n - \frac{h}{1+t_{n+1}^2}u_{n+1}.$$

Elle peut être résolue algébriquement et cela donne la suite

$$u_{n+1} = \frac{u_n}{1+\frac{h}{1+t_{n+1}^2}}.$$

Si à l'instant $t = 0$ la concentration est $y(0) = 5$, et si $h = 1/2$, alors $t_n = n/2$ et

$$u_{n+1} = \frac{4+(n+1)^2}{6+(n+1)^2}u_n.$$

On obtient donc

n	t_n	u_n
0	0	5
1	0.5	$\frac{4+1^2}{6+1^2}5 = \frac{5}{7}5 = \frac{25}{7} \approx 3.57$
2	1.0	$\frac{4+2^2}{6+2^2}\frac{25}{7} = \frac{8}{10}\frac{25}{7} = \frac{20}{7} \approx 2.86$
3	1.5	$\frac{4+3^2}{6+3^2}\frac{20}{7} = \frac{13}{15}\frac{20}{7} = \frac{52}{21} \approx 2.48$
4	2.0	$\frac{4+4^2}{6+4^2}\frac{52}{21} = \frac{20}{22}\frac{52}{21} = \frac{520}{231} \approx 2.25$

La concentration à $t = 2$ est d'environ 2.25 qu'on peut comparer avec le calcul exact $y(2) = 5e^{-\arctan(2)} \approx 1.652499838$.

17.4. Exercice : A-stabilité du schéma de Crank-Nicolson

Soit le problème de Cauchy :

$$\begin{cases} y'(t) + 10y(t) = 0, \quad \forall t \in \mathbb{R}, \\ y(0) = y_0 > 0. \end{cases}$$

1. Montrer qu'il existe une unique solution globale $y \in \mathcal{C}^\infty(\mathbb{R}, \mathbb{R})$ que vous préciserez explicitement.
2. Soit $(u_n)_{n\in\mathbb{N}}$ la suite obtenue avec le schéma numérique de Crank-Nicolson. Montrer que $(u_n)_{n\in\mathbb{N}}$ est une suite géométrique dont on précisera la raison. Donner l'expression de u_n en fonction de n.
3. Montrer que la raison r de la suite vérifie $|r| < 1$ pour tout $h > 0$. Ce schéma est-il inconditionnellement A-stable ?
4. Sous quelle condition sur $h > 0$ le schéma génère-t-il une suite positive ?

C'est un problème deCauchy du type

$$\begin{cases} y'(t) = \varphi(t, y(t)), \quad \forall t \in \mathbb{R}, \\ y(0) = y_0 > 0, \end{cases}$$

avec $\varphi(t, y) = g(y) = -10y$.

On montre d'abord qu'il existe une et une seule solution locale (ie sur $[-T; T]$) de classe $\mathcal{C}^1([-T, T], \mathbb{R})$.
On montre ensuite que cette solution est de classe $\mathcal{C}^\infty([-T, T], \mathbb{R})$.
On montre enfin que la solution admet un prolongement sur $\mathbb{R}$. + Comme $g \in \mathcal{C}^1(\mathbb{R}, \mathbb{R})$, d'après le théorème de Cauchy-Lipschitz il existe $T > 0$ et une unique solution $y \in \mathcal{C}^1([-T, T], \mathbb{R})$. + Par récurrence, en exploitant l'EDO et la régularité de g, on grimpe en régularité sur y ainsi $y \in \mathcal{C}^\infty([-T, T], \mathbb{R})$. + La fonctionne nulle est solution de l'EDO (mais non du problème de Cauchy donné). Par l'unicité de la solution du problème de Cauchy on en déduit que soit $y(t) > 0$ pour tout $t \in [-T, T]$ (ie lorsque $y_0 > 0$) soit $y(t) < 0$ pour tout $t \in [-T, T]$ (ie lorsque $y_0 > 0$). De plus, y est décroissante si $y_0 > 0$ et croissante si $y_0 < 0$. On en déduit par le théorème des extrémités que la solution u admet un prolongement sur $\mathbb{R}$ solution de l'EDO.

Pour en calculer la solution, on remarque qu'il s'agit d'une EDO à variables séparables. L'unique solution constante est $y(t) \equiv 0$, toutes les autres solutions sont du type $y(t) = Ce^{-10t}$. En prenant en compte la condition initiale on conclut que l'unique solution du problème de Cauchy est

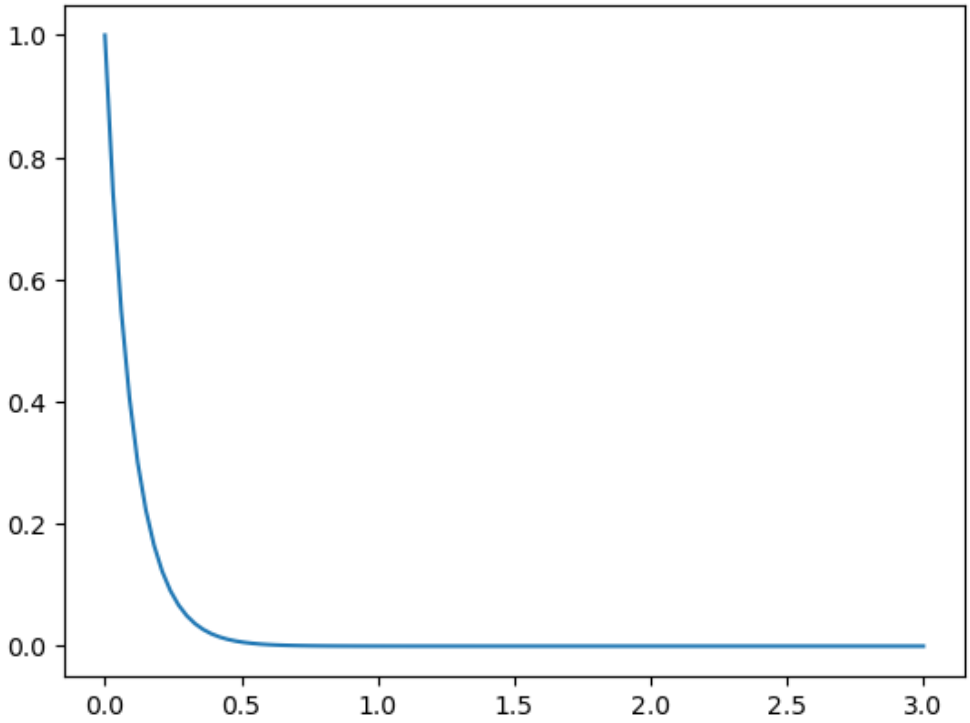
$$y(t) = y_0 e^{-10t}$$

définie pour tout $t \in \mathbb{R}$.

```
# Définition de la fonction exacte
y0 = 1
exacte = lambda t: y0 * np.exp(-10*t)

# Génération des données pour le tracé
tt = np.linspace(0, 3, 101)
yy = exacte(tt)

plt.plot(tt,yy);
```



Le schéma de Crank-Nicolson construit la suite $(u_n)_{n\in\mathbb{N}}$ selon la relation

$$u_{n+1} = u_n - 5h(u_{n+1} + u_n), \qquad \forall n \in \mathbb{N},$$

pour $h > 0$ fixé. On obtient une formule de récurrence rendue explicite par un calcul élémentaire :

$$u_{n+1} = -5hu_{n+1} - 5hu_n + u_n$$

d'où

$$u_{n+1} = \frac{1-5h}{1+5h}u_n.$$

Par récurrence on obtient

$$u_n = \left(\frac{1-5h}{1+5h}\right)^n u_0.$$

Il s'agit d'une suite géométrique de raison

$$r = \frac{1-5h}{1+5h}.$$

Pour tout $h > 0$ on a

$$r = \frac{1-5h}{1+5h} = 1 - \frac{10h}{1+5h}$$

et

$$-1 < 1 - \frac{10h}{1+5h} < 1.$$

Ce schéma est donc inconditionnellement A-stable car $|u_{n+1}| = |r^{n+1}u_0| \leq |u_0|$.

Le schéma génère une suite positive ssi

$$1 - \frac{10h}{1+5h} > 0$$

c'est-à-dire ssi

$$h < \frac{1}{5}.$$

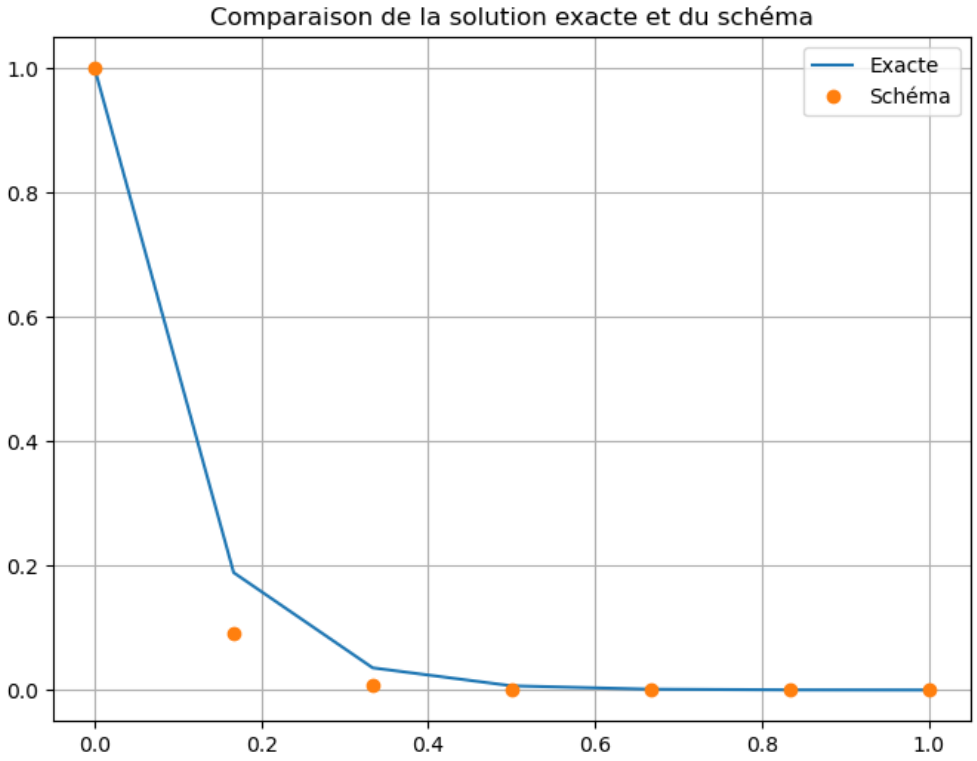
```
N = 6
tt = np.linspace(0, 1, N+1)
y0 = 1

exacte = lambda t: y0 * np.exp(-10*t)

phi = lambda t, y: -y**5
def schema(phi, tt, y0):
    h = tt[1] - tt[0]
    uu = np.empty_like(tt)
    uu[0] = y0
    for n in range(len(tt)-1):
        uu[n+1] = (1 - 5*h) / (1 + 5*h) * uu[n]
    return uu


yy = exacte(tt)
uu = schema(phi, tt, y0)

plt.figure(figsize=(8, 6))
plt.plot(tt, yy, label="Exacte")
plt.plot(tt, uu, "o", label="Schéma")
plt.title("Comparaison de la solution exacte et du schéma")
plt.grid()
plt.legend();
```



17.5. Exercice : A-stabilité d'un schéma

Soit le problème de Cauchy :

$$\begin{cases} y'(t) + \dfrac{\sqrt{y(t)}}{2} = 0, \quad \forall t \in \mathbb{R}^+, \\ y(0) = y_0 > 0. \end{cases}$$

Pour $h > 0$ fixé, soit le schéma numérique défini par la suite $(u_n)_{n\in\mathbb{N}}$ suivante

$$\frac{u_{n+1} - u_n}{h} + \frac{u_{n+1}}{2\sqrt{u_n}} = 0, \qquad \forall n \in \mathbb{N},$$

1. Expliciter l'expression de u_{n+1} en fonction de u_n.
2. Montrer que la suite $(u_n)_{n\in\mathbb{N}}$ est une suite positive, décroissante et convergente vers 0.

C'est un problème de Cauchy du type

$$\begin{cases} y'(t) = \varphi(t, y(t)), & \forall t \in \mathbb{R}, \\ y(0) = y_0 > 0, \end{cases}$$

avec $\varphi(t, y) = g(y) = -\frac{\sqrt{y}}{2}$.

Pour $h > 0$ fixé on obtient une formule de récurrence rendue explicite par un calcul élémentaire :

$$u_{n+1} = \frac{u_n}{1 + \frac{h}{2\sqrt{u_n}}} = \frac{2(u_n)^{3/2}}{2\sqrt{u_n} + h}, \qquad \forall n \in \mathbb{N}.$$

On étudie la suite

$$\begin{cases} u_0 > 0, \\ u_{n+1} = \frac{2(u_n)^{3/2}}{2\sqrt{u_n}+h}, \end{cases} \qquad \forall n \in \mathbb{N},$$

pour $h > 0$ fixé.

Par récurrence on montre que si $u_0 > 0$ alors $u_n > 0$ pour tout $n \in \mathbb{N}$. De plus, $\frac{u_{n+1}}{u_n} = \frac{1}{1+\frac{h}{2\sqrt{u_n}}} < 1$ pour tout $n \in \mathbb{N}$: la suite est monotone décroissante. On a alors une suite décroissante et bornée par zéro, donc elle converge. Soit ℓ la limite de cette suite, alors $\ell \geq 0$ et $\ell = \frac{2\ell^{3/2}}{2\sqrt{\ell}+h}$ donc $\ell = 0$.

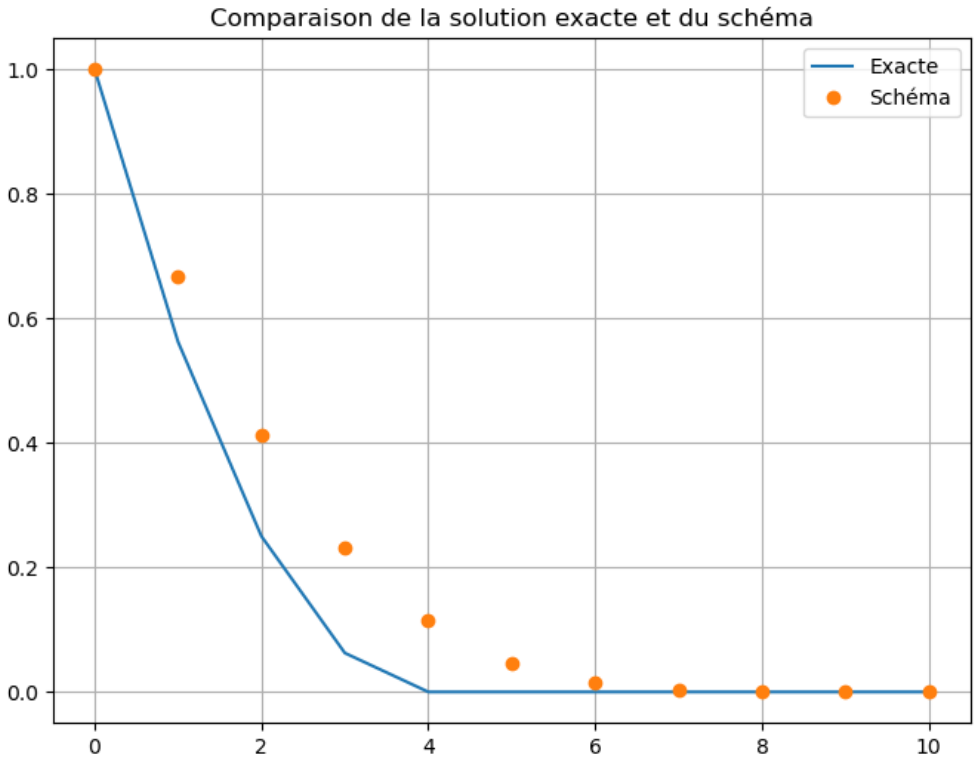
```
t0 = 0
y0 = 1
N = 10
tt = np.linspace(t0, 10, N+1)


phi = lambda t, y: -0.5 * np.sqrt(y)
exacte = lambda t: np.where(t >= 4 * np.sqrt(y0), 0, (np.sqrt(y0) - t / 4) ** 2)


def schema(phi, tt, y0):
    h = tt[1] - tt[0]
    uu = [y0]
    for n in range(len(tt)-1):
        uu.append(2 * (uu[n]) ** (3/2) / (h + 2 * np.sqrt(uu[n])))
    return uu

yy = exacte(tt)
uu = schema(phi, tt, y0)

plt.figure(figsize=(8, 6))
plt.plot(tt, yy, label="Exacte")
plt.plot(tt, uu, "o", label="Schéma")
plt.title("Comparaison de la solution exacte et du schéma")
plt.grid()
plt.legend();
```



17.6. Exercice : A-stabilité d'un schéma

Soit le problème de Cauchy :

$$\begin{cases} y'(t) + y^5(t) = 0, \quad \forall t \in \mathbb{R}^+, \\ y(0) = y_0 > 0. \end{cases}$$

1. Montrer qu'il existe une unique solution globale $y \in \mathcal{C}^\infty(\mathbb{R}^+, \mathbb{R}^+)$.
2. Soit le schéma numérique défini par la suite $(u_n)_{n\in\mathbb{N}}$ suivante

$$\frac{u_{n+1} - u_n}{h} + u_{n+1}u_n^4 = 0, \qquad \forall n \in \mathbb{N},$$

   pour $h > 0$ fixé. Expliciter l'expression de u_{n+1} en fonction de u_n.
3. Montrer que la suite $(u_n)_{n\in\mathbb{N}}$ est une suite décroissante et convergente vers 0 pour tout $h > 0$ fixé.

C'est un problème deCauchy du type

$$\begin{cases} y'(t) = \varphi(t, y(t)), \quad \forall t \in \mathbb{R}, \\ y(0) = y_0 > 0, \end{cases}$$

avec $\varphi(t, y) = g(y) = -y^5$.

On montre d'abord qu'il existe une et une seule solution locale (ie sur $[0; T]$) de classe $\mathcal{C}^1([0, T], \mathbb{R})$.
On montre ensuite que cette solution est de classe $\mathcal{C}^\infty([0, T], \mathbb{R})$.
On montre enfin que la solution admet un prolongement sur $\mathbb{R}$. + Comme $g \in \mathcal{C}^1(\mathbb{R}^+, \mathbb{R}^+)$, d'après le théorème de Cauchy-Lipschitz il existe $T > 0$ et une unique solution $y \in \mathcal{C}^1([0, T], \mathbb{R})$. + Par récurrence, en exploitant l'EDO et la régularité de g, on grimpe en régularité sur y ainsi $y \in \mathcal{C}^\infty([0, T], \mathbb{R})$. + La fonctionne nulle est solution de l'EDO (mais non du problème de Cauchy donné). Comme $y_0 > 0$, par l'unicité de la solution du problème de Cauchy on a $y(t) > 0$ pour tout $t \in [0, T]$ (car deux trajectoires ne peuvent pas se croiser). De plus, y est décroissante, ainsi la solution est bornée ($y(t) \in]0, y_0[$). On en déduit par le théorème des extrémités que la solution y admet un prolongement sur $\mathbb{R}^+$ solution de l'EDO.

Pour en calculer la solution, on remarque qu'il s'agit d'une EDO à variables séparables. L'unique solution du

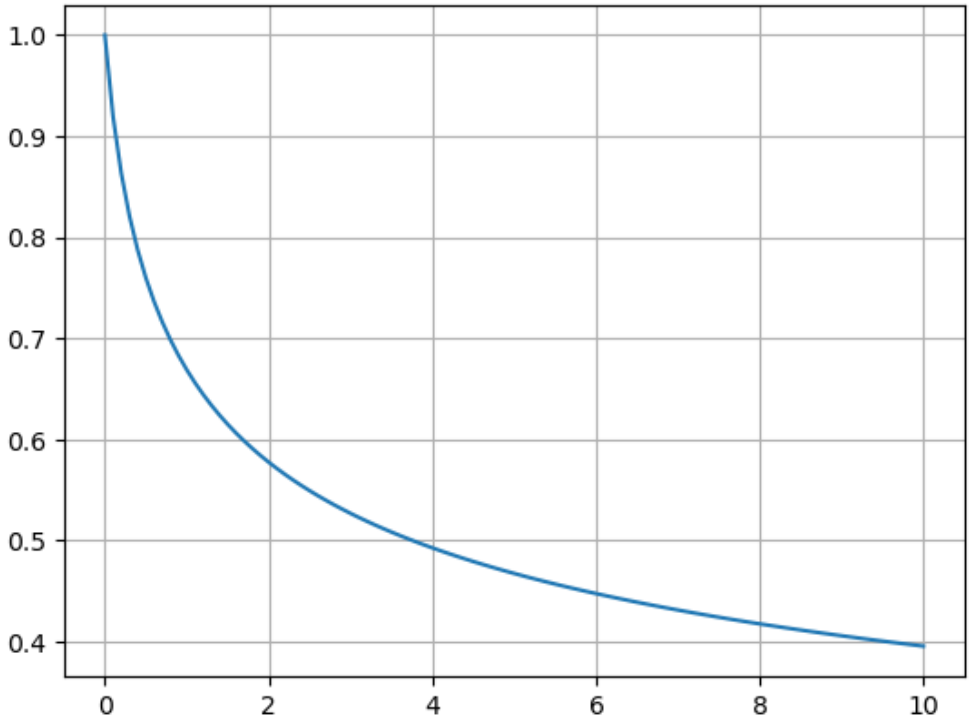
problème de Cauchy est

$$y(t) = y_0(4ty_0^4 + 1)^{-1/4}$$

```
t0 = 0
y0 = 1
exacte = lambda t: y0 * (4 * t * y0 ** 4 + 1) ** (-0.25)

tt = np.linspace(t0, 10, 101)
yy = exacte(tt)

plt.plot(tt, yy)
plt.grid()
```



Pour $h > 0$ fixé on obtient une formule de récurrence rendue explicite par un calcul élémentaire :

$$u_{n+1} = \frac{u_n}{1 + u_n^4 h}, \qquad \forall n \in \mathbb{N}.$$

On étudie la suite

$$\begin{cases} u_0 = y_0 > 0, \\ u_{n+1} = \frac{u_n}{1+u_n^4 h}, \end{cases} \qquad \forall n \in \mathbb{N},$$

pour $h > 0$ fixé.

Par récurrence on montre que si $u_0 > 0$ alors $u_n > 0$ pour tout $n \in \mathbb{N}$. De plus, $\frac{u_{n+1}}{u_n} = \frac{1}{1+u_n^4 h} < 1$ pour tout $n \in \mathbb{N}$: la suite est monotone décroissante. On a alors une suite décroissante et bornée par zéro, donc elle converge. Soit ℓ la limite de cette suite, alors $\ell \geq 0$ et $\ell = \frac{\ell}{1+\ell^4 h}$ donc $\ell = 0$. Ce schéma est donc inconditionnellement A-stable.

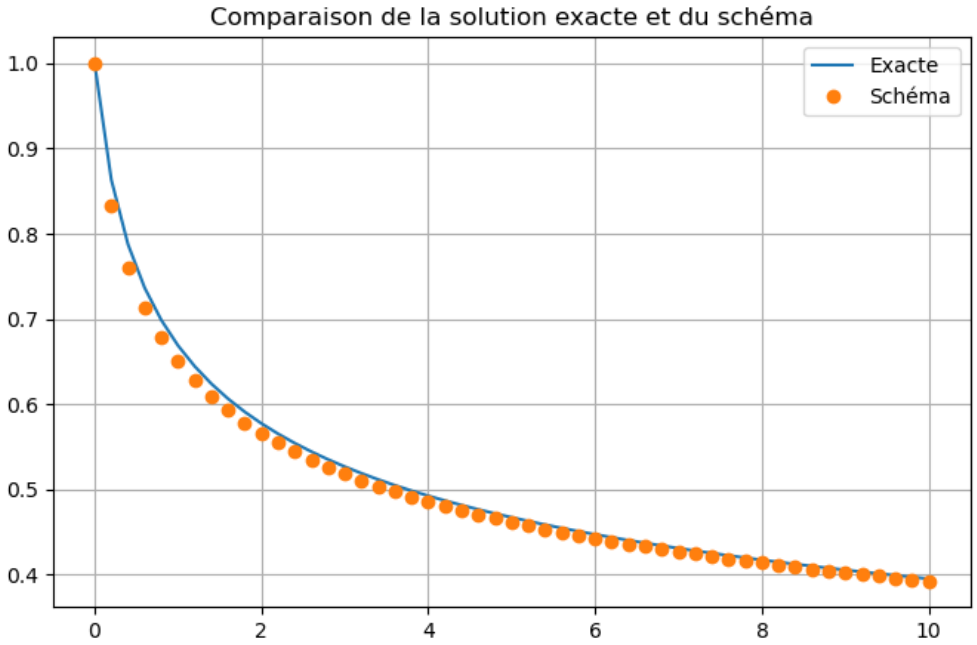
```
t0 = 0
y0 = 1
N = 50
tt = np.linspace(t0, 10, N+1)


phi = lambda t, y: -y ** 5
exacte = lambda t: y0 * (4 * t * y0 ** 4 + 1) ** (-0.25)
```

```
def schema(phi, tt, y0):
    h = tt[1] - tt[0]
    uu = np.empty_like(tt)
    uu[0] = y0
    for n in range(len(tt) - 1):
        uu[n + 1] = uu[n] / (1 + h * uu[n] ** 4)
    return uu

yy = exacte(tt)
uu = schema(phi, tt, y0)

plt.figure(figsize=(8, 5))
plt.plot(tt, yy, label="Exacte")
plt.plot(tt, uu, "o", label="Schéma")
plt.title("Comparaison de la solution exacte et du schéma")
plt.grid()
plt.legend();
```



17.7. Exercice : schéma d'Euler explicite et zéro-stabilité

Soit le problème de Cauchy :

$$\begin{cases} y'(t) + \sin(y(t)) = 0, & \forall t \in \mathbb{R}, \\ y(0) = y_0 > 0. \end{cases}$$

1. Montrer qu'il existe une unique solution globale $y \in \mathcal{C}^\infty(\mathbb{R}, \mathbb{R})$.
2. Écrire le schéma le schéma d'Euler explicite pour ce problème.
3. Montrer que la suite $(u_n)_{n\in\mathbb{N}}$ construite par ce schéma vérifie

$$|u_{n+1}| \leq |u_n| + h, \qquad \forall n \in \mathbb{N},$$

   où $h > 0$ est le pas de la suite.
4. En déduire que

$$|u_n| \leq |u_0| + nh, \qquad \forall n \in \mathbb{N}.$$

C'est un problème deCauchy du type

$$\begin{cases} y'(t) = \varphi(t, y(t)), & \forall t \in \mathbb{R}, \\ y(0) = y_0 > 0, \end{cases}$$

avec $\varphi(t, y) = g(y) = -\sin(y)$.

On montre d'abord qu'il existe une et une seule solution locale (ie sur $[0; T]$) de classe $\mathcal{C}^1([0, T], \mathbb{R})$.
On montre ensuite que cette solution est de classe $\mathcal{C}^\infty([0, T], \mathbb{R})$.
On montre enfin que la solution admet un prolongement sur $\mathbb{R}$. + Comme $g \in \mathcal{C}^1(\mathbb{R}, \mathbb{R})$, d'après le théorème de Cauchy-Lipschitz il existe $T > 0$ et une unique solution $y \in \mathcal{C}^1([-T, T], \mathbb{R})$. + Par récurrence, en exploitant l'EDO et la régularité de g, on grimpe en régularité sur y ainsi $y \in \mathcal{C}^\infty([-T, T], \mathbb{R})$. + Toutes les fonctions constante $y(t) = k\pi$ pour $k \in \mathbb{Z}$ sont solutions de l'équation différentielle car $g(k\pi) = 0$. Pour y_0 donné, soit $\tilde{k} \in \mathbb{Z}$ tel que $y_0 \in [\tilde{k}\pi; (\tilde{k} + 1)\pi]$; par l'unicité de la solution du problème de Cauchy on a $y(t) \in [\tilde{k}\pi; (\tilde{k} + 1)\pi]$ pour tout $t \in [-T, T]$ (car deux trajectoires ne peuvent pas se croiser), i.e. la solution est bornée. On en déduit par le théorème des extrémités que la solution y admet un prolongement sur $\mathbb{R}$ solution de l'EDO.

Soit $h > 0$ fixé et $t_n = nh$ pour tout $n \in \mathbb{Z}$. Le schéma d'Euler explicite pour l'EDO donnée construit la suite

$$u_{n+1} = u_n - h\sin(u_n), \qquad \forall n \in \mathbb{N}.$$

Comme $|a + b| \leq |a| + |b|$ et comme $|-\sin(x)| \leq 1$ pour tout $x \in \mathbb{R}$, on conclut que

$$|u_{n+1}| = |u_n - h\sin(u_n)| \leq |u_n| + |h\sin(u_n)| \leq |u_n| + h$$

pour $h > 0$ fixé.

Par récurrence : $|u_{n+1}| \leq |u_n| + h \leq |u_{n-1}| + 2h \leq ... \leq |u_0| + (n + 1)h$.

```
t0 = 0
y0 = 1

# SOLUTION EXACTE POUR y(0) = 1

t  = sp.Symbol('t')
y  = sp.Function('y')

right = -sp.sin(y(t)) # phi(t,y(t)) avec sympy
edo = sp.Eq( sp.diff(y(t),t) , right )
solgenLIST = sp.dsolve(edo,y(t))
display(solgenLIST)

solgen = solgenLIST[1]
display(solgen)

equation = sp.Eq( y0, solgen.rhs.subs(t,t0))
display(equation)

consts = sp.solve( equation , dict=True)[0]
display(consts)

solpar = solgen.subs(consts)
display(solpar)

func = sp.lambdify(t,solpar.rhs,'numpy')
```

```
[Eq(y(t), -acos((-C1 - exp(2*t))/(C1 - exp(2*t))) + 2*pi),
 Eq(y(t), acos((-C1 - exp(2*t))/(C1 - exp(2*t))))]
```

$$y(t) = \operatorname{acos}\left(\frac{-C_1 - e^{2t}}{C_1 - e^{2t}}\right)$$

$$1 = \operatorname{acos}\left(\frac{-C_1 - 1}{C_1 - 1}\right)$$

```
{C1: (-1 + cos(1))/(cos(1) + 1)}
```

$$y(t) = \operatorname{acos}\left(\frac{-e^{2t} - \frac{-1+\cos(1)}{\cos(1)+1}}{-e^{2t} + \frac{-1+\cos(1)}{\cos(1)+1}}\right)$$

```
# SOLUTION APPROCHEE PAR EULER EXPLICITE POUR y(0) = 1

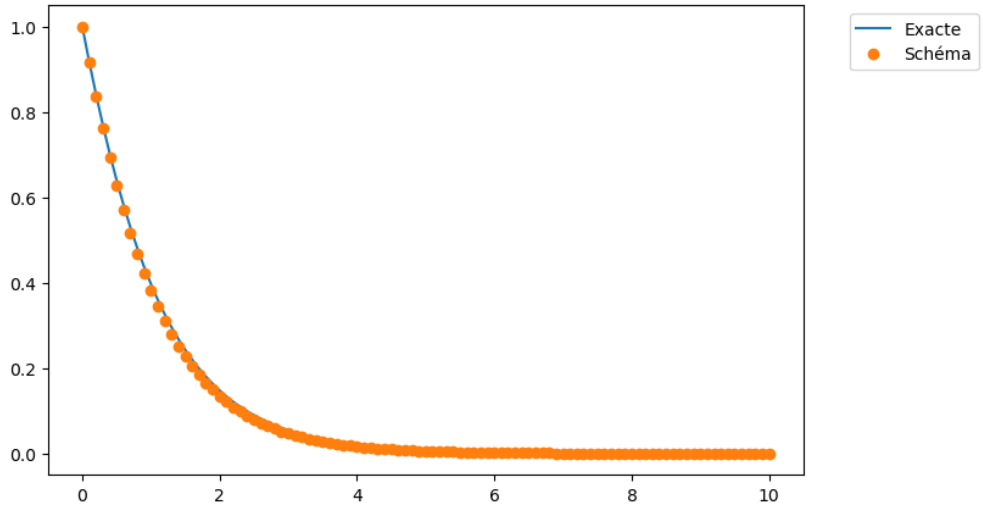
phi = lambda t, y: -np.sin(y)

def schema(phi, tt, y0):
    h = tt[1] - tt[0]
    uu = np.empty_like(tt)
    uu[0] = y0
    for i in range(len(tt) - 1):
        uu[i + 1] = uu[i]+h*phi(tt[i],uu[i])
    return uu
```

```
# COMPARAISONS
N = 100
tt = np.linspace(t0,10,N+1)

yy = func(tt)
uu = schema(phi, tt, y0)

plt.figure(figsize=(8,5))
plt.plot(tt, yy, label="Exacte")
plt.plot(tt, uu, 'o', label="Schéma")
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left');
```



17.8. Exercice : A-stabilité d'un schéma

Un modèle pour la diffusion d'une épidémie se base sur l'hypothèse que sa vitesse de propagation est proportionnelle au nombre d'individus infectés et au nombre d'individus sains.
Si on note $I(t) \geq 0$ le nombre d'individus infectés à l'instant $t \geq 0$ et $A > 0$ le nombre d'individus total, il existe une constante $k \in \mathbb{R}^+$ telle que $I'(t) = kI(t)(A - I(t))$.

1. Montrer qu'il existe $T > 0$ et une unique solution $I \in \mathcal{C}^\infty([0, T])$ au problème de Cauchy :

$$\begin{cases} I'(t) = kI(t)(A - I(t)), \\ I(0) = I_0 > 0. \end{cases}$$

2. Montrer que si $0 < I_0 < A$ alors $0 < I(t) < A$ pour tout $t > 0$.

3. Montrer que si $0 < I_0 < A$ alors $I(t)$ est croissante sur $\mathbb{R}^+$.
4. Soit $0 < I_0 < A$. On considère le schéma semi-implicite

$$\frac{I_{n+1} - I_n}{h} = kI_n(A - I_{n+1}).$$

   Montrer que $I_n \to A$ lorsque $n \to +\infty$ indépendamment du pas $h > 0$ fixé.
5. Calculer la solution exacte

C'est un problème deCauchy du type

$$\begin{cases} I'(t) = \varphi(t, I(t)), \quad \forall t \in \mathbb{R}^+, \\ I(0) = I_0 > 0, \end{cases}$$

avec $\varphi(t, I(t)) = g(I(t)) = kI(t)(A - I(t))$.

- Comme $g \in \mathcal{C}^1(\mathbb{R}, \mathbb{R})$, d'après le théorème de Cauchy-Lipschitz il existe $T > 0$ et une unique solution $I \in \mathcal{C}^1([0, T], \mathbb{R})$.
- Par récurrence, en exploitant l'EDO et la régularité de g, on grimpe en régularité sur I ainsi $I \in \mathcal{C}^\infty([0, T], \mathbb{R})$.
- Puisque la fonction nulle et la fonction constante $I(t) = A$ sont solutions de l'équation différentielle, si $0 < I_0 < A$ alors $0 < I(t) < A$ pour tout $t \in [0, T]$ (car, par l'unicité de la solution du problème de Cauchy, deux trajectoires ne peuvent pas se croiser).
- Puisque $I'(t) = kI(t)(A - I(t))$, si $0 < I_0 < A$ alors I est croissante pour tout $t \in [0, T]$. On en déduit par le théorème des extrémités que la solution I admet un prolongement sur $\mathbb{R}^+$ solution de l'EDO et que I est croissante pour tout $t \in \mathbb{R}^+$.

Soit $0 < I_0 < A$. On considère le schéma semi-implicite

$$\frac{I_{n+1} - I_n}{h} = kI_n(A - I_{n+1})$$

pour $h > 0$ fixé. On obtient une formule de récurrence rendue explicite par un calcul élémentaire :

$$I_{n+1} = \frac{1 + kAh}{1 + kI_n h} I_n.$$

Si $0 < I_0 < A$ alors $+\ I_n > 0$ quelque soit n ; $+\ I_n$ est majorée par A car

$$I_{n+1} \leq A \quad \iff \quad (1 + kAh)I_n \leq (1 + kI_n h)A \quad \iff \quad I_n \leq A$$

donc par récurrence $I_{n+1} \leq A$ quelque soit n ; $+$ si $I_n \to \ell$ alors $\ell = \frac{1+kAh}{1+k\ell h}\ell$ donc $\ell = 0$ ou $\ell = A$; $+\ I_n$ est une suite monotone croissante (encore par récurrence on montre que $|I_{n+1}| \geq |I_n| \geq ... \geq |I_0|$) ;

donc $I_n \to A$ lorsque $n \to +\infty$ indépendamment du pas $h > 0$ choisi.

On a déjà observé qu'il y a deux solutions constantes de l'EDO : la fonction $I(t) \equiv 0$ et la fonction $I(t) \equiv A$.

Pour chercher toutes les solutions non constantes on remarque qu'il s'agit d'une EDO à variables séparables donc on a

$$I(t) = \frac{A}{De^{-Akt} + 1}$$

La valeur numérique de la constante d'intégration D est obtenue grâce à la CI :

$$D = \frac{A - I_0}{I_0}$$

Exemple avec $A = 5000$, $I_0 = 160$, $k = \frac{\ln(363/38)}{35000}$ et $h = 1$:

```
# DONNEES
A = 5000
I0 = 160
k = np.log(363/38) / 35000

D = (A - I0) / I0
phi = lambda t, I: k * I * (A - I) # inutile pour notre schema

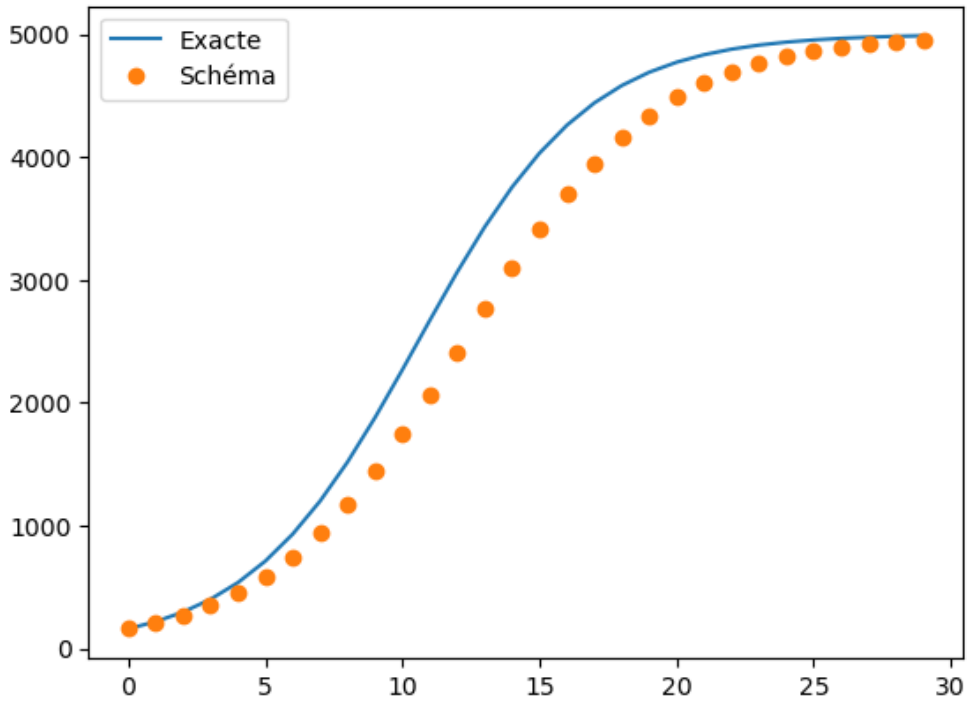

# SOLUTION EXACTE
exacte = lambda t: A / (D * np.exp(-A * k * t) + 1)

# SCHEMA
def schema(phi, tt, y0):
    uu = np.empty_like(tt)
    uu[0] = y0
    for n in range(len(tt) - 1):
        uu[n + 1] = (1 + k * A * h) / (1 + k * uu[n] * h) * uu[n]
    return uu


# MAIN
h = 1
tt = np.arange(0, 30, h)

yy = exacte(tt)
uu = schema(phi, tt, I0)

plt.plot(tt, yy, label="Exacte")
plt.plot(tt, uu, 'o', label="Schéma")
plt.legend();
```



17.9. Exercice : A-stabilité du ϑ-schéma

Pour calculer une approximation de la solution du problème de Cauchy

$$\begin{cases} y(t_0) = y_0, \\ y'(t) = \varphi(t, y(t)). \end{cases}$$

considérons la ϑ-méthode définie comme suit :

$$\begin{cases} u_0 = y_0, \\ u_{n+1} = u_n + h\Big(\vartheta\varphi(t_{n+1}, u_{n+1}) + (1-\vartheta)\varphi(t_n, u_n)\Big). \end{cases}$$

Ici ϑ est un paramètre dans l'intervalle $[0; 1]$.
L'objectif de cet exercice est d'étudier la A-stabilité de cette méthode en fonction du paramètre ϑ. Pour cela considérons l'EDO obtenue avec $\varphi(t, y(t)) = -\beta y(t)$ avec $\beta > 0$.
Donner une condition sur h pour que la ϑ-méthode soit A-stable. Pour quelles valeurs de ϑ la méthode est inconditionnellement A-stable ?
Remarque : pour $\vartheta = 0$ on retrouve la méthode d'Euler explicite, pour $\vartheta = 1$ la méthode d'Euler implicite, pour $\vartheta = \frac{1}{2}$ la méthode de Crank-Nicholson. On voit donc que pour certaines valeurs de ϑ la méthode est inconditionnellement A-stable et pour d'autres elle ne l'est pas.

$$u_{n+1} = u_n - \beta h(\vartheta u_{n+1} + (1-\vartheta)u_n) \iff u_{n+1} = \frac{1 - \beta h(1-\vartheta)}{1 + \beta h\vartheta} u_n = \frac{(1 + \beta h\vartheta) - \beta h}{1 + \beta h\vartheta} u_n$$

Par induction

$$u_n = \left(1 - \frac{\beta h}{1 + \beta h\vartheta}\right)^n y_0$$

Il s'agit d'une suite géométrique de raison $q = 1 - \frac{\beta h}{1+\beta h\vartheta}$.

La suite tends à zéro ssi $|q| < 1$ (la convergence est monotone ssi $0 \leq q < 1$) :

$$-1 < 1 - \frac{\beta h}{1 + \beta h\vartheta} < 1 \iff 0 < \frac{\beta h}{1 + \beta h\vartheta} < 2 \iff (1 - 2\vartheta)\beta h < 2$$

Pour $\vartheta \geq \frac{1}{2}$ la méthode est inconditionnellement A-stable,
pour $\vartheta < \frac{1}{2}$ la méthode est A-stable ssi $h < \frac{2}{(1-2\vartheta)\beta}$.

17.10. Exercice : Ordre d'un schéma

Une méthode numérique à un pas a été utilisée pour résoudre une équation différentielle avec condition initiale. Les résultats obtenus par cette méthode en prenant des pas de temps $h = 0.1$, $h = 0.05$ et $h = 0.025$ sont donnés dans le tableau suivant (remarque : une valeur sur deux est affichée pour $h = 0.05$ et une valeur sur quatre est affichée pour $h = 0.025$)

t_i	y_i pour $h = 0.1$	y_i pour $h = 0.05$	y_i pour $h = 0.025$
1.0	0.500000	0.500000	0.500000
1.1	0.512084	0.512242	0.512280
1.2	0.511698	0.512101	0.512196
1.3	0.500927	0.501559	0.501704
1.4	0.482686	0.483447	0.483619
1.5	0.459861	0.460633	0.460804

En calculant le rapport des erreurs et sachant que $y(1.5) = 0.460857$, déterminer l'ordre de la méthode numérique utilisée.

La méthode utilisée est d'ordre 2, en effet, pour $t = 1.5$ nous avons

$$\frac{E(h = 0.05)}{E(h = 0.025)} = \frac{|0.460633 - 0.460857|}{|0.460804 - 0.460857|} = 4.226 \simeq 2^2.$$

Si on a accès à `polyfit` on peut tracer la courbe d'erreur en echèlle logaritmique et estimer la pente :

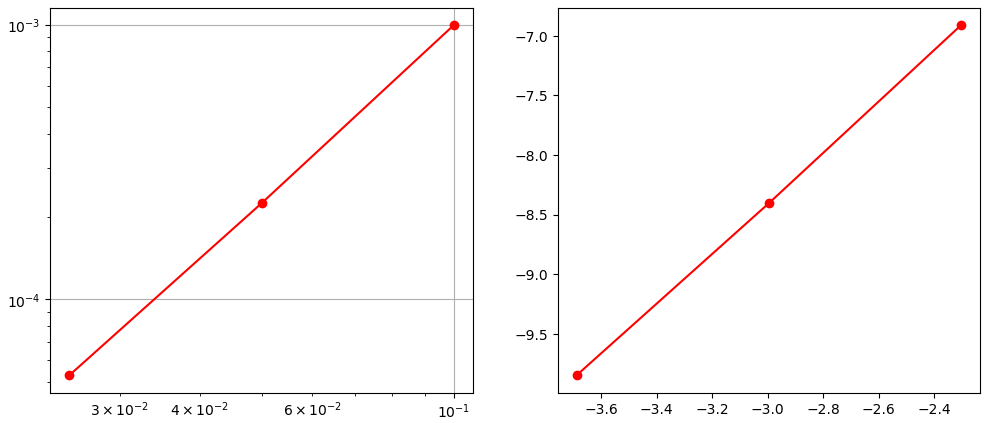
```
H = [0.1, 0.05, 0.025]
# Calcul des erreurs pour chaque valeur de H au point final
err = []
y = 0.460857
err.append(abs(y - 0.459861))
err.append(abs(y - 0.460633))
err.append(abs(y - 0.460804))

# Calcul de l'ordre de convergence
order = np.polyfit(np.log(H), np.log(err), 1)[0]
print('Ordre de convergence :', round(order, 2))

plt.figure(figsize=(12, 5))
# Tracé du graphique en double-log pour visualiser la convergence
plt.subplot(1, 2, 1)
plt.loglog(H, err, 'r-o')
plt.grid()

# Tracé du graphique des logarithmes de l'erreur en fonction du logarithme du pas de temps
plt.subplot(1, 2, 2)
plt.plot(np.log(H), np.log(err), 'r-o');
```

```
Ordre de convergence : 2.12
```



Chapitre 18.

TD2 - Exercices sur les schémas multi-pas linéaires et les schémas prédicteur-correcteur

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import fsolve
import sympy as sp
from IPython.display import display, Math, Markdown, Latex
```

18.1. PARTIE I : Étude de schémas multi-pas linéaires

18.1.1. Aide-mémoire

Définitions
Soit $p \geq 0$ un entier et notons $\varphi_{n-j} \overset{\text{déf}}{=} \varphi(t_{n-j}, u_{n-j})$.
Une méthode multi-pas linéaire (MPL) à $q = p + 1 \geq 1$ étapes (=pas) s'écrit

$$u_{n+1} = \sum_{j=0}^{p} \Big(a_j u_{n-j} + h b_j \varphi_{n-j} \Big) + h b_{-1} \varphi(t_{n+1}, u_{n+1}), \qquad n = p, p+1, \dots, N-1$$

où les $\{a_k\}_{j=0}^{p}$ et $\{b_k\}_{j=-1}^{p}$ sont $(2p+1)$ coefficients donnés.

- Méthode explicite : $b_{-1} = 0$.
- Méthode implicite : $b_{-1} \neq 0$.

Théorème de Lax-Richtmyer
Pour $\omega \geq 1$, le schéma est **convergent d'ordre** ω ssi il est **consistant d'ordre** ω **et zéro-stable**.

Zéro-stabilité (comportement à $h \to 0$)
On définit le premier polynôme caractéristique

$$\varrho(r) = r^{p+1} - \sum_{j=0}^{p} a_j r^{p-j}.$$

Condition des racines : la méthode est zéro-stable si toutes les racines r_j de $\varrho(r)$ sont dans le disque unité ($|r_j| \leq 1$), et si celles sur le cercle ($|r_j| = 1$) sont simples.

Consistance et ordre
On définit

$$\xi(i) = \sum_{j=0}^{p} (-j)^i a_j + i \sum_{j=-1}^{p} (-j)^{i-1} b_j, \qquad i = 0, 1, ...$$

Conditions d'ordre :
- Ordre 1 (= consistance) : $\xi(0) = \xi(1) = 1$.
- Ordre ω (= consistance d'ordre ω) : $\xi(0) = \xi(1) = \cdots = \xi(\omega) = 1$.

Première barrière de Dahlquist
Il existe une limite sur l'ordre maximal ω en fonction du nombre de pas q :

$$\omega_{\max} = \begin{cases} q & \text{si } b_{-1} = 0 \text{ (la méthode est explicite)}, \\ q+1 & \text{si } b_{-1} \neq 0 \text{ (la méthode est implicite) et } q \text{ est impair}, \\ q+2 & \text{si } b_{-1} \neq 0 \text{ (la méthode est implicite) et } q \text{ est pair}. \end{cases}$$

Intervalle de A-stabilité
Considérons le problème test $y'(t) = -\beta y(t)$, $y(0) = 1$ avec $\beta > 0$ un nombre réel positif. L'intervalle de stabilité absolue est l'ensemble des valeurs $h(\beta)$ pour lesquelles $u_n \to 0$ lorsque $n \to \infty$.
Soit le polynôme caractéristique

$$\begin{aligned} p_x(r) = \varrho(r) + x\sigma(r) &= r^{p+1} - \sum_{j=0}^{p} a_j r^{p-j} + x \sum_{j=-1}^{p} b_j r^{p-j} \\ &= (1 + xb_{-1}) r^{p+1} - \sum_{j=0}^{p} (a_j - xb_j) r^{p-j}. \end{aligned}$$

Remarque : $x = \beta h = -\lambda h = -z$.
Condition des racines strictes : si, pour $x \in]0, x_{\max}[$ donné, toutes les racines $r_j(x)$ de $p_x(r)$ sont dans le disque unité ($|r_j(x)| < 1$), alors la méthode est A-stable pour le pas $h \in]0, \frac{x_{\max}}{\beta}[$.

Le calcul des quantités $\xi(i)$ n'est pas difficile, mais il est fastidieux. On peut le faire à la main pour des méthodes avec un petit nombre de pas, mais pour des méthodes ayant plusieurs pas il est préférable de le faire avec un module de calcul formel. Voici un exemple avec `sympy` :

```
# ========================================================================
# FONCTION qui calcule xi(i) pour un schéma à q pas
# ========================================================================

def xi(i, q):
    # Définition des symboles pour le cas général
    aa = sp.symbols(f'a_0:{q}')
    bb = sp.symbols(f'b_0:{q}')
    bm1 = sp.Symbol('b_{-1}')
    p = len(aa)
    sa = sum((-j)**i * aa[j] for j in range(p))
    sb = bm1 + sum((-j)**(i-1) * bb[j] for j in range(1, p))
    # Cas particulier : si j=0 et i=0, il calcule (-j)**i =1 et on a bien a_0 dans la somme
    # mais si j=0 et i=1 il ne calcule pas b_0 et il faut l'ajouter à la main
    if i == 1: sb += bb[0]
    return (sa).factor() + (i * sb).factor()

# ========================================================================
# On doit seulement définir p
# ========================================================================

p = 3     # p+1 coeffs du schema d'indices 0 à p
```

```
# ========================================================================
# EXEMPLE D'APPLICATION 1 : calcul de xi(i) pour i = 0, ..., omega
# ========================================================================

# q et omega sont déduits de p, pour omega on considère le cas d'un schéma implicite
q = p + 1
omega = q + 2 if q%2==0 else q + 1 # i = 0, ..., omega

# Affichage des xi(i) séparément
display(Markdown(f"- Pour une méthode à $q={p + 1}$ pas, on affiche $\\xi(i)$ pour $i = 0, \\dots,
 ↪  {omega}$"))
for i in range(omega + 1):
    display(Markdown("$\\qquad$ "+f"$\\xi({i}) = {sp.latex(xi(i, q))}$"))

# Affichage sous forme de système
systeme = [sp.Eq(sp.Symbol(f"\\xi({i})") , sp.Eq(xi(i, q),1)) for i in range(omega + 1)]
display(Markdown(r"$$\begin{cases}"+",\\\\ ".join([sp.latex(eq) for eq in systeme])+r"\end{cases}$$"))


#
 ↪  ========================================================================================================================
# EXEMPLE D'APPLICATION 2 : on peut même résoudre le système {xi(i)=1}_{i=0}^{\omega}
# pour trouver les coefficients qui vérifient la condition de consistance d'ordre le plus élevé
#
 ↪  ========================================================================================================================

# display(Markdown(f"- On résout le système $\\xi(i) = 1$ pour $i = 0, \\dots, {omega}$"))
# systeme = [xi(i, q)-1 for i in range(omega + 1)]
# sol = solve(systeme)
# display(Markdown("$\\qquad$ "+"Les coefficients qui vérifient la condition de consistance d'ordre le
 ↪  plus élevé sont :"))
# for k, v in sol.items():
#     display(Markdown("$\\qquad$ "+f"${k} = {v}$"))
```

- Pour une méthode à $q = 4$ pas, on affiche $\xi(i)$ pour $i = 0, \dots, 6$

$$\xi(0) = a_0 + a_1 + a_2 + a_3$$

$$\xi(1) = -a_1 - 2a_2 - 3a_3 + b_0 + b_1 + b_2 + b_3 + b_{-1}$$

$$\xi(2) = a_1 + 4a_2 + 9a_3 - 2\left(b_1 + 2b_2 + 3b_3 - b_{-1}\right)$$

$$\xi(3) = -a_1 - 8a_2 - 27a_3 + 3\left(b_1 + 4b_2 + 9b_3 + b_{-1}\right)$$

$$\xi(4) = a_1 + 16a_2 + 81a_3 - 4\left(b_1 + 8b_2 + 27b_3 - b_{-1}\right)$$

$$\xi(5) = -a_1 - 32a_2 - 243a_3 + 5\left(b_1 + 16b_2 + 81b_3 + b_{-1}\right)$$

$$\xi(6) = a_1 + 64a_2 + 729a_3 - 6\left(b_1 + 32b_2 + 243b_3 - b_{-1}\right)$$

$$\begin{cases} \xi(0) = a_0 + a_1 + a_2 + a_3 = 1, \\ \xi(1) = -a_1 - 2a_2 - 3a_3 + b_0 + b_1 + b_2 + b_3 + b_{-1} = 1, \\ \xi(2) = a_1 + 4a_2 + 9a_3 - 2\left(b_1 + 2b_2 + 3b_3 - b_{-1}\right) = 1, \\ \xi(3) = -a_1 - 8a_2 - 27a_3 + 3\left(b_1 + 4b_2 + 9b_3 + b_{-1}\right) = 1, \\ \xi(4) = a_1 + 16a_2 + 81a_3 - 4\left(b_1 + 8b_2 + 27b_3 - b_{-1}\right) = 1, \\ \xi(5) = -a_1 - 32a_2 - 243a_3 + 5\left(b_1 + 16b_2 + 81b_3 + b_{-1}\right) = 1, \\ \xi(6) = a_1 + 64a_2 + 729a_3 - 6\left(b_1 + 32b_2 + 243b_3 - b_{-1}\right) = 1 \end{cases}$$

18.1.2. Exercice : consistance d'un schéma à deux pas explicite

On notera $\varphi_k \overset{\text{déf}}{=} \varphi(t_k, u_k)$. La méthode multipas suivante est-elle consistante ?

$$u_{n+1} = u_{n-1} + \frac{h}{4}(3\varphi_n - \varphi_{n-1})$$

C'est une méthode à q=2 pas explicite :

- $p = 1$
- $b_{-1} = 0$
- $a_0 = 0$ et $a_1 = 1$
- $b_0 = \frac{3}{4}$ et $b_1 = -\frac{1}{4}$

La méthode n'est pas consistante car

$$\begin{cases} \xi(0) = a_0 + a_1 = 1, \\ \xi(1) = -a_1 + b_0 + b_1 + b_{-1} = -1 + \frac{3}{4} - \frac{1}{4} \neq 1 \end{cases}$$

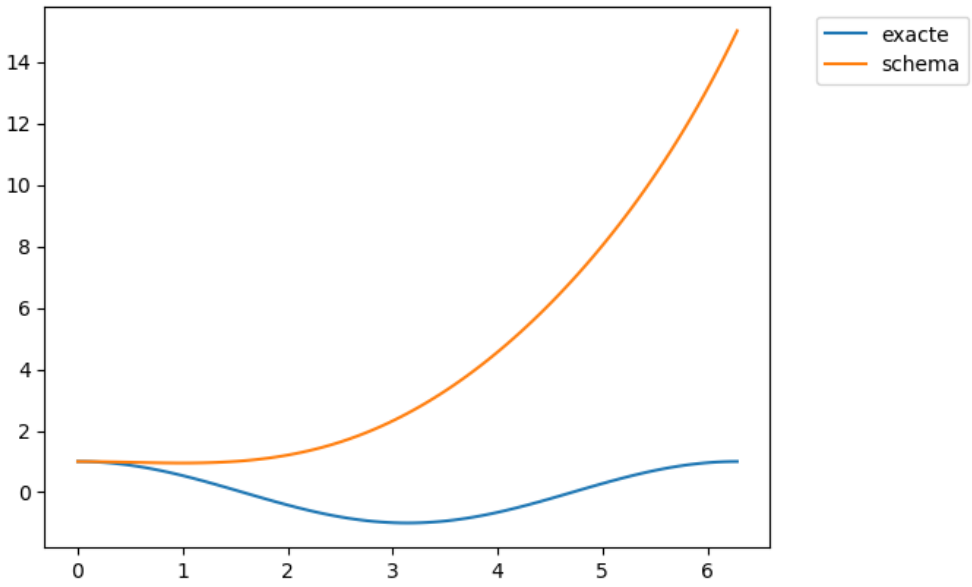
Voyons sur un exemple ce qui se passe lorsqu'on utilise un schéma non consistant. On compare la solution exacte et la solution approchée du problème de Cauchy $y'(t) = 2(y(t) - \cos(t)) - \sin(t)$, $y(0) = 1$ sur l'intervalle $[0, 2\pi]$ dont la solution exacte est $y(t) = cos(t)$. Comme visible sur la figure suivante, la solution numérique ne correspond pas à la solution exacte et on peut vérifier qu'un comportement similaire se produit également si on raffine la discrétisation. Il s'agit d'une situation typique de manque de consistance : l'application d'une méthode non consistante produit l'allure d'une autre fonction plutôt que celle reproduisant la solution exacte.

```
t0, y0 = 0,1

# Solution exacte
t = sp.symbols('t')
y = sp.Function('y')
edo = sp.Eq(sp.diff(y(t),t) , 2*(y(t)-sp.cos(t))-sp.sin(t) )
# display(edo)
exacte_sp = sp.dsolve( edo , y(t),  ics={y(t0):y0})
# display(exacte_sp)


# Solution approchée
def schema(tt,phi,sol_exacte):
    uu = np.zeros_like(tt)
    q = 2
    uu[:q] = sol_exacte(tt[:q])
    h = tt[1]-tt[0]
    for n in range(q-1,len(tt)-1):
        k0 = phi(tt[n],uu[n])
        k1 = phi(tt[n-1],uu[n-1])
        uu[n+1] = uu[n-1] + h/4*(3*k0-k1)
    return uu


# Comparaison
sol_exacte = sp.lambdify(t, exacte_sp.rhs, 'numpy')
tfin = 2*np.pi
phi =  lambda t,y: 2*(y-np.cos(t))-np.sin(t)
N = 100
tt = np.linspace(t0,tfin,1+N)
yy = sol_exacte(tt)
uu = schema(tt,phi,sol_exacte)
plt.plot(tt,yy,label='exacte')
plt.plot(tt,uu,label='schema')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left');
```



18.1.3. Exercice : convergence d'un schéma à 2 pas implicite

On notera $\varphi_k \overset{\text{déf}}{=} \varphi(t_k, u_k)$. La méthode multipas suivante est-elle convergente ?

$$u_{n+1} = -4u_n + 5u_{n-1} + h(4\varphi_{n+1} + 2\varphi_n)$$

C'est une méthode à 2 pas implicite :

- $p = 1$
- $b_{-1} = 4$
- $a_0 = -4$ et $a_1 = 5$
- $b_0 = 2$ et $b_1 = 0$

La méthode est consistante car

$$\begin{cases} \xi(0) = a_0 + a_1 = -4 + 5 = 1, \\ \xi(1) = -a_1 + b_0 + b_1 + b_{-1} = -5 + 4 + 2 + 0 = 1. \end{cases}$$

Le premier polynôme caractéristique est

$$\begin{aligned} \varrho(r) = r^{p+1} - \sum_{j=0}^{p} a_j r^{n-j} &= r^2 - (a_0 r^1 + a_1 r^0) \\ &= r^2 - (-4r + 5) = r^2 + 4r - 5 = (r-1)(r+5) \end{aligned}$$

dont les racines sont $r_0 = 1$ et $r_1 = -5$. La méthode n'est pas zéro-stable car $|r_1| > 1$, donc la méthode ne converge pas.

Voyons sur un exemple ce qui se passe lorsqu'on utilise un schéma consistant mais non zéro-stable. On compare la solution exacte et la solution approchée du problème de Cauchy $y'(t) = 2(y(t) - \cos(t)) - \sin(t)$, $y(0) = 1$ sur l'intervalle $[0, 2\pi]$ dont la solution exacte est $y(t) = cos(t)$.

La cosistance est une propriété de précision locale, tandis que la zéro-stabilité et la convergence sont des propriétés de précision globale. La zéro-stabilité assure que la solution numérique ne diverge pas lorsque h tend vers 0.

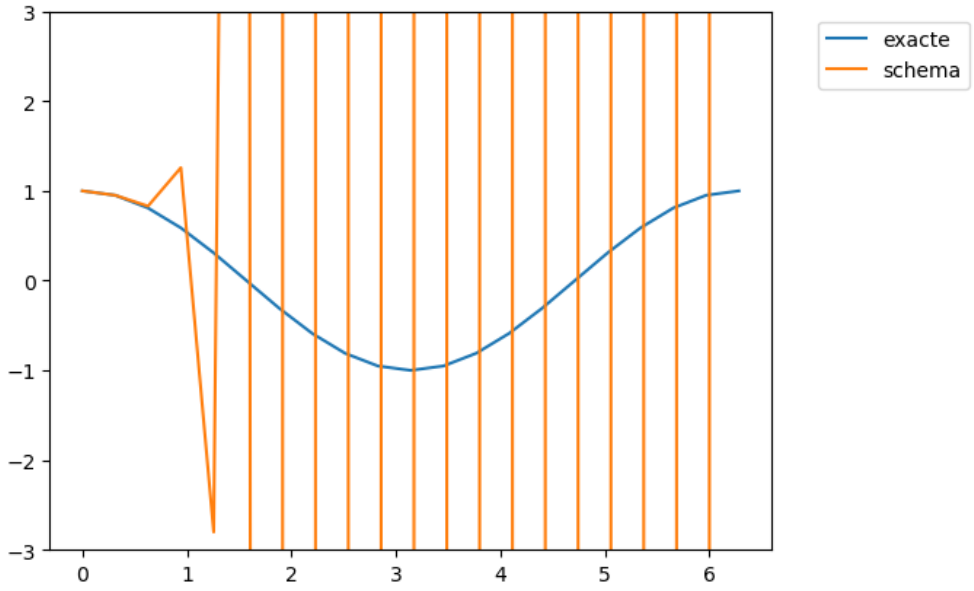
```
t0, y0 = 0,1

# Solution exacte
t = sp.symbols('t')
```

```
y = sp.Function('y')
edo = sp.Eq(sp.diff(y(t),t) , 2*(y(t)-sp.cos(t))-sp.sin(t) )
# display(edo)
exacte_sp = sp.dsolve( edo , y(t),  ics={y(t0):y0})
# display(exacte_sp)

# Solution approchée
def schema(tt,phi,sol_exacte):
    uu = np.zeros_like(tt)
    q = 2
    uu[:q] = sol_exacte(tt[:q])
    h = tt[1]-tt[0]
    for n in range(q-1,len(tt)-1):
        eq = lambda x : -x -4*uu[n]+5*uu[n-1] + h*(2*phi(tt[n],uu[n])+phi(tt[n+1],x))
        uu[n+1] =  fsolve(eq,uu[n])[0]
    return uu


# Comparaison
sol_exacte = sp.lambdify(t, exacte_sp.rhs, 'numpy')
tfin = 2*np.pi
phi =  lambda t,y: 2*(y-np.cos(t))-np.sin(t)
N = 20
tt = np.linspace(t0,tfin,1+N)
yy = sol_exacte(tt)
uu = schema(tt,phi,sol_exacte)
plt.plot(tt,yy,label='exacte')
plt.plot(tt,uu,label='schema')
plt.ylim(-3,3)
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left');
```



18.1.4. Exercice : convergence d'un schéma à 3 pas explicite

On notera $\varphi_k \overset{\text{déf}}{=} \varphi(t_k, u_k)$. La méthode multipas suivante est-elle convergente ?

$$u_{n+1} = -u_n + u_{n-1} + u_{n-2} + 4h\varphi_n$$

C'est une méthode à 3 pas explicite :

- $p = 2$

- $b_{-1} = 0$
- $a_0 = -1$, $a_1 = 1$ et $a_2 = 1$
- $b_0 = 4$, $b_1 = 0$ et $b_2 = 0$

La méthode est consistante car

$$\begin{cases} \xi(0) = a_0 + a_1 + a_2 = -1 + 1 + 1 = 1, \\ \xi(1) = -a_1 - 2a_2 + b_0 + b_1 + b_2 + b_{-1} = -1 - 2 + 4 + 0 + 0 + 0 = -3 + 4 = 1. \end{cases}$$

Le premier polynôme caractéristique est

$$\varrho(r) = r^{p+1} - \sum_{j=0}^{p} a_j r^{n-j} = r^3 - (-r^2 + r + 1) = r^3 + r^2 - r - 1 = (r-1)(r+1)^2$$

dont les racines sont $r_0 = 1$ de multiplicité 2 et $r_1 = -1$. La méthode n'est pas zéro-stable, donc la méthode ne converge pas.

```
sp.var('r')
rho = r**3+r**2-r-1
sp.factor(rho)
```

$(r-1)\,(r+1)^2$

18.1.5. Exercice : schéma à deux pas d'ordre 4

La première barrière de Dahlquist affirme qu'un schéma à $q = 2$ pas consistant et zéro-stable peut être d'ordre $\omega = q + 2 = 4$. (Pour cela on sait qu'il sera implicite).

- Construire l'unique schéma MPL implicite à $q = 2$ pas convergent d'ordre $\omega = q + 2 = 4$.
- En étudier sa stabilité absolue.

Un schéma implicite à $q = 2$ étapes s'écrit

$$u_{n+1} = a_0 u_n + a_1 u_{n-1} + h(b_{-1}\varphi_{n+1} + b_0 \varphi_n + b_1 \varphi_{n-1})$$

La méthode est **consistante d'ordre 4** ssi

$$\begin{cases} \xi(0) = a_0 + a_1 = 1, \\ \xi(1) = -a_1 + b_0 + b_1 + b_{-1} = 1, \\ \xi(2) = a_1 - 2\,(b_1 - b_{-1}) = 1, \\ \xi(3) = -a_1 + 3\,(b_1 + b_{-1}) = 1, \\ \xi(4) = a_1 - 4\,(b_1 - b_{-1}) = 1 \end{cases}$$

Sous forme matricielle on trouve

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & -1 & 1 & 1 \\ 0 & 0 & 1 & -2 & 2 \\ 0 & 0 & -1 & 3 & 3 \\ 0 & 0 & 1 & -4 & 4 \end{pmatrix} \begin{pmatrix} a_0 \\ b_0 \\ a_1 \\ b_1 \\ b_{-1} \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

On peut résoudre ce système avec la méthode de triangulation de Gauss :

$$\left(\begin{array}{ccccc|c} 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & -1 & 1 & 1 & 1 \\ 0 & 0 & 1 & -2 & 2 & 1 \\ 0 & 0 & -1 & 3 & 3 & 1 \\ 0 & 0 & 1 & -4 & 4 & 1 \end{array}\right) \to \left(\begin{array}{ccccc|c} 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & -1 & 1 & 1 & 1 \\ 0 & 0 & 1 & -2 & 2 & 1 \\ 0 & 0 & 0 & 1 & 5 & 2 \\ 0 & 0 & 0 & -2 & 2 & 0 \end{array}\right) \to \left(\begin{array}{ccccc|c} 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & -1 & 1 & 1 & 1 \\ 0 & 0 & 1 & -2 & 2 & 1 \\ 0 & 0 & 0 & 1 & 5 & 2 \\ 0 & 0 & 0 & 0 & 12 & 4 \end{array}\right)$$

donc $b_{-1} = \frac{1}{3}$, $b_0 = 2 - 5b_{-1} = \frac{4}{3}$, $b_1 = 1 - 2b_{-1} + 2b_1 = \frac{1}{3}$, $a_0 = 1$ et $a_1 = 0$.

```
a_0 = sp.symbols('a_0')
a_1 = sp.symbols('a_1')
b_m1 = sp.symbols('b_{-1}')
b_0 = sp.symbols('b_0')
b_1 = sp.symbols('b_1')

# p=1 => q=2 et omega=4
eq1 = sp.Eq(a_0 + a_1, 1) # eq1 = a_0+a_1-1
eq2 = sp.Eq(-a_1 + b_m1 + b_0 + b_1, 1) # eq2 = -a_1+b_{-1}+b_0+b_1-1
eq3 = sp.Eq(a_1 + 2 * b_m1 - 2 * b_1, 1) # eq3 = a_1+2*b_{-1}-2*b_1-1
eq4 = sp.Eq(-a_1 + 3 * b_m1 + 3 * b_1, 1) # eq4 = -a_1+3*b_{-1}+3*b_1-1
eq5 = sp.Eq(a_1 + 4 * b_m1 - 4 * b_1, 1) # eq5 = a_1+4*b_{-1}-4*b_1-1
sol = sp.solve([eq1, eq2, eq3, eq4, eq5]) # résolution du système d'équations
display(sol)

# Résultat
u_np1 = sp.symbols('u_{n+1}')
u_n = sp.symbols('u_n')
u_nm1 = sp.symbols('u_{n-1}')
phi_np1 = sp.symbols(r'\varphi_{n+1}')
phi_n = sp.symbols(r'\varphi_n')
phi_nm1 = sp.symbols(r'\varphi_{n-1}')
h = sp.symbols('h')
eq = sp.Eq(u_np1, (a_0 * u_n + a_1 * u_nm1 + h * (b_m1 * phi_np1 + b_0 * phi_n + b_1 *
 ↪  phi_nm1)).subs(sol))
display(eq)
```

```
{a_0: 0, a_1: 1, b_0: 4/3, b_1: 1/3, b_{-1}: 1/3}
```

$$u_{n+1} = h\left(\frac{4\varphi_n}{3} + \frac{\varphi_{n+1}}{3} + \frac{\varphi_{n-1}}{3}\right) + u_{n-1}$$

On reconnait la méthode MS_2.

Le premier polynôme caractéristique est

$$\varrho(r) = r^{p+1} - \sum_{j=0}^{p} a_j r^{p-j} = r^2 - a_0 r - a_1 = (r-1)(r+1)$$

dont les racines sont $r_0 = 1$ et $r_1 = -1$. La méthode est **zéro-stable** car les deux racines sont sur le cercle unité et simples.

Étudions maintenant la A-stabilité de ce schéma. Le polynôme caractéristique est

$$\begin{aligned} p_x(r) &= \varrho(r) + x\sigma(r) = (1 + xb_{-1})r^2 - (a_0 - xb_0)r - (a_1 - xb_1) \\ &= \left(1 + \frac{x}{3}\right) r^2 - \frac{4x}{3} r - \left(1 - \frac{x}{3}\right) \\ &= \frac{(3+x)\, r^2 - 4xr - (3-x)}{3}. \end{aligned}$$

Les racines de ce polynôme sont

$$r_{1,2}(x) = \frac{2x \pm \sqrt{9 + 3x^2}}{3 + x}$$

et la suite s'écrit

$$u_{n+1} = Ar_1^n(x) + Br_2^n(x)$$

pour des constantes A et B dépendant des conditions initiales.

Si $|r_1(x)| < 1$ et $|r_2(x)| < 1$ alors $u_n \to 0$ lorsque $n \to \infty$. Dans notre cas, $|r_1(x)| > 1$ pour tout $x > 0$, donc la méthode n'est pas A-stable. Le terme associé à cette racine est désigné dans la littérature sous le nom de « composante parasite ». Ces composantes nuisent à la précision globale (à long terme) de la méthode numérique.

```
r, x = sp.symbols('r x')
poly = (3 + x)*r**2 - 4*x*r - (3 - x)
roots = sp.solve(poly, r)  # liste de racines symboliques
for i, root in enumerate(roots):
    display(Math(f"r_{i+1}(x) = {sp.latex(root)}"))

# Conversion en fonctions numpy pour évaluation
r1_func = sp.lambdify(x, roots[0], 'numpy')
r2_func = sp.lambdify(x, roots[1], 'numpy')

x_vals = np.linspace(0, 10, 100)
r_1 = r1_func(x_vals)
r_2 = r2_func(x_vals)

# Tracé
plt.figure(figsize=(8,5))

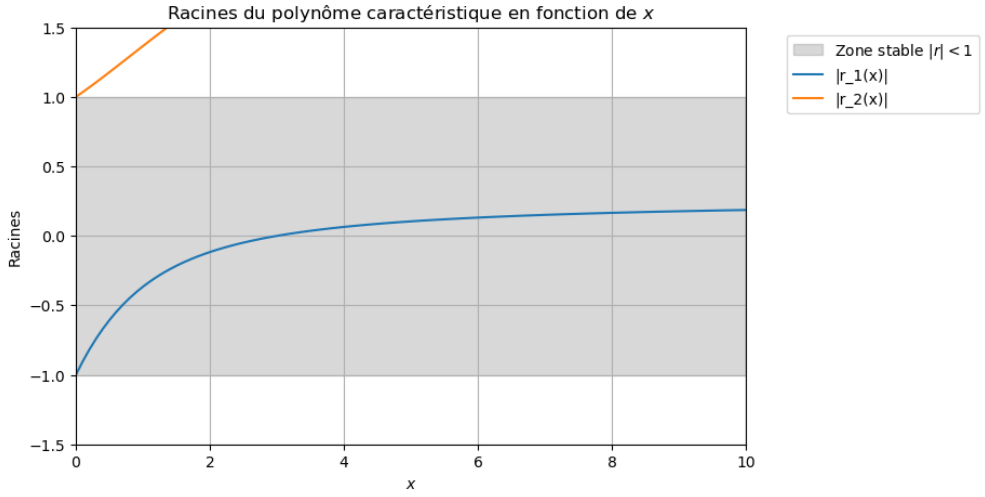
# Zone stable (|r|<1)
plt.fill_between(x_vals, -1, 1, color='gray', alpha=0.3, label=r'Zone stable $|r|<1$')

# Courbes des racines
plt.plot(x_vals, r_1, label=r'|r_1(x)|')
plt.plot(x_vals, r_2, label=r'|r_2(x)|')


plt.ylim(-1.5, 1.5)
plt.xlim(x_vals[0], x_vals[-1])
plt.xlabel('$x$')
plt.ylabel('Racines')
plt.title('Racines du polynôme caractéristique en fonction de $x$')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.grid(True)
plt.show()
```

$$r_1(x) = \frac{2x - \sqrt{3x^2 + 9}}{x + 3}$$

$$r_2(x) = \frac{2x + \sqrt{3x^2 + 9}}{x + 3}$$



Voici une figure montrant la solution numérique du problème de Cauchy $y'(t) = -\beta y(t)$, $y(0) = 1$ avec $\beta = 1$ sur l'intervalle $[0, 10]$ obtenue avec la méthode MS_2. On voit que la solution numérique diverge de la solution exacte $y(t) = e^{-t}$ à mesure que le temps augmente, ce qui est une manifestation de la composante parasite associée à la racine $r_1(x)$. Si vous changez l'intervalle de temps pour $[0, 1]$ vous verrez que la solution numérique est très proche de la solution exacte, ce qui illustre le fait que la composante parasite a un effet à long terme.

```
t_fin = 10

# ============================================================================

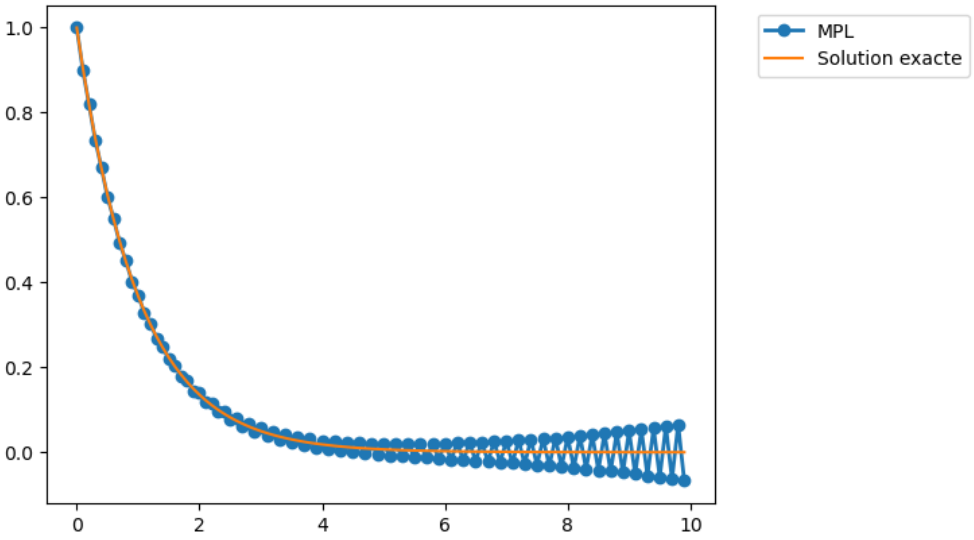
h = 0.1

phi = lambda t, y: -y
t0, y0 = 0,1
sol_exacte = lambda t: y0*np.exp(-t)

def MS2(phi,tt,yy_init):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    # initialisation
    q = 2
    uu[:q] = yy_init
    # récurrence
    for n in range(q-1,len(tt)-1):
        k0 = phi( tt[n], uu[n] )
        k1 = phi( tt[n-1], uu[n-1] )
        temp = fsolve(lambda x: -x+uu[n-1]+h*(phi(tt[n+1],x)+4*k0+k1)/3, uu[n])
        uu[n+1] = temp[0]
    return uu

tt = np.arange(t0, t_fin, h)
yy = sol_exacte(tt)
y1 = y0+h*phi(tt[0],y0)
uu = MS2(phi, tt, [y0, y1])
plt.plot(tt, uu, 'o-', label=f'MPL', lw=2)

plt.plot(tt, yy, label='Solution exacte')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()
```



18.1.6. Exercice : convergence d'un schéma à 2 pas implicite

On notera $\varphi_k \overset{\text{déf}}{=} \varphi(t_k, u_k)$. Soit la méthode multipas

$$u_{n+1} = (1-\gamma)u_n + \gamma u_{n-1} + \frac{h}{4}((\gamma+3)\varphi_{n+1} + (3\gamma+1)\varphi_{n-1}).$$

1. Pour quelles valeurs de γ est-elle consistante ?
2. Pour quelles valeurs de γ est-elle zéro-stable ?
3. Pour quelles valeurs de γ est-elle convergente ?
4. Pour quelles valeurs de γ est-elle convergente d'ordre 2 ? et d'ordre 3 ?
5. Pour $\gamma = \frac{1}{2}$ vérifier empiriquement l'ordre de convergence sur le problème de Cauchy

$$\begin{cases} y'(t) = -4y(t) + t^2, \quad \forall t \in I = [0,1], \\ y(0) = 1 \end{cases}$$

6. Pour $\gamma = \frac{1}{2}$, étudier la A-stabilité de ce schéma.

C'est une méthode à 2 pas implicite :

- $p = 1$
- $b_{-1} = \frac{\gamma+3}{4}$
- $a_0 = 1-\gamma$ et $a_1 = \gamma$
- $b_0 = 0$ et $b_1 = \frac{3\gamma+1}{4}$

Consistance La méthode est consistante pour tout γ car

$$\begin{cases} \xi(0) = a_0 + a_1 = 1 - \gamma + \gamma = 1 \text{ pour tout } \gamma, \\ \xi(1) = -a_1 + b_0 + b_1 + b_{-1} = -\gamma + (0 + \frac{3\gamma+1}{4} + \frac{\gamma+3}{4}) = 1 \text{ pour tout } \gamma \end{cases}$$

Zéro-stabilité Le premier polynôme caractéristique est

$$\varrho(r) = r^{p+1} - \sum_{j=0}^{p} a_j r^{p-j} = r^2 - (1-\gamma)r - \gamma = (r-1)(r+\gamma)$$

dont les racines sont $r_0 = 1$ et $r_1 = -\gamma$. La méthode est zéro-stable ssi

$$\begin{cases} |r_j| \leq 1 \quad \text{pour tout } j = 0, \dots, p \\ \text{si } r_j \text{ a multiplicité} > 1 \quad \Longrightarrow \quad |r_j| < 1 \end{cases}$$

donc ssi $-1 \leq -\gamma \leq 1$ et $-\gamma \neq 1$, i.e. ssi $-1 < \gamma \leq 1$.

Convergence La méthode est convergente ssi elle est consistante et zéro-stable donc ssi $\boxed{-1 < \gamma \leq 1}$.

Ordre de convergence La méthode est d'ordre au moins 2 pour toute valeur de γ pour laquelle elle est convergente, car

$$\xi(2) = a_1 - 2\,(b_1 - b_{-1}) = \gamma - 2\left(\frac{3\gamma+1}{4} - \frac{\gamma+3}{4}\right) = 1 \text{ pour tout } \gamma.$$

Pour que la méthode soit d'ordre 3 il faut que $\xi(3) = 1$. On a

$$\xi(3) = -a_1 + 3\,(b_1 + b_{-1}) = -\gamma + 3\left(\frac{3\gamma+1}{4} + \frac{\gamma+3}{4}\right) = 2\gamma + 3 = 1 \quad \iff \quad \gamma = -1.$$

Puisque cette valeur n'appartient pas au domaine de zéro-stabilité, on conclut que la méthode est convergente d'ordre 2 pour tout $-1 < \gamma \leq 1$ et non convergente sinon.

Vérification empirique On définit la solution exacte (calculée en utilisant le module `sympy`) pour estimer les erreurs, puis on définit le schéma et enfin on calcule les erreurs pour différentes valeurs de h.

```
# variables globales
t0     = 0
tfinal = 1
y0     = 1

phi = lambda t,y : -4*y+t**2


t  = sp.Symbol('t')
y  = sp.Function('y')
edo = sp.Eq( sp.diff(y(t),t) , phi(t,y(t)) )
display(edo)

solpar = sp.dsolve(edo, ics={y(t0):y0})
display(solpar)

sol_exacte = sp.lambdify(t,solpar.rhs,'numpy')
```

$$\frac{d}{dt}y(t) = t^2 - 4y(t)$$

$$y(t) = \frac{t^2}{4} - \frac{t}{8} + \frac{1}{32} + \frac{31e^{-4t}}{32}$$

On calcule la solution approchée pour différentes valeurs de N :

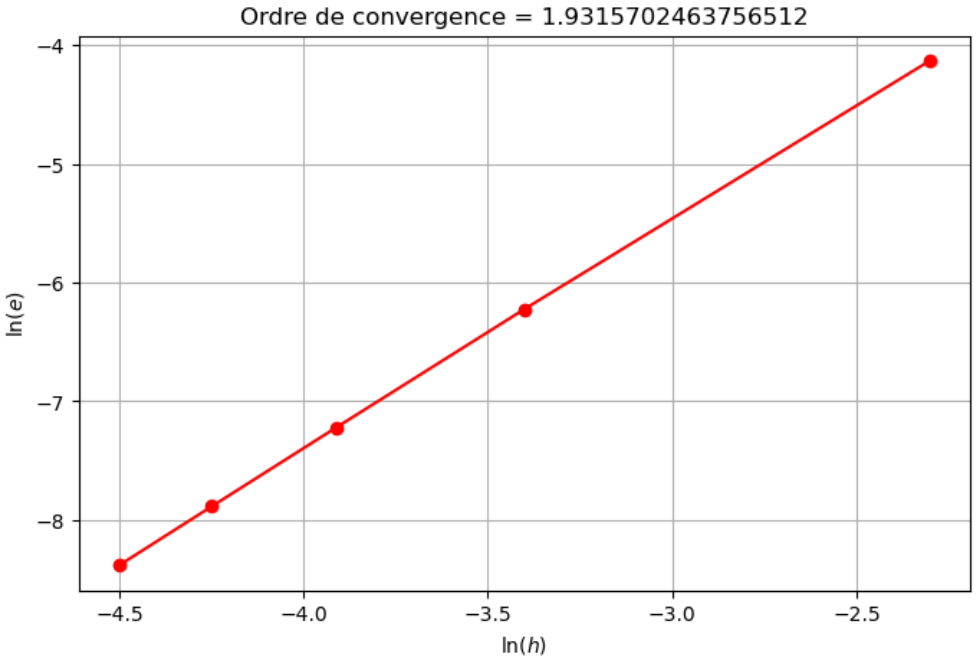
```
def multipas(phi, tt, yy, gamma):
    h  = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    # initialisation
    q = 2
    uu[:q] = yy[:q]
    # recurrence
    for n in range(q-1,len(tt) - 1):
        temp = fsolve(lambda x: -x + (1-gamma)*uu[n]+gamma*uu[n-1] + h/4 * (
↪   (gamma+3)*phi(tt[n+1],x)+(3*gamma+1)*phi(tt[n-1],uu[n-1]) ), uu[n])
        uu[n+1] = temp[0]
    return uu
```

```
H = []
err_mp = []
gamma = 1/2

for N in range(10,100,20):
    tt = np.linspace(t0, tfinal, N + 1)
    h = tt[1] - tt[0]
    yy = sol_exacte(tt)
    uu_mp = multipas(phi, tt, yy, gamma)
    H.append(h)
    err_mp.append(  np.linalg.norm(uu_mp-yy, np.inf) )



plt.figure(figsize=(8,5))
plt.plot(np.log(H), np.log(err_mp), 'r-o')
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
plt.title(f'Ordre de convergence = {(np.polyfit(np.log(H),np.log(err_mp), 1)[0])}')
plt.grid(True)
```



Intervalle de A-stabilité. Pour étudier la A-stabilité de ce schéma, on considère le problème test $y'(t) = -\beta y(t)$, $y(0) = 1$ avec $\beta > 0$ et on pose $x = h\beta$. Le polynôme caractéristique est

$$\begin{aligned} p_x(r) &= \varrho(r) + x\sigma(r) = (1 + xb_{-1})r^2 - (a_0 + xb_0)r - (a_1 + xb_1) \\ &= \left(1 + \frac{x}{4}(\gamma + 3)\right) r^2 - (1 - \gamma)r - \left(\gamma - \frac{x}{4}(3\gamma + 1)\right) \end{aligned}$$

Pour $\gamma = \frac{1}{2}$, on trouve

$$p_x(r) = \left(1 + \frac{7x}{8}\right) r^2 - \frac{1}{2}r - \left(\frac{5x}{8} - \frac{1}{2}\right)$$

En traçant la valeur absolue des racines en fonction de x, on peut visualiser l'intervalle de stabilité absolue. On trouve que la méthode est A-stable pour tout $x > 0$. En effet, les racines sont complexes conjuguées et de module strictement inférieur à 1 pour tout $x > 0$. Par conséquent, la méthode est A-stable pour tout $h > 0$.

```
r = sp.Symbol('r')
x = sp.Symbol('x', real=True, positive=True)
gamma = sp.Symbol('gamma')

p = 1
aa = [1-gamma, gamma]
bb = [0, (3*gamma+1)/4]
bm1 = (gamma+3)/4

# Construction du polynôme
rho_expr = r**(p+1) - sum(aa[i]*r**(p-i) for i in range(p+1))
sigma_expr = bm1*r**(p+1) + sum(bb[i]*r**(p-i) for i in range(p+1))
poly = sp.Poly(rho_expr + x * sigma_expr, r)

display(Math(f"p_x(r) = {sp.latex(poly.as_expr())}"))

coeffs = poly.all_coeffs() # Renvoie [A, B, C]
display(Math(rf"A(x) = {sp.latex(coeffs[0])}"))
display(Math(rf"B(x) = {sp.latex(coeffs[1])}"))
display(Math(rf"C(x) = {sp.latex(coeffs[2])}"))

# ============================================================================

# Cas gamma = 1/2
gamma_val = sp.Rational(1, 2)
poly_gamma = sp.Poly(poly.subs(gamma, gamma_val), r)

display(Math(f"$p_x(r)$ pour $\\gamma=1/2$: ${sp.latex(poly_gamma.as_expr())}$"))

display(Math(rf"A(x) = {sp.latex(coeffs[0].subs(gamma, gamma_val))}"))
display(Math(rf"B(x) = {sp.latex(coeffs[1].subs(gamma, gamma_val))}"))
display(Math(rf"C(x) = {sp.latex(coeffs[2].subs(gamma, gamma_val))}"))

roots = sp.solve(poly_gamma, r)

display(Markdown(f"Racines : $r_1(x) = {sp.latex(roots[0])}$, $r_2(x) = {sp.latex(roots[1])}$"))

# On définit une fonction qui calcule le module manuellement pour éviter le sqrt(négatif) dans numpy
def get_abs_expr(root):
    re_part = sp.re(root)
    im_part = sp.im(root)
    return sp.sqrt(re_part**2 + im_part**2)

abs_r1_expr = get_abs_expr(roots[0])
abs_r2_expr = get_abs_expr(roots[1])

r1_func = sp.lambdify(x, abs_r1_expr, modules=['numpy'])
r2_func = sp.lambdify(x, abs_r2_expr, modules=['numpy'])


# Le discriminant s'annule en
x_bifurcation = sp.solve(poly_gamma.discriminant(), x)[0]
display(Markdown(f"Point de bifurcation (racines fusionnent) : $x = {sp.latex(x_bifurcation)}$"))

x_vals = np.linspace(0, 5, 1000)

r_abs_1 = r1_func(x_vals)
r_abs_2 = r2_func(x_vals)


plt.figure(figsize=(10, 6))

plt.axhspan(0, 1, color='gray', alpha=0.1, label=r'Zone stable $|r| \leq 1$')
```

```
plt.plot(x_vals, r_abs_1, label=r'$|r_1(x)|$', lw=2)
plt.plot(x_vals, r_abs_2, label=r'$|r_2(x)|$', lw=2, linestyle='--')

# Point où les racines fusionnent (deviennent complexes)
plt.axvline(x_bifurcation, color='orange', linestyle=':', label=f'Bifurcation x = {x_bifurcation}')

plt.title(r'Module des racines ($y\' = -\beta y, \gamma=1/2$)')
plt.xlabel('$x = h\\beta$')
plt.ylabel('$|r(x)|$')
plt.legend()
plt.grid(True, alpha=0.3)
plt.ylim(0, 1.2)
plt.show()
```

$$p_x(r) = \frac{3\gamma x}{4} - \gamma + r^2\left(\frac{\gamma x}{4} + \frac{3x}{4} + 1\right) + r\left(\gamma - 1\right) + \frac{x}{4}$$

$$A(x) = \frac{\gamma x}{4} + \frac{3x}{4} + 1$$

$$B(x) = \gamma - 1$$

$$C(x) = \frac{3\gamma x}{4} - \gamma + \frac{x}{4}$$

$p_x(r)$ pour $\gamma = 1/2$: $r^2 \cdot \left(\frac{7x}{8} + 1\right) - \frac{r}{2} + \frac{5x}{8} - \frac{1}{2}$

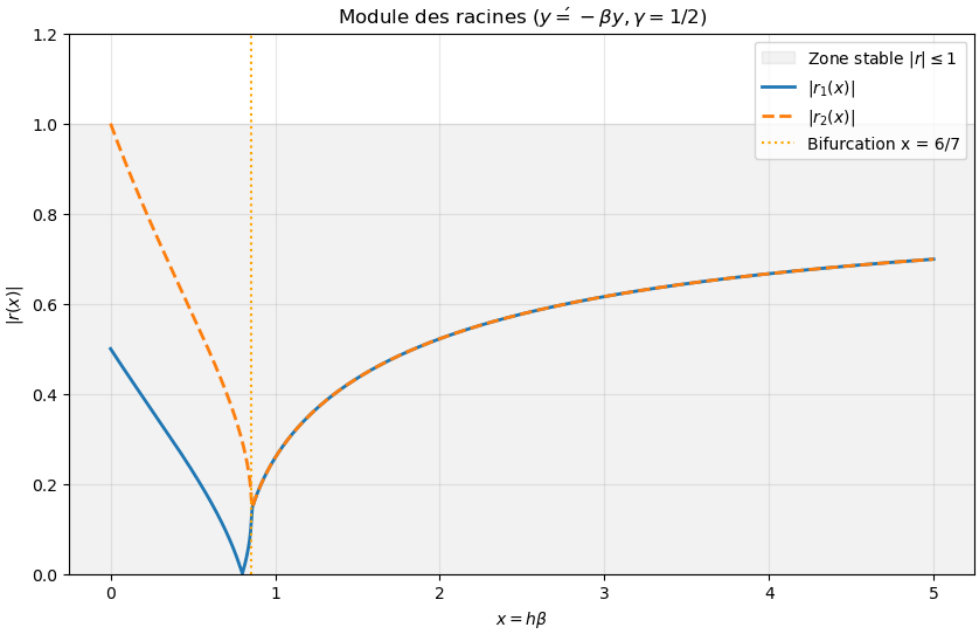
$$A(x) = \frac{7x}{8} + 1$$

$$B(x) = -\frac{1}{2}$$

$$C(x) = \frac{5x}{8} - \frac{1}{2}$$

Racines : $r_1(x) = \frac{2-\sqrt{-35x^2-12x+36}}{7x+8}$, $r_2(x) = \frac{\sqrt{-35x^2-12x+36}+2}{7x+8}$

Point de bifurcation (racines fusionnent) : $x = \frac{6}{7}$



18.1.7. Exercice : convergence d'un schéma à 2 pas explicite

On notera $\varphi_k \overset{\text{déf}}{=} \varphi(t_k, u_k)$. Soit la méthode multipas

$$u_{n+1} = \left(\frac{1}{3}\gamma^2 + \frac{1}{2}\gamma\right) u_n + \left(\frac{2}{3}\gamma^2 + \frac{1}{2}\gamma - 1\right) u_{n-1} + h\left[\left(-\frac{1}{6}\gamma^2 - \frac{1}{4}\gamma + 2\right)\varphi_n + \left(-\frac{1}{6}\gamma^2 - \frac{1}{4}\gamma\right)\varphi_{n-1}\right].$$

1. Pour quelles valeurs de γ est-elle consistante ?
2. Est-elle convergente pour ces valeurs ?
3. Quel ordre de convergente a-t-elle ?
4. Pour les valeurs de γ qui donnent la convergence, vérifier empiriquement l'ordre de convergence sur le problème de Cauchy
$$\begin{cases} y'(t) = -10(y(t)-1)^2, & \forall t \in I = [0,1], \\ y(0) = 2. \end{cases}$$
5. Pour les valeurs de γ qui donnent la convergence, étudier la A-stabilité de ce schéma.

C'est une méthode à $q = 2$ pas explicite :

- $p = 1$
- $b_{-1} = 0$
- $a_0 = \frac{1}{3}\gamma^2 + \frac{1}{2}\gamma$ et $a_1 = \frac{2}{3}\gamma^2 + \frac{1}{2}\gamma - 1$
- $b_0 = -\frac{1}{6}\gamma^2 - \frac{1}{4}\gamma + 2$ et $b_1 = -\frac{1}{6}\gamma^2 - \frac{1}{4}\gamma$

Consistance
La méthode est consistante ssi

$$\begin{cases} \xi(0) = a_0 + a_1 = 1, \\ \xi(1) = -a_1 + b_0 + b_1 + b_{-1} = 1 \end{cases} \iff \begin{cases} \gamma^2 + \gamma - 1 = 1, \\ -\gamma^2 - \gamma + 3 = 1 \end{cases} \iff \gamma^2 + \gamma - 2 = 0$$

donc ssi $\gamma = -2$ ou $\gamma = 1$. On obtient alors les valeurs des coefficients suivantes :

γ	a_0	a_1	b_0	b_1	b_{-1}
-2	$\frac{1}{3}$	$\frac{2}{3}$	$\frac{11}{6}$	$-\frac{1}{6}$	0
1	$\frac{5}{6}$	$\frac{1}{6}$	$\frac{19}{12}$	$-\frac{5}{12}$	0

Zéro-stabilité
On étudie la zéro-stabilité seulement pour les valeurs de γ qui donnent la consistance. Le premier polynôme caractéristique est

$$\varrho(r) = \begin{cases} r^2 - \frac{1}{3}r - \frac{2}{3} & \text{si } \gamma = -2, \\ r^2 - \frac{5}{6}r - \frac{1}{6} & \text{si } \gamma = 1, \end{cases}$$

dont les racines sont

$$\begin{cases} r_0 = 1,\ r_1 = -\frac{2}{3} & \text{si } \gamma = -2, \\ r_0 = 1,\ r_1 = -\frac{1}{6} & \text{si } \gamma = 1. \end{cases}$$

La méthode est zéro-stable ssi

$$\begin{cases} |r_j| \leq 1 \quad \text{pour tout } j = 0, \dots, p \\ \text{si } r_j \text{ a multiplicité} > 1 \end{cases} \implies |r_j| < 1$$

Cette condition est vérifiée pour les deux cas.

Convergence
La méthode est convergente ssi elle est consistante et zéro-stable donc ssi $\gamma = -2$ ou $\gamma = 1$.

Ordre
La méthode est d'ordre au moins 2 pour les deux choix de γ car

$$\xi(2) = a_1 - 2\,(b_1 - b_{-1}) = \begin{cases} \frac{2}{3} + 2\frac{1}{6} = 1 & \text{si } \gamma = -2, \\ \frac{1}{6} + 2\frac{5}{12} = 1 & \text{si } \gamma = 1, \end{cases}$$

La méthode n'est d'ordre 3 pour aucun des deux choix de γ car

$$\xi(3) = -a_1 + 3\,(b_1 + b_{-1}) = \begin{cases} -\frac{2}{3} + 3\frac{-1}{6} \neq 1 & \text{si } \gamma = -2, \\ -\frac{1}{6} + 3\frac{-5}{12} \neq 1 & \text{si } \gamma = 1, \end{cases}$$

Solution exacte
On définit la solution exacte (calculée en utilisant le module `sympy`) pour estimer les erreurs :

```
t0     = 0
tfinal = 1
y0     = 2

phi = lambda t,y : -10*(y-1)**2


t  = sp.Symbol('t')
y  = sp.Function('y')
edo= sp.Eq( sp.diff(y(t),t) , phi(t,y(t)) )
display(edo)
solpar = sp.dsolve(edo, ics={y(t0): y0})
display(solpar)

sol_exacte = sp.lambdify(t,solpar.rhs,'numpy')
```

$$\frac{d}{dt}y(t) = -10\,(y(t) - 1)^2$$

$$y(t) = \frac{-t - \frac{1}{5}}{-t - \frac{1}{10}}$$

On calcule la solution approchée pour différentes valeurs de N :

```
def multipas(gamma, phi, tt, sol_exacte):
    h  = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    uu[:2] = sol_exacte(tt[:2])
    for n in range(1,len(tt) - 1):
        uu[n+1] = ( (gamma**2/3+gamma/2)*uu[n] + (2*gamma**2/3+gamma/2-1)*uu[n-1]

                  ↪  +h*(-gamma**2/6-gamma/4+2)*phi(tt[n],uu[n])+h*(-gamma**2/6-gamma/4)*phi(tt[n-1],uu[n-1])
                  ↪  )
    return uu

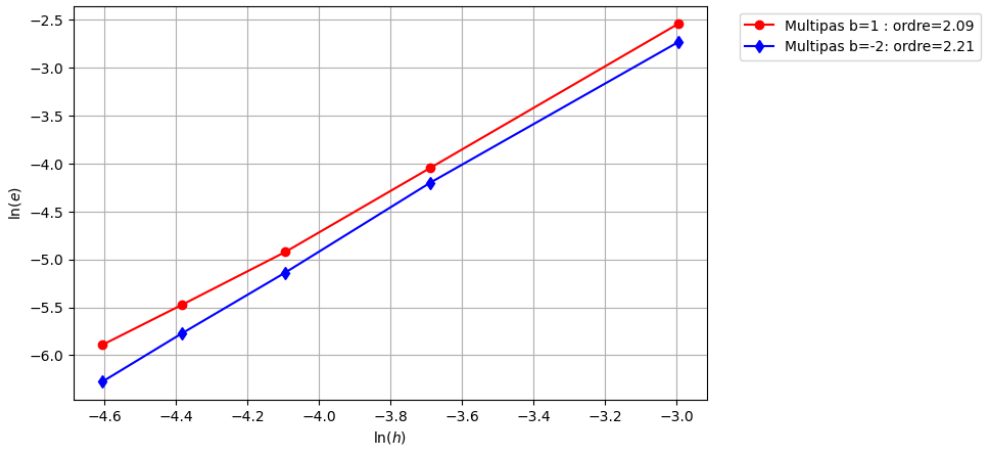

H = []
err_mp1 = []
err_mp2 = []

for N in range(20,120,20):
    tt = np.linspace(t0, tfinal, N + 1)
    h = tt[1] - tt[0]
    yy = sol_exacte(tt)
    uu_mp1 = multipas( 1, phi, tt, sol_exacte)
    uu_mp2 = multipas(-2, phi, tt, sol_exacte)
    H.append(h)
```

```
    err_mp1.append(max([abs(uu_mp1[n] - yy[n]) for n in range(len(yy))]))
    err_mp2.append(max([abs(uu_mp2[n] - yy[n]) for n in range(len(yy))]))

ordre_gamma_1 = np.polyfit(np.log(H),np.log(err_mp1), 1)[0]
ordre_gamma_m2 = np.polyfit(np.log(H),np.log(err_mp2), 1)[0]

plt.figure(figsize=(8,5))
plt.plot(np.log(H), np.log(err_mp1), 'r-o', label=f'Multipas b=1 : ordre={ordre_gamma_1:.2f}')
plt.plot(np.log(H), np.log(err_mp2), 'b-d', label=f'Multipas b=-2: ordre={ordre_gamma_m2:.2f}')
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
plt.legend(bbox_to_anchor=(1.04, 1), loc='upper left')
plt.grid(True)
```



Intervalle de A-stabilité

On étudie la A-stabilité seulement pour les valeurs de γ qui donnent la convergence.

```
r = sp.Symbol('r')
x = sp.Symbol('x', real=True, positive=True)
gamma = sp.Symbol('gamma')

p = 1
aa = [gamma**2/3+gamma/2, gamma**2/3+gamma/2-1]
bb = [-gamma**2/6-gamma/4+2, -gamma**2/6-gamma/4]
bm1 = sp.S(0)

# Construction du polynôme
rho_expr = r**(p+1) - sum(aa[i]*r**(p-i) for i in range(p+1))
sigma_expr = bm1*r**(p+1) + sum(bb[i]*r**(p-i) for i in range(p+1))
poly = sp.Poly(rho_expr + x * sigma_expr, r)

# display(Math(f"p_x(r) = {sp.latex(poly.as_expr())}"))

coeffs = poly.all_coeffs() # Renvoie [A, B, C]
# display(Math(rf"A(x) = {sp.latex(coeffs[0])}"))
# display(Math(rf"B(x) = {sp.latex(coeffs[1])}"))
# display(Math(rf"C(x) = {sp.latex(coeffs[2])}"))

# ============================================================================

for gamma_val in [1, -2]:

    display(Markdown(f"**Cas $\\gamma = {gamma_val}$**"))

    poly_gamma = sp.Poly(poly.subs(gamma, gamma_val), r)
```

```
    display(Math(f"$p_x(r) = {sp.latex(poly_gamma.as_expr())}$"))

    display(Math(rf"A(x) = {sp.latex(coeffs[0].subs(gamma, gamma_val))}"))
    display(Math(rf"B(x) = {sp.latex(coeffs[1].subs(gamma, gamma_val))}"))
    display(Math(rf"C(x) = {sp.latex(coeffs[2].subs(gamma, gamma_val))}"))

    roots = sp.solve(poly_gamma, r)

    display(Markdown(f"Racines : $r_1(x) = {sp.latex(roots[0])}$, $r_2(x) = {sp.latex(roots[1])}$"))

    # On définit une fonction qui calcule le module manuellement pour éviter le sqrt(négatif) dans numpy
    def get_abs_expr(root):
        re_part = sp.re(root)
        im_part = sp.im(root)
        return sp.sqrt(re_part**2 + im_part**2)

    abs_r1_expr = get_abs_expr(roots[0])
    abs_r2_expr = get_abs_expr(roots[1])

    r1_func = sp.lambdify(x, abs_r1_expr, modules=['numpy'])
    r2_func = sp.lambdify(x, abs_r2_expr, modules=['numpy'])


    # Le discriminant s'annule en
    x_bifurcation = sp.solve(poly_gamma.discriminant(), x)[0]
    display(Markdown(f"Point de bifurcation (racines fusionnent) : $x = {sp.latex(x_bifurcation)}$"))

    x_vals = np.linspace(0, 5, 1000)

    r_abs_1 = r1_func(x_vals)
    r_abs_2 = r2_func(x_vals)


    plt.figure(figsize=(10, 6))

    plt.axhspan(0, 1, color='gray', alpha=0.1, label=r'Zone stable $|r| \leq 1$')

    plt.plot(x_vals, r_abs_1, label=r'$|r_1(x)|$', lw=2)
    plt.plot(x_vals, r_abs_2, label=r'$|r_2(x)|$', lw=2, linestyle='--')

    plt.title(rf'Module des racines ($y\' = -\beta y, \gamma={gamma_val}$)')
    plt.xlabel('$x = h\\beta$')
    plt.ylabel('$|r(x)|$')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.ylim(0, 1.2)
    plt.show()
```

Cas $\gamma = 1$

$$p_x(r) = r^2 + r\left(\frac{19x}{12} - \frac{5}{6}\right) - \frac{5x}{12} + \frac{1}{6}$$

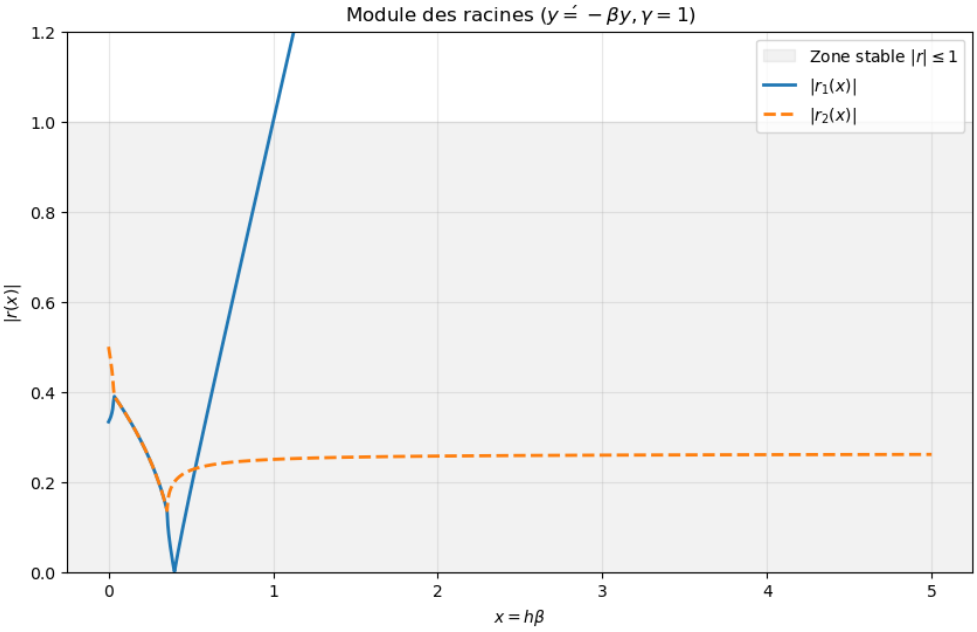
$$A(x) = 1$$

$$B(x) = \frac{19x}{12} - \frac{5}{6}$$

$$C(x) = \frac{1}{6} - \frac{5x}{12}$$

Racines : $r_1(x) = -\frac{19x}{24} - \frac{\sqrt{361x^2-140x+4}}{24} + \frac{5}{12}$, $r_2(x) = -\frac{19x}{24} + \frac{\sqrt{361x^2-140x+4}}{24} + \frac{5}{12}$

Point de bifurcation (racines fusionnent) : $x = \frac{70}{361} - \frac{24\sqrt{6}}{361}$



Cas $\gamma = -2$

$$p_x(r) = r^2 + r\left(\frac{11x}{6} - \frac{1}{3}\right) - \frac{x}{6} + \frac{2}{3}$$

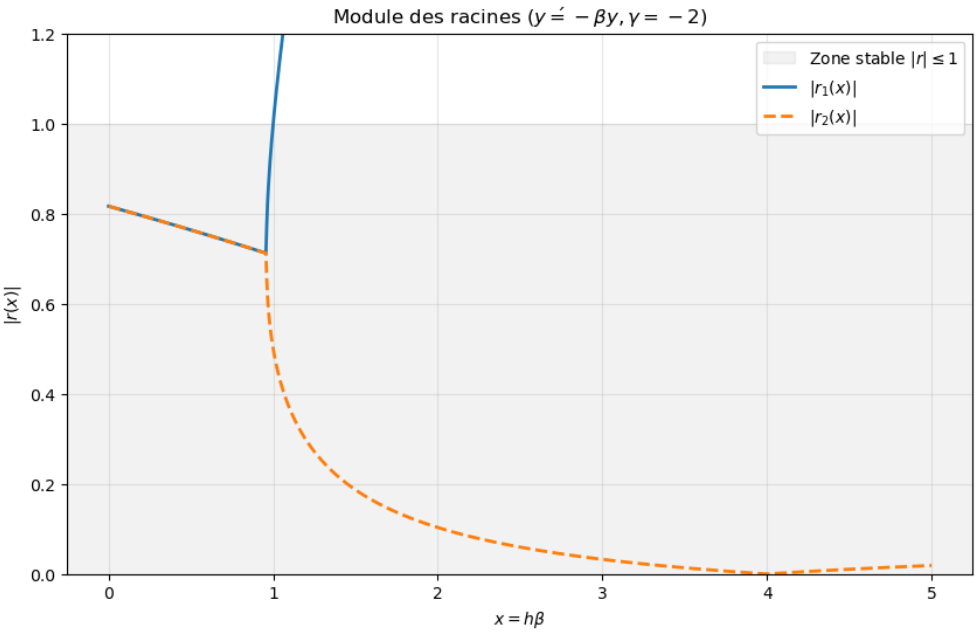
$$A(x) = 1$$

$$B(x) = \frac{11x}{6} - \frac{1}{3}$$

$$C(x) = \frac{2}{3} - \frac{x}{6}$$

Racines : $r_1(x) = -\frac{11x}{12} - \frac{\sqrt{121x^2-20x-92}}{12} + \frac{1}{6}$, $r_2(x) = -\frac{11x}{12} + \frac{\sqrt{121x^2-20x-92}}{12} + \frac{1}{6}$

Point de bifurcation (racines fusionnent) : $x = \frac{10}{121} + \frac{12\sqrt{78}}{121}$



18.1.8. Exercice : convergence d'un schéma à 2 pas explicite

On notera $\varphi_k \overset{\text{déf}}{=} \varphi(t_k, u_k)$. Soit la méthode multipas

$$u_{n+1} = \left(\gamma^2 - \frac{9}{2}\gamma + \frac{11}{2}\right) u_n + \left(\gamma^2 - \frac{7}{2}\gamma + \frac{3}{2}\right) u_{n-1} + h\left(\frac{\gamma}{2}(\gamma-2)\varphi_n + \frac{\gamma}{2}\left(\gamma - \frac{10}{3}\right)\varphi_{n-1}\right).$$

- Pour quelles valeurs de γ est-elle consistante ?
- Est-elle convergente pour ces valeurs ?
- Quel ordre de convergente a-t-elle ?
- Pour les valeurs de γ qui donnent la convergence, vérifier empiriquement l'ordre de convergence sur le problème de Cauchy
$$\begin{cases} y'(t) = y(t)(1-y(t)), \quad \forall t \in I = [0,1], \\ y(0) = 2. \end{cases}$$
- Pour les valeurs de γ qui donnent la convergence, étudier la A-stabilité de ce schéma.

C'est une méthode à $q = 2$ pas explicite :

- $p = 1$
- $b_{-1} = 0$
- $a_0 = \gamma^2 - \frac{9}{2}\gamma + \frac{11}{2}$ et $a_1 = \gamma^2 - \frac{7}{2}\gamma + \frac{3}{2}$
- $b_0 = \frac{\gamma}{2}(\gamma - 2)$ et $b_1 = \frac{\gamma}{2}\left(\gamma - \frac{10}{3}\right)$

La méthode est **consistante** ssi $\xi(0) = \xi(1) = 1$:

$$\begin{cases} \xi(0) = a_0 + a_1 = 1, \\ \xi(1) = -a_1 + b_0 + b_1 + b_{-1} = 1 \end{cases} \iff \begin{cases} 2\gamma^2 - 8\gamma + 7 = 1, \\ \frac{5}{6}\gamma - \frac{9}{6} = 1 \end{cases} \iff \begin{cases} \gamma^2 - 4\gamma + 3 = 0, \\ 5\gamma = 15 \end{cases} \iff \begin{cases} (\gamma-1)(\gamma-3) = \\ \gamma = 3. \end{cases}$$

On conclut que la méthode est consistante ssi $\gamma = 3$.

Vérifions ces calculs ci-dessous :

```
gamma = sp.Symbol(r'\gamma')
p = 1
bm1 = 0
a0 = gamma**2-sp.Rational(9,2)*gamma+sp.Rational(11,2)
a1 = gamma**2-sp.Rational(7,2)*gamma+sp.Rational(3,2)
b0 = sp.Rational(1,2)*gamma*(gamma-2)
b1 = sp.Rational(1,2)*gamma*(gamma-sp.Rational(10,3))


display(Math(r"a_0 = "+sp.latex(a0)))
display(Math(r"a_1 = "+sp.latex(a1)))
display(Math(r"b_{-1} = "+sp.latex(bm1)))
display(Math(r"b_0 = "+sp.latex(b0)))
display(Math(r"b_1 = "+sp.latex(b1)))

cond1 = a0+a1
cond2 = -(0*a0+1*a1)+(bm1+b0+b1)
display(sp.solve([sp.Eq(cond1,1),sp.Eq(cond2,1)],gamma))
```

$a_0 = \gamma^2 - \frac{9\gamma}{2} + \frac{11}{2}$

$a_1 = \gamma^2 - \frac{7\gamma}{2} + \frac{3}{2}$

$b_{-1} = 0$

$b_0 = \frac{\gamma\,(\gamma-2)}{2}$

$$b_1 = \frac{\gamma\left(\gamma - \frac{10}{3}\right)}{2}$$

```
[(3,)]
```

Dans la suite on considerera $\gamma = 3$.

```
display(Math('a_0='+sp.latex(a0.subs(gamma,3))))
display(Math('a_1='+sp.latex(a1.subs(gamma,3))))
display(Math('b_0='+sp.latex(b0.subs(gamma,3))))
display(Math('b_1='+sp.latex(b1.subs(gamma,3))))
```

$a_0 = 1$

$a_1 = 0$

$b_0 = \frac{3}{2}$

$b_1 = -\frac{1}{2}$

Le premier polynôme caractéristique est

$$\varrho(r) = r^2 - a_0 r - a_1 \overset{\gamma=3}{=} r^2 - r$$

dont les racines sont

$$r_0 = 1,\ r_1 = 0$$

La méthode est **zéro-stable** ssi

$$\begin{cases} |r_j| \leq 1 \quad \text{pour tout } j = 0, \dots, p \\ \text{si } r_j \text{ a multiplicité} > 1 \quad \Longrightarrow \quad |r_j| < 1 \end{cases}$$

ce qui est bien vérifié.

La méthode est **convergente** ssi elle est consistante et zéro-stable donc ssi $\gamma = 3$.

La méthode est d'**ordre au moins 2** car $\xi(2) = 1$ pour $\gamma = 3$:

$$\xi(2) = a_1 - 2\left(b_1 - b_{-1}\right) \overset{\gamma=3}{=} 1$$

La méthode **n'est pas d'ordre 3** car $\xi(3) \neq 1$:

$$\xi(3) = -a_1 + 3\left(b_1 + b_{-1}\right) \overset{\gamma=3}{\neq} 1$$

- On définit la solution exacte (calculée en utilisant le module `sympy`) pour estimer les erreurs :

```
t0     = 0
tfinal = 1
y0     = 2

phi = lambda t,y : y*(1-y)
```

```
t  = sp.Symbol('t')
y  = sp.Function('y')
edo = sp.Eq( sp.diff(y(t),t) , phi(t,y(t)) )
display(edo)
solpar = sp.dsolve(edo, ics={y(t0): y0})
display(solpar)

sol_exacte = sp.lambdify(t,solpar.rhs,'numpy')
```

$$\frac{d}{dt}y(t) = (1 - y(t))\, y(t)$$

$$y(t) = \frac{1}{1 - \frac{e^{-t}}{2}}$$

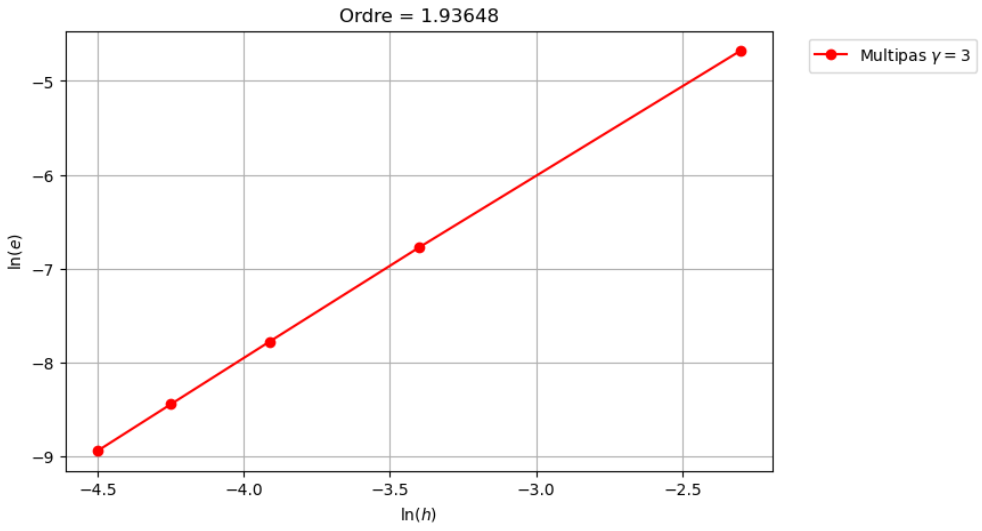
On calcule la solution approchée pour différentes valeurs de N :

```
def multipas(phi, tt, sol_exacte):
    h  = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    uu[:2] = sol_exacte(tt[:2])
    for n in range(1,len(tt) - 1):
        uu[n+1] = uu[n] + h*(3/2*phi(tt[n],uu[n])-1/2*phi(tt[n-1],uu[n-1]))
    return uu


H = []
err_mp = []

for N in range(10,100,20):
    tt = np.linspace(t0, tfinal, N + 1)
    h = tt[1] - tt[0]
    yy = sol_exacte(tt)
    uu_mp = multipas(phi, tt, sol_exacte)
    H.append(h)
    err_mp.append(max([abs(uu_mp[i] - yy[i]) for i in range(len(yy))]))

plt.figure(figsize=(8,5))
plt.plot(np.log(H), np.log(err_mp), 'r-o', label=r'Multipas $\gamma=3$')
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
plt.legend(bbox_to_anchor=(1.04, 1), loc='upper left')
plt.title(f'Ordre = {np.polyfit(np.log(H),np.log(err_mp), 1)[0]:g}')
plt.grid(True)
```



Intervalle de A-stabilité

On étudie la A-stabilité seulement pour les valeurs de γ qui donnent la convergence.

Le schéma étant explicite, il n'est au mieux A-stable que pour certaines valeurs de x. En traçant la valeur absolue des racines en fonction de x, on trouve que la méthode est A-stable pour $x \in]0, x_{\max}[$ avec $x_{\max} = 1$.

```
r = sp.Symbol('r')
x = sp.Symbol('x', real=True, positive=True)
gamma = sp.Symbol('gamma')

p = 1
aa = [gamma**2-9*gamma/2+11/sp.S(2), gamma**2-7*gamma/2+3/sp.S(2)]
bb = [gamma*(gamma-2)/2, gamma*(gamma-10/sp.S(3))/2]
bm1 = sp.S(0)

# Construction du polynôme
rho_expr = r**(p+1) - sum(aa[i]*r**(p-i) for i in range(p+1))
sigma_expr = bm1*r**(p+1) + sum(bb[i]*r**(p-i) for i in range(p+1))
poly = sp.Poly(rho_expr + x * sigma_expr, r)

# display(Math(f"p_x(r) = {sp.latex(poly.as_expr())}"))

coeffs = poly.all_coeffs() # Renvoie [A, B, C]
# display(Math(rf"A(x) = {sp.latex(coeffs[0])}"))
# display(Math(rf"B(x) = {sp.latex(coeffs[1])}"))
# display(Math(rf"C(x) = {sp.latex(coeffs[2])}"))

# ============================================================================

gamma_val = sp.S(3)

display(Markdown(f"**Cas $\\gamma = {gamma_val}$**"))

poly_gamma = sp.Poly(poly.subs(gamma, gamma_val), r)

display(Math(f"$p_x(r) = {sp.latex(poly_gamma.as_expr())}$"))

display(Math(rf"A(x) = {sp.latex(coeffs[0].subs(gamma, gamma_val))}"))
display(Math(rf"B(x) = {sp.latex(coeffs[1].subs(gamma, gamma_val))}"))
display(Math(rf"C(x) = {sp.latex(coeffs[2].subs(gamma, gamma_val))}"))

roots = sp.solve(poly_gamma, r)

display(Markdown(f"Racines : $r_1(x) = {sp.latex(roots[0])}$, $r_2(x) = {sp.latex(roots[1])}$"))

# On définit une fonction qui calcule le module manuellement pour éviter le sqrt(négatif) dans numpy
def get_abs_expr(root):
    re_part = sp.re(root)
    im_part = sp.im(root)
    return sp.sqrt(re_part**2 + im_part**2)

abs_r1_expr = get_abs_expr(roots[0])
abs_r2_expr = get_abs_expr(roots[1])


r1_func = sp.lambdify(x, abs_r1_expr, modules=['numpy'])
r2_func = sp.lambdify(x, abs_r2_expr, modules=['numpy'])


# On calcule les points où |r_1|=1
from scipy.optimize import fsolve
display(Markdown(f"Point où $|r_1|=1$ : $x = {fsolve(lambda x : r1_func(x) - 1, 1)[0]:.4f}$"))
```

```
x_vals = np.linspace(0, 5, 1000)

r_abs_1 = r1_func(x_vals)
r_abs_2 = r2_func(x_vals)

plt.figure(figsize=(10, 6))

plt.axhspan(0, 1, color='gray', alpha=0.1, label=r'Zone stable $|r| \leq 1$')

plt.plot(x_vals, r_abs_1, label=r'$|r_1(x)|$', lw=2)
plt.plot(x_vals, r_abs_2, label=r'$|r_2(x)|$', lw=2, linestyle='--')

plt.title(rf'Module des racines ($y\' = -\beta y, \gamma={gamma_val}$)')
plt.xlabel('$x = h\\beta$')
plt.ylabel('$|r(x)|$')
plt.legend()
plt.grid(True, alpha=0.3)
plt.ylim(0, 1.2)
plt.show()
```

Cas $\gamma = 3$

$$p_x(r) = r^2 + r\left(\frac{3x}{2} - 1\right) - \frac{x}{2}$$

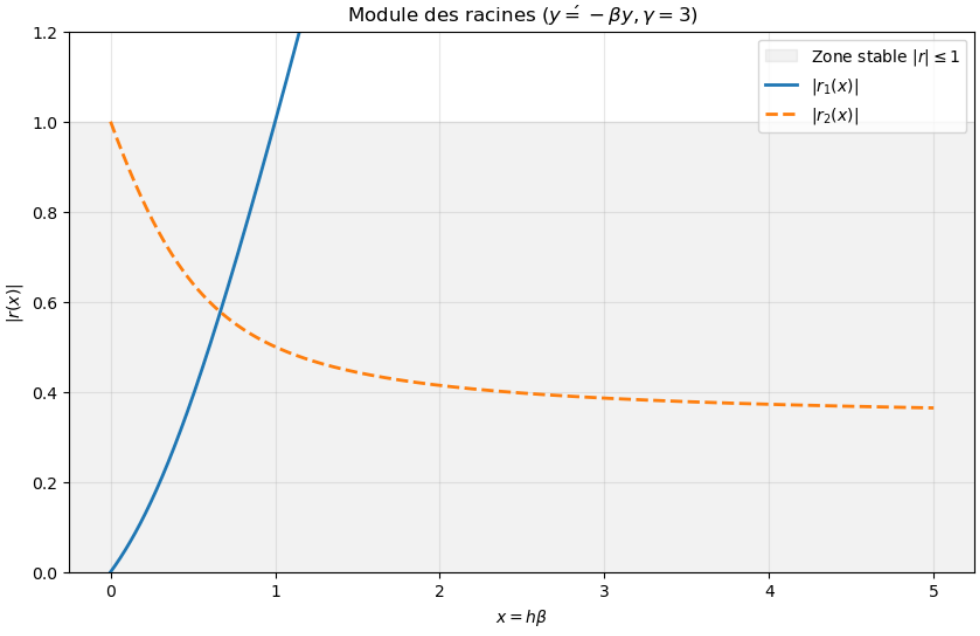
$$A(x) = 1$$

$$B(x) = \frac{3x}{2} - 1$$

$$C(x) = -\frac{x}{2}$$

Racines : $r_1(x) = -\frac{3x}{4} - \frac{\sqrt{9x^2-4x+4}}{4} + \frac{1}{2}$, $r_2(x) = -\frac{3x}{4} + \frac{\sqrt{9x^2-4x+4}}{4} + \frac{1}{2}$

Point où $|r_1| = 1$: $x = 1.0000$



18.2. PARTIE II : Construction de schémas multi-pas linéaires

18.2.1. Exercice : Interpolation, Quadrature et EDO

- Soit f une fonction de classe $\mathcal{C}^1([-1,1])$. Écrire le polynôme $p \in \mathbb{R}_2[\tau]$ qui interpole f aux points -1, 0 et 1.
- Construire une méthode de quadrature comme suit :

$$\int_0^1 f(\tau)\mathrm{d}\tau \approx \int_0^1 p(\tau)\mathrm{d}\tau.$$

  NB : on intègre sur $[0,1]$ mais on interpole en -1, 0 et 1.
- À l'aide d'un changement de variable affine entre l'intervalle $[0,1]$ et l'intervalle $[a,b]$, en déduire une formule de quadrature pour l'intégrale

$$\int_a^b f(x)\mathrm{d}x$$

  lorsque f est une fonction de classe $\mathcal{C}^1([2a-b,b])$.
  Remarque : $[2a-b,b] = [a-(b-a), a+(b-a)]$
- Considérons le problème de Cauchy : trouver $y\colon [t_0,T] \subset \mathbb{R} \to \mathbb{R}$ tel que

$$\begin{cases} y'(t) = \varphi(t, y(t)), & \forall t \in [t_0, T], \\ y(t_0) = y_0, \end{cases}$$

  dont on suppose l'existence d'une unique solution y.
  - On subdivise l'intervalle $[t_0;T]$ en N intervalles $[t_n;t_{n+1}]$ de largeur $h = \dfrac{T-t_0}{N}$ avec $t_n = t_0 + nh$ pour $n = 0, \dots, N$.
  - Utiliser la formule obtenue au point 3 pour approcher l'intégrale

$$\int_{t_n}^{t_{n+1}} \varphi(t, y(t))\mathrm{d}t.$$

  - En déduire un schéma à deux pas implicite pour l'approximation de la solution du problème de Cauchy.

On cherche les coefficients α, β et γ du polynôme $p(\tau) = \alpha + \beta\tau + \gamma\tau^2$ tels que

$$\begin{cases} p(-1) = f(-1), \\ p(0) = f(0), \\ p(1) = f(1), \end{cases} \quad \text{c'est à dire} \quad \begin{cases} \alpha - \beta + \gamma = f(-1), \\ \alpha = f(0), \\ \alpha + \beta + \gamma = f(1). \end{cases}$$

Donc $\alpha = f(0)$, $\beta = \frac{f(1)-f(-1)}{2}$ et $\gamma = \frac{f(1)-2f(0)+f(-1)}{2}$.

```
f_0 = sp.Symbol('f(0)')
f_1 = sp.Symbol('f(1)')
f_m1 = sp.Symbol('f(-1)')
t = sp.Symbol('t')

p = sp.interpolate([(1,f_1),(0,f_0),(-1,f_m1)], t).factor(t)
display(sp.Eq(sp.Symbol('p'), p))
q = (sp.integrate(p,(t,0,1)).simplify())
display(sp.Eq( sp.Integral(sp.Symbol('p(t)'), (t, 0, 1)) , q ))
```

$$p = \frac{2f(0) + t^2\left(f(-1) - 2f(0) + f(1)\right) + t\left(-f(-1) + f(1)\right)}{2}$$

$$\int_0^1 p(t)\,dt = -\frac{f(-1)}{12} + \frac{2f(0)}{3} + \frac{5f(1)}{12}$$

On en déduit la méthode de quadrature

$$\int_0^1 f(\tau)\mathrm{d}\tau \approx \int_0^1 p(\tau)\mathrm{d}\tau = \alpha + \frac{\beta}{2} + \frac{\gamma}{3} = \frac{-f(-1)+8f(0)+5f(1)}{12}.$$

Soit $x = m\tau + q$, alors

$$\int_a^b f(x)\mathrm{d}x = m\int_0^1 f(m\tau+q)\mathrm{d}\tau \quad \text{avec} \quad \begin{cases} a = q, \\ b = m+q, \end{cases} \qquad \text{i.e.} \quad \begin{cases} m = b-a, \\ q = a, \end{cases}$$

d'où le changement de variable $x = (b-a)\tau + a$. On en déduit la formule de quadrature

$$\int_a^b f(x)\mathrm{d}x = (b-a)\int_0^1 f((b-a)\tau+a)\,\mathrm{d}\tau \approx (b-a)\frac{-f(2a-b)+8f(a)+5f(b)}{12}.$$

On pose $a = t_n$ et $b = t_{n+1}$ d'où la formule de quadrature

$$\int_{t_n}^{t_{n+1}} f(t)\mathrm{d}t \approx (t_{n+1}-t_n)\frac{-f(2t_n-t_{n+1})+8f(t_n)+5f(t_{n+1})}{12} = h\frac{-f(t_{n-1})+8f(t_n)+5f(t_{n+1})}{12}.$$

En utilisant la formule de quadrature pour l'intégration de l'EDO $y'(t) = \varphi(t, y(t))$ entre t_n et t_{n+1} on obtient

$$y(t_{n+1}) = y(t_n) + \int_{t_n}^{t_{n+1}} \varphi(t, y(t))\mathrm{d}t \approx h\frac{-\varphi(t_{n-1}, y(t_{n-1}))+8\varphi(t_n, y(t_n))+5\varphi(t_{n+1}, y(t_{n+1}))}{12}.$$

Si on note u_n une approximation de la solution y au temps t_n, on obtient le schéma à deux pas implicite suivant :

$$\begin{cases} u_0 = y(t_0) = y_0, \\ u_1 \text{ à définir}, \\ u_{n+1} = u_n + h\dfrac{-\varphi(t_{n-1}, u_{n-1})+8\varphi(t_n, u_n)+5\varphi(t_{n+1}, u_{n+1})}{12} \end{cases} \quad n = 1, 2, \dots N-1$$

On peut utiliser une pédiction d'Euler explicite pour initialiser u_1 :

$$\begin{cases} u_0 = y(t_0) = y_0, \\ u_1 = u_0 + h\varphi(t_0, u_0), \\ u_{n+1} = u_n + h\dfrac{-\varphi(t_{n-1}, u_{n-1})+8\varphi(t_n, u_n)+5\varphi(t_{n+1}, u_{n+1})}{12} \end{cases} \quad n = 1, 2, \dots N-1$$

18.2.2. Exercice : Interpolation, Quadrature et EDO

1. Soit $h > 0$ et $f\colon [a-h, a+h] \to \mathbb{R}$ une fonction de classe $\mathcal{C}^1([a-h, a+h])$. Écrire le polynôme $p \in \mathbb{R}_1[x]$ qui interpole f aux points $a-h$ et a, i.e. l'équation de la droite $p \in \mathbb{R}_1[x]$ qui passe par les deux points $(a-h, f(a-h))$ et $(a, f(a))$.
2. Construire une méthode de quadrature comme suit :

$$\int_a^{a+h} f(x)\mathrm{d}x \approx \int_a^{a+h} p(x)\mathrm{d}x.$$

   NB : on intègre sur $[a, a+h]$ mais on interpole en $a-h$ et a.
3. Considérons le problème de Cauchy : trouver $y\colon [t_0, T] \subset \mathbb{R} \to \mathbb{R}$ tel que

$$\begin{cases} y'(t) = \varphi(t, y(t)), \quad \forall t \in [t_0, T], \\ y(t_0) = y_0, \end{cases}$$

   dont on suppose l'existence d'une unique solution y. On subdivise l'intervalle $[t_0; T]$ en N intervalles $[t_n; t_{n+1}]$ de largeur $h = \dfrac{T-t_0}{N}$ avec $t_n = t_0 + nh$ pour $n = 0, \dots, N$.

Utiliser la formule obtenue au point 2 pour approcher l'intégrale

$$\int_{t_n}^{t_{n+1}} \varphi(t, y(t))\mathrm{d}t.$$

En déduire un schéma à deux pas explicite pour l'approximation de la solution du problème de Cauchy.

$$p(x) = \frac{f(a) - f(a-h)}{a-(a-h)}(x-a) + f(a) = \frac{f(a) - f(a-h)}{h}(x-a) + f(a).$$

```
a = sp.Symbol('a')
h = sp.Symbol('h')
f_a = sp.Symbol('f(a)')
f_amh = sp.Symbol('f(a-h)')

p = sp.interpolate([(a,f_a),(a-h,f_amh)], t).factor(t)
display(sp.Eq(sp.Symbol('p'), p))
q = (sp.integrate(p,(t,a,a+h)).simplify())
display(sp.Eq( sp.Integral(sp.Symbol('p(t)'), (t, a, a+h)) , q ))
```

$$p = \frac{-af(a) + af(a-h) + f(a)h + t\left(f(a) - f(a-h)\right)}{h}$$

$$\int_a^{a+h} p(t)\,dt = \frac{h\left(3f(a) - f(a-h)\right)}{2}$$

On en déduit la méthode de quadrature

$$\begin{aligned}
\int_a^{a+h} f(x)\mathrm{d}x &\approx \int_a^{a+h} p(x)\mathrm{d}x \\
&= \frac{f(a) - f(a-h)}{h}\left[\frac{(x-a)^2}{2}\right]_a^{a+h} + f(a)\left[x\right]_a^{a+h} \\
&= \frac{f(a) - f(a-h)}{2h}\left((a+h-a)^2 - (a-a)^2\right) + f(a)(a+h-a) \\
&= \frac{f(a) - f(a-h)}{2h}h^2 + hf(a) \\
&= h\frac{3f(a) - f(a-h)}{2}.
\end{aligned}$$

On pose $a = t_n$ et $a + h = t_{n+1}$ d'où la formule de quadrature

$$\int_{t_n}^{t_{n+1}} f(t)\mathrm{d}t \approx (t_{n+1} - t_n)\frac{3f(t_n) - f(2t_n - t_{n+1})}{2} = h\frac{3f(t_n) - f(t_{n-1})}{2}.$$

En utilisant la formule de quadrature pour l'intégration de l'EDO $y'(t) = \varphi(t, y(t))$ entre t_n et t_{n+1} on obtient

$$y(t_{n+1}) = y(t_n) + \int_{t_n}^{t_{n+1}} \varphi(t, y(t))\mathrm{d}t \approx h\frac{3\varphi(t_n, y(t_n)) - \varphi(t_{n-1}, y(t_{n-1}))}{2}.$$

Si on note u_n une approximation de la solution y au temps t_n, on obtient le schéma à deux pas implicite suivant :

$$\begin{cases}
u_0 = y(t_0) = y_0, \\
u_1 \text{ à définir}, \\
u_{n+1} = u_n + h\dfrac{3\varphi(t_n, u_n) - \varphi(t_{n-1}, u_{n-1})}{2} \quad n = 1, 2, \dots N-1
\end{cases}$$

On peut utiliser une prédiction d'Euler explicite pour initialiser u_1 :

$$\begin{cases} u_0 = y(t_0) = y_0, \\ u_1 = u_0 + h\varphi(t_0, u_0), \\ u_{n+1} = u_n + h\dfrac{3\varphi(t_n, u_n) - \varphi(t_{n-1}, u_{n-1})}{2} \quad n = 1, 2, \dots N-1 \end{cases}$$

18.2.3. Exercice : Construction d'un schéma

Considérons le problème de Cauchy

$$\begin{cases} y'(t) = \varphi(t, y(t)), \quad \forall t \in I = [t_0, T], \\ y(t_0) = y_0, \end{cases}$$

avec y_0 une valeur donnée et supposons que l'on ait montré l'existence et l'unicité d'une solution y pour $t \in I$.
On subdivise l'intervalle $I = [t_0; T]$, avec $T < +\infty$, en N intervalles $[t_n; t_{n+1}]$ de largeur $h = (T - t_0)/N$ avec $t_n = t_0 + nh$ pour $n = 0, \dots, N$.
Pour chaque nœud t_n, on note $y_n = y(t_n)$ et on cherche la valeur inconnue u_n qui approche la valeur exacte y_n
Si nous intégrons l'EDO $y'(t) = \varphi(t, y(t))$ entre t_{n-2} et t_{n+1} nous obtenons

$$y_{n+1} - y_{n-2} = \int_{t_{n-2}}^{t_{n+1}} \varphi(t, y(t))dt.$$

Écrire le schéma obtenu en approchant l'intégrale $\int_{t_{n-2}}^{t_{n+1}} \varphi(t, y(t))\mathrm{d}t$ par l'intégrale $\int_{t_{n-2}}^{t_{n+1}} p(t)\mathrm{d}t$ où p est le polynôme interpolant φ en t_n et t_{n-1} (attention à bien initialiser la suite définie par récurrence pour qu'on puisse effectivement calculer tous les termes). Le schéma obtenu est-il explicite ?

Le polynôme interpolant φ en t_n et t_{n-1} a équation

$$p(t) = \frac{\varphi(t_n, y_n) - \varphi(t_{n-1}, y_{n-1})}{t_n - t_{n-1}}(t - t_n) + \varphi(t_n, y_n).$$

On intègre ce polynôme entre t_{n-2} et t_{n+1} :

$$\begin{aligned} \int_{t_{n-2}}^{t_{n+1}} p(t)\mathrm{d}t &= \frac{\varphi(t_n, y_n) - \varphi(t_{n-1}, y_{n-1})}{h}\left[\frac{(t-t_n)^2}{2}\right]_{t_{n-2}}^{t_{n+1}} + \varphi(t_n, y_n)\left[t\right]_{t_{n-2}}^{t_{n+1}} \\ &= \frac{\varphi(t_n, y_n) - \varphi(t_{n-1}, y_{n-1})}{2h}\left[(t_{n+1} - t_n)^2 - (t_{n-2} - t_n)^2\right] + \varphi(t_n, y_n)\left[t_{n+1} - t_{n-2}\right] \\ &= \frac{3}{2}h\left(\varphi(t_n, y_n) + \varphi(t_{n-1}, y_{n-1})\right). \end{aligned}$$

On obtient le schéma explicite

$$\begin{cases} u_0 = y_0, \\ u_1 = u_0 + h\varphi(t_0, u_0), \\ u_{n+1} = u_{n-2} + \frac{3}{2}h\left(\varphi(t_n, u_n) + \varphi(t_{n-1}, u_{n-1})\right). \end{cases}$$

```
h = sp.Symbol('h')
t_n = sp.Symbol('t_{n}')
t_nm1 = sp.Symbol('t_{n-1}')
t_np1 = sp.Symbol('t_{n+1}')
t_nm2 = sp.Symbol('t_{n-2}')
f_n = sp.Symbol('f(t_{n})')
f_nm1 = sp.Symbol('f(t_{n-1})')
```

```
p = sp.interpolate([(t_n,f_n),(t_nm1,f_nm1)], t).factor(t)
display(sp.Eq(sp.Symbol('p'), p))
q = (sp.integrate(p,(t,t_nm2,t_np1)).subs(t_np1,t_n+h).subs(t_nm1,t_n-h).subs(t_nm2,t_n-2*h)).simplify()
display(sp.Eq( sp.Integral(sp.Symbol('p(t)'), (t, t_nm2,t_np1)) , q ))
```

$$p = \frac{-f(t_{n-1})t_n + f(t_n)t_{n-1} + t\left(f(t_{n-1}) - f(t_n)\right)}{t_{n-1} - t_n}$$

$$\int_{t_{n-2}}^{t_{n+1}} p(t)\,dt = \frac{3h\left(f(t_{n-1}) + f(t_n)\right)}{2}$$

18.2.4. Exercice : construction d'un schéma

Considérons le problème de Cauchy :

$$\begin{cases} y'(t) = \varphi(t, y(t)), \quad \forall t \in [t_0, T], \\ y(t_0) = y_0. \end{cases}$$

Supposons que l'on ait montré l'existence d'une unique solution y.
On subdivise l'intervalle $[t_0, T]$ en N intervalles de longueur $h = (T - t_0)/N = t_{n+1} - t_n$. Pour chaque nœud $t_n = t_0 + nh$ $(1 \leq n \leq N)$ on cherche la valeur inconnue u_n qui approche $y(t_n)$. L'ensemble de $N + 1$ valeurs $\{u_0 = y_0, u_1, \ldots, u_N\}$ représente la solution numérique.
Dans cette exercice on va construire des nouveaux schémas numériques basés sur l'intégration de l'EDO $y'(t) = \varphi(t, y(t))$ entre t_n et t_{n+2} :

$$y(t_{n+2}) = y(t_n) + \int_{t_n}^{t_{n+2}} \varphi(t, y(t))\mathrm{d}t.$$

1. En utilisant la formule de quadrature du point milieu pour approcher le membre de droite écrire un schéma numérique explicite permettant de calculer u_{n+2} à partir de u_{n+1} et u_n. Notons que ce schéma a besoin de deux valeurs initiales ; on posera alors $u_0 = y_0$ et u_1 sera approché par une prédiction d'Euler progressive.
2. En utilisant la formule de quadrature de Cavalieri-Simpson pour approcher le membre de droite écrire un schéma numérique implicite permettant de calculer u_{n+2} à partir de u_{n+1} et u_n. Notons que ce schéma a besoin de deux valeurs initiales ; on posera alors $u_0 = y_0$ et u_1 sera approché par une prédiction d'Euler progressive.
3. Proposer une modification du schéma au point précédent pour qu'il devienne explicite.

Rappels : une formule de quadrature interpolatoire approche l'intégrale d'une fonction f par l'intègrale dun polynôme qui interpole f en des points donnés : $\int_a^b f(t)\mathrm{d}t \approx \int_a^b p(t)\mathrm{d}t$ + la formule de quadrature du point milieu considère comme points d'interpolation le point $\frac{a+b}{2}$ ainsi

$$\int_a^b f(t)\mathrm{d}t \approx \int_a^b f\left(\frac{a+b}{2}\right)\mathrm{d}t = (b-a)f\left(\frac{a+b}{2}\right)$$

+ la formule de Simpson considère comme points d'interpolation les points $\left\{a, \frac{a+b}{2}, b\right\}$ ainsi

$$\int_a^b f(t)\mathrm{d}t \approx \frac{b-a}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right]$$

Si on utilise la formule de quadrature du point milieu sur l'intervalle $[t_n; t_{n+2}]$, ie

$$\int_{t_n}^{t_{n+2}} \varphi(t, y(t))\mathrm{d}t \approx 2h\varphi\left(t_{n+1}, y(t_{n+1})\right)$$

on obtient

$$\begin{cases} u_0 = y(t_0) = y_0, \\ u_1 \approx y(t_1) \\ u_{n+2} = u_n + 2h\varphi(t_{n+1}, u_{n+1}) \quad n = 0, 1, \dots N-2 \end{cases}$$

où u_1 est une approximation de $y(t_1)$. Nous pouvons utiliser une prédiction d'Euler progressive pour approcher u_1. Nous avons construit ainsi un nouveau schéma explicite à 2 pas :

$$\begin{cases} u_0 = y(t_0) = y_0, \\ u_1 = u_0 + h\varphi(t_0, u_0), \\ u_{n+2} = u_n + 2h\varphi(t_{n+1}, u_{n+1}) \quad n = 0, 1, \dots N-2 \end{cases}$$

Si on utilise la formule de quadrature de Cavalieri-Simpson sur l'intervalle $[t_n; t_{n+2}]$, ie

$$\int_{t_n}^{t_{n+2}} \varphi(t, y(t))\mathrm{d}t \approx \frac{h}{3}\left(\varphi(t_n, y(t_n)) + 4\varphi(t_{n+1}, y(t_{n+1})) + \varphi(t_{n+2}, y(t_{n+2}))\right),$$

et avec une prédiction d'Euler explicite pour approcher u_1, nous obtenons un nouveau schéma implicite à 3 pas :

$$\begin{cases} u_0 = y(t_0) = y_0, \\ u_1 = u_0 + h\varphi(t_0, u_0), \\ u_{n+2} = u_n + \frac{h}{3}\left(\varphi(t_n, u_n) + 4\varphi(t_{n+1}, u_{n+1}) + \varphi(t_{n+2}, u_{n+2})\right) \quad n = 0, 1, \dots N-2 \end{cases}$$

Pour éviter le calcul implicite de u_{n+2}, nous pouvons utiliser une technique de type predictor-corrector et remplacer le u_{n+2} dans le terme $\varphi(t_{n+2}, u_{n+2})$ par exemple par $\tilde{u}_{n+2} = u_{n+1} + h\varphi(t_{n+1}, u_{n+1})$. Nous avons construit ainsi un nouveau schéma explicite à 3 pas :

$$\begin{cases} u_0 = y(t_0) = y_0, \\ u_1 = u_0 + h\varphi(t_0, u_0), \\ u_{n+2} = u_n + \frac{h}{3}\left(\varphi(t_n, u_n) + 4\varphi(t_{n+1}, u_{n+1}) + \varphi(t_{n+2}, u_{n+1} + h\varphi(t_{n+1}, u_{n+1}))\right) \quad n = 0, 1, \dots N-2 \end{cases}$$

18.2.5. Exercice : Construction du schéma de Boole-Moulton

La **formule d'intégration de Boole** (parfois appelée *Bode's rule* suite à une erreur historique de typographie) est une méthode de Newton-Cotes fermée. Elle approche l'intégrale d'une fonction φ sur l'intervalle $[t_{n-3}, t_{n+1}]$ par l'intégrale du polynôme d'interpolation de degré 4 passant par les points équirépartis :

$$(t_{n+1}, \phi_{n+1}), (t_n, \phi_n), (t_{n-1}, \phi_{n-1}), (t_{n-2}, \phi_{n-2}), (t_{n-3}, \phi_{n-3}).$$

Objectif : en utilisant cette formule de quadrature, construire une **méthode multi-pas linéaire** pour résoudre l'équation différentielle $y'(t) = \varphi(t, y(t))$. Cette méthode est connue sous le nom de **méthode de Boole-Moulton** (1960).
Suggestion : utilisez la bibliothèque `sympy` pour calculer le polynôme d'interpolation et intégrer ce dernier afin d'extraire les coefficients exacts du schéma.
Documentation MathWorld : Boole's Rule

Le principe des méthodes multi-pas linéaires consiste à intégrer l'équation différentielle $y'(t) = \varphi(t, y(t))$ sur un intervalle de largeur $4h$ et à approcher l'intégrale par une formule de quadrature interpolatoire. En intégrant l'EDO entre t_{n-3} et t_{n+1} on obtient :

$$y(t_{n+1}) = y(t_{n-3}) + \int_{t_{n-3}}^{t_{n+1}} \varphi(t, y(t))\mathrm{d}t$$

- *Étape 1 : Interpolation polynomiale.* On remplace la fonction inconnue $\varphi(t, y(t))$ sous l'intégrale par son polynôme d'interpolation de Lagrange $P(t)$ s'appuyant sur les points $\{t_{n+1}, t_n, t_{n-1}, t_{n-2}, t_{n-3}\}$. Astuce calculatoire : pour simplifier le calcul des coefficients du polynôme d'interpolation, on effectue le changement de variable affine $t = t_n + kh$:

$$\int_{t_{n-3}}^{t_{n+1}} \varphi(t, y(t))\mathrm{d}t \approx \int_{t_{n-3}}^{t_{n+1}} P(t)\mathrm{d}t = h\int_{-3}^{1} P(k)\mathrm{d}k$$

- *Étape 3 : Quadrature de Boole.* L'intégration du polynôme conduit à la formule de quadrature de Boole :

$$\int_{t_{n-3}}^{t_{n+1}} P(t)\mathrm{d}t = \frac{2h}{45}\left(7\varphi_{n+1} + 32\varphi_n + 12\varphi_{n-1} + 32\varphi_{n-2} + 7\varphi_{n-3}\right)$$

- *Étape 4 : Schéma numérique final.* En remplaçant l'intégrale dans l'équation initiale, on obtient le schéma de Boole-Moulton :

$$u_{n+1} = u_{n-3} + \frac{2h}{45}\left(7\varphi(t_{n+1}, u_{n+1}) + 32\varphi(t_n, u_n) + 12\varphi(t_{n-1}, u_{n-1}) + 32\varphi(t_{n-2}, u_{n-2}) + 7\varphi(t_{n-3}, u_{n-3})\right).$$

```
# 1. Variables symboliques
t, h, t_n = sp.symbols('t h t_n')
phi_np1, phi_n, phi_nm1, phi_nm2, phi_nm3 = sp.symbols('phi_{n+1} phi_n phi_{n-1} phi_{n-2} phi_{n-3}')
u_np1, u_nm3 = sp.symbols('u_{n+1} u_{n-3}')

# 2. Points d'interpolation (t_i, phi_i)
points = [
    (t_n + h,   phi_np1),
    (t_n,       phi_n),
    (t_n - h,   phi_nm1),
    (t_n - 2*h, phi_nm2),
    (t_n - 3*h, phi_nm3)
]

# 3. Calcul du polynôme d'interpolation P(t)
P = sp.interpolate(points, t)

# 4. Calcul de l'intégrale de t_{n-3} à t_{n+1} de P(t)
integrale = sp.integrate(P, (t, t_n - 3*h, t_n + h))
integrale = sp.simplify(integrale)

# 5. Construction et affichage du schéma final
# u_{n+1} = u_{n-3} + Intégrale
schema = sp.Eq(u_np1, u_nm3 + integrale)

print("Schéma de Boole-Moulton obtenu :")
display(schema)
```

```
Schéma de Boole-Moulton obtenu :
```

$$u_{n+1} = \frac{2h\left(32\phi_n + 7phi_{n+1} + 12phi_{n-1} + 32phi_{n-2} + 7phi_{n-3}\right)}{45} + u_{n-3}$$

BONUS : on généralise cette approche pour construire des méthodes multi-pas linéaires en utilisant des polynômes d'interpolation quelconques et en intégrant ces polynômes sur des intervalles quelconques.

```
def calculer_multipas(indices_points, a_shift, b_shift):

    t, h, t_n = sp.symbols('t h t_n')

    # 1. Création des symboles
    t_syms = [sp.Symbol(f't_{{n{f"+{k}" if k>0 else "" if k==0 else k}}}') for k in indices_points]
    phi_syms = [sp.Symbol(fr'\phi_{{n{f"+{k}" if k>0 else "" if k==0 else k}}}') for k in indices_points]

    # 2. Interpolation
    subs_t = {ts: t_n + k*h for ts, k in zip(t_syms, indices_points)}
    pts_calcul = [(ts.subs(subs_t), phis) for ts, phis in zip(t_syms, phi_syms)]
    polynome = sp.interpolate(pts_calcul, t)

    # 3. Quadrature
    borne_a, borne_b = t_n + a_shift * h, t_n + b_shift * h
```

```
    integrale = sp.integrate(polynome, (t, borne_a, borne_b))
    quadrature = sp.simplify(integrale)

    # 4. Coefficients du schéma
    expr_over_h = sp.simplify(quadrature / h)
    coeffs = [expr_over_h.coeff(p) for p in phi_syms]
    denoms = [sp.fraction(c)[1] for c in coeffs if c != 0]
    lcm_denom = sp.lcm(denoms) if denoms else 1

    termes_somme = [sp.simplify(c * lcm_denom) * p for c, p in zip(coeffs, phi_syms)]
    somme_interne = sp.Add(*termes_somme, evaluate=False)

    # 5. Retour des résultats structurés
    return {
        "t": t, "h": h, "t_n": t_n,
        "t_syms": t_syms, "phi_syms": phi_syms,
        "polynome": polynome.simplify(),
        "quadrature": quadrature,
        "lcm_denom": lcm_denom,
        "somme_interne": somme_interne,
        "indices": {"points": indices_points, "a": a_shift, "b": b_shift}
    }


def afficher_etude_schema(nom, data):

    format_idx = lambda k: f"+{k}" if k > 0 else f"{k}" if k < 0 else ""

    # Extraction des variables utiles
    idx = data["indices"]
    u_start = sp.Symbol(f'u_{{n{format_idx(idx["a"])}}}')
    u_target = sp.Symbol(f'u_{{n{format_idx(idx["b"])}}}')

    # Construction de l'équation du schéma
    schema = sp.Eq(u_target, u_start + sp.Mul(data["h"]/data["lcm_denom"], data["somme_interne"],
    ↪ evaluate=False), evaluate=False)

    # Points d'interpolation en LaTeX
    latex_points = ", ".join([rf"({sp.latex(ts)}, {sp.latex(phis)})" for ts, phis in zip(data["t_syms"],
    ↪ data["phi_syms"])])

    # Polynôme d'interpolation en forme canonique
    poly_canonique = sp.expand(data["polynome"])

    # --- Rendu ---
    display(Markdown(f"Étude du schéma : {nom}"))

    display(Markdown(r"- Points d'interpolation $(t_i, \phi_i)$"))
    display(Markdown(f"$${latex_points}$$"))

    display(Markdown("- Polynôme d'interpolation $P(t)$ (factorisé)"))
    display(poly_canonique)

    display(Markdown(f"- Formule de quadrature (Intégration de $t_{{n{format_idx(idx['a'])}}}$ à
    ↪ $t_{{n{format_idx(idx['b'])}}}$)"))
    symb_int = sp.Integral(sp.Symbol('P(t)'), (data["t"], data["t_n"] + idx['a']*data["h"], data["t_n"] +
    ↪ idx['b']*data["h"]))
    display(sp.Eq(symb_int, data["quadrature"]))

    display(Markdown("- Schéma EDO final"))
    display(schema)
```

```
res_boole = calculer_multipas(indices_points=[1, 0, -1, -2, -3], a_shift=-3, b_shift=1)
afficher_etude_schema("Boole-Moulton", res_boole)
```

Étude du schéma : Boole-Moulton

- Points d'interpolation (t_i, ϕ_i)

$$(t_{n+1}, \phi_{n+1}), (t_n, \phi_n), (t_{n-1}, \phi_{n-1}), (t_{n-2}, \phi_{n-2}), (t_{n-3}, \phi_{n-3})$$

- Polynôme d'interpolation $P(t)$ (factorisé)

$$\frac{\phi_{n+1}t}{4h} - \frac{\phi_{n+1}t_n}{4h} + \frac{11\phi_{n+1}t^2}{24h^2} - \frac{11\phi_{n+1}tt_n}{12h^2} + \frac{11\phi_{n+1}t_n^2}{24h^2} + \frac{\phi_{n+1}t^3}{4h^3} - \frac{3\phi_{n+1}t^2t_n}{4h^3} + \frac{3\phi_{n+1}tt_n^2}{4h^3} - \frac{\phi_{n+1}t_n^3}{4h^3} + \frac{\phi_{n+1}t^4}{24h^4} - \frac{\phi_{n+1}t^3t_n}{6h^4} + \frac{\phi_{n+1}t^2t_n^2}{4h^4} - \frac{\phi_{n+1}tt_n^3}{6h^4} + \frac{\phi_{n+1}t_n^4}{24h^4} - \frac{3\phi_{n-1}t}{2h} + \frac{3\phi_{n-1}t_n}{2h} + \frac{\phi_{n-1}t^2}{4h^2} - \frac{\phi_{n-1}tt_n}{2h^2} + \frac{\phi_{n-1}t_n^2}{4h^2} + \frac{\phi_{n-1}t^3}{h^3} - \frac{3\phi_{n-1}t^2t_n}{h^3} + \frac{3\phi_{n-1}tt_n^2}{h^3} - \frac{\phi_{n-1}t_n^3}{h^3} + \frac{\phi_{n-1}t^4}{4h^4} - \frac{\phi_{n-1}t^3t_n}{h^4} + \frac{3\phi_{n-1}t^2t_n^2}{2h^4} - \frac{\phi_{n-1}tt_n^3}{h^4} + \frac{\phi_{n-1}t_n^4}{4h^4} + \frac{\phi_{n-2}t}{2h} - \frac{\phi_{n-2}t_n}{2h} + \frac{\phi_{n-2}t^2}{6h^2} - \frac{\phi_{n-2}tt_n}{3h^2} + \frac{\phi_{n-2}t_n^2}{6h^2} - \frac{\phi_{n-2}t^3}{2h^3} + \frac{3\phi_{n-2}t^2t_n}{2h^3} - \frac{3\phi_{n-2}tt_n^2}{2h^3} + \frac{\phi_{n-2}t_n^3}{2h^3} - \frac{\phi_{n-2}t^4}{6h^4} + \frac{2\phi_{n-2}t^3t_n}{3h^4} - \frac{\phi_{n-2}t^2t_n^2}{h^4} + \frac{2\phi_{n-2}tt_n^3}{3h^4} - \frac{\phi_{n-2}t_n^4}{6h^4} - \frac{\phi_{n-3}t}{12h} + \frac{\phi_{n-3}t_n}{12h} - \frac{\phi_{n-3}t^2}{24h^2} + \frac{\phi_{n-3}tt_n}{12h^2} - \frac{\phi_{n-3}t_n^2}{24h^2} + \frac{\phi_{n-3}t^3}{12h^3} - \frac{\phi_{n-3}t^2t_n}{4h^3} + \frac{\phi_{n-3}tt_n^2}{4h^3} - \frac{\phi_{n-3}t_n^3}{12h^3} + \frac{\phi_{n-3}t^4}{24h^4} - \frac{\phi_{n-3}t^3t_n}{6h^4} + \frac{\phi_{n-3}t^2t_n^2}{4h^4} - \frac{\phi_{n-3}tt_n^3}{6h^4} + \frac{\phi_{n-3}t_n^4}{24h^4} + \phi_n + \frac{5\phi_nt}{6h} - \frac{5\phi_nt_n}{6h} - \frac{5\phi_nt^2}{6h^2} + \frac{5\phi_ntt_n}{3h^2} - \frac{5\phi_nt_n^2}{6h^2} - \frac{5\phi_nt^3}{6h^3} + \frac{5\phi_nt^2t_n}{2h^3} - \frac{5\phi_ntt_n^2}{2h^3} + \frac{5\phi_nt_n^3}{6h^3} - \frac{\phi_nt^4}{6h^4} + \frac{2\phi_nt^3t_n}{3h^4} - \frac{\phi_nt^2t_n^2}{h^4} + \frac{2\phi_ntt_n^3}{3h^4} - \frac{\phi_nt_n^4}{6h^4}$$

- Formule de quadrature (Intégration de t_{n-3} à t_{n+1})

$$\int_{-3h+t_n}^{h+t_n} P(t)\,dt = \frac{2h\left(7\phi_{n+1} + 12\phi_{n-1} + 32\phi_{n-2} + 7\phi_{n-3} + 32\phi_n\right)}{45}$$

- Schéma EDO final

$$u_{n+1} = \frac{h}{45} \cdot \left(14\phi_{n+1} + 24\phi_{n-1} + 64\phi_{n-2} + 14\phi_{n-3} + 64\phi_n\right) + u_{n-3}$$

18.3. PARTIE III : Schémas Prédicteur-Correcteur basés sur deux schémas multi-pas linéaires

18.3.1. Aide-mémoire

Principe de la méthode predictor-corrector (PC)
On choisi deux méthodes multi-pas linéaires :

- une méthode explicite à $\tilde{q} = \tilde{p} + 1$ pas (de coefficients $\{\tilde{a}_j\}$ et $\{\tilde{b}_j\}$) :

$$u_{n+1} = \sum_{j=0}^{\tilde{p}} \left(\tilde{a}_j u_{n-j} + h\tilde{b}_j \varphi_{n-j}\right) \quad \text{pour } n = p, p+1, \dots, N-1$$

- une méthode implicite à $q = p + 1$ pas (de coefficients $\{a_j\}$ et $\{b_j\}$) :

$$u_{n+1} = \sum_{j=0}^{p} \left(a_j u_{n-j} + hb_j \varphi_{n-j}\right) + hb_{-1}\varphi_{n+1} \quad \text{pour } n = p, p+1, \dots, N-1.$$

On construit un schéma prédicteur-correcteur (explicite) de la manière suivante :

$$\begin{cases} \tilde{u}_{n+1} = \sum_{j=0}^{\tilde{p}} \left(\tilde{a}_j u_{n-j} + h\tilde{b}_j \varphi_{n-j} \right) \\ u_{n+1} = \sum_{j=0}^{p} \left(a_j u_{n-j} + h b_j \varphi_{n-j} \right) + h b_{-1} \varphi(t_{n+1}, \tilde{u}_{n+1}) \end{cases}$$

Ordre : si on note $\omega_{[P]}$ l'ordre du predictor (schéma explicite) et $\omega_{[C]}$ l'ordre du corrector (schéma implicite), alors l'ordre du schéma *predictor-corrector* est donné par

$$\omega_{[PC]} = \min \left\{ \omega_{[C]}, \omega_{[P]} + 1 \right\}$$

18.3.2. Exercice : convergence d'un schéma predictor-corrector basé sur deux schémas

On notera $\varphi_k \overset{\text{déf}}{=} \varphi(t_k, u_k)$. Soit les méthodes multipas

$$u_{n+1} = u_n + \frac{h}{12}\left(23\varphi_n - 16\varphi_{n-1} + 5\varphi_{n-2}\right) \qquad [\text{P}]$$
$$u_{n+1} = u_n + \frac{h}{24}\left(9\varphi_{n+1} + 19\varphi_n - 5\varphi_{n-1} + \varphi_{n-2}\right) \qquad [\text{C}]$$

- Étudier consistance, ordre et zéro-stabilité de la méthode P
- Étudier consistance, ordre et zéro-stabilité de la méthode C
- Soit la méthode PC suivante :

$$\begin{cases} \tilde{u} = u_n + \frac{h}{12}\left(23\varphi_n - 16\varphi_{n-1} + 5\varphi_{n-2}\right) \\ u_{n+1} = u_n + \frac{h}{24}\left(9\varphi(t_{n+1}, \tilde{u}) + 19\varphi_n - 5\varphi_{n-1} + \varphi_{n-2}\right) \end{cases}$$

Calculer empiriquement l'ordre de convergence de la méthode PC sur le problème de Cauchy

$$\begin{cases} y'(t) = \frac{1}{1+t^2} - 2y^2(t), \quad \forall t \in I = [0,1], \\ y(0) = 0, \end{cases}$$

dont la solution exacte est $y(t) = \frac{t}{1+t^2}$.

Correction schéma P

P est une méthode à $q = 3$ pas explicite : - $p = 2$ - $b_{-1} = 0$ - $a_0 = 1$ et $a_1 = a_2 = 0$ - $b_0 = \frac{23}{12}$, $b_1 = \frac{-16}{12}$ et $b_2 = \frac{5}{12}$ - La méthode est consistante d'ordre $\omega = 3$ car

$$\begin{cases} \sum_{j=0}^{p} a_j = 1, \\ \sum_{j=0}^{p} -j a_j + \sum_{j=-1}^{p} b_j = -a_1 - 2a_2 + (b_{-1} + b_0 + b_1 + b_2) = 1 \\ \sum_{j=0}^{p} (-j)^2 a_j + 2 \sum_{j=-1}^{p} (-j)^1 b_j = a_1 + 4a_2 - 2(-b_{-1} + b_1 + 2b_2) = 1 \\ \sum_{j=0}^{p} (-j)^3 a_j + 3 \sum_{j=-1}^{p} (-j)^2 b_j = -a_1 - 8a_2 + 3(b_{-1} + b_1 + 4b_2) = 1 \\ \sum_{j=0}^{p} (-j)^4 a_j + 4 \sum_{j=-1}^{p} (-j)^3 b_j = a_1 + 16a_2 - 4(-b_{-1} + b_1 + 8b_2) \neq 1 \end{cases}$$

Vérifions ces calculs ci-dessous :

```
p = 1
bm1 = 0
a0 = 1
a1 = 0
a2 = 0
```

```
b0 = sp.Rational(23,12)
b1 = sp.Rational(-16,12)
b2 = sp.Rational(5,12)

aa = [a0, a1, a2]
bb = [b0, b1, b2]

cond = []
cond.append(sum(aa))
cond.append(sum([-j*aa[j]+bb[j] for j in range(len(aa))])+bm1)
if cond==[1,1]:
    display(Math(r"\omega\ge1"))
    for n in range(2,9):
        cond.append(sum( [ (-j)**n*aa[j]+n*(-j)**(n-1)*bb[j] for j in range(len(aa)) ])+n*bm1)
        if cond[n]==1:
            display(Math(r"\omega\ge"+str(n)))
        else:
            display(Math(r"\omega\le"+str(n)))
            break
```

$\omega \geq 1$

$\omega \geq 2$

$\omega \geq 3$

$\omega \leq 4$

- Le premier polynôme caractéristique est

$$\varrho(r) = r^{p+1} - \sum_{j=0}^{p} a_j r^{p-j} = r^3 - a_0 r^2 - a_1 r - a_2 = r^3 - r^2 = r^2(r-1)$$

dont les racines sont

$$r_0 = 1,\ r_1 = r_2 = 0$$

La méthode est zéro-stable ssi

$$\begin{cases} |r_j| \leq 1 \quad \text{pour tout } j = 0, \dots, p \\ \varrho'(r_j) \neq 0 \text{ si } |r_j| = 1, \end{cases}$$

ce qui est bien vérifié.

La méthode est convergente car consistante et zéro-stable.

Correction schéma C

C est une méthode à $q = 3$ pas implicite :

- $p = 2$
- $b_{-1} = \frac{9}{24}$
- $a_0 = 1$ et $a_1 = a_2 = 0$
- $b_0 = \frac{19}{24}$, $b_1 = \frac{-5}{24}$ et $b_2 = \frac{1}{24}$

- La méthode est consistante d'ordre $\omega = 4$ car

$$\begin{cases} \sum_{j=0}^{p} a_j = 1, \\ \sum_{j=0}^{p} -ja_j + \sum_{j=-1}^{p} b_j = -a_1 - 2a_2 + (b_{-1} + b_0 + b_1 + b_2) = 1 \\ \sum_{j=0}^{p} (-j)^2 a_j + 2\sum_{j=-1}^{p} (-j)^1 b_j = a_1 + 4a_2 - 2(-b_{-1} + b_1 + 2b_2) = 1 \\ \sum_{j=0}^{p} (-j)^3 a_j + 3\sum_{j=-1}^{p} (-j)^2 b_j = -a_1 - 8a_2 + 3(b_{-1} + b_1 + 4b_2) = 1 \\ \sum_{j=0}^{p} (-j)^4 a_j + 4\sum_{j=-1}^{p} (-j)^3 b_j = a_1 + 16a_2 - 4(-b_{-1} + b_1 + 8b_2) = 1 \\ \sum_{j=0} (-j)^5 a_j + 5\sum_{j=-1} (-j)^4 b_j = -a_1 + 32a_2 + 5(b_{-1} + b_1 + 16b_2) \neq 1 \end{cases}$$

  Vérifions ces calculs ci-dessous :

```
p = 1
bm1 = sp.Rational(9,24)
a0 = 1
a1 = 0
a2 = 0
b0 = sp.Rational(19,24)
b1 = sp.Rational(-5,24)
b2 = sp.Rational(1,24)

aa = [a0,a1,a2]
bb = [b0,b1,b2]

cond = []
cond.append(sum(aa))
cond.append(sum([-j*aa[j]+bb[j] for j in range(len(aa))])+bm1)
if cond==[1,1]:
    display(Math(r"\omega\ge1"))
    for n in range(2,9):
        cond.append(sum( [ (-j)**n*aa[j]+n*(-j)**(n-1)*bb[j] for j in range(len(aa)) ])+n*bm1)
        if cond[n]==1:
            display(Math(r"\omega\ge"+str(n)))
        else:
            display(Math(r"\omega\le"+str(n)))
            break
```

$\omega \geq 1$

$\omega \geq 2$

$\omega \geq 3$

$\omega \geq 4$

$\omega \leq 5$

- Le premier polynôme caractéristique est

$$\varrho(r) = r^{p+1} - \sum_{j=0}^{p} a_j r^{p-j} = r^3 - a_0 r^2 - a_1 r - a_2 = r^3 - r^2 = r^2(r-1)$$

  dont les racines sont

$$r_0 = 1,\ r_1 = r_2 = 0$$

La méthode est zéro-stable ssi

$$\begin{cases} |r_j| \leq 1 \quad \text{pour tout } j = 0, \dots, p \\ \varrho'(r_j) \neq 0 \text{ si } |r_j| = 1, \end{cases}$$

ce qui est bien vérifié.

La méthode est convergente car consistante et zéro-stable.

Correction schéma PC

Il est d'ordre 4 car le prédictor est d'ordre 3 et le corrector d'ordre 4.

On définit la solution exacte pour estimer les erreurs :

```
# variables globales
t0     = 0
tfinal = 1
y0     = 0

phi = lambda t,y : 1/(1+t**2)-2*y**2

sol_exacte = lambda t : t/(1+t**2)
```

On calcule la solution approchée pour différentes valeurs de N pour les trois schémas :

```
def multipas_P(phi, tt, sol_exacte):
    h  = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    uu[:3] = sol_exacte(tt[:3])
    for n in range(2,len(tt) - 1):
        k_0 = phi(tt[n],uu[n])
        k_1 = phi(tt[n-1],uu[n-1])
        k_2 = phi(tt[n-2],uu[n-2])
        uu[n+1] = uu[n]+h/12* ( 23*k_0-16*k_1+5*k_2 )
    return uu

def multipas_C(phi, tt, sol_exacte):
    h  = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    uu[:3] = sol_exacte(tt[:3])
    for n in range(2,len(tt) - 1):
        k_0 = phi(tt[n],uu[n])
        k_1 = phi(tt[n-1],uu[n-1])
        k_2 = phi(tt[n-2],uu[n-2])
        temp = fsolve ( lambda x : -x+uu[n]+h/24* ( 9*phi(tt[n+1],x) + 19*k_0-5*k_1+k_2 ), uu[n])
        uu[n+1] = temp[0]
    return uu

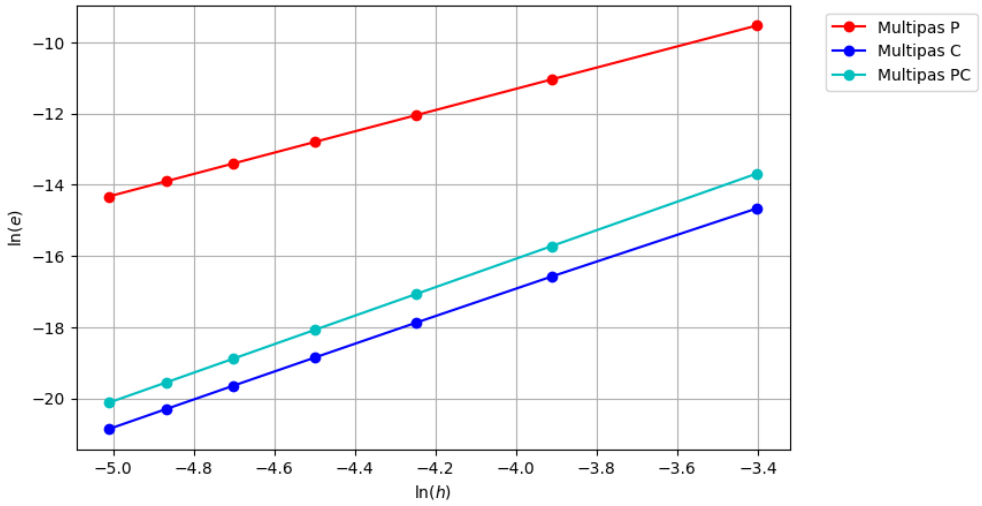
def multipas_PC(phi, tt, sol_exacte):
    h  = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    uu[:3] = sol_exacte(tt[:3])
    for n in range(2,len(tt) - 1):
        k_0 = phi(tt[n],uu[n])
        k_1 = phi(tt[n-1],uu[n-1])
        k_2 = phi(tt[n-2],uu[n-2])
        u_tilde = uu[n]+h/12 * (23*k_0-16*k_1+5*k_2)
        uu[n+1] = uu[n]+h/24* ( 9*phi(tt[n+1],u_tilde) + 19*k_0-5*k_1+k_2 )
    return uu
```

```
H = []
err_p = []
err_c = []
err_pc = []
N = 10
for k in range(7):
    N += 20
    tt = np.linspace(t0, tfinal, N + 1)
    h = tt[1] - tt[0]
    yy = sol_exacte(tt)
    uu_p = multipas_P(phi, tt, sol_exacte)
    uu_c = multipas_C(phi, tt, sol_exacte)
    uu_pc = multipas_PC(phi, tt, sol_exacte)
    H.append(h)
    err_p.append(np.linalg.norm(uu_p-yy, np.inf))
    err_c.append(np.linalg.norm(uu_c-yy, np.inf))
    err_pc.append(np.linalg.norm(uu_pc-yy, np.inf))

print (f'Multipas P ordre  = {np.polyfit(np.log(H),np.log(err_p) , 1)[0]:1.2f}' )
print (f'Multipas C ordre  = {np.polyfit(np.log(H),np.log(err_c) , 1)[0]:1.2f}' )
print (f'Multipas PC ordre = {np.polyfit(np.log(H),np.log(err_pc), 1)[0]:1.2f}')

plt.figure(figsize=(8,5))
plt.plot(np.log(H), np.log(err_p), 'r-o', label='Multipas P')
plt.plot(np.log(H), np.log(err_c), 'b-o', label='Multipas C')
plt.plot(np.log(H), np.log(err_pc), 'c-o', label='Multipas PC')
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
plt.legend(bbox_to_anchor=(1.04, 1), loc='upper left')
plt.grid(True);
```

```
Multipas P ordre  = 2.99
Multipas C ordre  = 3.86
Multipas PC ordre = 4.00
```



18.3.3. Exercice : convergence d'un schéma predictor-corrector basé sur deux schémas

On notera $\varphi_k \overset{\text{déf}}{=} \varphi(t_k, u_k)$. Soit les méthodes multipas

$$u_{n+1} = \frac{4}{3}u_n - \frac{1}{3}u_{n-1} + h\left(b_0\varphi_n + b_1\varphi_{n-1}\right) \quad [\text{P}]$$

$$u_{n+1} = \frac{4}{3}u_n - \frac{1}{3}u_{n-1} + h\frac{2}{3}\varphi_{n+1} \quad [\text{C}]$$

- Étudier consistance, ordre et zéro-stabilité des deux méthodes.
- Écrire et étudier consistance, ordre et zéro-stabilité de la méthode prédictor-corrector associée.
- Calculer empiriquement l'ordre de convergence des trois méthodes sur le problème de Cauchy

$$\begin{cases} y'(t) = \frac{1}{1+t^2} - 2y^2(t), \quad \forall t \in I = [0,1], \\ y(0) = 0, \end{cases}$$

dont la solution exacte est $y(t) = \frac{t}{1+t^2}$.

Les deux schémas multipas linéaires à $q = 2$ pas sont de la forme

$$u_{n+1} = a_0 u_n + a_1 u_{n-1} + h\left(b_0 \varphi_n + b_1 \varphi_{n-1}\right) + h b_{-1} \varphi_{n+1}$$

avec

	a_0	a_1	b_0	b_1	b_{-1}
[P]	$\frac{4}{3}$	$-\frac{1}{3}$	b_0	b_1	0
[C]	$\frac{4}{3}$	$-\frac{1}{3}$	0	0	$\frac{2}{3}$

Les deux schémas ont le même premier polynôme caractéristique :

$$\varrho(r) = r^2 - a_0 r - a_1 = (r-1)\left(r - \frac{1}{3}\right).$$

Ses racines sont $r_0 = 1$ et $r_1 = \frac{1}{3}$. Comme elles appartiennent au disque unité et sont simples, les deux méthodes sont zéro-stables.

D'après la première barrière de Dalquist, un schéma linéaire à $q = 2$ pas consistant et zéro-stable est au mieux

- d'ordre $\omega = q + 2 = 4$ s'il est implicite,
- d'ordre $\omega = q = 2$ s'il est explicite.

Étudions l'ordre de convergence en calculant $\xi(i)$ pour $i = 0, \ldots, 4$ (les deux méthodes ayant le même nombre de pas, on a la même expression pour $\xi(i)$) :

$$\begin{cases} \xi(0) = a_0 + a_1 \\ \xi(1) = -a_1 + b_0 + b_1 + b_{-1} \\ \xi(2) = a_1 - 2\left(b_1 - b_{-1}\right) \\ \xi(3) = -a_1 + 3\left(b_1 + b_{-1}\right) \\ \xi(4) = a_1 - 4\left(b_1 - b_{-1}\right) \end{cases}$$

Pour le schéma [C], on a $\xi(0) = \xi(1) = \xi(2) = 1$ et $\xi(3) \neq 1$, donc la méthode est d'ordre 2 :

$$\begin{cases} \xi(0) = a_0 + a_1 = \frac{4}{3} - \frac{1}{3} = 1 \\ \xi(1) = -a_1 + b_0 + b_1 + b_{-1} = \frac{1}{3} + \frac{2}{3} = 1 \\ \xi(2) = a_1 - 2\left(b_1 - b_{-1}\right) = -\frac{1}{3} + 2\frac{2}{3} = 1 \\ \xi(3) = -a_1 + 3\left(b_1 + b_{-1}\right) = \frac{1}{3} + 3\frac{2}{3} = \frac{7}{3} \neq 1 \end{cases}$$

Pour le schéma [P], nous allons chercher b_0 et b_1 pour obtenir un schéma d'ordre 2. Pour cela on doit résoudre le système linéaire :

$$\begin{cases} \xi(0) = a_0 + a_1 = \frac{4}{3} - \frac{1}{3} = 1 \\ \xi(1) = -a_1 + b_0 + b_1 + b_{-1} = \frac{1}{3} + b_0 + b_1 = 1 \\ \xi(2) = a_1 - 2\left(b_1 - b_{-1}\right) = -\frac{1}{3} - 2b_1 = 1 \end{cases}$$

On trouve $b_0 = \frac{4}{3}$ et $b_1 = -\frac{2}{3}$.

Le schéma predictor-corrector s'écrit

$$\begin{cases} u_0 = y_0, \\ u_1 \approx y_1, \\ \tilde{u} = \frac{4}{3} u_n - \frac{1}{3} u_{n-1} + h\left(\frac{4}{3}\varphi_n - \frac{2}{3}\varphi_{n-1}\right) \quad [\mathrm{P}] \\ u_{n+1} = \frac{4}{3} u_n - \frac{1}{3} u_{n-1} + h\frac{2}{3}\varphi(t_{n+1}, \tilde{u}) \end{cases}$$

Comme le prédictor est d'ordre 2 et le corrector d'ordre 2 aussi, le schéma prédictor-corrector est d'ordre 2 (car $\min(\omega_{[C]}, \omega_{[P]} + 1) = 2$).

```
# variables globales
t0     = 0
tfinal = 1
y0     = 0

phi = lambda t,y : 1/(1+t**2)-2*y**2
sol_exacte = lambda t : t/(1+t**2)


# ================================================
# Multipas
# ================================================

q = 2
aa = [4/3,-1/3] # P et C
bb = [4/3,-2/3] # P
bm1 = 2/3 # C

def multipas_P(phi, tt, sol_exacte):
    h  = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    uu[:q] = sol_exacte(tt[:q])
    for n in range(q-1,len(tt) - 1):
        k_0 = phi(tt[n],uu[n])
        k_1 = phi(tt[n-1],uu[n-1])
        uu[n+1] = aa[0]*uu[n] + aa[1]*uu[n-1] + h*(bb[0]*k_0+bb[1]*k_1)
    return uu

def multipas_C(phi, tt, sol_exacte):
    h  = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    uu[:q] = sol_exacte(tt[:q])
    for n in range(q-1,len(tt) - 1):
        k_0 = phi(tt[n],uu[n])
        k_1 = phi(tt[n-1],uu[n-1])
        temp = fsolve ( lambda x : -x + aa[0]*uu[n] + aa[1]*uu[n-1] + h*bm1*phi(tt[n+1],x) , uu[n])
        uu[n+1] = temp[0]
    return uu

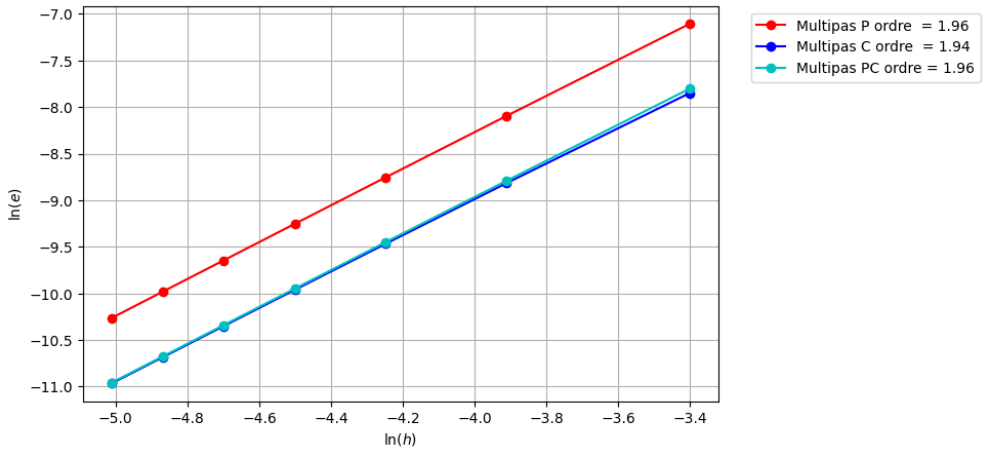
def multipas_PC(phi, tt, sol_exacte):
    h  = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    uu[:q] = sol_exacte(tt[:q])
    for n in range(q-1,len(tt) - 1):
        k_0 = phi(tt[n],uu[n])
        k_1 = phi(tt[n-1],uu[n-1])
        u_tilde = aa[0]*uu[n] + aa[1]*uu[n-1] + h*(bb[0]*k_0+bb[1]*k_1)
        uu[n+1] = aa[0]*uu[n] + aa[1]*uu[n-1] + h*bm1*phi(tt[n+1],u_tilde)
    return uu



H = []
err_p = []
err_c = []
err_pc = []
N = 10
for k in range(7):
    N += 20
    tt = np.linspace(t0, tfinal, N + 1)
```

```
    h = tt[1] - tt[0]
    yy = sol_exacte(tt)
    uu_p = multipas_P(phi, tt, sol_exacte)
    uu_c = multipas_C(phi, tt, sol_exacte)
    uu_pc = multipas_PC(phi, tt, sol_exacte)
    H.append(h)
    err_p.append(np.linalg.norm(uu_p-yy, np.inf))
    err_c.append(np.linalg.norm(uu_c-yy, np.inf))
    err_pc.append(np.linalg.norm(uu_pc-yy, np.inf))


plt.figure(figsize=(8,5))
plt.plot(np.log(H), np.log(err_p), 'r-o', label=f'Multipas P ordre  = {np.polyfit(np.log(H),np.log(err_p)
 ↪  , 1)[0]:1.2f}')
plt.plot(np.log(H), np.log(err_c), 'b-o', label=f'Multipas C ordre  = {np.polyfit(np.log(H),np.log(err_c)
 ↪  , 1)[0]:1.2f}' )
plt.plot(np.log(H), np.log(err_pc), 'c-o', label=f'Multipas PC ordre =
 ↪  {np.polyfit(np.log(H),np.log(err_pc), 1)[0]:1.2f}')
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
plt.legend(bbox_to_anchor=(1.04, 1), loc='upper left')
plt.grid(True);
```



18.3.4. Exercice : Construction d'un schéma Predictor-Corrector

Écrire le schéma predictor-corrector basé sur AM-3 AB-3. Attention à bien initialiser la suite récurrente.

- AB-1

$$u_{n+1} = u_n + h\varphi(t_n, u_n)$$

- AB-2

$$u_{n+1} = u_n + \frac{h}{2}\left(3\varphi(t_n, u_n) - \varphi(t_{n-1}, u_{n-1})\right)$$

- AB-3

$$u_{n+1} = u_n + \frac{h}{12}\left(23\varphi(t_n, u_n) - 16\varphi(t_{n-1}, u_{n-1}) + 5\varphi(t_{n-2}, u_{n-2})\right)$$

- AM-3

$$u_{n+1} = u_n + \frac{h}{24}\left(9\varphi(t_{n+1}, u_{n+1}) + 19\varphi(t_n, u_n) - 5\varphi(t_{n-1}, u_{n-1}) + \varphi(t_{n-2}, u_{n-2})\right)$$

donc

$$\begin{cases} u_0 = y_0, \\ u_1 = u_0 + h\varphi(t_0, u_0) \\ u_2 = u_1 + \frac{h}{2}\left(3\varphi(t_1, u_1) - \varphi(t_0, u_0)\right) \\ \\ \tilde{u} = u_n + \frac{h}{12}\left(23\varphi(t_n, u_n) - 16\varphi(t_{n-1}, u_{n-1}) + 5\varphi(t_{n-2}, u_{n-2})\right) \\ u_{n+1} = u_n + \frac{h}{24}\left(9\varphi(t_{n+1}, \tilde{u}) + 19\varphi(t_n, u_n) - 5\varphi(t_{n-1}, u_{n-1}) + \varphi(t_{n-2}, u_{n-2})\right) \end{cases}$$

AM-3 est un schéma implicite à 3 pas d'ordre 4,
AB-3 est un schéma explicite à 3 pas d'ordre 3,
le schéma predictor-corrector est donc un schéma à 3 pas d'ordre 4.

Chapitre 19.

TD3 - Exercices sur les schémas de Runge-Kutta

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import fsolve

import sympy as sp

from IPython.display import display, Math, Markdown
```

19.1. Aide-mémoire

Définitions
Soit $s \geq 1$ un entier. On se donne les $2s + s^2$ coefficients suivants :

- $\mathbf{c} = (c_1, c_2, \dots, c_s)$,
- $\mathbf{b} = (b_1, b_2, \dots, b_s)$.
- $\mathbb{A} = (a_{ij})_{1 \leq i,j \leq s}$,

qu'on regroupe dans un tableau de Butcher :

$$\begin{array}{c|c} \mathbf{c} & \mathbb{A} \\ \hline & \mathbf{b}^T \end{array}.$$

La méthode de Runge-Kutta à s étages associée à ce tableau est définie par :

$$\begin{cases} u_0 = y(t_0) = y_0, \\ u_{n+1} = u_n + h \displaystyle\sum_{i=1}^{s} b_i K_i & n = 0, 1, \dots N-1 \\ K_i = \varphi\left(t_n + hc_i, u_n + h \displaystyle\sum_{j=1}^{s} a_{ij} K_j\right) & i = 1, \dots s. \end{cases}$$

- **Méthode implicite :** s'il existe un terme $a_{ij} \neq 0$ pour $j > i$, alors la méthode est dite implicite. Pour calculer les K_i, il faut résoudre un **système d'équations** non linéaires.
- **Méthode semi-implicite :** si $a_{ij} = 0$ pour $j > i$ (la matrice $\mathbb{A}$ est triangulaire inférieure) et au moins un terme de la diagonale est non nul, alors la méthode est dite semi-implicite. Pour calculer les K_i, il suffit de résoudre l'une après l'autre les **équations** non linéaires (et non pas un système).
- **Méthode explicite :** si $a_{ij} = 0$ pour $j \geq i$ (la matrice $\mathbb{A}$ est triangulaire inférieure stricte), alors la méthode est dite explicite. Le calcul des K_i est alors **direct**.

Théorème de Lax-Richtmyer
Soit $\omega \geq 1$. Le schéma est **convergent (d'ordre ω)** ssi il est **zéro-stable** et **consistant (d'ordre $\omega \geq 1$)**.

Zéro-stabilité
Les méthodes de Runge-Kutta sont **zéro-stables**.

Consistance (et ordre de consistance ω)
Une méthode de Runge-Kutta est **consistante d'ordre** $\omega \geq 1$ si elle satisfait les conditions suivantes.

- Pour $\omega \geq 1$ il faut
$$\begin{cases} 1 = \sum_{j=1}^{s} b_j \\ c_i = \sum_{j=1}^{s} a_{ij}, \quad i = 1, \dots, s \end{cases}$$
- Pour $\omega \geq 2$ il faut les conditions pour $\omega \geq 1$ **et**
$$\sum_{j=1}^{s} b_j c_j = \frac{1}{2} \quad \text{i.e.} \quad \mathbf{b}^T \mathbf{c} = \frac{1}{2}.$$
- Pour $\omega \geq 3$ il faut les conditions pour $\omega \geq 2$ **et**
$$\begin{cases} \sum_{j=1}^{s} b_j c_j^2 = \frac{1}{3} \\ \sum_{i=1}^{s} \sum_{j=1}^{s} b_i a_{ij} c_j = \frac{1}{6} \end{cases} \quad \text{i.e.} \quad \mathbf{b}^T \mathbb{A} \mathbf{c} = \frac{1}{6}$$
- Pour $\omega \geq 4$ il faut les conditions pour $\omega \geq 3$ **et**
$$\begin{cases} \sum_{j=1}^{s} b_j c_j^3 = \frac{1}{4} \\ \sum_{i=1}^{s} \sum_{j=1}^{s} b_i c_i a_{ij} c_j = \frac{1}{8} \\ \sum_{i=1}^{s} \sum_{j=1}^{s} b_i a_{ij} c_j^2 = \frac{1}{12} \\ \sum_{i=1}^{s} \sum_{j=1}^{s} \sum_{k=1}^{s} b_i a_{ij} a_{jk} c_k = \frac{1}{24} \end{cases} \quad \text{i.e.} \quad \mathbf{b}^T \mathbb{A}^2 \mathbf{c} = \frac{1}{24}$$
- Pour $\omega \geq 5$ il faut les conditions pour $\omega \geq 4$ **et**
$$\begin{cases} \sum_{j=1}^{s} b_j c_j^4 = \frac{1}{5} \\ \sum_{i,j=1}^{s} b_i c_i^2 a_{ij} c_j = \frac{1}{10} \\ \sum_{i=1}^{s} b_i \left(\sum_{j=1}^{s} a_{ij} c_j \right)^2 = \frac{1}{20} \\ \sum_{i,j=1}^{s} b_i a_{ij} c_j^3 = \frac{1}{20} \\ \sum_{i,j=1}^{s} b_i c_i a_{ij} c_j^2 = \frac{1}{15} \\ \sum_{i,j,k=1}^{s} b_i c_i a_{ij} a_{jk} c_k = \frac{1}{30} \\ \sum_{i,j,k=1}^{s} b_i a_{ij} c_j a_{jk} c_k = \frac{1}{40} \\ \sum_{i,j,k=1}^{s} b_i a_{ij} a_{jk} c_k^2 = \frac{1}{60} \\ \sum_{i,j,k,\ell=1}^{s} b_i a_{ij} a_{jk} a_{k\ell} c_\ell = \frac{1}{120} \end{cases} \quad \text{i.e.} \quad \mathbf{b}^T \mathbb{A}^3 \mathbf{c} = \frac{1}{120}$$

Barrière de Butcher
L'ordre $\omega \geq 1$ d'une méthode de Runge-Kutta à $s \geq 1$ étapes convergente satisfait les inégalités suivantes :

$$\omega \leq \begin{cases} 2s, & \text{si la méthode est implicite,} \\ s, & \text{si la méthode est explicite et } s < 5, \\ s-1, & \text{si la méthode est explicite et } s \geq 5. \end{cases}$$

A-stabilité
Soit $\beta > 0$ un nombre réel positif et considérons le problème de Cauchy

$$\begin{cases} y'(t) = -\beta y(t), & \text{pour } t > 0, \\ y(0) = 1 \end{cases}$$

Sa solution est $y(t) = e^{-\beta t}$ et l'on a $y(t) \xrightarrow[t \to +\infty]{} 0$.
Une méthode de Runge-Kutta à $s \geq 1$ étages pour $y'(t) = -\beta y(t)$ s'écrit :

$$\begin{cases} u_0 = y_0, \\ u_{n+1} = u_n + h \displaystyle\sum_{i=1}^{s} b_i K_i & n = 0, 1, \dots N-1 \\ K_i = -\beta \left(u_n + h \displaystyle\sum_{j=1}^{s} a_{ij} K_j \right) & i = 1, \dots s. \end{cases}$$

Si, sous d'éventuelles conditions sur h, on a $|u_n| \xrightarrow[n \to +\infty]{} 0$ alors on dit que **le schéma est A-stable.**
Étant une méthode à un pas, on peut réécrire u_{n+1} en fonction de u_n :

$$u_{n+1} = q(h\beta) u_n.$$

Le schéma est A-stable ssi $|q(h\beta)| < 1$.
Remarque : en introduisant les notations vectorielles $\mathbf{K} = (K_1, \dots, K_s)^T$ et $\mathbf{1} = (1, \dots, 1)^T$, le coefficient d'amplification se réécrit de manière compacte :

$$q(\beta h) = (1 - \beta h \mathbf{b}^T (\mathbb{I} + \beta h \mathbb{A})^{-1} \mathbf{1}) = \frac{\det(\mathbb{I}_s + \beta h \mathbb{A} - \beta h \mathbf{1}_s \mathbf{b}^T)}{\det(\mathbb{I}_s + \beta h \mathbb{A})}.$$

Le calcul des conditions de consistance est un peu long et fastidieux dès que s est grand ou ω est élevé. On peut utiliser un logiciel de calcul formel pour les obtenir, par exemple le module `sympy`.

On doit simplement indiquer le nombre d'étages s et le type de méthode (implicite, explicite, semi-implicite) pour obtenir les conditions de consistance. Indiquer le type de méthode permet de simplifier les calculs car de nombreux coefficients de la matrice sont nuls.

```
# ========================================================================
# Fonctions pour calculer les conditions de consistance pour un schéma RK.
#
# La fonction ordre_RK calcule les conditions pour les ordres 1, 2, 3 et 4 (même si théoriquement cela
 ↪ n'est pas possible)
# - affiche la matrice de Butcher
# - affiche les conditions de consistance obtenues pour chaque ordre par la fonction ordre_i
# - renvoie une liste d'équations pour une éventuelle résolution
#
# Chaque fonction ordre_i calcule les conditions pour être d'ordre i
# - peut afficher les conditions de consistance [actuellement en commentaire]
# - renvoie la liste d'équations
#
# ========================================================================
```

```
def ordre_RK(s, type="implicite", A=None, b=None, c=None):

    j = sp.symbols('j')

    if type is not None:
        if type == "implicite":       ordre_max = 2*s
        elif type=="semi-implicite":  ordre_max = 2*s
        elif type=="explicite":       ordre_max = s
    else:
        ordre_max = 5 # par défaut, on affiche les conditions jusqu'à l'ordre 5

    if c is None: c = sp.symbols(f'c_0:{s}')
    if b is None: b = sp.symbols(f'b_0:{s}')
    if A is None:
        if type == "implicite":       A = sp.MatrixSymbol('a', s, s)
        elif type=="semi-implicite":  A = sp.Matrix(s, s, lambda i,j: sp.Symbol('a{}{}'.format(i, j)) if
         ↪  i >= j else 0)
        elif type=="explicite":       A = sp.Matrix(s, s, lambda i,j: sp.Symbol('a{}{}'.format(i, j)) if
         ↪  i > j else 0)
    else:
        A = sp.Matrix(A)

    display(Markdown("**Matrice de Butcher**"))
    But = matrice_Butcher(s, A, b, c)

    Eqs = []

    display(Markdown(f"**On a {s+1} conditions pour avoir consistance (= pour être d'ordre 1)**"))
    Eq_1 = ordre_1(s, A, b, c, j)
    Eqs.append(Eq_1)
    display(*Eq_1)

    if ordre_max < 2:
        display(Markdown("**On ne peut pas avoir d'ordre 2 avec ce schéma**"))
    else:
        display(Markdown("**On doit ajouter 1 condition pour être d'ordre 2**"))
        Eq_2 = ordre_2(s, A, b, c, j)
        Eqs.append([Eq_2])
        display(Eq_2)

    if ordre_max < 3:
        display(Markdown("**On ne peut pas avoir d'ordre 3 avec ce schéma**"))
    else:
        display(Markdown("**On doit ajouter 2 conditions pour être d'ordre 3**"))
        Eq_3 = ordre_3(s, A, b, c, j)
        Eqs.append(Eq_3)
        display(*Eq_3)

    if ordre_max < 4:
        display(Markdown("**On ne peut pas avoir d'ordre 4 avec ce schéma**"))
    else:
        display(Markdown("**On doit ajouter 4 conditions pour être d'ordre 4**"))
        Eq_4 = ordre_4(s, A, b, c, j)
        Eqs.append(Eq_4)
        display(*Eq_4)

    if ordre_max < 5:
        display(Markdown("**On ne peut pas avoir d'ordre 5 avec ce schéma**"))
    else:
        display(Markdown("**On doit ajouter 6 conditions pour être d'ordre 5**"))
        Eq_5 = ordre_5(s, A, b, c, j)
        Eqs.append(Eq_5)
```

```
        display(*Eq_5)

    return Eqs


def matrice_Butcher(s, A, b, c):
    But = sp.Matrix(A)
    But = But.col_insert(0, sp.Matrix(c))
    last = [sp.Symbol(" ")]
    last.extend(b)
    But = But.row_insert(s,sp.Matrix(last).T)
    display(Math(sp.latex(sp.Matrix(But))))
    return But


def ordre_1(s, A, b, c, j):
    texte = r"\sum_{j=1}^s b_j =" + f"{sum(b).simplify()}"
    texte += r"\text{ doit être égale à }1"
    # display(Math(texte))
    Eq_1 = [sp.Eq( sum(b).simplify(), 1 )]
    for i in range(s):
        somma = sp.summation(A[i,j],(j,0,s-1)).simplify()
        texte = r'\sum_{j=1}^s a_{'+str(i)+r'j}=' + sp.latex( somma )
        texte += r"\text{ doit être égale à }"+sp.latex(c[i])
        # display(Math( texte ))
        Eq_1.append(sp.Eq( somma, c[i] ))
    return Eq_1


def ordre_2(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j=' +sp.latex(sum([b[i]*c[i] for i in range(s)]).simplify())
    texte += r"\text{ doit être égale à }\frac{1}{2}"
    # display(Math(texte))
    Eq_2 = sp.Eq( sum([b[i]*c[i] for i in range(s)]).simplify(), sp.Rational(1,2) )
    return Eq_2


def ordre_3(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j^2='
    texte += sp.latex( sum([b[i]*c[i]**2 for i in range(s)]).simplify() )
    texte += r"\text{ doit être égale à }\frac{1}{3}"
    # display(Math(texte))
    Eq_3_1 = sp.Eq( sum([b[i]*c[i]**2 for i in range(s)]).simplify(), sp.Rational(1,3) )
    texte = r'\sum_{i,j=1}^s b_i a_{ij} c_j='
    somma = sum([b[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{6}"
    # display(Math(texte))
    Eq_3_2 = sp.Eq( sum([b[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,6) )
    return [Eq_3_1, Eq_3_2]


def ordre_4(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j^3='
    texte += sp.latex( sum([b[i]*c[i]**3 for i in range(s)]).simplify() )
    texte += r"\text{ doit être égale à }\frac{1}{4}"
    # display(Math(texte))
    Eq_4_1 = sp.Eq( sum([b[i]*c[i]**3 for i in range(s)]).simplify(), sp.Rational(1,4) )
    texte = r'\sum_{i,j=1}^s b_i c_i a_{ij} c_j='
    somma = sum([b[i]*c[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{8}"
```

```
    # display(Math(texte))
    Eq_4_2 = sp.Eq( sum([b[i]*c[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,8) )
    texte = r'\sum_{i,j=1}^s b_i a_{ij} c_j^2='
    somma = sum([b[i]*A[i,j]*c[j]**2 for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{12}"
    # display(Math(texte))
    Eq_4_3 = sp.Eq( sum([b[i]*A[i,j]*c[j]**2 for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,12) )
    texte = r'\sum_{i,j,k=1}^s b_i a_{ij} a_{jk} c_k='
    somma = sum([b[i]*A[i,j]*A[j,k]*c[k] for k in range(s) for j in range(s) for i in
↪   range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{24}"
    # display(Math(texte))
    Eq_4_4 = sp.Eq( sum([b[i]*A[i,j]*A[j,k]*c[k] for k in range(s) for j in range(s) for i in
↪   range(s)]).simplify(), sp.Rational(1,24) )
    return [Eq_4_1, Eq_4_2, Eq_4_3, Eq_4_4]



def ordre_5(s, A, b, c, j):
    Eq_5_1 = sp.Eq(sum([b[i]*c[i]**4 for i in range(s)]).simplify(), sp.Rational(1, 5))

    somma = sum([b[i]*c[i]**2*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
    Eq_5_2 = sp.Eq(somma, sp.Rational(1, 10))

    somma = sum([b[i]*(sum([A[i,j]*c[j] for j in range(s)]))**2 for i in range(s)]).simplify()
    Eq_5_3 = sp.Eq(somma, sp.Rational(1, 20))

    somma = sum([b[i]*A[i,j]*c[j]**3 for j in range(s) for i in range(s)]).simplify()
    Eq_5_4 = sp.Eq(somma, sp.Rational(1, 20))

    somma = sum([b[i]*c[i]*A[i,j]*c[j]**2 for j in range(s) for i in range(s)]).simplify()
    Eq_5_5 = sp.Eq(somma, sp.Rational(1, 15))

    somma = sum([b[i]*c[i]*A[i,j]*A[j,k]*c[k]
                 for k in range(s) for j in range(s) for i in range(s)]).simplify()
    Eq_5_6 = sp.Eq(somma, sp.Rational(1, 30))

    somma = sum([b[i]*A[i,j]*c[j]*A[j,k]*c[k]
                 for k in range(s) for j in range(s) for i in range(s)]).simplify()
    Eq_5_7 = sp.Eq(somma, sp.Rational(1, 40))

    somma = sum([b[i]*A[i,j]*A[j,k]*c[k]**2
                 for k in range(s) for j in range(s) for i in range(s)]).simplify()
    Eq_5_8 = sp.Eq(somma, sp.Rational(1, 60))

    somma = sum([b[i]*A[i,j]*A[j,k]*A[k,l]*c[l]
                 for l in range(s) for k in range(s)
                 for j in range(s) for i in range(s)]).simplify()
    Eq_5_9 = sp.Eq(somma, sp.Rational(1, 120))

    return [Eq_5_1, Eq_5_2, Eq_5_3, Eq_5_4, Eq_5_5, Eq_5_6, Eq_5_7, Eq_5_8, Eq_5_9]
```

```
# TEST 1
# ================================================================================
# Calcul des conditions de consistance pour un schéma RK
# On doit indiquer le nombre d'étages s et le type de schéma implicite, semi-implicite ou explicite
# Affichage des conditions de consistance : attention aux indices qui vont de 0 à s-1
# ================================================================================
s = 5
```

```
type = "explicite" # implicite, semi-implicite, explicite
display(Markdown(f"<mark>**Conditions pour un schéma {type} avec s = {s} étages**</mark>"))
Eqs = ordre_RK(s=s, type=type)
```

Conditions pour un schéma explicite avec s = 5 étages

Matrice de Butcher

$$\begin{bmatrix} c_0 & 0 & 0 & 0 & 0 & 0 \\ c_1 & a_{10} & 0 & 0 & 0 & 0 \\ c_2 & a_{20} & a_{21} & 0 & 0 & 0 \\ c_3 & a_{30} & a_{31} & a_{32} & 0 & 0 \\ c_4 & a_{40} & a_{41} & a_{42} & a_{43} & 0 \\ & b_0 & b_1 & b_2 & b_3 & b_4 \end{bmatrix}$$

On a 6 conditions pour avoir consistance (= pour être d'ordre 1)

$$b_0 + b_1 + b_2 + b_3 + b_4 = 1$$

$$0 = c_0$$

$$a_{10} = c_1$$

$$a_{20} + a_{21} = c_2$$

$$a_{30} + a_{31} + a_{32} = c_3$$

$$a_{40} + a_{41} + a_{42} + a_{43} = c_4$$

On doit ajouter 1 condition pour être d'ordre 2

$$b_0c_0 + b_1c_1 + b_2c_2 + b_3c_3 + b_4c_4 = \frac{1}{2}$$

On doit ajouter 2 conditions pour être d'ordre 3

$$b_0c_0^2 + b_1c_1^2 + b_2c_2^2 + b_3c_3^2 + b_4c_4^2 = \frac{1}{3}$$

$$a_{10}b_1c_0 + a_{20}b_2c_0 + a_{21}b_2c_1 + a_{30}b_3c_0 + a_{31}b_3c_1 + a_{32}b_3c_2 + a_{40}b_4c_0 + a_{41}b_4c_1 + a_{42}b_4c_2 + a_{43}b_4c_3 = \frac{1}{6}$$

On doit ajouter 4 conditions pour être d'ordre 4

$$b_0c_0^3 + b_1c_1^3 + b_2c_2^3 + b_3c_3^3 + b_4c_4^3 = \frac{1}{4}$$

$$a_{10}b_1c_0c_1 + a_{20}b_2c_0c_2 + a_{21}b_2c_1c_2 + a_{30}b_3c_0c_3 + a_{31}b_3c_1c_3 + a_{32}b_3c_2c_3 + a_{40}b_4c_0c_4 + a_{41}b_4c_1c_4 + a_{42}b_4c_2c_4 + a_{43}b_4c_3c_4 = \frac{1}{8}$$

$$a_{10}b_1c_0^2 + a_{20}b_2c_0^2 + a_{21}b_2c_1^2 + a_{30}b_3c_0^2 + a_{31}b_3c_1^2 + a_{32}b_3c_2^2 + a_{40}b_4c_0^2 + a_{41}b_4c_1^2 + a_{42}b_4c_2^2 + a_{43}b_4c_3^2 = \frac{1}{12}$$

$$a_{10}a_{21}b_2c_0 + a_{10}a_{31}b_3c_0 + a_{10}a_{41}b_4c_0 + a_{20}a_{32}b_3c_0 + a_{20}a_{42}b_4c_0 + a_{21}a_{32}b_3c_1 + a_{21}a_{42}b_4c_1 + a_{30}a_{43}b_4c_0 + a_{31}a_{43}b_4c_1 + a_{32}a_{43}b_4c_2 = \frac{1}{24}$$

On doit ajouter 6 conditions pour être d'ordre 5

$$b_0c_0^4 + b_1c_1^4 + b_2c_2^4 + b_3c_3^4 + b_4c_4^4 = \frac{1}{5}$$

$$a_{10}b_1c_0c_1^2 + a_{20}b_2c_0c_2^2 + a_{21}b_2c_1c_2^2 + a_{30}b_3c_0c_3^2 + a_{31}b_3c_1c_3^2 + a_{32}b_3c_2c_3^2 + a_{40}b_4c_0c_4^2 + a_{41}b_4c_1c_4^2 + a_{42}b_4c_2c_4^2 + a_{43}b_4c_3c_4^2 = \frac{1}{10}$$

$$a_{10}^2b_1c_0^2 + b_2\left(a_{20}c_0 + a_{21}c_1\right)^2 + b_3\left(a_{30}c_0 + a_{31}c_1 + a_{32}c_2\right)^2 + b_4\left(a_{40}c_0 + a_{41}c_1 + a_{42}c_2 + a_{43}c_3\right)^2 = \frac{1}{20}$$

$$a_{10}b_1c_0^3 + a_{20}b_2c_0^3 + a_{21}b_2c_1^3 + a_{30}b_3c_0^3 + a_{31}b_3c_1^3 + a_{32}b_3c_2^3 + a_{40}b_4c_0^3 + a_{41}b_4c_1^3 + a_{42}b_4c_2^3 + a_{43}b_4c_3^3 = \frac{1}{20}$$

$$a_{10}b_1c_0^2c_1+a_{20}b_2c_0^2c_2+a_{21}b_2c_1^2c_2+a_{30}b_3c_0^2c_3+a_{31}b_3c_1^2c_3+a_{32}b_3c_2^2c_3+a_{40}b_4c_0^2c_4+a_{41}b_4c_1^2c_4+a_{42}b_4c_2^2c_4+a_{43}b_4c_3^2c_4 = \frac{1}{15}$$

$$a_{10}a_{21}b_2c_0c_2+a_{10}a_{31}b_3c_0c_3+a_{10}a_{41}b_4c_0c_4+a_{20}a_{32}b_3c_0c_3+a_{20}a_{42}b_4c_0c_4+a_{21}a_{32}b_3c_1c_3+a_{21}a_{42}b_4c_1c_4+a_{30}a_{43}b_4c_0c_4+a_{31}a_{43}b_4c_1c_4+a_{32}a_{43}b_4c_2c_4 = \frac{1}{30}$$

$$a_{10}a_{21}b_2c_0c_1+a_{10}a_{31}b_3c_0c_1+a_{10}a_{41}b_4c_0c_1+a_{20}a_{32}b_3c_0c_2+a_{20}a_{42}b_4c_0c_2+a_{21}a_{32}b_3c_1c_2+a_{21}a_{42}b_4c_1c_2+a_{30}a_{43}b_4c_0c_3+a_{31}a_{43}b_4c_1c_3+a_{32}a_{43}b_4c_2c_3 = \frac{1}{40}$$

$$a_{10}a_{21}b_2c_0^2+a_{10}a_{31}b_3c_0^2+a_{10}a_{41}b_4c_0^2+a_{20}a_{32}b_3c_0^2+a_{20}a_{42}b_4c_0^2+a_{21}a_{32}b_3c_1^2+a_{21}a_{42}b_4c_1^2+a_{30}a_{43}b_4c_0^2+a_{31}a_{43}b_4c_1^2+a_{32}a_{43}b_4c_2^2 = \frac{1}{60}$$

$$a_{10}a_{21}a_{32}b_3c_0+a_{10}a_{21}a_{42}b_4c_0+a_{10}a_{31}a_{43}b_4c_0+a_{20}a_{32}a_{43}b_4c_0+a_{21}a_{32}a_{43}b_4c_1 = \frac{1}{120}$$

```
# TEST 2 :  schéma de Cash-Karp, un classique explicite à 6 étages d'ordre 5
# ================================================================================

s = 6
type = "explicite"

# Tableau de Butcher de Cash-Karp (1990) - explicite, ordre 5
A = sp.Matrix([
    [0, 0, 0, 0, 0, 0],
    [sp.Rational(1,5),       0, 0, 0, 0, 0],
    [sp.Rational(3,40),      sp.Rational(9,40),       0, 0, 0, 0],
    [sp.Rational(3,10),     -sp.Rational(9,10),       sp.Rational(6,5),       0, 0, 0],
    [-sp.Rational(11,54),    sp.Rational(5,2),        -sp.Rational(70,27),    sp.Rational(35,27),      0,
 ↪  0],
    [sp.Rational(1631,55296), sp.Rational(175,512),   sp.Rational(575,13824), sp.Rational(44275,110592),
 ↪  sp.Rational(253,4096), 0]
])

# c_i = somme des lignes de A (condition de consistance)
c = [sum(A[i, j] for j in range(s)) for i in range(s)]
# c = [0, 1/5, 3/10, 3/5, 1, 7/8]

b = [
    sp.Rational(37, 378),
    0,
    sp.Rational(250, 621),
    sp.Rational(125, 594),
    0,
    sp.Rational(512, 1771)
]

display(Markdown(f"### Conditions pour un schéma {type} avec s = {s} étages"))
Eqs = ordre_RK(s=s, type=type, A=A, b=b, c=c)
```

19.1.1. Conditions pour un schéma explicite avec s = 6 étages

Matrice de Butcher

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{5} & \frac{1}{5} & 0 & 0 & 0 & 0 & 0 \\ \frac{3}{10} & \frac{3}{40} & \frac{9}{40} & 0 & 0 & 0 & 0 \\ \frac{3}{5} & \frac{3}{10} & -\frac{9}{10} & \frac{6}{5} & 0 & 0 & 0 \\ 1 & -\frac{11}{54} & \frac{5}{2} & -\frac{70}{27} & \frac{35}{27} & 0 & 0 \\ \frac{7}{8} & \frac{1631}{55296} & \frac{175}{512} & \frac{575}{13824} & \frac{44275}{110592} & \frac{253}{4096} & 0 \\ & \frac{37}{378} & 0 & \frac{250}{621} & \frac{125}{594} & 0 & \frac{512}{1771} \end{bmatrix}$$

On a 7 conditions pour avoir consistance (= pour être d'ordre 1)

True

True

True

True

True

True

True

On doit ajouter 1 condition pour être d'ordre 2

True

On doit ajouter 2 conditions pour être d'ordre 3

True

True

On doit ajouter 4 conditions pour être d'ordre 4

True

True

True

True

On doit ajouter 6 conditions pour être d'ordre 5

True

True

True

True

True

True

True

True

True

Le calcul des conditions de consistance peut aussi s'averer fastidieux pour les méthodes à plusieurs étages. On peut aussi utiliser module `sympy` pour les obtenir :

```
# ============================================================================
# Calcul de la fonction de stabilité q(x) pour un schéma de Runge-Kutta
# ============================================================================

def calculer_q_rk_general(s, A=None, b=None):
    x = sp.symbols('x', positive=True)
    # beta, h = sp.symbols('beta h')
    # x = beta * h

    if A is None: A = sp.Matrix(s, s, lambda i, j: sp.symbols(f'a_{i+1}{j+1}'))
    if b is None: b = sp.Matrix(s, 1, lambda i, j: sp.symbols(f'b_{i+1}'))

    ones = sp.Matrix.ones(s, 1)
    I = sp.eye(s)
```

```
    q = sp.det(I + x * A - x * ones * b.T) / sp.det(I + x * A)
    return sp.simplify(q)
```

```
# TEST
# ============================================================================

A = sp.Matrix([
    [0, 0, 0],
    [sp.Rational(1, 2), 0, 0],
    [0, sp.Rational(1, 2), 0]
])
b = sp.Matrix([sp.Rational(1, 6), sp.Rational(2, 3), sp.Rational(1, 6)])
s = len(b)
q = calculer_q_rk_general(s=s, A=A, b=b)
display(Markdown(f"$$q(x) = {sp.latex(q)}$$"))
```

$$q(x) = -\frac{x^3}{24} + \frac{5x^2}{12} - x + 1$$

19.2. Exercices

19.2.1. Exercice : schéma de Ralston ($s = 2$, explicite)

On considère la méthode RK donnée par la matrice suivante :

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \frac{2}{3} & \frac{2}{3} & 0 \\ \hline & \frac{1}{4} & \frac{3}{4} \end{array}$$

1. Écrire le schéma correspondant.
2. Déterminer l'ordre de convergence.
3. Déterminer sous quelle condition sur h ce schéma est A-stable. Déterminer sous quelle condition sur h la convergence est monotone.
4. Tester empiriquement l'ordre de convergence de ce schéma sur le problème de Cauchy $y'(t) = t(1-t^2)-y(t)$ et $y(0) = 1$ pour $t \in [0, 2]$.

$$\begin{cases} u_0 = y_0 \\ k_0 = \varphi\Big(t_n, u_n\Big) \\ k_1 = \varphi\Big(t_n + \frac{2}{3}h, u_n + \frac{2}{3}hk_0\Big) \\ u_{n+1} = u_n + h(\frac{1}{4}k_0 + \frac{3}{4}k_1) \end{cases}$$

Il s'agit d'un schéma à $s = 2$ étages explicite.

D'après la barrière de Butcher, un schéma RK à $s = 2$ étages explicite ne peut pas être d'ordre supérieur à $s = 2$. On peut bien-sûr écrire les formules à la main, mais on peut aussi utiliser sympy comme suit :

```
s = 2

b = [ sp.Rational(1,4), sp.Rational(3,4) ]
c = [ 0, sp.Rational(2,3) ]
A = [ [ 0, 0] , [ sp.Rational(2,3),  0] ]

Eqs = ordre_RK(s=s, A=A, b=b, c=c, type="explicite")
```

Matrice de Butcher

$$\begin{bmatrix} 0 & 0 & 0 \\ \frac{2}{3} & \frac{2}{3} & 0 \\ & \frac{1}{4} & \frac{3}{4} \end{bmatrix}$$

On a 3 conditions pour avoir consistance (= pour être d'ordre 1)

True

True

True

On doit ajouter 1 condition pour être d'ordre 2

True

On ne peut pas avoir d'ordre 3 avec ce schéma

On ne peut pas avoir d'ordre 4 avec ce schéma

On ne peut pas avoir d'ordre 5 avec ce schéma

En conclusion, le schéma est d'ordre 2.

On applique le schéma au problème de Cauchy $y'(t) = -\beta y(t)$ et $y(0) = 1$ avec $\beta > 0$ dont la solution est $y(t) = e^{-\beta t}$. On obtient alors la relation de récurrence suivante :

```
u_n = sp.symbols(r'u_n')
u_np1 = sp.symbols(r'u_{n+1}')

dt = sp.symbols(r'h', positive=True)
x = sp.symbols(r'x', positive=True)

k_0 = sp.symbols(r'k_0')
k_1 = sp.symbols(r'k_1')
kk = [k_0, k_1]

beta = sp.symbols(r'\beta', positive=True)
phi = lambda y: -beta*y

Eqs = [ sp.Eq( kk[i] ,  phi(u_n + dt*sum([kk[j]*A[i][j] for j in range(s)])) ) for i in range(s) ]
for eq in Eqs:
    display(eq)

sol = sp.solve( Eqs, kk ) # c'est un dictionnaire
KK = [sol[kk[i]] for i in range(s)]
schema = sp.Eq( u_np1 , (u_n + dt*sum([KK[j]*b[j] for j in range(s)])).factor(u_n) )
display(schema)
```

$$k_0 = -\beta u_n$$

$$k_1 = -\beta\left(\frac{2hk_0}{3} + u_n\right)$$

$$u_{n+1} = u_n\left(\frac{\beta^2 h^2}{2} - \beta h + 1\right)$$

```
# sinon directement avec la fonction calculer_q_rk_general
b = sp.Matrix(b)
A = sp.Matrix(A)
q = calculer_q_rk_general(s=s,A=A,b=b)
display(Markdown(f"$$q(x) = {sp.latex(q)}$$"))
```

$$q(x) = \frac{x^2}{2} - x + 1$$

On peut expliciter u_n en fonction de u_0 et du produit βh (on notera x le produit βh) :

```
q = (schema.rhs/u_n).simplify().subs(beta*dt,x)
display(Math(sp.latex(sp.Eq(sp.Symbol('q(x)'),q.simplify()))))
display( sp.solve(abs(q)<1)  )
```

$$q(x) = \frac{x^2}{2} - x + 1$$

$$x < 2$$

La suite vérifie $u_n \xrightarrow[n\to\infty]{} 0$ ssi $|q(x)| < 1$. En étudiant brièvement la fonction $q(x)$, on trouve que $q(x) < 1$ ssi $x < 6$ donné ci-dessous:.

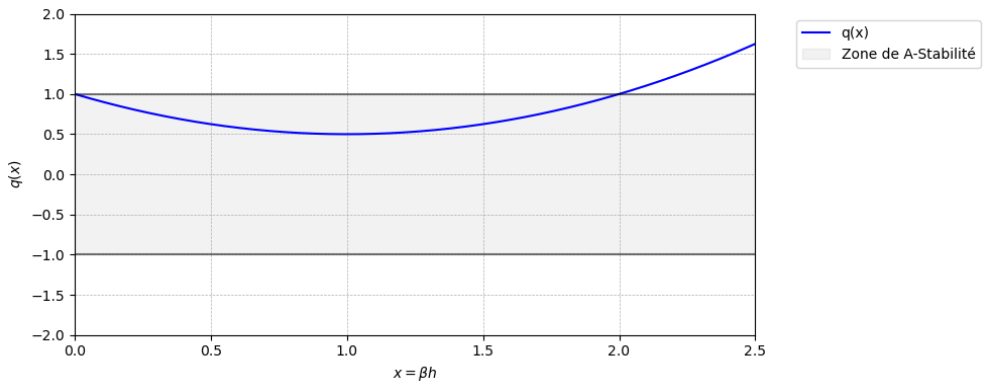
```
q_np = sp.lambdify(x, q, 'numpy')

x_max = 2.5
xx = np.linspace(0, x_max, 500)
plt.figure(figsize=(10, 4))
plt.plot(xx, q_np(xx), label='q(x)', color='blue')

# Tracé des limites de stabilité (|q(x)| = 1)
plt.axhline(y=1, color='black', linestyle='-', alpha=0.5)
plt.axhline(y=-1, color='black', linestyle='-', alpha=0.5)
plt.fill_between(xx, -1, 1, color='gray', alpha=0.1, label='Zone de A-Stabilité')

# Configuration des axes
plt.xlabel(r'$x = \beta h$')
plt.ylabel(r'$q(x)$')
plt.ylim(-2, 2) # On zoome sur la zone intéressante [-1, 1]
plt.xlim(0, x_max)
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left');


plt.tight_layout()
plt.show()
```




```
##################################################
# EDO
##################################################
t0     = 0
tfinal = 2
y0     = 1
```

```
phi = lambda t,y: t*(1-t**2)-y

#####################################################
# solution exacte
#####################################################
t   = sp.Symbol('t')
y   = sp.Function('y')
edo = sp.Eq( (y(t)).diff() , phi(t,y(t)) )
solpar = sp.dsolve(edo , ics={y(t0):y0})
display(solpar)

sol_exacte = sp.lambdify(t,solpar.rhs,'numpy')

#####################################################
# schéma
#####################################################

# Paramètres du schéma recuperés depuis la matrice de Butcher obtenue précédemment
cc = np.array(c)
bb = np.array(b)
AA = np.array(A)

s = len(cc)
# Schéma de Runge-Kutta implicite à s étages
def RK(phi, tt, y0):
    h  = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    uu[0] = y0
    for n in range(len(tt) - 1):
        k_0 = phi(tt[n], uu[n])
        k_1 = phi(tt[n]+cc[1]*h, uu[n]+h*AA[1,0]*k_0)
        KK = [k_0, k_1]
        uu[n+1] = uu[n] + h*sum([KK[j]*b[j] for j in range(s)])
    return uu


#####################################################
# ordre
#####################################################
H = []
err = []


plt.figure(figsize=(16,5))

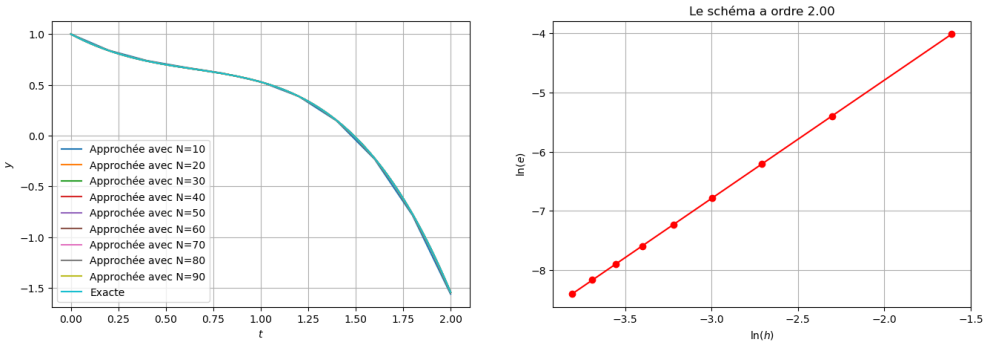
ax1 = plt.subplot(1,2,1)
plt.xlabel('$t$')
plt.ylabel('$y$')
ax1.grid(True)

for N in range(10,100,10):
    tt = np.linspace(t0, tfinal, N + 1)
    H.append( tt[1] - tt[0] )
    yy = sol_exacte(tt)
    uu = RK(phi, tt, y0)
    err.append( np.linalg.norm(uu-yy,np.inf) )
    if err[-1]<1.e-10:
        break
    ax1.plot(tt,uu,label=f'Approchée avec N={N}')

ax1.plot(tt,yy,label='Exacte') # sur la grille la plus fine
ax1.legend()
```

```
ax2 = plt.subplot(1,2,2)
ax2.plot( np.log(H), np.log(err), 'r-o')
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
plt.title(f'Le schéma a ordre {np.polyfit(np.log(H),np.log(err),1)[0]:1.2f}')
ax2.grid(True)
```

$y(t) = -t^3 + 3t^2 - 5t + 5 - 4e^{-t}$



19.2.2. Exercice : schéma de Radau IA ($s = 2$, implicite)

On considère la méthode RK donnée par la matri suivante :

$$\begin{array}{c|cc} 0 & \frac{1}{4} & -\frac{1}{4} \\ \frac{2}{3} & \frac{1}{4} & \frac{5}{12} \\ \hline & \frac{1}{4} & \frac{3}{4} \end{array}$$

1. Écrire le schéma correspondant.
2. Déterminer l'ordre de convergence.
3. Déterminer sous quelle condition sur h ce schéma est A-stable.
4. Tester empiriquement l'ordre de convergence de ce schéma sur le problème de Cauchy $y'(t) = t(1-t^2) - y(t)$ et $y(0) = 1$ pour $t \in [0, 2]$.

$$\begin{cases} u_0 = y_0 \\ k_0 = \varphi\Big(t_n, u_n + \frac{h}{4}(k_0 - k_1)\Big) \\ k_1 = \varphi\Big(t_n + \frac{2}{3}h, u_n + \frac{h}{12}(3k_0 + 5k_1)\Big) \\ u_{n+1} = u_n + \frac{h}{4}(k_0 + 3k_1) \end{cases}$$

Il s'agit d'un schéma à $s = 2$ étages implicite.

D'après la barrière de Butcher, un schéma RK à $s = 2$ étages implicite ne peut pas être d'ordre supérieur à $s = 4$. On peut bien-sûr écrire les formules à la main, mais on peut aussi utiliser sympy comme suit :

```
s = 2
b = [ sp.Rational(1,4), sp.Rational(3,4) ]
c = [ 0, sp.Rational(2,3) ]
A = [ [ sp.Rational(1,4), -sp.Rational(1,4)] , [ sp.Rational(1,4),  sp.Rational(5,12) ] ]

Eqs = ordre_RK(s=s, type="implicite", A=A, b=b, c=c)
```

Matrice de Butcher

$$\begin{bmatrix} 0 & \frac{1}{4} & -\frac{1}{4} \\ \frac{2}{3} & \frac{1}{4} & \frac{5}{12} \\ & \frac{1}{4} & \frac{3}{4} \end{bmatrix}$$

On a 3 conditions pour avoir consistance (= pour être d'ordre 1)

True

True

True

On doit ajouter 1 condition pour être d'ordre 2

True

On doit ajouter 2 conditions pour être d'ordre 3

True

True

On doit ajouter 4 conditions pour être d'ordre 4

False

False

False

False

On ne peut pas avoir d'ordre 5 avec ce schéma

En conclusion, le schéma est d'ordre 3.

On applique le schéma au problème de Cauchy $y'(t) = -\beta y(t)$ et $y(0) = 1$ avec $\beta > 0$ dont la solution est $y(t) = e^{-\beta t}$. On obtient alors la relation de récurrence suivante :

```
u_n = sp.symbols(r'u_n')
u_np1 = sp.symbols(r'u_{n+1}')

dt = sp.symbols(r'h', positive=True)
x = sp.symbols(r'x', positive=True)

k_0 = sp.symbols(r'k_0')
k_1 = sp.symbols(r'k_1')
kk = [k_0, k_1]

beta = sp.symbols(r'\beta', positive=True)
phi = lambda y: -beta*y

# Eqs = [ sp.Eq( k_0, phi(u_n+dt/4*(k_0-k_1))   ) ,
#         sp.Eq( k_1, phi(u_n+dt/4*(3*k_0+k_1)) ) ]

Eqs = [ sp.Eq( kk[i] ,  phi(u_n + dt*sum([kk[j]*A[i][j] for j in range(s)])) ) for i in range(s) ]
for eq in Eqs:
    display(eq)

sol = sp.solve( Eqs, kk ) # c'est un dictionnaire
for key,val in sol.items():
    display(sp.Eq(key, val))

KK = [sol[kk[i]] for i in range(s)]
schema = sp.Eq( u_np1 , (u_n + dt*sum([KK[j]*b[j] for j in range(s)])).factor() )
display(schema)
```

$$k_0 = -\beta\left(h\left(\frac{k_0}{4} - \frac{k_1}{4}\right) + u_n\right)$$

$$k_1 = -\beta\left(h\left(\frac{k_0}{4} + \frac{5k_1}{12}\right) + u_n\right)$$

$$k_0 = \frac{-4\beta^2 h u_n - 6\beta u_n}{\beta^2 h^2 + 4\beta h + 6}$$

$$k_1 = -\frac{6\beta u_n}{\beta^2 h^2 + 4\beta h + 6}$$

$$u_{n+1} = -\frac{2u_n\left(\beta h - 3\right)}{\beta^2 h^2 + 4\beta h + 6}$$

On peut expliciter u_n en fonction de u_0 et du produit βh (on notera x le produit βh) :

```
q = (schema.rhs/u_n).simplify().subs(beta*dt,x)
display(Markdown(f"$$q(x) = {sp.latex(q)}$$"))
```

$$q(x) = \frac{6 - 2x}{x^2 + 4x + 6}$$

```
# sinon directement avec la fonction calculer_q_rk_general
b = sp.Matrix(b)
A = sp.Matrix(A)
q = calculer_q_rk_general(s=s,A=A,b=b)
display(Markdown(f"$$q(x) = {sp.latex(q)}$$"))
```

$$q(x) = \frac{2 \cdot (3 - x)}{x^2 + 4x + 6}$$

La suite vérifie $u_n \xrightarrow[n\to\infty]{} 0$ ssi $|q(x)| < 1$. En étudiant brièvement la fonction $q(x)$, on trouve que $-1 < q(x) < 1$ pour tout $x > 0$:

```
display(Markdown(f"Est-ce que $|q(x)|<1$ pour tout $x>0$ ? {sp.solve(abs(q)<1)}"))
```

Est-ce que $|q(x)| < 1$ pour tout $x > 0$? True

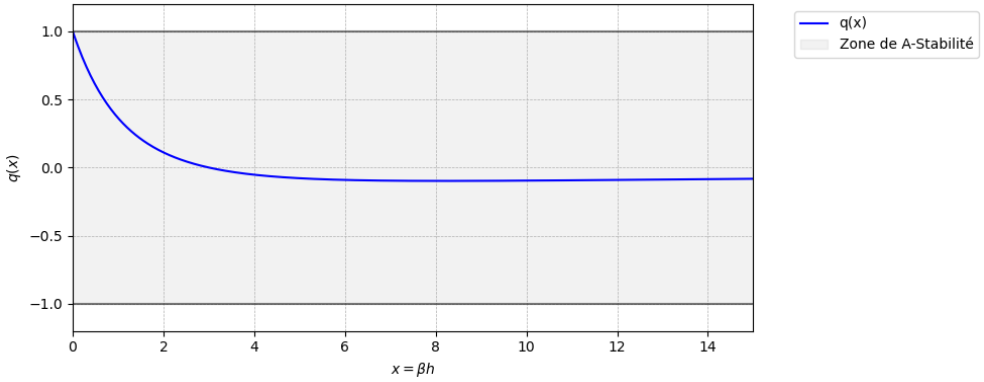
```
q_np = sp.lambdify(x, q, 'numpy')

x_max = 15
xx = np.linspace(0, x_max, 500)
plt.figure(figsize=(10, 4))
plt.plot(xx, q_np(xx), label='q(x)', color='blue')

# Tracé des limites de stabilité (|q(x)| = 1)
plt.axhline(y=1, color='black', linestyle='-', alpha=0.5)
plt.axhline(y=-1, color='black', linestyle='-', alpha=0.5)
plt.fill_between(xx, -1, 1, color='gray', alpha=0.1, label='Zone de A-Stabilité')

# Configuration des axes
plt.xlabel(r'$x = \beta h$')
plt.ylabel(r'$q(x)$')
plt.ylim(-1.2, 1.2) # On zoome sur la zone intéressante [-1, 1]
plt.xlim(0, x_max)
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left');


plt.tight_layout()
plt.show()
```




```
# Justifions ce calcul de sympy par une rapide étude de q(x)
# ============================================================
# q(0)
display(Markdown(f"$q(0) = {q.subs(x,0)}$"))
# q(x) -> 0^- si x -> +\infty
display(Markdown( r"$\lim_{x\to +\infty} q(x) = " + sp.latex(sp.limit(q,x,sp.oo)) + "$"))
# q'(x)
q_prime = sp.diff(q,x).factor()
display(Markdown(f"$q'(x) = {sp.latex(q_prime)}$"))
# Elle admer un seul minimum en
# q est décroissante sur [0,x_min] et croissante sur [x_min,+\infty]
x_min = sp.solve(q_prime,x)
display(Markdown(f"Le minimum de q(x) est atteint en $x = {sp.latex(x_min[0])}$"))
display(Markdown(f"$q'(x)>0$ ssi ${sp.latex(sp.solve(q_prime>0))}$"))
display(Markdown(f"$q'(x)<0$ ssi ${sp.latex(sp.solve(q_prime<0))}$"))
display(Markdown(f"Valeur du minimum : $q({sp.latex(x_min[0])}) = {sp.latex(q.subs(x,x_min[0]))}$ qui est
 ↪  >-1"))
```

$q(0) = 1$

$\lim_{x\to+\infty} q(x) = 0$

$q'(x) = \frac{2(x^2-6x-18)}{(x^2+4x+6)^2}$

Le minimum de q(x) est atteint en $x = 3 + 3\sqrt{3}$

$q'(x) > 0$ ssi $3 + 3\sqrt{3} < x$

$q'(x) < 0$ ssi $x < 3 + 3\sqrt{3}$

Valeur du minimum : $q(3 + 3\sqrt{3}) = -\frac{6\sqrt{3}}{18+12\sqrt{3}+(3+3\sqrt{3})^2}$ qui est >-1

```
#####################################################
# EDO
#####################################################
t0     = 0
tfinal = 2
y0     = 1

phi = lambda t,y: t*(1-t**2)-y

#####################################################
# solution exacte
#####################################################
t   = sp.Symbol('t')
y   = sp.Function('y')
edo = sp.Eq( (y(t)).diff() , phi(t,y(t)) )
solpar = sp.dsolve(edo , ics={y(t0):y0})
display(solpar)
```

```
sol_exacte = sp.lambdify(t,solpar.rhs,'numpy')

##################################################
# schéma
##################################################

# Paramètres du schéma recuperés depuis la matrice de Butcher obtenue précédemment
cc = np.array(c)
bb = np.array(b)
AA = np.array(A)

s = len(cc)
# Schéma de Runge-Kutta implicite à s étages
def RK(phi, tt, y0):
    h  = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    uu[0] = y0
    for n in range(len(tt) - 1):
        k_init = phi(tt[n], uu[n])
        eqs = lambda KK: [ -KK[0] + phi(tt[n]        , uu[n] + h/4*( KK[0] - KK[1]) ) ,
                           -KK[1] + phi(tt[n] + 2/3*h, uu[n] + h/12*( 3*KK[0] + 5*KK[1]) )
                           ]
        KK = fsolve( eqs, [k_init,k_init] )
        uu[n+1] = uu[n] + h/4*(KK[0]+ 3*KK[1])
        # eqs = lambda KK: [ -KK[i] + phi(tt[n] + cc[i]*h, uu[n] + h*sum([AA[i,j]*KK[j] for j in
        ↪  range(s)])) for i in range(s)]
        # KK = fsolve( eqs , k_init*np.ones(s) )
        # uu[n+1] = uu[n] + h*sum([KK[j]*b[j] for j in range(s)])
    return uu


##################################################
# ordre
##################################################
H = []
err = []


plt.figure(figsize=(16,5))

ax1 = plt.subplot(1,2,1)
plt.xlabel('$t$')
plt.ylabel('$y$')
ax1.grid(True)

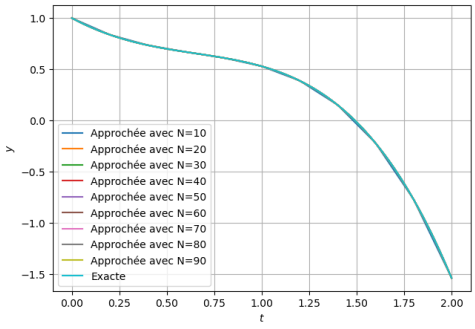
for N in range(10,100,10):
    tt = np.linspace(t0, tfinal, N + 1)
    yy = sol_exacte(tt)
    uu = RK(phi, tt, y0)
    H.append( tt[1] - tt[0] )
    err.append( np.linalg.norm(uu-yy,np.inf) )
    ax1.plot(tt,uu,label=f'Approchée avec N={N}')
    if err[-1]<1.e-10:
        break

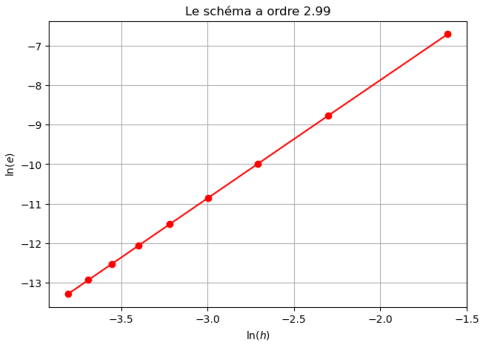
ax1.plot(tt,yy,label='Exacte') # sur la grille la plus fine
ax1.legend()

ax2 = plt.subplot(1,2,2)
ax2.plot( np.log(H), np.log(err), 'r-o')
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
```

```
plt.title(f'Le schéma a ordre {np.polyfit(np.log(H),np.log(err),1)[0]:1.2f}')
ax2.grid(True)
```

$y(t) = -t^3 + 3t^2 - 5t + 5 - 4e^{-t}$





19.2.3. Exercice : schéma de Radau IIA ($s = 2$, implicite)

On considère la méthode RK donnée par la matri suivante :

$$\begin{array}{c|cc} \frac{1}{3} & \frac{5}{12} & -\frac{1}{12} \\ 1 & \frac{3}{4} & \frac{1}{4} \\ \hline & \frac{3}{4} & \frac{1}{4} \end{array}$$

1. Écrire le schéma correspondant.
2. Déterminer l'ordre de convergence.
3. Déterminer sous quelle condition sur h ce schéma est A-stable.
4. Tester empiriquement l'ordre de convergence de ce schéma sur le problème de Cauchy $y'(t) = t(1-t^2) - y(t)$ et $y(0) = 1$ pour $t \in [0, 2]$.

$$\begin{cases} u_0 = y_0 \\ k_0 = \varphi\left(t_n + \frac{1}{3}h, u_n + \frac{h}{12}(5k_0 - k_1)\right) \\ k_1 = \varphi\left(t_{n+1}, u_n + \frac{h}{4}(3k_0 + k_1)\right) \\ u_{n+1} = u_n + \frac{h}{4}(3k_0 + k_1) \end{cases}$$

Il s'agit d'un schéma à $s = 2$ étages implicite.

D'après la barrière de Butcher, un schéma RK à $s = 2$ étages implicite ne peut pas être d'ordre supérieur à $s = 4$. On peut bien-sûr écrire les formules à la main, mais on peut aussi utiliser sympy comme suit :

```
s = 2

b = [ sp.Rational(3,4), sp.Rational(1,4) ]
c = [ sp.Rational(1,3) , 1 ]
A = [ [ sp.Rational(5,12), -sp.Rational(1,12)] , [ sp.Rational(3,4),  sp.Rational(1,4) ] ]

Eqs = ordre_RK(s=s, type="implicite", A=A, b=b, c=c)
```

Matrice de Butcher

$$\begin{bmatrix} \frac{1}{3} & \frac{5}{12} & -\frac{1}{12} \\ 1 & \frac{3}{4} & \frac{1}{4} \\ & \frac{3}{4} & \frac{1}{4} \end{bmatrix}$$

On a 3 conditions pour avoir consistance (= pour être d'ordre 1)

True

True

True

On doit ajouter 1 condition pour être d'ordre 2

True

On doit ajouter 2 conditions pour être d'ordre 3

True

True

On doit ajouter 4 conditions pour être d'ordre 4

False

False

False

False

On ne peut pas avoir d'ordre 5 avec ce schéma

En conclusion, le schéma est d'ordre 3.

On applique le schéma au problème de Cauchy $y'(t) = -\beta y(t)$ et $y(0) = 1$ avec $\beta > 0$ dont la solution est $y(t) = e^{-\beta t}$. On obtient alors la relation de récurrence suivante :

```
u_n = sp.symbols(r'u_n')
u_np1 = sp.symbols(r'u_{n+1}')

dt = sp.symbols(r'h', positive=True)
x = sp.symbols(r'x', positive=True)

k_0 = sp.symbols(r'k_0')
k_1 = sp.symbols(r'k_1')
kk = [k_0, k_1]

beta = sp.symbols(r'\beta', positive=True)
phi = lambda y: -beta*y

Eqs = [ sp.Eq( kk[i] ,  phi(u_n + dt*sum([kk[j]*A[i][j] for j in range(s)])) ) for i in range(s) ]
for eq in Eqs:
    display(eq)

sol = sp.solve( Eqs, kk ) # c'est un dictionnaire
KK = [sol[kk[i]] for i in range(s)]
schema = sp.Eq( u_np1 , u_n + dt*sum([KK[j]*b[j] for j in range(s)]) )
display(schema)
```

$$k_0 = -\beta\left(h\left(\frac{5k_0}{12} - \frac{k_1}{12}\right) + u_n\right)$$

$$k_1 = -\beta\left(h\left(\frac{3k_0}{4} + \frac{k_1}{4}\right) + u_n\right)$$

$$u_{n+1} = h\left(\frac{3\left(-2\beta^2 h u_n - 6\beta u_n\right)}{4\left(\beta^2h^2 + 4\beta h + 6\right)} + \frac{2\beta^2 h u_n - 6\beta u_n}{4\left(\beta^2h^2 + 4\beta h + 6\right)}\right) + u_n$$

```
display(Math(sp.latex(sp.Eq(sp.Symbol('u_{n+1}'),schema.rhs.simplify()))))
```

$$u_{n+1} = \frac{2u_n\left(-\beta h + 3\right)}{\beta^2 h^2 + 4\beta h + 6}$$

On peut expliciter u_n en fonction de u_0 et du produit βh (on notera x le produit βh) :

```
q = (schema.rhs/u_n).simplify().subs(beta*dt,x)
display(Math(sp.latex(sp.Eq(sp.Symbol('q(x)'),q.simplify()))))
```

$$q(x) = \frac{2 \cdot (3 - x)}{x^2 + 4x + 6}$$

```
# sinon directement avec la fonction calculer_q_rk_general
b = sp.Matrix(b)
A = sp.Matrix(A)
q = calculer_q_rk_general(s=s,A=A,b=b)
display(Markdown(f"$$q(x) = {sp.latex(q)}$$"))
```

$$q(x) = \frac{2 \cdot (3 - x)}{x^2 + 4x + 6}$$

La suite vérifie $u_n \xrightarrow[n\to\infty]{} 0$ ssi $|q(x)| < 1$. En étudiant brièvement la fonction $q(x)$, on trouve que $q(x) < 1$ pour tout $x > 0$:

```
sp.solve(abs(q)<1)
```

True

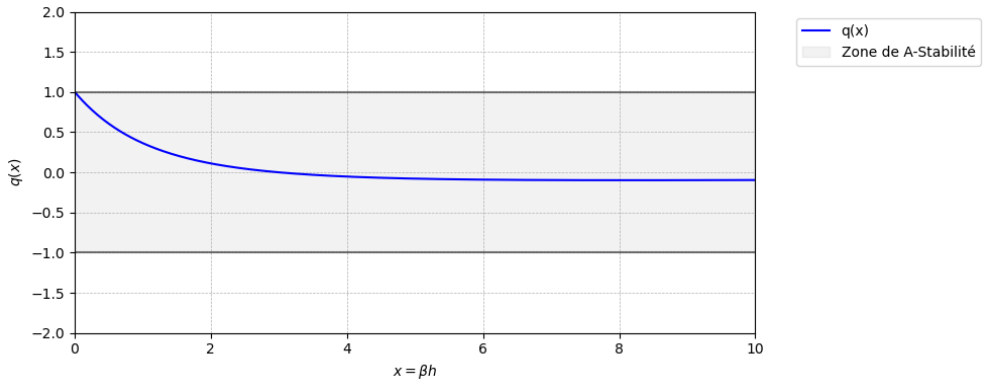
```
q_np = sp.lambdify(x, q, 'numpy')

x_max = 10
xx = np.linspace(0, x_max, 500)
plt.figure(figsize=(10, 4))
plt.plot(xx, q_np(xx), label='q(x)', color='blue')

# Tracé des limites de stabilité (|q(x)| = 1)
plt.axhline(y=1, color='black', linestyle='-', alpha=0.5)
plt.axhline(y=-1, color='black', linestyle='-', alpha=0.5)
plt.fill_between(xx, -1, 1, color='gray', alpha=0.1, label='Zone de A-Stabilité')

# Configuration des axes
plt.xlabel(r'$x = \beta h$')
plt.ylabel(r'$q(x)$')
plt.ylim(-2, 2) # On zoome sur la zone intéressante [-1, 1]
plt.xlim(0, x_max)
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left');


plt.tight_layout()
plt.show()
```




```
######################################################
# EDO
######################################################
t0     = 0
tfinal = 2
y0     = 1

phi = lambda t,y: t*(1-t**2)-y

######################################################
# solution exacte
######################################################
t   = sp.Symbol('t')
y   = sp.Function('y')
edo = sp.Eq( (y(t)).diff() , phi(t,y(t)) )
solpar = sp.dsolve(edo , ics={y(t0):y0})
display(solpar)

sol_exacte = sp.lambdify(t,solpar.rhs,'numpy')

######################################################
# schéma
######################################################

# Paramètres du schéma recuperés depuis la matrice de Butcher obtenue précédemment
cc = np.array(c)
bb = np.array(b)
AA = np.array(A)

s = len(cc)
# Schéma de Runge-Kutta implicite à s étages
def RK(phi, tt, y0):
    h  = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    uu[0] = y0
    for n in range(len(tt) - 1):
        eqs = lambda KK: [ -KK[i] + phi(tt[n] + cc[i]*h, uu[n] + h*sum([AA[i,j]*KK[j] for j in
↪   range(s)])) for i in range(s)]
        k_init = phi(tt[n], uu[n])
        KK = fsolve( eqs , k_init*np.ones(s) )
        uu[n+1] = uu[n] + h*sum([KK[j]*b[j] for j in range(s)])
    return uu


######################################################
# ordre
######################################################
H = []
err = []
```

```
plt.figure(figsize=(16,5))

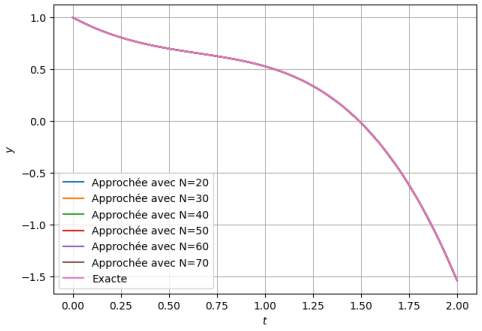
ax1 = plt.subplot(1,2,1)
plt.xlabel('$t$')
plt.ylabel('$y$')
ax1.grid(True)

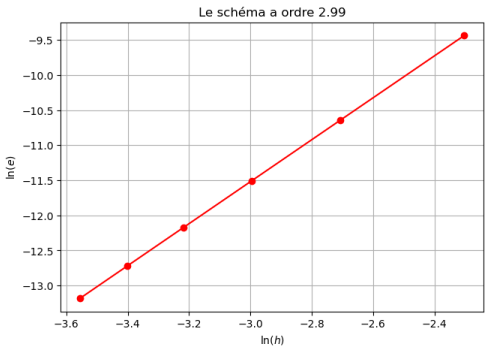
N = 10
for k in range(6):
    N += 10
    tt = np.linspace(t0, tfinal, N + 1)
    H.append( tt[1] - tt[0] )
    yy = sol_exacte(tt)
    uu = RK(phi, tt, y0)
    err.append( np.linalg.norm(uu-yy,np.inf) )
    ax1.plot(tt,uu,label=f'Approchée avec N={N}')

ax1.plot(tt,yy,label='Exacte') # sur la grille la plus fine
ax1.legend()

ax2 = plt.subplot(1,2,2)
ax2.plot( np.log(H), np.log(err), 'r-o')
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
plt.title(f'Le schéma a ordre {np.polyfit(np.log(H),np.log(err),1)[0]:1.2f}')
ax2.grid(True)
```

$y(t) = -t^3 + 3t^2 - 5t + 5 - 4e^{-t}$





19.2.4. Exercice ($s = 3$, explicite)

On considère la méthode RK donnée par la matrice suivante :

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 1 & 0 \\ \hline & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{array}$$

1. Écrire le schéma correspondant.
2. Déterminer l'ordre de convergence.
3. Déterminer sous quelle condition sur h ce schéma est A-stable.

4. Tester empiriquement l'ordre de convergence de ce schéma sur le problème de Cauchy $y'(t) = t(1-t^2) - y(t)$ et $y(0) = 1$ pour $t \in [0, 2]$.

$$\begin{cases} u_0 = y_0 \\ k_0 = \varphi\Big(t_n, u_n\Big) \\ k_1 = \varphi\Big(t_n + \frac{h}{2}, u_n + \frac{h}{2}k_0\Big) \\ k_2 = \varphi\Big(t_n + h, u_n + hk_1\Big) \\ u_{n+1} = u_n + h(\frac{1}{3}k_0 + \frac{1}{3}k_1 + \frac{1}{3}k_2) \end{cases}$$

Il s'agit d'un schéma à $s = 3$ étages explicite.

D'après la barrière de Butcher, un schéma RK à $s = 3$ étages explicite ne peut pas être d'ordre supérieur à $s = 3$. On peut bien-sûr écrire les formules à la main, mais on peut aussi utiliser un solveur pour résoudre le système d'équations non-linéaires comme suit :

```
s = 3
type = "explicite"
b = [ sp.Rational(1,3), sp.Rational(1,3), sp.Rational(1,3)]
c = [ 0, sp.Rational(1,2), 1 ]
A = [[ 0, 0, 0],
    [ sp.Rational(1,2), 0, 0],
    [ 0, 1, 0]]

Eqs = ordre_RK(s=s, type=type, A=A, b=b, c=c)
```

Matrice de Butcher

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 1 & 0 \\ & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{bmatrix}$$

On a 4 conditions pour avoir consistance (= pour être d'ordre 1)

True

True

True

True

On doit ajouter 1 condition pour être d'ordre 2

True

On doit ajouter 2 conditions pour être d'ordre 3

False

True

On ne peut pas avoir d'ordre 4 avec ce schéma

On ne peut pas avoir d'ordre 5 avec ce schéma

En conclusion, le schéma est d'ordre 2.

On applique le schéma au problème de Cauchy $y'(t) = -\beta y(t)$ et $y(0) = 1$ avec $\beta > 0$ dont la solution est $y(t) = e^{-\beta t}$. On obtient alors la relation de récurrence suivante :

```
u_n = sp.symbols(r'u_n')
u_np1 = sp.symbols(r'u_{n+1}')

dt = sp.symbols(r'h', positive=True)
x = sp.symbols(r'x', positive=True)

k_0 = sp.symbols(r'k_0')
k_1 = sp.symbols(r'k_1')
k_2 = sp.symbols(r'k_2')
kk = [k_0, k_1, k_2]

beta = sp.symbols(r'\beta', positive=True)
phi = lambda y: -beta*y

Eqs = [ sp.Eq( kk[i] ,  phi(u_n + dt*sum([kk[j]*A[i][j] for j in range(s)])) ) for i in range(s) ]
for eq in Eqs:
    display(eq)

sol = sp.solve( Eqs, kk ) # c'est un dictionnaire
KK = [sol[kk[i]] for i in range(s)]
schema = sp.Eq( u_np1 , u_n + dt*sum([KK[j]*b[j] for j in range(s)]) )
display(schema)
```

$$k_0 = -\beta u_n$$

$$k_1 = -\beta\left(\frac{hk_0}{2} + u_n\right)$$

$$k_2 = -\beta\left(hk_1 + u_n\right)$$

$$u_{n+1} = h\left(-\frac{\beta^3 h^2 u_n}{6} + \frac{\beta^2 h u_n}{2} - \beta u_n\right) + u_n$$

```
display(Math(sp.latex(sp.Eq(sp.Symbol('u_{n+1}'),schema.rhs.simplify()))))
```

$$u_{n+1} = \frac{u_n\left(-\beta h\left(\beta^2 h^2 - 3\beta h + 6\right) + 6\right)}{6}$$

On peut expliciter u_n en fonction de u_0 et du produit βh (on notera x le produit βh) :

```
q = (schema.rhs/u_n).simplify().subs(beta*dt,x)
display(Math(sp.latex(sp.Eq(sp.Symbol('q(x)'),q.simplify()))))
```

$$q(x) = -\frac{x\left(x^2 - 3x + 6\right)}{6} + 1$$

```
# sinon directement avec la fonction calculer_q_rk_general
b = sp.Matrix(b)
A = sp.Matrix(A)
q = calculer_q_rk_general(s=s,A=A,b=b)
display(Markdown(f"$$q(x) = {sp.latex(q)}$$"))
```

$$q(x) = -\frac{x^3}{6} + \frac{x^2}{2} - x + 1$$

La suite vérifie $u_n \xrightarrow[n\to\infty]{} 0$ ssi $|q(x)| < 1$. En étudiant brièvement la fonction $q(x)$, on trouve que $q(x) < 1$ ssi $x < 6$ donné ci-dessous:.

```
sol = sp.solveset(q+1, x, domain=sp.S.Reals)
display(Markdown(f"$\\beta h<{sol.args[0].evalf()}$"))
```

$\beta h < 2.51274532661833$

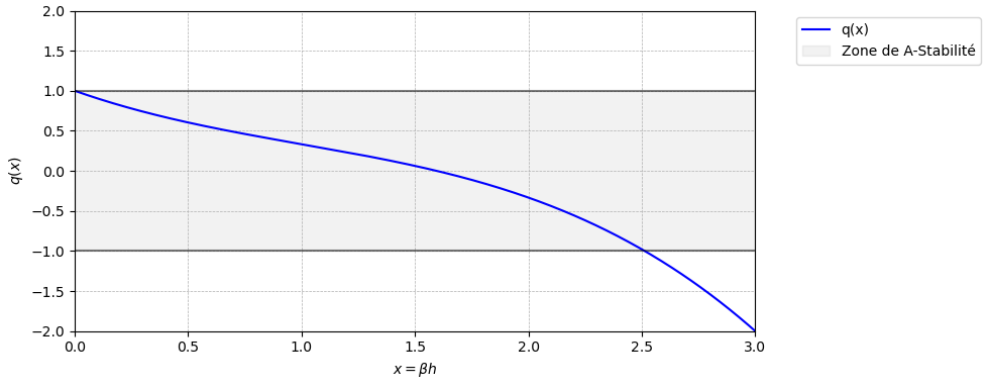
```
q_np = sp.lambdify(x, q, 'numpy')

x_max = 3
xx = np.linspace(0, x_max, 500)
plt.figure(figsize=(10, 4))
plt.plot(xx, q_np(xx), label='q(x)', color='blue')

# Tracé des limites de stabilité (|q(x)| = 1)
plt.axhline(y=1, color='black', linestyle='-', alpha=0.5)
plt.axhline(y=-1, color='black', linestyle='-', alpha=0.5)
plt.fill_between(xx, -1, 1, color='gray', alpha=0.1, label='Zone de A-Stabilité')

# Configuration des axes
plt.xlabel(r'$x = \beta h$')
plt.ylabel(r'$q(x)$')
plt.ylim(-2, 2) # On zoome sur la zone intéressante [-1, 1]
plt.xlim(0, x_max)
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left');


plt.tight_layout()
plt.show()
```




```
#####################################################
# EDO
#####################################################
t0     = 0
tfinal = 2
y0     = 1

phi = lambda t,y: t*(1-t**2)-y

#####################################################
# solution exacte
#####################################################
t   = sp.Symbol('t')
y   = sp.Function('y')
edo = sp.Eq( (y(t)).diff() , phi(t,y(t)) )
solpar = sp.dsolve(edo , ics={y(t0):y0})
display(solpar)
```

```
sol_exacte = sp.lambdify(t,solpar.rhs,'numpy')

#####################################################
# schéma
#####################################################

# Paramètres du schéma recuperés depuis la matrice de Butcher obtenue précédemment
cc = np.array(c)
bb = np.array(b)
AA = np.array(A)

s = len(cc)
# Schéma de Runge-Kutta implicite à s étages
def RK(phi, tt, y0):
    h  = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    uu[0] = y0
    for n in range(len(tt) - 1):
        k_0 = phi(tt[n], uu[n])
        k_1 = phi(tt[n]+cc[1]*h, uu[n]+h*AA[1,0]*k_0)
        k_2 = phi(tt[n]+cc[2]*h, uu[n]+h*(AA[2,0]*k_0+AA[2,1]*k_1))
        KK = [k_0, k_1, k_2]
        uu[n+1] = uu[n] + h*sum([KK[j]*b[j] for j in range(s)])
    return uu


#####################################################
# ordre
#####################################################
H = []
err = []


plt.figure(figsize=(16,5))

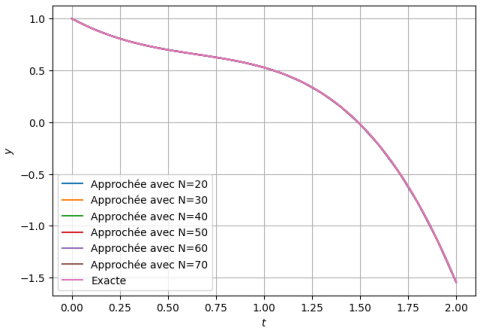
ax1 = plt.subplot(1,2,1)
plt.xlabel('$t$')
plt.ylabel('$y$')
ax1.grid(True)

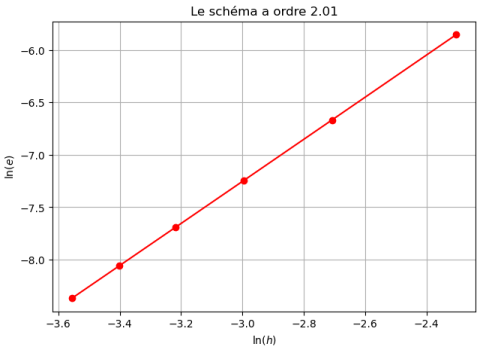
N = 10
for k in range(6):
    N += 10
    tt = np.linspace(t0, tfinal, N + 1)
    H.append( tt[1] - tt[0] )
    yy = sol_exacte(tt)
    uu = RK(phi, tt, y0)
    err.append( np.linalg.norm(uu-yy,np.inf) )
    ax1.plot(tt,uu,label=f'Approchée avec N={N}')

ax1.plot(tt,yy,label='Exacte') # sur la grille la plus fine
ax1.legend()

ax2 = plt.subplot(1,2,2)
ax2.plot( np.log(H), np.log(err), 'r-o')
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
plt.title(f'Le schéma a ordre {np.polyfit(np.log(H),np.log(err),1)[0]:1.2f}')
ax2.grid(True)
```

$y(t) = -t^3 + 3t^2 - 5t + 5 - 4e^{-t}$





19.2.5. Exercice ($s = 3$, implicite)

On considère la méthode RK donnée par la matrice suivante :

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{5}{24} & \frac{1}{3} & -\frac{1}{24} \\ 1 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{array}$$

1. Écrire le schéma correspondant.
2. Déterminer l'ordre de convergence.
3. Déterminer sous quelle condition sur h ce schéma est A-stable.
4. Tester empiriquement l'ordre de convergence de ce schéma sur le problème de Cauchy $y'(t) = -y(t)$ et $y(0) = 1$ pour $t \in [0, 10]$.

$$\begin{cases} u_0 = y_0 \\ k_0 = \varphi\Big(t_n, u_n\Big) \\ k_1 = \varphi\Big(t_n + \frac{h}{2}, u_n + h(\frac{5}{24}k_0 + \frac{1}{3}k_1 - \frac{1}{24}k_2)\Big) \\ k_2 = \varphi\Big(t_n + h, u_n + hk_1\Big) \\ u_{n+1} = u_n + h\left(\frac{1}{6}k_0 + \frac{2}{3}k_1 + \frac{1}{6}k_2\right) \end{cases}$$

Il s'agit d'un schéma à $s = 3$ étages implicite.

D'après la barrière de Butcher, un schéma RK à $s = 3$ étages implicite ne peut pas être d'ordre supérieur à $s = 6$. On peut bien-sûr écrire les formules à la main, mais on peut aussi utiliser un solveur pour résoudre le système d'équations non-linéaires comme suit :

```
s = 3
type = "implicite"

b = [ sp.Rational(1,6), sp.Rational(4,6), sp.Rational(1,6)]
c = [ 0, sp.Rational(1,2), 1 ]
A = [[ 0, 0, 0],
    [ sp.Rational(5,24), sp.Rational(1,3), -sp.Rational(1,24)],
    [ 0, 1, 0]]

Eqs = ordre_RK(s=s, type=type, A=A, b=b, c=c)
```

Matrice de Butcher

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{5}{24} & \frac{1}{3} & -\frac{1}{24} \\ 1 & 0 & 1 & 0 \\ & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{bmatrix}$$

On a 4 conditions pour avoir consistance (= pour être d'ordre 1)

True

True

True

True

On doit ajouter 1 condition pour être d'ordre 2

True

On doit ajouter 2 conditions pour être d'ordre 3

True

True

On doit ajouter 4 conditions pour être d'ordre 4

True

True

False

False

On doit ajouter 6 conditions pour être d'ordre 5

False

False

False

False

False

False

False

False

False

En conclusion, le schéma est d'ordre 3.

On applique le schéma au problème de Cauchy $y'(t) = -\beta y(t)$ et $y(0) = 1$ avec $\beta > 0$ dont la solution est $y(t) = e^{-\beta t}$. On obtient alors la relation de récurrence suivante :

```
u_n = sp.symbols(r'u_n')
u_np1 = sp.symbols(r'u_{n+1}')

dt = sp.symbols(r'h', positive=True)
x = sp.symbols(r'x', positive=True)

k_0 = sp.symbols(r'k_0')
k_1 = sp.symbols(r'k_1')
k_2 = sp.symbols(r'k_2')
kk = [k_0, k_1, k_2]

beta = sp.symbols(r'\beta', positive=True)
phi = lambda y: -beta*y
```

```
Eqs = [ sp.Eq( kk[i] ,  phi(u_n + dt*sum([kk[j]*A[i][j] for j in range(s)])) ) for i in range(s) ]
for eq in Eqs:
    display(eq)

sol = sp.solve( Eqs, kk ) # c'est un dictionnaire
KK = [sol[kk[i]] for i in range(s)]
schema = sp.Eq( u_np1 , u_n + dt*sum([KK[j]*b[j] for j in range(s)]) )
display(schema)
```

$$k_0 = -\beta u_n$$

$$k_1 = -\beta\left(h\left(\frac{5k_0}{24} + \frac{k_1}{3} - \frac{k_2}{24}\right) + u_n\right)$$

$$k_2 = -\beta\left(hk_1 + u_n\right)$$

$$u_{n+1} = h\left(-\frac{\beta u_n}{6} + \frac{2\cdot\left(4\beta^2 h u_n - 24\beta u_n\right)}{3\left(\beta^2h^2 + 8\beta h + 24\right)} + \frac{-5\beta^3h^2u_n + 16\beta^2hu_n - 24\beta u_n}{6\left(\beta^2h^2 + 8\beta h + 24\right)}\right) + u_n$$

```
display(Math(sp.latex(sp.Eq(sp.Symbol('u_{n+1}'),schema.rhs.simplify()))))
```

$$u_{n+1} = \frac{u_n\left(-\beta^3h^3 + 5\beta^2h^2 - 16\beta h + 24\right)}{\beta^2h^2 + 8\beta h + 24}$$

On peut expliciter u_n en fonction de u_0 et du produit βh (on notera x le produit βh) :

```
q = (schema.rhs/u_n).simplify().subs(beta*dt,x)
display(Math(sp.latex(sp.Eq(sp.Symbol('q(x)'),q.simplify()))))
```

$$q(x) = \frac{-x^3 + 5x^2 - 16x + 24}{x^2 + 8x + 24}$$

```
# sinon directement avec la fonction calculer_q_rk_general
b = sp.Matrix(b)
A = sp.Matrix(A)
q = calculer_q_rk_general(s=s,A=A,b=b)
display(Markdown(f"$$q(x) = {sp.latex(q)}$$"))
```

$$q(x) = \frac{-x^3 + 5x^2 - 16x + 24}{x^2 + 8x + 24}$$

La suite vérifie $u_n \xrightarrow[n\to\infty]{} 0$ ssi $|q(x)| < 1$. En étudiant brièvement la fonction $q(x)$, on trouve que $q(x) < 1$ ssi $x < 6$ donné ci-dessous:.

```
sol = sp.solveset(q+1, x, domain=sp.S.Reals)
display(Markdown(f"$\\beta h<{sol.args[0]}$"))
```

$\beta h < 6$

```
q_np = sp.lambdify(x, q, 'numpy')

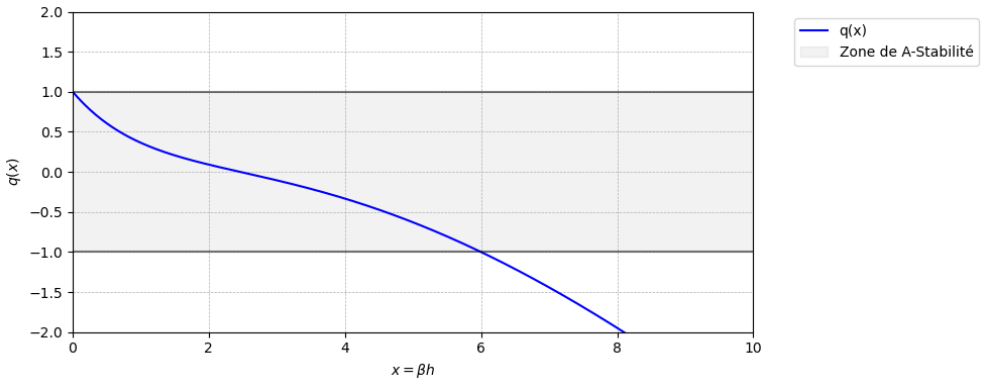
x_max = 10
xx = np.linspace(0, x_max, 500)
plt.figure(figsize=(10, 4))
plt.plot(xx, q_np(xx), label='q(x)', color='blue')

# Tracé des limites de stabilité (|q(x)| = 1)
```

```
plt.axhline(y=1, color='black', linestyle='-', alpha=0.5)
plt.axhline(y=-1, color='black', linestyle='-', alpha=0.5)
plt.fill_between(xx, -1, 1, color='gray', alpha=0.1, label='Zone de A-Stabilité')

# Configuration des axes
plt.xlabel(r'$x = \beta h$')
plt.ylabel(r'$q(x)$')
plt.ylim(-2, 2) # On zoome sur la zone intéressante [-1, 1]
plt.xlim(0, x_max)
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left');


plt.tight_layout()
plt.show()
```




```
#####################################################
# EDO
#####################################################
t0     = 0
tfinal = 10
y0     = 1

phi = lambda t,y: -y

#####################################################
# solution exacte
#####################################################
t   = sp.Symbol('t')
y   = sp.Function('y')
edo = sp.Eq( (y(t)).diff() , phi(t,y(t)) )
solpar = sp.dsolve(edo , ics={y(t0):y0})
display(solpar)

sol_exacte = sp.lambdify(t,solpar.rhs,'numpy')

#####################################################
# schéma
#####################################################

# Paramètres du schéma recuperés depuis la matrice de Butcher obtenue précédemment
cc = np.array(c)
bb = np.array(b)
AA = np.array(A)

s = len(cc)
# Schéma de Runge-Kutta implicite à s étages
def RK(phi, tt, y0):
```

```
    h  = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    uu[0] = y0
    for n in range(len(tt) - 1):
        k_init = phi(tt[n],uu[n])
        KK = fsolve( lambda kk: [ -kk[i] +  phi( tt[n]+cc[i]*h ,  uu[n] + h*sum([kk[j]*AA[i,j] for j in
↪   range(s)]) ) for i in range(s) ], k_init*np.ones((s,1)) )
        uu[n+1] = uu[n] + h*sum([KK[j]*b[j] for j in range(s)])
    return uu


##################################################
# ordre
##################################################
H = []
err = []


plt.figure(figsize=(16,5))

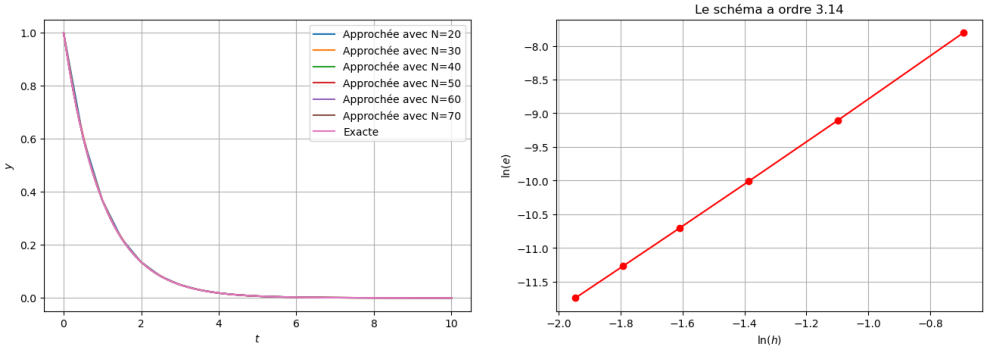
ax1 = plt.subplot(1,2,1)
plt.xlabel('$t$')
plt.ylabel('$y$')
ax1.grid(True)

N = 10
for k in range(6):
    N += 10
    tt = np.linspace(t0, tfinal, N + 1)
    H.append( tt[1] - tt[0] )
    yy = sol_exacte(tt)
    uu = RK(phi, tt, y0)
    err.append( np.linalg.norm(uu-yy,np.inf) )
    ax1.plot(tt,uu,label=f'Approchée avec N={N}')

ax1.plot(tt,yy,label='Exacte') # sur la grille la plus fine
ax1.legend()

ax2 = plt.subplot(1,2,2)
ax2.plot( np.log(H), np.log(err), 'r-o')
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
plt.title(f'Le schéma a ordre {np.polyfit(np.log(H),np.log(err),1)[0]:1.2f}')
ax2.grid(True)
```

$y(t) = e^{-t}$



19.2.6. Exercice ($s = 3$, explicite avec paramètres)

On considère la méthode RK donnée par la matrice suivante :

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & 0 \\ \frac{2}{3} & 0 & \frac{2}{3} & 0 \\ \hline & b_0 & b_1 & 0 \end{array}$$

1. Écrire le schéma correspondant.
2. Déterminer b_0 et b_1 pour que l'ordre de convergence soit maximal. Dans la suite de l'exercice on prendra ces valeurs.
3. Déterminer sous quelle condition sur h ce schéma est A-stable.
4. Tester empiriquement l'ordre de convergence de ce schéma sur le problème de Cauchy $y'(t) = y(t)(1-y(t))t$ et $y(0) = 2$ pour $t \in [0, 2]$.

$$\begin{cases} u_0 = y_0 \\ k_0 = \varphi\Big(t_n, u_n\Big) \\ k_1 = \varphi\Big(t_n + \frac{h}{3}, u_n + \frac{h}{3}k_0\Big) \\ k_2 = \varphi\Big(t_n + \frac{2}{3}h, u_n + \frac{2}{3}hk_1\Big) \\ u_{n+1} = u_n + h(b_0k_0 + b_1k_1) \end{cases}$$

Il s'agit d'un schéma à $s = 3$ étages explicite.

D'après la barrière de Butcher, un schéma RK à $s = 3$ étages explicite ne peut pas être d'ordre supérieur à $s = 3$. On peut bien-sûr écrire les formules à la main, mais on peut aussi utiliser sympy comme suit :

```
s = 3

type = "explicite"

b_0, b_1 = sp.symbols('b_0 b_1', real=True)
b = [ b_0, b_1, 0 ]
c = [ 0, sp.Rational(1,3), sp.Rational(2,3) ]
A = [[ 0, 0, 0],
    [ sp.Rational(1,3), 0, 0],
    [ 0, sp.Rational(2,3), 0]]

Eqs = ordre_RK(s=s, type=type, A=A, b=b, c=c)
```

Matrice de Butcher

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & 0 \\ \frac{2}{3} & 0 & \frac{2}{3} & 0 \\ & b_0 & b_1 & 0 \end{bmatrix}$$

On a 4 conditions pour avoir consistance (= pour être d'ordre 1)

$b_0 + b_1 = 1$

True

True

True

On doit ajouter 1 condition pour être d'ordre 2

$$\frac{b_1}{3} = \frac{1}{2}$$

On doit ajouter 2 conditions pour être d'ordre 3

$$\frac{b_1}{9} = \frac{1}{3}$$

False

On ne peut pas avoir d'ordre 4 avec ce schéma

On ne peut pas avoir d'ordre 5 avec ce schéma

En conclusion, si $b_1 = \frac{3}{2}$ et $b_0 = -\frac{1}{2}$ le schéma est d'ordre 2 :

```
b_1 = sp.Rational(3,2)
b_0 = 1-b_1
b = [ b_0, b_1, 0 ]
Eqs = ordre_RK(s=s, type=type, A=A, b=b, c=c)
```

Matrice de Butcher

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & 0 \\ \frac{2}{3} & 0 & \frac{2}{3} & 0 \\ & -\frac{1}{2} & \frac{3}{2} & 0 \end{bmatrix}$$

On a 4 conditions pour avoir consistance (= pour être d'ordre 1)

True

True

True

True

On doit ajouter 1 condition pour être d'ordre 2

True

On doit ajouter 2 conditions pour être d'ordre 3

False

False

On ne peut pas avoir d'ordre 4 avec ce schéma

On ne peut pas avoir d'ordre 5 avec ce schéma

On applique le schéma au problème de Cauchy $y'(t) = -\beta y(t)$ et $y(0) = 1$ avec $\beta > 0$ dont la solution est $y(t) = e^{-\beta t}$.
On obtient alors la relation de récurrence suivante :

```
u_n = sp.symbols(r'u_n')
u_np1 = sp.symbols(r'u_{n+1}')

dt = sp.symbols(r'h', positive=True)
x = sp.symbols(r'x', positive=True)

k_0 = sp.symbols(r'k_0')
k_1 = sp.symbols(r'k_1')
k_2 = sp.symbols(r'k_2')
kk = [k_0, k_1, k_2]

beta = sp.symbols(r'\beta', positive=True)
```

```
phi = lambda y: -beta*y

Eqs = [ sp.Eq( kk[i] ,  phi(u_n + dt*sum([kk[j]*A[i][j] for j in range(s)])) ) for i in range(s) ]
for eq in Eqs:
    display(eq)

sol = sp.solve( Eqs, kk ) # c'est un dictionnaire
KK = [sol[kk[i]] for i in range(s)]
schema = sp.Eq( u_np1 , u_n + dt*sum([KK[j]*b[j] for j in range(s)]) )
display(schema)
```

$$k_0 = -\beta u_n$$

$$k_1 = -\beta\left(\frac{hk_0}{3} + u_n\right)$$

$$k_2 = -\beta\left(\frac{2hk_1}{3} + u_n\right)$$

$$u_{n+1} = h\left(\frac{\beta^2 h u_n}{2} - \beta u_n\right) + u_n$$

```
display(Math(sp.latex(sp.Eq(sp.Symbol('u_{n+1}'),schema.rhs.simplify()))))
```

$$u_{n+1} = \frac{u_n\left(\beta h\left(\beta h - 2\right) + 2\right)}{2}$$

On peut expliciter u_n en fonction de u_0 et du produit βh (on notera x le produit βh) :

```
q = (schema.rhs/u_n).simplify().subs(beta*dt,x)
display(Math(sp.latex(sp.Eq(sp.Symbol('q(x)'),q.simplify()))))
```

$$q(x) = \frac{x\left(x - 2\right)}{2} + 1$$

La suite vérifie $u_n \xrightarrow[n\to\infty]{} 0$ ssi $|q(x)| < 1$. En étudiant brièvement la fonction $q(x)$, on trouve que $q(x) < 1$ ssi $x < 6$ donné ci-dessous:.

```
sp.solve(abs(q)<1)
```

$x < 2$

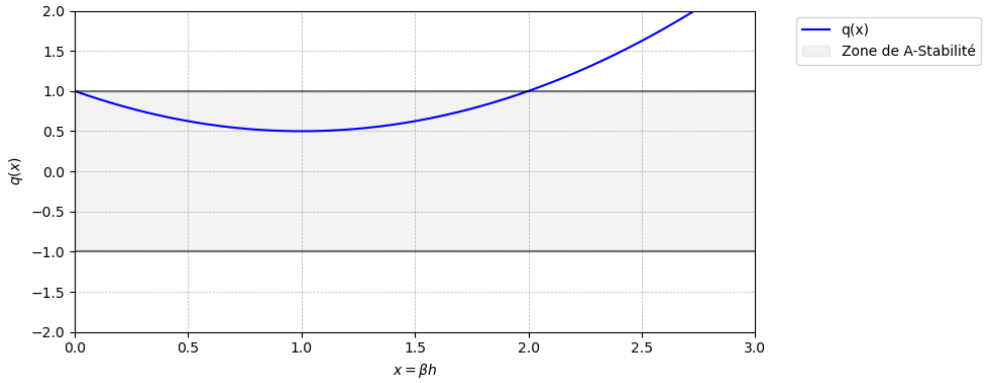
```
q_np = sp.lambdify(x, q, 'numpy')

x_max = 3
xx = np.linspace(0, x_max, 500)
plt.figure(figsize=(10, 4))
plt.plot(xx, q_np(xx), label='q(x)', color='blue')

# Tracé des limites de stabilité (|q(x)| = 1)
plt.axhline(y=1, color='black', linestyle='-', alpha=0.5)
plt.axhline(y=-1, color='black', linestyle='-', alpha=0.5)
plt.fill_between(xx, -1, 1, color='gray', alpha=0.1, label='Zone de A-Stabilité')

# Configuration des axes
plt.xlabel(r'$x = \beta h$')
plt.ylabel(r'$q(x)$')
plt.ylim(-2, 2) # On zoome sur la zone intéressante [-1, 1]
plt.xlim(0, x_max)
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left');
```

```
plt.tight_layout()
plt.show()
```




```
#####################################################
# EDO
#####################################################
t0     = 0
tfinal = 2
y0     = 2

phi = lambda t,y: y*(1-y)*t

#####################################################
# solution exacte
#####################################################
t   = sp.Symbol('t')
y   = sp.Function('y')
edo = sp.Eq( (y(t)).diff() , phi(t,y(t)) )
solpar = sp.dsolve(edo , ics={y(t0):y0})
display(solpar)

sol_exacte = sp.lambdify(t,solpar.rhs,'numpy')

#####################################################
# schéma
#####################################################

# Paramètres du schéma recuperés depuis la matrice de Butcher obtenue précédemment
cc = np.array(c)
bb = np.array(b)
AA = np.array(A)

s = len(cc)
# Schéma de Runge-Kutta implicite à s étages
def RK(phi, tt, y0):
    h  = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    uu[0] = y0
    for n in range(len(tt) - 1):
        k_0 = phi(tt[n], uu[n])
        k_1 = phi(tt[n]+cc[1]*h, uu[n]+h*AA[1,0]*k_0)
        k_2 = phi(tt[n]+cc[2]*h, uu[n]+h*(AA[2,0]*k_0+AA[2,1]*k_1))
        KK = [k_0, k_1, k_2]
        uu[n+1] = uu[n] + h*sum([KK[j]*b[j] for j in range(s)])
    return uu
```

```
##################################################
# ordre
##################################################
H = []
err = []


plt.figure(figsize=(16,5))

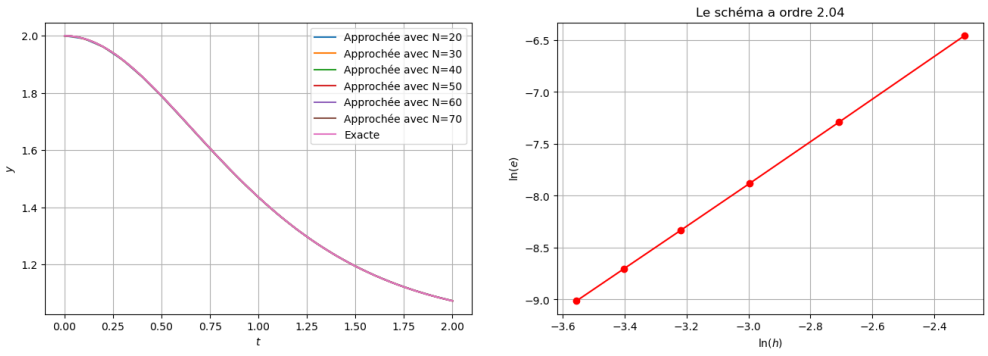
ax1 = plt.subplot(1,2,1)
plt.xlabel('$t$')
plt.ylabel('$y$')
ax1.grid(True)

N = 10
for k in range(6):
    N += 10
    tt = np.linspace(t0, tfinal, N + 1)
    H.append( tt[1] - tt[0] )
    yy = sol_exacte(tt)
    uu = RK(phi, tt, y0)
    err.append( np.linalg.norm(uu-yy,np.inf) )
    ax1.plot(tt,uu,label=f'Approchée avec N={N}')

ax1.plot(tt,yy,label='Exacte') # sur la grille la plus fine
ax1.legend()

ax2 = plt.subplot(1,2,2)
ax2.plot( np.log(H), np.log(err), 'r-o')
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
plt.title(f'Le schéma a ordre {np.polyfit(np.log(H),np.log(err),1)[0]:1.2f}')
ax2.grid(True)
```

$$y(t) = \frac{-\frac{\sqrt{e^{t^2}}}{2} - e^{t^2}}{\frac{1}{4} - e^{t^2}}$$



19.3. Catalogue de quelques schémas Runge-Kutta rencontrés dans la littérature

Dans la littérature on trouve beaucoup de schémas de Runge-Kutta. Ci-dessous une petite liste de ces méthodes. Pour chaque méthode de Runge-Kutta donné à travers sa matrice de Butcher, il est indiqué le nombre d'étages s, son ordre de convergence ω comme reporté dans la littérature et si le schéma est explicite, semi-implicite ou implicite.

Cf. Numerical Methods for Ordinary Differential Equations, J. C. Butcher, John Wiley & Sons, 2004

Exercice : étudier l'ordre théorique de convergence (au plus jusqu'à 4) et vérifier empiriquement si l'ordre annoncé coïncide avec l'ordre observé sur un problème de Cauchy dont on connaît la solution exacte.

19.3.1. Schémas à $s = 1$ étage

	Matrice de Butcher	s	ω	Exp/Imp/Semi-Implicite
Gauss-1	$\begin{array}{c\|c} \frac{1}{2} & \frac{1}{2} \\ \hline & 1 \end{array}$	1	2	I
EE	$\begin{array}{c\|c} 1 & 1 \\ \hline & 1 \end{array}$	1	1	E
EI	$\begin{array}{c\|c} 0 & 0 \\ \hline & 1 \end{array}$	1	1	I

19.3.2. Schémas à $s = 2$ étages

	Matrice de Butcher	s	ω	Exp/Imp/Semi-Implicite
SDIRK order 3	$\begin{array}{c\|cc} \gamma & \gamma & 0 \\ 1-\gamma & 1-2\gamma & \gamma \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$ avec $\gamma = \frac{3\pm\sqrt{3}}{6}$	2	3	SI
Gauss-2	$\begin{array}{c\|cc} \frac{1}{2}-\frac{\sqrt{3}}{6} & \frac{1}{4} & \frac{1}{4}-\frac{\sqrt{3}}{6} \\ \frac{1}{2}+\frac{\sqrt{3}}{6} & \frac{1}{4}+\frac{\sqrt{3}}{6} & \frac{1}{4} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$	2	4	I
Lobatto IIIC-2	$\begin{array}{c\|cc} 0 & \frac{1}{2} & -\frac{1}{2} \\ 1 & \frac{1}{2} & \frac{1}{2} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$	2	2	I
DIRK-2	$\begin{array}{c\|cc} \frac{1}{3} & \frac{1}{3} & 0 \\ 1 & 1 & 0 \\ \hline & \frac{3}{4} & \frac{1}{4} \end{array}$	2	3	SI
Hammer and Hollingsworth	$\begin{array}{c\|cc} 0 & 0 & 0 \\ \frac{2}{3} & \frac{1}{3} & \frac{1}{3} \\ \hline & \frac{1}{4} & \frac{3}{4} \end{array}$	2	?	SI
Randau-I-2	$\begin{array}{c\|cc} 0 & 0 & 0 \\ \frac{2}{3} & \frac{1}{3} & \frac{1}{3} \\ \hline & \frac{1}{4} & \frac{3}{4} \end{array}$	2	3	SI
Randau-Ia-2	$\begin{array}{c\|cc} 0 & \frac{1}{4} & -\frac{1}{4} \\ \frac{2}{3} & \frac{1}{4} & \frac{5}{12} \\ \hline & \frac{1}{4} & \frac{3}{4} \end{array}$	2	3	I
Randau-II-2	$\begin{array}{c\|cc} \frac{1}{3} & \frac{1}{3} & 0 \\ 1 & 1 & 0 \\ \hline & \frac{3}{4} & \frac{1}{4} \end{array}$	2	3	SI
Randau-IIa-2	$\begin{array}{c\|cc} \frac{1}{3} & \frac{5}{12} & -\frac{1}{12} \\ 1 & \frac{3}{4} & \frac{1}{4} \\ \hline & \frac{3}{4} & \frac{1}{4} \end{array}$	2	3	I
CN (=Lobatto IIIa)	$\begin{array}{c\|cc} 0 & 0 & 0 \\ 1 & \frac{1}{2} & \frac{1}{2} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$	2	2	SI

19.3.3. Schémas à $s = 3$ étages

	Matrice de Butcher	s	ω	Exp/Imp/Semi-Implicite
Gauss-3	$\begin{array}{c\|ccc} \frac{1}{2}-\frac{\sqrt{15}}{10} & \frac{5}{36} & \frac{2}{9}-\frac{\sqrt{15}}{15} & \frac{5}{36}-\frac{\sqrt{15}}{30} \\ \frac{1}{2} & \frac{5}{36}+\frac{\sqrt{15}}{24} & \frac{2}{9} & \frac{5}{36}-\frac{\sqrt{15}}{24} \\ \frac{1}{2}+\frac{\sqrt{15}}{10} & \frac{5}{36}+\frac{\sqrt{15}}{30} & \frac{2}{9}+\frac{\sqrt{15}}{15} & \frac{5}{36} \\ \hline & \frac{5}{18} & \frac{4}{9} & \frac{5}{18} \end{array}$	3	6	I
Radau-I-3	$\begin{array}{c\|ccc} 0 & 0 & 0 & 0 \\ \frac{6-\sqrt{6}}{10} & \frac{9+\sqrt{6}}{75} & \frac{24+\sqrt{6}}{120} & \frac{168-73\sqrt{6}}{600} \\ \frac{6+\sqrt{6}}{10} & \frac{9-\sqrt{6}}{75} & \frac{168+73\sqrt{6}}{600} & \frac{24-\sqrt{6}}{120} \\ \hline & \frac{1}{9} & \frac{16+\sqrt{6}}{36} & \frac{16-\sqrt{6}}{36} \end{array}$	3	5	I
Radau-Ia-3	$\begin{array}{c\|ccc} 0 & \frac{1}{9} & \frac{-1-\sqrt{6}}{18} & \frac{-1+\sqrt{6}}{18} \\ \frac{6-\sqrt{6}}{10} & \frac{1}{9} & \frac{88+7\sqrt{6}}{360} & \frac{88-43\sqrt{6}}{360} \\ \frac{6+\sqrt{6}}{10} & \frac{1}{9} & \frac{88+43\sqrt{6}}{360} & \frac{88-7\sqrt{6}}{360} \\ \hline & \frac{1}{9} & \frac{16+\sqrt{6}}{36} & \frac{16-\sqrt{6}}{36} \end{array}$	3	5	I
Radau-II-3	$\begin{array}{c\|ccc} \frac{4-\sqrt{6}}{10} & \frac{24-\sqrt{6}}{120} & \frac{24-11\sqrt{6}}{120} & 0 \\ \frac{4+\sqrt{6}}{10} & \frac{24+11\sqrt{6}}{120} & \frac{24+\sqrt{6}}{120} & 0 \\ 1 & \frac{6-\sqrt{6}}{12} & \frac{6+6\sqrt{6}}{12} & 0 \\ \hline & \frac{16-\sqrt{6}}{36} & \frac{16+\sqrt{6}}{36} & \frac{1}{9} \end{array}$	3	5	I
Radau-IIa-3	$\begin{array}{c\|ccc} \frac{4-\sqrt{6}}{10} & \frac{88-7\sqrt{6}}{360} & \frac{296-169\sqrt{6}}{1800} & \frac{-2+3\sqrt{6}}{225} \\ \frac{4+\sqrt{6}}{10} & \frac{296+169\sqrt{6}}{1800} & \frac{88+7\sqrt{6}}{360} & \frac{-2-3\sqrt{6}}{225} \\ 1 & \frac{16-\sqrt{6}}{36} & \frac{16+\sqrt{6}}{36} & \frac{1}{9} \\ \hline & \frac{16-\sqrt{6}}{36} & \frac{16+\sqrt{6}}{36} & \frac{1}{9} \end{array}$	3	5	I
Lobatto III	$\begin{array}{c\|ccc} 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{4} & \frac{1}{4} & 0 \\ 1 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{array}$	3	4	I
Lobatto IIIa	$\begin{array}{c\|ccc} 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{5}{24} & \frac{1}{3} & -\frac{1}{24} \\ 1 & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \\ \hline & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{array}$	3	4	I
Lobatto IIIb	$\begin{array}{c\|ccc} 0 & \frac{1}{6} & -\frac{1}{6} & 0 \\ \frac{1}{2} & \frac{1}{6} & \frac{1}{3} & 0 \\ 1 & \frac{1}{6} & \frac{5}{6} & 0 \\ \hline & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{array}$	3	4	I

19.3.4. Schémas à $s = 4$ étages

	Matrice de Butcher	s	ω	Exp/Imp/Semi-Implicite
Lobatto III	$\begin{array}{c\|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{5-\sqrt{5}}{10} & \frac{5+\sqrt{5}}{60} & \frac{1}{6} & \frac{15-7\sqrt{5}}{60} & 0 \\ \frac{5+\sqrt{5}}{10} & \frac{5-\sqrt{5}}{60} & \frac{15+7\sqrt{5}}{60} & \frac{1}{6} & 0 \\ 1 & \frac{1}{6} & \frac{5-\sqrt{5}}{12} & \frac{5+\sqrt{5}}{12} & 0 \\ \hline & \frac{1}{12} & \frac{5}{12} & \frac{5}{12} & \frac{1}{12} \end{array}$	4	6	I
Lobatto IIIa	$\begin{array}{c\|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{5-\sqrt{5}}{10} & \frac{11+\sqrt{5}}{120} & \frac{25-\sqrt{5}}{120} & \frac{25-13\sqrt{5}}{120} & \frac{-1+\sqrt{5}}{120} \\ \frac{5+\sqrt{5}}{10} & \frac{11-\sqrt{5}}{120} & \frac{25+13\sqrt{5}}{120} & \frac{25+\sqrt{5}}{120} & \frac{-1-\sqrt{5}}{120} \\ 1 & \frac{1}{12} & \frac{5}{12} & \frac{5}{12} & \frac{1}{12} \\ \hline & \frac{1}{12} & \frac{5}{12} & \frac{5}{12} & \frac{1}{12} \end{array}$	4	6	I
Lobatto IIIb	$\begin{array}{c\|cccc} 0 & \frac{1}{12} & \frac{-1-\sqrt{5}}{24} & \frac{-1+\sqrt{5}}{24} & 0 \\ \frac{5-\sqrt{5}}{10} & \frac{1}{12} & \frac{25+\sqrt{5}}{120} & \frac{25-13\sqrt{5}}{120} & 0 \\ \frac{5+\sqrt{5}}{10} & \frac{1}{12} & \frac{25+13\sqrt{5}}{120} & \frac{25-\sqrt{5}}{120} & 0 \\ 1 & \frac{1}{12} & \frac{11-\sqrt{5}}{120} & \frac{11+\sqrt{5}}{120} & 0 \\ \hline & \frac{1}{12} & \frac{5}{12} & \frac{5}{12} & \frac{1}{12} \end{array}$	4	6	I
Lobatto IIIc	$\begin{array}{c\|cccc} 0 & \frac{1}{12} & \frac{-\sqrt{5}}{12} & \frac{\sqrt{5}}{12} & -\frac{1}{12} \\ \frac{5-\sqrt{5}}{10} & \frac{1}{12} & \frac{1}{4} & \frac{10-7\sqrt{5}}{60} & \frac{\sqrt{5}}{60} \\ \frac{5+\sqrt{5}}{10} & \frac{1}{12} & \frac{10+7\sqrt{5}}{60} & \frac{1}{4} & -\frac{\sqrt{5}}{60} \\ 1 & \frac{1}{12} & \frac{5}{12} & \frac{5}{12} & \frac{1}{12} \\ \hline & \frac{1}{12} & \frac{5}{12} & \frac{5}{12} & \frac{1}{12} \end{array}$	4	6	I
	$\begin{array}{c\|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{4} & \frac{1}{8} & \frac{1}{8} & 0 & 0 \\ \frac{7}{10} & -\frac{1}{100} & \frac{14}{25} & \frac{3}{20} & 0 \\ 1 & \frac{2}{7} & 0 & \frac{5}{7} & 0 \\ \hline & \frac{1}{14} & \frac{32}{81} & \frac{250}{567} & \frac{5}{54} \end{array}$	4	5	SI
RK4-1	$\begin{array}{c\|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}$	4	4	E
	$\begin{array}{c\|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{4} & \frac{1}{4} & 0 & 0 \\ \frac{1}{2} & \frac{1}{4} & \frac{1}{4} & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}$	4	4	SI

19.3.5. Schémas à $s = 5$ étages

	Matrice de Butcher						s	ω	Exp/Imp/Semi-Implicite
Merson	0	0	0	0	0	0	5	5	E
	$\frac{1}{3}$	$\frac{1}{3}$	0	0	0	0			
	$\frac{1}{3}$	$\frac{1}{6}$	$\frac{1}{6}$	0	0	0			
	$\frac{1}{2}$	$\frac{1}{8}$	0	$\frac{3}{8}$	0	0			
	1	$\frac{1}{2}$	0	$-\frac{3}{2}$	2	0			
		$\frac{1}{6}$	0	0	$\frac{2}{3}$	$\frac{1}{6}$			

19.3.6. Schémas à $s = 6$ étages

	Matrice de Butcher	s	ω	Exp/Imp/Semi-Implicite
Fehlberg	$\begin{array}{c\|cccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 \\ \frac{3}{8} & \frac{3}{32} & \frac{9}{32} & 0 & 0 & 0 & 0 \\ \frac{12}{13} & \frac{1932}{2197} & -\frac{7200}{2197} & \frac{7269}{2197} & 0 & 0 & 0 \\ 1 & \frac{439}{216} & -8 & \frac{3680}{513} & -\frac{845}{4104} & 0 & 0 \\ \frac{1}{2} & -\frac{8}{27} & 2 & -\frac{3544}{2565} & \frac{1859}{4104} & -\frac{11}{40} & 0 \\ \hline & \frac{25}{216} & 0 & \frac{1408}{2565} & \frac{2197}{4104} & -\frac{1}{5} & 0 \end{array}$	6	4	E
Butcher	$\begin{array}{c\|cccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{4} & \frac{1}{8} & \frac{1}{8} & 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & -\frac{1}{2} & 1 & 0 & 0 & 0 \\ \frac{3}{4} & \frac{3}{16} & 0 & 0 & \frac{9}{16} & 0 & 0 \\ 1 & \frac{-3}{7} & \frac{2}{7} & \frac{12}{7} & \frac{-12}{7} & \frac{8}{7} & 0 \\ \hline & \frac{7}{90} & 0 & \frac{32}{90} & \frac{12}{90} & \frac{32}{90} & \frac{7}{90} \end{array}$	6	5	E
RK5	$\begin{array}{c\|cccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{4} & \frac{1}{8} & \frac{1}{8} & 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 \\ \frac{3}{4} & \frac{3}{16} & -\frac{3}{8} & \frac{3}{8} & \frac{9}{16} & 0 & 0 \\ 1 & \frac{-3}{7} & \frac{8}{7} & \frac{6}{7} & \frac{-12}{7} & \frac{8}{7} & 0 \\ \hline & \frac{7}{90} & 0 & \frac{32}{90} & \frac{12}{90} & \frac{32}{90} & \frac{7}{90} \end{array}$	6	5	E
	$\begin{array}{c\|cccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ 1 & \frac{-9}{4} & \frac{13}{4} & 0 & 0 & 0 & 0 \\ \frac{1}{4} & \frac{9}{64} & \frac{5}{32} & \frac{-3}{64} & 0 & 0 & 0 \\ \frac{7}{10} & \frac{63}{625} & \frac{259}{2500} & \frac{231}{2500} & \frac{252}{625} & 0 & 0 \\ 1 & \frac{-27}{50} & \frac{-139}{50} & \frac{-21}{50} & \frac{56}{25} & \frac{5}{2} & 0 \\ \hline & \frac{1}{14} & 0 & 0 & \frac{32}{81} & \frac{250}{567} & \frac{5}{54} \end{array}$	6	5	E
	$\begin{array}{c\|cccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 & 0 & 0 \\ \frac{2}{3} & \frac{4}{9} & \frac{6}{9} & 0 & 0 & 0 & 0 \end{array}$			

19.3.7. Schémas à $s = 7$ étages

	Matrice de Butcher								s	ω	Exp/Imp/Semi-Implicite
Butcher	0	0	0	0	0	0	0	0	7	6	E
	$\frac{1}{2}$	$\frac{1}{2}$	0	0	0	0	0	0			
	$\frac{2}{3}$	$\frac{2}{9}$	$\frac{4}{9}$	0	0	0	0	0			
	$\frac{1}{3}$	$\frac{7}{36}$	$\frac{8}{36}$	$-\frac{3}{36}$	0	0	0	0			
	$\frac{5}{6}$	$-\frac{35}{144}$	$-\frac{220}{144}$	$\frac{105}{144}$	$\frac{270}{144}$	0	0	0			
	$\frac{1}{6}$	$-\frac{1}{360}$	$-\frac{110}{360}$	$-\frac{45}{360}$	$\frac{180}{360}$	$\frac{36}{360}$	0	0			
	1	$-\frac{41}{160}$	$\frac{22}{13}$	$\frac{43}{156}$	$-\frac{118}{39}$	$\frac{32}{195}$	$\frac{80}{39}$	0			
		$\frac{13}{200}$	0	$\frac{11}{40}$	$\frac{11}{40}$	$\frac{4}{25}$	$\frac{4}{25}$	$\frac{13}{200}$			

Chapitre 20.

TD4 - Révisions

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import fsolve
from scipy.integrate import solve_ivp, odeint
import sympy as sp
from IPython.display import display, Math, Markdown

# Configuration des paramètres d'affichage
plt.rcParams.update({'font.size': 14})
```

20.1. Exercice : Étude d'un problème mathématiquement mal posé

Considérons le problème de Cauchy

$$\begin{cases} y'(t) = y^2(t) + t^2, \quad t \in [0; 1-\varepsilon], \\ y(0) = 1. \end{cases}$$

- Soit $\varepsilon = 0.2$. Utiliser la méthode d'Euler explicite puis les méthodes d'Euler implicite, de Crank-Nicolson et la méthode RK4 pour calculer la solution approchée avec un pas $h = 0.1$ et comparer les deux résultats en $t = 1$. Même exercice avec $h = 0.05$ et $h = 0.01$.
- Répéter avec $\varepsilon = 0.1$ puis $\varepsilon = 0$.

Commenter les résultats (*hint :* tracer le champ de vecteurs).

- Source : https://www.johndcook.com/blog/2009/08/11/approximating-a-solution-that-doesnt-exist/

- Source : (page 438 et 447) Boyce and DiPrima *Elementary Differential Equations and Boundary Value Problems*, seventh edition

Il s'agit de schémas à un pas fixe. Ils ont donc tous la même structure de base et diffèrent uniquement par la "formule de mise à jour" utilisée pour calculer u_{n+1} à partir de u_n. Nous allons donc implémenter une fonction générique `solve_ode` qui prend en argument la fonction ϕ, le vecteur de temps `tt`, la condition initiale `y0` et la fonction de mise à jour `step_func` qui correspond au schéma numérique choisi (Euler explicite, Euler implicite, Crank-Nicolson ou RK4).

```
# Base unique pour tous les schémas à un pas.
def solve_ode(phi, tt, y0, method):
    uu = np.empty_like(tt)
    h = tt[1] - tt[0]
    uu[0] = y0
    for n in range(len(tt) - 1):
        uu[n+1] = method(phi, tt[n], uu[n], h)
    return uu

def step_euler_explicite(phi, t, u, h):
    return u + h * phi(t, u)

def step_euler_implicite(phi, t, u, h):
```

```
    return fsolve(lambda x: -x + u + h * phi(t + h, x), u)[0]

def step_crank_nicolson(phi, t, u, h):
    f = lambda x: -x + u + h * (phi(t, u) + phi(t + h, x)) / 2
    return fsolve(f, u)[0]

def step_rk4(phi, t, u, h):
    k1 = phi(t, u)
    k2 = phi(t + h/2, u + h*k1/2)
    k3 = phi(t + h/2, u + h*k2/2)
    k4 = phi(t + h, u + h*k3)
    return u + (h/6) * (k1 + 2*k2 + 2*k3 + k4)


# EDO
phi = lambda t,y : y**2 + t**2

# Initialisation des paramètres
t0, tfinal, y0 = 0, 1, 1

# Création d'un tableau pour stocker tous les résultats
results = []

# Création de la figure avec 3 lignes et 3 colonnes
fig, axes = plt.subplots(3, 3, figsize=(18, 18))

for row, epsilon in enumerate([0.2, 0.1, 0]):
    for col, h in enumerate([0.1, 0.05, 0.01]):
        N = int((1 - epsilon) / h)
        tt = np.linspace(0, 1 - epsilon, N + 1)  # Utilisation de linspace pour éviter les erreurs
 ↪  d'arrondi
        uu_EE = solve_ode(phi, tt, y0, step_euler_explicite)
        uu_EI = solve_ode(phi, tt, y0, step_euler_implicite)
        uu_CN = solve_ode(phi, tt, y0, step_crank_nicolson)
        uu_RK4 = solve_ode(phi, tt, y0, step_rk4)

        # Enregistrer les résultats dans le tableau
        results.append([epsilon, h, uu_EE[-1], uu_EI[-1], uu_CN[-1], uu_RK4[-1]])

        # Tracer les solutions dans le sous-graphe correspondant
        if row == 0:
            axes[row, col].set_title(f"$h={h}$", fontsize=14)
        axes[row, col].plot(tt, uu_EE, '*-', label=f'EE', color='r')
        axes[row, col].plot(tt, uu_EI, 'o-', label=f'EI', color='g')
        axes[row, col].plot(tt, uu_CN, 's-', label=f'CN', color='m')
        axes[row, col].plot(tt, uu_RK4, 'D-', label=f'RK4', color='b')
        axes[row, col].set_xlabel('$t$', fontsize=12)
        axes[row, col].set_ylabel('$y(t)$', fontsize=12)
        axes[row, col].set_xlim([0, 1])
        axes[row, col].grid(True)
        axes[row, col].legend(loc='best')

    axes[row, 0].set_ylabel(f'tfinal=$={1 - epsilon}$', fontsize=16)  # Ajouter l'étiquette de epsilon

plt.tight_layout()  # Pour un espacement optimal entre les sous-figures
plt.show()

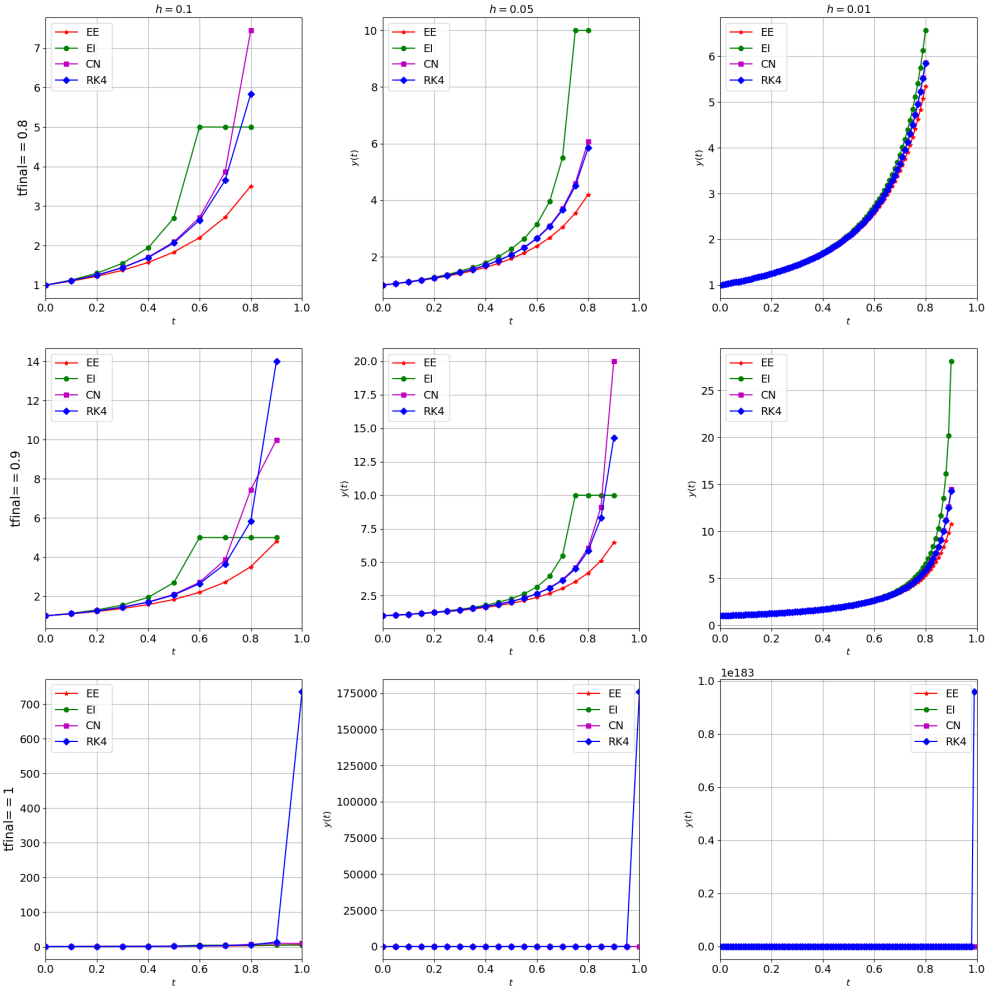
# Afficher tous les résultats dans un tableau
from tabulate import tabulate

headers = ["tfinal", "h", "EE(tfinal)", "EI(tfinal)", "CN(tfinal)", "RK4(tfinal)"]
table = []
```

```
# Remplir le tableau avec les résultats, en ajoutant un séparateur après chaque changement de tfinal
for result in results:
    table.append([1 - result[0], result[1], result[2], result[3], result[4], result[5]])

table_str = tabulate(table, headers=headers, tablefmt="html", floatfmt=".2f")
display(table_str)
```

```
/usr/lib/python3/dist-packages/scipy/optimize/_minpack_py.py:177: RuntimeWarning: The iteration is not makin
  improvement from the last five Jacobian evaluations.
  warnings.warn(msg, RuntimeWarning)
/usr/lib/python3/dist-packages/scipy/optimize/_minpack_py.py:177: RuntimeWarning: The iteration is not makin
  improvement from the last ten iterations.
  warnings.warn(msg, RuntimeWarning)
/tmp/ipykernel_1671041/1329900489.py:29: RuntimeWarning: overflow encountered in scalar power
  phi = lambda t,y : y**2 + t**2
```



tfinal	h	EE(tfinal)	EI(tfinal)	CN(tfinal)	RK4(tfinal)
0.80	0.10	3.51	5.00	7.45	5.84
0.80	0.05	4.20	10.00	6.08	5.85
0.80	0.01	5.34	6.57	5.86	5.85
0.90	0.10	4.80	5.00	10.00	14.02
0.90	0.05	6.46	10.00	20.01	14.27
0.90	0.01	10.80	28.12	14.44	14.30
1.00	0.10	7.19	5.00	10.00	735.10
1.00	0.05	12.32	10.00	20.00	175862.67

tfinal	h	EE(tfinal)	EI(tfinal)	CN(tfinal)	RK4(tfinal)
1.00	0.01	90.76	50.00	100.00	inf

Le problème est que nous avons présupposé que le problème est mathématiquement bien posé, c'est-à-dire qu'une unique solution existe sur l'intervalle d'étude $[0;1]$. Il se trouve qu'**aucune solution n'existe sur l'intervalle** $[0;1]$.

La théorie générale permet de montrer qu'une solution unique existe pour un certain intervalle contenant $t=0$, mais elle ne nous dit pas jusqu'où s'étend cet intervalle. Nous pouvons montrer qu'une solution existe pour au moins $t > \pi/4$. Cependant, **la solution devient non bornée quelque part entre** $\pi/4$ **et** 1. Nous pouvons le prouver en observant que la solution cherchée y est bornée par les deux fonctions z et w suivantes

$$z(t) \leq y(t) \leq w(t), \qquad t \in [0;1]$$

vérifiant les problèmes de Cauchy

$$\begin{cases} z'(t) = z^2(t), & t \in [0;1], \\ z(0) = 1; \end{cases} \qquad \begin{cases} w'(t) = w^2(t) + 1, & t \in [0;1], \\ w(0) = 1. \end{cases}$$

On trouve

- $z(t) = \frac{1}{1-t}$ qui est définie sur $[0;1[$,
- $w(t) = \tan\left(t + \frac{\pi}{4}\right)$ qui est définie sur $[0;\pi/4[$.

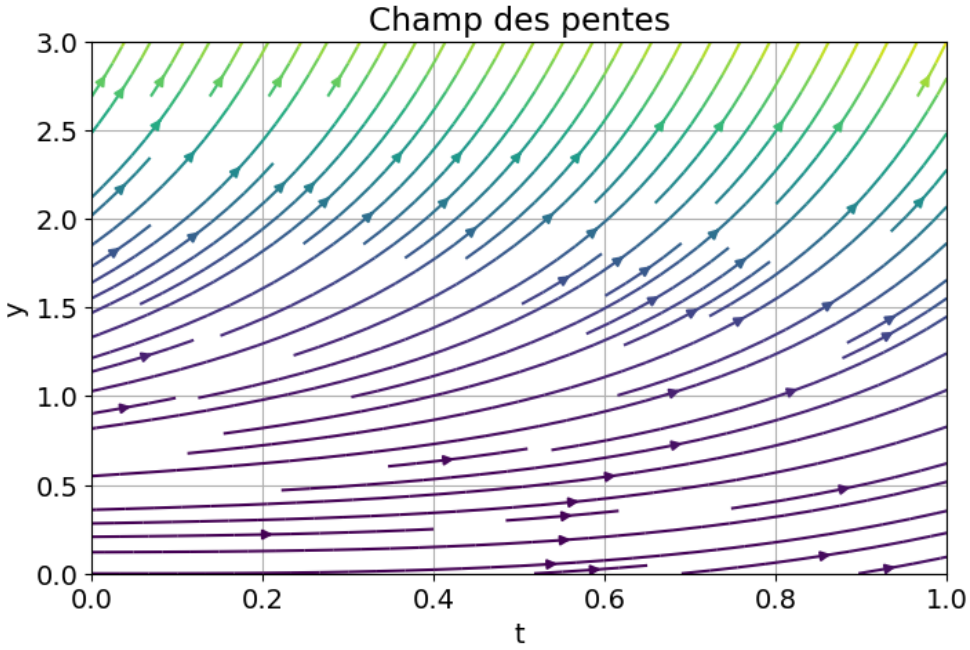
Puisque $z(t) \to +\infty$ quand $t \to 1^-$, la solution cherchée y devient non bornée quelque part entre $\pi/4$ et 1.

Même sans connaître la solution exacte, nous pouvons étudier le comportement sur l'intervalle $[0;1]$ en affichant le champ de vecteurs :

```
plt.figure(figsize=(8,5))

g1  = np.linspace(0,1,21)
g2  = np.linspace(0,3,21)
T,Y = np.meshgrid(g1,g2)          # create a grid
DT, DY = 1, phi(T,Y)              # compute growth rate on the grid
M = np.sqrt(DT**2+DY**2)          # norm growth rate
M[ M==0 ] = 1                     # avoid zero division errors
# plt.quiver(T,Y, DT/M, DY/M, M, pivot='mid')
plt.streamplot(T,Y, DT/M, DY/M, color=M)

plt.grid()
plt.xlabel('t')
plt.ylabel('y')
plt.title('Champ des pentes');
```



Cet exemple montre **l'importance de l'interaction entre la théorie et le calcul numérique.**

- Si vous vous contentez de calculer numériquement une solution approchée avec un schéma et un seul pas h sans savoir qu'une solution unique existe, vous pouvez penser que vous avez réussi alors que vous êtes en train de calculer quelque chose qui n'existe pas.
- Lorsque cela est possible, il faut essayer d'abord de prouver que le problème est mathématiquement bien posé avant d'en calculer une solution approchée. Par exemple, ce serait bien de connaître la taille de l'intervalle d'existence de la solution avant de calculer quoi que ce soit. Bien-sûr, ce n'est pas toujours possible. Dans notre exemple il n'est pas évident, en regardant l'équation, de voir qu'il y a un problème pour $t \to 1$. L'affichage du champ de vecteurs peut aider à voir ce genre de problème.
- Une bonne pratique numérique, c'est-à-dire l'essai de plus d'une taille de pas et/ou de plus d'une méthode numérique, est également précieuse. Les difficultés que nous avons rencontrées avec les comparaisons suggèrent qu'il pourrait y avoir un problème théorique sous-jacent qu'il faut clarifier.

On ne peut pas dire que la théorie ou le calcul viennent nécessairement en premier, il faut itérer entre les deux, en commençant par l'approche la plus facile à mettre en œuvre. Les résultats théoriques sont plus satisfaisants lorsqu'ils sont disponibles, mais la théorie ne nous en dit souvent pas autant que ce que nous aimerions savoir. En outre, les gens font des erreurs dans les calculs théoriques, tout comme dans les calculs numériques. Il est préférable que **les travaux théoriques et numériques se valident mutuellement**.

Ce problème montre l'importance de se préoccuper de l'existence et de l'unicité, mais les méthodes théoriques ne sont pas les seules méthodes pour explorer l'existence. Quoi qu'il en soit, le problème montre que sans une certaine diligence - théorique ou numérique - vous pourriez naïvement calculer une "solution" approximative là où il n'existe aucune solution.

20.2. Exercice : Étude d'un problème numériquement mal posé

Considérons le problème de Cauchy

$$\begin{cases} y'(t) = 3y(t) - 4e^{-t}, & t \in [0;1], \\ y(0) = 1 + \varepsilon. \end{cases}$$

1. Calculer la solution exacte.
2. Utiliser la méthode RK4 pour calculer la solution approchée avec un pas $h = \frac{1}{10}$ et comparer à la solution exacte pour cinq valeurs des ε, à savoir $\varepsilon = \pm 1$, $\varepsilon = \pm\frac{1}{10}$ et $\varepsilon = 0$.
3. Même exercice avec la méthode d'Euler explicite.
4. Commenter les résultats.

La solution exacte est $y(t) = \varepsilon e^{3t} + e^{-t}$ (ci-dessous, le calcul de la solution exacte avec sympy).

```
t = sp.Symbol('t', real=True, positive=True)
epsilon = sp.Symbol(r'\varepsilon', real=True, positive=True)
y = sp.Function('y')

edo = sp.Eq( sp.diff(y(t),t), 3*y(t)-4*sp.exp(-t) )
display(edo)

t0 = 0
y0 = 1+epsilon
display( sp.Eq(y(t).subs(t,t0), y0) )

sol = sp.dsolve(edo,y(t), ics={y(t0):y0})
display(sol.expand())
```

$$\frac{d}{dt}y(t) = 3y(t) - 4e^{-t}$$

$$y(0) = \varepsilon + 1$$

$$y(t) = \varepsilon e^{3t} + e^{-t}$$

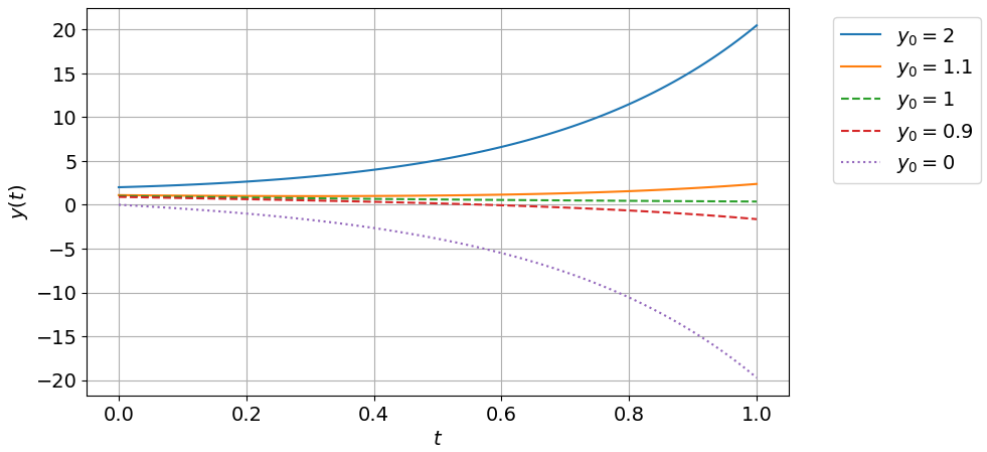
Si $\varepsilon = 0$, la solution exacte devient donc $y(t) = e^{-t}$ mais le problème est **mal conditionné** : toute (petite) erreur de calcul a le même effet qu'une perturbation de la condition initiale. En d'autres termes, on "réveil" le terme dormant e^{3t}.

Affichons l'allure de la solution exacte dans les cinq cas $\varepsilon = \pm 1$, $\varepsilon = \pm\frac{1}{10}$ et $\varepsilon = 0$. On remarque que la solution exacte est très sensible à la condition initiale.

```
func = sp.lambdify((t,epsilon),sol.rhs,'numpy')

tt = np.linspace(0,1,101)
plt.figure(figsize=(9,5))
EPSILON=[1,1.e-1,0,-1.e-1,-1]
for i,epsi in enumerate(EPSILON):
    plt.plot( tt, func(tt,epsi),
         #label=r"$\epsilon=$"+str(epsi),
         label=r'$y_0=$'+str(1+epsi),
         linestyle=['-','--',':','-.'][i//2])

plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.grid(); plt.xlabel('$t$'); plt.ylabel('$y(t)$');
```



```
phi = lambda t,y : 3*y-4*np.exp(-t)

sol_exacte = lambda t,y0 : (y0-1)*np.exp(3*t)+np.exp(-t)

def EE(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = [y0]
    for i in range(N):
        uu.append( uu[i] + h*phi(tt[i],uu[i]) )
    return uu

def RK4(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = [y0]
    for i in range(N):
        k1 = phi( tt[i], uu[i] )
        k2 = phi( tt[i]+h/2 , uu[i]+h*k1/2 )
        k3 = phi( tt[i]+h/2 , uu[i]+h*k2/2 )
        k4 = phi( tt[i+1]    , uu[i]+h*k3 )
        uu.append( uu[i] + (k1+2*k2+2*k3+k4)*h/6 )
    return uu

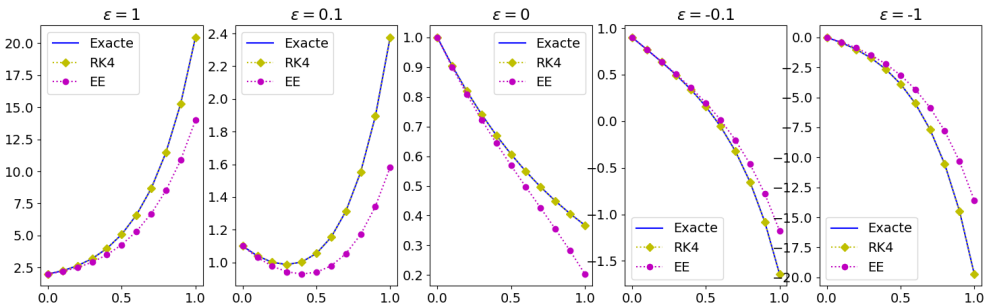
# INITIALISATION
t0     = 0
tfinal = 1

# DISCRETISATION
N  = 10
h  = (tfinal-t0)/N
tt = [ t0+i*h for i in range(N+1) ]

EPSILON=[1,1.e-1,0,-1.e-1,-1]

plt.figure(figsize=(18,5))

for i,epsilon in enumerate(EPSILON):
    y0 = 1+epsilon
    yy = [sol_exacte(t,y0) for t in tt]
    uu_RK4 = RK4(phi,tt,y0)
    uu_EE = EE(phi,tt,y0)
    plt.subplot(1,5,i+1)
    plt.plot(tt,yy,'b-',tt,uu_RK4,'y:D',tt,uu_EE,'m:o')
    plt.legend(['Exacte','RK4','EE']);
    plt.title(r"$\epsilon=$"+str(epsilon))
```



20.3. Exercice : EDO d'ordre 2

Considérons le problème de Cauchy

$$\begin{cases} y''(t) = ty(t), & t \in [0; 4.5] \\ y(0) = 0.355028053887817, \\ y'(0) = -0.258819403792807. \end{cases}$$

- Transformer l'EDO d'ordre 2 en un système de deux EDO d'ordre 1.
- Calculer la valeur approchée de y en $t = 4.5$ obtenue avec le schéma RK4 avec un pas de discrétisation $h = 0.001$.

On appelle $x(t) = y(t)$ et $z(t) = y'(t)$, alors on a $x'(t) = y'(t) = z(t)$ et $z'(t) = y''(t) = ty(t) = tx(t)$ donc

$$\begin{cases} x'(t) = z(t) \\ z'(t) = tx(t) \\ x(0) = 0.355028053887817, \\ z(0) = -0.258819403792807. \end{cases}$$

```
def RK4(phi1,phi2,tt,y0,yp0):
    uu1 = [y0]
    uu2 = [yp0]
    for i in range(len(tt)-1):
        k1_1 = phi1( tt[i]     , uu1[i]          , uu2[i] )
        k1_2 = phi2( tt[i]     , uu1[i]          , uu2[i] )
        k2_1 = phi1( tt[i]+h/2 , uu1[i]+h*k1_1/2, uu2[i]+h*k1_2/2 )
        k2_2 = phi2( tt[i]+h/2 , uu1[i]+h*k1_1/2, uu2[i]+h*k1_2/2 )
        k3_1 = phi1( tt[i]+h/2 , uu1[i]+h*k2_1/2, uu2[i]+h*k2_2/2 )
        k3_2 = phi2( tt[i]+h/2 , uu1[i]+h*k2_1/2, uu2[i]+h*k2_2/2 )
        k4_1 = phi1( tt[i+1]   , uu1[i]+h*k3_1  , uu2[i]+h*k3_2 )
        k4_2 = phi2( tt[i+1]   , uu1[i]+h*k3_1  , uu2[i]+h*k3_2 )
        uu1.append( uu1[i] + (k1_1+2.0*k2_1+2.0*k3_1+k4_1)*h/6 )
        uu2.append( uu2[i] + (k1_2+2.0*k2_2+2.0*k3_2+k4_2)*h/6 )
    return [uu1,uu2]

phi1 = lambda t,y1,y2 : y2
phi2 = lambda t,y1,y2 : t*y1

t0 = 0.0
y0 = 0.355028053887817
yp0 = -0.258819403792807
h = 0.001

tt = [t0+i*h for i in range(4501)]
[uu1,uu2] = RK4(phi1,phi2,tt,y0,yp0)
print(f"u({tt[-1]:.2f})={uu1[-1]:.16f}")
```

```
u(4.50)=0.0003302503231810
```

20.4. Exercice : A-stabilité du schéma d'Euler expicite dans le cas d'un système

Soit le problème de Cauchy :

$$\begin{cases} y''(t) + 1001y'(t) + 1000y(t) = 0, \quad \forall t \in \mathbb{R}, \\ y(0) = 1, \\ y'(0) = 0. \end{cases}$$

1. Écrire le problème comme un système de 2 EDO d'ordre 1 et en calculer la solution.
2. Écrire le schéma le schéma d'Euler explicite pour ce problème.

3. Pour quelles valeurs de h le schéma est A-stable ?

Soit $\mathbf{z}(t) = (y(t), y'(t))^T$ alors $y''(t) + 1001y'(t) + 1000y(t) = 0$ se réécrit

$$\mathbf{z}'(t) = -\begin{pmatrix} 0 & -1 \\ 1000 & 1001 \end{pmatrix}\mathbf{z}(t)$$

Cette matrice a pour valeurs propres $\lambda_1 = 1$ et $\lambda_2 = 1000$ et pour vecteurs propres associés $\mathbf{v}_1 = (-1, 1)^T$ et $\mathbf{v}_2 = (0, 1000)^T$ (ci dessous le calcul formel des valeurs et vecteurs propes). Ainsi

$$\mathbf{z}(t) = c_1\mathbf{v}_1 e^{-\lambda_1 t} + c_2\mathbf{v}_2 e^{-\lambda_2 t}$$

et

$$y(t) = -c_1 e^{-t} - c_2 e^{-1001t}.$$

Il s'agit d'un problème stiff !

```
sp.var('t')
M = sp.Matrix(((0,-1), (1000,1001)))
display(M)

p = M.charpoly(t)
display(sp.factor(p))

P, D = M.diagonalize()
display(P,D)

display(sp.simplify(P*D*P**-1))
```

$$\begin{bmatrix} 0 & -1 \\ 1000 & 1001 \end{bmatrix}$$

$\text{PurePoly}\left(t^2 - 1001t + 1000, t, domain = \mathbb{Z}\right)$

$$\begin{bmatrix} -1 & -1 \\ 1 & 1000 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 \\ 0 & 1000 \end{bmatrix}$$

$$\begin{bmatrix} 0 & -1 \\ 1000 & 1001 \end{bmatrix}$$

Soit $h > 0$ fixé et $t_n = nh$ pour tout $n \in \mathbb{Z}$. Le schéma d'Euler explicite pour le système donné construit la suite

$$\begin{aligned}
\mathbf{u}_{n+1} &= \mathbf{u}_n - h\begin{pmatrix} 0 & -1 \\ 1000 & 1001 \end{pmatrix}\mathbf{u}_n \\
&= \begin{pmatrix} 1 & h \\ -1000h & 1 - 1001h \end{pmatrix}\mathbf{u}_n \\
&= \begin{pmatrix} 1 & h \\ -1000h & 1 - 1001h \end{pmatrix}^{n+1}\mathbf{u}_0, \\
&= \begin{pmatrix} -1 & -1 \\ 1000 & 1 \end{pmatrix}\begin{pmatrix} 1 - 1000h & 0 \\ 0 & 1 - h \end{pmatrix}^{n+1}\begin{pmatrix} -1 & -1 \\ 1000 & 1 \end{pmatrix}^{-1}\mathbf{u}_0 \\
&= \begin{pmatrix} -1 & -1 \\ 1000 & 1 \end{pmatrix}\begin{pmatrix} (1 - 1000h)^{n+1} & 0 \\ 0 & (1 - h)^{n+1} \end{pmatrix}\begin{pmatrix} -1 & -1 \\ 1000 & 1 \end{pmatrix}^{-1}\mathbf{u}_0, \qquad \forall n \in \mathbb{N}.
\end{aligned}$$

```
sp.var('h')
M = sp.Matrix(((1,h), (-1000*h,1-1001*h)))
display(M)

P, D = M.diagonalize()
display(P , D)
```

$$\begin{bmatrix} 1 & h \\ -1000h & 1-1001h \end{bmatrix}$$

$$\begin{bmatrix} -1 & -1 \\ 1000 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1-1000h & 0 \\ 0 & 1-h \end{bmatrix}$$

Le schéma est A-stable si $\lim_{n\to+\infty} \mathbf{u}_{n+1} = \mathbf{0}$. Il faut donc que

$$\begin{cases} |1-1000h| < 1 \\ |1-h| < 1 \end{cases}$$

d'où la condition $h < \frac{2}{1000}$.

20.5. Exercice : Étude théorique et numérique du mouvement d'un pendule

Considérons le problème du mouvement d'un pendule de masse m suspendu au point O par un fil non pesant de longueur ℓ.
On se propose d'étudier l'évolution au cours du temps $t \geq 0$ de l'angle $\vartheta(t)$ autour du point 0.
L'angle $\vartheta(t)$ est mesuré par rapport à une verticale passante par 0.
Considérons pour simplifier seulement la force de gravité g.
La fonction ϑ est alors solution de l'EDO d'ordre 2

$$\vartheta''(t) = -\frac{g}{\ell}\sin(\vartheta(t)).$$

Si on introduit le paramètre ω tel que $\omega^2 = \frac{g}{\ell}$, l'équation devient

$$\vartheta''(t) + \omega^2 \sin(\vartheta(t)) = 0.$$

Pour simplicité on pose $\omega = 1$ et on note $q = \vartheta$ et $p = \vartheta'$.
On peut alors transformer l'EDO d'ordre 2 en un système de deux EDO d'ordre 1 :

$$\begin{cases} q'(t) = p(t), \\ p'(t) = -\sin(q(t)). \end{cases}$$

Considérons l'énergie totale du système :

$$E(q,p) = \underbrace{-\cos(q)}_{\text{Énergie potentielle}} + \underbrace{\frac{1}{2}p^2}_{\text{Énergie cinétique}}.$$

1. **Étude théorique**
   - Montre analytiquement que l'énergie totale reste constante au cours du temps, i.e. que $\frac{\mathrm{d}}{\mathrm{d}t}E(q(t),p(t)) = 0$.
   - Tracer avec Python les courbes de niveau de la fonction $(q,p) \mapsto E(q,p)$, i.e. les ensembles

$$\mathcal{N}_\kappa = \{(q,p) \in [-\pi;\pi] \times \mathbb{R} \mid E(q,p) = \kappa\} = \{(q,p) \in [-\pi;\pi] \times \mathbb{R} \mid p^2 = 2(\kappa + \cos(q))\}.$$

   - Lorsque ϑ est petit on peut considérer l'approximation $\sin(\vartheta) \simeq \vartheta$. Calculer la solution exacte de cette équation approchée.
2. **Étude numérique**
   Dans une simulation numérique on aimerait que la propriété de conservation de l'énergie soit préservée aussi bien que possible. Nous allons regarder ce qui se passe avec certaines méthodes vues en cours.
   On notera $x_n \approx q(t_n) = \vartheta(t_n)$ et $y_n \approx p(t_n) = \vartheta'(t_n)$.
   À chaque instant t_n, on calculera les valeurs approchées de l'énergie cinétique $E_c(t_n) = \frac{1}{2}y_n^2$, de l'énergie potentielle $E_p(t_n) = -\cos(x_n)$ et de l'énergie totale $E(t_n) = E_c(t_n) + E_p(t_n)$.
   Choisissons $h = t_{n+1} - t_n = 0.1$ et $(x_0, y_0) = (\pi/4, 0)$.

1. Dans un premier temps on se propose d'appliquer la méthode d'**Euler explicite** à la résolution du système.
   - Tracer (t_n, x_n) et (t_n, y_n) sur un même graphique ;
   - sur un autre graphique représenter les trois énergies simultanément en fonction du temps ;
   - tracer enfin (x_n, y_n) sur un autre graphique et vérifier que la solution numérique tourne vers l'extérieur (i.e. l'énergie augmente au cours du temps).
2. Considérons le schéma obtenu en écrivant le schéma d'**Euler explicite** pour la première équation et le schéma d'**Euler implicite** pour la seconde.
   - Montrer que ce schéma peut s'écrire sous forme parfaitement explicite ;
   - tracer (t_n, x_n) et (t_n, y_n) sur un même graphique ;
   - sur un autre graphique représenter les trois énergies simultanément en fonction du temps ;
   - tracer enfin (x_n, y_n) sur un autre graphique. Que peut-on constater ? Commenter les résultats.
3. On se propose d'appliquer les méthodes d'**Euler modifiée, de Heun, AB2, AB3 et RK4** à la résolution du système. Pour chaque méthode :
   - Tracer (t_n, x_n) et (t_n, y_n) sur un même graphique ;
   - sur un autre graphique représenter les trois énergies simultanément en fonction du temps ;
   - tracer enfin (x_n, y_n) sur un autre graphique. Que peut-on constater ? Commenter les résultats.

Il s'agit d'un système de deux EDO scalaires d'ordre 1 du type

$$\begin{cases} q'(t) = \varphi_1(t, q(t), p(t)), \\ p'(t) = \varphi_2(t, q(t), p(t)), \end{cases} \quad \text{avec} \quad \begin{cases} \varphi_1(t, q(t), p(t)) = p(t), \\ \varphi_2(t, q(t), p(t)) = -\sin(q(t)). \end{cases}$$

- L'énergie totale reste constante au cours du temps :

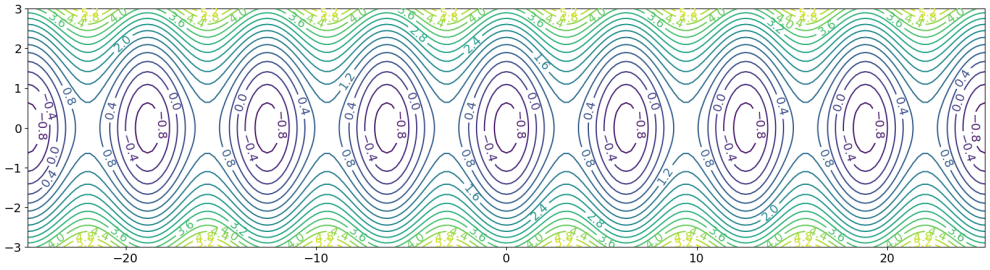
$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}t} E(q(t), p(t)) &= \sin(q(t))q'(t) + p(t)p'(t) = \sin(q(t))p(t) + p(t)p'(t) \\ &= p(t)\left(\sin(q(t)) + p'(t)\right) = p(t)\left(\sin(q(t)) - \sin(q(t))\right) = 0. \end{aligned}$$

- Courbes de niveau de la fonction $(q, p) \mapsto E(q, p)$:

```
Etot = lambda q,p : -np.cos(q)+p**2/2

q,p = np.meshgrid(np.linspace(-8*np.pi,8*np.pi,201),np.linspace(-3,3,201))
E = Etot(q,p)

plt.figure(1,figsize=(20,5))
graphe = plt.contour(q,p,E,20)
plt.clabel(graphe,inline=1);
```



- $\vartheta''(t) + \omega^2 \vartheta(t) = 0$ est une EDO d'ordre 2 homogène à coefficients constants dont la solution est $\vartheta(t) = \vartheta(0)\cos(\omega t) + \frac{\vartheta'(0)}{\omega}\sin(\omega t)$.

```
t0 = 0
tfinal = 8*np.pi
y0 = [np.pi/4,0]

N = 100
tt = np.linspace(t0,tfinal,N+1)
```

```
#h = tt[1]-tt[0]

phi1 = lambda t,q,p : p
phi2 = lambda t,q,p : -np.sin(q)
```

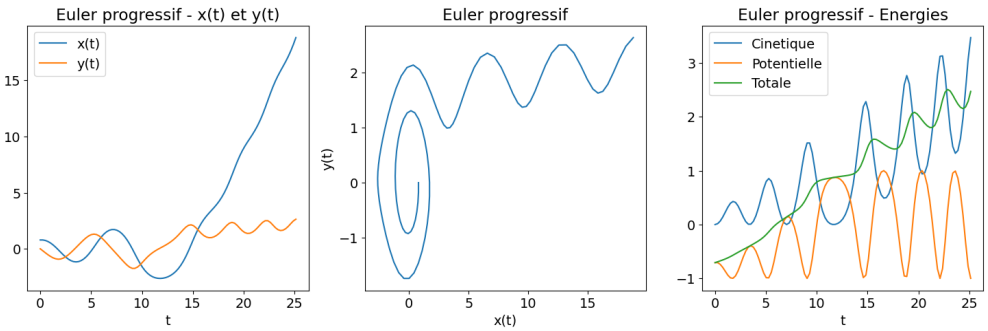
20.5.0.1. Méthode d'Euler explicite

$$\begin{cases} x_0 = q(0), \\ y_0 = p(0), \\ x_{n+1} = x_n + h\varphi_1(t_n, x_n, y_n), \\ y_{n+1} = y_n + h\varphi_2(t_n, x_n, y_n). \end{cases}$$

```
def euler_progressif(phi1,phi2,tt,y0):
    h = tt[1]-tt[0]
    q = [y0[0]]
    p = [y0[1]]
    for i in range(len(tt)-1):
        q.append(q[i]+h*phi1(tt[i],q[i],p[i]))
        p.append(p[i]+h*phi2(tt[i],q[i],p[i]))
    return [q,p]
[q_ep, p_ep] = euler_progressif(phi1,phi2,tt,y0)
Ec_ep = [p**2/2 for p in p_ep]
Ep_ep = [-np.cos(q) for q in q_ep]
Et_ep = [Ec_ep[i]+Ep_ep[i] for i in range(len(Ec_ep))]
print ('Euler explicite : |energie totale (t=Tfinal) - Energie totale (t=0)|=', abs(Et_ep[-1]-Et_ep[0]))

plt.figure(figsize=(18,5))
plt.subplot(131)
plt.plot(tt,q_ep,tt,p_ep)
plt.xlabel('t')
plt.legend(['x(t)','y(t)'])
plt.title('Euler progressif - x(t) et y(t)')
plt.subplot(132)
plt.plot(q_ep,p_ep)
plt.xlabel('x(t)')
plt.ylabel('y(t)')
plt.title('Euler progressif')
plt.subplot(133)
plt.plot(tt,Ec_ep,tt,Ep_ep,tt,Et_ep)
plt.xlabel('t')
plt.legend(['Cinetique','Potentielle','Totale'])
plt.title('Euler progressif - Energies');
```

```
Euler explicite : |energie totale (t=Tfinal) - Energie totale (t=0)|= 3.1764943624320936
```



20.5.0.2. Méthode d'Euler explicite/implicite

Schéma obtenu en écrivant le schéma d'Euler explicite pour la première équation et le schéma d'Euler implicite pour la seconde :

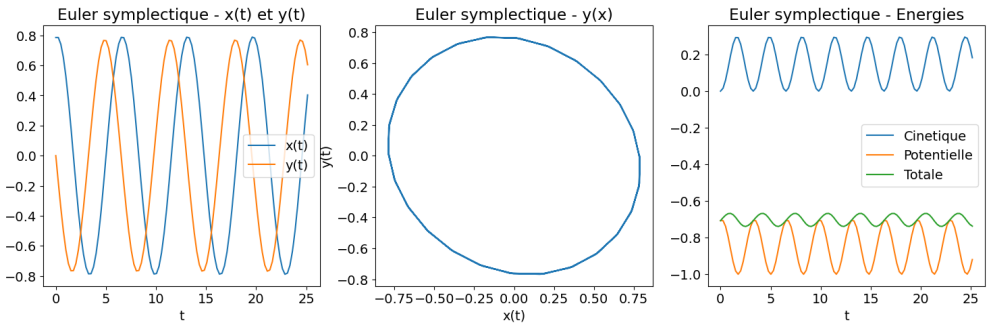
$$\begin{cases} x_0 = q(0), \\ y_0 = p(0), \\ x_{n+1} = x_n + h\varphi_1(t_n, x_n, y_n), \\ y_{n+1} = y_n + h\varphi_2(t_{n+1}, x_{n+1}, y_{n+1}). \end{cases}$$

Dans notre cas $\varphi_2(t_{n+1}, x_{n+1}, y_{n+1}) = -\sin(x_{n+1})$ donc le schéma est parfaitement explicite :

```
def euler_symplectique(phi1,phi2,tt,y0):
    h = tt[1]-tt[0]
    q = [y0[0]]
    p = [y0[1]]
    for i in range(len(tt)-1):
        q.append(q[i]+h*phi1(tt[i],q[i],p[i]))
        p.append(p[i]+h*phi2(tt[i+1],q[i+1],p[i])) # il faudrait p[i+1] (schema implicite) mais comme
 ↪  dans notre cas phi2 ne depend pas de "p", le schema est explicite
    return [q,p]
[q_es, p_es] = euler_symplectique(phi1,phi2,tt,y0)
Ec_es = [p**2/2 for p in p_es]
Ep_es = [-np.cos(q) for q in q_es]
Et_es = [Ec_es[i]+Ep_es[i] for i in range(len(Ec_es))]
print ('Euler symplectique : |energie totale (t=Tfinal)-Energie totale (t=0)|=', abs(Et_es[-1]-Et_es[0]))

plt.figure(figsize=(18,5))
plt.subplot(131)
plt.plot(tt,q_es,tt,p_es)
plt.xlabel('t')
plt.legend(['x(t)','y(t)'])
plt.title('Euler symplectique - x(t) et y(t)')
plt.subplot(132)
plt.plot(q_es,p_es)
plt.xlabel('x(t)')
plt.ylabel('y(t)')
plt.title('Euler symplectique - y(x)')
plt.subplot(133)
plt.plot(tt,Ec_es,tt,Ep_es,tt,Et_es)
plt.xlabel('t')
plt.legend(['Cinetique','Potentielle','Totale'])
plt.title('Euler symplectique - Energies');
```

```
Euler symplectique : |energie totale (t=Tfinal)-Energie totale (t=0)|= 0.030048843024607974
```



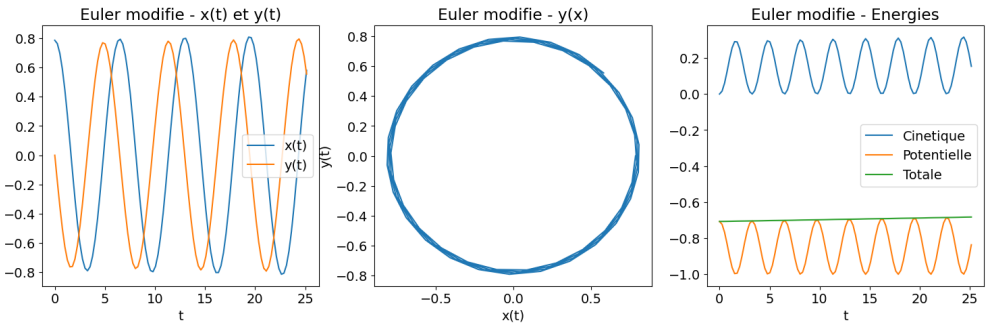
20.5.0.3. Méthodes d'Euler modifiée

$$\begin{cases} x_0 = q(0), \\ y_0 = p(0), \\ \tilde{x}_n = x_n + \frac{h}{2}\varphi_1(t_n, x_n, y_n), \\ \tilde{y}_n = y_n + \frac{h}{2}\varphi_2(t_n, x_n, y_n), \\ x_{n+1} = x_n + h\varphi_1(t_n, \tilde{x}_n, \tilde{y}_n), \\ y_{n+1} = y_n + h\varphi_2(t_n, \tilde{x}_n, \tilde{y}_n). \end{cases}$$

```
def euler_modifie(phi1,phi2,tt,y0):
    h = tt[1]-tt[0]
    q = [y0[0]]
    p = [y0[1]]
    for i in range(len(tt)-1):
        qtilde = phi1( tt[i], q[i], p[i] )
        ptilde = phi2( tt[i], q[i], p[i] )
        q.append( q[i]+h*phi1(tt[i]+h/2.,q[i]+qtilde*h/2.,p[i]+ptilde*h/2.) )
        p.append( p[i]+h*phi2(tt[i]+h/2.,q[i]+qtilde*h/2.,p[i]+ptilde*h/2.) )
    return [q,p]
[q_em, p_em] = euler_modifie(phi1,phi2,tt,y0)
Ec_em = [p**2/2 for p in p_em]
Ep_em = [-np.cos(q) for q in q_em]
Et_em = [Ec_em[i]+Ep_em[i] for i in range(len(Ec_em))]
print ('Euler modifie : |energie totale (t=Tfinal)-Energie totale (t=0)|=', abs(Et_em[-1]-Et_em[0]))

plt.figure(figsize=(18,5))
plt.subplot(131)
plt.plot(tt,q_em,tt,p_em)
plt.xlabel('t')
plt.legend(['x(t)','y(t)'])
plt.title('Euler modifie - x(t) et y(t)')
plt.subplot(132)
plt.plot(q_em,p_em)
plt.xlabel('x(t)')
plt.ylabel('y(t)')
plt.title('Euler modifie - y(x)')
plt.subplot(133)
plt.plot(tt,Ec_em,tt,Ep_em,tt,Et_em)
plt.xlabel('t')
plt.legend(['Cinetique','Potentielle','Totale'])
plt.title('Euler modifie - Energies');
```

```
Euler modifie : |energie totale (t=Tfinal)-Energie totale (t=0)|= 0.02457609735191002
```



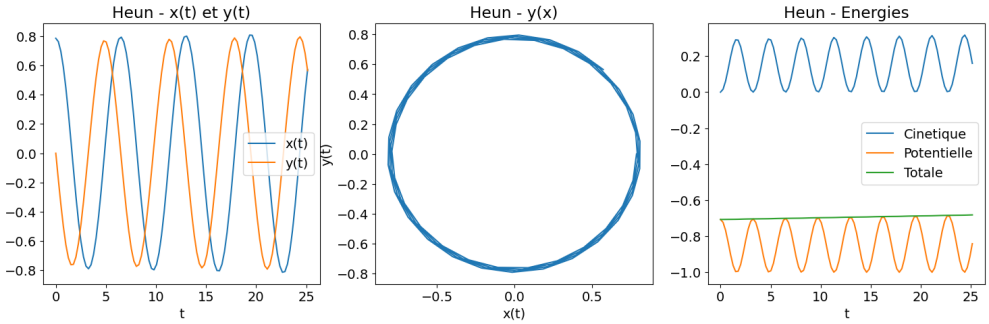
20.5.0.4. Méthodes de Heun

$$\begin{cases} x_0 = q(0), \\ y_0 = p(0), \\ \tilde{x}_n = x_n + h\varphi_1(t_n, x_n, y_n), \\ \tilde{y}_n = y_n + h\varphi_2(t_n, x_n, y_n), \\ x_{n+1} = x_n + \frac{h}{2}\left(\varphi_1(t_n, x_n, y_n) + \varphi_1(t_{n+1}, \tilde{x}_n, \tilde{y}_n)\right), \\ y_{n+1} = y_n + \frac{h}{2}\left(\varphi_2(t_n, x_n, y_n) + \varphi_2(t_{n+1}, \tilde{x}_n, \tilde{y}_n)\right). \end{cases}$$

```
def heun(phi1,phi2,tt,y0):
    h = tt[1]-tt[0]
    q = [y0[0]]
    p = [y0[1]]
    for i in range(len(tt)-1):
        qtilde = phi1( tt[i], q[i], p[i] )
        ptilde = phi2( tt[i], q[i], p[i] )
        q.append( q[i]+h*0.5*( phi1(tt[i],q[i],p[i]) + phi1(tt[i+1],q[i]+h*qtilde,p[i]+h*ptilde) ) )
        p.append( p[i]+h*0.5*( phi2(tt[i],q[i],p[i]) + phi2(tt[i+1],q[i]+h*qtilde,p[i]+h*ptilde) ) )
    return [q,p]
[q_heun, p_heun] = heun(phi1,phi2,tt,y0)
Ec_heun = [p**2/2 for p in p_heun]
Ep_heun = [-np.cos(q) for q in q_heun]
Et_heun = [Ec_heun[i]+Ep_heun[i] for i in range(len(Ec_heun))]
print ('Heun : |energie totale (t=Tfinal)-Energie totale (t=0)|=', abs(Et_heun[-1]-Et_heun[0]))

plt.figure(figsize=(18,5))
plt.subplot(131)
plt.plot(tt,q_heun,tt,p_heun)
plt.xlabel('t')
plt.legend(['x(t)','y(t)'])
plt.title('Heun - x(t) et y(t)')
plt.subplot(132)
plt.plot(q_heun,p_heun)
plt.xlabel('x(t)')
plt.ylabel('y(t)')
plt.title('Heun - y(x)')
plt.subplot(133)
plt.plot(tt,Ec_heun,tt,Ep_heun,tt,Et_heun)
plt.xlabel('t')
plt.legend(['Cinetique','Potentielle','Totale'])
plt.title('Heun - Energies');
```

```
Heun : |energie totale (t=Tfinal)-Energie totale (t=0)|= 0.025432751256537878
```



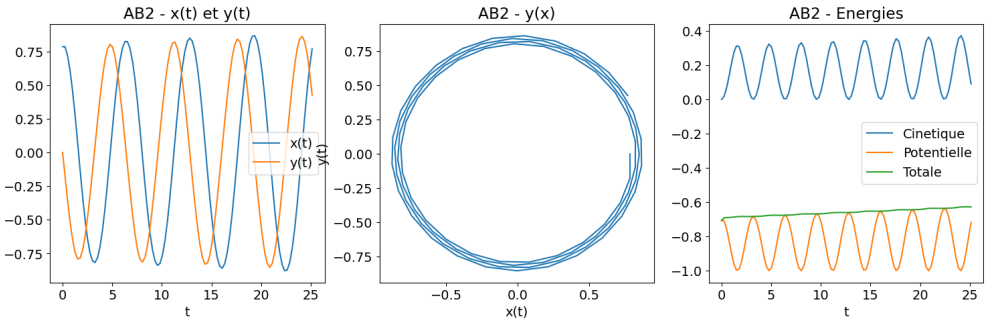
20.5.0.5. Méthode AB2

$$\begin{cases} x_0 = q(0), \\ y_0 = p(0), \\ x_1 = x_0 + h\varphi_1(t_0, x_0, y_0), \\ y_1 = y_0 + h\varphi_2(t_0, x_0, y_0), \\ x_{n+1} = x_n + \frac{h}{2}\left(3\varphi_1(t_n, x_n, y_n) - \varphi_1(t_{n-1}, x_{n-1}, y_{n-1})\right), \\ y_{n+1} = y_n + \frac{h}{2}\left(3\varphi_2(t_n, x_n, y_n) - \varphi_2(t_{n-1}, x_{n-1}, y_{n-1})\right). \end{cases}$$

```
def AB2(phi1,phi2,tt,y0):
    h = tt[1]-tt[0]
    q = [y0[0]]
    p = [y0[1]]
    q.append(q[0]+h*phi1(tt[0],q[0],p[0]))
    p.append(p[0]+h*phi2(tt[0],q[0],p[0]))
    for i in range(1,len(tt)-1):
        qtilde1 = phi1( tt[i],q[i],p[i] )
        qtilde2 = phi1( tt[i-1],q[i-1],p[i-1] )
        ptilde1 = phi2( tt[i],q[i],p[i] )
        ptilde2 = phi2( tt[i-1],q[i-1],p[i-1] )
        q.append( q[i] + (3*qtilde1-qtilde2)*h/2 )
        p.append( p[i] + (3*ptilde1-ptilde2)*h/2 )
    return [q,p]
[q_AB2, p_AB2] = AB2(phi1,phi2,tt,y0)
Ec_AB2 = [p**2/2. for p in p_AB2]
Ep_AB2 = [-np.cos(q) for q in q_AB2]
Et_AB2 = [Ec_AB2[i]+Ep_AB2[i] for i in range(len(Ec_AB2))]
print ('AB2 : |energie totale (t=Tfinal)-Energie totale (t=0)|=', abs(Et_AB2[-1]-Et_AB2[0]))

plt.figure(figsize=(18,5))
plt.subplot(131)
plt.plot(tt,q_AB2,tt,p_AB2)
plt.xlabel('t')
plt.legend(['x(t)','y(t)'])
plt.title('AB2 - x(t) et y(t)')
plt.subplot(132)
plt.plot(q_AB2,p_AB2)
plt.xlabel('x(t)')
plt.ylabel('y(t)')
plt.title('AB2 - y(x)')
plt.subplot(133)
plt.plot(tt,Ec_AB2,tt,Ep_AB2,tt,Et_AB2)
plt.xlabel('t')
plt.legend(['Cinetique','Potentielle','Totale'])
plt.title('AB2 - Energies');
```

```
AB2 : |energie totale (t=Tfinal)-Energie totale (t=0)|= 0.07960924681985027
```



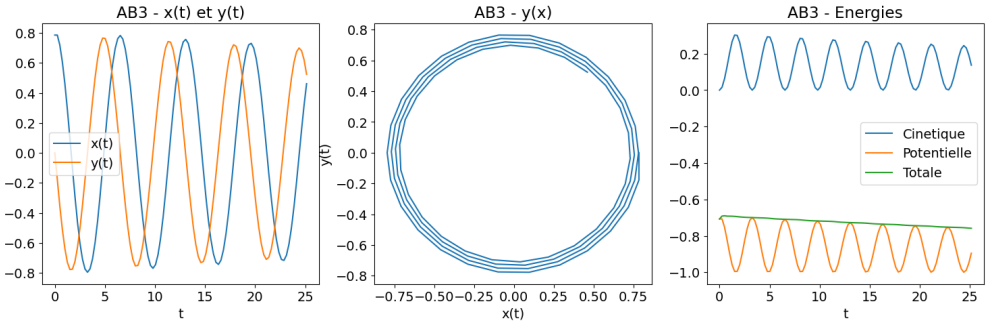
20.5.0.6. Méthode AB3

$$
\begin{cases}
x_0 = q(0), \\
y_0 = p(0), \\
x_1 = x_0 + h\varphi_1(t_0, x_0, y_0), \\
y_1 = y_0 + h\varphi_2(t_0, x_0, y_0), \\
x_2 = x_1 + \frac{h}{2}\left(3\varphi_1(t_1, x_1, y_1) - \varphi_1(t_0, x_0, y_0)\right), \\
y_2 = y_1 + \frac{h}{2}\left(3\varphi_2(t_1, x_1, y_1) - \varphi_2(t_0, x_0, y_0)\right), \\
x_{n+1} = x_n + \frac{h}{12}\left(23\varphi_1(t_n, x_n, y_n) - 16\varphi_1(t_{n-1}, x_{n-1}, y_{n-1}) + 5\varphi_1(t_{n-2}, x_{n-2}, y_{n-2})\right), \\
y_{n+1} = y_n + \frac{h}{12}\left(23\varphi_2(t_n, x_n, y_n) - 16\varphi_2(t_{n-1}, x_{n-1}, y_{n-1}) + 5\varphi_2(t_{n-2}, x_{n-2}, y_{n-2})\right).
\end{cases}
$$

```
def AB3(phi1,phi2,tt,y0):
    h = tt[1]-tt[0]
    q = [y0[0]]
    p = [y0[1]]
    q.append(q[0]+h*phi1(tt[0],q[0],p[0]))
    p.append(p[0]+h*phi2(tt[0],q[0],p[0]))
    q.append(q[1]+h*(3*phi1(tt[1],q[1],p[1])-phi1(tt[0],q[0],p[0]))/2. )
    p.append(p[1]+h*(3*phi2(tt[1],q[1],p[1])-phi2(tt[0],q[0],p[0]))/2.)
    for i in range(2,len(tt)-1):
        qtilde1 = phi1( tt[i],q[i],p[i] )
        qtilde2 = phi1( tt[i-1],q[i-1],p[i-1] )
        qtilde3 = phi1( tt[i-2],q[i-2],p[i-2] )
        ptilde1 = phi2( tt[i],q[i],p[i] )
        ptilde2 = phi2( tt[i-1],q[i-1],p[i-1] )
        ptilde3 = phi2( tt[i-2],q[i-2],p[i-2] )
        q.append( q[i] + (23*qtilde1-16*qtilde2+5*qtilde3)*h/12 )
        p.append( p[i] + (23*ptilde1-16*ptilde2+5*ptilde3)*h/12 )
    return [q,p]
[q_AB3, p_AB3] = AB3(phi1,phi2,tt,y0)
Ec_AB3 = [p**2/2. for p in p_AB3]
Ep_AB3 = [-np.cos(q) for q in q_AB3]
Et_AB3 = [Ec_AB3[i]+Ep_AB3[i] for i in range(len(Ec_AB3))]
print( 'AB3 : |energie totale (t=Tfinal)-Energie totale (t=0)|=', abs(Et_AB3[-1]-Et_AB3[0]))

plt.figure(figsize=(18,5))
plt.subplot(131)
plt.plot(tt,q_AB3,tt,p_AB3)
plt.xlabel('t')
plt.legend(['x(t)','y(t)'])
plt.title('AB3 - x(t) et y(t)')
plt.subplot(132)
plt.plot(q_AB3,p_AB3)
plt.xlabel('x(t)')
plt.ylabel('y(t)')
plt.title('AB3 - y(x)')
plt.subplot(133)
plt.plot(tt,Ec_AB3,tt,Ep_AB3,tt,Et_AB3)
plt.xlabel('t')
plt.legend(['Cinetique','Potentielle','Totale'])
plt.title('AB3 - Energies');
```

```
AB3 : |energie totale (t=Tfinal)-Energie totale (t=0)|= 0.051119920140796915
```



20.5.0.7. Méthode RK4

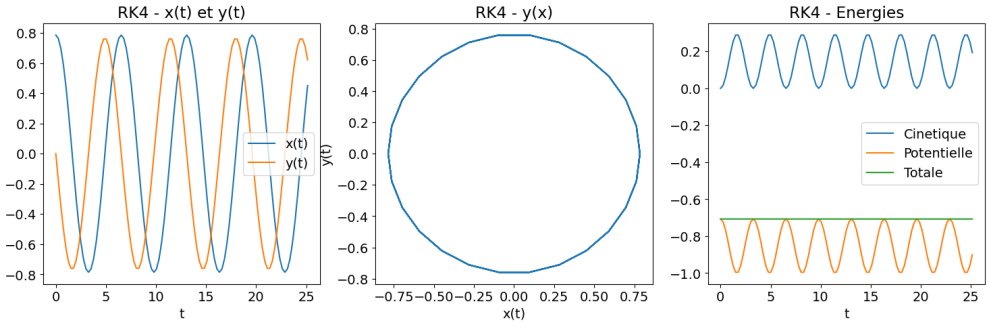
$$
\begin{cases}
x_0 = q(0), \\
y_0 = p(0), \\
\tilde{x}_n = x_n + \frac{h}{2}\varphi_1(t_n, x_n, y_n), \\
\tilde{y}_n = y_n + \frac{h}{2}\varphi_2(t_n, x_n, y_n), \\
\check{x}_n = x_n + \frac{h}{2}\varphi_1(t_n + h/2, \tilde{x}_n, \tilde{y}_n), \\
\check{y}_n = y_n + \frac{h}{2}\varphi_2(t_n + h/2, \tilde{x}_n, \tilde{y}_n), \\
\hat{x}_n = x_n + h\varphi_1(t_{n+1}, \check{x}_n, \check{y}_n), \\
\hat{y}_n = y_n + h\varphi_2(t_{n+1}, \check{x}_n, \check{y}_n), \\
x_{n+1} = x_n + \frac{h}{6}\left(\varphi_1(t_n, x_n, y_n) + 2\varphi_1(t_n + h/2, \tilde{x}_n, \tilde{y}_n) + 2\varphi_1(t_n + h/2, \check{x}_n, \check{y}_n) + \varphi_1(t_{n+1}, \hat{x}_n, \hat{y}_n)\right), \\
y_{n+1} = y_n + \frac{h}{6}\left(\varphi_2(t_n, x_n, y_n) + 2\varphi_2(t_n + h/2, \tilde{x}_n, \tilde{y}_n) + 2\varphi_2(t_n + h/2, \check{x}_n, \check{y}_n) + \varphi_2(t_{n+1}, \hat{x}_n, \hat{y}_n)\right).
\end{cases}
$$

```
def RK4(phi1,phi2,tt,y0):
    h = tt[1]-tt[0]
    q = [y0[0]]
    p = [y0[1]]
    for i in range(len(tt)-1):
        qtilde = q[i] + 0.5*h * phi1( tt[i], q[i], p[i] )
        ptilde = p[i] + 0.5*h * phi2( tt[i], q[i], p[i] )
        qv = q[i] + 0.5*h * phi1( tt[i]+h*0.5, qtilde, ptilde )
        pv = p[i] + 0.5*h * phi2( tt[i]+h*0.5, qtilde, ptilde )
        qhat = q[i] + h * phi1( tt[i+1], qv, pv )
        phat = p[i] + h * phi2( tt[i+1], qv, pv )
        q.append( q[i]+h/6*( phi1(tt[i],q[i],p[i]) + 2*phi1(tt[i]+0.5*h,qtilde,ptilde)+
↪   2*phi1(tt[i]+0.5*h,qv,pv)+ phi1(tt[i+1],qhat,phat) ) )
        p.append( p[i]+h/6*( phi2(tt[i],q[i],p[i]) + 2*phi2(tt[i]+0.5*h,qtilde,ptilde)+
↪   2*phi2(tt[i]+0.5*h,qv,pv)+ phi2(tt[i+1],qhat,phat) ) )
    return [q,p]
[q_RK4, p_RK4] = RK4(phi1,phi2,tt,y0)
Ec_RK4 = [p**2/2. for p in p_RK4]
Ep_RK4 = [-np.cos(q) for q in q_RK4]
Et_RK4 = [Ec_RK4[i]+Ep_RK4[i] for i in range(len(Ec_RK4))]
print ('RK4 : |energie totale (t=Tfinal)-Energie totale (t=0)|=', abs(Et_RK4[-1]-Et_RK4[0]))

plt.figure(figsize=(18,5))
plt.subplot(131)
plt.plot(tt,q_RK4,tt,p_RK4)
plt.xlabel('t')
plt.legend(['x(t)','y(t)'])
plt.title('RK4 - x(t) et y(t)')
plt.subplot(132)
plt.plot(q_RK4,p_RK4)
plt.xlabel('x(t)')
```

```
plt.ylabel('y(t)')
plt.title('RK4 - y(x)')
plt.subplot(133)
plt.plot(tt,Ec_RK4,tt,Ep_RK4,tt,Et_RK4)
plt.xlabel('t')
plt.legend(['Cinetique','Potentielle','Totale'])
plt.title('RK4 - Energies');
```

```
RK4 : |energie totale (t=Tfinal)-Energie totale (t=0)|= 8.32124615898211e-05
```



Cinquième partie

Annales

Chapitre 21.

2025 Contrôle Continu

2h, déposer le notebook dans Moodle (si 1/3 temps supplémentaire, 2h40)

Autorisés : notebooks de cours, notes personnelles.

Le contrôle continu est composé de 3 exercices indépendants. Deux versions de chaque exercice sont proposées ou bien l'énoncé dépend de votre numéro étudiant.

Compléter ici :

- NOM :
- Prénom :
- N° étudiant :

```
import numpy as np
import matplotlib.pyplot as plt

from scipy.optimize import fsolve
from scipy.integrate import solve_ivp, odeint

import sympy as sp

from IPython.display import display, Markdown, Math
```

21.1. EXERCICE 1 - sujet A : Système

On considère le système d'équations différentielles suivant

$$\begin{cases} x'(t) = 2x(t) + 3y^2(t), \\ y'(t) = 3x^2(t) - 2y(t), \\ x(0) = -\frac{1}{2}, \\ y(0) = \frac{1}{2}. \end{cases} \qquad t \in [0, 5].$$

- Vérifier **analytiquement** que $I(t) = x^3(t) - 2x(t)y(t) - y^3(t)$ est un invariant.
- Calculer une solution approchée avec la méthode de **Heun** avec 30 sous-intervalles. Tracer ensuite
  - les courbes $x(t)$ et $y(t)$,
  - la courbe paramétrée $(x(t), y(t))$,
  - la courbe $I(t)$.
- Pour estimer l'**ordre** de convergence de la méthode de Heun, on ne peut pas utiliser la solution exacte car elle n'est pas connue. On peut cependant utiliser l'invariant $I(t)$ dont on connaît la valeur exacte. Vérifier sur l'invariant que l'ordre de convergence est au moins 2. Avez-vous une explication pour le résultat obtenu ?

Correction

```
# 2D system
# x' = 2x +3y^2
# y' = 3x^2 - 2y
# x(0) = -1/2
# y(0) = 1/2
# Invariant: x^3 - y^3 - 2xy
# ============================

t0 = 0
tfin = 5
x0 = -1/2
y0 = 1/2

phi = lambda t, zz : np.array([ 2*zz[0]+3*zz[1]**2,
                                3*zz[0]**2 - 2*zz[1] ])

Invariant = lambda zz : zz[0]**3 - zz[1]**3 -2*zz[0]*zz[1]

I_init = Invariant([x0, y0])
```

On vérifie que I est bien un invariant pour le système donné. On peut calculer I' à la main ou utiliser sympy.

Notons $\mathcal{I}(x, y) = x^3 - y^3 - 2xy$ ainsi $I(t) = \mathcal{I}(x(t), y(t))$.

$$\begin{aligned} I'(t) &= \frac{\partial \mathcal{I}}{\partial x} \cdot x'(t) + \frac{\partial \mathcal{I}}{\partial y} \cdot y'(t) \\ &= \Big(3x^2(t) - 2y(t)\Big)x'(t) - y'(t)\Big(3y^2(t) + 2x(t)\Big) \\ &= y'(t)x'(t) - y'(t)x'(t) \\ &= 0. \end{aligned}$$

On a donc bien $I'(t) = 0$ ainsi $I(t)$ est invariant et vaut $I(0) = \frac{1}{4}$ pour tout t.

```
t = sp.symbols('t')
x = sp.Function('x')(t)
y = sp.Function('y')(t)

eq1 = sp.Eq(sp.diff(x, t), phi(t, [x, y])[0])
eq2 = sp.Eq(sp.diff(y, t), phi(t, [x, y])[1])

I = Invariant([x, y])
display(sp.Eq(sp.Symbol('I'), I))
dI = sp.diff(I, t)
dI = dI.subs(sp.diff(x, t), eq1.rhs)
dI = dI.subs(sp.diff(y, t), eq2.rhs)
dI = dI.simplify()
display(sp.Eq(sp.Symbol("I'(t)"), dI))
display(sp.Eq(sp.Symbol("I(0)"), I_init))
```

$I = x^3(t) - 2x(t)y(t) - y^3(t)$

$I'(t) = 0$

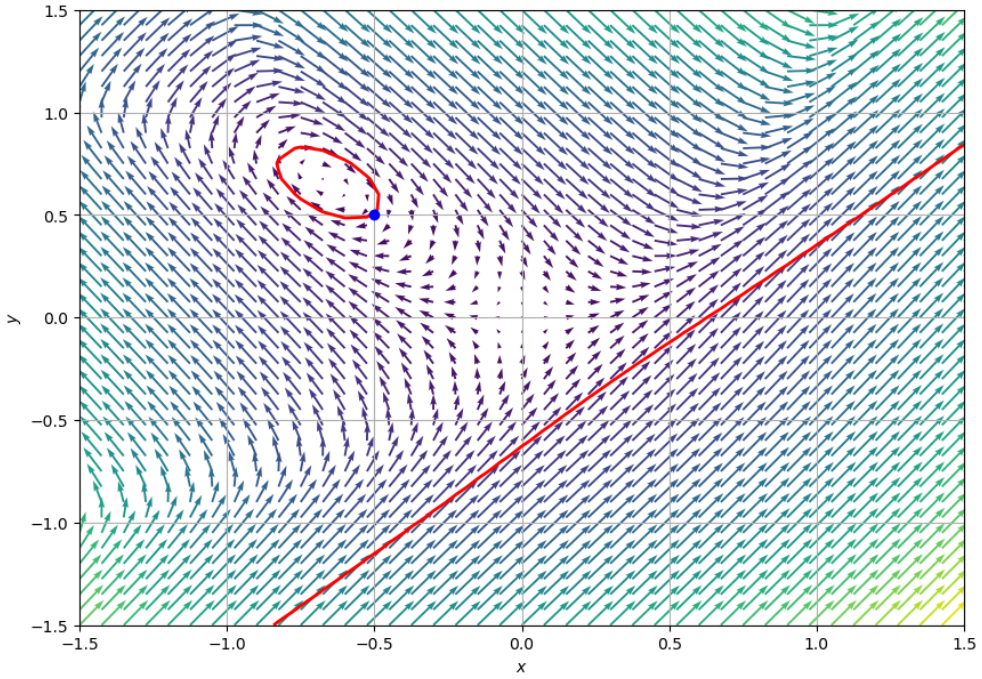
$I(0) = 0.25$

Même si ce n'est pas explicitement demandé dans l'énoncé, c'est toujours une bonne idée d'afficher le portrait de phase pour ce genre de système. Ci-dessous on affiche, de plus, la courbe de niveau de l'invariant $I(t)$ correspondant à $I(t) = I(0)$.

```
# Plotting the vector field
# ========================================

Y1,Y2 = np.meshgrid(np.linspace(-1.5,1.5,41),np.linspace(-1.5,1.5,41))
tt = np.linspace(t0,tfin,100)
V1,V2 = phi(tt,[Y1,Y2])
r1 = np.sqrt(1+V1**2)
r2 = np.sqrt(1+V2**2)
M = np.sqrt(V1**2 + V2**2)

plt.figure(figsize=(10,7))
plt.quiver(Y1, Y2, V1/r1, V2/r2, M, cmap=plt.cm.viridis, scale_units='xy',scale=10.)
plt.contour(Y1,Y2,Invariant([Y1,Y2]),levels=[I_init],colors='r',linewidths=2)
plt.plot(x0,y0,'bo')
plt.grid()
plt.xlabel(r'$x$')
plt.ylabel(r'$y$');
```




```
# Schéma de Heun
# ================================================================================

def Heun_system(phi,tt, yy0):
    nb_eq = len(yy0) # nombre d'équations
    N = len(tt)-1 # N+1 points, donc N intervalles
    uu = np.zeros((N+1,nb_eq)) # colonnes = variables, lignes = temps
    uu[0] = yy0
    h = tt[1]-tt[0]
    for n in range(N):
        K1 = phi(tt[n],uu[n])
        K2 = phi(tt[n+1],uu[n]+h*K1)
        uu[n+1] = uu[n] + h/2 * ( K1 + K2 )
    return uu
```

```
# TEST 1
# =======
N = 30
t_values = np.linspace(t0, tfin, N+1)
```

```
yy0 = np.array([x0, y0])
uu = Heun_system(phi, t_values, yy0)
# from scipy.integrate import odeint
# uu = odeint(phi, yy0, t_values, tfirst=True)
II = Invariant(uu.T)

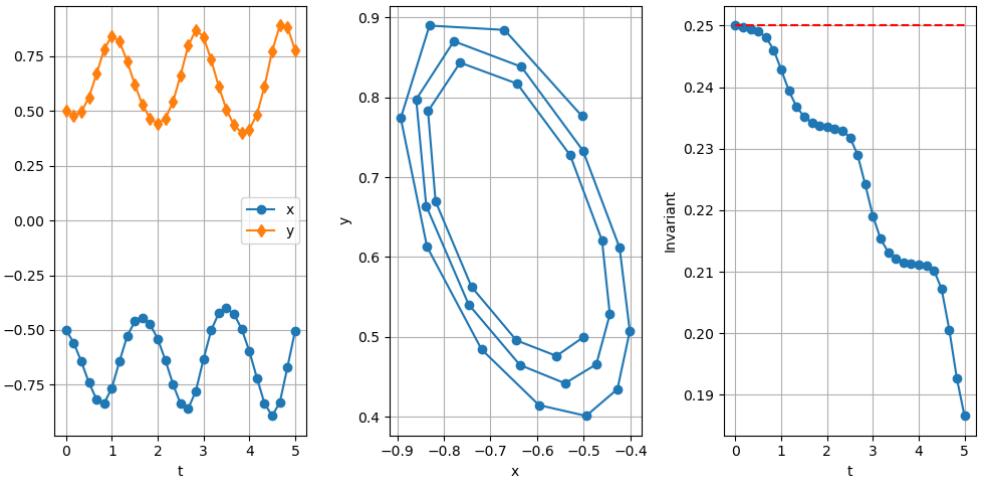
fig, axx = plt.subplots(1,3, figsize=(10,5))

axx[0].plot(t_values, uu[:,0], 'o-',label='x')
axx[0].plot(t_values, uu[:,1], 'd-',label='y')
axx[0].set_xlabel('t')
axx[0].legend()
axx[0].grid()

axx[1].plot(uu[:,0], uu[:,1], 'o-')
axx[1].set_xlabel('x')
axx[1].set_ylabel('y')
axx[1].grid()

axx[2].plot(t_values, II, 'o-')
axx[2].plot(t_values, I_init*np.ones(N+1), 'r--')
axx[2].set_xlabel('t')
axx[2].set_ylabel('Invariant')
axx[2].grid()

plt.tight_layout()
```



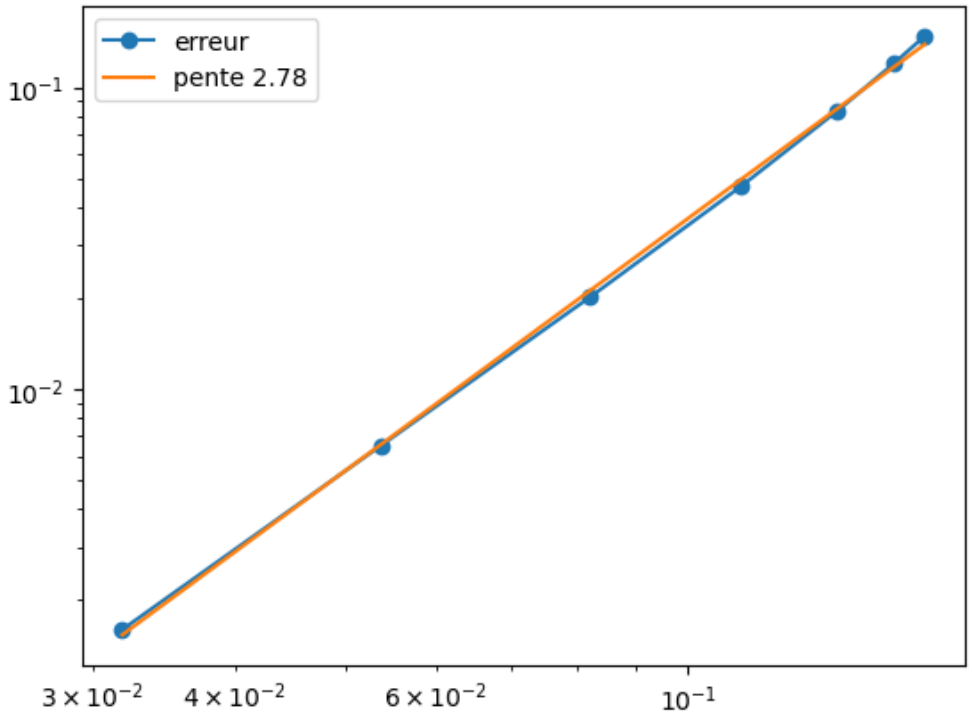

```
# TEST 2 : ordre de convergence
# ==============================

H, err = [], []

for k in range(7):
    N += 2**k
    t_values = np.linspace(t0, tfin, N+1)
    uu = Heun_system(phi, t_values, yy0)
    II = Invariant(uu.T)
    err.append(np.linalg.norm(II-I_init))
    H.append(t_values[1]-t_values[0])
    if err[-1] <1.e-14 :
        break

plt.loglog(H, err, 'o-', label='erreur')
a,b = np.polyfit(np.log(H), np.log(err), 1)
```

```
plt.loglog(H, np.exp(a*np.log(H)+b), label=f'pente {a:.2f}')
plt.legend()
plt.show()
```



L'ordre calculé sur l'invariant ne peut pas être inférieur à l'ordre théorique sur la solution. En revanche, il peut être meilleur grâce à la structure particulière de l'invariant et aux compensations possibles entre les erreurs sur chaque composante de la solution.

21.2. EXERCICE 1 - sujet B : Système

On considère le système d'équations différentielles suivant

$$\begin{cases} x'(t) = -(2x(t) + 3y^2(t)), \\ y'(t) = 2y(t) - 3x^2(t), \\ x(0) = -\frac{1}{2}, \\ y(0) = \frac{1}{2}. \end{cases} \qquad t \in [0, 5].$$

- Vérifier **analytiquement** que $I(t) = y^3(t) - x^3(t) + 2x(t)y(t)$ est un invariant.
- Calculer une solution approchée avec la méthode d'**Euler Modifiée** avec 30 sous-intervalles. Tracer ensuite
  - les courbes $x(t)$ et $y(t)$,
  - la courbe paramétrée $(x(t), y(t))$,
  - la courbe $I(t)$.
- Pour estimer l'**ordre** de convergence de la méthode d'Euler Modifiée, on ne peut pas utiliser la solution exacte car elle n'est pas connue. On peut cependant utiliser l'invariant $I(t)$ dont on connaît la valeur exacte. Vérifier sur l'invariant que l'ordre de convergence est au moins 2. Avez-vous une explication pour le résultat obtenu ?

Correction

```
# 2D system
# x' = -2x -3y^2
```

```
# y' = -3x^2 + 2y
# x(0) = -1/2
# y(0) = 1/2
# Invariant: y^3 - x^3 + 2xy
# ==============================

t0 = 0
tfin = 5
x0 = -1/2
y0 = 1/2

phi = lambda t, zz : np.array([ -2*zz[0] -3*zz[1]**2,
                        -3*zz[0]**2 + 2*zz[1] ])

Invariant = lambda zz : zz[1]**3 - zz[0]**3 +2*zz[0]*zz[1]

I_init = Invariant([x0, y0])
```

On vérifie que I est bien un invariant pour le système donné. On peut calculer I' à la main ou utiliser sympy.

Notons $\mathcal{I}(x, y) = y^3 - x^3 + 2xy$ ainsi $I(t) = \mathcal{I}(x(t), y(t))$.

$$\begin{aligned} I'(t) &= \frac{\partial \mathcal{I}}{\partial x} \cdot x'(t) + \frac{\partial \mathcal{I}}{\partial y} \cdot y'(t) \\ &= \Big(3y^2(t) + 2x(t)\Big)y'(t) - x'(t)\Big(3x^2(t) - 2y(t)\Big) \\ &= -x'(t)y'(t) + x'(t)y'(t) \\ &= 0. \end{aligned}$$

On a donc bien $I'(t) = 0$ ainsi $I(t)$ est invariant et vaut $I(0) = -\frac{1}{4}$ pour tout t.

```
t = sp.symbols('t')
x = sp.Function('x')(t)
y = sp.Function('y')(t)

eq1 = sp.Eq(sp.diff(x, t), phi(t, [x, y])[0])
eq2 = sp.Eq(sp.diff(y, t), phi(t, [x, y])[1])

I = Invariant([x, y])
display(sp.Eq(sp.Symbol('I'), I))
dI = sp.diff(I, t)
dI = dI.subs(sp.diff(x, t), eq1.rhs)
dI = dI.subs(sp.diff(y, t), eq2.rhs)
dI = dI.simplify()
display(sp.Eq(sp.Symbol("I'(t)"), dI))
display(sp.Eq(sp.Symbol("I(0)"), I_init))
```

$I = -x^3(t) + 2x(t)y(t) + y^3(t)$

$I'(t) = 0$

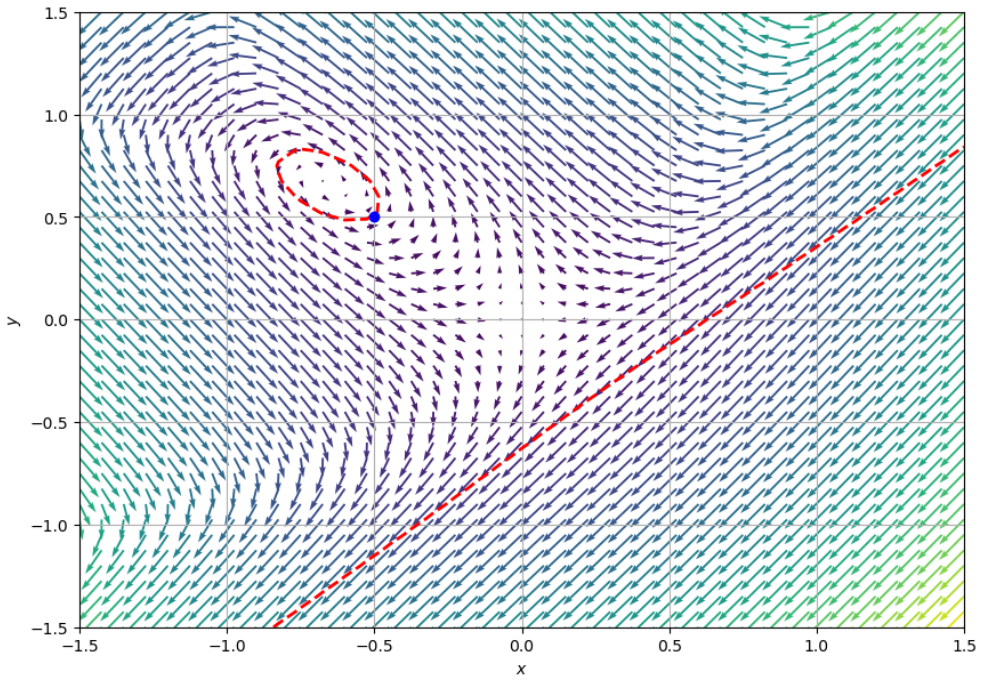
$I(0) = -0.25$

Même si ce n'est pas explicitement demandé dans l'énoncé, c'est toujours une bonne idée d'afficher le portrait de phase pour ce genre de système. Ci-dessous on affiche, de plus, la courbe de niveau de l'invariant $I(t)$ correspondant à $I(t) = I(0)$.

```
# Plotting the vector field
# ========================================

Y1,Y2 = np.meshgrid(np.linspace(-1.5,1.5,41),np.linspace(-1.5,1.5,41))
tt = np.linspace(t0,tfin,100)
V1,V2 = phi(tt,[Y1,Y2])
r1 = np.sqrt(1+V1**2)
r2 = np.sqrt(1+V2**2)
M = np.sqrt(V1**2 + V2**2)

plt.figure(figsize=(10,7))
plt.quiver(Y1, Y2, V1/r1, V2/r2, M, cmap=plt.cm.viridis, scale_units='xy',scale=10.)
plt.contour(Y1,Y2,Invariant([Y1,Y2]),levels=[I_init],colors='r',linewidths=2)
plt.plot(x0,y0,'bo')
plt.grid()
plt.xlabel(r'$x$')
plt.ylabel(r'$y$');
```




```
# Schéma d'Euler Modifié
# ================================================================================

def EM_system(phi,tt, yy0):
    nb_eq = len(yy0) # nombre d'équations
    N = len(tt)-1 # N+1 points, donc N intervalles
    uu = np.zeros((N+1,nb_eq)) # colonnes = variables, lignes = temps
    uu[0] = yy0
    h = tt[1]-tt[0]
    for n in range(N):
        K1 = phi(tt[n],uu[n])
        uu[n+1] = uu[n] + h*phi(tt[n]+h/2,uu[n]+h/2*K1)
    return uu
```

```
# TEST 1
#=======

N = 30
t_values = np.linspace(t0, tfin, N+1)
```

```
yy0 = np.array([x0, y0])
uu = EM_system(phi, t_values, yy0)
# from scipy.integrate import odeint
# uu = odeint(phi, yy0, t_values, tfirst=True)
II = Invariant(uu.T)

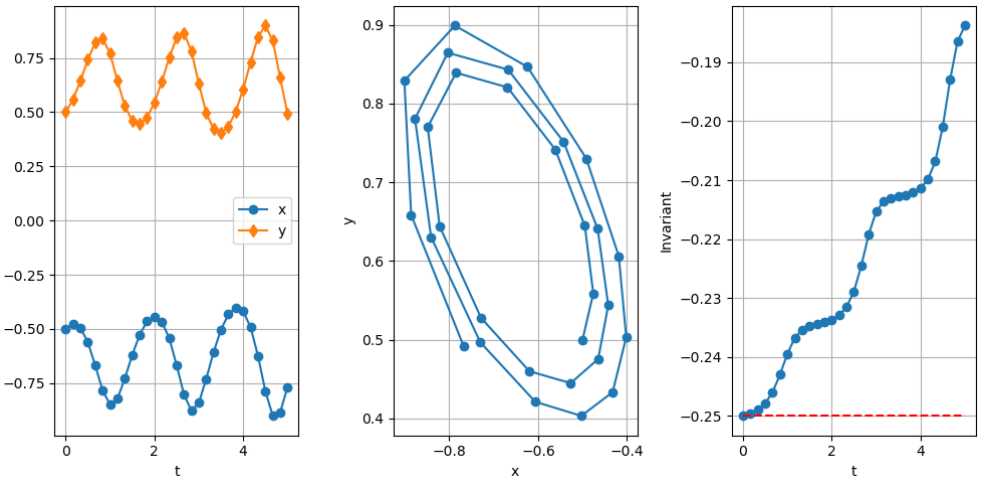
fig, axx = plt.subplots(1,3, figsize=(10,5))

axx[0].plot(t_values, uu[:,0], 'o-',label='x')
axx[0].plot(t_values, uu[:,1], 'd-',label='y')
axx[0].set_xlabel('t')
axx[0].legend()
axx[0].grid()

axx[1].plot(uu[:,0], uu[:,1], 'o-')
axx[1].set_xlabel('x')
axx[1].set_ylabel('y')
axx[1].grid()

axx[2].plot(t_values, II, 'o-')
axx[2].plot(t_values, I_init*np.ones(N+1), 'r--')
axx[2].set_xlabel('t')
axx[2].set_ylabel('Invariant')
axx[2].grid()

plt.tight_layout()
```



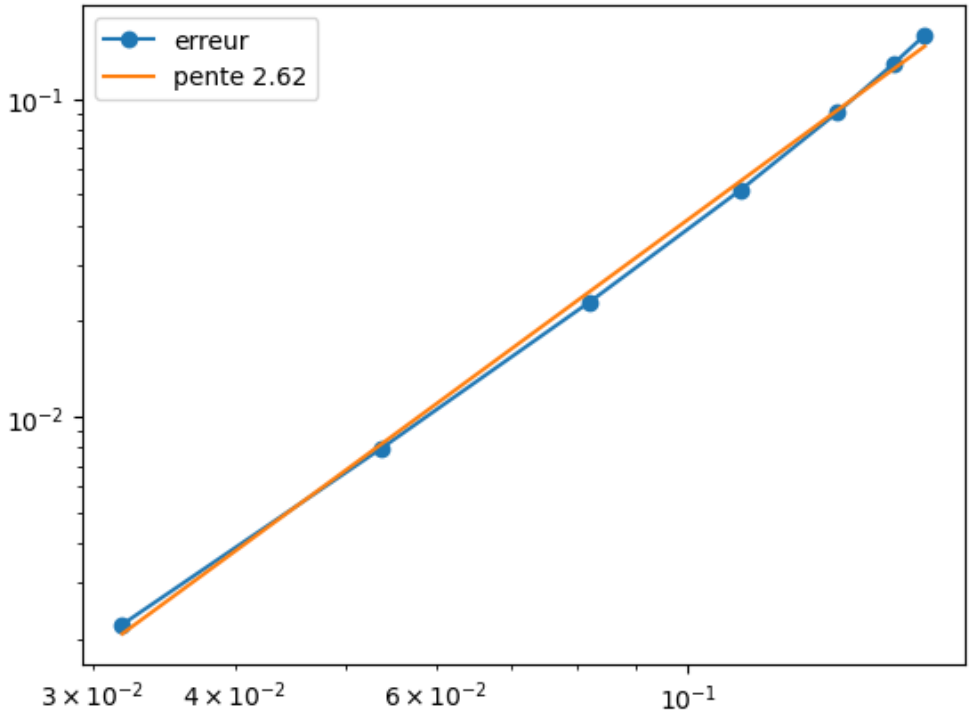

```
# TEST 2 : ordre de convergence
# ==============================

H, err = [], []

for k in range(7):
    N += 2**k
    t_values = np.linspace(t0, tfin, N+1)
    uu = EM_system(phi, t_values, yy0)
    II = Invariant(uu.T)
    err.append(np.linalg.norm(II-I_init))
    H.append(t_values[1]-t_values[0])
    if err[-1] <1.e-14 :
        break

plt.loglog(H, err, 'o-', label='erreur')
a,b = np.polyfit(np.log(H), np.log(err), 1)
```

```
plt.loglog(H, np.exp(a*np.log(H)+b), label=f'pente {a:.2f}')
plt.legend()
plt.show()
```



L'ordre calculé sur l'invariant ne peut pas être inférieur à l'ordre théorique sur la solution. En revanche, il peut être meilleur grâce à la structure particulière de l'invariant et aux compensations possibles entre les erreurs sur chaque composante de la solution.

21.3. EXERCICE 2 - sujet A : Schéma MPL

La méthode multi-pas linéaire suivante est-elle convergente ? Justifier.

$$u_{n+1} = -\frac{1}{4}u_n + \frac{1}{2}u_{n-1} + \frac{3}{4}u_{n-2} + \frac{h}{8}(9\varphi_n + 5\varphi_{n-1})$$

Correction

Une méthode multi-pas linéaire est convergente ssi elle est consistante et zéro-stable.

Cette méthode est bien zéro-stable, car toutes les racines de son polynôme caractéristique sont dans le cercle unité et seule la racine 1 est sur la frontière du cercle unité.

Cependant, elle n'est pas consistante, car $\xi(1) \neq 1$.

Vérifions-le avec symmpy :

```
# ========================
q = 3
# ========================

r = sp.Symbol('r')

# ========================
def xi(i, q):
    sa = sum((-j)**i * aa[j] for j in range(q))
```

```
    sb = b_m1 + sum((-j)**(i-1) * bb[j] for j in range(1, q))
    if i == 1:
        sb += bb[0]
    return (sa).factor() + (i * sb).factor()

# ========================
aa = [-1/sp.S(4), 1/sp.S(2), 3/sp.S(4)]
bb = [9/sp.S(8), -5/sp.S(8), 0]
b_m1 = 0

display(Markdown(f"C'est une méthode à ${q = }$ pas"))

display(Markdown(f"Pour étudier la zéro-stabilité, on écrit le premier polynôme caractéristique"))
rho = sp.poly(r**(q) - sum( aa[j] * r**(q-1-j) for j in range(q)), r)
texte =  r"$\varrho(r)=" + sp.latex(rho.as_expr(),order='old') +  r"$"
display(Math(texte))
display(Markdown(f"Il a 3 racines :"))
racines = sp.roots(rho,r)
for k,v in racines.items():
    display(Markdown(f" - ${sp.latex(k)}$ de module ${sp.latex(abs(k))} \\approx {float(abs(k))}$"))

display(Markdown(f"Pour étudier la consistance, on calcule $\\xi(i)$"))
for i in range(2):
    display(Markdown(f"$\\xi({i}) = {sp.latex(xi(i, q))}$"))
```

C'est une méthode à $q = 3$ pas

Pour étudier la zéro-stabilité, on écrit le premier polynôme caractéristique

$$\varrho(r) = -\frac{3}{4} - \frac{r}{2} + \frac{r^2}{4} + r^3$$

Il a 3 racines :

- 1 de module $1 \approx 1.0$
- $-\frac{5}{8} - \frac{\sqrt{23}i}{8}$ de module $\frac{\sqrt{3}}{2} \approx 0.8660254037844386$
- $-\frac{5}{8} + \frac{\sqrt{23}i}{8}$ de module $\frac{\sqrt{3}}{2} \approx 0.8660254037844386$

Pour étudier la consistance, on calcule $\xi(i)$

$\xi(0) = 1$

$\xi(1) = -\frac{3}{2}$

21.4. EXERCICE 2 - sujet B : Schéma MPL

La méthode multi-pas linéaire suivante est-elle convergente ? Justifier.

$$u_{n+1} = -\frac{1}{4}u_n + \frac{1}{2}u_{n-1} + \frac{3}{4}u_{n-2} + \frac{h}{8}(19\varphi_n + 5\varphi_{n-2})$$

Correction

Une méthode multi-pas est convergente ssi elle est consistante et zéro-stable.

Elle est zéro-stable, car toutes les racines de son polynôme caractéristique sont dans le cercle unité et seul 1 est sur la frontière du cercle unité.

Elle est aussi consistante d'ordre 3, car $\xi(i) = 1$ pour $i = 1, 2, 3$. Étant une méthode explicite à 3 pas, c'est l'ordre maximal possible d'après la première barrière de Dalquist.

Vérifions ces calculs avec sympy :

```
# ========================
q = 3
# ========================

r = sp.Symbol('r')

# ========================
def xi(i, q):
    sa = sum((-j)**i * aa[j] for j in range(q))
    sb = b_m1 + sum((-j)**(i-1) * bb[j] for j in range(1, q))
    if i == 1:
        sb += bb[0]
    return (sa).factor() + (i * sb).factor()

# ========================
aa = [-1/sp.S(4), 1/sp.S(2), 3/sp.S(4)]
bb = [19/sp.S(8), 0, 5/sp.S(8)]
b_m1 = 0

display(Markdown(f"C'est une méthode à ${q = }$ pas"))

display(Markdown(f"Pour étudier la zéro-stabilité, on écrit le premier polynôme caractéristique"))
rho = sp.poly(r**(q) - sum( aa[j] * r**(q-1-j) for j in range(q)), r)
texte =  r"$\varrho(r)=" + sp.latex(rho.as_expr(),order='old') +  r"$"
display(Math(texte))
display(Markdown(f"dont les racines sont :"))
racines = sp.roots(rho,r)
for k,v in racines.items():
    display(Markdown(f" - ${sp.latex(k)}$ de module ${sp.latex(abs(k))} \\approx {float(abs(k))}$"))

display(Markdown(f"Pour étudier la consistance (et l'ordre de consistance), on calcule $\\xi(i)$"))
for i in range(4):
    display(Markdown(f"$\\xi({i}) = {sp.latex(xi(i, q))}$"))
```

C'est une méthode à $q = 3$ pas

Pour étudier la zéro-stabilité, on écrit le premier polynôme caractéristique

$$\varrho(r) = -\frac{3}{4} - \frac{r}{2} + \frac{r^2}{4} + r^3$$

dont les racines sont :

- 1 de module $1 \approx 1.0$
- $-\frac{5}{8} - \frac{\sqrt{23}i}{8}$ de module $\frac{\sqrt{3}}{2} \approx 0.8660254037844386$
- $-\frac{5}{8} + \frac{\sqrt{23}i}{8}$ de module $\frac{\sqrt{3}}{2} \approx 0.8660254037844386$

Pour étudier la consistance (et l'ordre de consistance), on calcule $\xi(i)$

$\xi(0) = 1$

$\xi(1) = 1$

$\xi(2) = 1$

$\xi(3) = 1$

21.5. EXERCICE 3 - sujet A : Schéma RK semi-implicite à 2 étages

On considère le schéma de Runge-Kutta explicite à 2 étages défini par le tableau de Butcher suivant :

$$\begin{array}{c|cc} 1 & 1 & 0 \\ 1/3 & 1/3 & 0 \\ \hline & 1/4 & 3/4 \end{array}$$

Après avoir calculé l'ordre de convergence théorique, implémenter le schéma et vérifier expérimentalement son ordre de convergence sur un exemple que vous choisirez.

Correction

On calcule l'ordre de convergence théorique du schéma de Runge-Kutta à 2 étages donné.

C'est un schéma semi-implicite. D'après la barrière de Butcher, l'ordre est au plus 4.

En vérifiant les conditions de convergence, on trouve que l'ordre est 2.

```
# ================================================================================
# Fonctions pour calculer les conditions de consistance pour un schéma RK.
#
# La fonction ordre_RK calcule les conditions pour les ordres 1, 2, 3 et 4 (même si théoriquement cela
 ↪  n'est pas possible)
# - affiche la matrice de Butcher
# - affiche les conditions de consistance obtenues pour chaque ordre par la fonction ordre_i
# - renvoie une liste d'équations pour une éventuelle résolution
#
# Chaque fonction ordre_i calcule les conditions pour être d'ordre i
# - peut afficher les conditions de consistance [actuellement en commentaire]
# - renvoie la liste d'équations
#
# ================================================================================

prlat = lambda *args : display(Math(''.join( sp.latex(arg) for arg in args )))




def ordre_RK(s, type="implicite", A=None, b=None, c=None):

    j = sp.symbols('j')

    if A is None:
        if type == "implicite":
            A = sp.MatrixSymbol('a', s, s)
        elif type=="semi-implicite":
            A = sp.Matrix(s, s, lambda i,j: sp.Symbol('a{}{}'.format(i, j)) if i >= j else 0)
        elif type=="explicite":
            A = sp.Matrix(s, s, lambda i,j: sp.Symbol('a{}{}'.format(i, j)) if i > j else 0)

    else:
        A = sp.Matrix(A)
    if c is None:
        c = sp.symbols(f'c_0:{s}')
    if b is None:
        b = sp.symbols(f'b_0:{s}')



    display(Markdown("**Matrice de Butcher**"))
    But = matrice_Butcher(s, A, b, c)

    Eqs = []

    display(Markdown(f"**On a {s+1} conditions pour avoir consistance (= pour être d'ordre 1)**"))
    Eq_1 = ordre_1(s, A, b, c, j)
    Eqs.append(Eq_1)
```

```
    display(*Eq_1)

    display(Markdown("**On doit ajouter 1 condition pour être d'ordre 2**"))
    Eq_2 = ordre_2(s, A, b, c, j)
    Eqs.append([Eq_2])
    display(Eq_2)

    display(Markdown("**On doit ajouter 2 conditions pour être d'ordre 3**"))
    Eq_3 = ordre_3(s, A, b, c, j)
    Eqs.append(Eq_3)
    display(*Eq_3)

    display(Markdown("**On doit ajouter 4 conditions pour être d'ordre 4**"))
    Eq_4 = ordre_4(s, A, b, c, j)
    Eqs.append(Eq_4)
    display(*Eq_4)

    return Eqs


def matrice_Butcher(s, A, b, c):
    But = sp.Matrix(A)
    But = But.col_insert(0, sp.Matrix(c))
    last = [sp.Symbol(" ")]
    last.extend(b)
    But = But.row_insert(s,sp.Matrix(last).T)
    display(Math(sp.latex(sp.Matrix(But))))
    return But


def ordre_1(s, A, b, c, j):
    texte = r"\sum_{j=1}^s b_j =" + f"{sum(b).simplify()}"
    texte += r"\text{ doit être égale à }1"
    # display(Math(texte))
    Eq_1 = [sp.Eq( sum(b).simplify(), 1 )]
    for i in range(s):
        somma = sp.summation(A[i,j],(j,0,s-1)).simplify()
        texte = r'\sum_{j=1}^s a_{'+str(i)+r'j}=' + sp.latex( somma )
        texte += r"\text{ doit être égale à }"+sp.latex(c[i])
        # display(Math( texte ))
        Eq_1.append(sp.Eq( somma, c[i] ))
    return Eq_1


def ordre_2(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j=' +sp.latex(sum([b[i]*c[i] for i in range(s)]).simplify())
    texte += r"\text{ doit être égale à }\frac{1}{2}"
    # display(Math(texte))
    Eq_2 = sp.Eq( sum([b[i]*c[i] for i in range(s)]).simplify(), sp.Rational(1,2) )
    return Eq_2


def ordre_3(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j^2='
    texte += sp.latex( sum([b[i]*c[i]**2 for i in range(s)]).simplify() )
    texte += r"\text{ doit être égale à }\frac{1}{3}"
    # display(Math(texte))
    Eq_3_1 = sp.Eq( sum([b[i]*c[i]**2 for i in range(s)]).simplify(), sp.Rational(1,3) )
    texte = r'\sum_{i,j=1}^s b_i a_{ij} c_j='
    somma = sum([b[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{6}"
    # display(Math(texte))
```

```
    Eq_3_2 = sp.Eq( sum([b[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,6) )
    return [Eq_3_1, Eq_3_2]


def ordre_4(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j^3='
    texte += sp.latex( sum([b[i]*c[i]**3 for i in range(s)]).simplify() )
    texte += r"\text{ doit être égale à }\frac{1}{4}"
    # display(Math(texte))
    Eq_4_1 = sp.Eq( sum([b[i]*c[i]**3 for i in range(s)]).simplify(), sp.Rational(1,4) )
    texte = r'\sum_{i,j=1}^s b_i c_i a_{ij} c_j='
    somma = sum([b[i]*c[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{8}"
    # display(Math(texte))
    Eq_4_2 = sp.Eq( sum([b[i]*c[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,8) )
    texte = r'\sum_{i,j=1}^s b_i a_{ij} c_j^2='
    somma = sum([b[i]*A[i,j]*c[j]**2 for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{12}"
    # display(Math(texte))
    Eq_4_3 = sp.Eq( sum([b[i]*A[i,j]*c[j]**2 for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,12) )
    texte = r'\sum_{i,j,k=1}^s b_i a_{ij} a_{jk} c_k='
    somma = sum([b[i]*A[i,j]*A[j,k]*c[k] for k in range(s) for j in range(s) for i in
↪   range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{24}"
    # display(Math(texte))
    Eq_4_4 = sp.Eq( sum([b[i]*A[i,j]*A[j,k]*c[k] for k in range(s) for j in range(s) for i in
↪   range(s)]).simplify(), sp.Rational(1,24) )
    return [Eq_4_1, Eq_4_2, Eq_4_3, Eq_4_4]


# ============================================================================
# Calcul des conditions de consistance pour un schéma RK
# On doit indiquer le nombre d'étages s et le type de schéma
# parmi implicite, semi-implicite ou explicite
# ============================================================================
s = 2
type = "semi-implicite" # implicite, semi-implicite, explicite

A = [[1, 0], [1/sp.S(3), 0]]
c = [1, 1/sp.S(3)]
b = [1/sp.S(4),3/sp.S(4)]


display(Markdown(f"**Conditions pour un schéma {type} avec s = {s} étages**"))
Eqs = ordre_RK(s=s, type=type)
display(Markdown(f"**Pour notre exemple on trouve**"))
Eqs = ordre_RK(s=s, type=type, A=A, b=b, c=c)
```

Conditions pour un schéma semi-implicite avec s = 2 étages

Matrice de Butcher

$$\begin{bmatrix} c_0 & a_{00} & 0 \\ c_1 & a_{10} & a_{11} \\ & b_0 & b_1 \end{bmatrix}$$

On a 3 conditions pour avoir consistance (= pour être d'ordre 1)

$b_0 + b_1 = 1$

$a_{00} = c_0$

$a_{10} + a_{11} = c_1$

On doit ajouter 1 condition pour être d'ordre 2

$b_0 c_0 + b_1 c_1 = \frac{1}{2}$

On doit ajouter 2 conditions pour être d'ordre 3

$b_0 c_0^2 + b_1 c_1^2 = \frac{1}{3}$

$a_{00} b_0 c_0 + a_{10} b_1 c_0 + a_{11} b_1 c_1 = \frac{1}{6}$

On doit ajouter 4 conditions pour être d'ordre 4

$b_0 c_0^3 + b_1 c_1^3 = \frac{1}{4}$

$a_{00} b_0 c_0^2 + a_{10} b_1 c_0 c_1 + a_{11} b_1 c_1^2 = \frac{1}{8}$

$a_{00} b_0 c_0^2 + a_{10} b_1 c_0^2 + a_{11} b_1 c_1^2 = \frac{1}{12}$

$a_{00}^2 b_0 c_0 + a_{00} a_{10} b_1 c_0 + a_{10} a_{11} b_1 c_0 + a_{11}^2 b_1 c_1 = \frac{1}{24}$

Pour notre exemple on trouve

Matrice de Butcher

$$\begin{bmatrix} 1 & 1 & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 \\ & \frac{1}{4} & \frac{3}{4} \end{bmatrix}$$

On a 3 conditions pour avoir consistance (= pour être d'ordre 1)

True

True

True

On doit ajouter 1 condition pour être d'ordre 2

True

On doit ajouter 2 conditions pour être d'ordre 3

True

False

On doit ajouter 4 conditions pour être d'ordre 4

False

False

False

False

On implemente le schéma et on vérifie expérimentalement son ordre de convergence sur le problème de Cauchy $y'(t) = t^2 e^{-t} - (3t^2 + 1)y(t)$ avec $y(0) = 1$.

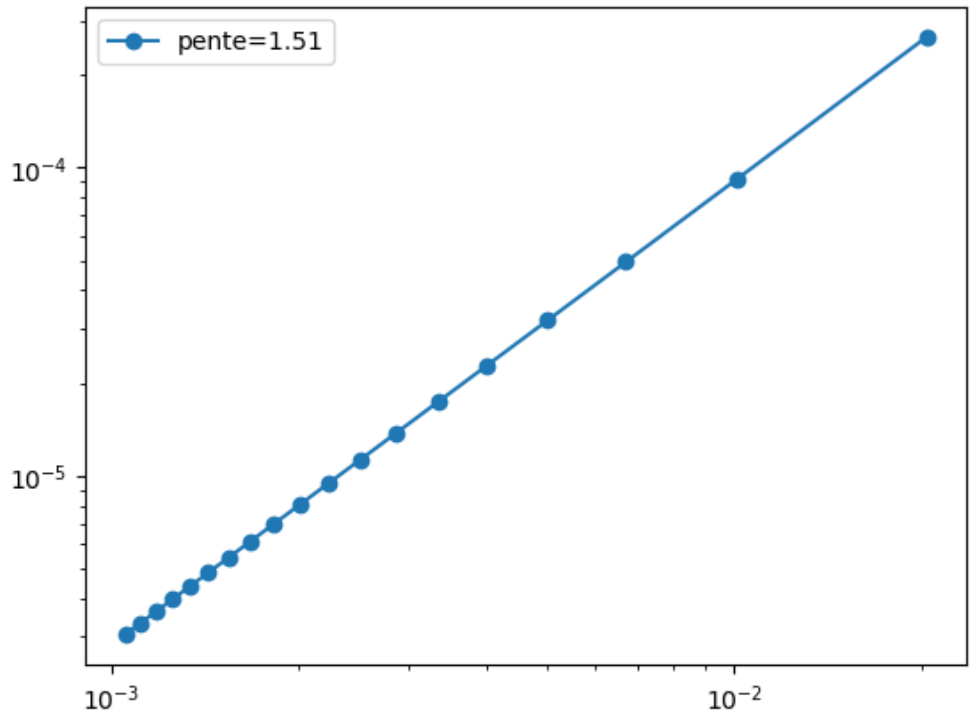
```python
def RK(phi, tt, y0):
    h = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    uu[0] = y0
    for n in range(len(tt)-1):
        k_init = phi(tt[n], uu[n])
        k1 = fsolve(lambda k: k - phi(tt[n+1], uu[n]+h*k), k_init)[0]
        k2 = phi(tt[n]+h/3, uu[n]+h*k1/3)
        uu[n+1] = uu[n] + h*(k1+3*k2)/4
    return uu


t0 = 0
y0 = 1
phi = lambda t, y: -(3*t**2+1)*y+t**2*np.exp(-t)
sol_exacte = lambda t: (2+np.exp(t**3))/3*np.exp(-t**3-t)



H, err = [], []
for N in range(50,1000,50):
    tt = np.linspace(0, 1, N)
    yy = sol_exacte(tt)
    uu = RK(phi, tt, y0)
    H.append(tt[1]-tt[0])
    err.append(np.linalg.norm(yy-uu)) # norme L2

pente = np.polyfit(np.log(H), np.log(err), 1)[0]
plt.loglog(H, err, 'o-', label=f'pente={pente:.2f}')
plt.legend();
```



21.6. EXERCICE 3 - sujet B : Schéma RK semi-implicite à 2 étages

On considère le schéma de Runge-Kutta explicite à 2 étages défini par le tableau de Butcher suivant :

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 2/3 & 1/3 & 1/3 \\ \hline & 1/4 & 3/4 \end{array}$$

Après avoir calculé l'ordre de convergence théorique, implémenter le schéma et vérifier expérimentalement son ordre de convergence sur un exemple que vous choisirez.

Correction

On calcule l'ordre de convergence théorique du schéma de Runge-Kutta à 2 étages donné.

C'est un schéma semi-implicite. D'après la barrière de Butcher, l'ordre est au plus 4.

En vérifiant les conditions de convergence, on trouve que l'ordre est 3.

```
# ============================================================================
# Fonctions pour calculer les conditions de consistance pour un schéma RK.
#
# La fonction ordre_RK calcule les conditions pour les ordres 1, 2, 3 et 4 (même si théoriquement cela
 ↪ n'est pas possible)
# - affiche la matrice de Butcher
# - affiche les conditions de consistance obtenues pour chaque ordre par la fonction ordre_i
# - renvoie une liste d'équations pour une éventuelle résolution
#
# Chaque fonction ordre_i calcule les conditions pour être d'ordre i
# - peut afficher les conditions de consistance [actuellement en commentaire]
# - renvoie la liste d'équations
#
# ============================================================================

prlat = lambda *args : display(Math(''.join( sp.latex(arg) for arg in args )))




def ordre_RK(s, type="implicite", A=None, b=None, c=None):

    j = sp.symbols('j')

    if A is None:
        if type == "implicite":
            A = sp.MatrixSymbol('a', s, s)
        elif type=="semi-implicite":
            A = sp.Matrix(s, s, lambda i,j: sp.Symbol('a{}{}'.format(i, j)) if i >= j else 0)
        elif type=="explicite":
            A = sp.Matrix(s, s, lambda i,j: sp.Symbol('a{}{}'.format(i, j)) if i > j else 0)

    else:
        A = sp.Matrix(A)
    if c is None:
        c = sp.symbols(f'c_0:{s}')
    if b is None:
        b = sp.symbols(f'b_0:{s}')



    display(Markdown("**Matrice de Butcher**"))
    But = matrice_Butcher(s, A, b, c)

    Eqs = []

    display(Markdown(f"**On a {s+1} conditions pour avoir consistance (= pour être d'ordre 1)**"))
    Eq_1 = ordre_1(s, A, b, c, j)
    Eqs.append(Eq_1)
```

```
    display(*Eq_1)

    display(Markdown("**On doit ajouter 1 condition pour être d'ordre 2**"))
    Eq_2 = ordre_2(s, A, b, c, j)
    Eqs.append([Eq_2])
    display(Eq_2)

    display(Markdown("**On doit ajouter 2 conditions pour être d'ordre 3**"))
    Eq_3 = ordre_3(s, A, b, c, j)
    Eqs.append(Eq_3)
    display(*Eq_3)

    display(Markdown("**On doit ajouter 4 conditions pour être d'ordre 4**"))
    Eq_4 = ordre_4(s, A, b, c, j)
    Eqs.append(Eq_4)
    display(*Eq_4)

    return Eqs


def matrice_Butcher(s, A, b, c):
    But = sp.Matrix(A)
    But = But.col_insert(0, sp.Matrix(c))
    last = [sp.Symbol(" ")]
    last.extend(b)
    But = But.row_insert(s,sp.Matrix(last).T)
    display(Math(sp.latex(sp.Matrix(But))))
    return But


def ordre_1(s, A, b, c, j):
    texte = r"\sum_{j=1}^s b_j =" + f"{sum(b).simplify()}"
    texte += r"\text{ doit être égale à }1"
    # display(Math(texte))
    Eq_1 = [sp.Eq( sum(b).simplify(), 1 )]
    for i in range(s):
        somma = sp.summation(A[i,j],(j,0,s-1)).simplify()
        texte = r'\sum_{j=1}^s a_{'+str(i)+r'j}=' + sp.latex( somma )
        texte += r"\text{ doit être égale à }"+sp.latex(c[i])
        # display(Math( texte ))
        Eq_1.append(sp.Eq( somma, c[i] ))
    return Eq_1


def ordre_2(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j=' +sp.latex(sum([b[i]*c[i] for i in range(s)]).simplify())
    texte += r"\text{ doit être égale à }\frac{1}{2}"
    # display(Math(texte))
    Eq_2 = sp.Eq( sum([b[i]*c[i] for i in range(s)]).simplify(), sp.Rational(1,2) )
    return Eq_2


def ordre_3(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j^2='
    texte += sp.latex( sum([b[i]*c[i]**2 for i in range(s)]).simplify() )
    texte += r"\text{ doit être égale à }\frac{1}{3}"
    # display(Math(texte))
    Eq_3_1 = sp.Eq( sum([b[i]*c[i]**2 for i in range(s)]).simplify(), sp.Rational(1,3) )
    texte = r'\sum_{i,j=1}^s b_i a_{ij} c_j='
    somma = sum([b[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{6}"
    # display(Math(texte))
```

```
    Eq_3_2 = sp.Eq( sum([b[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,6) )
    return [Eq_3_1, Eq_3_2]


def ordre_4(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j^3='
    texte += sp.latex( sum([b[i]*c[i]**3 for i in range(s)]).simplify() )
    texte += r"\text{ doit être égale à }\frac{1}{4}"
    # display(Math(texte))
    Eq_4_1 = sp.Eq( sum([b[i]*c[i]**3 for i in range(s)]).simplify(), sp.Rational(1,4) )
    texte = r'\sum_{i,j=1}^s b_i c_i a_{ij} c_j='
    somma = sum([b[i]*c[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{8}"
    # display(Math(texte))
    Eq_4_2 = sp.Eq( sum([b[i]*c[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,8) )
    texte = r'\sum_{i,j=1}^s b_i a_{ij} c_j^2='
    somma = sum([b[i]*A[i,j]*c[j]**2 for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{12}"
    # display(Math(texte))
    Eq_4_3 = sp.Eq( sum([b[i]*A[i,j]*c[j]**2 for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,12) )
    texte = r'\sum_{i,j,k=1}^s b_i a_{ij} a_{jk} c_k='
    somma = sum([b[i]*A[i,j]*A[j,k]*c[k] for k in range(s) for j in range(s) for i in
↪   range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{24}"
    # display(Math(texte))
    Eq_4_4 = sp.Eq( sum([b[i]*A[i,j]*A[j,k]*c[k] for k in range(s) for j in range(s) for i in
↪   range(s)]).simplify(), sp.Rational(1,24) )
    return [Eq_4_1, Eq_4_2, Eq_4_3, Eq_4_4]


# ============================================================================
# Calcul des conditions de consistance pour un schéma RK
# On doit indiquer le nombre d'étages s et le type de schéma implicite, semi-implicite ou explicite
# ============================================================================
s = 2
type = "semi-implicite" # implicite, semi-implicite, explicite
A = [[0, 0], [1/sp.S(3), 1/sp.S(3)]]
b = [1/sp.S(4), 3/sp.S(4)]
c = [0, 2/sp.S(3)]

display(Markdown(f"**Conditions pour un schéma {type} avec s = {s} étages**"))
Eqs = ordre_RK(s=s, type=type, A=A, b=b, c=c)
```

Conditions pour un schéma semi-implicite avec s = 2 étages

Matrice de Butcher

$$\begin{bmatrix} 0 & 0 & 0 \\ \frac{2}{3} & \frac{1}{3} & \frac{1}{3} \\ & \frac{1}{4} & \frac{3}{4} \end{bmatrix}$$

On a 3 conditions pour avoir consistance (= pour être d'ordre 1)

True

True

True

On doit ajouter 1 condition pour être d'ordre 2

True

On doit ajouter 2 conditions pour être d'ordre 3

True

True

On doit ajouter 4 conditions pour être d'ordre 4

False

False

False

False

On implemente le schéma et on vérifie expérimentalement son ordre de convergence sur le problème de Cauchy $y'(t) = t^2e^{-t} - (3t^2+1)y(t)$ avec $y(0) = 1$.

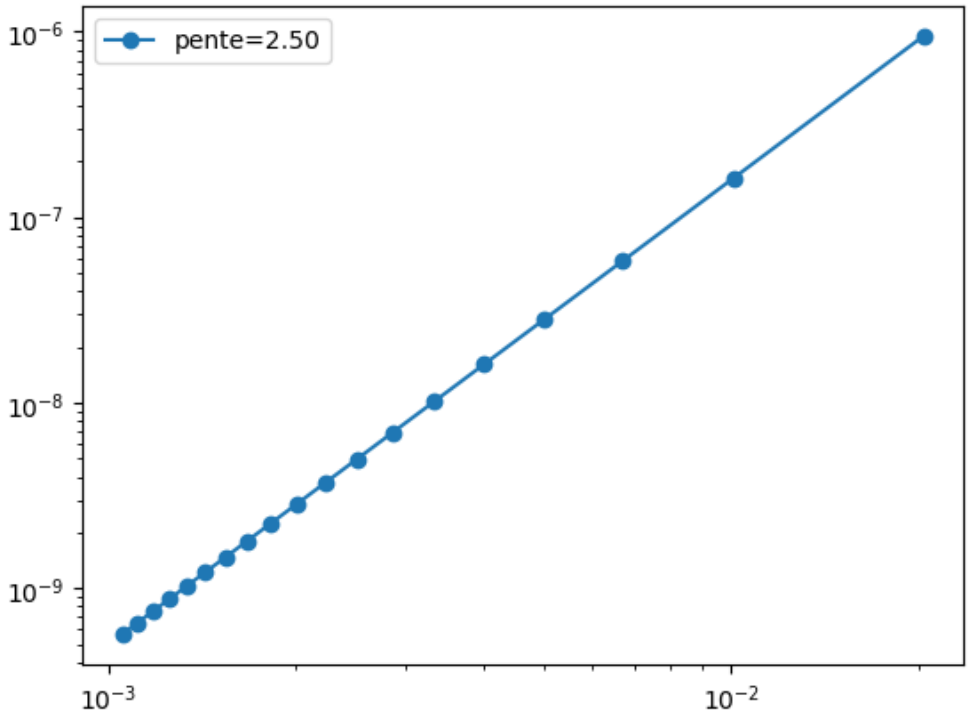
```
def RK(phi, tt, y0):
    h = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    uu[0] = y0
    for n in range(len(tt)-1):
        k1 = phi(tt[n], uu[n])
        k2 = fsolve(lambda k: k - phi(tt[n]+h*2/3, uu[n]+h*(k1+k)/3), phi(tt[n], uu[n]))[0]
        uu[n+1] = uu[n] + h*(k1+3*k2)/4
    return uu


t0 = 0
y0 = 1
phi = lambda t, y: -(3*t**2+1)*y+t**2*np.exp(-t)
sol_exacte = lambda t: (2+np.exp(t**3))/3*np.exp(-t**3-t)



H, err = [], []
for N in range(50,1000,50):
    tt = np.linspace(0, 1, N)
    yy = sol_exacte(tt)
    uu = RK(phi, tt, y0)
    H.append(tt[1]-tt[0])
    err.append(np.linalg.norm(yy-uu)) # norme L2

pente = np.polyfit(np.log(H), np.log(err), 1)[0]
plt.loglog(H, err, 'o-', label=f'pente={pente:.2f}')
plt.legend();
```


pente=2.50
10^{-6}
10^{-7}
10^{-8}
10^{-9}
10^{-3}
10^{-2}

Chapitre 22.

2025 Contrôle Terminale session 1

2h, déposer le notebook dans Moodle (si 1/3 temps supplémentaire, 2h40)

Autorisés : notebooks de cours, notes personnelles.

Compléter ici :

- NOM :
- Prénom :
- N° étudiant :

```
import numpy as np
import matplotlib.pyplot as plt

from scipy.optimize import fsolve
from scipy.integrate import solve_ivp, odeint

import sympy as sp

from IPython.display import display, Markdown
```

22.1. EXERCICE 1 : Schéma multi-pas linéaire explicite

1. Déterminer les **coefficients** d'un schéma multi-pas linéaire **explicite** à **deux pas** sous les conditions suivantes :
   - le premier polynôme caractéristique associé admet $r = \frac{1}{6}$ comme racine ;
   - l'ordre de convergence du schéma est supérieur ou égal à 2.
2. Considérer le problème de Cauchy suivant :

$$\begin{cases} y'(t) = \frac{y^2(t)}{t}, & t \in [1, t_{\max}], \\ y(1) = 2. \end{cases}$$

   Déterminer $t_{\max}$. (*Suggéstion :* calculer la solution exacte de ce problème de Cauchy).
3. Implémenter le schéma déterminé à la question 1 pour résoudre le problème de Cauchy. Vérifier expérimentalement l'**ordre** de convergence du schéma sur l'intervalle $[1, \frac{1+t_{\max}}{2}]$.

Correction

Un schéma multi-pas linéaire explicite à $q = 2$ pas s'écrit comme

$$u_{n+1} = a_0 u_n + a_1 u_{n-1} + h\Big(b_0 \varphi_n + b_1 \varphi_{n-1}\Big).$$

Le premier polynomial caractéristique est donné par

$$\varrho(r) = r^2 - a_0 r - a_1.$$

Le schéma est convergent, ainsi $r = 1$ est une racine. On sait aussi que $r = \frac{1}{6}$ est une racine, donc

$$\varrho(r) = (r-1)\left(r - \frac{1}{6}\right) = r^2 - \frac{7}{6}r + \frac{1}{6}.$$

Par identification des coefficients, on trouve $a_0 = \frac{7}{6}$ et $a_1 = -\frac{1}{6}$.

Il ne reste qu'à trouver les coefficients b_0 et b_1. Le schéma est convergent à l'ordre au moins 2 ssi $\xi(0) = \xi(1) = \xi(2) = 1$. La condition $\xi(0) = 1$ est déjà satisfaite car $r = 1$ est une racine du polynôme caractéristique. On a alors les deux équations

$$\begin{cases} -a_1 + b_0 + b_1 = 1 \\ a_1 - 2b_1 = 1 \end{cases} \quad \text{soit} \quad \begin{cases} \frac{1}{6} + b_0 + b_1 = 1 \\ -\frac{1}{6} - 2b_1 = 1 \end{cases} \quad \text{ainsi} \quad \begin{cases} b_0 = \frac{17}{12} \\ b_1 = -\frac{7}{12} \end{cases}$$

Conclusion : un seul schéma satisfait à toutes les conditions, il s'agit du schéma

$$u_{n+1} = \frac{7}{6}u_n - \frac{1}{6}u_{n-1} + \frac{h}{12}\left(17\varphi_n - 7\varphi_{n-1}\right).$$

```
# Vérifions nos calculs avec sympy
# ================================

a0, a1 = sp.symbols('a0 a1')
b0, b1  = sp.symbols('b0 b1')
r = sp.symbols('r')
Eq0 = sp.Eq( (r-1)*(r-1/sp.S(6)) , r**2 - a0*r-a1 )
aa = sp.solve(Eq0, (a0,a1))
for k,v in aa.items():
    display(sp.Eq(k,v))
a0 = aa[a0]
a1 = aa[a1]
Eq1 = sp.Eq(-a1 + b0+ b1, 1)
Eq2 = sp.Eq(a1 -2* b1, 1)
sol = sp.solve((Eq1, Eq2), (b0, b1))
for k,v in sol.items():
    display(sp.Eq(k,v))
```

$$a_0 = \frac{7}{6}$$

$$a_1 = -\frac{1}{6}$$

$$b_0 = \frac{17}{12}$$

$$b_1 = -\frac{7}{12}$$

Calculons la solution exacte du problème de Cauchy. L'EDO est à variables séparables : $\frac{dy}{y^2} = \frac{dt}{t}$. En intègrant de chaque côté on trouve $-\frac{1}{y} = \ln|t| + C$. Avec $y(1) = 2$ on trouve la constante d'intégration : $-\frac{1}{2} = \ln(1) + C \rightsquigarrow C = -\frac{1}{2}$. Donc $y(t) = \frac{1}{\frac{1}{2} - \ln(t)}$.

La solution exacte du problème de Cauchy est donnée par $y(t) = \frac{2}{1 - 2\log(t)}$ qui n'est pas définie en $t = \sqrt{e}$. Ainsi, $t_{\max} = \sqrt{e}$.

```
# ================================================
# TEST
# ================================================

phi = lambda t,y : y**2/t
t0, y0 = 1, 2


# Solution exacte
# ---------------
t = sp.symbols('t')
y = sp.Function('y')(t)
eq = sp.Eq(y.diff(t), phi(t, y))
display(eq)
ics = {y.subs(t, t0): y0 }
display(sp.Eq(y.subs(t, t0), y0))
sol = sp.dsolve(eq, y, ics=ics)
display(Markdown(f"La solution exacte est"))
display(sol)

display(Markdown(r"On en déduit que $t < \sqrt{e}$"))

# Transfomation en fonction numpy
# -------------------------------

sol = sp.lambdify(t, sol.rhs, 'numpy')
tfin = (t0+np.sqrt(np.exp(1)))/2
```

$$\frac{d}{dt}y(t) = \frac{y^2(t)}{t}$$

$$y(1) = 2$$

La solution exacte est

$$y(t) = -\frac{1}{\log(t) - \frac{1}{2}}$$

On en déduit que $t < \sqrt{e}$

```
def MPL(phi, tt, sol_exacte) :
    uu = np.zeros_like(tt)
    h = tt[1]-tt[0]
    q = 2
    uu[:q] = sol_exacte(tt[:q])
    for n in range(q-1, len(tt)-1) :
        k0 = phi(tt[n], uu[n])
        k1 = phi(tt[n-1], uu[n-1])
        uu[n+1] = (7*uu[n]-uu[n-1])/6 + h*(17*k0-7*k1)/12
    return uu

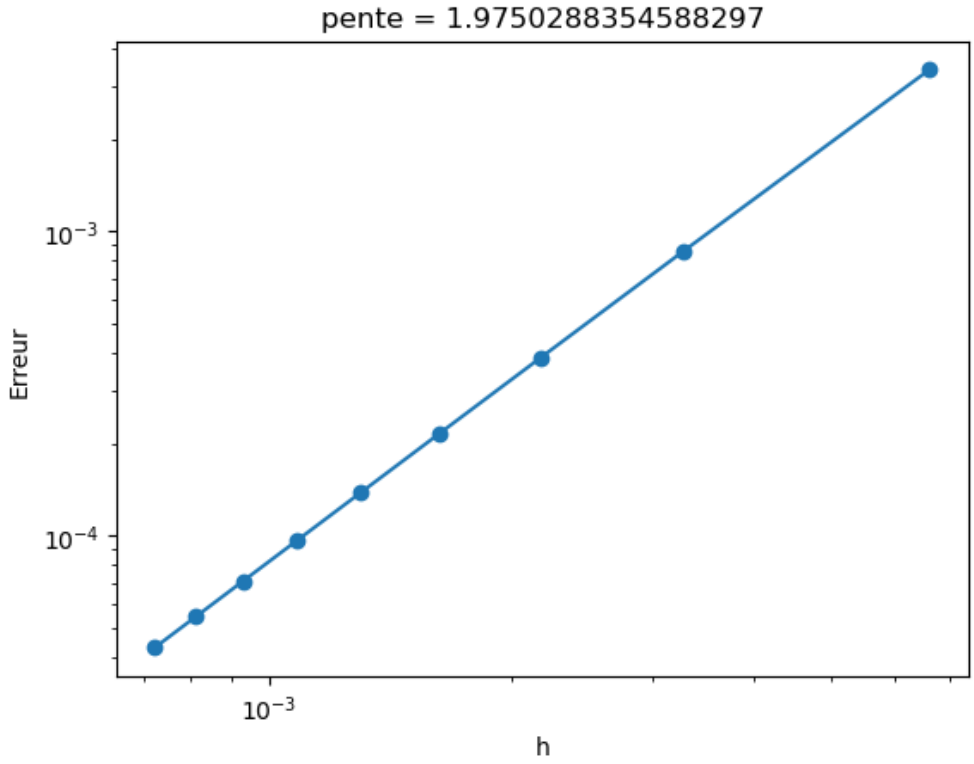


# Solution numérique : vérification de l'implémentation
# -----------------------------------------------------
# tt = np.linspace(t0, tfin, 100)
# uu = MPL(phi, tt, sol)
# plt.plot(tt, uu, label='MPL')
# plt.plot(tt, sol(tt), label='Solution exacte')
# plt.legend();

# Calcul de l'ordre de convergence
# --------------------------------
```

```
H, err = [], []
for N in range(50,500,50) :
    tt = np.linspace(t0, tfin, N)
    uu = MPL(phi, tt, sol)
    err.append(np.linalg.norm(uu-sol(tt), np.inf))
    H.append(tt[1]-tt[0])

pente = np.polyfit(np.log(H), np.log(err), 1)[0]
plt.loglog(H, err, 'o-')
plt.title(f"{pente = }")
plt.xlabel('h')
plt.ylabel('Erreur');
```



22.2. EXERCICE 2 : Étude de l'A-stabilité d'un schéma RK explicite à 3 étages

On considère le schéma de Runge-Kutta explicite à 3 étages défini par le tableau de Butcher suivant :

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 \\ 1 & -1 & 2 & 0 \\ \hline & 1/6 & 2/3 & 1/6 \end{array}$$

On considère le problème de Cauchy $y'(t) = -\beta y(t)$ et $y(0) = 1$ avec $\beta > 0$.

1. Déterminer les valeurs de h pour lesquelles le schéma est **A-stable**. On notera $h_A(\beta)$ la plus grande valeur h assurant la A-stabilité.
2. Déterminer les valeurs de h garantissant que le schéma produit une **solution monotone**. On notera $h_M(\beta)$ la plus grande valeur de h assurant la monotonie.
3. Implémenter le schéma et vérifier expérimentalement son A-stabilité en prenant $h = h_M(\beta)/2$.

Le schéma est donné par

$$
\begin{aligned}
k_1 &= \varphi\Big(t_n, u_n\Big) \\
k_2 &= \varphi\Big(t_n + \frac{1}{2}h, u_n + \frac{1}{2}hk_1\Big) \\
k_3 &= \varphi\Big(t_n + h, u_n + h(-k_1 + 2k_2)\Big) \\
u_{n+1} &= u_n + \frac{1}{6}h\Big(k_1 + 4k_2 + k_3\Big)
\end{aligned}
$$

On applique le schéma au problème de Cauchy $y'(t) = -\beta y(t)$ et $y(0) = 1$ avec $\beta > 0$ dont la solution est $y(t) = e^{-\beta t}$. Avec $\varphi(t, y) = -\beta y$, on obtient alors la relation de récurrence suivante :

$$
\begin{aligned}
k_1 &= -\beta u_n \\
k_2 &= -\beta\Big(u_n + \frac{1}{2}hk_1\Big) = -\Big(1 - \frac{1}{2}\beta h\Big)\beta u_n \\
k_3 &= -\beta\Big(u_n + h(-k_1 + 2k_2)\Big) = -\Big(\beta^2 h^2 - \beta h + 1\Big)\beta u_n \\
u_{n+1} &= u_n + \frac{1}{6}h\Big(k_1 + 4k_2 + k_3\Big) = -\frac{(\beta h)^3}{6} + \frac{(\beta h)^2}{2} - (\beta h) + 1
\end{aligned}
$$

On notera $x = \beta h > 0$ et $q(x)$ le facteur d'amplification du schéma :

$$
q(x) = \frac{u_{n+1}}{u_n} = -\frac{x^3}{6} + \frac{x^2}{2} - x + 1
$$

La suite vérifie $u_n \xrightarrow[n\to\infty]{} 0$ ssi $|q(x)| < 1$. En étudiant brièvement la fonction $q(x)$, on trouve que q est une fonction monotone décroissante avec $q(0) = 1$ et $q(x) \xrightarrow[x\to+\infty]{} -\infty$. On cherche alors les solutions de $q(x) + 1 = 0$.

Il n'est pas possible de résoudre analytiquement cette équation, on utilisera donc `fsolve` pour déterminer une valeur approchée de x pour laquelle le schéma est A-stable.

De même, le schéma génère une solution monotone ssi $q(x) > 0$. On cherche alors numériquement les solutions de $q(x) = 0$.

```
q_func = lambda x: -x**3/6+x**2/2-x+1

x = np.linspace(0,4,101)
plt.plot(x, q_func(x))
plt.ylim(-2, 2) # On zoome sur la zone intéressante [-1, 1]
plt.xlim(x.min(), x.max())
plt.axhline(y=1, color='black', linestyle='-', alpha=0.5)
plt.axhline(y=-1, color='black', linestyle='-', alpha=0.5)
plt.fill_between(x, -1, 1, color='gray', alpha=0.1, label='Zone de A-Stabilité')
plt.hlines(0,0,4, color='red')
plt.ylim(-1.5,1.5)
plt.xlim(0,4)
plt.grid()
plt.xlabel('x')
plt.ylabel('q(x)')
plt.title('q(x) en fonction de x')

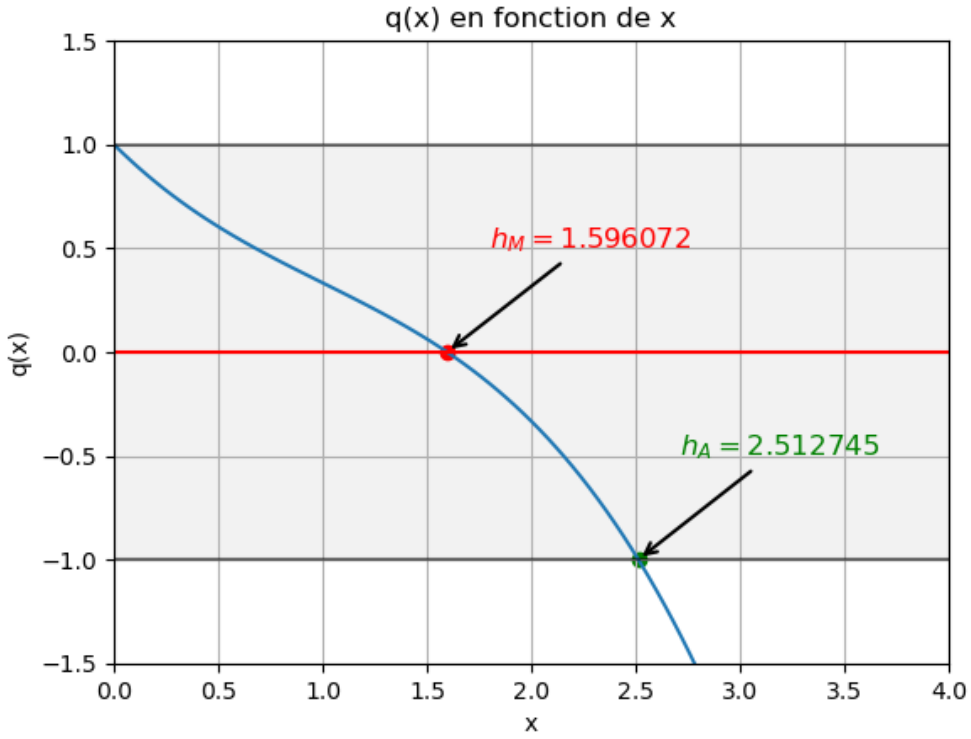
h_A = fsolve(lambda x: q_func(x)+1, 2)[0]
h_M = fsolve(lambda x: q_func(x), 1)[0]

plt.scatter(h_A, -1, color='green')
plt.annotate(f'$h_A = {h_A:.6f}$', xy=(h_A, -1), xytext=(h_A+0.2, -0.5), arrowprops=dict(arrowstyle='->',
 ↪  lw=1.5), fontsize=12, color='green')
plt.scatter(h_M, 0, color='red')
plt.annotate(f'$h_M = {h_M:.6f}$', xy=(h_M, 0), xytext=(h_M+0.2, 0.5), arrowprops=dict(arrowstyle='->',
 ↪  lw=1.5), fontsize=12, color='red')
```

```
display(Markdown(f"La condition de A-stabilité est satisfaite pour $\\beta h < {h_A:.6f}$"))
display(Markdown(f"La condition de monotonie est satisfaite pour $\\beta h < {h_M:.6f}$"))
```

La condition de A-stabilité est satisfaite pour $\beta h < 2.512745$

La condition de monotonie est satisfaite pour $\beta h < 1.596072$



On peut vérifier nos calculs avec sympy :

```
b = [1/sp.S(6), 2/sp.S(3), 1/sp.S(6)]
c = [0,1/sp.S(2),1]
A = [[0, 0, 0], [1/sp.S(2), 0, 0], [-1, 2, 0]]

s = len(c)

u_n = sp.symbols(r'u_n')
u_np1 = sp.symbols(r'u_{n+1}')

dt = sp.symbols(r'h', positive=True)
x = sp.symbols(r'x', positive=True)

k_1 = sp.symbols(r'k_1')
k_2 = sp.symbols(r'k_2')
k_3 = sp.symbols(r'k_3')
kk = [k_1, k_2, k_3]

beta = sp.symbols(r'\beta', positive=True)
phi = lambda y: -beta*y

display(Markdown("**Les équations**"))
Eqs = [ sp.Eq(kk[i] ,  phi(u_n + dt*sum([kk[j]*A[i][j] for j in range(s)]))) for i in range(s) ]
for eq in Eqs:
    display(eq)

display(Markdown("**Ce qui donne**"))
```

```
sol = sp.solve( Eqs, kk ) # c'est un dictionnaire
KK = [sol[kk[i]] for i in range(s)]
for i,k in enumerate(KK):
    display(sp.Eq(kk[i]/u_n, k.factor()/u_n))

display(Markdown("**Enfin**"))
schema = sp.Eq( u_np1/u_n , (u_n + dt*sum([KK[j]*b[j] for j in range(s)])).factor()/u_n )
display(schema)


display(Markdown("**Le facteur d'amplification**"))
q = (schema.rhs).simplify().subs(beta*dt,x)
q_func = sp.lambdify(x, q, modules=['numpy'])
display(sp.Eq(sp.Symbol('q(x)'), q.factor()))

# sp.plot(q, (x,0,10), ylim=(-1.5,1.5),
#             ylabel='$q(x)$', xlabel='$x$')

# # sp.solve(abs(q)<1, x)
display(Markdown(f"La **condition de A-stabilité** est satisfaite pour $\\beta h < {sp.nsolve(abs(q)-1,
 ↪  x, 2):.6f}$"))
display(Markdown(f"La **condition de monotonie** est satisfaite pour $\\beta h < {sp.nsolve(abs(q), x,
 ↪  2):.6f}$"))
```

Les équations

$$k_1 = -\beta u_n$$

$$k_2 = -\beta\left(\frac{hk_1}{2} + u_n\right)$$

$$k_3 = -\beta\left(h\left(-k_1 + 2k_2\right) + u_n\right)$$

Ce qui donne

$$\frac{k_1}{u_n} = -\beta$$

$$\frac{k_2}{u_n} = \frac{\beta\left(\beta h - 2\right)}{2}$$

$$\frac{k_3}{u_n} = -\beta\left(\beta^2 h^2 - \beta h + 1\right)$$

Enfin

$$\frac{u_{n+1}}{u_n} = -\frac{\beta^3 h^3}{6} + \frac{\beta^2 h^2}{2} - \beta h + 1$$

Le facteur d'amplification

$$q(x) = -\frac{x^3 - 3x^2 + 6x - 6}{6}$$

La **condition de A-stabilité** est satisfaite pour $\beta h < 2.512745$

La **condition de monotonie** est satisfaite pour $\beta h < 1.596072$

Vérifions empiriquement la stabilité du schéma pour $\beta = 1$.

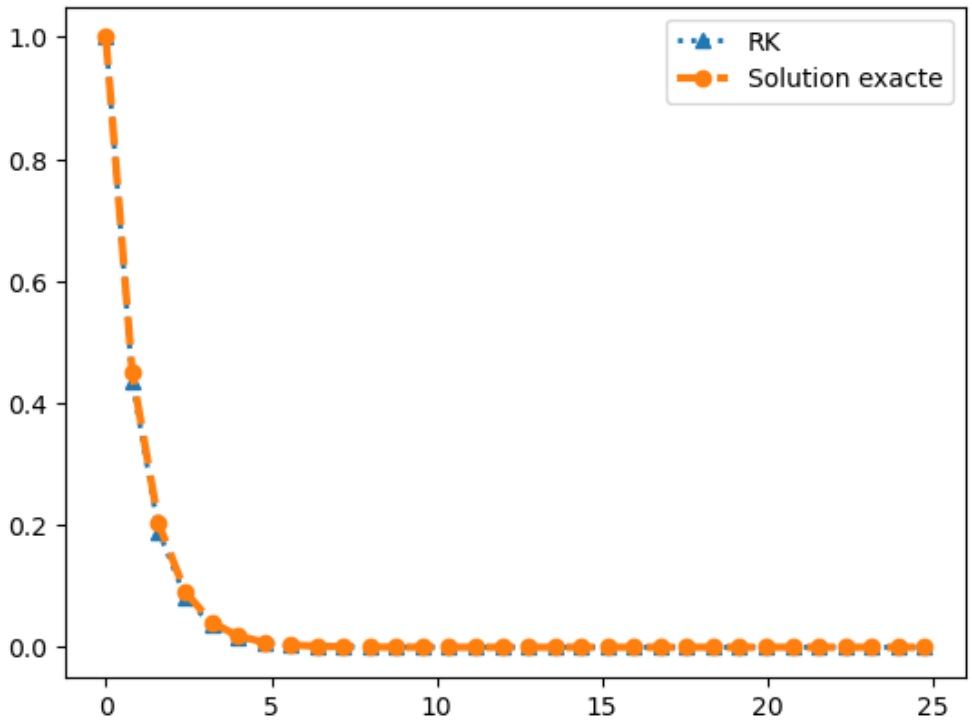
```
def RK(phi, tt, yy):
    h = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    uu[0] = yy[0]
    for n in range(len(tt)-1):
        k1 = phi(tt[n], uu[n])
        k2 = phi(tt[n] + h/2, uu[n] + h/2*k1)
```

```
        k3 = phi(tt[n] + h, uu[n] - h*k1 + 2*h*k2)
        uu[n+1] = uu[n] + h*(k1 + 4*k2 + k3)/6
    return uu


sol_exacte = lambda t: np.exp(-t)
phi = lambda t, y: -y

tt_M = np.arange(0, 25, h_M/2)
yy_M = sol_exacte(tt_M)
uu_M = RK(phi, tt_M, yy_M)

plt.plot(tt_M, uu_M, '^:', label='RK', lw=2)
plt.plot(tt_M, yy_M, 'o--', label='Solution exacte', lw=3)
plt.legend()
plt.show()
```



22.3. EXERCICE 3 : schéma prédicteur-correcteur

- **Schéma Prédicteur-Correcteur (PC)** : Écrire (en Markdown) le schéma Prédicteur-Correcteur en utilisant comme prédicteur le schéma d'Euler explicite (**EE**) et comme correcteur le schéma BDF à deux pas (**BDF2**).
- **Ordre de convergence théorique** : Quel est l'ordre de convergence de ce schéma PC ?
- **Implementation** : Implémenter le schémas EE, BDF2 et PC dans les fonction `PRED`, `CORR` et `PRED_CORR` prénant en entrée les paramètres suivants :
  - `phi` : la fonction φ du problème de Cauchy
  - `tt` : le vecteur des instants d'évaluation
  - `yy` : le vecteur des valeurs de la solution exacte aux temps t_0 et t_1 et qui renvoie `uu` le vecteur des valeurs de la solution approchée aux instants d'évaluation `tt`.
- **Ordre de convergence numérique** : Vérifier empiriquement l'ordre de convergence de ces 3 schémas

(EE, BDF2, PC) sur le problème de Cauchy suivant :

$$\begin{cases} y'(t) = \arctan(5(1-t))y(t) \quad \text{pour } t \in]0,3] \\ y(0) = \sqrt[10]{26} \end{cases}$$

Correction

Le schéma PC est donné par

$$\begin{cases} u_0 = y_0 \\ u_1 = y_1 \\ \tilde{u}_{n+1} = u_n + h\varphi(t_n, u_n) \\ u_{n+1} = \frac{1}{3}\Big(4u_n - u_{n-1} + 2h\varphi(t_{n+1}, \tilde{u}_{n+1})\Big) \end{cases}$$

Ce schéma est explicite d'ordre 2, car le prédicteur est d'ordre 1 et le correcteur est d'ordre 2.

Le code suivant montre comment implémenter les schémas d'Euler explicite, BDF2 et Prédicteur-Correcteur, et vérifie leur ordre de convergence.

- PRED : Implémentation du schéma prédicteur (Euler explicite).
- CORR : Implémentation du schéma correcteur (BDF2).
- PRED_CORR : Implémentation du schéma Prédicteur-Correcteur combinant les deux.
- Test 1 : Résolution du problème de Cauchy pour vérifier visuellement la précision des schémas.
- Test 2 : Vérification de l'ordre de convergence en calculant les erreurs pour différentes tailles de pas et en ajustant la pente pour déterminer l'ordre de convergence empirique.

```
# Schéma Prédicteur : Euler explicite
def PRED(phi, tt, yy):
    h = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    uu[0] = yy[0]  # conditions initiales
    for n in range(len(tt)-1):
        k1 = phi(tt[n], uu[n])
        uu[n+1] = uu[n] + h * k1  # Schéma Euler explicite
    return uu

# Schéma Correcteur : BDF2
def CORR(phi, tt, yy):
    h = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    uu[0] = yy[0]  # conditions initiales
    uu[1] = yy[1]
    for n in range(1, len(tt)-1):
        temp = fsolve(lambda x: -x + 4/3*uu[n] - 1/3*uu[n-1] + 2/3*h*phi(tt[n+1], x), uu[n])
        uu[n+1] = temp[0]  # Correcteur BDF2
    return uu

# Schéma Prédicteur-Correcteur (PC)
def PRED_CORR(phi, tt, yy):
    h = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    uu[0] = yy[0]  # conditions initiales
    uu[1] = yy[1]
    for n in range(1, len(tt)-1):
        k1 = phi(tt[n], uu[n])
        k2 = phi(tt[n-1], uu[n-1])
        pred = uu[n] + h*k1  # Prédicteur
        uu[n+1] = 4/3*uu[n] - 1/3*uu[n-1] + 2/3*h*phi(tt[n+1], pred)  # Correcteur
    return uu
```

```
# Résolution symbolique du problème de Cauchy donné
# ================================================================================

t = sp.symbols('t')
y = sp.Function('y')(t)
phi_sym = lambda t, y: sp.atan(5*(1-t))*y
t0, y0 = 0, 26**(1/sp.S(10))
sol = sp.dsolve(y.diff(t) - phi_sym(t, y), y, ics={y.subs(t, t0): y0})
display(sol)

# Conversion en fonction numérique
y0 = float(y0)
phi = sp.lambdify((t, y), phi_sym(t, y), 'numpy')
sol_exacte = sp.lambdify(t, sol.rhs, 'numpy')

tfin = 3


# Test 1 : Résolution numérique du problème de Cauchy donné
# ================================================================================

# N = 20
# tt = np.linspace(t0, tfin, N+1)
# yy = sol_exacte(tt)

# uu_PRED = PRED(phi, tt, yy)
# uu_CORR = CORR(phi, tt, yy)
# uu_PRED_CORR = PRED_CORR(phi, tt, yy)

# # Affichage des résultats
# plt.plot(tt, yy, 'o--', label='Solution exacte', lw=3)
# plt.plot(tt, uu_PRED, 'v--', label='Prédicteur', lw=2)
# plt.plot(tt, uu_CORR, '^:', label='Correcteur', lw=2)
# plt.plot(tt, uu_PRED_CORR, 'd--', label='Prédicteur-Correcteur', lw=2)
# plt.legend()
# plt.show()

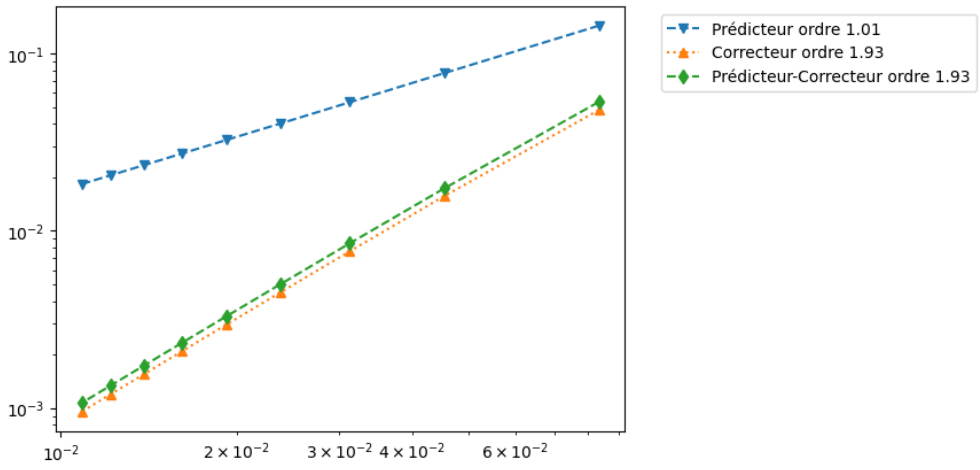
# Test 2 : Ordre de convergence
# ================================================================================
H = []
err_PRED = []
err_CORR = []
err_PRED_CORR = []
for N in range(50,500,50):
    N += 10
    tt = np.linspace(0, 5, N+1)
    H.append(tt[1] - tt[0])
    yy = sol_exacte(tt)
    uu_PRED = PRED(phi, tt, yy)
    uu_CORR = CORR(phi, tt, yy)
    uu_PRED_CORR = PRED_CORR(phi, tt, yy)
    err_PRED.append(np.linalg.norm(yy - uu_PRED, np.inf))
    err_CORR.append(np.linalg.norm(yy - uu_CORR, np.inf))
    err_PRED_CORR.append(np.linalg.norm(yy - uu_PRED_CORR, np.inf))

# Calcul des pentes (ordres de convergence)
pente_PRED = np.polyfit(np.log(H), np.log(err_PRED), 1)[0]
pente_CORR = np.polyfit(np.log(H), np.log(err_CORR), 1)[0]
pente_PRED_CORR = np.polyfit(np.log(H), np.log(err_PRED_CORR), 1)[0]

# Affichage de l'ordre de convergence
```

```
plt.loglog(H, err_PRED, 'v--', label=f'Prédicteur ordre {pente_PRED:.2f}')
plt.loglog(H, err_CORR, '^:', label=f'Correcteur ordre {pente_CORR:.2f}')
plt.loglog(H, err_PRED_CORR, 'd--', label=f'Prédicteur-Correcteur ordre {pente_PRED_CORR:.2f}')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()
```

$$y(t) = \sqrt[10]{25t^2 - 50t + 26}e^{-(t-1)\operatorname{atan}(5t-5)}e^{\operatorname{atan}(5)}$$



Chapitre 23.

2025 Contrôle Terminale session 2

2h, déposer le notebook dans Moodle (si 1/3 temps supplémentaire, 2h40)

Autorisés : notebooks de cours, notes personnelles.

Compléter ici :

- NOM :
- Prénom :
- N° étudiant :

```
import numpy as np
import matplotlib.pyplot as plt

from scipy.optimize import fsolve
from scipy.integrate import solve_ivp, odeint

import sympy as sp

from IPython.display import display, Markdown, Math
```

23.1. EXERCICE 1 : schéma RK explicite à 3 étage

On considère le schéma de Runge-Kutta explicite à 3 étages défini par le tableau de Butcher suivant :

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{array}$$

1. **Implémenter** le schéma
2. Calculer son **ordre** de convergence et le vérifier expérimentalement sur le problème de Cauchy $y'(t) = -\frac{y(t)}{t} - y(t)^2 \ln(t)$, $y(1) = 1$.
3. Déterminer les valeurs de h pour lesquelles le schéma est **A-stable**. On notera $h_A(\beta)$ la plus grande valeur h assurant la A-stabilité.
4. Déterminer les valeurs de h garantissant que le schéma produit une **solution monotone**. On notera $h_M(\beta)$ la plus grande valeur de h assurant la monotonie.

Correction

C'est une méthode de Runge-Kutta explicite à $s = 3$ étages. Elle s'écrit comme

$$\begin{cases} k_1 = \varphi(t_n, u_n) \\ k_2 = \varphi(t_n + \frac{h}{2}, u_n + \frac{h}{2}k_1) \\ k_3 = \varphi(t_n + h, u_n + hk_2) \\ u_{n+1} = u_n + \frac{h}{6}(k_1 + 4k_2 + k_3) \end{cases}$$

```
# IMPLEMENTATION
# ================================================================================

def RK(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.zeros(len(tt))
    uu[0] = y0
    for n in range(len(tt)-1):
        k1 = phi(tt[n],uu[n])
        k2 = phi(tt[n]+h/2,uu[n]+h/2*k1)
        k3 = phi(tt[n]+h,uu[n]+h*k2)
        uu[n+1] = uu[n] + h/6*(k1+4*k2+k3)
    return uu
```

La barrière de Butcher donne une borne supérieure pour l'ordre de convergence : étant une méthode explicite et $s < 5$, l'ordre vérifie $\omega \leq s$ donc $\omega \leq 3$.

On peut calculer explicitement l'ordre de convergence avec sympy et on trouve qu'il n'est que d'ordre 2 :

```
prlat = lambda *args : display(Math(''.join( sp.latex(arg) for arg in args )))

def ordre_RK(s, A=None, b=None, c=None):

    j = sp.symbols('j')

    if A is None: A = sp.MatrixSymbol('a', s, s)
    else:         A = sp.Matrix(A)
    if c is None: c = sp.symbols(f'c_0:{s}')
    if b is None: b = sp.symbols(f'b_0:{s}')

    display(Markdown("**Matrice de Butcher**"))
    But = matrice_Butcher(s, A, b, c)

    Eqs = []

    display(Markdown(f"**On a {s+1} conditions pour avoir consistance = pour être d'ordre 1**"))
    Eq_1 = ordre_1(s, A, b, c, j)
    Eqs.append(Eq_1)

    display(Markdown("**On doit ajouter 1 condition pour être d'ordre 2**"))
    Eq_2 = ordre_2(s, A, b, c, j)
    Eqs.append([Eq_2])

    display(Markdown("**On doit ajouter 2 conditions pour être d'ordre 3**"))
    Eq_3 = ordre_3(s, A, b, c, j)
    Eqs.append(Eq_3)

    # display(Markdown("**On doit ajouter 4 conditions pour être d'ordre 4**"))
    # Eq_4 = ordre_4(s, A, b, c, j)
    # Eqs.append(Eq_4)

    return Eqs


def matrice_Butcher(s, A, b, c):
    But = sp.Matrix(A)
```

```
    But = But.col_insert(0, sp.Matrix(c))
    last = [sp.Symbol(" ")]
    last.extend(b)
    But = But.row_insert(s,sp.Matrix(last).T)
    display(Math(sp.latex(sp.Matrix(But))))
    return But


def ordre_1(s, A, b, c, j):
    texte = r"\sum_{j=1}^s b_j =" + f"{sum(b).simplify()}"
    texte += r"\text{ doit être égale à }1"
    display(Math(texte))
    Eq_1 = [sp.Eq( sum(b).simplify(), 1 )]
    for i in range(s):
        somma = sp.summation(A[i,j],(j,0,s-1)).simplify()
        texte = r'\sum_{j=1}^s a_{'+str(i)+r'j}=' + sp.latex( somma )
        texte += r"\text{ doit être égale à }"+sp.latex(c[i])
        display(Math( texte ))
        Eq_1.append(sp.Eq( somma, c[i] ))
    return Eq_1


def ordre_2(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j=' +sp.latex(sum([b[i]*c[i] for i in range(s)]).simplify())
    texte += r"\text{ doit être égale à }\frac{1}{2}"
    display(Math(texte))
    Eq_2 = sp.Eq( sum([b[i]*c[i] for i in range(s)]).simplify(), sp.Rational(1,2) )
    return Eq_2


def ordre_3(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j^2='
    texte += sp.latex( sum([b[i]*c[i]**2 for i in range(s)]).simplify() )
    texte += r"\text{ doit être égale à }\frac{1}{3}"
    display(Math(texte))
    Eq_3_1 = sp.Eq( sum([b[i]*c[i]**2 for i in range(s)]).simplify(), sp.Rational(1,3) )
    texte = r'\sum_{i,j=1}^s b_i a_{ij} c_j='
    somma = sum([b[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{6}"
    display(Math(texte))
    Eq_3_2 = sp.Eq( sum([b[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,6) )
    return [Eq_3_1, Eq_3_2]


def ordre_4(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j^3='
    texte += sp.latex( sum([b[i]*c[i]**3 for i in range(s)]).simplify() )
    texte += r"\text{ doit être égale à }\frac{1}{4}"
    display(Math(texte))
    Eq_4_1 = sp.Eq( sum([b[i]*c[i]**3 for i in range(s)]).simplify(), sp.Rational(1,4) )
    texte = r'\sum_{i,j=1}^s b_i c_i a_{ij} c_j='
    somma = sum([b[i]*c[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{8}"
    display(Math(texte))
    Eq_4_2 = sp.Eq( sum([b[i]*c[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,8) )
    texte = r'\sum_{i,j=1}^s b_i a_{ij} c_j^2='
    somma = sum([b[i]*A[i,j]*c[j]**2 for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{12}"
```

```
    display(Math(texte))
    Eq_4_3 = sp.Eq( sum([b[i]*A[i,j]*c[j]**2 for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,12) )
    texte = r'\sum_{i,j,k=1}^s b_i a_{ij} a_{jk} c_k='
    somma = sum([b[i]*A[i,j]*A[j,k]*c[k] for k in range(s) for j in range(s) for i in
↪   range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{24}"
    display(Math(texte))
    Eq_4_4 = sp.Eq( sum([b[i]*A[i,j]*A[j,k]*c[k] for k in range(s) for j in range(s) for i in
↪   range(s)]).simplify(), sp.Rational(1,24) )
    return [Eq_4_1, Eq_4_2, Eq_4_3, Eq_4_4]



# ============================================================================

s = 3
b = [1/sp.S(6), 2/sp.S(3), 1/sp.S(6)]
c = [ 0, 1/sp.S(2), 1]
A = [[0, 0, 0], [1/sp.S(2), 0, 0], [0, sp.S(1), 0]]
Eqs = ordre_RK(s, A=A, b=b, c=c)
```

Matrice de Butcher

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 1 & 0 \\ & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{bmatrix}$$

On a 4 conditions pour avoir consistance = pour être d'ordre 1

$$\sum_{j=1}^{s} b_j = 1 \text{ doit être égale à } 1$$

$$\sum_{j=1}^{s} a_{0j} = 0 \text{ doit être égale à } 0$$

$$\sum_{j=1}^{s} a_{1j} = \frac{1}{2} \text{ doit être égale à } \frac{1}{2}$$

$$\sum_{j=1}^{s} a_{2j} = 1 \text{ doit être égale à } 1$$

On doit ajouter 1 condition pour être d'ordre 2

$$\sum_{j=1}^{s} b_j c_j = \frac{1}{2} \text{ doit être égale à } \frac{1}{2}$$

On doit ajouter 2 conditions pour être d'ordre 3

$$\sum_{j=1}^{s} b_j c_j^2 = \frac{1}{3} \text{ doit être égale à } \frac{1}{3}$$

$$\sum_{i,j=1}^{s} b_i a_{ij} c_j = \frac{1}{12} \text{ doit être égale à } \frac{1}{6}$$

```
# Test de l'ordre
# ============================================================================

t = sp.symbols('t')
y = sp.Function('y')(t)
```

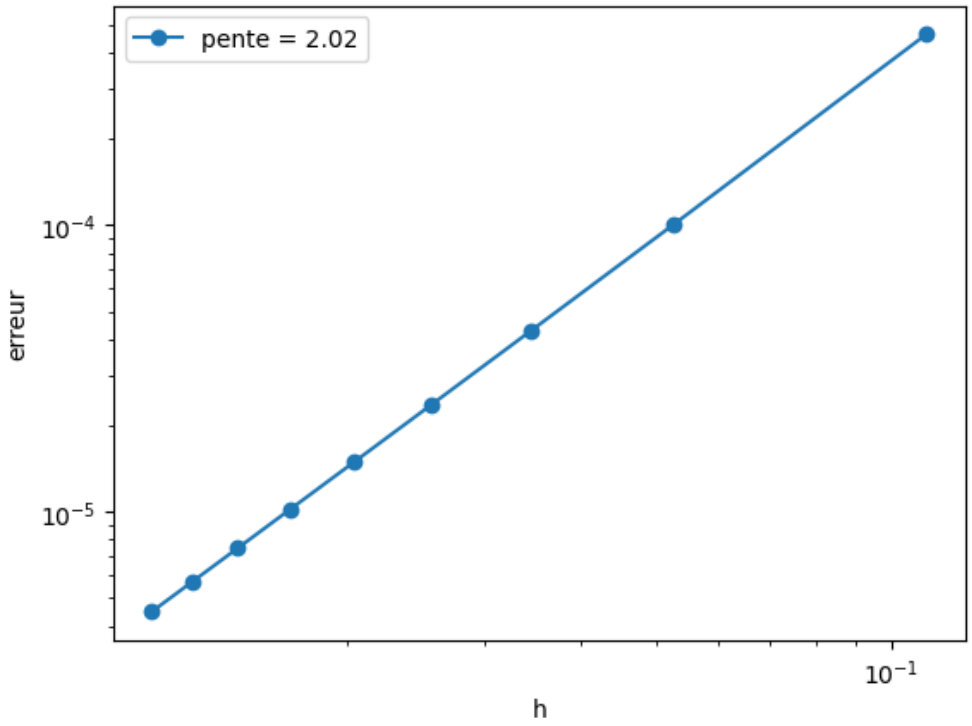
```
t0, y0 = 1, 1
tfin = 2
phi_sp = lambda t,y : -y/t-y**2*sp.log(t)
sol_sp = sp.dsolve(y.diff(t)-phi_sp(t,y), y, ics={y.subs(t,t0):y0})
display(sol_sp)

# On convertit les expressions symboliques en fonctions numpy
sol_np = sp.lambdify(t, sol_sp.rhs, 'numpy')
phi = sp.lambdify([t,y], phi_sp(t,y), 'numpy')

# On résout le problème avec différentes valeurs de N
H, err = [], []
for N in range(10,100,10):
    tt = np.linspace(t0,tfin,N)
    uu = RK(phi,tt,y0)
    yy = sol_np(tt)
    err.append(np.linalg.norm(uu-yy,np.inf))
    H.append(tt[1]-tt[0])

# On calcule la pente de la droite de meilleur ajustement en échelle log-log
pente = np.polyfit(np.log(H),np.log(err),1)[0]
plt.figure()
plt.loglog(H,err,'o-',label=f'pente = {pente:.2f}')
plt.xlabel('h')
plt.ylabel('erreur')
plt.legend() ;
```

$$y(t) = \frac{2}{t\left(\log(t)^2 + 2\right)}$$



Étudions la A-stabilité de la méthode en appliquant le schéma au problème ayant $\varphi(t, y) = -\beta y$ avec $\beta > 0$. On a alors

$$\begin{cases} k_1 = -\beta u_n \\ k_2 = -\beta(u_n + \frac{h}{2}k_1) = -\beta u_n + \frac{\beta^2 h}{2}u_n \\ k_3 = -\beta(u_n + hk_2) = -\beta u_n + \beta^2 h u_n - \frac{\beta^3 h^2}{2}u_n \\ u_{n+1} = u_n + \frac{h}{6}(k_1 + 4k_2 + k_3) = \left(1 - (\beta h) + \frac{(\beta h)^2}{2} - \frac{(\beta h)^3}{12}\right) u_n \end{cases}$$

Notons $x = \beta h > 0$ et $q(x)$ le coefficient d'amplification. On a alors

$$q(x) = \frac{u_{n+1}}{u_n} = 1 - x + \frac{x^2}{2} - \frac{x^3}{12}$$

La suite vérifie $u_n \xrightarrow[n\to\infty]{} 0$ ssi $|q(x)| < 1$. En étudiant brièvement la fonction $q(x)$, on trouve que q est une fonction monotone décroissante ayant un point d'inflexion en $x = 2$ et telle que $q(0) = 1$ et $q(x) \xrightarrow[x\to+\infty]{} -\infty$. On cherche alors les solutions de $q(x) + 1 = 0$.

Il n'est pas possible de résoudre analytiquement cette équation, on utilisera donc une méthode numérique pour déterminer la valeur de x pour laquelle le schéma est A-stable.

De même, le schéma génère une solution monotone ssi $q(x) > 0$. On cherche alors numériquement les solutions de $q(x) = 0$.

```
u_n = sp.symbols(r'u_n')
u_np1 = sp.symbols(r'u_{n+1}', escape=True)

dt = sp.symbols(r'h', positive=True)
x = sp.symbols(r'x', positive=True)

k_0 = sp.symbols(r'k_0')
k_1 = sp.symbols(r'k_1')
k_2 = sp.symbols(r'k_2')
kk = [k_0, k_1, k_2]

beta = sp.symbols(r'\beta', positive=True)
phi = lambda y: -beta*y

Eqs = [ sp.Eq( kk[i] ,  phi(u_n + dt*sum([kk[j]*A[i][j] for j in range(s)])) ) for i in range(s) ]
for eq in Eqs:
    display(eq)

sol = sp.solve( Eqs, kk ) # c'est un dictionnaire
KK = [sol[kk[i]] for i in range(s)]
schema = sp.Eq( u_np1 , u_n + dt*sum([KK[j]*b[j] for j in range(s)]) )
display(schema)

q = (schema.rhs/u_n).simplify().subs(beta*dt,x)
display(Math(sp.latex(sp.Eq(sp.Symbol('q(x)'),q.simplify()))))

# Tracer q(x) avec matplotlib
xx = np.linspace(0, 5, 400)
q_func = sp.lambdify(x, q, 'numpy')

plt.plot(xx, q_func(xx), label='q(x)')
plt.ylim(-2, 2) # On zoome sur la zone intéressante [-1, 1]
plt.xlim(xx.min(), xx.max())
plt.axhline(y=1, color='black', linestyle='-', alpha=0.5)
plt.axhline(y=-1, color='black', linestyle='-', alpha=0.5)
plt.fill_between(xx, -1, 1, color='gray', alpha=0.1, label='Zone de A-Stabilité')
plt.xlabel('$x$')
plt.ylabel('$q(x)$')
```

```
# Conditions de stabilité et de monotonie
from scipy.optimize import fsolve
x_A = fsolve(lambda x: q_func(x)+1, 4)[0] # q(x) = -1
x_M = fsolve(lambda x: q_func(x)  , 3)[0] # q(x) = 0

display(Markdown(f"La condition de A-stabilité est satisfaite pour $\\beta h < {x_A:.6f}$"))
display(Markdown(f"La condition de monotonie est satisfaite pour $\\beta h < {x_M:.6f}$"))
plt.axhline(y=0,color='red',linestyle='--', label='$y=0$')

plt.scatter(x_A, -1, color='green')
plt.annotate(f'$x_A = {x_A:.6f}$', xy=(x_A, -1), xytext=(x_A+0.2, -0.5), arrowprops=dict(arrowstyle='->',
 ↪  lw=1.5), fontsize=12, color='green')
plt.scatter(x_M, 0, color='red')
plt.annotate(f'$x_M = {x_M:.6f}$', xy=(x_M, 0), xytext=(x_M+0.2, 0.5), arrowprops=dict(arrowstyle='->',
 ↪  lw=1.5), fontsize=12, color='red');
# plt.axvline(x=x_A, color='g', linestyle=':', label=f'$x_A = {x_A:.6f}$')
# plt.axvline(x=x_M, color='b', linestyle=':', label=f'$x_M = {x_M:.6f}$');
```

$$k_0 = -\beta u_n$$

$$k_1 = -\beta\left(\frac{hk_0}{2} + u_n\right)$$

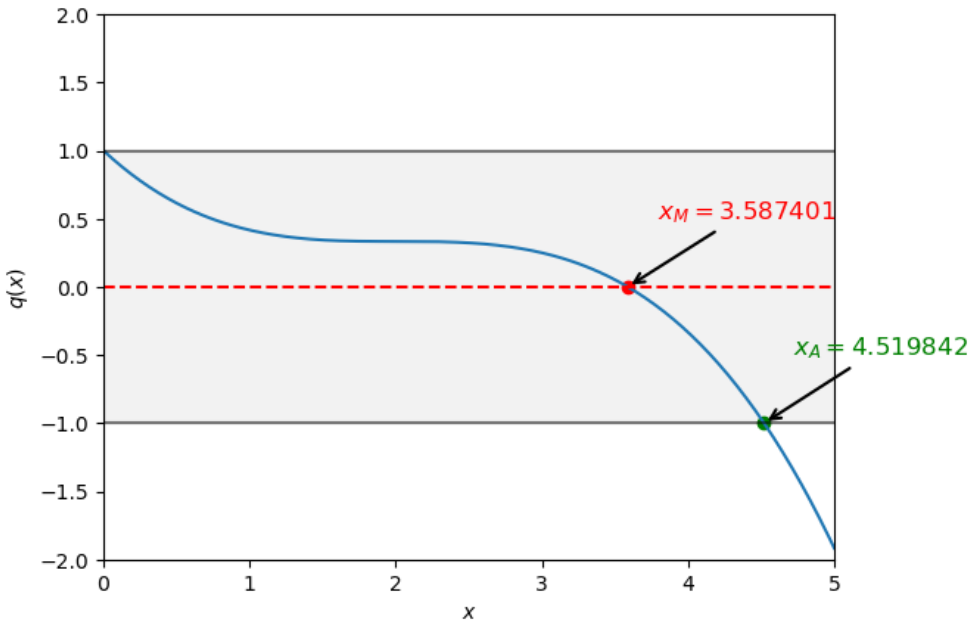
$$k_2 = -\beta\left(hk_1 + u_n\right)$$

$$u_{n+1} = h\left(-\frac{\beta^3 h^2 u_n}{12} + \frac{\beta^2 h u_n}{2} - \beta u_n\right) + u_n$$

$$q(x) = -\frac{x\left(x^2 - 6x + 12\right)}{12} + 1$$

La condition de A-stabilité est satisfaite pour $\beta h < 4.519842$

La condition de monotonie est satisfaite pour $\beta h < 3.587401$



23.2. EXERCICE 2 : schéma multi-pas linéaire explicite

1. Déterminer les **coefficients** d'un schéma multi-pas linéaire explicite à 3 pas sous les conditions suivantes :
   - le schéma est convergent ;
   - $b_0 = \frac{5}{12}$ et $b_2 = \frac{23}{12}$;

- le premier polynôme caractéristique admet $r = 0$ comme racine double.

2. **Calculer** l'ordre de convergence du schéma déterminé à la question 1.
3. Implémenter le schéma et **vérifier expérimentalement** son ordre de convergence en appliquant ce schéma au problème de Cauchy suivant

$$\begin{cases} y'(t) = t^2 y(t), \\ y(0) = 1, \end{cases} \qquad t \in [0, 1].$$

Correction

Un schéma multi-pas linéaire explicite à $q = 3$ trois pas s'écrit

$$u_{n+1} = a_0 u_n + a_1 u_{n-1} + a_2 u_{n-2} + h\Big(b_0 \varphi_n + b_1 \varphi_{n-1} + b_2 \varphi_{n-2}\Big).$$

:one : Calcule des coefficients a_0, a_1, a_2. Le premier polynomial caractéristique est

$$\varrho(r) = r^3 - a_0 r^2 - a_1 r - a_2$$

qui a 3 racines. On sait que $r = 0$ est une racine double et que le schéma est convergent, donc $r = 1$ est une racine simple de ϱ. Le polynôme caractéristique est alors

$$\varrho(r) = (r-1)r^2 = r^3 - r^2.$$

Par identification des coefficients, on trouve $a_0 = 1$ et $a_1 = a_2 = 0$.

:two : Calcul des coefficients b_0, b_1, b_2.

Les valeurs de b_0 et b_2 sont donnés dans l'énoncé :

$$u_{n+1} = u_n + \frac{h}{12}\Big(5\varphi_n + 12 b_1 \varphi_{n-1} + 23\varphi_{n-2}\Big).$$

Il reste à déterminer b_1. Le schéma étant convergent, donc consistant et zéro-stable. La condition de stabilité est donnée par les deux relations $\xi(0) = 1$ et $\xi(1) = 1$. La condition $\xi(0) = 1$ est déjà satisfaite, car cela équivaut à dire que $r = 1$ est une racine du polynôme caractéristique. La condition $\xi(1) = 1$ donne

$$-a_1 + 2a_2 + b_0 + b_1 + b_2 = 1$$

ce qui permet de calculer b_1 :

$$b_1 = 1 - b_0 - b_2 = 1 - \frac{5}{12} - \frac{23}{12} = 1 - \frac{28}{12} = -\frac{16}{12} = -\frac{4}{3}.$$

Le schéma cherché est donc

$$\boxed{u_{n+1} = u_n + h\Big(\frac{5}{12}\varphi_n - \frac{4}{3}\varphi_{n-1} + \frac{23}{12}\varphi_{n-2}\Big).}$$

:three : Calcul de l'ordre de convergence. Il est d'ordre 1 car :

$$\xi(2) = a_1 + 4a_2 - 2\left(b_1 + 2b_2 - b_{-1}\right) = 0 + 0 - 2\left(-\frac{4}{3} + 2\frac{23}{12}\right) = -5 \neq 1$$

:four : Implémentation du schéma et vérification expérimentale de l'ordre de convergence.

```
def MP(phi,tt,sol_exacte):
    h = tt[1] - tt[0]
    q = 3
    uu = np.zeros_like(tt)
    uu[:q] = sol_exacte(tt[:q])
    for n in range(len(tt)-1):
        k0 = phi(tt[n], uu[n])
        k1 = phi(tt[n-1], uu[n-1])
        k2 = phi(tt[n-2], uu[n-2])
        uu[n+1] = uu[n] + h*(5*k0 - 16*k1 +23*k2)/12
    return uu


# PB de Cauchy

t0, y0 = 0, 1
phi = lambda t,y: t**2*y

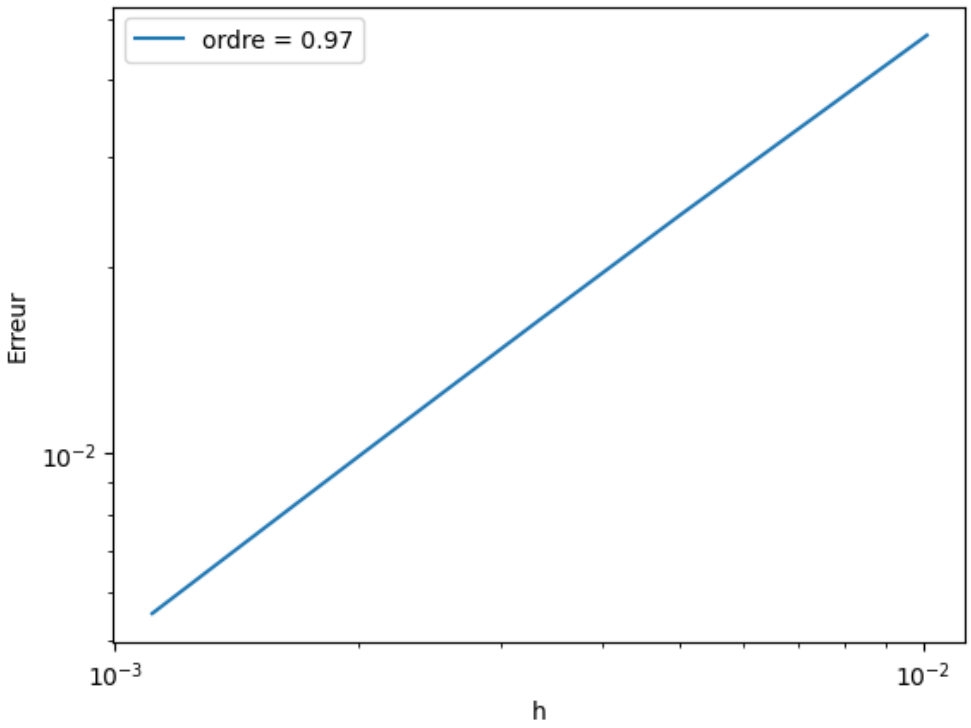
import sympy as sp
t = sp.symbols('t')
y = sp.Function('y')
eq = sp.Eq(y(t).diff(t), t**2*y(t))
sol = sp.dsolve(eq, y(t), ics={y(0): 1})
display(sol)

sol_exacte = sp.lambdify(t, sol.rhs, 'numpy')

H, err = [], []
for N in range(100,1000,100):
    tt = np.linspace(t0, 1, N)
    uu = MP(phi,tt,sol_exacte)
    yy = sol_exacte(tt)
    H.append(tt[1]-tt[0])
    err.append(np.linalg.norm(uu-yy, np.inf))


ordre = np.polyfit(np.log(H), np.log(err), 1)[0]
plt.loglog(H,err, label=f"ordre = {ordre:.2f}")
plt.xlabel('h')
plt.ylabel('Erreur')
plt.legend();
```

$y(t) = e^{\frac{t^3}{3}}$


ordre = 0.97
Erreur
10^{-2}
10^{-3}
10^{-2}
h

Chapitre 24.

2024 Contrôle Continu

2h30, deposer le notebook dans Moodle

```
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp
from scipy.optimize import fsolve
# from scipy.integrate import solve_ivp, odeint
from IPython.display import display, Markdown, Math
```

24.1. EXERCICE 1 : choix d'un schéma selon le problème de Cauchy donné

Considérons le problème de Cauchy suivant

$$\begin{cases} y'(t) = \lambda\Big(y(t) - \cos(t)\Big) - \sin(t) \quad \text{pour } t \in [0, 10], \\ y(0) = 1 + \varepsilon. \end{cases}$$

1. Calculer la solution exacte en fonction de $\lambda \in \mathbb{R}$ et ε.
2. Si $\lambda = 10$ et $\varepsilon = 0$, quel type de schéma doit-on privilégier pour résoudre ce problème de Cauchy ?
3. Si $\lambda = -10$ et $\varepsilon \neq 0$, quel type de schéma doit-on privilégier pour résoudre ce problème de Cauchy ?

On peut résoudre ce problème de Cauchy à la main en remarquant que l'edo se réécrit comme

$$\lambda(y(t) - \cos(t)) = y'(t) + \sin(t) = \big(y(t) - \cos(t)\big)'.$$

Si on pose $z(t) = y(t) - \cos(t)$ on a le problème de Cauchy $z'(t) = \lambda z(t)$ avec $z(0) = y(0) - 1$. La solution est donc $z(t) = (y(0) - 1)e^{\lambda t}$ et donc

$$y(t) = \varepsilon e^{\lambda t} + \cos(t).$$

On peut vérifier nos calculs en utilisant `sympy` :

```
t = sp.symbols('t')
y = sp.Function('y')(t)
lamb = sp.symbols(r'\lambda', real=True)
epsilon = sp.symbols(r'\varepsilon', real=True)
dydt = y.diff(t)
ode = sp.Eq(dydt, lamb*(y-sp.cos(t))-sp.sin(t))
sol = sp.dsolve(ode, y, ics={y.subs(t, 0): 1+epsilon})
sol
```

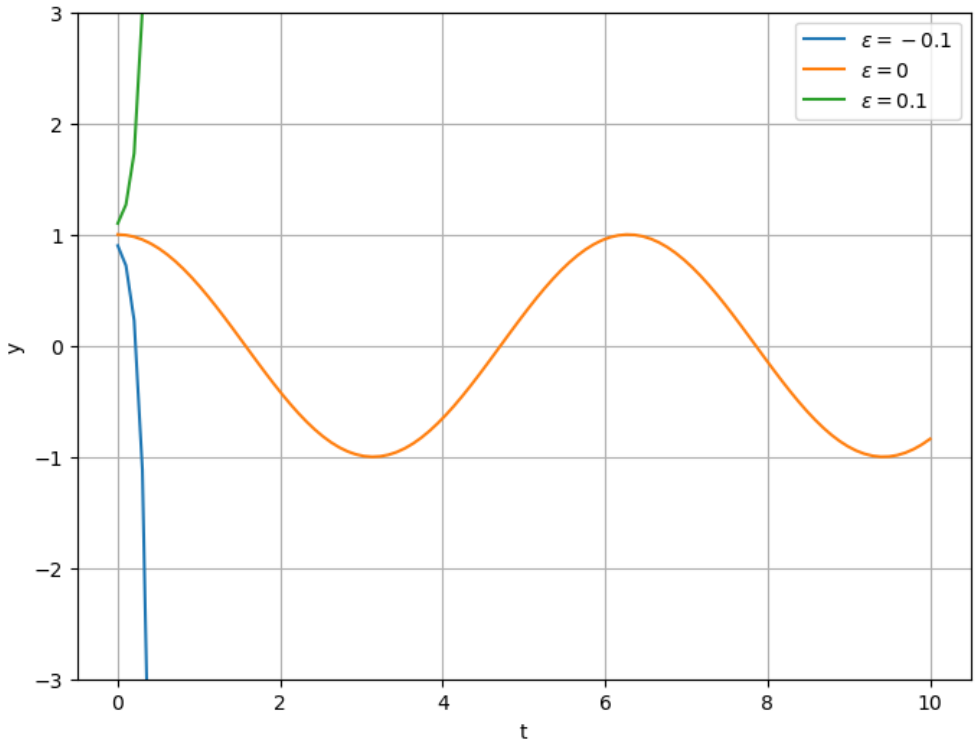
$y(t) = \varepsilon e^{\lambda t} + \cos\left(t\right)$

- Si $\lambda > 0$, il s'agit d'un problème de Cauchy **numériquement mal posé** (c'est-à-dire, il est très sensible aux perturbations sur la donnée initiale). En effet, si $\varepsilon = 0$, la solution exacte est simplement $y(t) = \cos(t)$, mais pour $\varepsilon > 0$, elle croît comme une exponentielle.
  On doit donc choisir un **schéma d'ordre élévé.**

```
tt = np.linspace(0, 10, 100)
fig, ax = plt.subplots(figsize=(8, 6))


for epsi in [-0.1,0,0.1]:
    y_sol = sp.lambdify(t, sol.rhs.subs({lamb:10,epsilon:epsi}), 'numpy')
    plt.plot(tt, y_sol(tt), label=fr'$\epsilon={epsi}$')

ax.set_xlabel('t')
ax.set_ylabel('y')
ax.grid()
ax.legend()
ax.axes.set_ylim(-3, 3);
```

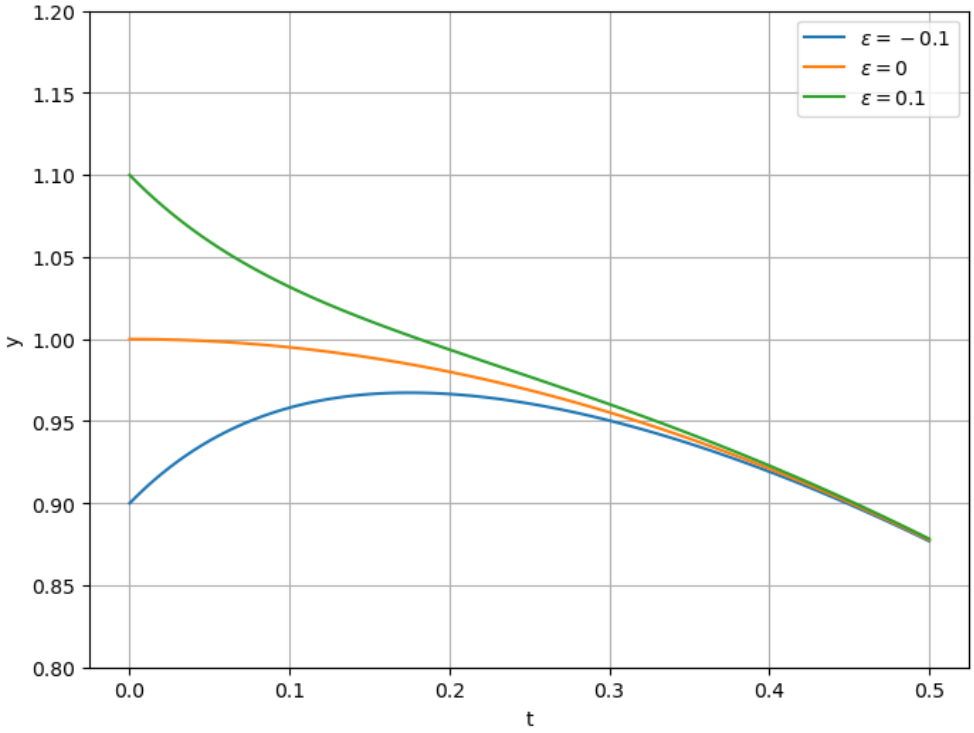


- Si $\lambda < 0$ et $|\lambda| \gg 1$, il s'agit d'un problème de Cauchy **raide**. En effet, si $\varepsilon \neq 0$, la solution s'amorti vers $\cos(t)$ sur une intervalle très court.
  On doit donc choisir un **schéma inconditionellement A-stable.**

```
tt = np.linspace(0, 0.5, 100)
fig, ax = plt.subplots(figsize=(8, 6))

for epsi in [-0.1,0,0.1]:
    y_sol = sp.lambdify(t, sol.rhs.subs({lamb:-10,epsilon:epsi}), 'numpy')
    plt.plot(tt, y_sol(tt), label=fr'$\epsilon={epsi}$')

ax.set_xlabel('t')
ax.set_ylabel('y')
ax.grid()
ax.legend()
ax.axes.set_ylim(1-0.2, 1+0.2);
```



24.2. EXERCICE 2 : schémas de Runge-Kutta à 4 étages explicite d'ordre maximal

La matrice de Butcher d'un schéma RK à $s = 4$ étages quelconque s'écrit :

$$\begin{array}{c|cccc} c_0 & a_{00} & a_{01} & a_{02} & a_{03} \\ c_1 & a_{10} & a_{11} & a_{12} & a_{13} \\ c_2 & a_{20} & a_{21} & a_{22} & a_{23} \\ c_3 & a_{30} & a_{31} & a_{32} & a_{33} \\ \hline & b_0 & b_1 & b_2 & b_3 \end{array}$$

Elle contient 24 coefficients.

1. Écrire la matrice de Butcher d'un schéma RK à $s = 4$ étages **explicite consistant** (elle ne comportera que les 9 coefficients c_1, c_2, c_3, b_1, b_2, b_3, a_{21}, a_{31}, a_{32}).
2. Sans effectuer de calculs, donner l'ordre maximal d'un schéma explicite à 4 étages.
3. Soit κ le nombre de lettres de votre prénom, dans mon cas je pose `kappa = len("Gloria")`.
   Poser $c_1 = \frac{1}{\kappa}$, $c_2 = 1 - c_1$ et $c_3 = 1$. Déterminer les autres 6 paramètres pour que le schéma explicite soit d'ordre maximal.
4. Tester empiriquement l'ordre de convergence de ce schéma sur le problème de Cauchy suivant : $y'(t) = 10(y(t) - \sin(t)) + \cos(t)$ et $y(0) = 0$ pour $t \in [0, \pi/2]$. Est-ce un schéma adapté à ce type de problème ?
5. Déterminer sous quelle condition sur h ce schéma est A-stable.

Si le schéma est explicite, la matrice $\mathbb{A}$ est triangulaire inférieure et la diagonale ne contient que des zéros, donc $a_{00} = a_{01} = a_{02} = a_{03} = a_{11} = a_{12} = a_{13} = a_{22} = a_{23} = a_{33} = 0$.

Si le schéma est consistant alors

$$\begin{cases} \displaystyle\sum_{j=0}^{3} b_i = 1 \\ \displaystyle c_i = \sum_{j=0}^{3} a_{ij} \quad i = 0, \dots, 3 \end{cases}$$

ainsi $c_0 = 0$, $a_{10} = c_1$, $a_{20} = c_2 - a_{21}$, $a_{30} = c_3 - a_{31} - a_{32}$ et enfin $b_0 = 1 - b_1 - b_2 - b_3$.

Avec ces 15 relations, il ne reste plus que 24-15=9 coefficients à déterminer, à savoir a_{21}, a_{31}, a_{32}, b_1, b_2, b_3, c_1, c_2 et c_3 :

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ c_1 & c_1 & 0 & 0 & 0 \\ c_2 & c_2 - a_{21} & a_{21} & 0 & 0 \\ c_3 & c_3 - a_{31} - a_{32} & a_{31} & a_{32} & 0 \\ \hline & 1 - b_1 - b_2 - b_3 & b_1 & b_2 & b_3 \end{array}$$

La barrière de Butcher affirme qu'un schéma RK à $s = 4$ étages **explicite** ne peut pas être d'ordre supérieur à $s = 4$.

On a déjà pris en compte les conditions de consistance. Il reste à déterminer les paramètres pour que le schéma soit d'ordre maximal. Cela revient à résoudre le système d'équations non-linéaires suivant :

$$\begin{cases} b_1c_1 + b_2c_2 + b_3c_3 = \frac{1}{2}, \\ b_1c_1^2 + b_2c_2^2 + b_3c_3^2 = \frac{1}{3}, \\ a_{21}b_2c_1 + a_{31}b_3c_1 + a_{32}b_3c_2 = \frac{1}{6}, \\ b_1c_1^3 + b_2c_2^3 + b_3c_3^3 = \frac{1}{4}, \\ a_{21}b_2c_1c_2 + a_{31}b_3c_1c_3 + a_{32}b_3c_2c_3 = \frac{1}{8}, \\ a_{21}b_2c_1^2 + a_{31}b_3c_1^2 + a_{32}b_3c_2^2 = \frac{1}{12}, \\ a_{21}a_{32}b_3c_1 = \frac{1}{24} \end{cases}$$

On peut aussi utiliser `sympy` pour retrouver ce système :

```
prlat = lambda *args : display(Math(''.join( sp.latex(arg) for arg in args )))

def ordre_RK(s, A=None, b=None, c=None):

    j = sp.symbols('j')

    if A is None:
        A = sp.MatrixSymbol('a', s, s)
    else:
        A = sp.Matrix(A)
    if c is None:
        c = sp.symbols(f'c_0:{s}')
    if b is None:
        b = sp.symbols(f'b_0:{s}')


    display(Markdown("**Matrice de Butcher**"))
    But = matrice_Butcher(s, A, b, c)

    Eqs = []

    # display(Markdown(f"**On a {s+1} conditions pour avoir consistance = pour être d'ordre 1**"))
    # ordre_1(s, A, b, c, j)

    display(Markdown("**On doit ajouter 1 condition pour être d'ordre 2**"))
    Eq_2 = ordre_2(s, A, b, c, j)
    Eqs.append(Eq_2)

    display(Markdown("**On doit ajouter 2 conditions pour être d'ordre 3**"))
    Eq_3 = ordre_3(s, A, b, c, j)
    Eqs.extend(Eq_3)

    display(Markdown("**On doit ajouter 4 conditions pour être d'ordre 4**"))
    Eq_4 = ordre_4(s, A, b, c, j)
```

```
    Eqs.extend(Eq_4)

    return Eqs


def matrice_Butcher(s, A, b, c):
    But = sp.Matrix(A)
    But = But.col_insert(0, sp.Matrix(c))
    last = [sp.Symbol(" ")]
    last.extend(b)
    But = But.row_insert(s,sp.Matrix(last).T)
    display(Math(sp.latex(sp.Matrix(But))))
    return But


def ordre_1(s, A, b, c, j):
    texte = r"\sum_{j=1}^s b_j =" + f"{sum(b).simplify()}"
    texte += r"\text{ doit être égale à }1"
    display(Math(texte))
    for i in range(s):
        somma = sp.summation(A[i,j],(j,0,s-1)).simplify()
        texte = r'\sum_{j=1}^s a_{'+str(i)+r'j}=' + sp.latex( somma )
        texte += r"\text{ doit être égale à }"+sp.latex(c[i])
        display(Math( texte ))
    return None


def ordre_2(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j=' +sp.latex(sum([b[i]*c[i] for i in range(s)]).simplify())
    texte += r"\text{ doit être égale à }\frac{1}{2}"
    display(Math(texte))
    Eq_2 = sp.Eq( sum([b[i]*c[i] for i in range(s)]).simplify(), sp.Rational(1,2) )
    return Eq_2


def ordre_3(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j^2='
    texte += sp.latex( sum([b[i]*c[i]**2 for i in range(s)]).simplify() )
    texte += r"\text{ doit être égale à }\frac{1}{3}"
    display(Math(texte))
    Eq_3_1 = sp.Eq( sum([b[i]*c[i]**2 for i in range(s)]).simplify(), sp.Rational(1,3) )
    texte = r'\sum_{i,j=1}^s b_i a_{ij} c_j='
    somma = sum([b[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{6}"
    display(Math(texte))
    Eq_3_2 = sp.Eq( sum([b[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,6) )
    return [Eq_3_1, Eq_3_2]


def ordre_4(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j^3='
    texte += sp.latex( sum([b[i]*c[i]**3 for i in range(s)]).simplify() )
    texte += r"\text{ doit être égale à }\frac{1}{4}"
    display(Math(texte))
    Eq_4_1 = sp.Eq( sum([b[i]*c[i]**3 for i in range(s)]).simplify(), sp.Rational(1,4) )
    texte = r'\sum_{i,j=1}^s b_i c_i a_{ij} c_j='
    somma = sum([b[i]*c[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{8}"
    display(Math(texte))
    Eq_4_2 = sp.Eq( sum([b[i]*c[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,8) )
```

```
    texte = r'\sum_{i,j=1}^s b_i a_{ij} c_j^2='
    somma = sum([b[i]*A[i,j]*c[j]**2 for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{12}"
    display(Math(texte))
    Eq_4_3 = sp.Eq( sum([b[i]*A[i,j]*c[j]**2 for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,12) )
    texte = r'\sum_{i,j,k=1}^s b_i a_{ij} a_{jk} c_k='
    somma = sum([b[i]*A[i,j]*A[j,k]*c[k] for k in range(s) for j in range(s) for i in
↪   range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{24}"
    display(Math(texte))
    Eq_4_4 = sp.Eq( sum([b[i]*A[i,j]*A[j,k]*c[k] for k in range(s) for j in range(s) for i in
↪   range(s)]).simplify(), sp.Rational(1,24) )
    return [Eq_4_1, Eq_4_2, Eq_4_3, Eq_4_4]



################################################
# Conditions avec s étages
################################################

s = 4

# Construction de la matrice de Butcher en supposant
# que le schéma est Explicite
# et qu'il est consistant (i.e. au moins d'ordre 1)

b_1, b_2, b_3 = sp.symbols('b_1 b_2 b_3', real=True)
c_1, c_2, c_3 = sp.symbols('c_1 c_2 c_3', real=True)
a_21,  a_31, a_32 = sp.symbols('a_21 a_31 a_32', real=True)

b = [1-b_1-b_2-b_3, b_1, b_2, b_3]
c = [0, c_1, c_2, c_3]
A = [[0, 0, 0, 0],
     [c_1, 0, 0, 0],
     [c_2-a_21, a_21, 0, 0],
     [c_3-a_31-a_32, a_31, a_32, 0]]


Eqs = ordre_RK(s, A, b, c)

# for eq in Eqs:
#     display(eq)
```

Matrice de Butcher

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ c_1 & c_1 & 0 & 0 & 0 \\ c_2 & -a_{21} + c_2 & a_{21} & 0 & 0 \\ c_3 & -a_{31} - a_{32} + c_3 & a_{31} & a_{32} & 0 \\ & -b_1 - b_2 - b_3 + 1 & b_1 & b_2 & b_3 \end{bmatrix}$$

On doit ajouter 1 condition pour être d'ordre 2

$$\sum_{j=1}^{s} b_j c_j = b_1 c_1 + b_2 c_2 + b_3 c_3 \text{ doit être égale à } \frac{1}{2}$$

On doit ajouter 2 conditions pour être d'ordre 3

$$\sum_{j=1}^{s} b_j c_j^2 = b_1 c_1^2 + b_2 c_2^2 + b_3 c_3^2 \text{ doit être égale à } \frac{1}{3}$$

$\sum_{i,j=1}^{s} b_i a_{ij} c_j = a_{21} b_2 c_1 + a_{31} b_3 c_1 + a_{32} b_3 c_2$ doit être égale à $\frac{1}{6}$

On doit ajouter 4 conditions pour être d'ordre 4

$\sum_{j=1}^{s} b_j c_j^3 = b_1 c_1^3 + b_2 c_2^3 + b_3 c_3^3$ doit être égale à $\frac{1}{4}$

$\sum_{i,j=1}^{s} b_i c_i a_{ij} c_j = a_{21} b_2 c_1 c_2 + a_{31} b_3 c_1 c_3 + a_{32} b_3 c_2 c_3$ doit être égale à $\frac{1}{8}$

$\sum_{i,j=1}^{s} b_i a_{ij} c_j^2 = a_{21} b_2 c_1^2 + a_{31} b_3 c_1^2 + a_{32} b_3 c_2^2$ doit être égale à $\frac{1}{12}$

$\sum_{i,j,k=1}^{s} b_i a_{ij} a_{jk} c_k = a_{21} a_{32} b_3 c_1$ doit être égale à $\frac{1}{24}$

On résout ce système NON LINÉAIRE pour trouver les coefficients **dans le cas particulier où c_1, c_2 et c_3 sont donnés.** Voici le démarche de résolution :

- on trouve b_1, b_2 et b_3 en résolvant le sous-sytème (linéaire) :

$$\begin{cases} b_1 c_1 + b_2 c_2 + b_3 c_3 = \frac{1}{2}, \\ b_1 c_1^2 + b_2 c_2^2 + b_3 c_3^2 = \frac{1}{3}, \\ b_1 c_1^3 + b_2 c_2^3 + b_3 c_3^3 = \frac{1}{4}, \end{cases}$$

- on trouve a_{21}, a_{31} et a_{32} en résolvant le sous-sytème (linéaire) :

$$\begin{cases} a_{21} b_2 c_1 + a_{31} b_3 c_1 + a_{32} b_3 c_2 = \frac{1}{6}, \\ a_{21} b_2 c_1 c_2 + a_{31} b_3 c_1 c_3 + a_{32} b_3 c_2 c_3 = \frac{1}{8}, \\ a_{21} b_2 c_1^2 + a_{31} b_3 c_1^2 + a_{32} b_3 c_2^2 = \frac{1}{12}, \end{cases}$$

- on trouve le dernier paramètre a_{32} en résolvant l'équation :

$$a_{21} a_{32} b_3 c_1 = \frac{1}{24}.$$

Remarque Dans l'énoncé j'ai donné $c_3 = 1$ car cela simplifie les calculs à la main, mais en réalité il suffit de fixer c_1 et c_2 et on retrouve $c_3 = 1$ en résolvant le système :

```
# 7 équations (non linéaires) et 9 inconnues
# On fixe 2 paramètres et on résoud le système non linéaire

kappa = len("Gloria")

choix = {c_1 : sp.Rational(1,kappa), c_2 : sp.Rational(kappa-1,kappa)}
display(Markdown("On fixe $c_1$ et $c_2$"))
display(choix)

sol = sp.solve( [eq.subs(choix) for eq in Eqs] )[0]
display(Markdown("En imposant d'être d'ordre 4, on trouve les 7 autres paramètres :"))
display(sol)

display(Markdown("Autrement dit, la matrice de Butcher d'un schéma explicite consistant à 4 étages"))
But = matrice_Butcher(s, A, b, c)
display(Markdown("devient, avec les valeurs de $c_1$ et $c_2$ données en imposant l'ordre 4"))
But = But.subs(choix).subs(sol)
display(But)
```

On fixe c_1 et c_2

```
{c_1: 1/6, c_2: 5/6}
```

En imposant d'être d'ordre 4, on trouve les 7 autres paramètres :

```
{a_21: 5/2, a_31: 10, a_32: -1, b_1: 3/5, b_2: 3/5, b_3: -1/10, c_3: 1}
```

Autrement dit, la matrice de Butcher d'un schéma explicite consistant à 4 étages

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ c_1 & c_1 & 0 & 0 & 0 \\ c_2 & -a_{21}+c_2 & a_{21} & 0 & 0 \\ c_3 & -a_{31}-a_{32}+c_3 & a_{31} & a_{32} & 0 \\ & -b_1-b_2-b_3+1 & b_1 & b_2 & b_3 \end{bmatrix}$$

devient, avec les valeurs de c_1 et c_2 données en imposant l'ordre 4

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{6} & \frac{1}{6} & 0 & 0 & 0 \\ \frac{5}{6} & -\frac{5}{3} & \frac{5}{2} & 0 & 0 \\ 1 & -8 & 10 & -1 & 0 \\ & -\frac{1}{10} & \frac{3}{5} & \frac{3}{5} & -\frac{1}{10} \end{bmatrix}$$

```
######################################################
# EDO
######################################################
t0     = 0
y0     = 0
tfinal = np.pi/2

# Prothero-Robinson problem (stiff ou numériquement mal posé selon lambda)
phi_sym = lambda t,y: sp.cos(t)-10*(y-sp.sin(t))
phi = lambda t,y: np.cos(t)-10*(y-np.sin(t))

######################################################
# solution exacte
######################################################
t   = sp.Symbol('t')
y   = sp.Function('y')
edo = sp.Eq( (y(t)).diff() , phi_sym(t,y(t)) )
solpar = sp.dsolve(edo , ics={y(t0):y0})
display(solpar)

sol_exacte = sp.lambdify(t,solpar.rhs,'numpy')

######################################################
# schéma
######################################################

# Paramètres du schéma recuperés depuis la matrice de Butcher obtenue précédemment
cc = np.array(But[:-1,0]).astype(float)
bb = np.array(list(But[-1,1:])).astype(float)
AA = np.array(But[:-1,1:]).astype(float)

# Schéma de Runge-Kutta explicite à 4 étages
def RK(phi, tt, y0):
    h  = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    # initialisation
    uu[0] = y0
    # recurrence
    for n in range(len(tt) - 1):
        k0 = phi(tt[n]            , uu[n] )
        k1 = phi(tt[n] + cc[1,0]*h, uu[n] + h* AA[1,0]*k0)
        k2 = phi(tt[n] + cc[2,0]*h, uu[n] + h*(AA[2,0]*k0 + AA[2,1]*k1))
        k3 = phi(tt[n] + cc[3,0]*h, uu[n] + h*(AA[3,0]*k0 + AA[3,1]*k1 + AA[3,2]*k2))
```

```
        uu[n+1] = uu[n] + h*(bb[0]*k0 + bb[1]*k1 + bb[2]*k2 + bb[3]*k3)
    return uu


#####################################################
# ordre
#####################################################
H = []
err = []

plt.figure(figsize=(16,5))

ax1 = plt.subplot(1,2,1)
ax1.set_xlabel('$t$')
ax1.set_ylabel('$y$')
ax1.grid(True)

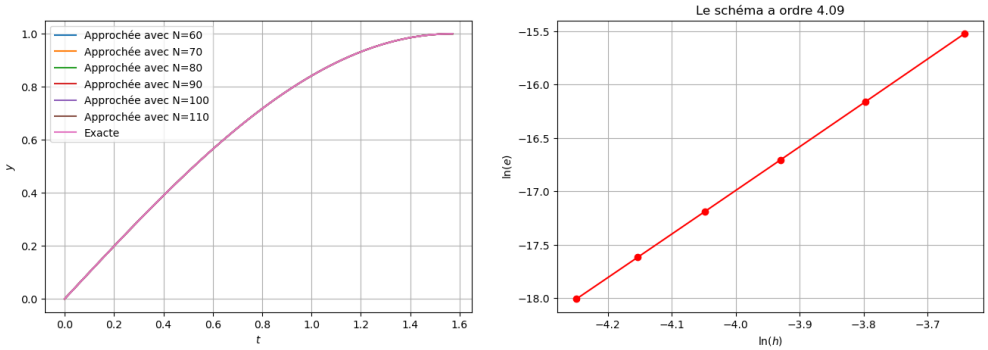

ax2 = plt.subplot(1,2,2)
ax2.set_xlabel(r'$\ln(h)$')
ax2.set_ylabel(r'$\ln(e)$')
ax2.grid(True)

N = 50
for k in range(6):
    N += 10
    tt = np.linspace(t0, tfinal, N + 1)
    H.append( tt[1] - tt[0] )
    yy = sol_exacte(tt)
    uu = RK(phi, tt, y0)
    err.append( np.linalg.norm(uu-yy,np.inf) )
    ax1.plot(tt,uu,label=f'Approchée avec N={N}')

ax1.plot(tt,yy,label='Exacte') # sur la grille la plus fine
ax1.legend()

ax2.plot( np.log(H), np.log(err), 'r-o')
ax2.set_title(f'Le schéma a ordre {np.polyfit(np.log(H),np.log(err),1)[0]:1.2f}');
```

$y(t) = \sin(t)$



Étudions la A-stabilité

```
c = list(But[:-1,0])
b = list(But[-1,1:])
A = But[:-1,1:]
```

On applique le schéma au problème de Cauchy $y'(t) = -\beta y(t)$ et $y(0) = 1$ avec $\beta > 0$ dont la solution est $y(t) = e^{-\beta t}$.

On obtient alors la relation de récurrence suivante :

```
u_n = sp.symbols(r'u_n')
u_np1 = sp.symbols(r'u_{n+1}')

dt = sp.symbols(r'h', positive=True)
x = sp.symbols(r'x', positive=True)

beta = sp.symbols(r'\beta', positive=True)

phi_A = lambda y : -beta*y

k0 = phi_A(u_n)
k1 = phi_A(u_n + dt* A[1,0]*k0)
k2 = phi_A(u_n + dt*(A[2,0]*k0 + A[2,1]*k1))
k3 = phi_A(u_n + dt*(A[3,0]*k0 + A[3,1]*k1 + A[3,2]*k2))
for i,k in enumerate([k0,k1,k2,k3]):
    display(Math(sp.latex(sp.Eq(sp.Symbol(f'k_{i}'), k.factor(beta*dt)))))

u_np1 = u_n + dt*(b[0]*k0 + b[1]*k1 + b[2]*k2 + b[3]*k3)
display(Math(sp.latex(sp.Eq(sp.Symbol('u_{n+1}'),u_np1.simplify()))))
```

$$k_0 = -\beta u_n$$

$$k_1 = -\frac{\beta u_n \left(-\beta h + 6\right)}{6}$$

$$k_2 = -\frac{\beta u_n \left(5\beta^2 h^2 - 10\beta h + 12\right)}{12}$$

$$k_3 = -\frac{\beta u_n \left(5\beta^3 h^3 + 10\beta^2 h^2 - 12\beta h + 12\right)}{12}$$

$$u_{n+1} = \frac{u_n \left(\beta^4 h^4 - 4\beta^3 h^3 + 12\beta^2 h^2 - 24\beta h + 24\right)}{24}$$

```
# Variante "semi-automatique"

# k_0 = sp.symbols(r'k_0')
# k_1 = sp.symbols(r'k_1')
# k_2 = sp.symbols(r'k_2')
# k_3 = sp.symbols(r'k_3')
# kk = [k_0, k_1, k_2, k_3]


# phi_A = lambda y: -beta*y

# Eqs = [ sp.Eq( kk[i] ,  phi_A(u_n + dt*sum([kk[j]*A[i,j] for j in range(s)])) ) for i in range(s) ]
# for eq in Eqs:
#     display(eq)

# sol = sp.solve( Eqs, kk ) # c'est un dictionnaire
# KK = [sol[kk[i]] for i in range(s)]
# schema = sp.Eq( u_np1 , u_n + dt*sum([KK[j]*b[j] for j in range(s)]) )
# # display(schema)

# display(Math(sp.latex(sp.Eq(sp.Symbol('u_{n+1}'),schema.rhs.simplify()))))
```

On a donc une suite du type $u_{n+1} = q(\beta h)u_n$. On peut expliciter u_n en fonction de u_0 et du produit βh (on notera x le produit βh) car $u_n = q(x)^n u_0$ avec $q(x) = u_{n+1}/u_n$:

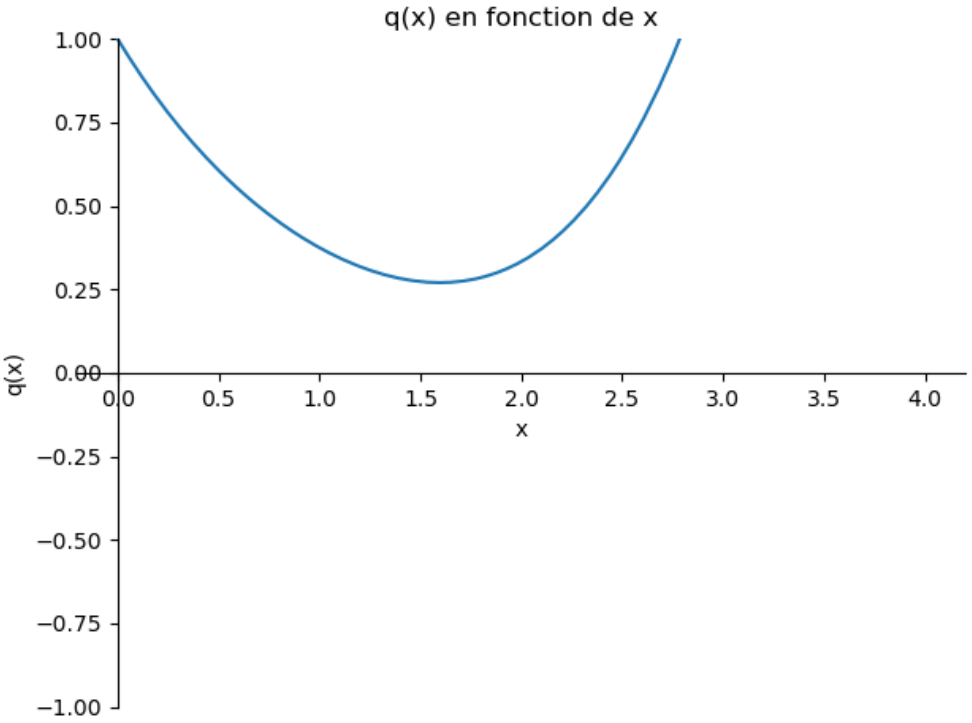
```
q = (u_np1/u_n).simplify().subs(beta*dt,x)
display(Math(sp.latex(sp.Eq(sp.Symbol('q(x)'),q.simplify()))))
```

$$q(x) = \frac{x^4}{24} - \frac{x^3}{6} + \frac{x^2}{2} - x + 1$$

La suite vérifie $\lim_{n\to\infty} u_n = 0$ ssi $|q(x)| < 1$. En étudiant brièvement la fonction $q(x)$, on trouve que $q(x) < 1$ ssi $x < x_0$ donné ci-dessous:.

```
sp.plot(q, (x,0,4), ylim=(-1,1),
            ylabel='q(x)', xlabel='x', title='q(x) en fonction de x')

sol = sp.solveset(q-1, x,domain=sp.S.Reals)
sol0, x0 = sol.args
display(Markdown(f"$\\beta h<{x0.evalf()}$"))
```



$\beta h < 2.78529356340528$

24.3. EXERCICE 3 : schémas MPL à 2 pas d'ordre maximal

1. Écrire le schéma multipas à $q = 2$ pas (comportant 5 coefficients).
2. Sans effectuer de calculs, répondre aux deux questions suivantes :
   - quel est l'ordre maximal d'un schéma à 2 pas convergent ?
   - quel est l'ordre maximal d'un schéma à 2 pas explicite convergent ?
3. Parmi ces schémas, calculer les coefficients du seul schéma consistant d'ordre 4. Montrer que le schéma ainsi trouvé est zéro-stable.
4. Tester empiriquement l'ordre de convergence du schéma ainsi obtenu sur le problème de Cauchy $y'(t) = 4t(t^2-1) + y(t)$ et $y(0) = 1$ pour $t \in [0, 1]$.
5. Considérons les schémas explicites à 2 pas (comportant 4 coefficients).
   - Montrer que le schéma consistant d'ordre 3 n'est pas zéro-stable.
   - Écrire la famille de schémas consistants d'ordre 2 zéro-stables en les paramètrant par a_0 (autrement-dit, indiquer sous quelles conditions sur a_0 le schéma est convergent d'ordre 2).

Un schéma implicite à $q = 2$ étapes s'écrit

$$u_{n+1} = a_0 u_n + a_1 u_{n-1} + h(b_{-1}\varphi_{n+1} + b_0\varphi_n + b_1\varphi_{n-1})$$

La première barrière de Dahlquist affirme qu'un schéma à $q = 2$ pas

- s'il est implicite, il est au mieux d'ordre $q + 2 = 4$;
- s'il est explicite, il est au mieux d'ordre $q = 2$.

Il contient les 5 paramètres a_0, a_1, b_{-1}, b_0, b_1, il est donc théoriquement possible de construire un schéma d'ordre 4 en résolvant le système linéaire de $q + 1 = 5$ équations suivant

$$\begin{cases} \xi(0) = a_0 + a_1 = 1 \\ \xi(1) = -a_1 + b_0 + b_1 + b_{-1} = 1 \\ \xi(2) = a_1 - 2\,(b_1 - b_{-1}) = 1 \\ \xi(3) = -a_1 + 3\,(b_1 + b_{-1}) = 1 \\ \xi(4) = a_1 - 4\,(b_1 - b_{-1}) = 1 \end{cases}$$

On trouve $a_0 = 0$, $a_1 = 1$, $b_{-1} = \frac{1}{3}$, $b_0 = \frac{4}{3}$ et $b_1 = \frac{1}{3}$.

Ci-dessous la résolution avec `sympy` :

```
q = 2

def xi(i, q):
    sa = sum((-j)**i * aa[j] for j in range(q))
    sb = b_m1 + sum((-j)**(i-1) * bb[j] for j in range(1, q))
    if i == 1:
        sb += bb[0]
    return (sa).factor() + (i * sb).factor()

# Définition des symboles
a_0, a_1  = sp.symbols('a_0 a_1 ', real=True)
b_0, b_1  = sp.symbols('b_0 b_1 ', real=True)
b_m1 = sp.symbols('b_{-1}', real=True)
aa = [a_0, a_1]
bb = [b_0, b_1]

systeme = [sp.Eq(xi(i, q),1) for i in range(2*q+1)]
display(Markdown(r"$\begin{cases}"+' '.join([sp.latex(s)+r"\\" for s in systeme]) + r"\end{cases}$"))

sol = sp.solve(systeme)
display(sol)
```

$$\begin{cases} a_0 + a_1 = 1 \\ -a_1 + b_0 + b_1 + b_{-1} = 1 \\ a_1 - 2\,(b_1 - b_{-1}) = 1 \\ -a_1 + 3\,(b_1 + b_{-1}) = 1 \\ a_1 - 4\,(b_1 - b_{-1}) = 1 \end{cases}$$

```
{a_0: 0, a_1: 1, b_0: 4/3, b_1: 1/3, b_{-1}: 1/3}
```

Le schéma s'écrit donc :

```
n = sp.Symbol('n')
txt = r"u_{n+1} = "
txt += sp.latex(  sum( aa[j].subs(sol) * sp.Symbol(f"u_{{{n-j}}}") for j in range(q))  )
txt += r"+ h \left("
txt += sp.latex( sum( bb[j].subs(sol) * sp.Symbol(f"\\varphi_{{{n-j}}}") for j in range(q)) )
txt += r"+"
txt += sp.latex( b_m1.subs(sol) * sp.Symbol(r"\varphi_{n+1}") )
txt += r"\right)"
display(Math( txt ))
```

$$u_{n+1} = u_{n-1} + h\left(\frac{\varphi_{n-1}}{3} + \frac{4\varphi_n}{3} + \frac{\varphi_{n+1}}{3}\right)$$

Vérifions si le schéma ainsi obtenu est zéro-stable en calculant les racines du premier polynôme caractéristique qui est défini par

$$\varrho(r) = r^{p+1} - \sum_{j=0}^{p} a_j r^{p-j} = r^2 - 1 = (r-1)(r+1)$$

La méthode est zéro-stable car ses racines sont de module égal à 1 mais simples.

Remarque : on reconnaît le schéma de Milne Simpson à deux pas.

Voici le code pour vérifier la zéro-stabilité avec `sympy` :

```
r = sp.Symbol('r')
p = q-1
rho_gen =  sp.poly(r**(p+1) - sum( aa[j] * r**(p-j) for j in range(p+1)), r)
rho =  sp.poly(r**(p+1) - sum( aa[j].subs(sol) * r**(p-j) for j in range(p+1)), r)


display(Markdown(f"Le premier polynome caractéristique est :"))
display(Math(sp.latex(rho_gen.as_expr() ,order='old') + "=" + sp.latex(rho.as_expr() ,order='old')))

display(Markdown(f"Ses racines sont :"))
racines = sp.solve(rho,r)
display(racines)
display(Markdown(f"dont les modules valent :"))
display([abs(rac).evalf() for rac in racines])
```

Le premier polynome caractéristique est :

$-a_1 - ra_0 + r^2 = -1 + r^2$

Ses racines sont :

```
[-1, 1]
```

dont les modules valent :

```
[1.00000000000000, 1.00000000000000]
```

```
# AUTOMATISATION : conversion des valeurs sympy en nombres flottants de NumPy et affectation
dico = {cle: float(valeur) for cle, valeur in sol.items()}
for cle, valeur in dico.items():
    if str(cle) == "b_{-1}":
        cle = "b_m1"
        globals()[cle] = valeur
    else :
        globals()[str(cle)] = valeur

# TEST de l'affectation
for var in ['a_0', 'a_1', 'b_0', 'b_1', 'b_m1']:
    print(f"{var} = {globals()[var]}")
```

```
a_0 = 0.0
a_1 = 1.0
b_0 = 1.3333333333333333
b_1 = 0.3333333333333333
b_m1 = 0.3333333333333333
```

```
####################################################
# EDO
####################################################
t0     = 0
tfinal = 1
y0     = 1

phi = lambda t,y: 4*t*(t**2-1)+y


####################################################
# solution exacte
####################################################
t   = sp.Symbol('t')
y   = sp.Function('y')
edo = sp.Eq( sp.diff(y(t),t) , phi(t,y(t)) )
solpar = sp.dsolve(edo, ics={y(t0): y0})
display(solpar)

sol_exacte = sp.lambdify(t,solpar.rhs,'numpy')

####################################################
# schéma
####################################################
def multipas(phi, tt, sol_exacte):
    h  = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    # initialisation avec solution exacte pour ordre de convergence
    uu[0:q] = sol_exacte(tt[0:q])
    # recurrence
    for i in range(q-1,len(tt) - 1):
        eq = lambda x : -x + a_0*uu[i] + a_1*uu[i-1] + h*(b_0*phi(tt[i],uu[i]) + b_1*phi(tt[i-1],uu[i-1])
↪   + b_m1*phi(tt[i+1],x) )
        tmp = fsolve ( eq , uu[i] )
        uu[i+1] = tmp[0]
    return uu

####################################################
# ordre
####################################################
H = []
err = []


plt.figure(figsize=(16,5))

ax1 = plt.subplot(1,2,1)
plt.xlabel('$t$')
plt.ylabel('$y$')
ax1.grid(True)

ax2 = plt.subplot(1,2,2)
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
ax2.grid(True)

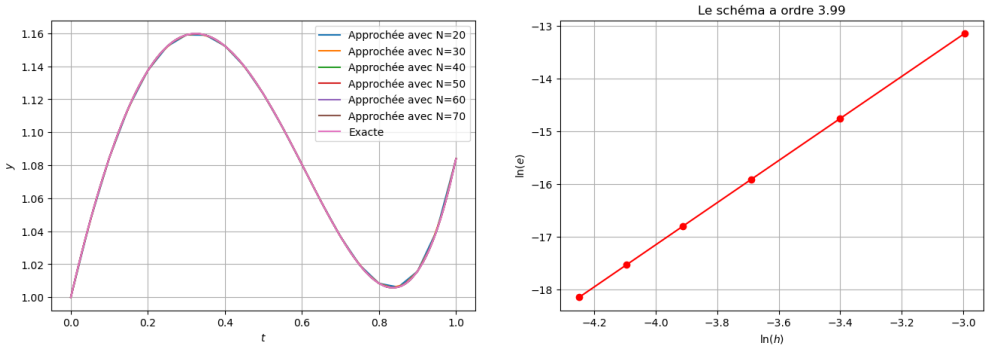

N = 10
for k in range(6):
    N += 10
    tt = np.linspace(t0, tfinal, N + 1)
    H.append( tt[1] - tt[0] )
    yy = sol_exacte(tt)
```

```
    uu = multipas(phi, tt, sol_exacte)
    err.append( np.linalg.norm(uu-yy,np.inf) )
    ax1.plot(tt,uu,label=f'Approchée avec N={N}')

ax1.plot(tt,yy,label='Exacte') # sur la grille la plus fine
ax1.legend()

ax2.plot( np.log(H), np.log(err), 'r-o')
plt.title(f'Le schéma a ordre {np.polyfit(np.log(H),np.log(err),1)[0]:1.2f}');
```

$y(t) = -4t^3 - 12t^2 - 20t + 21e^t - 20$



Calculons les paramètres pour les schémas explicites

Il contient les 4 paramètres a_0, a_1, b_0, b_1, il est donc théoriquement possible de construire un schéma d'ordre 3 en résolvant le système linéaire de 4 équations suivant

$$\begin{cases} \xi(0) = a_0 + a_1 = 1 \\ \xi(1) = -a_1 + b_0 + b_1 = 1 \\ \xi(2) = a_1 - 2b_1 = 1 \\ \xi(3) = -a_1 + 3b_1 = 1 \end{cases}$$

On trouve $a_0 = -4$, $a_1 = 5$, $b_0 = 4$ et $b_1 = 2$.

Ci-dessous la résolution avec `sympy` :

```
# Définition des symboles
a_0, a_1  = sp.symbols('a_0 a_1 ', real=True)
b_0, b_1  = sp.symbols('b_0 b_1 ', real=True)
aa = [a_0, a_1]
bb = [b_0, b_1]

b_m1 = 0

q = 2


systeme = [sp.Eq(xi(i, q),1) for i in range(4)]
display(Markdown(r"$\begin{cases}"+' '.join([sp.latex(s)+r"\\" for s in systeme]) + r"\end{cases}$"))

sol = sp.solve(systeme)
display(sol)
```

$$\begin{cases} a_0 + a_1 = 1 \\ -a_1 + b_0 + b_1 = 1 \\ a_1 - 2b_1 = 1 \\ -a_1 + 3b_1 = 1 \end{cases}$$

```
{a_0: -4, a_1: 5, b_0: 4, b_1: 2}
```

Le schéma s'écrit donc :

```
n = sp.Symbol('n')
txt = r"u_{n+1} = "
txt += sp.latex( sum( aa[j].subs(sol) * sp.Symbol(f"u_{{{n-j}}}") for j in range(q)) )
txt += r" + h \left("
txt += sp.latex(   b_m1 * sp.Symbol(r"\varphi_{n+1}")  +  sum( bb[j].subs(sol) *
 ↪  sp.Symbol(f"\\varphi_{{{n-j}}}") for j in range(q))  )
txt += r"\right)"
display(Math(txt))
```

$u_{n+1} = 5u_{n-1} - 4u_n + h\left(2\varphi_{n-1} + 4\varphi_n\right)$

Cépendant on sait, d'après la première barrière de Dalquist, qu'un schéma multipas linéaire explicite à q pas est au mieux d'ordre q. Donc un schéma à 2 pas ne peut pas être d'ordre 3. On peut le vérifier en calculant le polynôme caractéristique du schéma :

$$\varrho(r) = r^{p+1} - \sum_{j=0}^{p} a_j r^{p-j} = r^2 + 4r - 5 = (r-1)(r+5.)$$

La méthode n'est pas zéro-stable car une des racines n'est pas dans le cercle unitaire.

Voici le code pour vérifier la zéro-stabilité avec `sympy` :

```
r = sp.Symbol('r')
p = q-1
rho_gen =  sp.poly(r**(p+1) - sum( aa[j] * r**(p-j) for j in range(p+1)), r)
rho =  sp.poly(r**(p+1) - sum( aa[j].subs(sol) * r**(p-j) for j in range(p+1)), r)


display(Markdown(f"Le premier polynome caractéristique est :"))
display(Math(sp.latex(rho_gen.as_expr() ,order='old') + "=" + sp.latex(rho.as_expr() ,order='old')))

display(Markdown(f"Ses racines sont :"))
racines = sp.solve(rho,r)
display(racines)
display(Markdown(f"dont les modules valent :"))
display([abs(rac).evalf() for rac in racines])
```

Le premier polynome caractéristique est :

$-a_1 - ra_0 + r^2 = -5 + 4r + r^2$

Ses racines sont :

```
[-5, 1]
```

dont les modules valent :

```
[5.00000000000000, 1.00000000000000]
```

Pour trouver un schéma explicite d'ordre 2, on résout le système linéaire de 3 équations suivant qui contient donc un paramètre libre (ici a_0) :

$$\begin{cases} \xi(0) = a_0 + a_1 = 1 \\ \xi(1) = -a_1 + b_0 + b_1 = 1 \\ \xi(2) = a_1 - 2b_1 = 1 \end{cases}$$

On trouve $a_1 = 1 - a_0$, $b_0 = 2 - \frac{a_0}{2}$ et $b_1 = -\frac{a_0}{2}$.

Ci-dessous la résolution avec `sympy` :

```
systeme = [sp.Eq(xi(i, q),1) for i in range(3)]
display(Markdown(r"$\begin{cases}"+' '.join([sp.latex(s)+r"\\" for s in systeme]) + r"\end{cases}$"))

sol = sp.solve(systeme, (a_1, b_0, b_1))
display(sol)
```

$$\begin{cases} a_0 + a_1 = 1 \\ -a_1 + b_0 + b_1 = 1 \\ a_1 - 2b_1 = 1 \end{cases}$$

```
{a_1: 1 - a_0, b_0: 2 - a_0/2, b_1: -a_0/2}
```

Imposons maintenant la zéro-stabilite. Le premier polynôme caractéristique est

$$\varrho(r) = r^{p+1} - \sum_{j=0}^{p} a_j r^{p-j} = r^2 - ra_0 - a_1 = r^2 - ra_0 + (a_0 - 1)$$

Les racines sont 1 et $a_0 - 1$. La méthode est zéro-stable ssi $-1 \leq a_0 - 1 < 1$, c'est-à-dire $\boxed{0 \leq a_0 < 2}$.

Voici le code pour vérifier la zéro-stabilité avec `sympy` :

```
r = sp.Symbol('r')
p = q-1
rho_gen =  sp.poly(r**(p+1) - sum( aa[j] * r**(p-j) for j in range(p+1)), r)
rho =  sp.poly(r**(p+1) - sum( aa[j].subs(sol) * r**(p-j) for j in range(p+1)), r)


display(Markdown(f"Le premier polynome caractéristique est :"))
display(Math(sp.latex(rho_gen.as_expr() ,order='old') + "=" + sp.latex(rho.as_expr() ,order='old')))

display(Markdown(f"Ses racines sont :"))
racines = sp.solve(rho,r)
display(racines)
display(Markdown(f"dont les modules valent :"))
display([abs(rac).evalf() for rac in racines])
```

Le premier polynome caractéristique est :

$-a_1 - ra_0 + r^2 = -1 + a_0 - ra_0 + r^2$

Ses racines sont :

```
[1, a_0 - 1]
```

dont les modules valent :

```
[1.00000000000000, Abs(a_0 - 1)]
```

Le schéma s'écrit donc :

```
n = sp.Symbol('n')
txt = r"u_{n+1} = "
txt += sp.latex( sum( aa[j].subs(sol) * sp.Symbol(f"u_{{{n-j}}}") for j in range(q)) )
txt += r" + h \left("
txt += sp.latex(   b_m1 * sp.Symbol(r"\varphi_{n+1}")  +  sum( bb[j].subs(sol) *
 ↪  sp.Symbol(f"\\varphi_{{{n-j}}}") for j in range(q))  )
txt += r"\right)"
display(Math(txt))
```

$$u_{n+1} = a_0 u_n + u_{n-1} \cdot (1 - a_0) + h \left(-\frac{\varphi_{n-1} a_0}{2} + \varphi_n \left(2 - \frac{a_0}{2} \right) \right)$$

Chapitre 25.

2024 Contrôle Terminal session 1

2h30, deposer le notebook dans Moodle

```
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp
from scipy.optimize import fsolve
from IPython.display import display, Markdown, Math
```

25.1. EXERCICE 1 : Choix d'un schéma

Soient $\beta, \varepsilon \in \mathbb{R}$ et considérons le problème de Cauchy suivant

$$\begin{cases} y'(t) = \beta\Big(\sin(t) - y(t)\Big) + \cos(t) \quad \text{pour } t > 0, \\ y(0) = \varepsilon. \end{cases}$$

1. Calculer la solution exacte en fonction de β et ε.
2. Si $\beta = -5$ et $\varepsilon = 0$, quel type de schéma doit-on privilégier pour résoudre ce problème de Cauchy ?

Méthode 1 On peut résoudre ce problème de Cauchy directement en remarquant que l'edo se réécrit comme

$$y'(t) - \cos(t) = -\beta(y(t) - \sin(t)),$$

soit encore

$$\big(y(t) - \sin(t)\big)' = -\beta(y(t) - \sin(t)).$$

Si on pose $z(t) = y(t) - \sin(t)$ on a le problème de Cauchy

$$z'(t) = -\beta z(t)$$

avec $z(0) = y(0) - 0 = \varepsilon$. La solution est donc $z(t) = z(0)e^{-\beta t} = \varepsilon e^{-\beta t}$ et donc

$$\boxed{y(t)} = z(t) + \sin(t) \boxed{= \sin(t) + \varepsilon e^{-\beta t}}$$

Méthode 2 On peut résoudre le problème de Cauchy comme en L1 : la solution générale de l'EDO $a(t)y'(t) + b(t)y(t) = g(t)$ est donnée par

$$y(t) = Ce^{-A(t)} + B(t)e^{-A(t)}$$

où $A(t) = \int^t \frac{b(s)}{a(s)} ds$ et $B(t) = \int^t \frac{g(s)}{a(s)} e^{A(s)} ds$. Dans notre cas, on a $y'(t) + \beta y(t) = \beta \sin(t) + \cos(t)$, donc $A(t) = \int^t \beta ds = \beta t$ et $B(t) = \int^t e^{\beta s}(\beta \sin(s) + \cos(s)) ds = e^{\beta t} \sin(t)$. La solution générale est alors $y(t) = Ce^{-\beta t} + \sin(t)$. En utilisant la condition initiale $y(0) = \varepsilon$, on trouve $C = \varepsilon$ et enfin $y(t) = \sin(t) + \varepsilon e^{-\beta t}$.

Méthode 3 On peut calculer la solution en utilisant `sympy` :

```
t = sp.symbols('t')
y = sp.Function('y')(t)
beta = sp.symbols(r'\beta', real=True)
epsilon = sp.symbols(r'\varepsilon', real=True)
dydt = y.diff(t)
phi_sp = lambda t,z : beta*(sp.sin(t)-z)+sp.cos(t)
ode = sp.Eq(dydt, phi_sp(t,y) )
sol = sp.dsolve(ode, y, ics={y.subs(t, 0) : epsilon})
sol
```

$y(t) = \varepsilon e^{-\beta t} + \sin(t)$

Si $\beta < 0$, il s'agit d'un problème de Cauchy **numériquement mal posé** (c'est-à-dire, il est très sensible aux perturbations sur la donnée initiale). En effet,

- si $\varepsilon > 0$, $y(t) \sim \varepsilon e^{-\beta t} \xrightarrow[t\to\infty]{} +\infty$,
- si $\varepsilon < 0$, $y(t) \sim \varepsilon e^{-\beta t} \xrightarrow[t\to\infty]{} -\infty$,
- si $\varepsilon = 0$, $y(t) = \sin(t) \in [-1;1]$ pour tout t.

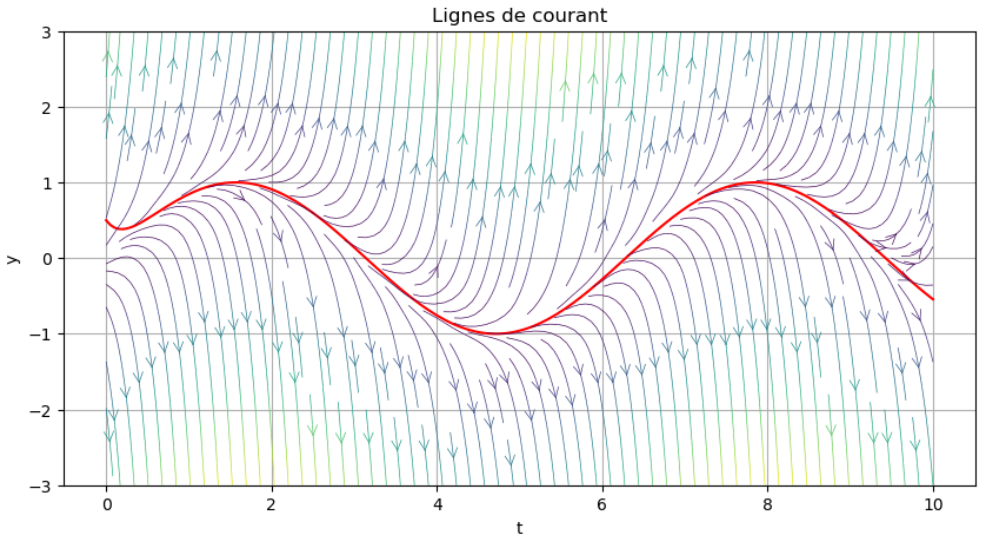
Une toute (petite) erreur de calcul aura le même effet qu'une perturbation de la condition initiale : on "réveille" le terme exponentielle. On doit donc choisir un **schéma d'ordre élevé** car les erreurs d'approximation amènent toujours à résoudre un problème perturbé.

```
t0     = 0
tfinal = 10

phi = lambda t,y: -5*(np.sin(t)-y) + np.cos(t)

N = 200
tt = np.linspace(t0, tfinal, N + 1)
func = sp.lambdify(t, sol.rhs.subs({epsilon:0.5, beta:5}), 'numpy')
yy = func(tt)

plt.figure(figsize=(10,5))
g1  = np.linspace(0,10,21)
g2  = np.linspace(-3,3,21)
T,Y = np.meshgrid(g1,g2)
DT, DY = 1, phi(T,Y)
plt.plot(tt,yy,color='red')
M = np.hypot(DT,DY)
plt.streamplot(T,Y, DT/M, DY/M, color=M, linewidth=0.5, cmap=plt.cm.viridis, density=2, arrowstyle='->',
 ↪  arrowsize=1.5)
plt.grid()
plt.xlabel('t')
plt.ylabel('y')
plt.title('Lignes de courant');
```



Pour bien comprendre, allons voir l'influence de l'ordre du schéma en comparant les approximations obtenues avec le schéma d'Euler explicite et un schéma RK aussi explicite d'ordre 4 dans le cas $\varepsilon = 0$ et $\beta = -5$. On remarque qu'initialement les approximations sont proches de la solution exacte, mais l'erreur d'approximation du schéma d'Euler explicite fait que la solution approchée correspond plutôt à une solution ayant $\varepsilon > 0$. Le schéma RK est plus précis mais finit par diverger lui aussi.

```
tt = np.linspace(0, 5, 300)

fig, (ax1, ax2) = plt.subplots(nrows=1,ncols=2,figsize=(16, 6))

# Solutions exactes pour différentes valeurs de epsilon et beta = -5
for epsi in [-1e-3, -1e-8, 0, 1e-8, 1e-3]:
    y_sol = sp.lambdify(t, sol.rhs.subs({beta:-5,epsilon:epsi}), 'numpy')
    ax1.plot(tt, y_sol(tt), label=fr'$\varepsilon={epsi}$')

ax1.set_xlabel('t')
ax1.set_ylabel('y')
ax1.grid()
ax1.legend()
ax1.axes.set_ylim(-5, 5)
ax1.set_title(r'Solutions exactes pour différents $\varepsilon$')

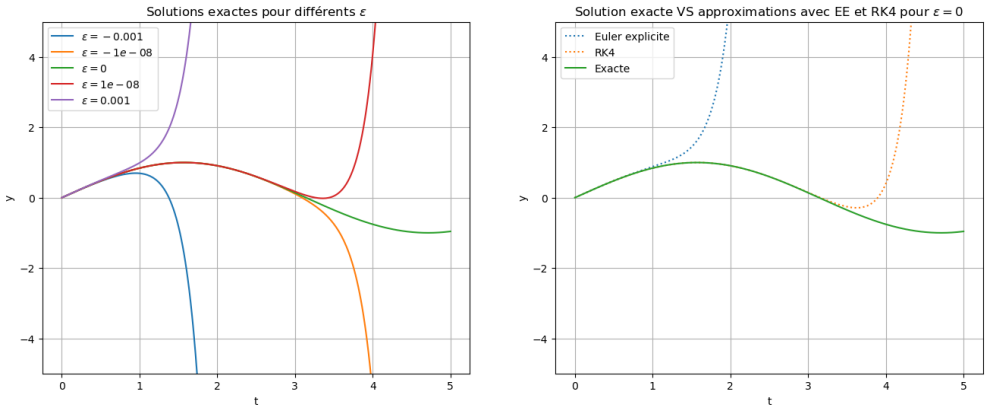

# Approximations du cas epsilon = 0 avec un schéma d'ordre 1 et un schéma d'ordre 4
def EE(phi, tt, y0):
    h  = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    uu[0] = y0
    for i in range(len(tt) - 1):
        uu[i+1] = uu[i] + h*phi(tt[i],uu[i])
    return uu

def RK4(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.zeros_like(tt)
    uu[0] = y0
    for n in range(len(tt)-1):
        k1 = phi( tt[n], uu[n] )
        k2 = phi( tt[n]+h/2 , uu[n]+k1*h/2 )
        k3 = phi( tt[n]+h/2 , uu[n]+h*k2/2 )
        k4 = phi( tt[n+1]   , uu[n]+h*k3 )
        uu[n+1] = uu[n] + (k1+2*k2+2*k3+k4)*h/6
    return uu
```

```
phi = lambda t,y : -5*(np.sin(t)-y)+np.cos(t)
y0 = 0

ax2.plot(tt, EE(phi, tt, y0), ':', label='Euler explicite')
ax2.plot(tt, RK4(phi, tt, y0), ':', label='RK4')
y_sol = sp.lambdify(t, sol.rhs.subs({beta:-5,epsilon:0}), 'numpy')
ax2.plot(tt, y_sol(tt), label='Exacte')

ax2.set_xlabel('t')
ax2.set_ylabel('y')
ax2.grid()
ax2.legend()
ax2.axes.set_ylim(-5, 5)
ax2.set_title(r"Solution exacte VS approximations avec EE et RK4 pour $\varepsilon=0$");
```



25.2. EXERCICE 2 : Schémas MPL à 3 pas d'ordre maximal

1. Écrire le schéma à $q = 3$ pas linéaire (il comportera donc 7 coefficients).
2. Il **n'est pas nécessaire** de répondre à cette question pour la suite de l'exercice.
   Les 7 équations linéaires $\xi(i) = 1$ pour $i = 0, \dots, 6$ constituent un système linéaire dont les 7 inconnues sont les coefficients du schéma. Calculer les coefficients de ce système consistant d'ordre 6 et montrer qu'il n'est pas zéro-stable.
3. Sans effectuer de calculs, répondre aux deux questions suivantes :
   - quel est l'ordre maximal d'un schéma à 3 pas implicite convergent ?
   - quel est l'ordre maximal d'un schéma à 3 pas explicite convergent ?
4. Dans la suite on fixera $b_{-1} = 0$ et $b_1 = -4/3$. De plus, a_1 sera fixé en fonction de **votre prénom et nom** (pour cela, remplacez mon prénom et mon nom par les vôtres dans la cellule ci-dessous).
   Déterminer les 4 paramètres manquants pour que le schéma soit d'ordre maximal et convergent.
5. Tester empiriquement l'ordre de convergence du schéma ainsi obtenu sur le problème de Cauchy $y'(t) = \sin^2(t)y(t)$ et $y(0) = 1$ pour $t \in [0, 2]$.

```
# ================================================
# On fixe $a_1$ pour le point 4
# ================================================
nom = "Faccanoni"
prenom = "Gloria"

S = sum([ord(c) for c in nom+prenom])
Lista = [ sp.Rational(num,12) for num in range(-24,13) ]
idx = S%len(Lista)
display(Markdown(f"**{nom} {prenom}** : $a_1 = {Lista[idx]}$"))
```

Faccanoni Gloria : $a_1 = 0$

Un schéma implicite à $q = 3$ étapes s'écrit

$$u_{n+1} = a_0 u_n + a_1 u_{n-1} + a_2 u_{n-2} + h(b_{-1}\varphi_{n+1} + b_0\varphi_n + b_1\varphi_{n-1} + b_2\varphi_{n-2})$$

Il contient les 7 paramètres a_0, a_1, a_2, b_{-1}, b_0, b_1, b_2, il est donc théoriquement possible de construire un schéma d'ordre 6 en résolvant le système linéaire de 7 équations et 7 inconnues suivant

$$\begin{cases} \xi(0) = a_0 + a_1 + a_2 = 1 \\ \xi(1) = -a_1 - 2a_2 + b_0 + b_1 + b_2 + b_{-1} = 1 \\ \xi(2) = a_1 + 4a_2 - 2\left(b_1 + 2b_2 - b_{-1}\right) = 1 \\ \xi(3) = -a_1 - 8a_2 + 3\left(b_1 + 4b_2 + b_{-1}\right) = 1 \\ \xi(4) = a_1 + 16a_2 - 4\left(b_1 + 8b_2 - b_{-1}\right) = 1 \\ \xi(5) = -a_1 - 32a_2 + 5\left(b_1 + 16b_2 + b_{-1}\right) = 1 \\ \xi(6) = a_1 + 64a_2 - 6\left(b_1 + 32b_2 - b_{-1}\right) = 1 \end{cases}$$

```
q = 3

def xi(i, q):
    sa = sum((-j)**i * aa[j] for j in range(q))
    sb = b_m1 + sum((-j)**(i-1) * bb[j] for j in range(1, q))
    if i == 1:
        sb += bb[0]
    return (sa).factor() + (i * sb).factor()


# Ordre maximal selon Dalquist
omega = q + 2 if q%2==0 else q + 1 # i = 0, ..., omega

# Définition des symboles
a_0, a_1, a_2 = sp.symbols('a_0 a_1 a_2', real=True)
b_0, b_1, b_2 = sp.symbols('b_0 b_1 b_2', real=True)
b_m1 = sp.symbols('b_{-1}', real=True)
aa = [a_0, a_1, a_2]
bb = [b_0, b_1, b_2]


# display(Markdown(f"Pour une méthode à q={q} pas, on affiche $\\xi(i)$ pour i = 0, ..., {2*q+1}"))
# for i in range(7):
#     display(Markdown(f"$\\xi({i}) = {sp.latex(xi(i, q))}$"))

display(Markdown(f"Une méthode à q={q} pas contient {2*q+1} paramètres, on peut donc résoudre le système
 ↪  de {2*q+1} équations suivant (garantissant la consistance d'ordre {2*q}) pour les déterminer :"))
systeme = [sp.Eq(xi(i, q),1) for i in range(7)]
display(Markdown(r"$\begin{cases}"+' '.join([sp.latex(s)+r"\\" for s in systeme]) + r"\end{cases}$"))
```

Une méthode à q=3 pas contient 7 paramètres, on peut donc résoudre le système de 7 équations suivant (garantissant la consistance d'ordre 6) pour les déterminer :

$$\begin{cases} a_0 + a_1 + a_2 = 1 \\ -a_1 - 2a_2 + b_0 + b_1 + b_2 + b_{-1} = 1 \\ a_1 + 4a_2 - 2\left(b_1 + 2b_2 - b_{-1}\right) = 1 \\ -a_1 - 8a_2 + 3\left(b_1 + 4b_2 + b_{-1}\right) = 1 \\ a_1 + 16a_2 - 4\left(b_1 + 8b_2 - b_{-1}\right) = 1 \\ -a_1 - 32a_2 + 5\left(b_1 + 16b_2 + b_{-1}\right) = 1 \\ a_1 + 64a_2 - 6\left(b_1 + 32b_2 - b_{-1}\right) = 1 \end{cases}$$

Calculons la solution de ce système linéaire :

```
sol = sp.solve(systeme)
display(sol)
```

```
{a_0: -27/11,
 a_1: 27/11,
 a_2: 1,
 b_0: 27/11,
 b_1: 27/11,
 b_2: 3/11,
 b_{-1}: 3/11}
```

Cépendant on sait, d'après la première barrière de Dalquist, qu'un schéma multipas linéaire implicite à q pas avec q impair est au mieux d'ordre $q+1$. Donc un schéma à 3 pas ne peut pas être d'ordre 6. On peut le vérifier en calculant le polynôme caractéristique du schéma :

$$\varrho(r) = r^{p+1} - \sum_{j=0}^{p} a_j r^{p-j} = r^3 - a_0 r^2 - a_1 r - a_2 = (r-1)(r^2 + \frac{38}{11} r + 1).$$

La méthode n'est pas zéro-stable car une des racines n'est pas dans le cercle unité (voir le calcul des racines ci-dessous) :

```
r = sp.Symbol('r')
p = q-1
rho =  sp.poly(r**(p+1) - sum( aa[j].subs(sol) * r**(p-j) for j in range(p+1)), r)


display(Markdown(f"Le premier polynome caractéristique est :"))
txt = sp.latex(sp.Eq(sp.symbols(r'\varrho(r)'),rho.as_expr()),order='old')
txt += "=" + sp.latex(rho.as_expr().factor(),order='old')
display(Math(txt))

display(Markdown(f"Ses racines sont :"))
racines = sp.solve(rho,r)
display(racines)
display(Markdown(f"dont les modules valent :"))
display([abs(rac).evalf() for rac in racines])
```

Le premier polynome caractéristique est :

$$\varrho(r) = -1 - \frac{27r}{11} + \frac{27r^2}{11} + r^3 = \frac{(-1+r)\,(11 + 38r + 11r^2)}{11}$$

Ses racines sont :

```
[1, -19/11 - 4*sqrt(15)/11, -19/11 + 4*sqrt(15)/11]
```

dont les modules valent :

```
[1.00000000000000, 3.13563030771179, 0.318915146833667]
```

La première barrière de Dahlquist affirme qu'un schéma à $q = 3$ pas

- s'il est implicite, il est au mieux d'ordre $q + 1 = 4$;
- s'il est explicite, il est au mieux d'ordre $q = 3$.

```
for ell in Lista:
    choix = {b_m1 : 0 , a_1 : ell , b_1 : sp.Rational(-4,3)}
    systeme = [sp.Eq(xi(i, q).subs(choix),1) for i in range(4)]
    sol = sp.solve(systeme, [a_0, a_2, b_0, b_2])
    display(Markdown(f"Choix : ${sp.latex(choix)}$"))
    display(Markdown(r"Système : $\begin{cases}"+' '.join([sp.latex(s)+r"\\" for s in systeme]) +
↪   r"\end{cases}$"))
    display(Markdown(f"Solution : ${sp.latex(sol)}$"))
    display(Markdown("=============================="))
```

Choix : $\left\{a_{1} : -2,\ b_{1} : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_{0} + a_{2} - 2 = 1 \\ -2a_{2} + b_{0} + b_{2} + \frac{2}{3} = 1 \\ 4a_{2} - 4b_{2} + \frac{2}{3} = 1 \\ -8a_{2} + 12b_{2} - 2 = 1 \end{cases}$

Solution : $\left\{a_{0} : 2,\ a_{2} : 1,\ b_{0} : \frac{17}{12},\ b_{2} : \frac{11}{12}\right\}$

==========

Choix : $\left\{a_{1} : -\frac{23}{12},\ b_{1} : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_{0} + a_{2} - \frac{23}{12} = 1 \\ -2a_{2} + b_{0} + b_{2} + \frac{7}{12} = 1 \\ 4a_{2} - 4b_{2} + \frac{3}{4} = 1 \\ -8a_{2} + 12b_{2} - \frac{25}{12} = 1 \end{cases}$

Solution : $\left\{a_{0} : \frac{47}{24},\ a_{2} : \frac{23}{24},\ b_{0} : \frac{23}{16},\ b_{2} : \frac{43}{48}\right\}$

==========

Choix : $\left\{a_{1} : -\frac{11}{6},\ b_{1} : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_{0} + a_{2} - \frac{11}{6} = 1 \\ -2a_{2} + b_{0} + b_{2} + \frac{1}{2} = 1 \\ 4a_{2} - 4b_{2} + \frac{5}{6} = 1 \\ -8a_{2} + 12b_{2} - \frac{13}{6} = 1 \end{cases}$

Solution : $\left\{a_{0} : \frac{23}{12},\ a_{2} : \frac{11}{12},\ b_{0} : \frac{35}{24},\ b_{2} : \frac{7}{8}\right\}$

==========

Choix : $\left\{a_{1} : -\frac{7}{4},\ b_{1} : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_{0} + a_{2} - \frac{7}{4} = 1 \\ -2a_{2} + b_{0} + b_{2} + \frac{5}{12} = 1 \\ 4a_{2} - 4b_{2} + \frac{11}{12} = 1 \\ -8a_{2} + 12b_{2} - \frac{9}{4} = 1 \end{cases}$

Solution : $\left\{a_{0} : \frac{15}{8},\ a_{2} : \frac{7}{8},\ b_{0} : \frac{71}{48},\ b_{2} : \frac{41}{48}\right\}$

==========

Choix : $\left\{a_{1} : -\frac{5}{3},\ b_{1} : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_{0} + a_{2} - \frac{5}{3} = 1 \\ -2a_{2} + b_{0} + b_{2} + \frac{1}{3} = 1 \\ 4a_{2} - 4b_{2} + 1 = 1 \\ -8a_{2} + 12b_{2} - \frac{7}{3} = 1 \end{cases}$

Solution : $\left\{a_{0} : \frac{11}{6},\ a_{2} : \frac{5}{6},\ b_{0} : \frac{3}{2},\ b_{2} : \frac{5}{6}\right\}$

==========

Choix : $\left\{a_{1} : -\frac{19}{12},\ b_{1} : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_{0} + a_{2} - \frac{19}{12} = 1 \\ -2a_{2} + b_{0} + b_{2} + \frac{1}{4} = 1 \\ 4a_{2} - 4b_{2} + \frac{13}{12} = 1 \\ -8a_{2} + 12b_{2} - \frac{29}{12} = 1 \end{cases}$

Solution : $\left\{a_{0} : \frac{43}{24},\ a_{2} : \frac{19}{24},\ b_{0} : \frac{73}{48},\ b_{2} : \frac{13}{16}\right\}$

==========

Choix : $\left\{a_{1} : -\frac{3}{2},\ b_{1} : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_{0} + a_{2} - \frac{3}{2} = 1 \\ -2a_{2} + b_{0} + b_{2} + \frac{1}{6} = 1 \\ 4a_{2} - 4b_{2} + \frac{7}{6} = 1 \\ -8a_{2} + 12b_{2} - \frac{5}{2} = 1 \end{cases}$

Solution : $\left\{a_{0} : \frac{7}{4},\ a_{2} : \frac{3}{4},\ b_{0} : \frac{37}{24},\ b_{2} : \frac{19}{24}\right\}$

==========

Choix : $\left\{a_{1} : -\frac{17}{12},\ b_{1} : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_{0} + a_{2} - \frac{17}{12} = 1 \\ -2a_{2} + b_{0} + b_{2} + \frac{1}{12} = 1 \\ 4a_{2} - 4b_{2} + \frac{5}{4} = 1 \\ -8a_{2} + 12b_{2} - \frac{31}{12} = 1 \end{cases}$

Solution : $\left\{a_{0} : \frac{41}{24},\ a_{2} : \frac{17}{24},\ b_{0} : \frac{25}{16},\ b_{2} : \frac{37}{48}\right\}$

==========

Choix : $\left\{a_{1} : -\frac{4}{3},\ b_{1} : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_{0} + a_{2} - \frac{4}{3} = 1 \\ -2a_{2} + b_{0} + b_{2} = 1 \\ 4a_{2} - 4b_{2} + \frac{4}{3} = 1 \\ -8a_{2} + 12b_{2} - \frac{8}{3} = 1 \end{cases}$

Solution : $\left\{a_{0} : \frac{5}{3},\ a_{2} : \frac{2}{3},\ b_{0} : \frac{19}{12},\ b_{2} : \frac{3}{4}\right\}$

==========

Choix : $\left\{a_{1} : -\frac{5}{4},\ b_{1} : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_{0} + a_{2} - \frac{5}{4} = 1 \\ -2a_{2} + b_{0} + b_{2} - \frac{1}{12} = 1 \\ 4a_{2} - 4b_{2} + \frac{17}{12} = 1 \\ -8a_{2} + 12b_{2} - \frac{11}{4} = 1 \end{cases}$

Solution : $\left\{a_{0} : \frac{13}{8},\ a_{2} : \frac{5}{8},\ b_{0} : \frac{77}{48},\ b_{2} : \frac{35}{48}\right\}$

==========

Choix : $\left\{a_{1} : -\frac{7}{6},\ b_{1} : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_{0} + a_{2} - \frac{7}{6} = 1 \\ -2a_{2} + b_{0} + b_{2} - \frac{1}{6} = 1 \\ 4a_{2} - 4b_{2} + \frac{3}{2} = 1 \\ -8a_{2} + 12b_{2} - \frac{17}{6} = 1 \end{cases}$

Solution : $\left\{a_0 : \frac{19}{12},\ a_2 : \frac{7}{12},\ b_0 : \frac{13}{8},\ b_2 : \frac{17}{24}\right\}$

==========

Choix : $\left\{a_1 : -\frac{13}{12},\ b_1 : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_0 + a_2 - \frac{13}{12} = 1 \\ -2a_2 + b_0 + b_2 - \frac{1}{4} = 1 \\ 4a_2 - 4b_2 + \frac{19}{12} = 1 \\ -8a_2 + 12b_2 - \frac{35}{12} = 1 \end{cases}$

Solution : $\left\{a_0 : \frac{37}{24},\ a_2 : \frac{13}{24},\ b_0 : \frac{79}{48},\ b_2 : \frac{11}{16}\right\}$

==========

Choix : $\left\{a_1 : -1,\ b_1 : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_0 + a_2 - 1 = 1 \\ -2a_2 + b_0 + b_2 - \frac{1}{3} = 1 \\ 4a_2 - 4b_2 + \frac{5}{3} = 1 \\ -8a_2 + 12b_2 - 3 = 1 \end{cases}$

Solution : $\left\{a_0 : \frac{3}{2},\ a_2 : \frac{1}{2},\ b_0 : \frac{5}{3},\ b_2 : \frac{2}{3}\right\}$

==========

Choix : $\left\{a_1 : -\frac{11}{12},\ b_1 : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_0 + a_2 - \frac{11}{12} = 1 \\ -2a_2 + b_0 + b_2 - \frac{5}{12} = 1 \\ 4a_2 - 4b_2 + \frac{7}{4} = 1 \\ -8a_2 + 12b_2 - \frac{37}{12} = 1 \end{cases}$

Solution : $\left\{a_0 : \frac{35}{24},\ a_2 : \frac{11}{24},\ b_0 : \frac{27}{16},\ b_2 : \frac{31}{48}\right\}$

==========

Choix : $\left\{a_1 : -\frac{5}{6},\ b_1 : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_0 + a_2 - \frac{5}{6} = 1 \\ -2a_2 + b_0 + b_2 - \frac{1}{2} = 1 \\ 4a_2 - 4b_2 + \frac{11}{6} = 1 \\ -8a_2 + 12b_2 - \frac{19}{6} = 1 \end{cases}$

Solution : $\left\{a_0 : \frac{17}{12},\ a_2 : \frac{5}{12},\ b_0 : \frac{41}{24},\ b_2 : \frac{5}{8}\right\}$

==========

Choix : $\left\{a_1 : -\frac{3}{4},\ b_1 : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_0 + a_2 - \frac{3}{4} = 1 \\ -2a_2 + b_0 + b_2 - \frac{7}{12} = 1 \\ 4a_2 - 4b_2 + \frac{23}{12} = 1 \\ -8a_2 + 12b_2 - \frac{13}{4} = 1 \end{cases}$

Solution : $\left\{a_0 : \frac{11}{8},\ a_2 : \frac{3}{8},\ b_0 : \frac{83}{48},\ b_2 : \frac{29}{48}\right\}$

==========

Choix : $\left\{a_1 : -\frac{2}{3},\ b_1 : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_0 + a_2 - \frac{2}{3} = 1 \\ -2a_2 + b_0 + b_2 - \frac{2}{3} = 1 \\ 4a_2 - 4b_2 + 2 = 1 \\ -8a_2 + 12b_2 - \frac{10}{3} = 1 \end{cases}$

Solution : $\left\{a_0 : \frac{4}{3},\ a_2 : \frac{1}{3},\ b_0 : \frac{7}{4},\ b_2 : \frac{7}{12}\right\}$

==========

Choix : $\left\{a_1 : -\frac{7}{12},\ b_1 : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_0 + a_2 - \frac{7}{12} = 1 \\ -2a_2 + b_0 + b_2 - \frac{3}{4} = 1 \\ 4a_2 - 4b_2 + \frac{25}{12} = 1 \\ -8a_2 + 12b_2 - \frac{41}{12} = 1 \end{cases}$

Solution : $\left\{a_0 : \frac{31}{24},\ a_2 : \frac{7}{24},\ b_0 : \frac{85}{48},\ b_2 : \frac{9}{16}\right\}$

==========

Choix : $\left\{a_1 : -\frac{1}{2},\ b_1 : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_0 + a_2 - \frac{1}{2} = 1 \\ -2a_2 + b_0 + b_2 - \frac{5}{6} = 1 \\ 4a_2 - 4b_2 + \frac{13}{6} = 1 \\ -8a_2 + 12b_2 - \frac{7}{2} = 1 \end{cases}$

Solution : $\left\{a_0 : \frac{5}{4},\ a_2 : \frac{1}{4},\ b_0 : \frac{43}{24},\ b_2 : \frac{13}{24}\right\}$

==========

Choix : $\left\{a_1 : -\frac{5}{12},\ b_1 : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_0 + a_2 - \frac{5}{12} = 1 \\ -2a_2 + b_0 + b_2 - \frac{11}{12} = 1 \\ 4a_2 - 4b_2 + \frac{9}{4} = 1 \\ -8a_2 + 12b_2 - \frac{43}{12} = 1 \end{cases}$

Solution : $\left\{a_0 : \frac{29}{24},\ a_2 : \frac{5}{24},\ b_0 : \frac{29}{16},\ b_2 : \frac{25}{48}\right\}$

==========

Choix : $\left\{a_1 : -\frac{1}{3},\ b_1 : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_0 + a_2 - \frac{1}{3} = 1 \\ -2a_2 + b_0 + b_2 - 1 = 1 \\ 4a_2 - 4b_2 + \frac{7}{3} = 1 \\ -8a_2 + 12b_2 - \frac{11}{3} = 1 \end{cases}$

Solution : $\left\{a_0 : \frac{7}{6},\ a_2 : \frac{1}{6},\ b_0 : \frac{11}{6},\ b_2 : \frac{1}{2}\right\}$

==========

Choix : $\left\{a_1 : -\frac{1}{4},\ b_1 : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_0 + a_2 - \frac{1}{4} = 1 \\ -2a_2 + b_0 + b_2 - \frac{13}{12} = 1 \\ 4a_2 - 4b_2 + \frac{29}{12} = 1 \\ -8a_2 + 12b_2 - \frac{15}{4} = 1 \end{cases}$

Solution : $\left\{a_0 : \frac{9}{8},\ a_2 : \frac{1}{8},\ b_0 : \frac{89}{48},\ b_2 : \frac{23}{48}\right\}$

==========

Choix : $\left\{a_1 : -\frac{1}{6},\ b_1 : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_0 + a_2 - \frac{1}{6} = 1 \\ -2a_2 + b_0 + b_2 - \frac{7}{6} = 1 \\ 4a_2 - 4b_2 + \frac{5}{2} = 1 \\ -8a_2 + 12b_2 - \frac{23}{6} = 1 \end{cases}$

Solution : $\left\{a_0 : \frac{13}{12},\ a_2 : \frac{1}{12},\ b_0 : \frac{15}{8},\ b_2 : \frac{11}{24}\right\}$

==========

Choix : $\left\{a_1 : -\frac{1}{12},\ b_1 : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_0 + a_2 - \frac{1}{12} = 1 \\ -2a_2 + b_0 + b_2 - \frac{5}{4} = 1 \\ 4a_2 - 4b_2 + \frac{31}{12} = 1 \\ -8a_2 + 12b_2 - \frac{47}{12} = 1 \end{cases}$

Solution : $\left\{a_0 : \frac{25}{24},\ a_2 : \frac{1}{24},\ b_0 : \frac{91}{48},\ b_2 : \frac{7}{16}\right\}$

==========

Choix : $\left\{a_1 : 0,\ b_1 : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_0 + a_2 = 1 \\ -2a_2 + b_0 + b_2 - \frac{4}{3} = 1 \\ 4a_2 - 4b_2 + \frac{8}{3} = 1 \\ -8a_2 + 12b_2 - 4 = 1 \end{cases}$

Solution : $\left\{a_0 : 1,\ a_2 : 0,\ b_0 : \frac{23}{12},\ b_2 : \frac{5}{12}\right\}$

==========

Choix : $\left\{a_1 : \frac{1}{12},\ b_1 : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_0 + a_2 + \frac{1}{12} = 1 \\ -2a_2 + b_0 + b_2 - \frac{17}{12} = 1 \\ 4a_2 - 4b_2 + \frac{11}{4} = 1 \\ -8a_2 + 12b_2 - \frac{49}{12} = 1 \end{cases}$

Solution : $\left\{a_0 : \frac{23}{24},\ a_2 : -\frac{1}{24},\ b_0 : \frac{31}{16},\ b_2 : \frac{19}{48}\right\}$

==========

Choix : $\left\{a_1 : \frac{1}{6},\ b_1 : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_0 + a_2 + \frac{1}{6} = 1 \\ -2a_2 + b_0 + b_2 - \frac{3}{2} = 1 \\ 4a_2 - 4b_2 + \frac{17}{6} = 1 \\ -8a_2 + 12b_2 - \frac{25}{6} = 1 \end{cases}$

Solution : $\left\{a_0 : \frac{11}{12},\ a_2 : -\frac{1}{12},\ b_0 : \frac{47}{24},\ b_2 : \frac{3}{8}\right\}$

==========

Choix : $\left\{a_1 : \frac{1}{4},\ b_1 : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_0 + a_2 + \frac{1}{4} = 1 \\ -2a_2 + b_0 + b_2 - \frac{19}{12} = 1 \\ 4a_2 - 4b_2 + \frac{35}{12} = 1 \\ -8a_2 + 12b_2 - \frac{17}{4} = 1 \end{cases}$

Solution : $\left\{a_0 : \frac{7}{8},\ a_2 : -\frac{1}{8},\ b_0 : \frac{95}{48},\ b_2 : \frac{17}{48}\right\}$

==========

Choix : $\left\{a_1 : \frac{1}{3},\ b_1 : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_0 + a_2 + \frac{1}{3} = 1 \\ -2a_2 + b_0 + b_2 - \frac{5}{3} = 1 \\ 4a_2 - 4b_2 + 3 = 1 \\ -8a_2 + 12b_2 - \frac{13}{3} = 1 \end{cases}$

Solution : $\left\{a_0 : \frac{5}{6},\ a_2 : -\frac{1}{6},\ b_0 : 2,\ b_2 : \frac{1}{3}\right\}$

==========

Choix : $\left\{a_1 : \frac{5}{12},\ b_1 : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_0 + a_2 + \frac{5}{12} = 1 \\ -2a_2 + b_0 + b_2 - \frac{7}{4} = 1 \\ 4a_2 - 4b_2 + \frac{37}{12} = 1 \\ -8a_2 + 12b_2 - \frac{53}{12} = 1 \end{cases}$

Solution : $\left\{a_0 : \frac{19}{24},\ a_2 : -\frac{5}{24},\ b_0 : \frac{97}{48},\ b_2 : \frac{5}{16}\right\}$

==========

Choix : $\left\{a_1 : \frac{1}{2},\ b_1 : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_0 + a_2 + \frac{1}{2} = 1 \\ -2a_2 + b_0 + b_2 - \frac{11}{6} = 1 \\ 4a_2 - 4b_2 + \frac{19}{6} = 1 \\ -8a_2 + 12b_2 - \frac{9}{2} = 1 \end{cases}$

Solution : $\left\{a_0 : \frac{3}{4},\ a_2 : -\frac{1}{4},\ b_0 : \frac{49}{24},\ b_2 : \frac{7}{24}\right\}$

==========

Choix : $\left\{a_1 : \frac{7}{12},\ b_1 : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_0 + a_2 + \frac{7}{12} = 1 \\ -2a_2 + b_0 + b_2 - \frac{23}{12} = 1 \\ 4a_2 - 4b_2 + \frac{13}{4} = 1 \\ -8a_2 + 12b_2 - \frac{55}{12} = 1 \end{cases}$

Solution : $\left\{a_0 : \frac{17}{24},\ a_2 : -\frac{7}{24},\ b_0 : \frac{33}{16},\ b_2 : \frac{13}{48}\right\}$

==========

Choix : $\left\{a_1 : \frac{2}{3},\ b_1 : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_0 + a_2 + \frac{2}{3} = 1 \\ -2a_2 + b_0 + b_2 - 2 = 1 \\ 4a_2 - 4b_2 + \frac{10}{3} = 1 \\ -8a_2 + 12b_2 - \frac{14}{3} = 1 \end{cases}$

Solution : $\left\{a_0 : \frac{2}{3},\ a_2 : -\frac{1}{3},\ b_0 : \frac{25}{12},\ b_2 : \frac{1}{4}\right\}$

==========

Choix : $\left\{a_1 : \frac{3}{4},\ b_1 : -\frac{4}{3},\ b_{-1} : 0\right\}$

Système : $\begin{cases} a_0 + a_2 + \frac{3}{4} = 1 \\ -2a_2 + b_0 + b_2 - \frac{25}{12} = 1 \\ 4a_2 - 4b_2 + \frac{41}{12} = 1 \\ -8a_2 + 12b_2 - \frac{19}{4} = 1 \end{cases}$

Solution : $\{a_0 : \frac{5}{8},\ a_2 : -\frac{3}{8},\ b_0 : \frac{101}{48},\ b_2 : \frac{11}{48}\}$

==========

Choix : $\{a_1 : \frac{5}{6},\ b_1 : -\frac{4}{3},\ b_{-1} : 0\}$

Système : $\begin{cases} a_0 + a_2 + \frac{5}{6} = 1 \\ -2a_2 + b_0 + b_2 - \frac{13}{6} = 1 \\ 4a_2 - 4b_2 + \frac{7}{2} = 1 \\ -8a_2 + 12b_2 - \frac{29}{6} = 1 \end{cases}$

Solution : $\{a_0 : \frac{7}{12},\ a_2 : -\frac{5}{12},\ b_0 : \frac{17}{8},\ b_2 : \frac{5}{24}\}$

==========

Choix : $\{a_1 : \frac{11}{12},\ b_1 : -\frac{4}{3},\ b_{-1} : 0\}$

Système : $\begin{cases} a_0 + a_2 + \frac{11}{12} = 1 \\ -2a_2 + b_0 + b_2 - \frac{9}{4} = 1 \\ 4a_2 - 4b_2 + \frac{43}{12} = 1 \\ -8a_2 + 12b_2 - \frac{59}{12} = 1 \end{cases}$

Solution : $\{a_0 : \frac{13}{24},\ a_2 : -\frac{11}{24},\ b_0 : \frac{103}{48},\ b_2 : \frac{3}{16}\}$

==========

Choix : $\{a_1 : 1,\ b_1 : -\frac{4}{3},\ b_{-1} : 0\}$

Système : $\begin{cases} a_0 + a_2 + 1 = 1 \\ -2a_2 + b_0 + b_2 - \frac{7}{3} = 1 \\ 4a_2 - 4b_2 + \frac{11}{3} = 1 \\ -8a_2 + 12b_2 - 5 = 1 \end{cases}$

Solution : $\{a_0 : \frac{1}{2},\ a_2 : -\frac{1}{2},\ b_0 : \frac{13}{6},\ b_2 : \frac{1}{6}\}$

==========

Le cas correspondant à mon prénom et nom est le suivant :

```
# voir la cellule initiale pour fixer le paramètre
choix = {b_m1 : 0 , a_1 : Lista[idx] , b_1 : sp.Rational(-4,3)}

display(Markdown(f"Pour le choix ${sp.latex(choix)}$"))

systeme = [sp.Eq(xi(i, q).subs(choix),1) for i in range(4)]
display(Markdown("le système de 4 équations devient :"))
display(Markdown(r"$\begin{cases}"+' '.join([sp.latex(s)+r"\\" for s in systeme]) + r"\end{cases}$"))
```

Pour le choix $\{a_1 : 0,\ b_1 : -\frac{4}{3},\ b_{-1} : 0\}$

le système de 4 équations devient :

$\begin{cases} a_0 + a_2 = 1 \\ -2a_2 + b_0 + b_2 - \frac{4}{3} = 1 \\ 4a_2 - 4b_2 + \frac{8}{3} = 1 \\ -8a_2 + 12b_2 - 4 = 1 \end{cases}$

Ces 4 équations permettent d'exprimer b_0, b_2, a_0 et a_2 en fonction de a_1, b_1 et b_{-1} :

```
sol = sp.solve(systeme, [a_0, a_2, b_0, b_2])
# Un seul dictionnaire avec toutes les valeurs:
sol.update(choix)
display(sol)
```

```
{a_0: 1, a_2: 0, b_0: 23/12, b_2: 5/12, b_{-1}: 0, a_1: 0, b_1: -4/3}
```

Vérifions si le schéma ainsi obtenu est zéro-stable en calculant les racines du premier polynôme caractéristique.

Le premier polynôme caractéristique est

$$\varrho(r) = r^{p+1} - \sum_{j=0}^{p} a_j r^{p-j} = r^3 - r^2 = r^3 - a_0 r^2 - a_1 r - a_2.$$

La méthode est zéro-stable ssi ses racines sont de module inférieur ou égal à 1 et si elles sont simples lorsqu'elles ont module égal à 1. Ce qui est bien le cas ici :

```
r = sp.Symbol('r')
p = q-1
rho =  sp.poly(r**(p+1) - sum( aa[j].subs(sol).subs(choix) * r**(p-j) for j in range(p+1)), r)


display(Markdown(f"Le premier polynome caractéristique est :"))
display(Math(sp.latex(rho.as_expr() ,order='old')))

display(Markdown(f"Ses racines, avec leur multiplicité, sont :"))
racines = sp.roots(rho,r)
display(racines)
display([float(abs(rac)) for rac in racines.keys()])
```

Le premier polynome caractéristique est :

$-r^2 + r^3$

Ses racines, avec leur multiplicité, sont :

```
{1: 1, 0: 2}
```

```
[1.0, 0.0]
```

Remarque : juste en considérant la condition de consistance, on peut factoriser le polynôme caractéristique par $r-1$, car $a_0 = 1 - a_1 - a_2$ ainsi :

$$\begin{aligned}\varrho(r) &= (r-1)\left(r^2 + \left(a_2 - \frac{1}{2}\right) r + a_2\right) \\ &= (r-1)\left(r - \frac{1 - 2a_2 - \sqrt{4a_2^2 - 20a_2 + 1}}{4}\right)\left(r - \frac{1 - 2a_2 + \sqrt{4a_2^2 - 20a_2 + 1}}{4}\right)\end{aligned}$$

```
# AUTOMATISATION : convertion des valeurs sympy en nombres flottants de NumPy et affectation
dico = {cle: float(valeur) for cle, valeur in sol.items()}
for cle, valeur in dico.items():
    globals()[str(cle)] = valeur


################################################
# EDO
################################################
t0     = 0
tfinal = 2
y0     = 1

phi_np = lambda t,y: (np.sin(t))**2*y
phi_sp = lambda t,y: (sp.sin(t))**2*y
```

```
####################################################
# solution exacte
####################################################
t   = sp.Symbol('t')
y   = sp.Function('y')
edo = sp.Eq( sp.diff(y(t),t) , phi_sp(t,y(t)) )
solpar = sp.dsolve(edo, ics={y(t0): y0})
display(Markdown(f"**Solution exacte :** $${sp.latex(solpar)}$$"))

sol_exacte = sp.lambdify(t,solpar.rhs,'numpy')

####################################################
# schéma
####################################################
def multipas(phi, tt, sol_exacte):
    h  = tt[1] - tt[0]
    uu = np.empty_like(tt)
    # initialisation avec solution exacte pour ordre de convergence
    uu[0:3] = sol_exacte(tt[0:3])
    # recurrence
    for i in range(2,len(tt) - 1):
        uu[i+1] = a_0*uu[i] + a_1*uu[i-1] + a_2*uu[i-2] + h*(b_0*phi(tt[i],uu[i]) +
 ↪  b_1*phi(tt[i-1],uu[i-1]) + b_2*phi(tt[i-2],uu[i-2]))
    return uu

####################################################
# ordre
####################################################
H = []
err = []


plt.figure(figsize=(16,5))

ax1 = plt.subplot(1,2,1)
plt.xlabel('$t$')
plt.ylabel('$y$')
ax1.grid(True)

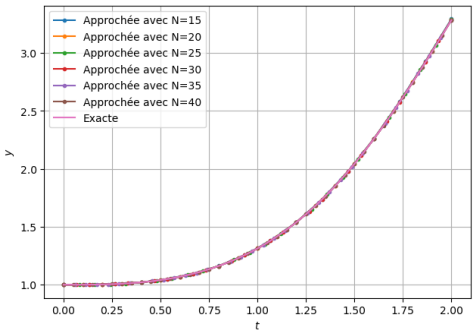
N = 10
for k in range(6):
    N += 5
    tt = np.linspace(t0, tfinal, N + 1)
    H.append( tt[1] - tt[0] )
    yy = sol_exacte(tt)
    uu = multipas(phi_np, tt, sol_exacte)
    err.append( np.linalg.norm(uu-yy,np.inf) )
    ax1.plot(tt,uu,'.-',label=f'Approchée avec N={N}')

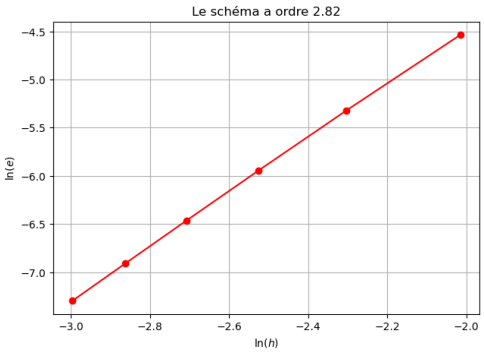
ax1.plot(tt,yy,label='Exacte') # sur la grille la plus fine
ax1.legend()

ax2 = plt.subplot(1,2,2)
ax2.plot( np.log(H), np.log(err), 'r-o')
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
plt.title(f'Le schéma a ordre {np.polyfit(np.log(H),np.log(err),1)[0]:1.2f}')
ax2.grid(True)
```

Solution exacte :

$$y(t) = e^{\frac{t}{2} - \frac{\sin(2t)}{4}}$$





25.3. EXERCICE 3 : Étude d'un schéma RK implicite à 2 étages

Considérons le schéma RK à $s = 2$ étages dont la matrice de Butcher est la suivante :

$$\begin{array}{c|cc} \alpha & 0 & \alpha \\ 1 & 1 & 0 \\ \hline & b_0 & b_1 \end{array}$$

Les coefficients b_0 et b_1 dépendent de la première lettre de votre prénom :

Lettre	b_0	b_1
$A - F$	$2/3$	$1/3$
$G - L$	$1/3$	$2/3$
$M - R$	$4/3$	$-1/3$
$S - Z$	$-1/3$	$4/3$

Attention : ils sont donnés comme des fractions sympy, il faudra les convertir en flottants pour les utiliser dans vos codes d'approximation numérique.

1. Déterminer α pour que le schéma soit d'ordre maximal.
2. Tester empiriquement l'ordre de convergence de ce schéma sur le problème de Cauchy $y'(t) = 4t(1-t^2) - y(t)$ et $y(0) = 1$ pour $t \in [0, 1]$.
3. Étudier la A-stabilité du schéma.

```
# ================================================
# On fixe $b_0$ et $b_1$
# ================================================
prenom = "Gloria"


if prenom[0].upper() >= 'A' and prenom[0].upper() <= 'F':
    b0 = sp.Rational(2, 3)
    b1 = sp.Rational(1, 3)
elif prenom[0].upper() >= 'G' and prenom[0].upper() <= 'L':
    b0 = sp.Rational(1, 3)
    b1 = sp.Rational(2, 3)
elif prenom[0].upper() >= 'M' and prenom[0].upper() <= 'R':
    b0 = sp.Rational(4, 3)
    b1 = sp.Rational(-1, 3)
elif prenom[0].upper() >= 'S' and prenom[0].upper() <= 'Z':
    b0 = sp.Rational(-1, 3)
    b1 = sp.Rational(4, 3)

display(Markdown(f"**{prenom}** : $b_0 = {b0}$, $b_1 = {b1}$"))
```

Gloria : $b_0 = 1/3$, $b_1 = 2/3$

C'est un schéma implicite. La barrière de Butcher affirme qu'un schéma RK à $s = 2$ étages **implicite** ne peut pas être d'ordre supérieur à $2s = 4$.

Calculons son ordre en fonction de α :

- il est consistant pour tout α (et tout couple (b_0, b_1) donné) car

$$\begin{cases} b_0 + b_1 = 1 \\ a_{0,0} + a_{0,1} = \alpha = c_0 \\ a_{1,0} + a_{1,1} = 1 = c_1 \end{cases}$$

- il est d'ordre 2 ssi $\alpha = 1 - \frac{1}{2b_0}$ car

$$b_0 c_0 + b_1 c_1 = \frac{1}{2} \quad \iff \quad b_0 \times \alpha + (1 - b_0) \times 1 = \frac{1}{2} \quad \iff \quad \alpha = 1 - \frac{1}{2b_0}$$

- il est d'ordre 3 ssi $b_0 = \frac{3}{4}$ car

$$b_0 c_0^2 + b_1 c_1^2 = \frac{1}{3} \quad \iff \quad b_0 \left(1 - \frac{1}{2b_0}\right)^2 + (1 - b_0) \times 1^2 = \frac{1}{3} \quad \iff \quad b_0 = \frac{3}{4}.$$

Conclusion : tous les schémas donnés sont d'ordre 2 si on pose α comme ci-dessous :

Lettre	b_0	b_1	α
$A - F$	$2/3$	$1/3$	$1/4$
$G - L$	$1/3$	$2/3$	$-1/2$
$M - R$	$4/3$	$-1/3$	$5/8$
$S - Z$	$-1/3$	$4/3$	$5/2$

```
prlat = lambda *args : display(Math(''.join( sp.latex(arg) for arg in args )))


def ordre_RK(s, A=None, b=None, c=None):

    j = sp.symbols('j')

    if A is None:
        A = sp.MatrixSymbol('a', s, s)
    else:
        A = sp.Matrix(A)
    if c is None:
        c = sp.symbols(f'c_0:{s}')
    if b is None:
        b = sp.symbols(f'b_0:{s}')


    display(Markdown("**Matrice de Butcher**"))
    But = matrice_Butcher(s, A, b, c)

    Eqs = []

    display(Markdown(f"**On a {s+1} conditions pour avoir consistance = pour être d'ordre 1**"))
    Eq_1 = ordre_1(s, A, b, c, j)
    Eqs.append(Eq_1)

    display(Markdown("**On doit ajouter 1 condition pour être d'ordre 2**"))
    Eq_2 = ordre_2(s, A, b, c, j)
    Eqs.append([Eq_2])

    display(Markdown("**On doit ajouter 2 conditions pour être d'ordre 3**"))
    Eq_3 = ordre_3(s, A, b, c, j)
    Eqs.append(Eq_3)
```

```
    # display(Markdown("**On doit ajouter 4 conditions pour être d'ordre 4**"))
    # Eq_4 = ordre_4(s, A, b, c, j)
    # Eqs.append(Eq_4)

    return Eqs


def matrice_Butcher(s, A, b, c):
    But = sp.Matrix(A)
    But = But.col_insert(0, sp.Matrix(c))
    last = [sp.Symbol(" ")]
    last.extend(b)
    But = But.row_insert(s,sp.Matrix(last).T)
    display(Math(sp.latex(sp.Matrix(But))))
    return But


def ordre_1(s, A, b, c, j):
    texte = r"\sum_{j=1}^s b_j =" + f"{sum(b).simplify()}"
    texte += r"\text{ doit être égale à }1"
    display(Math(texte))
    Eq_1 = [sp.Eq( sum(b).simplify(), 1 ) ]
    for i in range(s):
        somma = sp.summation(A[i,j],(j,0,s-1)).simplify()
        texte = r'\sum_{j=1}^s a_{'+str(i)+r'j}=' + sp.latex( somma )
        texte += r"\text{ doit être égale à }"+sp.latex(c[i])
        display(Math( texte ))
        Eq_1.append( sp.Eq( somma, c[i] ) )
    return Eq_1


def ordre_2(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j=' +sp.latex(sum([b[i]*c[i] for i in range(s)]).simplify())
    texte += r"\text{ doit être égale à }\frac{1}{2}"
    display(Math(texte))
    Eq_2 = sp.Eq( sum([b[i]*c[i] for i in range(s)]).simplify(), sp.Rational(1,2) )
    return Eq_2


def ordre_3(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j^2='
    texte += sp.latex( sum([b[i]*c[i]**2 for i in range(s)]).simplify() )
    texte += r"\text{ doit être égale à }\frac{1}{3}"
    display(Math(texte))
    Eq_3_1 = sp.Eq( sum([b[i]*c[i]**2 for i in range(s)]).simplify(), sp.Rational(1,3) )
    texte = r'\sum_{i,j=1}^s b_i a_{ij} c_j='
    somma = sum([b[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{6}"
    display(Math(texte))
    Eq_3_2 = sp.Eq( sum([b[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,6) )
    return [Eq_3_1, Eq_3_2]


def ordre_4(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j^3='
    texte += sp.latex( sum([b[i]*c[i]**3 for i in range(s)]).simplify() )
    texte += r"\text{ doit être égale à }\frac{1}{4}"
    display(Math(texte))
    Eq_4_1 = sp.Eq( sum([b[i]*c[i]**3 for i in range(s)]).simplify(), sp.Rational(1,4) )
    texte = r'\sum_{i,j=1}^s b_i c_i a_{ij} c_j='
    somma = sum([b[i]*c[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
```

```
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{8}"
    display(Math(texte))
    Eq_4_2 = sp.Eq( sum([b[i]*c[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,8) )
    texte = r'\sum_{i,j=1}^s b_i a_{ij} c_j^2='
    somma = sum([b[i]*A[i,j]*c[j]**2 for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{12}"
    display(Math(texte))
    Eq_4_3 = sp.Eq( sum([b[i]*A[i,j]*c[j]**2 for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,12) )
    texte = r'\sum_{i,j,k=1}^s b_i a_{ij} a_{jk} c_k='
    somma = sum([b[i]*A[i,j]*A[j,k]*c[k] for k in range(s) for j in range(s) for i in
↪   range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{24}"
    display(Math(texte))
    Eq_4_4 = sp.Eq( sum([b[i]*A[i,j]*A[j,k]*c[k] for k in range(s) for j in range(s) for i in
↪   range(s)]).simplify(), sp.Rational(1,24) )
    return [Eq_4_1, Eq_4_2, Eq_4_3, Eq_4_4]



##################################################
# Conditions avec s étages
##################################################

s = 2

# alpha = sp.symbols(r'\alpha')

b = [ b0, b1]
alpha= 1-1/(2*b0)
display(Math(f"\\alpha = {alpha}, b = {b}"))

c = [alpha,1]
A = [[0, alpha],
     [1,0]]
Eqs = ordre_RK(s, A, b, c)
```

$\alpha = -1/2, b = [1/3, 2/3]$

Matrice de Butcher

$$\begin{bmatrix} -\frac{1}{2} & 0 & -\frac{1}{2} \\ 1 & 1 & 0 \\ & \frac{1}{3} & \frac{2}{3} \end{bmatrix}$$

On a 3 conditions pour avoir consistance = pour être d'ordre 1

$\sum_{j=1}^{s} b_j = 1$ doit être égale à 1

$\sum_{j=1}^{s} a_{0j} = -\frac{1}{2}$ doit être égale à $-\frac{1}{2}$

$\sum_{j=1}^{s} a_{1j} = 1$ doit être égale à 1

On doit ajouter 1 condition pour être d'ordre 2

$\sum_{j=1}^{s} b_j c_j = \frac{1}{2}$ doit être égale à $\frac{1}{2}$

On doit ajouter 2 conditions pour être d'ordre 3

$\sum_{j=1}^{s} b_j c_j^2 = \frac{3}{4}$ doit être égale à $\frac{1}{3}$

$\sum_{i,j=1}^{s} b_i a_{ij} c_j = -\frac{1}{2}$ doit être égale à $\frac{1}{6}$

Le schéma correspond à la méthode suivante :

$$\begin{cases} u_0 = y_0 \\ k_0 = \varphi\Big(t_n + c_0 h, u_n + h(a_{00}k_0 + a_{01}k_1)\Big) \\ k_1 = \varphi\Big(t_n + c_1 h, u_n + h(a_{10}k_0 + a_{11}k_1)\Big) \\ u_{n+1} = u_n + h(b_0 k_0 + b_1 k_1) \end{cases}$$

soit encore, dans notre cas particulier,

$$\begin{cases} u_0 = y_0 \\ k_0 = \varphi\Big(t_n + \alpha h, u_n + \alpha h k_1\Big) \\ k_1 = \varphi\Big(t_{n+1}, u_n + h k_0\Big) \\ u_{n+1} = u_n + h(b_0 k_0 + b_1 k_1) \end{cases}$$

```
##################################################
# EDO
##################################################
t0     = 0
tfinal = 1
y0     = 1

phi = lambda t,y: 4*t*(1-t**2)-y

##################################################
# solution exacte
##################################################
t   = sp.Symbol('t')
y   = sp.Function('y')
edo = sp.Eq( (y(t)).diff() , phi(t,y(t)) )
solpar = sp.dsolve(edo , ics={y(t0):y0})
display(solpar)

sol_exacte = sp.lambdify(t,solpar.rhs,'numpy')

##################################################
# schéma
##################################################

# Paramètres du schéma recuperés depuis la matrice de Butcher obtenue précédemment
cc = np.array(c)
bb = np.array(b)
AA = np.array(A)


# Schéma de Runge-Kutta implicite à s étages
def RK(phi, tt, y0):
    h  = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    uu[0] = y0
    for n in range(len(tt) - 1):
        k_init = phi(tt[n],uu[n])
```

```
        KK = fsolve( lambda kk: [ -kk[i] +  phi( tt[n]+cc[i]*h , uu[n] + h*sum([kk[j]*AA[i,j] for j in
↪   range(s)]) ) for i in range(s) ], k_init*np.ones((s,1)) )
        uu[n+1] = uu[n] + h*sum([KK[j]*b[j] for j in range(s)])
    return uu


##################################################
# ordre
##################################################
H = []
err = []


plt.figure(figsize=(16,5))

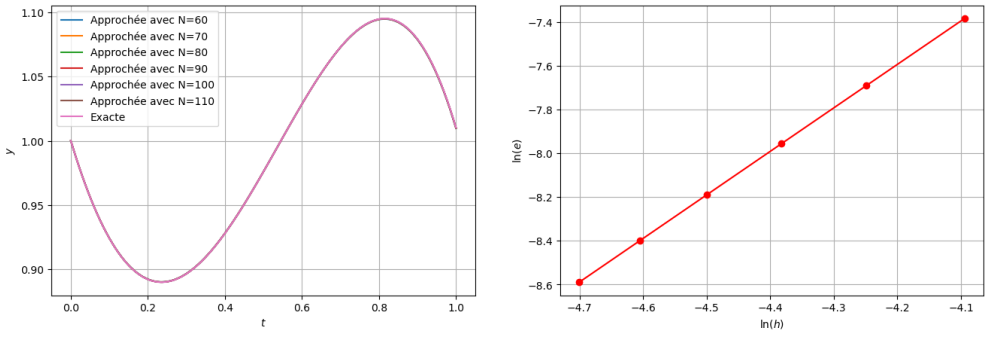
ax1 = plt.subplot(1,2,1)
plt.xlabel('$t$')
plt.ylabel('$y$')
ax1.grid(True)

N = 50
for k in range(6):
    N += 10
    tt = np.linspace(t0, tfinal, N + 1)
    H.append( tt[1] - tt[0] )
    yy = sol_exacte(tt)
    uu = RK(phi, tt, y0)
    err.append( np.linalg.norm(uu-yy,np.inf) )
    ax1.plot(tt,uu,label=f'Approchée avec N={N}')

ax1.plot(tt,yy,label='Exacte') # sur la grille la plus fine
ax1.legend()

ax2 = plt.subplot(1,2,2)
ax2.plot( np.log(H), np.log(err), 'r-o')
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
plt.title(f'Le schéma a ordre {np.polyfit(np.log(H),np.log(err),1)[0]:1.2f}')
ax2.grid(True)
```

$y(t) = -4t^3 + 12t^2 - 20t + 20 - 19e^{-t}$



On applique le schéma au problème de Cauchy $y'(t) = -\beta y(t)$ et $y(0) = 1$ avec $\beta > 0$ dont la solution est $y(t) = e^{-\beta t}$.

On obtient alors la relation de récurrence suivante :

```
u_n = sp.symbols(r'u_n')
u_np1 = sp.symbols(r'u_{n+1}')
```

```
dt = sp.symbols(r'h', positive=True)
x = sp.symbols(r'x', positive=True)

k_0 = sp.symbols(r'k_0')
k_1 = sp.symbols(r'k_1')
kk = [k_0, k_1]

beta = sp.symbols(r'\beta', positive=True)
phi = lambda y: -beta*y

Eqs = [ sp.Eq(kk[i] ,  phi(u_n + dt*sum([kk[j]*A[i][j] for j in range(s)]))) for i in range(s) ]
display(Eqs)
sol = sp.solve( Eqs, kk ) # c'est un dictionnaire
KK = [sol[kk[i]] for i in range(s)]
schema = sp.Eq( u_np1 , u_n + dt*sum([KK[j]*b[j] for j in range(s)]).factor() )
display(schema)
```

```
[Eq(k_0, -\beta*(-h*k_1/2 + u_n)), Eq(k_1, -\beta*(h*k_0 + u_n))]
```

$$u_{n+1} = \frac{\beta h u_n \left(\beta h - 2\right)}{\beta^2 h^2 + 2} + u_n$$

On peut expliciter u_n en fonction de u_0 et du produit βh (on notera x le produit βh) :

```
q = (schema.rhs/u_n).simplify().subs(beta*dt,x)
display(Math(sp.latex(sp.Eq(sp.Symbol('q(x)'),q.simplify()))))
```

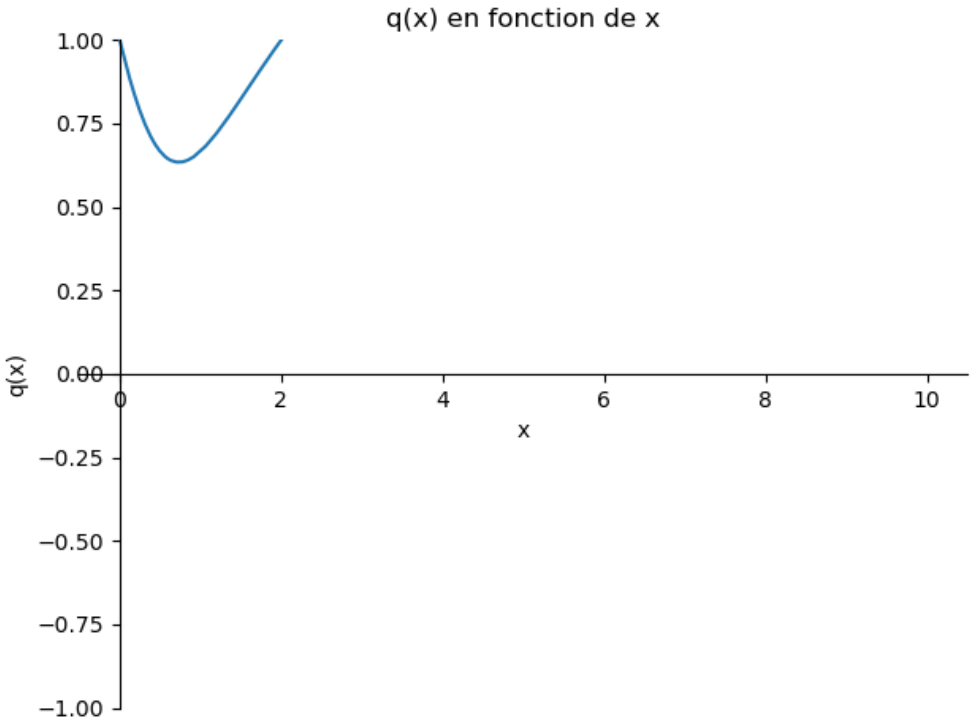
$$q(x) = \frac{2\left(x^2 - x + 1\right)}{x^2 + 2}$$

La suite vérifie $u_n \xrightarrow[n\to\infty]{} 0$ ssi $|q(x)| < 1$. En étudiant brièvement la fonction $q(x)$, on trouve que

Lettre	b_0	b_1	A-stabilité
$A-F$	$2/3$	$1/3$	$\forall h$
$G-L$	$1/3$	$2/3$	$\beta h < 2$
$M-R$	$4/3$	$-1/3$	$h < \frac{2(\sqrt{7}-1)}{3}$
$S-Z$	$-1/3$	$4/3$	$h < \frac{\sqrt{37}-1}{9}$

```
sp.plot(q, (x,0,10), ylim=(-1,1),
            ylabel='q(x)', xlabel='x', title='q(x) en fonction de x')


sp.solve(abs(q)<1, x)
```


q(x) en fonction de x
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
q(x)
0
2
4
6
8
10
x

$x < 2$

Chapitre 26.

2024 Contrôle Terminal session 2

2h30, deposer le notebook dans Moodle

```
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp
from scipy.optimize import fsolve
from scipy.integrate import solve_ivp
from IPython.display import display, Markdown, Math
```

26.1. EXERCICE 1 : Étude de l'amour entre Roméo et Juliette

Considérons une histoire d'amour entre Roméo et Juliette. Nous définissons $R(t)$ comme l'amour de Roméo pour Juliette à l'instant t et $J(t)$ comme l'amour de Juliette pour Roméo au même instant. Leur amour est modélisé par le système d'équations différentielles suivant :

$$\begin{cases} R'(t) = aR(t) + bJ(t) + f(t) \\ J'(t) = cR(t) + dJ(t) + g(t) \end{cases}$$

Les paramètres a, b, c et d sont des constantes réelles qui décrivent les styles romantiques de Roméo et Juliette. Plus précisément, a et b caractérisent le style romantique de Roméo, tandis que c et d décrivent celui de Juliette.
Nous introduisons également une influence externe ou une fonction de perturbation $f(t)$ pour Roméo et $g(t)$ pour Juliette, pour tenir compte des influences ou perturbations externes.
Nous allons maintenant examiner comment leur dynamique amoureuse évolue dans différentes situations, avec différents types de fonctions de perturbation $f(t)$ et $g(t)$. Pour tous les tests, nous prenons $a = -1$, $b = 2$, $c = -2$, $d = 1$ et la valeur initiale est $(0.1; 0.1)$. Nous considérons l'intervalle de temps d'un mois (c'est-à-dire 30 jours).

1. **Sans Perturbation Externe** : Nous commençons par exclure les termes de perturbation externes dans le système : $f(t) = g(t) = 0$ pour tout t. Nous examinerons le comportement périodique de leur amour.
2. **Surprise pour Juliette** : Supposons que Juliette reçoit une surprise de la part de Roméo, comme un cadeau inattendu. Cette situation peut être décrite par $g(t)$, une fonction de stimulation, qui n'a de valeur qu'à un jour donné :
$$g(t) = \begin{cases} 12 & \text{si } 10 < t < 11 \\ 0 & \text{sinon} \end{cases}$$
Nous examinerons le comportement de leur amour dans cette situation avec $f(t) = 0$.
3. **Nouvel Emploi pour Roméo** : Supposons que Roméo obtient un nouvel emploi satisfaisant et est confiant dans leur vie future. Ici, $f(t) = -0.03t$ et $g(t) = 0$ pour tout t.
4. **Perturbation Périodique Mineure** : Enfin, nous considérons une situation de perturbation périodique mineure, qui représente l'influence de toutes sortes de petits faits quotidiens. Ici, $f(t) = \sin(0.2\pi t)$ et $g(t) = 0$ pour tout t.

EXERCICE : utiliser le schéma de Heun pour approcher la solution du système d'équations différentielles. Pour chaque situation, tracer sur le même plan les graphiques de $R(t)$ et $J(t)$ sur l'intervalle $[0, 30]$. Tracer ensuite le portrait de phase de leur amour, c'est-à-dire le graphe de $J(t)$ en fonction de $R(t)$.

Les schémas

```
t0, R0, J0 = 0, 0.1, 0.1
tfin = 30 # on étudie l'évolution sur 30 jours

a, b, c, d = -1, 2, -2, 1
pphi = lambda t, yy, f, g : np.array([a*yy[0]+b*yy[1]+f(t), c*yy[0]+d*yy[1]+g(t)])

def HEUN(pphi, f, g, yy0, tt):
    RR_JJ = np.zeros((len(tt), len(yy0)))
    RR_JJ[0] = yy0
    h = tt[1]-tt[0]
    for i in range(len(tt)-1):
        k1 = pphi(tt[i], RR_JJ[i], f, g)
        k2 = pphi(tt[i+1], RR_JJ[i]+h*k1, f, g)
        RR_JJ[i+1] = RR_JJ[i] + h/2*(k1+k2)
    return RR_JJ

def EM(pphi, f, g, yy0, tt):
    RR_JJ = np.zeros((len(tt), len(yy0)))
    RR_JJ[0] = yy0
    h = tt[1]-tt[0]
    for i in range(len(tt)-1):
        k1 = pphi(tt[i], RR_JJ[i], f, g)
        RR_JJ[i+1] = RR_JJ[i] + h*pphi(tt[i]+h/2, RR_JJ[i]+h/2*k1, f, g)
    return RR_JJ

def deja(pphi, f, g, yy0, tt):
    RR_JJ = solve_ivp(pphi, (t0, tfin), yy0, t_eval=tt, args=(f, g),
                      #   method='BDF'
                      #   method='RK45'
                      #   method='RK23'
                      #   method='DOP853'
                      #   method='Radau'
                        method='LSODA'
                        )
    return RR_JJ.y.T
```

La fonction pour l'affichage

```
def affichage(heures=24):
    mask = range(heures)
    fig = plt.figure(figsize=(24, 8))
    for i, (RR, JJ, title) in enumerate(zip([RR_test1, RR_test2, RR_test3, RR_test4],
                                            [JJ_test1, JJ_test2, JJ_test3, JJ_test4],
                                            ["1. Sans Perturbation Externe", "2. Surprise pour Juliette",
 ↪  "3. Nouvel emploi pour Roméo", "4. Perturbation périodique mineure"])):
        RR_local, JJ_local, tt_local = RR[mask], JJ[mask], tt[mask]
        plt.subplot(2, 4, i+1)
        plt.title(title)
        plt.plot(tt_local, RR_local, label='R(t)')
        plt.plot(tt_local, JJ_local, label='J(t)')
        plt.xlim([t0, tfin])
        plt.xlabel('t')
        plt.legend()
        plt.subplot(2, 4, i+1+4)
        plt.plot(RR_local, JJ_local)
        plt.scatter(RR[0], JJ[0], c='r', label='t=0')
        plt.scatter(RR_local[-1], JJ_local[-1], c='g', label=f't={tt_local[-1] : .2f}', marker='x')
        plt.xlabel('R(t)')
        plt.ylabel('J(t)')# Ajouter les axes x=0 et y=0
        plt.axhline(0, color='black',linewidth=0.5)
        plt.axvline(0, color='black',linewidth=0.5)
```

```
        plt.xlim([min(RR), max(RR)])
        plt.ylim([min(JJ), max(JJ)])
        plt.legend()

    fig.tight_layout()
```

Les calculs

```
tt = np.linspace(t0, tfin, 24*tfin+1) # on veut une solution toutes les heures

schema = 'HEUN'
# schema = 'EM'
# schema = 'deja'


# 1. Sans Perturbation Externe
f = lambda t: 0
g = lambda t: 0
RR_test1, JJ_test1 = eval(schema)(pphi, f, g, [R0, J0], tt).T

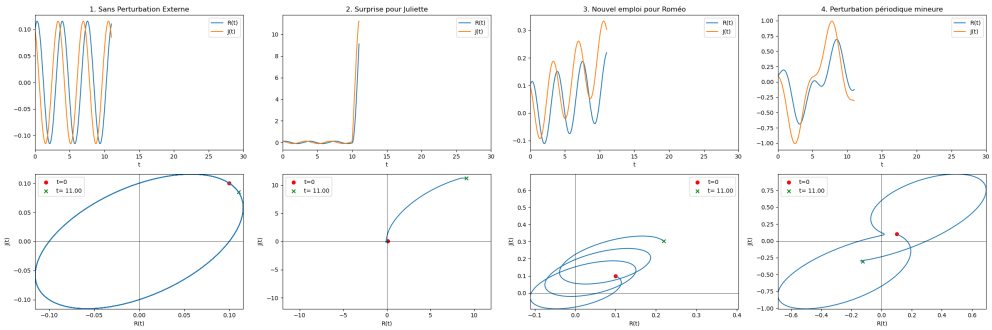
# 2. Surprise pour Juliette au jour 10
f = lambda t: 0
g = lambda t: 12 if 10<t<11 else 0
RR_test2, JJ_test2 = eval(schema)(pphi, f, g, [R0, J0], tt).T

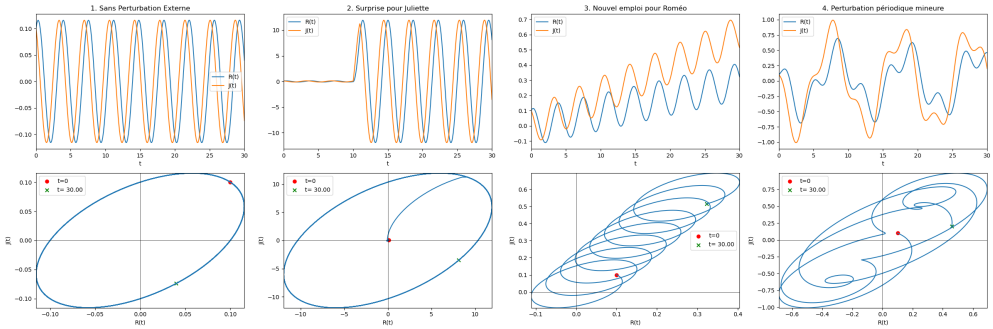
# 3. Nouvel emploi pour Roméo
f = lambda t: -0.03*t
g = lambda t: 0
RR_test3, JJ_test3 = eval(schema)(pphi, f, g, [R0, J0], tt).T

# 4. Perturbation riodique mineure
f = lambda t: np.sin(0.2*np.pi*t)
g = lambda t: 0
RR_test4, JJ_test4 = eval(schema)(pphi, f, g, [R0, J0], tt).T
```

Affichage statique

```
affichage(24*11+1) # 11 jours
affichage(24*tfin+1) # 30 jours
```





1. **Sans Perturbation Externe** : le plan de phase suggère que leur amour est périodique. Leurs sentiments l'un pour l'autre oscillent périodiquement.
2. **Surprise pour Juliette** : le plan de phase suggère que ce type de perturbation ou de stimulus romantique peut entraîner un amour mutuel.
3. **Nouvel Emploi pour Roméo** : le plan de phase suggère que leur relation se développe vers un amour mutuel.
4. **Perturbation Périodique Mineure** : il est difficile de prévoir le développement de leur relation.

Affichage dynamique (si `interact` installé)

```
from ipywidgets import interact
interact(affichage, heures=(24,24*tfin,24)); # on increment par 24h
```

```
interactive(children=(IntSlider(value=24, description='heures', max=720, min=24, step=24), Output()),
```

26.2. EXERCICE 2 : Schéma MPL explicite avec paramètres

1. Pour quelles valeurs de η et ϑ le schéma multipas linéaire suivant est convergent ?

$$u_{n+1} = (3\vartheta - 2\eta)u_n + \left(2 - \frac{5}{2}\eta\right)u_{n-1} + h\eta(\varphi_n + \varphi_{n-1})$$

2. Sans effectuer de calculs, quel est l'ordre maximal théorique de ce schéma ?
   Calculer l'ordre de convergence de ce schéma.
3. Tester empiriquement l'ordre de convergence du schéma ainsi obtenu sur le problème de Cauchy $y'(t) = \sin(2t) - y(t)\sin(t)$ et $y(\pi/2) = 1$ pour $t \in [\pi/2, 2\pi]$.

Remarques :

- les deux premiers points peuvent être résolus "à la main" ou avec `sympy` (dans ma correction, je vous montrerai comment faire avec `sympy`).
- dans le dernier point, attention à bien indiquer à chaque étape si les fonctions que vous utlisez appartiennent à `numpy` ou à `sympy`.

C'est un schéma à $q = 2$ pas, c'est-à-dire de la forme

$$u_{n+1} = a_0 u_n + a_1 u_{n-1} + h(b_{-1}\varphi_{n+1} + b_0\varphi_n + b_1\varphi_{n-1})$$

avec $a_0 = 3\vartheta - 2\eta$, $a_1 = 2 - \frac{5}{2}\eta$, $b_{-1} = 0$ et $b_0 = b_1 = \eta$. Il est donc explicite.

Pour qu'il soit convergent, il faut que le schéma soit zéro-stable et consistant.

Écrivons donc qui sont $\xi(0)$, $\xi(1)$, $\xi(2)$, $\xi(3)$ puis le premier polynôme caractéristique ϱ et calculons ses racines dans le cas général d'un schéma à $q = 2$ pas :

```
# ========================
q = 2
# ========================


a_0, a_1 = sp.symbols('a_0 a_1 ', real=True)
b_0, b_1, b_m1 = sp.symbols('b_0 b_1 b_{-1}', real=True)
r = sp.Symbol('r')

# ========================
def xi(i, q):
    sa = sum((-j)**i * aa[j] for j in range(q))
    sb = b_m1 + sum((-j)**(i-1) * bb[j] for j in range(1, q))
    if i == 1:
        sb += bb[0]
    return (sa).factor() + (i * sb).factor()

# ========================
aa = [a_0, a_1]
bb = [b_0, b_1]

display(Markdown(f"Pour une méthode à q={q} pas"))

display(Markdown(f"- on écrit le premier polynôme caractéristique (pour étudier la zéro-stabilité)"))
rho = sp.poly(r**(q) - sum( aa[j] * r**(q-1-j) for j in range(q)), r)
texte =  r"$\varrho(r)=" + sp.latex(rho.as_expr(),order='old') +  r"$"
display(Math(texte))
display(Markdown(f"dont les racines sont :"))
racines = sp.roots(rho,r)
for k,v in racines.items():
    display(k)

display(Markdown(f"- on affiche $\\xi(i)$ (pour étudier la consistance et l'ordre de consistance)"))
for i in range(4):
    display(Markdown(f"$\\xi({i}) = {sp.latex(xi(i, q))}$"))
```

Pour une méthode à q=2 pas

- on écrit le premier polynôme caractéristique (pour étudier la zéro-stabilité)

$\varrho(r) = -a_1 - r a_0 + r^2$

dont les racines sont :

$\frac{a_0}{2} - \frac{\sqrt{a_0^2 + 4a_1}}{2}$

$\frac{a_0}{2} + \frac{\sqrt{a_0^2 + 4a_1}}{2}$

- on affiche $\xi(i)$ (pour étudier la consistance et l'ordre de consistance)

$\xi(0) = a_0 + a_1$

$\xi(1) = -a_1 + b_0 + b_1 + b_{-1}$

$\xi(2) = a_1 - 2\,(b_1 - b_{-1})$

$\xi(3) = -a_1 + 3\,(b_1 + b_{-1})$

Étude de la consistance

Pour que le schéma soit consistant il faut que $\xi(0) = \xi(1) = 1$:

$$\begin{cases} a_0 + a_1 = 1 \\ -a_1 + b_0 + b_1 + b_{-1} = 1 \end{cases}$$

Dans notre cas, cela donne

```
eta, theta = sp.symbols(r'\eta \vartheta', real=True)

aa = [3*theta-2*eta, 2-5*eta/2]
bb = [eta, eta]
b_m1 = 0

systeme = [sp.Eq(xi(i, q),1) for i in range(2)]
display(Markdown(r"$\begin{cases}"+' '.join([  r"\displaystyle"+sp.latex(s)+r"\\" for s in systeme]) +
 ↪  r"\end{cases}$"))
```

$$\begin{cases} -\frac{9\eta - 6\vartheta - 4}{2} = 1 \\ 2\eta + \frac{5\eta - 4}{2} = 1 \end{cases}$$

dont la solution est

```
sol = sp.solve(systeme)
for cle, valeur in sol.items():
    display(Math(sp.latex(cle) + "=" + sp.latex(valeur)))
```

$$\eta = \frac{2}{3}$$

$$\vartheta = \frac{2}{3}$$

ainsi

```
for i in range(2):
    display(Markdown(f"$a_{i} = {sp.latex(aa[i])} = {sp.latex(aa[i].subs(sol))}$"))
    display(Markdown(f"$b_{i} = {sp.latex(bb[i])} = {sp.latex(bb[i].subs(sol))}$"))
display(Markdown(f"$b_{{-1}} = {sp.latex(b_m1)} $"))
```

$a_0 = -2\eta + 3\vartheta = \frac{2}{3}$

$b_0 = \eta = \frac{2}{3}$

$a_1 = 2 - \frac{5\eta}{2} = \frac{1}{3}$

$b_1 = \eta = \frac{2}{3}$

$b_{-1} = 0 $

Étude de la zéro-stabilité

Pour que le schéma soit zéro-stable il faut que le premier polynôme caractéristique $\varrho(r) = r^2 - a_0 r - a_1$ ait ses racines de module inférieur ou égal à 1. Si une racine est de module 1, elle doit être simple.

Le premier polynôme caractéristique est $r^2 + 2r - 3$. Ses racines sont $r_1 = -1/3$ et $r_2 = 1$. Le schéma est donc zéro-stable.

```
rho = sp.poly(r**(q) - sum( aa[j].subs(sol) * r**(q-1-j) for j in range(q)), r)
texte =  r"$\varrho(r)=" + sp.latex(rho.as_expr(),order='old') + r"$"
display(Math(texte))
racines = sp.roots(rho.subs(sol),r)
display(racines)
```

$$\varrho(r) = -\frac{1}{3} - \frac{2r}{3} + r^2$$

```
{1: 1, -1/3: 1}
```

D'après la première barrière de Dahlquist, un schéma à $q = 2$ pas explicite est au mieux d'ordre $q = 2$. On doit donc juste vérifier si $\xi(2) = 1$.

```
xi2 = xi(2, q)
texte = r"\xi(2) ="
texte += sp.latex(xi2)
texte += "=" + sp.latex(xi2.subs(sol))

display(Math(texte))
```

$$\xi(2) = -2\eta - \frac{5\eta - 4}{2} = -1$$

On conclut que le schéma n'est que d'ordre 1.

Dans cette partie, attention à bien indiquer à chaque étape si les fonctions utlisées appartiennent à `numpy` ou à `sympy`.

```
####################################################
# EDO sympy
####################################################
y0     = 1
t0_sp = sp.pi/2
phi_sp = lambda t,z: -sp.sin(t)*z+sp.sin(2*t)

####################################################
# solution exacte sympy
####################################################
t, z   = sp.symbols('t z')
y   = sp.Function('y')
edo = sp.Eq( sp.diff(y(t),t) , phi_sp(t,y(t)) )
solpar = sp.dsolve(edo, ics={y(t0_sp): y0})
display(solpar)

####################################################
# sympy -> numpy
####################################################

sol_exacte = sp.lambdify(t,solpar.rhs,'numpy')

t0, tfinal = float(t0_sp), 2*np.pi
y0 = float(y0)

phi = sp.lambdify((t,z),phi_sp(t,z),'numpy')

####################################################
# schéma
####################################################
def multipas(phi, tt, sol_exacte):
    h  = tt[1] - tt[0]

    uu = np.zeros_like(tt)

    q = 2
    a0, a1,  = 2/3, 1/3
    b0, b1 = 2/3, 2/3

    # initialisation avec solution exacte pour ordre de convergence
    uu[0:q+1] = sol_exacte(tt[0:q+1])
    # recurrence
    for i in range(q, len(tt) - 1):
        uu[i+1] = a0*uu[i] + a1*uu[i-1] + h*(b0*phi(tt[i],uu[i]) + b1*phi(tt[i-1],uu[i-1]) )
    return uu
```

```
##################################################
# ordre
##################################################
H = []
err = []


plt.figure(figsize=(16,5))

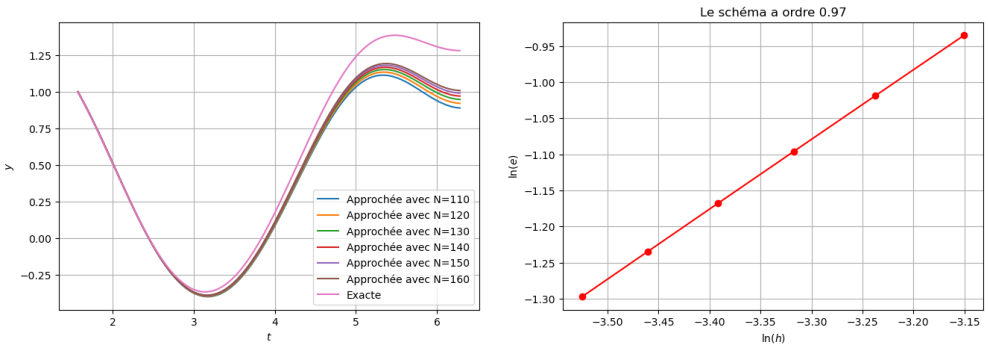
ax1 = plt.subplot(1,2,1)
plt.xlabel('$t$')
plt.ylabel('$y$')
ax1.grid(True)

N = 100
for k in range(6):
    N += 10
    tt = np.linspace(t0, tfinal, N + 1)
    H.append( tt[1] - tt[0] )
    yy = sol_exacte(tt)
    uu = multipas(phi, tt, sol_exacte)
    err.append( np.linalg.norm(uu-yy,np.inf) )
    ax1.plot(tt,uu,label=f'Approchée avec N={N}')

ax1.plot(tt,yy,label='Exacte') # sur la grille la plus fine
ax1.legend()

ax2 = plt.subplot(1,2,2)
ax2.plot( np.log(H), np.log(err), 'r-o')
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
plt.title(f'Le schéma a ordre {np.polyfit(np.log(H),np.log(err),1)[0]:1.2f}')
ax2.grid(True)
```

$y(t) = -e^{\cos(t)} + 2\cos(t) + 2$



26.3. EXERCICE 3 : Schéma RK à 3 étages explicite

La matrice de Butcher d'un schéma RK à $s = 3$ étages quelconque s'écrit :

c_0	a_{00}	a_{01}	a_{02}
c_1	a_{10}	a_{11}	a_{12}
c_2	a_{20}	a_{21}	a_{22}
	b_0	b_1	b_2

1. Écrire la matrice de Butcher correspondante au schéma suivant :

$$\begin{cases} u_0 = y_0 \\ k_0 = \varphi\Big(t_n, u_n\Big) \\ k_1 = \varphi\Big(t_n + \frac{h}{3}, u_n + \frac{h}{3}k_0\Big) \\ k_2 = \varphi\Big(t_n + \frac{2h}{3}, u_n + \frac{2h}{3}k_1\Big) \\ u_{n+1} = u_n + h\left(b_0 k_0 + (1-b_0)k_2\right) \end{cases}$$

2. Déterminer pour quelle valeur de b_0 l'ordre de convergence de ce schéma est maximal.
3. Tester empiriquement l'ordre de convergence de ce schéma sur le problème de Cauchy $y'(t) = 4t(t^2 - 1) + y(t)$ et $y(0) = 1$ pour $t \in [0, 1]$.
4. Déterminer sous quelle condition sur h ce schéma est A-stable.

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & 0 \\ \frac{2}{3} & 0 & \frac{2}{3} & 0 \\ \hline & b_0 & 0 & (1-b_0) \end{array}$$

Il s'agit d'un schéma explicite, car la matrice $\mathbb{A}$ est triangulaire inférieure stricte.

La barrière de Dalquist affirme qu'un schéma RK à $s = 3$ étages **explicite** ne peut pas être d'ordre supérieur à $s = 3$.

On peut bien-sûr écrire les formules à la main, mais on peut aussi utiliser un solveur pour résoudre le système d'équations non-linéaires comme suit :

```
prlat = lambda *args : display(Math(''.join( sp.latex(arg) for arg in args )))


def ordre_RK(s, A=None, b=None, c=None):

    j = sp.symbols('j')

    if A is None:
        A = sp.MatrixSymbol('a', s, s)
    else:
        A = sp.Matrix(A)
    if c is None:
        c = sp.symbols(f'c_0:{s}')
    if b is None:
        b = sp.symbols(f'b_0:{s}')


    display(Markdown("**Matrice de Butcher**"))
    But = matrice_Butcher(s, A, b, c)

    Eqs = []

    display(Markdown(f"**On a {s+1} conditions pour avoir consistance = pour être d'ordre 1**"))
    Eq_1 = ordre_1(s, A, b, c, j)
    Eqs.append(Eq_1)

    display(Markdown("**On doit ajouter 1 condition pour être d'ordre 2**"))
    Eq_2 = ordre_2(s, A, b, c, j)
    Eqs.append([Eq_2])
```

```
    display(Markdown("**On doit ajouter 2 conditions pour être d'ordre 3**"))
    Eq_3 = ordre_3(s, A, b, c, j)
    Eqs.append(Eq_3)

    # display(Markdown("**On doit ajouter 4 conditions pour être d'ordre 4**"))
    # Eq_4 = ordre_4(s, A, b, c, j)
    # Eqs.append(Eq_4)

    return Eqs


def matrice_Butcher(s, A, b, c):
    But = sp.Matrix(A)
    But = But.col_insert(0, sp.Matrix(c))
    last = [sp.Symbol(" ")]
    last.extend(b)
    But = But.row_insert(s,sp.Matrix(last).T)
    display(Math(sp.latex(sp.Matrix(But))))
    return But


def ordre_1(s, A, b, c, j):
    texte = r"\sum_{j=1}^s b_j =" + f"{sum(b).simplify()}"
    texte += r"\text{ doit être égale à }1"
    display(Math(texte))
    Eq_1 = [sp.Eq( sum(b).simplify(), 1 )]
    for i in range(s):
        somma = sp.summation(A[i,j],(j,0,s-1)).simplify()
        texte = r'\sum_{j=1}^s a_{'+str(i)+r'j}=' + sp.latex( somma )
        texte += r"\text{ doit être égale à }"+sp.latex(c[i])
        display(Math( texte ))
        Eq_1.append(sp.Eq( somma, c[i] ))
    return Eq_1


def ordre_2(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j=' +sp.latex(sum([b[i]*c[i] for i in range(s)]).simplify())
    texte += r"\text{ doit être égale à }\frac{1}{2}"
    display(Math(texte))
    Eq_2 = sp.Eq( sum([b[i]*c[i] for i in range(s)]).simplify(), sp.Rational(1,2) )
    return Eq_2


def ordre_3(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j^2='
    texte += sp.latex( sum([b[i]*c[i]**2 for i in range(s)]).simplify() )
    texte += r"\text{ doit être égale à }\frac{1}{3}"
    display(Math(texte))
    Eq_3_1 = sp.Eq( sum([b[i]*c[i]**2 for i in range(s)]).simplify(), sp.Rational(1,3) )
    texte = r'\sum_{i,j=1}^s b_i a_{ij} c_j='
    somma = sum([b[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{6}"
    display(Math(texte))
    Eq_3_2 = sp.Eq( sum([b[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,6) )
    return [Eq_3_1, Eq_3_2]


def ordre_4(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j^3='
    texte += sp.latex( sum([b[i]*c[i]**3 for i in range(s)]).simplify() )
    texte += r"\text{ doit être égale à }\frac{1}{4}"
```

```
    display(Math(texte))
    Eq_4_1 = sp.Eq( sum([b[i]*c[i]**3 for i in range(s)]).simplify(), sp.Rational(1,4) )
    texte = r'\sum_{i,j=1}^s b_i c_i a_{ij} c_j='
    somma = sum([b[i]*c[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{8}"
    display(Math(texte))
    Eq_4_2 = sp.Eq( sum([b[i]*c[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,8) )
    texte = r'\sum_{i,j=1}^s b_i a_{ij} c_j^2='
    somma = sum([b[i]*A[i,j]*c[j]**2 for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{12}"
    display(Math(texte))
    Eq_4_3 = sp.Eq( sum([b[i]*A[i,j]*c[j]**2 for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,12) )
    texte = r'\sum_{i,j,k=1}^s b_i a_{ij} a_{jk} c_k='
    somma = sum([b[i]*A[i,j]*A[j,k]*c[k] for k in range(s) for j in range(s) for i in
↪   range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{24}"
    display(Math(texte))
    Eq_4_4 = sp.Eq( sum([b[i]*A[i,j]*A[j,k]*c[k] for k in range(s) for j in range(s) for i in
↪   range(s)]).simplify(), sp.Rational(1,24) )
    return [Eq_4_1, Eq_4_2, Eq_4_3, Eq_4_4]



##################################################
# Conditions avec s étages
##################################################

s = 3

# paramètres dans le cas générale
b_0, b_1, b_2 = sp.symbols('b_0 b_1 b_2', real=True)
c_0, c_1, c_2 = sp.symbols('c_0 c_1 c_2', real=True)
a_00, a_01, a_02, a_10, a_11, a_12, a_20, a_21, a_22 = sp.symbols('a_00 a_01 a_02 a_10 a_11 a_12 a_20
↪  a_21 a_22', real=True)

# matrice dans le cas général
b = [b_0, b_1, b_2]
c = [c_0, c_1, c_2]
A = [[a_00, a_01, a_02], [a_10, a_11, a_12], [a_20, a_21, a_22]]

# matrice dans notre cas particulier
b = [ b_0, 0, 1-b_0]
c = [ 0, sp.Rational(1,3), sp.Rational(2,3) ]
A = [[ 0, 0, 0], [ sp.Rational(1,3), 0, 0], [ 0, sp.Rational(2,3), 0]]

Eqs = ordre_RK(s, A, b, c)
```

Matrice de Butcher

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & 0 \\ \frac{2}{3} & 0 & \frac{2}{3} & 0 \\ & b_0 & 0 & 1 - b_0 \end{bmatrix}$$

On a 4 conditions pour avoir consistance = pour être d'ordre 1

$\sum_{j=1}^{s} b_j = 1$ doit être égale à 1

$\sum_{j=1}^{s} a_{0j} = 0$ doit être égale à 0

$\sum_{j=1}^{s} a_{1j} = \frac{1}{3}$ doit être égale à $\frac{1}{3}$

$\sum_{j=1}^{s} a_{2j} = \frac{2}{3}$ doit être égale à $\frac{2}{3}$

On doit ajouter 1 condition pour être d'ordre 2

$\sum_{j=1}^{s} b_j c_j = \frac{2}{3} - \frac{2b_0}{3}$ doit être égale à $\frac{1}{2}$

On doit ajouter 2 conditions pour être d'ordre 3

$\sum_{j=1}^{s} b_j c_j^2 = \frac{4}{9} - \frac{4b_0}{9}$ doit être égale à $\frac{1}{3}$

$\sum_{i,j=1}^{s} b_i a_{ij} c_j = \frac{2}{9} - \frac{2b_0}{9}$ doit être égale à $\frac{1}{6}$

En conclusion, le schéma est d'ordre 1 par construction. De plis, le paramètre b_0 doit vérifier les conditions suivantes pour que le schéma soit d'ordre 2 et éventuellement 3 :

```
for i,eq in enumerate(Eqs[1:]):
    display(Markdown(f"**Pour être d'ordre {i+2} il faut vérifier**"))
    display(Math(sp.latex((eq))))
```

Pour être d'ordre 2 il faut vérifier

$$\left[\frac{2}{3} - \frac{2b_0}{3} = \frac{1}{2}\right]$$

Pour être d'ordre 3 il faut vérifier

$$\left[\frac{4}{9} - \frac{4b_0}{9} = \frac{1}{3},\ \frac{2}{9} - \frac{2b_0}{9} = \frac{1}{6}\right]$$

La première condition qui garantie l'ordre 2 impose le choix de b_0 :

```
sol = sp.solve(Eqs[1])
sol
```

```
{b_0: 1/4}
```

Vérifions si cette valeur satisfait les deux conditions pour être d'ordre 3 :

```
Eqs[2][0].subs(sol), Eqs[2][1].subs(sol)
```

```
(True, True)
```

On conclut que le schéma avec $b_0 = \frac{1}{4}$ est d'ordre 3.

```
# Le vecteur b est donc
b = [ b_0.subs(sol), 0, (1-b_0).subs(sol),]
```

```
#####################################################
# EDO
#####################################################
t0     = 0
tfinal = 1
y0     = 1

phi = lambda t,y: 4*t*(t**2-1)+y

#####################################################
# solution exacte
#####################################################
t   = sp.Symbol('t')
y   = sp.Function('y')
edo = sp.Eq( (y(t)).diff() , phi(t,y(t)) )
solpar = sp.dsolve(edo , ics={y(t0):y0})
display(solpar)

sol_exacte = sp.lambdify(t,solpar.rhs,'numpy')

#####################################################
# schéma
#####################################################

# Paramètres du schéma recuperés depuis la matrice de Butcher obtenue précédemment
cc = np.array(c)
bb = np.array(b)
AA = np.array(A)

# Schéma de Runge-Kutta explicite à 4 étages
def RK(phi, tt, y0):
    h  = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    # initialisation
    uu[0] = y0
    # recurrence
    for i in range(len(tt) - 1):
        k0 = phi(tt[i]          , uu[i] )
        k1 = phi(tt[i] + cc[1]*h, uu[i] + h* AA[1,0]*k0)
        k2 = phi(tt[i] + cc[2]*h, uu[i] + h*(AA[2,0]*k0 + AA[2,1]*k1))
        uu[i+1] = uu[i] + h*(bb[0]*k0 + bb[1]*k1 + bb[2]*k2)
    return uu


#####################################################
# ordre
#####################################################
H = []
err = []


plt.figure(figsize=(16,5))

ax1 = plt.subplot(1,2,1)
plt.xlabel('$t$')
plt.ylabel('$y$')
ax1.grid(True)

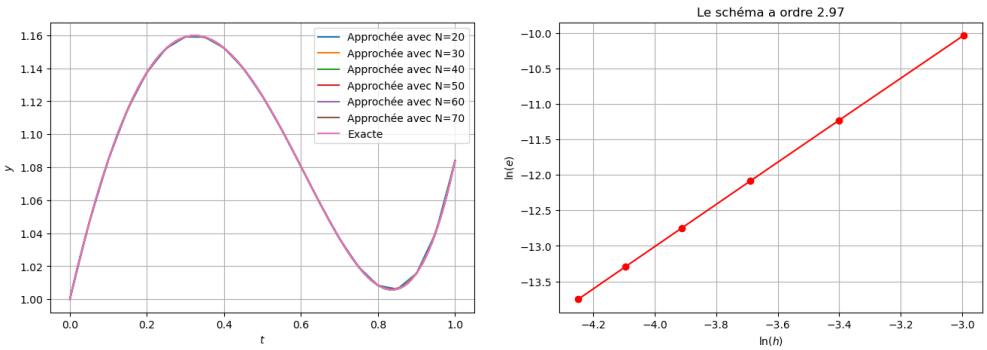
N = 10
for k in range(6):
    N += 10
    tt = np.linspace(t0, tfinal, N + 1)
    H.append( tt[1] - tt[0] )
```

```
    yy = sol_exacte(tt)
    uu = RK(phi, tt, y0)
    err.append( np.linalg.norm(uu-yy,np.inf) )
    ax1.plot(tt,uu,label=f'Approchée avec N={N}')

ax1.plot(tt,yy,label='Exacte') # sur la grille la plus fine
ax1.legend()

ax2 = plt.subplot(1,2,2)
ax2.plot( np.log(H), np.log(err), 'r-o')
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
plt.title(f'Le schéma a ordre {np.polyfit(np.log(H),np.log(err),1)[0]:1.2f}')
ax2.grid(True)
```

$y(t) = -4t^3 - 12t^2 - 20t + 21e^t - 20$



On applique le schéma au problème de Cauchy $y'(t) = -\beta y(t)$ et $y(0) = 1$ avec $\beta > 0$ dont la solution est $y(t) = e^{-\beta t}$.

On obtient alors la relation de récurrence suivante :

```
u_n = sp.symbols(r'u_n')
u_np1 = sp.symbols(r'u_{n+1}')
beta = sp.symbols(r'\beta', positive=True)
dt = sp.symbols(r'h', positive=True)
x = sp.symbols(r'x', positive=True)

k0 = -beta*u_n
k1 = -beta*(u_n + dt* AA[1,0]*k0)
k2 = -beta*(u_n + dt*(AA[2,0]*k0 + AA[2,1]*k1))
u_np1 = u_n + dt*(bb[0]*k0 + bb[1]*k1 + bb[2]*k2)

display(Math(sp.latex(sp.Eq(sp.Symbol('u_{n+1}'),u_np1.simplify()))))
```

$$u_{n+1} = \frac{u_n \left(-\beta h \left(\beta h \left(\beta h - 3\right) + 6\right) + 6\right)}{6}$$

On peut expliciter u_n en fonction de u_0 et du produit βh (on notera x le produit βh) :

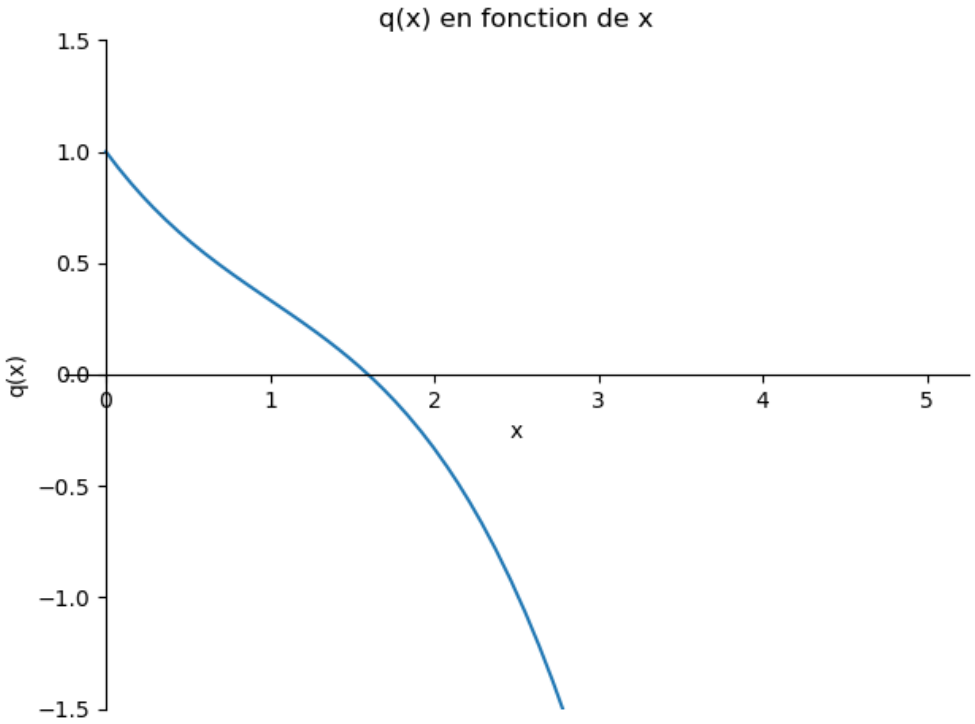
```
q = (u_np1/u_n).simplify().subs(beta*dt,x)
display(Math(sp.latex(sp.Eq(sp.Symbol('q(x)'),q.expand().simplify()))))
```

$$q(x) = -\frac{x^3}{6} + \frac{x^2}{2} - x + 1$$

La suite vérifie $u_n \xrightarrow[n\to\infty]{} 0$ ssi $|q(x)| < 1$. En étudiant brièvement la fonction $q(x)$, on trouve que $q(x) < 1$ ssi $x < x_0$ donné ci-dessous:.

```
sp.plot(q, (x,0,5), ylim=(-1.5,1.5),
            ylabel='q(x)', xlabel='x', title='q(x) en fonction de x')

sol = sp.solveset(q+1, x, domain=sp.S.Reals)
display(Markdown(f"$\\beta h<{sol.args[0].evalf()}$"))
```



$\beta h < 2.51274532661833$

Chapitre 27.

2023 Contrôle Continu

2h30, deposer le notebook dans Moodle

```
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp
from scipy.optimize import fsolve
from scipy.integrate import solve_ivp, odeint
from IPython.display import display, Markdown, Math
```

27.1. EXERCICE 1 : choix d'un schéma

Soit le problème de Cauchy

$$\begin{cases} y'(t) = \left(1 - y(t) - \dfrac{1}{4+t}\right) \sin(2\pi y(t)), \quad t \in [0;10] \\ y(0) = y_0 \ \in]0.7;0.9[\end{cases}$$

On ne peut pas calculer analytiquement la solution exacte.
Pour calculer des solutions approchées on doit utiliser un schéma numérique. Quelle caractéristique doit-il avoir ? Justifier la réponse.

Il s'agit d'un des problèmes de Cauchy proposés lors du CT 2023 de l'ECUE M61 *Équations différentielles*.

Bien qu'on ne puisse pas calculer analytiquement la solution exacte, nous pouvons tracer les lignes de courant pour avoir une idée du comportement de la solution en fonction de la donnée initiale. On en déduit que **le problème de Cauchy est numériquement mal posé**. Il faut alors utiliser un schéma d'ordre elevé.

Pour s'en convaincre nous pouvons aussi demander à `odeint` du module `scipy` de tracer la solution approchée pour différentes valeurs de y_0.

```
# variables globales
t0     = 0
tfinal = 10

phi = lambda t,y: np.sin(np.pi*2*y)*(1-1/(4+t)-y)

N = 200
tt = np.linspace(t0, tfinal, N + 1)

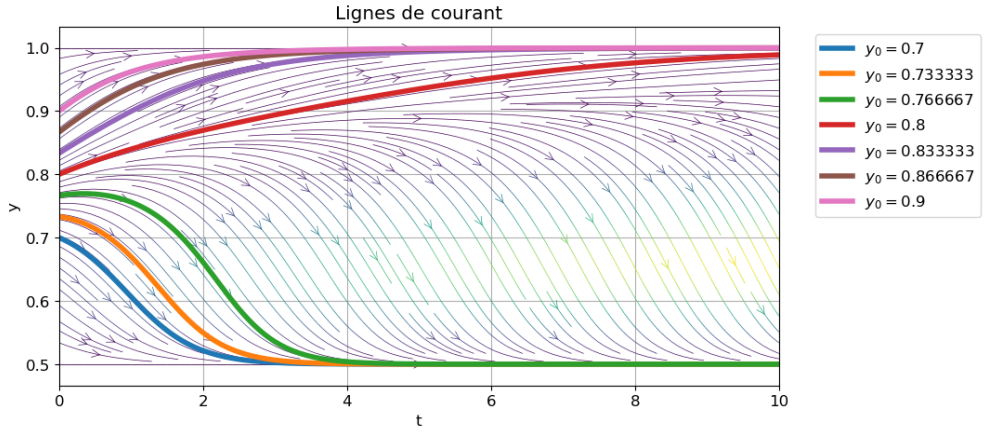
plt.figure(figsize=(10,5))

# On trace d'abord les lignes de courant
g1  = np.linspace(0,10,21)
g2  = np.linspace(0.5,1,21)
T,Y = np.meshgrid(g1,g2)
DT, DY = 1, phi(T,Y)          # compute growth rate on the grid
M = np.hypot(DT,DY) # = sqrt(DT**2+DY**2)         # norm growth rate
plt.streamplot(T,Y, DT/M, DY/M, color=M, linewidth=0.5, cmap=plt.cm.viridis, density=2, arrowstyle='->',
 ↪  arrowsize=1.5)
```

```
plt.grid()
plt.xlabel('t')
plt.ylabel('y')
plt.title('Lignes de courant');

# On trace quelque solution approchée
for y0 in np.linspace(0.7,0.9,7):
    uu = odeint(phi,y0,tt, tfirst=True)
    plt.plot(tt,uu,lw=4,label=f'$y_0={y0:g}$')

plt.legend(bbox_to_anchor=(1.04, 1), loc='upper left');
```



27.2. EXERCICE 2 : étude d'un schéma multipas

On notera $\varphi_k \overset{\text{déf}}{=} \varphi(t_k, u_k)$. Considérons la méthode multipas

$$u_{n+1} = u_{n-1} + h\left(\varphi_{n+1} + \varphi_{n-1}\right)$$

1. Étudier la zéro-stabilité de cette méthode.
2. La méthode est elle consistente ? Est elle convergente ? Si oui, étudier théoriquement l'ordre de convergence.
3. Vérifier empiriquement l'ordre de convergence sur le problème de Cauchy

$$\begin{cases} y'(t) = (y(t)+2)\dfrac{2t^2+2t-1}{1+t}, & \forall t \in I = [0,2], \\ y(0) = -1, \end{cases}$$

On remarque qu'il s'agit de la méthode de Crank-Nicolson sur l'intervalle $[t_{n-1}, t_{n+1}]$.

Q2 [**1 points**] Étudier la zéro-stabilité.

Le schéma est de la forme

$$u_{n+1} = a_0 u_n + a_1 u_{n-1} + h b_0 \varphi_n + h b_1 \varphi_{n-1} + h b_{-1} \varphi_{n+1}.$$

On a

- $p = 1$, i.e. c'est un méthode à $q = p + 1 = 2$ pas ;
- $a_0 = 0$ et $a_1 = 1$,
- $b_0 = 0$ et $b_1 = 1$,
- $b_{-1} = 1 \neq 0$ ($\implies$ la méthode est implicite).

Le premier polynôme caractéristique est

$$\varrho(r) = r^2 - a_0 r - a_1 = r^2 - 1 = (r-1)(r+1).$$

Les racines sont

$$r_0 = 1, \quad r_1 = -1.$$

La méthode est donc zéro-stable car

$$\begin{cases} |r_j| \leq 1 \quad \text{pour tout } j = 0, \dots, p \\ \text{multiplicité de } r_j > 1 \implies |r_j| < 1 \end{cases}$$

Q2 [3 points] La méthode est elle consistente ? Est elle convergente ? Si oui, étudier théoriquement l'ordre de convergence.

La méthode étant zéro-stable, si elle est consistante, alors elle est convergente. Pour étudier la convergence introduisons la quantité

$$\begin{aligned} \xi(i) &= \sum_{j=0}^{p} (-j)^i a_j + i \sum_{j=-1}^{p} (-j)^{i-1} b_j \\ &= (-0)^i a_0 + (-1)^i a_1 + i\Big((1)^{i-1} b_{-1} + (-0)^{i-1} b_0 + (-1)^{i-1} b_1 \Big) \end{aligned}$$

avec $(-0)^0 = 1$.

Pour que la méthode soit consistante il faut que $\xi(0) = \xi(1) = 1$. De plus, pour que la méthode soit d'ordre ω, il faut que $\xi(i) = 1$ pour tout $i \leq \omega$.

De plus, ω vérifie la première barrière de Dahlquist : le schéma étant **implicite** à $q = 2$ (**pair**) pas consistante et zéro-stable, $\omega = q + 2 \leq 4$.

En calculant ξ on trouve que la méthode est d'ordre 2 (cf. calculs avec `sympy` ci-dessous).

```
aa_eval = [0,1]
bb_eval = [0,1]
bm1_eval = 1


################################################################################


def xi(i,aa,bb,bm1):
    sa = sum( [ (-j)**i * aa[j] for j in range(q) ] )
    sb = bm1+sum( [(-j)**(i-1)*bb[j] for j in range(1,q)] )
    if i==1:
        sb += bb[0]
    return (sa)+(i*sb)

q = len(aa_eval) # nb de pas
p = q-1  # j = 0...p # coeffs du schema
omega = q+2 if q%2==0 else q+1

aa = sp.symbols(f'a_0:{q}')
bb = sp.symbols(f'b_0:{q}')
bm1 = sp.Symbol('b_{-1}')

display(Markdown(f"C'est une méthode à $q={p + 1}$ pas d'ordre $\\omega\\le{omega}$ avec"))

texte = ''.join([ f"{sp.latex(aa[j])}={sp.latex(aa_eval[j])},"+r"\quad " for j in range(p+1)])
texte += ''.join([ f"{sp.latex(bb[j])}={sp.latex(bb_eval[j])},"+r"\quad " for j in range(p+1)])
```

```
texte += f"{sp.latex(bm1)}={sp.latex(bm1_eval)}"
display(Math( texte ))


xi_eval = [xi(i,aa_eval,bb_eval,bm1_eval) for i in range(omega+1)]
for i in range(omega+1):
    display(Math( f"\\xi({i}) = {sp.latex(xi(i,aa,bb,bm1))} = {sp.latex(xi_eval[i])}" ))
    if xi_eval[i]!=1:
        display(Markdown( f"La méthode est donc d'ordre $\\omega = {i-1}$" ))
        break
```

C'est une méthode à $q = 2$ pas d'ordre $\omega \leq 4$ avec

$a_0 = 0, \quad a_1 = 1, \quad b_0 = 0, \quad b_1 = 1, \quad b_{-1} = 1$

$\xi(0) = a_0 + a_1 = 1.0$

$\xi(1) = -a_1 + b_0 + b_1 + b_{-1} = 1$

$\xi(2) = a_1 - 2b_1 + 2b_{-1} = 1$

$\xi(3) = -a_1 + 3b_1 + 3b_{-1} = 5$

La méthode est donc d'ordre $\omega = 2$

Q3 [3 points] Vérifier empiriquement l'ordre de convergence sur le problème de Cauchy

$$\begin{cases} y'(t) = (y(t)+2)\dfrac{2t^2+2t-1}{1+t}, & \forall t \in I = [0,2], \\ y(0) = -1, \end{cases}$$

```
# variables globales
t0     = 0
tfinal = 2
y0     = -1

phi = lambda t,y: (y+2)*(2*t**2+2*t-1)/(1+t)

################################################
# solution exacte
################################################
t   = sp.Symbol('t')
y   = sp.Function('y')
edo = sp.Eq( sp.diff(y(t),t) , phi(t,y(t)) )
solgen = sp.dsolve(edo)
# display(solgen)
consts = sp.solve( sp.Eq( y0, solgen.rhs.subs(t,t0)) , dict=True)[0]
# display(consts)
solpar = solgen.subs(consts)
display(solpar)

sol_exacte = sp.lambdify(t,solpar.rhs,'numpy')

################################################
# schéma (initialisation avec sol exacte pour ordre de convergence)
################################################
def multipas(phi, tt, sol_exacte):
    h  = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    uu[0:2] = sol_exacte(tt[0:2])
    for i in range(1,len(tt) - 1):
        eq = lambda x : -x+uu[i-1]+h*(phi(tt[i+1],x)+phi(tt[i-1],uu[i-1]))
        temp = fsolve( eq ,uu[i])
        uu[i+1] = temp[0]
```

```
    return uu

##############################################
# ordre
##############################################
H = []
err = []
N = 10

plt.figure(figsize=(16,5))

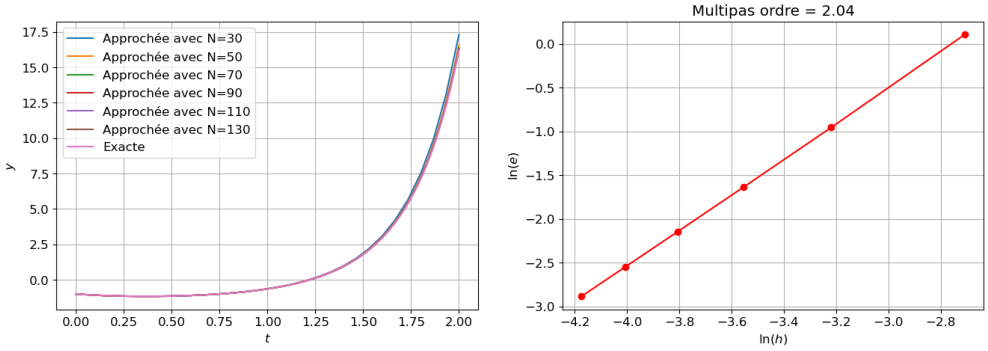
ax1 = plt.subplot(1,2,1)
plt.xlabel('$t$')
plt.ylabel('$y$')
ax1.grid(True)

for k in range(6):
    N += 20
    tt = np.linspace(t0, tfinal, N + 1)
    H.append( tt[1] - tt[0] )
    yy = sol_exacte(tt)
    uu = multipas(phi, tt, sol_exacte)
    err.append( np.linalg.norm(uu-yy,np.inf) )
    ax1.plot(tt,uu,label=f'Approchée avec N={N}')

ax1.plot(tt,yy,label='Exacte') # sur la grille la plus fine
ax1.legend()


ax2 = plt.subplot(1,2,2)
ax2.plot( np.log(H), np.log(err), 'r-o')
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
plt.title(f'Multipas ordre = {np.polyfit(np.log(H),np.log(err),1)[0]:1.2f}')
ax2.grid(True)
```

$$y(t) = \frac{-2t + e^{t^2} - 2}{t+1}$$



27.3. EXERCICE 3 : étude d'un schéma RK

Considérons le schéma dont la matrice de Butcher est donnée ci-dessous :

$$\begin{array}{c|cc} \sigma & \sigma & 0 \\ 1 & \vartheta & 1-\vartheta \\ \hline & \gamma & 1-\gamma \end{array}$$

1. Déterminer les coefficients σ, ϑ, γ pour que le schéma soit d'ordre 3. Est-il d'ordre 4 ?
2. Étudier empiriquement l'ordre de convergence sur le problème de Cauchy

$$\begin{cases} y'(t) = \dfrac{t^2 + y^2(t)}{ty(t)}, & t \in [1;5] \\ y(1) = 1 \end{cases}$$

3. Étudier théoriquement la A-stabilité du schéma optimal.

Q1 [**5 points**] Pour quelles valeurs des paramètres le schéma est d'ordre 3 ? Et d'ordre 4 ?

Soit ω l'ordre de la méthode.

C'est un schéma implicite à 2 étages ($s = 2$) donc $\omega \leq 2s = 4$.

```
sigma = sp.Symbol(r"\sigma")
theta = sp.Symbol(r"\vartheta")
gamma = sp.Symbol(r"\gamma")


def ordre_RK(s, A=None, b=None, c=None):

    j = sp.symbols('j')

    if A is None:
        A = sp.MatrixSymbol('a', s, s)
    else:
        A = sp.Matrix(A)
    if c is None:
        c = sp.symbols(f'c_0:{s}')
    if b is None:
        b = sp.symbols(f'b_0:{s}')


    display(Markdown("**Matrice de Butcher**"))
    matrice_Butcher(s, A, b, c)

    display(Markdown(f"**On a {s+1} conditions pour avoir consistance = pour être d'ordre 1**"))
    ordre_1(s, A, b, c, j)

    display(Markdown("**On doit ajouter 1 condition pour être d'ordre 2**"))
    ordre_2(s, A, b, c, j)

    display(Markdown("**On doit ajouter 2 conditions pour être d'ordre 3**"))
    ordre_3(s, A, b, c, j)

    display(Markdown("**On doit ajouter 4 conditions pour être d'ordre 4**"))
    ordre_4(s, A, b, c, j)

    return None


def matrice_Butcher(s, A, b, c):
    But = sp.Matrix(A)
    But = But.col_insert(0, sp.Matrix(c))
    last = [sp.Symbol(" ")]
    last.extend(b)
    But = But.row_insert(s,sp.Matrix(last).T)
    display(Math(sp.latex(sp.Matrix(But))))
    return None


def ordre_1(s, A, b, c, j):
    texte = r"\sum_{j=1}^s b_j =" + f"{sum(b).simplify()}"
    texte += r"\text{ doit être égale à }1"
```

```
    display(Math(texte))
    for i in range(s):
        somma = sp.summation(A[i,j],(j,0,s-1)).simplify()
        texte = r'\sum_{j=1}^s a_{'+str(i)+r'j}=' + sp.latex( somma )
        texte += r"\text{ doit être égale à }"+sp.latex(c[i])
        display(Math( texte ))
    return None


def ordre_2(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j=' +sp.latex(sum([b[i]*c[i] for i in range(s)]).simplify())
    texte += r"\text{ doit être égale à }\frac{1}{2}"
    display(Math(texte))
    return None


def ordre_3(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j^2='
    texte += sp.latex( sum([b[i]*c[i]**2 for i in range(s)]).simplify() )
    texte += r"\text{ doit être égale à }\frac{1}{3}"
    display(Math(texte))
    texte = r'\sum_{i,j=1}^s b_i a_{ij} c_j='
    somma = sum([b[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{6}"
    display(Math(texte))
    return None


def ordre_4(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j^3='
    texte += sp.latex( sum([b[i]*c[i]**3 for i in range(s)]).simplify() )
    texte += r"\text{ doit être égale à }\frac{1}{4}"
    display(Math(texte))
    texte = r'\sum_{i,j=1}^s b_i c_i a_{ij} c_j='
    somma = sum([b[i]*c[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{8}"
    display(Math(texte))
    texte = r'\sum_{i,j=1}^s b_i a_{ij} c_j^2='
    somma = sum([b[i]*A[i,j]*c[j]**2 for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{12}"
    display(Math(texte))
    texte = r'\sum_{i,j,k=1}^s b_i a_{ij} a_{jk} c_k='
    somma = sum([b[i]*A[i,j]*A[j,k]*c[k] for k in range(s) for j in range(s) for i in
 ↪  range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{24}"
    display(Math(texte))
    return None


# APPLICATION A NOTRE CAS
c = [sigma,sp.S(1)]
b = [gamma,1-gamma]
A = [[sigma,sp.S(0)],[theta,1-theta]]
s = len(c)
display(Markdown(f"La méthode de Runge-Kutta est à {s} étages"))
ordre_RK(s,A,b,c)
```

La méthode de Runge-Kutta est à 2 étages

Matrice de Butcher

$$\begin{bmatrix} \sigma & \sigma & 0 \\ 1 & \vartheta & 1-\vartheta \\ & \gamma & 1-\gamma \end{bmatrix}$$

On a 3 conditions pour avoir consistance = pour être d'ordre 1

$\sum_{j=1}^{s} b_j = 1$ doit être égale à 1

$\sum_{j=1}^{s} a_{0j} = \sigma$ doit être égale à σ

$\sum_{j=1}^{s} a_{1j} = 1$ doit être égale à 1

On doit ajouter 1 condition pour être d'ordre 2

$\sum_{j=1}^{s} b_j c_j = \gamma\sigma - \gamma + 1$ doit être égale à $\frac{1}{2}$

On doit ajouter 2 conditions pour être d'ordre 3

$\sum_{j=1}^{s} b_j c_j^2 = \gamma\sigma^2 - \gamma + 1$ doit être égale à $\frac{1}{3}$

$\sum_{i,j=1}^{s} b_i a_{ij} c_j = \gamma\sigma^2 - \sigma\vartheta\left(\gamma-1\right) + \left(\gamma-1\right)\left(\vartheta-1\right)$ doit être égale à $\frac{1}{6}$

On doit ajouter 4 conditions pour être d'ordre 4

$\sum_{j=1}^{s} b_j c_j^3 = \gamma\sigma^3 - \gamma + 1$ doit être égale à $\frac{1}{4}$

$\sum_{i,j=1}^{s} b_i c_i a_{ij} c_j = \gamma\sigma^3 - \sigma\vartheta\left(\gamma-1\right) + \left(\gamma-1\right)\left(\vartheta-1\right)$ doit être égale à $\frac{1}{8}$

$\sum_{i,j=1}^{s} b_i a_{ij} c_j^2 = \gamma\sigma^3 - \sigma^2\vartheta\left(\gamma-1\right) + \left(\gamma-1\right)\left(\vartheta-1\right)$ doit être égale à $\frac{1}{12}$

$\sum_{i,j,k=1}^{s} b_i a_{ij} a_{jk} c_k = \gamma\sigma^3 - \sigma^2\vartheta\left(\gamma-1\right) + \sigma\vartheta\left(\gamma-1\right)\left(\vartheta-1\right) + \left(1-\gamma\right)\left(\vartheta-1\right)^2$ doit être égale à $\frac{1}{24}$

On commence à resoudre le système dont les inconnues sont les paramètres $\sigma, \gamma, \vartheta$ pour avoir au moins l'ordre 3 :

```
cnst = sp.solve( [ sum([b[i]*c[i] for i in range(s)])-1/sp.S(2),
                   sum([b[i]*c[i]**2 for i in range(s)])-1/sp.S(3),
                   sum([b[i]*A[i][j]*c[j] for i in range(s) for j in range(s)])-1/sp.S(6)
        ] )[0]

display(Math(sp.latex(cnst)))


c_substituted = [expr.subs(cnst) for expr in c]
b_substituted = [expr.subs(cnst) for expr in b]
A_substituted = [[expr.subs(cnst) for expr in row] for row in A]
```

$$\left\{\gamma : \frac{3}{4},\ \sigma : \frac{1}{3},\ \vartheta : 1\right\}$$

Remarque : le choix optimal conduit au même schéma du DM de 2020, l'étude a donc déjà été fait en TD et en TP !

Le schéma n'est pas d'ordre 4 :

```
ordre_RK(s,A_substituted,b_substituted,c_substituted)
```

Matrice de Butcher

$$\begin{bmatrix} \frac{1}{3} & \frac{1}{3} & 0 \\ 1 & 1 & 0 \\ & \frac{3}{4} & \frac{1}{4} \end{bmatrix}$$

On a 3 conditions pour avoir consistance = pour être d'ordre 1

$\sum_{j=1}^{s} b_j = 1$ doit être égale à 1

$\sum_{j=1}^{s} a_{0j} = \frac{1}{3}$ doit être égale à $\frac{1}{3}$

$\sum_{j=1}^{s} a_{1j} = 1$ doit être égale à 1

On doit ajouter 1 condition pour être d'ordre 2

$\sum_{j=1}^{s} b_j c_j = \frac{1}{2}$ doit être égale à $\frac{1}{2}$

On doit ajouter 2 conditions pour être d'ordre 3

$\sum_{j=1}^{s} b_j c_j^2 = \frac{1}{3}$ doit être égale à $\frac{1}{3}$

$\sum_{i,j=1}^{s} b_i a_{ij} c_j = \frac{1}{6}$ doit être égale à $\frac{1}{6}$

On doit ajouter 4 conditions pour être d'ordre 4

$\sum_{j=1}^{s} b_j c_j^3 = \frac{5}{18}$ doit être égale à $\frac{1}{4}$

$\sum_{i,j=1}^{s} b_i c_i a_{ij} c_j = \frac{1}{9}$ doit être égale à $\frac{1}{8}$

$\sum_{i,j=1}^{s} b_i a_{ij} c_j^2 = \frac{1}{18}$ doit être égale à $\frac{1}{12}$

$\sum_{i,j,k=1}^{s} b_i a_{ij} a_{jk} c_k = \frac{1}{18}$ doit être égale à $\frac{1}{24}$

Q2 [3 points] Étudier empiriquement l'ordre de convergence sur le problème de Cauchy

$$\begin{cases} y'(t) = \dfrac{t^2 + y^2(t)}{t y(t)}, & t \in [1;5] \\ y(1) = 1 \end{cases}$$

```
# variables globales
t0     = 1
tfinal = 5
y0     = 1

phi = lambda t,y: (t**2+y**2)/(t*y)

################################################
# solution exacte
################################################
```

```
t   = sp.Symbol('t')
y   = sp.Function('y')
edo = sp.Eq( sp.diff(y(t),t) , phi(t,y(t)) )
solpar = sp.dsolve(edo,y(t),ics={y(t0):y0})
display(solpar)


##############################################
# solution approchée
##############################################
sol_exacte = sp.lambdify(t,solpar.rhs,'numpy')

# Je note sigma_e etc pour sigma evaluée comme un float
sigma_e = 1/3
gamma_e = 1/(2*(1-sigma_e)) # pour ordre 2
theta_e = 1

def RK(tt,phi,y0):
    uu = [y0]
    h = tt[1]-tt[0]
    for i in range(len(tt)-1):
        sys = lambda X : [ -X[0]+phi( tt[i]+h*sigma_e , uu[i]+h*X[0]*sigma_e  ) ,
                           -X[1]+phi( tt[i]+h*theta_e , uu[i]+h*(X[0]*theta_e+X[1]*(1-theta_e))  )
                           ]
        K_start = phi(tt[i],uu[i])
        K1, K2 = fsolve( sys , [K_start,K_start ] )
        uu.append( uu[i]+h*(gamma_e*K1+(1-gamma_e)*K2) )
    return uu


##############################################
# ordre
##############################################
H = []
err = []

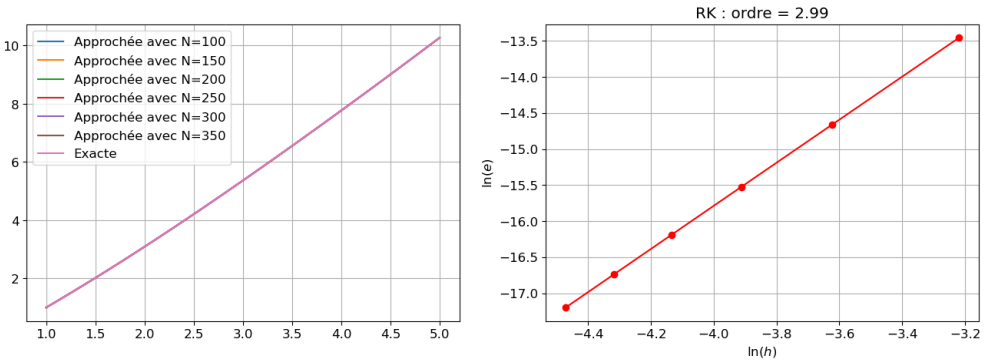
plt.figure(figsize=(16,5))
ax1 = plt.subplot(1,2,1)
ax2 = plt.subplot(1,2,2)

N = 50
for k in range(6):
    N += 50
    tt = np.linspace(t0, tfinal, N + 1)
    H.append( tt[1] - tt[0] )
    yy = sol_exacte(tt)
    uu = RK(tt,phi,y0)
    err.append( np.linalg.norm(np.array(uu)-yy,np.inf) )
    ax1.plot(tt,uu,label=f'Approchée avec N={N}')

ax1.plot(tt,yy,label='Exacte')
plt.xlabel('$t$')
plt.ylabel('$y$')
ax1.grid(True)
ax1.legend()

ax2.plot( np.log(H), np.log(err), 'r-o')
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
plt.title(f'RK : ordre = {np.polyfit(np.log(H),np.log(err),1)[0]:1.2f}')
ax2.grid(True)
```

$y(t) = t\sqrt{2\log(t) + 1}$



Q3 [2 points] Étudier théoriquement la A-stabilité du schéma optimal.

La méthode appliquée à l'EDO $y'(t) = -\beta y(t)$ avec $\beta > 0$ s'écrit :

$$\begin{cases} u_0 = y_0 \\ K_1 = -\beta u_n - (\beta h)\sigma K_1 \rightsquigarrow K_1 = \frac{-\beta}{1+(\beta h)\sigma} u_n \\ K_2 = -\beta\left(u_n + hK_1\vartheta + hK_2(1-\vartheta)\right) \rightsquigarrow K_2 = \frac{1+\frac{(\beta h)\vartheta}{1+(\beta h)\sigma}}{1+(\beta h)(1-\vartheta)}(-\beta u_n) \\ u_{n+1} = u_n + \frac{h}{4}\left(3K_1 + K_2\right) \end{cases} \qquad n = 0, 1, \dots N-1$$

En particulier, dans le cas d'ordre 3, on a

$$\begin{cases} u_0 = y_0 \\ K_1 = -\beta u_n - \beta\frac{h}{3}K_1 \rightsquigarrow K_1 = \frac{-3\beta}{3+\beta h} u_n \\ K_2 = -\beta\left(u_n + hK_1\right) \rightsquigarrow K_2 = \frac{2\beta^2 h - 3\beta}{3+\beta h} u_n \\ u_{n+1} = u_n + \frac{h}{4}\left(3K_1 + K_2\right) \end{cases} \qquad n = 0, 1, \dots N-1$$

L'étude ci-dessous montre que, dans ce cas, le schéma est A-stable ssi $h < \frac{6}{\beta}$.

```
sp.var('u_n, h,  , K1, K2,')

g = lambda y : - *y
eq1 = sp.Eq( K1 , g(u_n+h*K1*sigma))
eq2 = sp.Eq( K2 , g(u_n+h*(K1*theta+K2*(1-theta))) )
sol = sp.solve([eq1,eq2],[K1,K2])
sol = { key:value.factor() for key,value in sol.items() }
for key,value in sol.items():
    texte = sp.latex(key) + '=' + sp.latex(value)
    display(Math(texte))

RHS = u_n+h*(gamma*K1+(1-gamma)*K2).factor()
RHS = RHS.subs(sol).simplify().factor()
texte = 'u_{n+1}=' + sp.latex(RHS)
display(Math(texte))

print(r"Notons x=h >0. On trouve donc")
x = sp.Symbol('x',positive=True)
RHS = RHS.subs(h* ,x).simplify()
texte = 'u_{n+1}=' + sp.latex(RHS)
display(Math(texte))

print("Considérons le schéma d'ordre 3:")
for key,value in cnst.items():
    texte = sp.latex(key) + '=' + sp.latex(value)
    display(Math(texte))

RHS = RHS.subs(cnst)
```

```
texte = r"\text{La méthode est A-stable ssi }|q(x)|<1\text{ avec }q(x)=\frac{u_{n+1}}{u_n}="
texte += sp.latex(RHS/u_n)
display(Math(texte))
sp.solve([-1<RHS/u_n,RHS/u_n<1])
```

$$K_1 = -\frac{u_n \beta}{\sigma h \beta + 1}$$

$$K_2 = \frac{u_n \beta \left(\sigma h \beta - \vartheta h \beta + 1\right)}{\left(\sigma h \beta + 1\right) \left(\vartheta h \beta - h \beta - 1\right)}$$

$$u_{n+1} = -\frac{u_n \left(\gamma \sigma h^2 \beta^2 - \gamma h^2 \beta^2 - \sigma \vartheta h^2 \beta^2 + \sigma h \beta + \vartheta h^2 \beta^2 - \vartheta h \beta + 1\right)}{\left(\sigma h \beta + 1\right) \left(\vartheta h \beta - h \beta - 1\right)}$$

Notons x=h >0. On trouve donc

$$u_{n+1} = \frac{u_n \left(\gamma \sigma x^2 - \gamma x^2 - \sigma \vartheta x^2 + \sigma x + \vartheta x^2 - \vartheta x + 1\right)}{\left(\sigma x + 1\right) \left(-\vartheta x + x + 1\right)}$$

Considérons le schéma d'ordre 3:

$$\gamma = \frac{3}{4}$$

$$\sigma = \frac{1}{3}$$

$$\vartheta = 1$$

La méthode est A-stable ssi $|q(x)| < 1$ avec $q(x) = \frac{u_{n+1}}{u_n} = \frac{\frac{x^2}{6} - \frac{2x}{3} + 1}{\frac{x}{3} + 1}$

$$x < 6$$

Chapitre 28.

2023 Contrôle Terminal session 1

2h30, deposer le notebook dans Moodle

```
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp
from scipy.optimize import fsolve
from scipy.integrate import solve_ivp, odeint
from IPython.display import display, Markdown, Math
```

28.1. EXERCICE 1 : étude d'un système

On s'intèresse au problème de Cauchy

$$\begin{cases} y'(t) = \sin(y(t)) - 2z(t)e^{y(t)}, \\ z'(t) = z(t)\Big(z(t)e^{y(t)} - \cos(y(t))\Big), \\ y(0) = \frac{1}{2}, \\ z(0) = \frac{1}{4}, \end{cases} \qquad t \in [0, 30]$$

Q1 [1 point] Notons $E(y, z) = z\sin(y) - z^2e^y$ et $\mathcal{E}(t) = E(y(t), z(t))$. Montrer que $\mathcal{E}$ est un invariant.

On remarque que $\frac{\partial E}{\partial y}(y(t), z(t)) = -z'(t)$ et $\frac{\partial E}{\partial z}(y(t), z(t)) = y'(t)$. On a alors

$$\mathcal{E}'(t) = \frac{d}{dt}E(y(t), z(t)) = \frac{\partial E}{\partial y}y'(t) + \frac{\partial E}{\partial z}z'(t) = -z'(t)y'(t) + y'(t)z'(t) = 0.$$

Donc $\mathcal{E}(t) = \mathcal{E}(0) = 0$.

Q2 [2 points] Calculer la solution approchée obtenue avec le schéma d'Euler Explicite avec $N = 100$. Dans trois repères différents afficher

- dans le premier repère $t \mapsto y(t)$ et $t \mapsto z(t)$ dans le même,
- dans le deuxième repère $y \mapsto z(y)$,
- dans le troisième repère $t \mapsto \mathcal{E}(t)$.

```
def affichage(tt,uu,ww,HH_num,s):

    plt.suptitle(f'Schéma {s}')
    #plt.tight_layout()

    plt.subplot(1,3,1)
    plt.plot(tt,uu,'o--',label=r'$y(t)$')
    plt.plot(tt,ww,'d--',label=r'$z(t)$')
    plt.xlabel('t')
    plt.legend()
    plt.grid()
```

```
    plt.subplot(1,3,2)
    plt.plot(uu,ww,'o--')
    plt.xlabel('y')
    plt.ylabel('z')
    plt.grid()
    Y1,Y2 = np.meshgrid(np.linspace(min(uu),max(uu),21),np.linspace(min(ww),max(ww),21))

    V1,V2 = pphi(tt,[Y1,Y2])
    r1 = np.sqrt(1+V1**2)
    r2 = np.sqrt(1+V2**2)
    plt.quiver(Y1, Y2, V1/r1, V2/r2, cmap=plt.cm.viridis, scale_units='xy')
    # plt.axis('equal')

    plt.subplot(1,3,3)
    plt.plot(tt,HH_num,'o--')
    plt.xlabel('t')
    plt.ylabel(r'$\mathcal{E}$')
    plt.grid();


t0 = 0
y0, z0 = 0.5, 0.25

tfinal = 30
N = 100

phi1 = lambda t,y,z : np.sin(y)-2*z*np.exp(y)
phi2 = lambda t,y,z : z*(z*np.exp(y)-np.cos(y))
Ham = lambda y,z : z*np.sin(y)-z**2*np.exp(y)

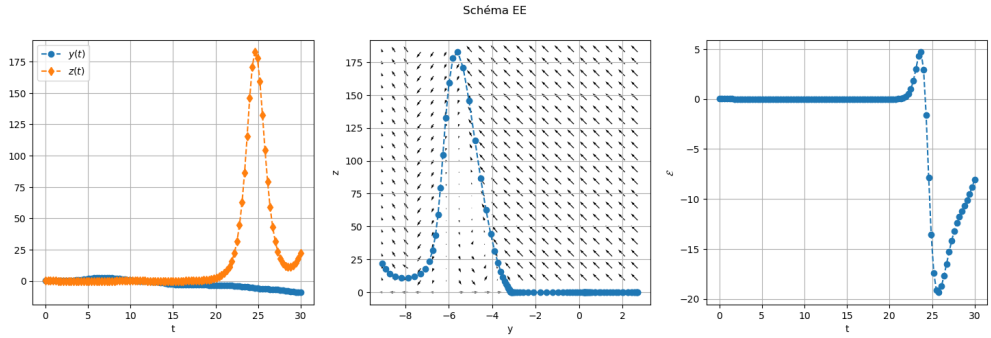
tt = np.linspace(t0,tfinal,N+1)

# pour odeint
pphi = lambda t,yy : [phi1(t,yy[0],yy[1]),phi2(t,yy[0],yy[1])]
yy0  = [y0,z0]
```

```
def EE(phi1,phi2,tt,y0,z0):
    h  = tt[1]-tt[0]
    uu = np.zeros_like(tt)
    ww = np.zeros_like(tt)
    uu[0] = y0
    ww[0] = z0
    for i in range(len(tt)-1):
        uu[i+1] = uu[i]+h*phi1(tt[i],uu[i],ww[i])
        ww[i+1] = ww[i]+h*phi2(tt[i],uu[i],ww[i])
    return [uu,ww]


[uu, ww] = EE(phi1,phi2,tt,y0,z0)
HH_num = Ham(uu,ww)

plt.figure(figsize=(18,5))
affichage(tt,uu,ww,HH_num,"EE")
```

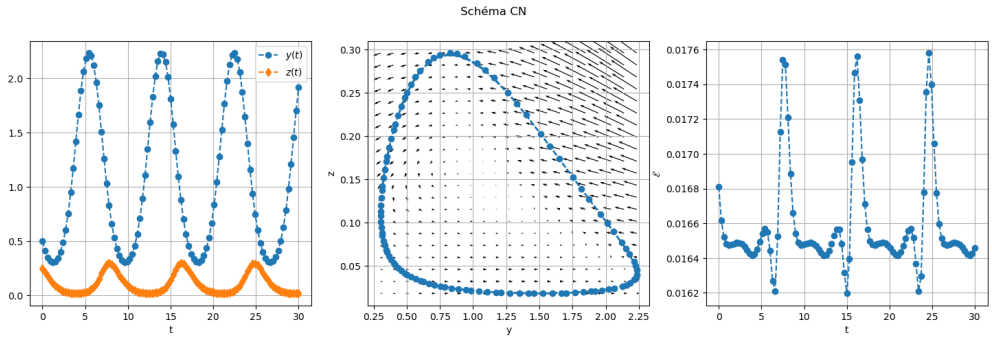


Q3 [3 points]
Même exercice avec le schéma de Cranck Nicolson.

```
def CN(phi1,phi2,tt,y0,z0):
    h  = tt[1]-tt[0]
    uu = np.zeros_like(tt)
    ww = np.zeros_like(tt)
    uu[0] = y0
    ww[0] = z0
    for i in range(len(tt)-1):
        sys = lambda vv : [-vv[0]+uu[i]+h/2*(phi1(tt[i],uu[i],ww[i])+phi1(tt[i+1], vv[0], vv[1])),
                           -vv[1]+ww[i]+h/2*(phi2(tt[i],uu[i],ww[i])+phi2(tt[i+1], vv[0], vv[1])) ]
        uu[i+1], ww[i+1] = fsolve( sys, [uu[i],ww[i]] )
    return [uu,ww]

[uu, ww] = CN(phi1,phi2,tt,y0,z0)
HH_num = Ham(uu,ww)

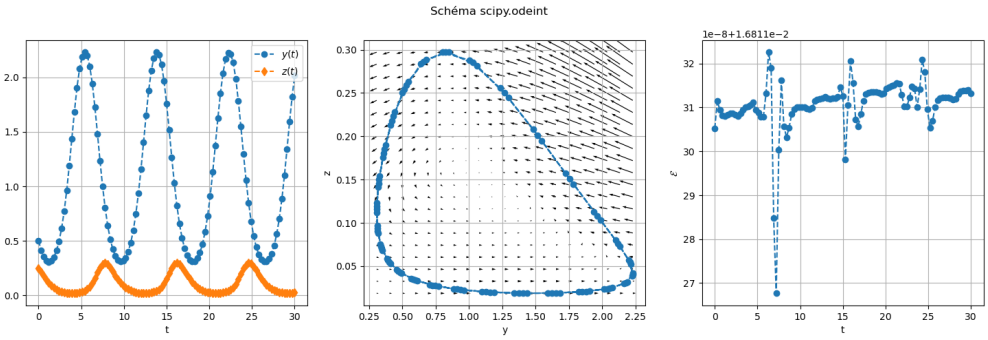
plt.figure(figsize=(18,5))
affichage(tt,uu,ww,HH_num,"CN")
```



Q4 [1 points]
Même exercice avec `odeint`.

```
uu,ww = odeint(pphi,yy0,tt,tfirst=True).T
HH_num = Ham(uu,ww)

plt.figure(figsize=(18,5))
affichage(tt,uu,ww,HH_num,"scipy.odeint")
```



28.2. EXERCICE 2 : étude d'un schéma multipas

On notera $\varphi_k \overset{\text{déf}}{=} \varphi(t_k, u_k)$. Considérons la méthode multipas

$$u_{n+1} = \frac{4}{3}u_n - \frac{1}{3}u_{n-1} + h\frac{2}{3}\varphi_{n+1}$$

Q2 [1 points] Étudier la zéro-stabilité.

Le schéma est de la forme

$$u_{n+1} = a_0 u_n + a_1 u_{n-1} + h\left(b_0\varphi_n + b_1\varphi_{n-1}\right) + hb_{-1}\varphi_{n+1}.$$

On a

- $p = 1$: c'est un méthode à $q = p + 1 = 2$ pas ;
- $a_0 = \frac{4}{3}$ et $a_1 = -\frac{1}{3}$,
- $b_0 = 0$ et $b_1 = 0$,
- $b_{-1} = \frac{2}{3} \neq 0$ ($\implies$ la méthode est implicite).

Le premier polynôme caractéristique est

$$\varrho(r) = r^2 - a_0 r - a_1 = r^2 - \frac{4}{3}r + \frac{1}{3} = (r-1)\left(r - \frac{1}{3}\right).$$

Les racines sont

$$r_0 = 1, \quad r_1 = \frac{1}{3}.$$

La méthode est donc zéro-stable car

$$\begin{cases} |r_j| \leq 1 \quad \text{pour tout } j = 0, \dots, p \\ \text{multiplicité de } r_j > 1 \implies |r_j| < 1 \end{cases}$$

Q2 [3 points] La méthode est elle consistente ? Est elle convergente ? Si oui, étudier théoriquement l'ordre de convergence.

On a prouvé que la méthode est zéro-stable. Ainsi, si elle est consistante, alors elle est convergente.

Introduisons la quantité

$$\begin{aligned} \xi(i) &= \sum_{j=0}^{p}(-j)^i a_j + i\sum_{j=-1}^{p}(-j)^{i-1}b_j \\ &= (-0)^i a_0 + (-1)^i a_1 + i\Big((1)^{i-1}b_{-1} + (-0)^{i-1}b_0 + (-1)^{i-1}b_1\Big) \\ &= (-0)^i\frac{4}{3} - (-1)^i\frac{1}{3} + (1)^{i-1}i\frac{2}{3} \end{aligned}$$

avec $(-0)^0 = 1$.

Pour que la méthode soit consistante il faut que $\xi(0) = \xi(1) = 1$. De plus, pour que la méthode soit d'ordre ω, il faut que $\xi(i) = 1$ pour tout $i \leq \omega$.

De plus, ω vérifie la première barrière de Dahlquist : le schéma étant **implicite** à $q = 2$ (**pair**) pas consistante et zéro-stable, $\omega = q + 2 \leq 4$.

En calculant ξ on trouve que la méthode est d'ordre 2 :

- $\xi(0) = (-0)^0 \frac{4}{3} - (-1)^0 \frac{1}{3} + (1)^{0-1} 0 \frac{2}{3} = \frac{4}{3} - \frac{1}{3} = 1$
- $\xi(1) = (-0)^1 \frac{4}{3} - (-1)^1 \frac{1}{3} + (1)^{1-1} 1 \frac{2}{3} = \frac{1}{3} + \frac{2}{3} = 1$
- $\xi(2) = (-0)^2 \frac{4}{3} - (-1)^2 \frac{1}{3} + (1)^{2-1} 2 \frac{2}{3} = -\frac{1}{3} + 2\frac{2}{3} = 1$
- $\xi(3) = \frac{1}{3} + 3\frac{2}{3} = \frac{7}{3}$

(vérification avec `sympy` ci-dessous).

```
aa_eval = [sp.Rational(4,3), -sp.Rational(1,3)]
bb_eval = [0,0]
bm1_eval = sp.Rational(2,3)

######################################################


def xi(i,aa,bb,bm1):
    sa = sum( [ (-j)**i * aa[j] for j in range(q) ] )
    sb = bm1+sum( [(-j)**(i-1)*bb[j] for j in range(1,q)] )
    if i==1:
        sb += bb[0]
    return (sa).factor()+(i*sb).factor()

q = len(aa_eval) # nb de pas
p = q-1  # j = 0...p # coeffs du schema
omega = q+2 if q%2==0 else q+1

aa = sp.symbols(f'a_0:{q}')
bb = sp.symbols(f'b_0:{q}')
bm1 = sp.Symbol('b_{-1}')

display(Markdown(f"C'est une méthode à $q={p + 1}$ pas d'ordre $\\omega\\le{omega}$ avec"))

texte = ''.join([ f"{sp.latex(aa[j])}={sp.latex(aa_eval[j])},"+r"\quad " for j in range(p+1)])
texte += ''.join([ f"{sp.latex(bb[j])}={sp.latex(bb_eval[j])},"+r"\quad " for j in range(p+1)])
texte += f"{sp.latex(bm1)}={sp.latex(bm1_eval)}"
display(Math( texte ))


xi_eval = [xi(i,aa_eval,bb_eval,bm1_eval) for i in range(omega+1)]
for i in range(omega+1):
    display(Math( f"\\xi({i}) = {sp.latex(xi(i,aa,bb,bm1))} = {sp.latex(xi_eval[i])}" ))
    if xi_eval[i]!=1:
        display(Markdown( f"La méthode est donc d'ordre $\\omega = {i-1}$" ))
        break
```

C'est une méthode à $q = 2$ pas d'ordre $\omega \leq 4$ avec

$$a_0 = \frac{4}{3}, \quad a_1 = -\frac{1}{3}, \quad b_0 = 0, \quad b_1 = 0, \quad b_{-1} = \frac{2}{3}$$

$$\xi(0) = a_0 + a_1 = 1$$

$$\xi(1) = -a_1 + b_0 + b_1 + b_{-1} = 1$$

$$\xi(2) = a_1 - 2\,(b_1 - b_{-1}) = 1$$

$$\xi(3) = -a_1 + 3\,(b_1 + b_{-1}) = \frac{7}{3}$$

La méthode est donc d'ordre $\omega = 2$

Q3 [3 points] Vérifier empiriquement l'ordre de convergence sur le problème de Cauchy

$$\begin{cases} y'(t) = (y(t)+2)\dfrac{2t^2+2t-1}{1-t^2}, & \forall t \in I = [2,5], \\ y(2) = 1, \end{cases}$$

```
# variables globales
t0     = 2
tfinal = 5
y0     = 1

phi = lambda t,y: (y+2)*(2*t**2+2*t-1)/(1-t**2)

################################################
# solution exacte
################################################
t   = sp.Symbol('t')
y   = sp.Function('y')
edo = sp.Eq( sp.diff(y(t),t) , phi(t,y(t)) )
solgen = sp.dsolve(edo)
# display(solgen)
consts = sp.solve( sp.Eq( y0, solgen.rhs.subs(t,t0)) , dict=True)[0]
# display(consts)
solpar = solgen.subs(consts)
display(solpar)

sol_exacte = sp.lambdify(t,solpar.rhs,'numpy')

################################################
# schéma (initialisation avec sol exacte pour ordre de convergence)
################################################
def multipas(phi, tt, sol_exacte):
    h  = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    uu[0:2] = sol_exacte(tt[0:2])
    # uu = [sol_exacte(tt[i]) for i in range(2)]
    for i in range(1,len(tt) - 1):
        eq = lambda x : -x+4/3*uu[i]-1/3*uu[i-1]+h*2/3*phi(tt[i+1],x)
        temp = fsolve( eq ,uu[i])
        uu[i+1] = temp[0]
    return uu

################################################
# ordre
################################################
H = []
err = []
N = 10


plt.figure(figsize=(16,5))

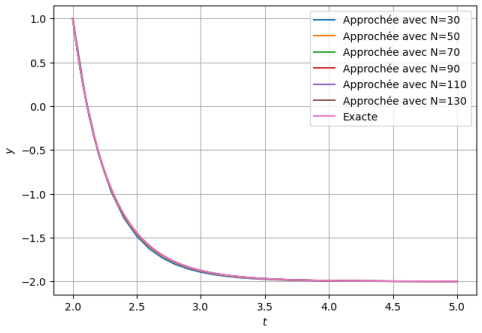
ax1 = plt.subplot(1,2,1)
plt.xlabel('$t$')
plt.ylabel('$y$')

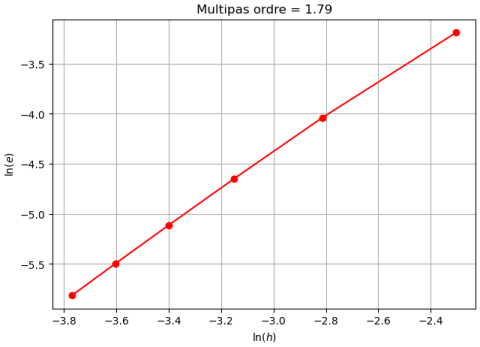
for k in range(6):
    N += 20
    tt = np.linspace(t0, tfinal, N + 1)
    H.append( tt[1] - tt[0] )
    yy = sol_exacte(tt)
    uu = multipas(phi, tt, sol_exacte)
    err.append( np.linalg.norm(uu-yy,np.inf) )
    ax1.plot(tt,uu,label=f'Approchée avec N={N}')
```

```
ax1.plot(tt,yy,label='Exacte') # sur la grille la plus fine
ax1.grid(True)
ax1.legend()


ax2 = plt.subplot(1,2,2)
ax2.plot( np.log(H), np.log(err), 'r-o')
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
plt.title(f'Multipas ordre = {np.polyfit(np.log(H),np.log(err),1)[0]:1.2f}')
ax2.grid(True)
```

$$y(t) = -2 + \frac{3\sqrt{3}e^4 e^{-2t}}{(t-1)^{\frac{3}{2}}\sqrt{t+1}}$$





28.3. EXERCICE 3 : étude d'un schéma RK

Considérons le schéma dont la matrice de Butcher est donnée ci-dessous :

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ \eta & \eta & 0 & 0 \\ \sigma & 0 & \sigma & 0 \\ \hline & 1-\gamma & 0 & \gamma \end{array}$$

Dans le sujet de cet exercice, il y a un paramètre η. Pour le fixer, remplacez mon nom et prénom par les vôtres et vous obtiendrez la valeur pour ce paramètre dans tout l'exercice.

```
    # ========================================================
nom = "Faccanoni"
prenom = "Gloria"
# ========================================================
L = list(range(2,5))
idx = sum([ord(c) for c in nom+prenom])%len(L)
print("Paramètre de l'exercice ")
display(Math(f"\\eta = {sp.latex(sp.Rational(1,L[idx]))}"))
```

```
Paramètre de l'exercice
```

$$\eta = \frac{1}{3}$$

Q1 [3 points] Pour quelles valeurs des paramètres σ et γ le schéma est d'ordre 2 ? Et d'ordre 3 ?

Soit ω l'ordre de la méthode.

- C'est un schéma explicite à 3 étages ($s = 3$) donc $\omega \leq s = 3$.

- Il est consistant (i.e. d'ordre 1) pour tout σ et tout γ.

Les conditions pour être d'ordre 2 et 3 sont rappelées ci-dessous. On trouve que :

- il est d'ordre 2 ssi $\sigma\gamma = \frac{1}{2}$,
- il est d'ordre 3 ssi, de plus, $\sigma^2\gamma = \frac{1}{3}$ et $\sigma\gamma\eta = \frac{1}{6}$; autrement dit, ssi $\sigma\gamma = \frac{1}{2}$, $\sigma = \frac{2}{3}$ et $\eta = \frac{1}{3}$.

Conclusion : il est au moins d'ordre 2 ssi $\sigma\gamma = \frac{1}{2}$, et d'ordre 3 ssi $\sigma = \frac{2}{3}$, $\gamma = \frac{3}{4}$ et $\eta = \frac{1}{3}$.

Ci-dessous la vérification avec `sympy`.

```
sigma = sp.Symbol(r"\sigma")
gamma = sp.Symbol(r"\gamma")

# ======================================================
eta = sp.Rational(1,3)
# eta = sp.Symbol(r"\eta")
# ======================================================


def ordre_RK(s, A=None, b=None, c=None):

    j = sp.symbols('j')

    if A is None: A = sp.MatrixSymbol('a', s, s)
    else:         A = sp.Matrix(A)

    if c is None: c = sp.symbols(f'c_0:{s}')

    if b is None: b = sp.symbols(f'b_0:{s}')

    display(Markdown("**Matrice de Butcher**"))
    matrice_Butcher(s, A, b, c)

    display(Markdown(f"**On a {s+1} conditions pour avoir consistance = pour être d'ordre 1**"))
    ordre_1(s, A, b, c, j)

    display(Markdown("**On doit ajouter 1 condition pour être d'ordre 2**"))
    ordre_2(s, A, b, c, j)

    display(Markdown("**On doit ajouter 2 conditions pour être d'ordre 3**"))
    ordre_3(s, A, b, c, j)

    # display(Markdown("**On doit ajouter 4 conditions pour être d'ordre 4**"))
    # ordre_4(s, A, b, c, j)

    return None


def matrice_Butcher(s, A, b, c):
    But = sp.Matrix(A)
    But = But.col_insert(0, sp.Matrix(c))
    last = [sp.Symbol(" ")]
    last.extend(b)
    But = But.row_insert(s,sp.Matrix(last).T)
    display(Math(sp.latex(sp.Matrix(But))))
    return None


def ordre_1(s, A, b, c, j):
    texte = r"\sum_{j=1}^s b_j =" + f"{sum(b).simplify()}"
    texte += r"\text{ doit être égale à }1"
    display(Math(texte))
    for i in range(s):
        somma = sp.summation(A[i,j],(j,0,s-1)).simplify()
```

```
        texte = r'\sum_{j=1}^s a_{'+str(i)+r'j}=' + sp.latex( somma )
        texte += r"\text{ doit être égale à }"+sp.latex(c[i])
        display(Math( texte ))
    return None


def ordre_2(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j=' +sp.latex(sum([b[i]*c[i] for i in range(s)]).simplify())
    texte += r"\text{ doit être égale à }\frac{1}{2}"
    display(Math(texte))
    return None


def ordre_3(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j^2='
    texte += sp.latex( sum([b[i]*c[i]**2 for i in range(s)]).simplify() )
    texte += r"\text{ doit être égale à }\frac{1}{3}"
    display(Math(texte))
    texte = r'\sum_{i,j=1}^s b_i a_{ij} c_j='
    somma = sum([b[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{6}"
    display(Math(texte))
    return None


def ordre_4(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j^3='
    texte += sp.latex( sum([b[i]*c[i]**3 for i in range(s)]).simplify() )
    texte += r"\text{ doit être égale à }\frac{1}{4}"
    display(Math(texte))
    texte = r'\sum_{i,j=1}^s b_i c_i a_{ij} c_j='
    somma = sum([b[i]*c[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{8}"
    display(Math(texte))
    texte = r'\sum_{i,j=1}^s b_i a_{ij} c_j^2='
    somma = sum([b[i]*A[i,j]*c[j]**2 for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{12}"
    display(Math(texte))
    texte = r'\sum_{i,j,k=1}^s b_i a_{ij} a_{jk} c_k='
    somma = sum([b[i]*A[i,j]*A[j,k]*c[k] for k in range(s) for j in range(s) for i in
 ↪  range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{24}"
    display(Math(texte))
    return None


# APPLICATION A NOTRE CAS

c = [0,eta,sigma]
b = [1-gamma,0,gamma]
A = [[0,0,0],[eta,0,0],[0,sigma,0]]
s = len(c)
display(Markdown(f"La méthode de Runge-Kutta est à {s} étages"))
ordre_RK(s,A,b,c)
```

La méthode de Runge-Kutta est à 3 étages

Matrice de Butcher

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & 0 \\ \sigma & 0 & \sigma & 0 \\ & 1-\gamma & 0 & \gamma \end{bmatrix}$$

On a 4 conditions pour avoir consistance = pour être d'ordre 1

$\sum_{j=1}^{s} b_j = 1$ doit être égale à 1

$\sum_{j=1}^{s} a_{0j} = 0$ doit être égale à 0

$\sum_{j=1}^{s} a_{1j} = \frac{1}{3}$ doit être égale à $\frac{1}{3}$

$\sum_{j=1}^{s} a_{2j} = \sigma$ doit être égale à σ

On doit ajouter 1 condition pour être d'ordre 2

$\sum_{j=1}^{s} b_j c_j = \gamma\sigma$ doit être égale à $\frac{1}{2}$

On doit ajouter 2 conditions pour être d'ordre 3

$\sum_{j=1}^{s} b_j c_j^2 = \gamma\sigma^2$ doit être égale à $\frac{1}{3}$

$\sum_{i,j=1}^{s} b_i a_{ij} c_j = \frac{\gamma\sigma}{3}$ doit être égale à $\frac{1}{6}$

Q2 [3 points] Étudier empiriquement l'ordre de convergence sur le problème de Cauchy

$$\begin{cases} y'(t) = \left(\cos(2\pi t) - 4t^2\right) y(t), \quad t \in [0;5] \\ y(0) = 4 \end{cases}$$

On affichera les courbes de convergence pour un schéma d'ordre 1 (à vous de donner un choix de σ et γ pour obtenir un tel schéma) et pour un schéma d'ordre 2 (ou 3 si possible).

ne pas confondre `sp.cos` ou `sp.pi` … (du module `sympy` pour le calcul de la solution exacte) et `np.cos` ou `np.pi` … (du module `numpy` pour le calcul numérique).

```
# variables globales
t0     = 0
tfinal = 5
y0     = 4

################################################
# solution exacte
################################################

phi_sym = lambda t,y: (4*sp.cos(2*sp.pi*t)-4*t**2)*y

t   = sp.Symbol('t')
y   = sp.Function('y')
edo = sp.Eq( sp.diff(y(t),t) , phi_sym(t,y(t)) )
sol = sp.dsolve(edo,y(t), ics={y(t0):y0})
display(sol)


################################################
# solution approchée
################################################
```

```
sol_exacte = sp.lambdify(t,sol.rhs,'numpy')

phi = lambda t,y: (4*np.cos(2*np.pi*t)-4*t**2)*y


def RK(tt,phi,y0,sigma,gamma,eta):
    h = tt[1]-tt[0]
    uu = np.zeros_like(tt)
    uu[0] = y0
    for i in range(len(tt)-1):
        K1 = phi( tt[i] , uu[i] )
        K2 = phi( tt[i]+h*eta , uu[i]+h*eta*K1 )
        K3 = phi( tt[i]+h*sigma , uu[i]+h*sigma*K2 )
        uu[i+1] = uu[i]+h*((1-gamma)*K1+gamma*K3)
    return uu

##############################################
# ordre
##############################################
H = []
err_1, err_2, err_3 = [], [], []

plt.figure(figsize=(16,10))
ax11 = plt.subplot(3,2,1)
ax12 = plt.subplot(3,2,2)
ax21 = plt.subplot(3,2,3)
ax22 = plt.subplot(3,2,4)
ax31 = plt.subplot(3,2,5)
ax32 = plt.subplot(3,2,6)

ax11.set_title("Schéma d'ordre 1")
ax21.set_title("Schéma d'ordre 2")
ax31.set_title("Schéma d'ordre 3")

N = 0
for k in range(6):
    N += 100
    tt = np.linspace(t0, tfinal, N + 1)
    H.append( tt[1] - tt[0] )
    yy = sol_exacte(tt)

    sigma, gamma, eta = 1/2, 1/2, 1/3 # cas ordre 1
    uu_1 = RK(tt,phi,y0,sigma,gamma,eta)
    err_1.append( np.linalg.norm(uu_1-yy,np.inf) )
    ax11.plot(tt,uu_1,label=f'Approchée avec N={N}')

    sigma, gamma, eta = 1/4, 2, 1/6 # cas ordre 2
    uu_2 = RK(tt,phi,y0,sigma,gamma,eta)
    err_2.append( np.linalg.norm(uu_2-yy,np.inf) )
    ax21.plot(tt,uu_2,label=f'Approchée avec N={N}')

    sigma, gamma, eta = 2/3, 3/4, 1/3 # cas ordre 3
    uu_3 = RK(tt,phi,y0,sigma,gamma,eta)
    err_3.append( np.linalg.norm(uu_3-yy,np.inf) )
    ax31.plot(tt,uu_3,label=f'Approchée avec N={N}')


ax11.plot(tt,yy,label='Exacte')
plt.xlabel('$t$')
plt.ylabel('$y$')
ax11.grid(True)
ax11.legend()
```

```
ax12.plot( np.log(H), np.log(err_1), 'r-o')
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
ax12.set_title(f'RK : ordre = {np.polyfit(np.log(H),np.log(err_1),1)[0]:1.2f}')
ax12.grid(True)

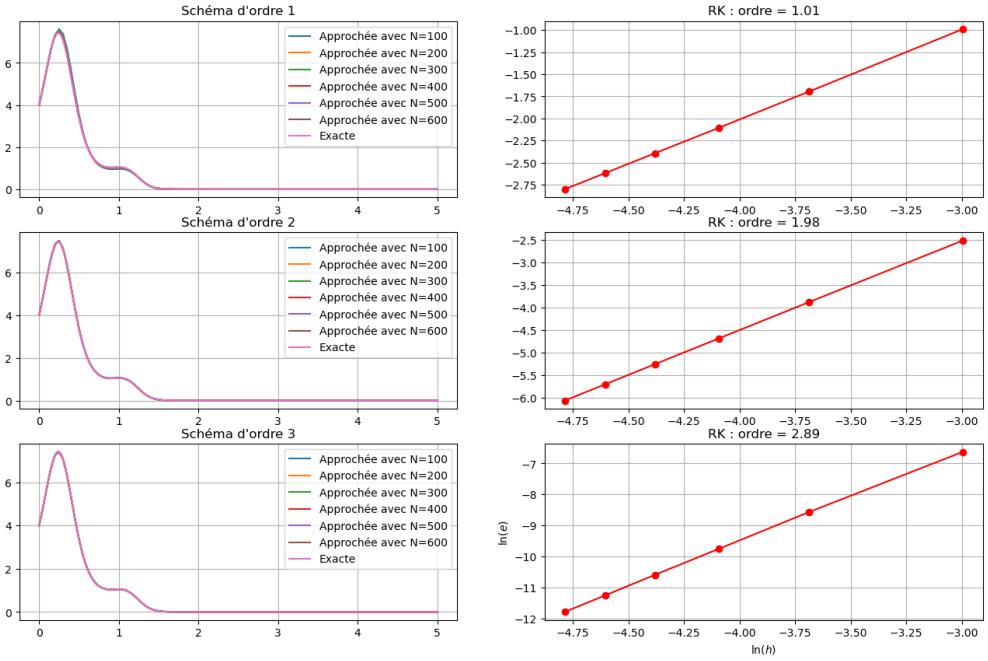
ax21.plot(tt,yy,label='Exacte')
plt.xlabel('$t$')
plt.ylabel('$y$')
ax21.grid(True)
ax21.legend()

ax22.plot( np.log(H), np.log(err_2), 'r-o')
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
ax22.set_title(f'RK : ordre = {np.polyfit(np.log(H),np.log(err_2),1)[0]:1.2f}')
ax22.grid(True)


ax31.plot(tt,yy,label='Exacte')
plt.xlabel('$t$')
plt.ylabel('$y$')
ax31.grid(True)
ax31.legend()

ax32.plot( np.log(H), np.log(err_3), 'r-o')
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
ax32.set_title(f'RK : ordre = {np.polyfit(np.log(H),np.log(err_3),1)[0]:1.2f}')
ax32.grid(True)
```

$y(t) = 4e^{-\frac{4t^3}{3}+\frac{2\sin(2\pi t)}{\pi}}$



Q3 [2 points] Étudier théoriquement la A-stabilité du schéma d'ordre maximal (dans la correction ci-dessous $\eta = \frac{1}{3}$ et donc l'ordre maximal est 3).

La méthode appliquée à l'EDO $y'(t)=-\beta y(t)$ avec $\beta>0$ s'écrit :

$$\begin{cases} u_0=y_0 \\ K_1=-\beta u_n \\ K_2=-\beta\left(u_n+\frac{h}{3}K_1\right) \rightsquigarrow K_2=\left(1-\frac{\beta h}{3}\right)K_1 \\ K_3=-\beta\left(u_n+\frac{2h}{3}K_2\right) \rightsquigarrow K_3=\left(1-\frac{2\beta h}{3}\left(1-\frac{\beta h}{3}\right)\right)K_1 \\ u_{n+1}=u_n+\frac{h}{4}\left(K_1+4K_3\right) \end{cases} \qquad n=0,1,\dots N-1$$

L'étude ci-dessous montre que le schéma est A-stable ssi $h<\frac{\vartheta}{\beta}$ avec $\vartheta\approx 2.7$.

```
u_n = sp.Symbol('u_n')
h = sp.Symbol('h')
K1 = sp.Symbol('K1')
K2 = sp.Symbol('K2')
beta = sp.Symbol(r'\beta')
sigma = sp.Symbol(r"\sigma")
gamma = sp.Symbol(r"\gamma")
eta = sp.Symbol(r"\eta")
g = lambda y : -beta*y
K1 = g(u_n)
K2 = g(u_n+h*eta*K1)
K3 = g(u_n+h*sigma*K2)
RHS = u_n+h*((1-gamma)*K1+gamma*K3)
RHS = RHS.factor()
texte = 'u_{n+1}=' + sp.latex(RHS)
display(Math(texte))

display(Markdown(r"Notons $x=h\beta>0$. On trouve donc"))
x = sp.Symbol('x',positive=True)
RHS = RHS.subs(h*beta,x).simplify()
texte = 'u_{n+1}=' + sp.latex(RHS)
display(Math(texte))


texte = r"La méthode est A-stable ssi $|q(x)|<1$ avec $q(x)=\dfrac{u_{n+1}}{u_n}="
texte += sp.latex(RHS/u_n) + "$"
display(Markdown(texte))

# ======== On remplace les paramètres par les valeurs optimales ==========
RHS = RHS.subs(gamma,sp.Rational(4,3)).subs(sigma,sp.Rational(2,3)).subs(eta,sp.Rational(1,3)).simplify()
q = sp.apart(RHS/u_n)
# ========================================================================

texte = r"Dans le cas optimal on a $\sigma=\dfrac{2}{3}$ et $\gamma=\dfrac{4}{3}$ ainsi
 ↪  $$q(x)=\dfrac{u_{n+1}}{u_n}="
texte += sp.latex(RHS/u_n) + "$$"
display(Markdown(texte))

display(Markdown(r"Étudions la fonction $q(x)$ pour $x>0$:"))

sp.solve( [ -1<RHS/u_n , RHS/u_n<1 ] )

display(Markdown(f"$q(0)={q.subs(x,0)}$"))

ell = sp.Limit(q,x,sp.oo)
display(Math(f"{sp.latex(ell)}={sp.latex(ell.doit())}"))

dq = sp.diff(q,x).simplify()
display(Markdown(f"$\\displaystyle q'(x)={sp.latex(dq)}$"))

sol=sp.solveset(dq,domain=sp.S.Reals)
display(Markdown(f"$q'(x)<0$ pour tout $x>0$ et l'on a"))
```

```
sol = sp.solveset(q+1,domain=sp.S.Reals)
texte = r"q(x)=-1 \iff x= " + sp.latex(sol) + r"\approx" + sp.latex(sol.evalf())
display( Math( texte ) )

# p1 = sp.plot(q,(x,0,5),ylim=(-2,2),xlabel=r'$x$',ylabel=r'$q(x)$',show=False,grid=True)
# p2 = sp.plot(1,(x,0,5),ylim=(-2,2),show=False,line_color='r')
# p3 = sp.plot(-1,(x,0,5),ylim=(-2,2),show=False,line_color='r')
# p1.extend(p2)
# p1.extend(p3)
# p1.show()

plt.figure(figsize=(12,6))
q = sp.lambdify(x,RHS/u_n,'numpy')
xx = np.linspace(0,5,100)
plt.plot(xx,q(xx),label=r'$q(x)$')
plt.hlines(1,0,5,'r')
plt.hlines(-1,0,5,'r')
plt.axis([0,5,-2,2])
plt.grid();
```

$$u_{n+1} = -u_n\left(\beta^3\eta\gamma\sigma h^3 - \beta^2\gamma\sigma h^2 + \beta h - 1\right)$$

Notons $x = h\beta > 0$. On trouve donc

$$u_{n+1} = u_n\left(-\eta\gamma\sigma x^3 + \gamma\sigma x^2 - x + 1\right)$$

La méthode est A-stable ssi $|q(x)| < 1$ avec $q(x) = \dfrac{u_{n+1}}{u_n} = -\eta\gamma\sigma x^3 + \gamma\sigma x^2 - x + 1$

Dans le cas optimal on a $\sigma = \dfrac{2}{3}$ et $\gamma = \dfrac{4}{3}$ ainsi

$$q(x) = \frac{u_{n+1}}{u_n} = -\frac{8x^3}{27} + \frac{8x^2}{9} - x + 1$$

Étudions la fonction $q(x)$ pour $x > 0$:

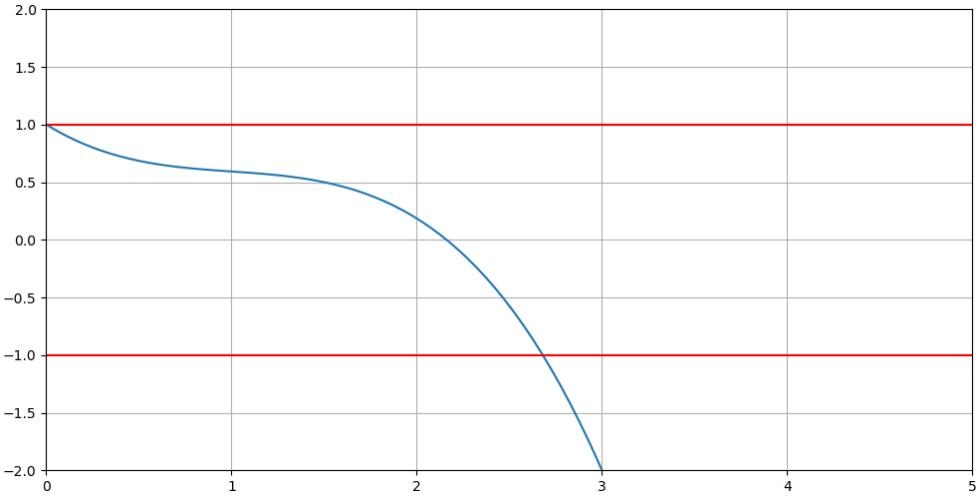
$q(0) = 1$

$$\lim_{x\to\infty}\left(-\frac{8x^3}{27} + \frac{8x^2}{9} - x + 1\right) = -\infty$$

$$q'(x) = -\frac{8x^2}{9} + \frac{16x}{9} - 1$$

$q'(x) < 0$ pour tout $x > 0$ et l'on a

$$q(x) = -1 \iff x = \left\{-\frac{1}{24\sqrt[3]{\frac{43}{432} + \frac{\sqrt{822}}{288}}} + 1 + 3\sqrt[3]{\frac{43}{432} + \frac{\sqrt{822}}{288}}\right\} \approx \{2.68038081801272\}$$


2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
0
1
2
3
4
5

Chapitre 29.

2023 Contrôle Terminal session 2

2h30, deposer le notebook dans Moodle

```
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp
from scipy.optimize import fsolve
from scipy.integrate import solve_ivp, odeint
from IPython.display import display, Markdown, Math
```

29.1. EXERCICE 1 : choix d'un schéma

Soit le problème de Cauchy

$$\begin{cases} y'(t) = 15y(t) + \frac{1-30t}{2\sqrt{t}}, & t \in [1;2] \\ y(1) = 1+\varepsilon \end{cases}$$

Calculer la solution exacte pour $\varepsilon \geq 0$.
Soit $\varepsilon = 0$. Approcher la solution du problème avec un des schémas classiques (EE, EI, EM, CN, Heun) qu'on choisira en fonction de la nature du problème de Cauchy (on justifiera ce choix). Afficher la solution exacte et la solution approchée sur le même répère avec $N = 100, 200, 300, 400, 500, 600$. Justifier le comportement de la solution approchée lorsque N est petit.

ne pas confondre `sp.sqrt` (du module `sympy` pour le calcul de la solution exacte) et `np.sqrt` (du module `numpy` pour le calcul numérique).

On calcule la solution exacte. On remarque que c'est un problème de Cauchy numériquement mal posé : si $\varepsilon = 0$ la fonction croît comme $\sqrt{t}$, si $\varepsilon > 0$ comme une exponentielle. On doit donc choisir le schéma le plus précis parmi les schémas proposés, à savoir le schéma de Heun ou le schéma d'Euler Modifié ou de Crank-Nicolson qui sont d'ordre 2.

```
# variables globales
t0     = 1
tfinal = 2

phi = lambda t,y: 15*y+(1-30*t)/(2*np.sqrt(t))


##############################################
# solution exacte
##############################################

phi_sym = lambda t,y: 15*y+(1-30*t)/(2*sp.sqrt(t))
eps   = sp.Symbol(r'\varepsilon', positive=True)
y0 = 1+eps

t   = sp.Symbol('t')
y   = sp.Function('y')
edo = sp.Eq( sp.diff(y(t),t) , phi_sym(t,y(t)) )
solpar = sp.dsolve(edo,y(t), ics={y(t0):y0})
```

```
display(solpar)


################################################
# solution approchée
################################################
y0 = 1
sol_exacte = sp.lambdify(t,solpar.rhs.subs(eps,0),'numpy')

def EE(tt,phi,y0):
    h = tt[1]-tt[0]
    uu = np.zeros(len(tt))
    uu[0] = y0
    for i in range(N):
        uu[i+1] = uu[i]+h*phi(tt[i],uu[i])
    return uu

def EI(tt,phi,y0):
    h = tt[1]-tt[0]
    uu = np.zeros(len(tt))
    uu[0] = y0
    for i in range(N):
        temp = fsolve(lambda x: -x+uu[i]+h*phi(tt[i+1],x), uu[i])
        uu[i+1] = temp[0]
    return uu

def CN(tt,phi,y0):
    h = tt[1]-tt[0]
    uu = np.zeros(len(tt))
    uu[0] = y0
    for i in range(len(tt)-1):
        temp = fsolve(lambda x: -x+uu[i]+h/2*(phi(tt[i],uu[i])+phi(tt[i+1],x)), uu[i])
        uu[i+1] = temp[0]
    return uu

def Heun(tt,phi,y0):
    h = tt[1]-tt[0]
    uu = np.zeros(len(tt))
    uu[0] = y0
    for i in range(len(tt)-1):
        uu[i+1] = uu[i]+h/2*(phi(tt[i],uu[i])+phi(tt[i+1],uu[i]+h*phi(tt[i],uu[i])))
    return uu

def EM(tt,phi,y0):
    h = tt[1]-tt[0]
    uu = np.zeros(len(tt))
    uu[0] = y0
    for i in range(len(tt)-1):
        uu[i+1] = uu[i]+h*phi(tt[i]+h/2,uu[i]+h/2*phi(tt[i],uu[i]))
    return uu


################################################
# ordre
################################################

for schema in ["CN","Heun","EM","EE","EI"]:

    plt.figure(figsize=(16,5))
    ax1 = plt.subplot(1,2,1)
    ax2 = plt.subplot(1,2,2)

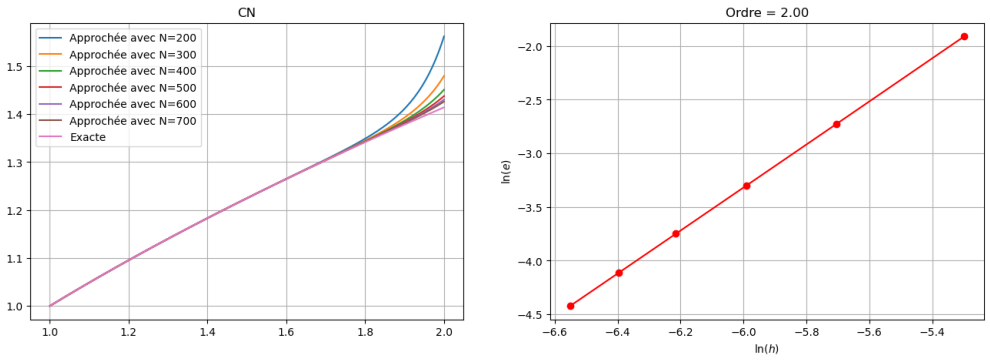
    H = []
```

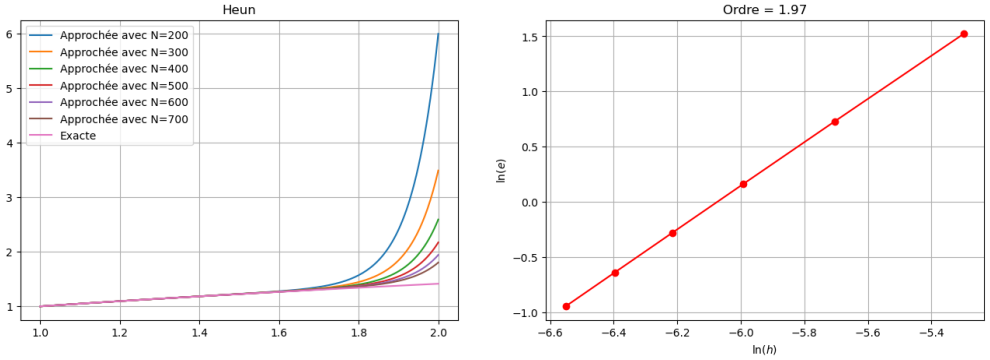
```
    err = []
    N = 100
    for k in range(6):
        N += 100
        tt = np.linspace(t0, tfinal, N + 1)
        H.append( tt[1] - tt[0] )
        yy = sol_exacte(tt)
        uu = eval(schema)(tt,phi,y0)
        err.append( np.linalg.norm(uu-yy,np.inf) )
        ax1.plot(tt,uu,label=f'Approchée avec N={N}')

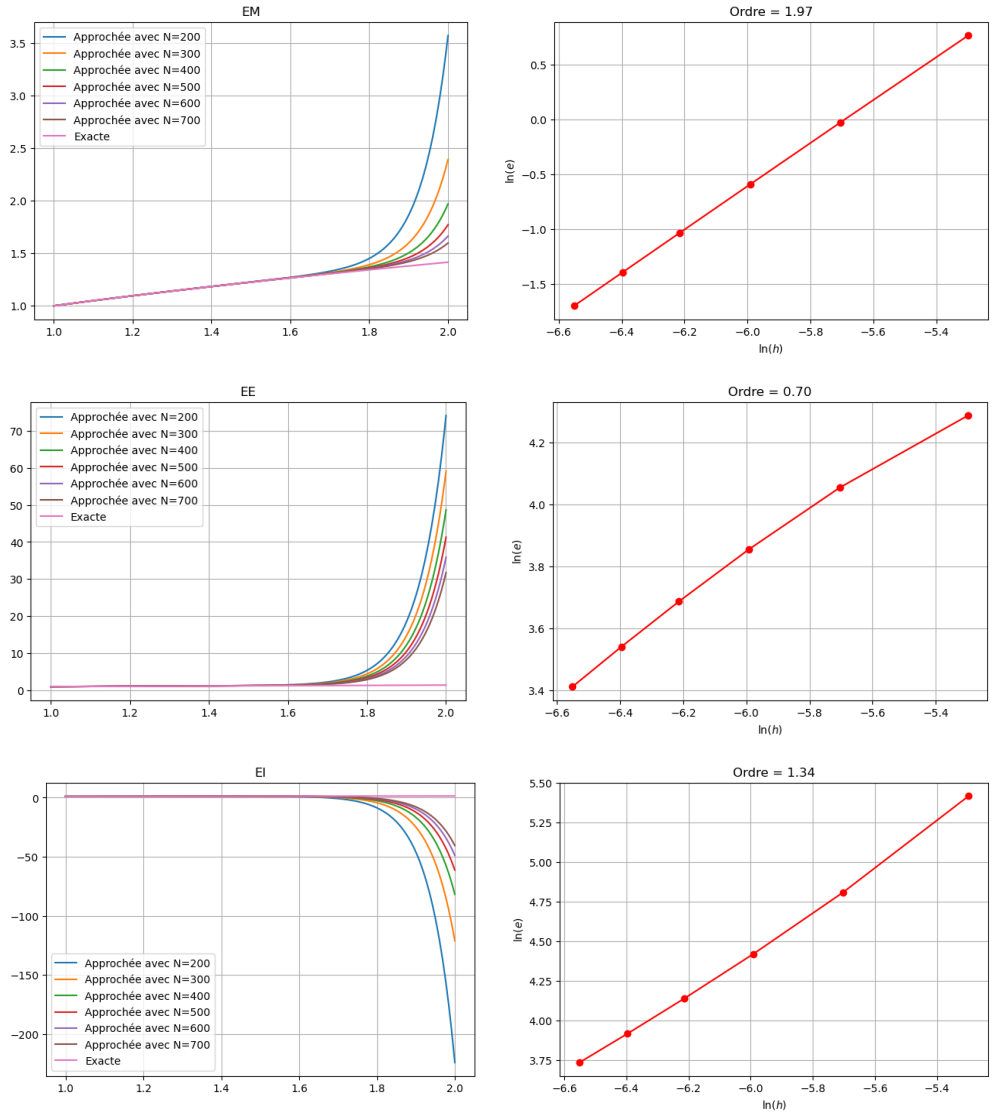
    ax1.plot(tt,yy,label='Exacte')
    plt.xlabel('$t$')
    plt.ylabel('$y$')
    ax1.grid(True)
    ax1.legend()
    ax1.set_title(f'{schema}')

    ax2.plot( np.log(H), np.log(err), 'r-o')
    plt.xlabel(r'$\ln(h)$')
    plt.ylabel(r'$\ln(e)$')
    plt.title(f'Ordre = {np.polyfit(np.log(H),np.log(err),1)[0]:1.2f}')
    ax2.grid(True)
```

$$y(t) = \frac{\varepsilon e^{15t}}{e^{15}} + \sqrt{t}$$







Comparons la valeurs de $y(2)$ obtenue avec les differents schémas :

```
N = 600
tt = np.linspace(t0, tfinal, N + 1)
y_ex = sol_exacte(tt)[-1]
u_cn = CN(tt,phi,y0)[-1]
u_he = Heun(tt,phi,y0)[-1]
u_em = EM(tt,phi,y0)[-1]
u_ee = EE(tt,phi,y0)[-1]
u_ei = EI(tt,phi,y0)[-1]

from tabulate import tabulate
T = []
T.append(["y(2)", y_ex])
T.append(["CN", u_cn])
T.append(["EM", u_em])
T.append(["Heun", u_he])
T.append(["EE", u_ee])
T.append(["EI", u_ei])
tabulate(T,tablefmt="html",floatfmt=".2f")
```

y(2)	1.41
CN	1.43

EM	1.66
Heun	1.94
EE	35.89
EI	-48.81

29.2. EXERCICE 2 : étude d'un schéma Runge-Kutta

Considérons le schéma dont la matrice de Butcher est donnée ci-dessous :

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{4} & \frac{1}{4} & 0 \\ 1 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \hline & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{array}$$

Q1 [1 point] Écrire la suite définie par réccurence associée à ce schéma.

$$\begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n, u_n\right) \\ K_2 = \varphi\left(t_n + \frac{h}{2}, u_n + \frac{h}{4}\left(K_1 + K_2\right)\right) \\ K_3 = \varphi\left(t_{n+1}, u_n + \frac{h}{3}\left(K_1 + K_2 + K_3\right)\right) \\ u_{n+1} = u_n + \frac{h}{3}\left(K_1 + K_2 + K_3\right) \qquad\qquad n = 0, 1, \ldots N-1 \end{cases}$$

Notons que $c_s = 1$ et $a_{sj} = b_j$ pour tout $j = 1, \ldots, s$ ainsi $K_s = \varphi(t_{n+1}, u_{n+1})$.

Q2 [5 points] Étudier empiriquement l'ordre de convergence sur le problème de Cauchy

$$\begin{cases} y'(t) = y(t) - (1+t^2)e^{-t}, \quad t \in [1;2] \\ y(1) = 2/e \end{cases}$$

ne pas confondre `sp.exp` (du module `sympy` pour le calcul de la solution exacte) et `np.exp` (du module `numpy` pour le calcul numérique).

```
# variables globales
t0     = 1
tfinal = 2

phi = lambda t,y: y*(2*t-t**2-1)/(1+t**2)


################################################
# solution exacte
################################################
y0     = 2/sp.E

t   = sp.Symbol('t')
y   = sp.Function('y')
edo = sp.Eq( sp.diff(y(t),t) , phi(t,y(t)) )
solpar = sp.dsolve(edo,y(t), ics={y(t0):y0})
display(solpar)

sol_exacte = sp.lambdify(t,solpar.rhs,'numpy')

################################################
# solution approchée
################################################
```

```
y0     = 2/np.exp(1)

def RK(tt,phi,y0):
    h = tt[1]-tt[0]
    uu = np.zeros_like(tt)
    uu[0] = y0
    for i in range(len(tt)-1):
        sys = lambda X : [ -X[0]+phi( tt[i] , uu[i]                                    ) ,
                           -X[1]+phi( tt[i]+h/2 , uu[i]+h/4*(X[0]+X[1])                ),
                           -X[2]+phi( tt[i+1]   ,uu[i]+h/3*(X[0]+X[1]+X[2])       )
                           ]
        K_start = phi(tt[i],uu[i])
        K1, K2, K3 = fsolve( sys , [K_start,K_start,K_start ] )
        uu[i+1] = uu[i]+h/3*(K1+K2+K3)
    return uu


##############################################
# ordre
##############################################
H = []
err = []
N = 10

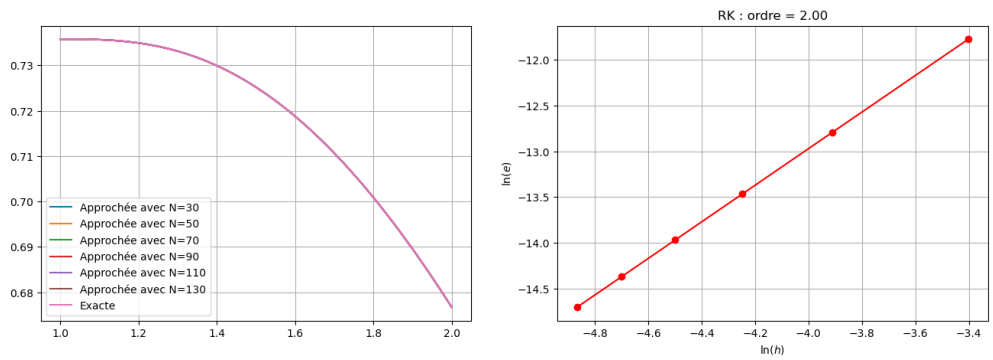
plt.figure(figsize=(16,5))
ax1 = plt.subplot(1,2,1)
ax2 = plt.subplot(1,2,2)

for k in range(6):
    N += 20
    tt = np.linspace(t0, tfinal, N + 1)
    H.append( tt[1] - tt[0] )
    yy = sol_exacte(tt)
    uu = RK(tt,phi,y0)
    err.append( np.linalg.norm(uu-yy,np.inf) )
    ax1.plot(tt,uu,label=f'Approchée avec N={N}')

ax1.plot(tt,yy,label='Exacte')
plt.xlabel('$t$')
plt.ylabel('$y$')
ax1.grid(True)
ax1.legend()

ax2.plot( np.log(H), np.log(err), 'r-o')
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
plt.title(f'RK : ordre = {np.polyfit(np.log(H),np.log(err),1)[0]:1.2f}')
ax2.grid(True)
```

$y(t) = (t^2 + 1)\, e^{-t}$



Q3 [3 points] Étudier théoriquement l'ordre du schéma.

Soit ω l'ordre de la méthode.

C'est un schéma (semi-)implicite à 3 étages ($s = 3$) donc $\omega \leq 2s = 6$

```
prlat= lambda *args: display(Math(''.join( sp.latex(arg) for arg in args )))

def ordre_RK(s, A=None, b=None, c=None):

    j = sp.symbols('j')

    if A is None:
        A = sp.MatrixSymbol('a', s, s)
    else:
        A = sp.Matrix(A)
    if c is None:
        c = sp.symbols(f'c_0:{s}')
    if b is None:
        b = sp.symbols(f'b_0:{s}')


    display(Markdown("**Matrice de Butcher**"))
    matrice_Butcher(s, A, b, c)

    display(Markdown(f"**On a {s+1} conditions pour avoir consistance = pour être d'ordre 1**"))
    ordre_1(s, A, b, c, j)

    display(Markdown("**On doit ajouter 1 condition pour être d'ordre 2**"))
    ordre_2(s, A, b, c, j)

    display(Markdown("**On doit ajouter 2 conditions pour être d'ordre 3**"))
    ordre_3(s, A, b, c, j)

    # display(Markdown("**On doit ajouter 4 conditions pour être d'ordre 4**"))
    # ordre_4(s, A, b, c, j)

    return None


def matrice_Butcher(s, A, b, c):
    But = sp.Matrix(A)
    But = But.col_insert(0, sp.Matrix(c))
    last = [sp.Symbol(" ")]
    last.extend(b)
    But = But.row_insert(s,sp.Matrix(last).T)
    display(Math(sp.latex(sp.Matrix(But))))
    return None


def ordre_1(s, A, b, c, j):
    texte = r"\sum_{j=1}^s b_j =" + f"{sum(b).simplify()}"
    texte += r"\text{ doit être égale à }1"
    display(Math(texte))
    for i in range(s):
        somma = sp.summation(A[i,j],(j,0,s-1)).simplify()
        texte = r'\sum_{j=1}^s a_{'+str(i)+r'j}=' + sp.latex( somma )
        texte += r"\text{ doit être égale à }"+sp.latex(c[i])
        display(Math( texte ))
    return None


def ordre_2(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j=' +sp.latex(sum([b[i]*c[i] for i in range(s)]).simplify())
```

```
    texte += r"\text{ doit être égale à }\frac{1}{2}"
    display(Math(texte))
    return None


def ordre_3(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j^2='
    texte += sp.latex( sum([b[i]*c[i]**2 for i in range(s)]).simplify() )
    texte += r"\text{ doit être égale à }\frac{1}{3}"
    display(Math(texte))
    texte = r'\sum_{i,j=1}^s b_i a_{ij} c_j='
    somma = sum([b[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{6}"
    display(Math(texte))
    return None


def ordre_4(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j^3='
    texte += sp.latex( sum([b[i]*c[i]**3 for i in range(s)]).simplify() )
    texte += r"\text{ doit être égale à }\frac{1}{4}"
    display(Math(texte))
    texte = r'\sum_{i,j=1}^s b_i c_i a_{ij} c_j='
    somma = sum([b[i]*c[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{8}"
    display(Math(texte))
    texte = r'\sum_{i,j=1}^s b_i a_{ij} c_j^2='
    somma = sum([b[i]*A[i,j]*c[j]**2 for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{12}"
    display(Math(texte))
    texte = r'\sum_{i,j,k=1}^s b_i a_{ij} a_{jk} c_k='
    somma = sum([b[i]*A[i,j]*A[j,k]*c[k] for k in range(s) for j in range(s) for i in
↪   range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{24}"
    display(Math(texte))
    return None


# APPLICATION A NOTRE CAS
c = [0,sp.Rational(1,2),1]
b = [sp.Rational(1,3),sp.Rational(1,3),sp.Rational(1,3)]
A = [[0,0,0],[sp.Rational(1,4),sp.Rational(1,4),0],[sp.Rational(1,3),sp.Rational(1,3),sp.Rational(1,3)]]
s = len(c)

display(Markdown(f"La méthode de Runge-Kutta est à {s} étages"))
ordre_RK(s,A,b,c)
```

La méthode de Runge-Kutta est à 3 étages

Matrice de Butcher

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{4} & \frac{1}{4} & 0 \\ 1 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{bmatrix}$$

On a 4 conditions pour avoir consistance = pour être d'ordre 1

$\sum_{j=1}^{s} b_j = 1$ doit être égale à 1

$\sum_{j=1}^{s} a_{0j} = 0$ doit être égale à 0

$\sum_{j=1}^{s} a_{1j} = \frac{1}{2}$ doit être égale à $\frac{1}{2}$

$\sum_{j=1}^{s} a_{2j} = 1$ doit être égale à 1

On doit ajouter 1 condition pour être d'ordre 2

$\sum_{j=1}^{s} b_j c_j = \frac{1}{2}$ doit être égale à $\frac{1}{2}$

On doit ajouter 2 conditions pour être d'ordre 3

$\sum_{j=1}^{s} b_j c_j^2 = \frac{5}{12}$ doit être égale à $\frac{1}{3}$

$\sum_{i,j=1}^{s} b_i a_{ij} c_j = \frac{5}{24}$ doit être égale à $\frac{1}{6}$

D'après ces résultats, le schéma est d'ordre 2.

Q3 [4 points] Étudier théoriquement la A-stabilité du schéma.

La méthode appliquée à l'EDO $y'(t) = -\beta y(t)$ s'écrit :

$$\begin{cases} u_0 = y_0 \\ K_1 = -\beta\left(u_n\right) \\ K_2 = -\beta\left(u_n + \frac{h}{4}K_1 + \frac{h}{4}K_2\right) \\ K_3 = -\beta\left(u_n + \frac{h}{3}K_1 + \frac{h}{3}K_2 + \frac{h}{3}K_3\right) \\ u_{n+1} = u_n + \frac{h}{3}\left(K_1 + K_2 + K_3\right) \qquad n = 0, 1, \ldots N-1 \end{cases}$$

```
u_n, h, beta, K1, K2, K3 = sp.symbols(r'u_n h \beta K_1 K_2 K_3')
g = lambda y : -beta*y
eq1 = sp.Eq( K1 , g(u_n))
eq2 = sp.Eq( K2 , g(u_n+h/4*K1+h/4*K2) )
eq3 = sp.Eq( K3 , g(u_n+h/3*K1+h/3*K2+h/3*K3) )
sol = sp.solve([eq1,eq2,eq3],[K1,K2,K3])
for k in sol:
    texte = f"{k}=" + sp.latex(sol[k])
    display(Math(texte))
RHS = u_n+h/3*(K1+K2+K3)
RHS = RHS.subs(sol).simplify()
texte = 'u_{n+1}=' + sp.latex(RHS)
display(Math(texte))
display(Markdown(r"Notons $x=h\beta>0$. On trouve donc"))
x = sp.Symbol('x',positive=True)
RHS = RHS.subs(h*beta,x).simplify()
texte = 'u_{n+1}=' + sp.latex(RHS)
display(Math(texte))
```

$K_1 = -\beta u_n$

$K_2 = \frac{\beta^2 h u_n - 4\beta u_n}{\beta h + 4}$

$K_3 = \frac{5\beta^2 h u_n - 12\beta u_n}{\beta^2 h^2 + 7\beta h + 12}$

$u_{n+1} = \frac{u_n\left(-5\beta h + 12\right)}{\beta^2 h^2 + 7\beta h + 12}$

Notons $x = h\beta > 0$. On trouve donc

$$u_{n+1} = \frac{u_n\,(12-5x)}{x^2+7x+12}$$

Par induction on obtient

$$u_n = \left(\frac{12-5x}{x^2+7x+12}\right)^n u_0.$$

Par conséquent, $\lim\limits_{n\to+\infty} u_n = 0$ si et seulement si

$$\left|\frac{12-5x}{x^2+7x+12}\right| < 1.$$

Étudions la fonction $x \mapsto \frac{12-5x}{x^2+7x+12}$ pour $x > 0$:

```
q = sp.apart(RHS/u_n)
prlat( r'q(x)=' , RHS/u_n  )
prlat( r'q(0)=' , q.subs(x,0) )

aaa = sp.Limit(q,x,sp.oo)
prlat( aaa, "=", aaa.doit()  )

prlat( r"q(x)=0", " ssi x =", sp.solve(q)[0] )


dq = sp.diff(q,x).apart()
prlat( r"q'(x)=" , dq )
prlat( r"q'(x)=0", " ssi x =", sp.solve(dq)[0] )

q_numpy = sp.lambdify(x,q,'numpy')
plt.figure(figsize=(10,5))
xx = np.linspace(0,50,100)
yy = q_numpy(xx)
plt.plot(xx,yy)
plt.plot([xx[0],xx[-1]],[-1,-1])
plt.plot([xx[0],xx[-1]],[1,1])
plt.grid();
```

q(x)=$\dfrac{12-5x}{x^2+7x+12}$

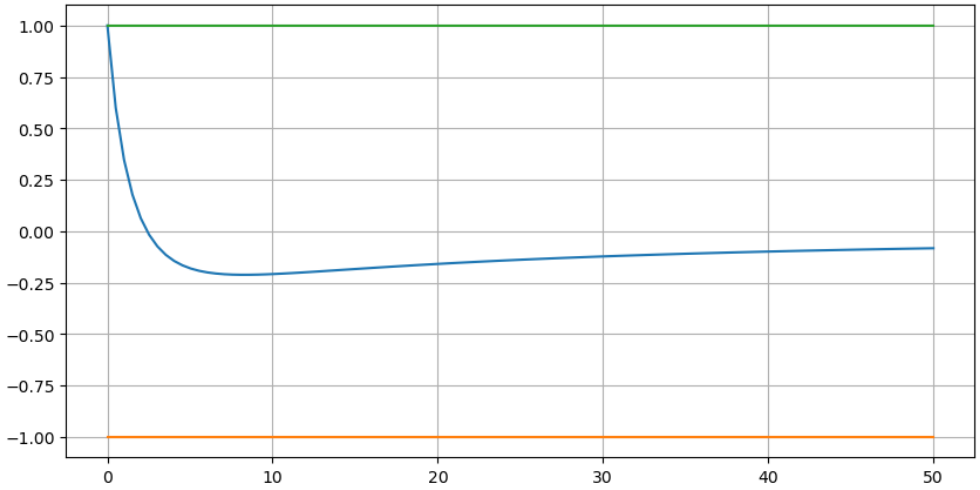
q(0)=1

$\lim\limits_{x\to\infty}\left(-\dfrac{32}{x+4}+\dfrac{27}{x+3}\right)$=0

q(x)=0 ssi x =$\dfrac{12}{5}$

q'(x)=$\dfrac{32}{(x+4)^2}-\dfrac{27}{(x+3)^2}$

q'(x)=0 ssi x =$\dfrac{12}{5}+\dfrac{12\sqrt{6}}{5}$



Elle est décroissante puis croissante. Elle atteint son minimum en $x_0 = \frac{12}{5}(1+\sqrt{6})$ et $-1 < q(x_0) < 0$. Pour $x > x_0$ elle est croissante et $q(x) < 0$.

La condition de stabilité est donc satisfaite pour tout x donc pour tout h.
Notons que $q(x) > 0$ ssi $x < \frac{12}{5}$.

> **Q4 [3 points]** Considérons le problème de Cauchy $y'(t) = -50y(t)$ pour $t \in]0,1]$ et $y(0) = 1$.
> Vérifier empiriquement qu'avec $h > \frac{12}{250}$ le schéma est stable mais non monotone, tandis que pour $h < \frac{12}{250}$ le schéma est monotone.

```
# variables globales
t0     = 0
tfinal = 1
y0     = 1
beta      = 50

phi = lambda t,y: -beta*y

################################################
# solution exacte
################################################
sol_exacte = lambda t : y0*np.exp(-beta*t)

################################################
# solution approchée
################################################
def RK(tt,phi,y0):
    h = tt[1]-tt[0]
    uu = np.zeros_like(tt)
    uu[0] = y0
    for i in range(len(tt)-1):
        K1 = phi(tt[i],uu[i])
        sys = lambda X : [-X[0]+phi(tt[i]+h/2,uu[i]+h/4*(K1+X[0])) ,
 ↪  -X[1]+phi(tt[i+1],uu[i]+h/3*(K1+X[0]+X[1]))]
        K2, K3 = fsolve( sys , [K1,K1] )
        uu[i+1] = uu[i]+h/3*(K1+K2+K3)
    return uu

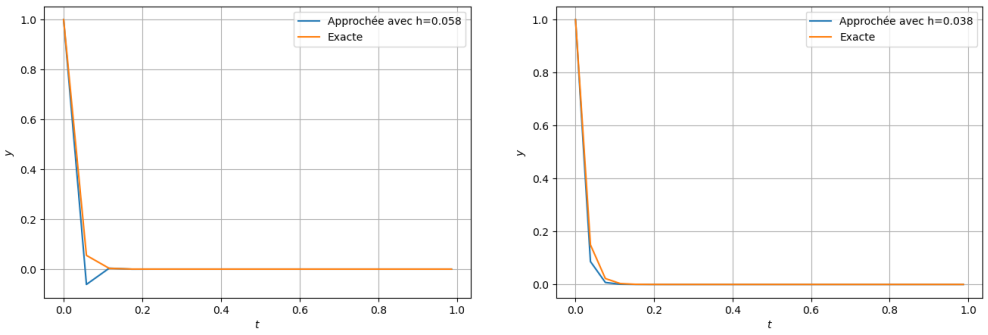
################################################
# ordre
################################################

plt.figure(figsize=(16,5))

plt.subplot(1,2,1)
```

```
tt = np.arange(t0, tfinal, 12/250 + 0.01 )
h  = tt[1] - tt[0]
yy = np.array(sol_exacte(tt))
uu = RK(tt, phi, y0)
err = np.linalg.norm(uu-yy,np.inf)
plt.plot(tt,uu,label=f'Approchée avec h={h}')
plt.plot(tt,yy,label='Exacte')
plt.xlabel('$t$')
plt.ylabel('$y$')
plt.grid(True)
plt.legend();

plt.subplot(1,2,2)
tt = np.arange(t0, tfinal, 12/250 - 0.01 )
h  = tt[1] - tt[0]
yy = np.array(sol_exacte(tt))
uu = RK(tt, phi, y0)
err = np.linalg.norm(uu-yy,np.inf)
plt.plot(tt,uu,label=f'Approchée avec h={h}')
plt.plot(tt,yy,label='Exacte')
plt.xlabel('$t$')
plt.ylabel('$y$')
plt.grid(True)
plt.legend();
```



Q_Bonus [1 point] Montrer que ce schéma de type Runge-Kutta peut se réécrire sous la forme :

$$\text{(CN-BDF}_2\text{)} \qquad \begin{cases} u_0 = y_0, \\ \tilde{u} = u_n + \frac{h}{4}\left(\varphi(t_n, u_n) + \varphi\left(t_n + \frac{h}{2}, \tilde{u}\right)\right) \\ u_{n+1} = \frac{1}{3}\left(-u_n + 4\tilde{u} + h\varphi(t_{n+1}, u_{n+1})\right) \quad n = 0, 1, 2, \dots N-1. \end{cases}$$

Autrement dit, u_{n+1} est obtenu par une itération du schéma BDF_2 sur l'intervalle $[t_n; t_{n+1}]$. Ce schéma fait intervenir l'évaluation de u en $t_n + \frac{h}{2}$. Cette valeur est alors approchée par la méthode de Crank-Nicolson sur le demi-interval $\left[t_n; t_n + \frac{h}{2}\right]$.

Les deux équations

$$\begin{cases} K_3 = \varphi\left(t_{n+1}, u_n + \frac{h}{3}\left(K_1 + K_2 + K_3\right)\right) \\ u_{n+1} = u_n + \frac{h}{3}\left(K_1 + K_2 + K_3\right) \end{cases}$$

impliquent

$$K_3 = \varphi\left(t_{n+1}, u_{n+1}\right)$$

ainsi, comme $K_1 = \varphi\left(t_n, u_n\right)$, on a

$$u_{n+1} = u_n + \frac{h}{3}\left(\varphi\left(t_n, u_n\right) + K_2 + \varphi\left(t_{n+1}, u_{n+1}\right)\right).$$

Si on note

$$A = u_n + \frac{h}{4}\left(K_1 + K_2\right),$$

alors

$$(A-u_n)\frac{4}{h}-K_1=K_2=\varphi\left(t_n+\frac{h}{2},u_n+\frac{h}{4}\left(K_1+K_2\right)\right)=\varphi\left(t_n+\frac{h}{2},A\right)$$

soit encore

$$A=u_n+\frac{h}{4}\left(K_1+\varphi\left(t_n+\frac{h}{2},A\right)\right)$$

et enfin

$$A=u_n+\frac{h}{4}\left(\varphi(t_n,u_n)+\varphi\left(t_n+\frac{h}{2},A\right)\right)$$

On conclut alors que

$$u_{n+1}=u_n+\frac{h}{3}\left((A-u_n)\frac{4}{h}+\varphi\left(t_{n+1},u_{n+1}\right)\right)=\frac{1}{3}\left(3u_n+4(A-u_n)+h\varphi\left(t_{n+1},u_{n+1}\right)\right)=\frac{1}{3}\left(-u_n+4A+h\varphi\left(t_{n+1},u_{n+1}\right)\right).$$

Chapitre 30.

2022 Contrôle Continu

- 2H30
- Deposer votre fichier .ipynb sur Moodle (dépôt de devoir)

```
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp
from scipy.optimize import fsolve
from scipy.integrate import solve_ivp, odeint
from IPython.display import display, Markdown, Math

prlat= lambda *args: display(Math(''.join( sp.latex(arg) for arg in args )))
```

30.1. EXERCICE 1 : étude d'un schéma multipas

On notera $\varphi_k \overset{\text{déf}}{=} \varphi(t_k, u_k)$.
Soit la méthode multipas

$$u_{n+1} = \alpha u_n + (1-\alpha)u_{n-1} + h\left(2\varphi_{n+1} - \frac{3\vartheta}{2}\varphi_n + \frac{\vartheta}{2}\varphi_{n-1}\right).$$

1. Pour quelles valeurs des paramètres α et ϑ la méthode est zéro-stable ?
2. Quel est l'ordre de convergence de la méthode ?
3. Nous allons maintenant fixer α. Pour ϑ, choisir une valeur qui garantie la convergence. Vérifier empiriquement l'ordre de convergence sur le problème de Cauchy

$$\begin{cases} y'(t) = t^2 - y(t), \quad \forall t \in I = [0,1], \\ y(0) = 2. \end{cases}$$

Q1 [2 points]
Pour quelles valeurs des paramètres α et ϑ la méthode est zéro-stable ?

Le premier polynôme caractéristique est

$$\varrho(r) = r^{p+1} - \sum_{j=0}^{p} a_j r^{p-j} = r^2 - \alpha r + (\alpha - 1) = (r-1)(r-(\alpha-1))$$

dont les racines sont

$$r_0 = 1, \quad r_1 = \alpha - 1.$$

La méthode est donc zéro-stable ssi

$$\begin{cases} |r_j| \leq 1 \quad \text{pour tout } j = 0, 1 \\ \varrho'(r_j) \neq 0 \text{ si } |r_j| = 1 \end{cases}$$

donc ssi $0 \leq \alpha < 2$ pour tout ϑ.

> **Q2 [3 points]**
> Quel est l'ordre de convergence de la méthode ?

On rappelle qu'une méthode est convergente ssi elle est zéro-stable et consistante. Nous allons donc étudier la consistance (puis l'ordre de consistance = ordre de convergence) pour les seules valeurs de α pour lesquelles on a la zéro-stabilité.

- On a $p = 1$: c'est une méthode à $q = p + 1 = 2$ pas.
- La méthode est implicite car $b_{-1} = 2 \neq 0$.

Pour que la méthode soit consistante il faut que $\xi(0) = \xi(1) = 1$.

De plus, la première barrière de Dahlquist affirme qu'un schéma implicite à $q = 2$ pas consistante et zéro-stable peut être au mieux d'ordre $\omega = q + 2 = 4$.

Pour que la méthode soit

- au moins d'ordre 2 il faut qu'elle soit consistante et que $\xi(2) = 1$,
- au moins d'ordre 3 il faut qu'elle soit au moins d'ordre 2 et que $\xi(3) = 1$,
- au moins d'ordre 4 il faut qu'elle soit au moins d'ordre 2 et que $\xi(4) = 1$.

```
q = 2
omega = 3

def xi(i, q):
    sa = sum((-j)**i * aa[j] for j in range(q))
    sb = b_m1 + sum((-j)**(i-1) * bb[j] for j in range(1, q))
    if i == 1:
        sb += bb[0]
    return (sa).factor() + (i * sb).factor()

def coeff(cas,q):
    if CAS_GENERAL:
        aa = sp.symbols(f'a_0:{q}')
        bb = sp.symbols(f'b_0:{q}')
        b_m1 = sp.Symbol('b_{-1}')
    else : # cas particulier
         = sp.Symbol(' ')
         = sp.Symbol(' ')
        aa = [ , 1- ]
        bb = [-3* /2,  /2]
        b_m1 = 2
    return aa, bb, b_m1


display(Markdown(f"Les conditions de consistance d'ordre {omega} sont :"))

CAS_GENERAL = True
aa, bb, b_m1 = coeff(CAS_GENERAL, q)
systeme = [sp.Eq(xi(i, q),1) for i in range(omega)]
display(Markdown(r"$\begin{cases}"+' '.join([r"\xi("+str(i)+")="+sp.latex(s)+r"\\" for i,s in
 ↪  enumerate(systeme)]) + r"\end{cases}$"))

display(Markdown(r"Ce qui donne le système suivant :"))

CAS_GENERAL = False
aa, bb, b_m1 = coeff(CAS_GENERAL, q)
systeme = [sp.Eq(xi(i, q),1) for i in range(omega)]
display(Markdown(r"$\begin{cases}"+' '.join([sp.latex(s)+r"\\" for s in systeme]) + r"\end{cases}$"))

display(Markdown(r"La solution est :"))
```

```
sol = sp.solve(systeme)
for key, s in sol.items():
    display(Math(sp.latex(key) + '=' + sp.latex(s)))
```

Les conditions de consistance d'ordre 3 sont :

$$\begin{cases} \xi(0) = a_0 + a_1 = 1 \\ \xi(1) = -a_1 + b_0 + b_1 + b_{-1} = 1 \\ \xi(2) = a_1 - 2\,(b_1 - b_{-1}) = 1 \end{cases}$$

Ce qui donne le système suivant :

$$\begin{cases} \text{True} \\ \alpha - \vartheta + 1 = 1 \\ -\alpha - \vartheta + 5 = 1 \end{cases}$$

La solution est :

$\alpha = 2$

$\vartheta = 2$

Dans notre cas

- la méthode est consistante ssi $\vartheta = \alpha$ car $a_0 + a_1 = \alpha + (1-\alpha) = 1$ et $-a_1 + b_0 + b_1 + b_{-1} = -(1-\alpha) - \frac{3\vartheta}{2} + \frac{\vartheta}{2} + 2 = \alpha - \vartheta + 1$;
- pour que la méthode sois d'ordre au moins 2 il faudrait avoir $1 = a_1 - 2(b_1 - b_{-1}) = 1 - \alpha - 2(\frac{\alpha}{2} - 2) = 5 - 2\alpha$ ce qui n'est pas compatible avec la condition de zéro-stabilité.

Conclusion : si $0 \leq \alpha = \vartheta < 1$ le schéma est convergent à l'ordre 1, dans les autres cas il n'est pas convergente.

> **Q3 [3 points]**
> Nous allons maintenant fixer α. Pour cela, écrivez votre nom et prénom dans la cellule ci-dessous et vous obtiendrez un choix pour le paramètre α. Pour ϑ, choisir une valeur qui garantie la convergence.
>
> Vérifier empiriquement l'ordre de convergence sur le problème de Cauchy
>
> $$\begin{cases} y'(t) = t^2 - y(t), \quad \forall t \in I = [0, 1], \\ y(0) = 2, \end{cases}$$

```
nom = "Faccanoni"
prenom = "Gloria"

NUM = list(range(1,16))
idx = sum([ord(c) for c in nom+prenom])%len(NUM)
my_alpha = NUM[idx]/sp.S(8)
display(Markdown(rf"On trouve pour $\alpha$ la valeur ${sp.latex(my_alpha)}$"))
my_alpha = float(my_alpha) # conversion en float
```

On trouve pour α la valeur $\frac{5}{8}$

- Pour estimer les erreurs on définit la solution exacte (calculée en utilisant le module `sympy` ou à la main).
- On calcule la solution approchée pour différentes valeurs de N. Pour le calcul de l'ordre de convergence on initialise la suite avec la solution exacte.

```
# variables globales
t0     = 0
tfinal = 1
y0     = 2

phi = lambda t,y: t**2-y

##############################################
# solution exacte
##############################################
t   = sp.Symbol('t')
y   = sp.Function('y')
edo = sp.Eq( sp.diff(y(t),t) , phi(t,y(t)) )
solpar = sp.dsolve(edo, ics={y(t0):y0}).simplify()
display(solpar)

sol_exacte = sp.lambdify(t,solpar.rhs,'numpy')

##############################################
# schéma (initialisation avec sol exacte pour ordre de convergence)
##############################################
def multipas(phi, tt, sol_exacte):
    h  = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    uu[:q] = sol_exacte(tt[:q])
    for n in range(q-1,len(tt) - 1):
        eq = lambda x :
 ↪  -x+my_alpha*uu[n]+(1-my_alpha)*uu[n-1]+h*(2*phi(tt[n+1],x)-3*my_alpha/2*phi(tt[n],uu[n])+my_alpha/2*phi(tt[n-
        temp = fsolve( eq ,uu[n])
        uu[n+1] = temp[0]
    return uu

##############################################
# ordre
##############################################
H = []
err = []
N = 50

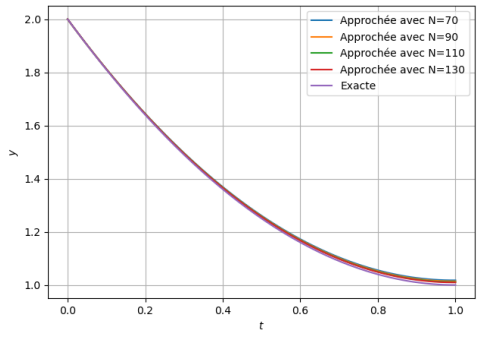
plt.figure(figsize=(16,5))
ax1 = plt.subplot(1,2,1)
ax2 = plt.subplot(1,2,2)

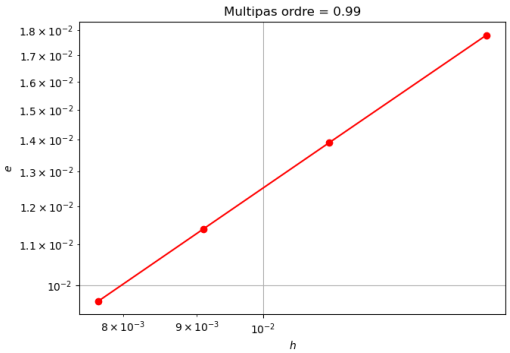
for k in range(4):
    N += 20
    tt = np.linspace(t0, tfinal, N + 1)
    H.append( tt[1] - tt[0] )
    yy = sol_exacte(tt)
    uu = multipas(phi, tt, sol_exacte)
    err.append( np.linalg.norm(uu-yy,np.inf) )
    ax1.plot(tt,uu,label=f'Approchée avec N={N}')

ax1.plot(tt,yy,label='Exacte')
ax1.set_xlabel('$t$')
ax1.set_ylabel('$y$')
ax1.grid(True)
ax1.legend()

ax2.loglog( H, err, 'r-o')
ax2.set_xlabel('$h$')
ax2.set_ylabel('$e$')
ax2.set_title(f'Multipas ordre = {np.polyfit(np.log(H),np.log(err),1)[0]:1.2f}')
ax2.grid(True)
```

$y(t) = t^2 - 2t + 2$





30.2. EXERCICE 2 : étude d'un système

Soit le problème de Cauchy

$$\begin{cases} y'(t) = \eta z(t), \\ z'(t) = y(t) + \eta z(t), \\ y(0) = 1, \\ z(0) = 1. \end{cases}$$

Pour fixer η, écrivez votre nom et prénom dans la cellule ci-dessous et vous obtiendrez un choix pour ce paramètre. Cela fixera aussi le nombre de points N.

```
nom = "Faccanoni"
prenom = "Gloria"

L = list(range(-5,0))
idx = sum([ord(c) for c in nom+prenom])%len(L)
eta = L[idx]
display(Math(r'\eta='+sp.latex(eta)))
N = -15*eta
print(f'{N = }')
```

$\eta = -1$

```
N = 15
```

Q1 [1 point]
Calculer la solution exacte.

```
t0 = 0
tfinal = 10

y0 = 1
z0 = 1
```

Calculons la solution exacte avec `sympy` par exemple :

```
t = sp.Symbol('t')
y = sp.Function('y')
z = sp.Function('z')

phi1 = lambda t,y,z : eta*z
phi2 = lambda t,y,z : y+eta*z
```

```
edo1 = sp.Eq( sp.diff(y(t),t) , phi1(t,y(t),z(t)) )
edo2 = sp.Eq( sp.diff(z(t),t) , phi2(t,y(t),z(t)) )
# display(edo1)
# display(edo2)
solpar_1, solpar_2 = sp.dsolve([edo1,edo2],[y(t),z(t)], ics={y(t0):y0, z(t0):z0})
display(solpar_1,solpar_2)
```

$$y(t) = -\frac{\sqrt{3}e^{-\frac{t}{2}}\sin\left(\frac{\sqrt{3}t}{2}\right)}{3} + e^{-\frac{t}{2}}\cos\left(\frac{\sqrt{3}t}{2}\right)$$

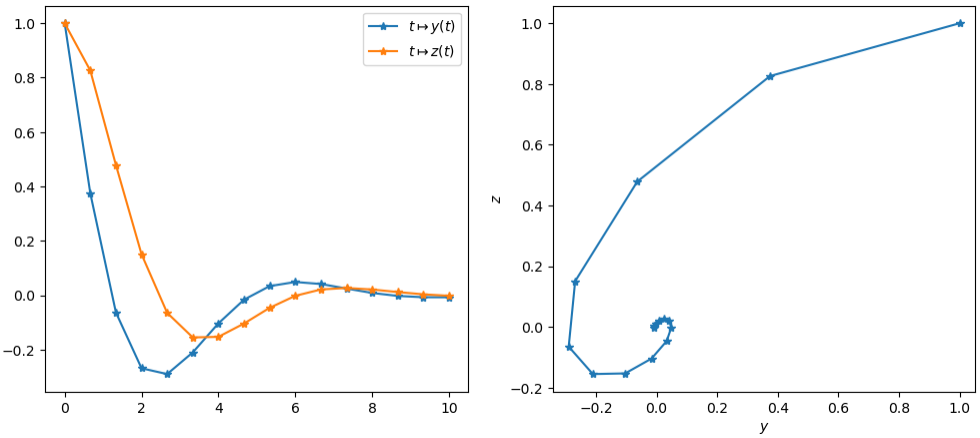
$$z(t) = \frac{\sqrt{3}e^{-\frac{t}{2}}\sin\left(\frac{\sqrt{3}t}{2}\right)}{3} + e^{-\frac{t}{2}}\cos\left(\frac{\sqrt{3}t}{2}\right)$$

On remarque que $y(t) \xrightarrow[t\to+\infty]{} 0$ et $z(t) \xrightarrow[t\to+\infty]{} 0$

```
func_1 = sp.lambdify(t,solpar_1.rhs,'numpy')
func_2 = sp.lambdify(t,solpar_2.rhs,'numpy')

tt = np.linspace(t0,tfinal,N+1)
h  = tt[1]-tt[0]

plt.figure(figsize=(12,5))
yy = func_1(tt)
zz = func_2(tt)
plt.subplot(1,2,1)
plt.plot(tt,yy,'-*',tt,zz,'-*')
plt.legend([r'$t\mapsto y(t)$',r'$t\mapsto z(t)$'])
plt.subplot(1,2,2)
plt.plot(yy,zz,'-*')
plt.xlabel(r'$y$')
plt.ylabel(r'$z$')
plt.axis('equal');
```



Dans la suite on notera $u_n \approx y(t_n)$ et $w_n \approx z(t_n)$.

Pour l'affichage des solutions exactes VS approchées, on peut écrire une fonction qui prend en paramètres la discrétisation `tt`, les deux listes solution exacte `yy` et `zz`, les deux listes solution approchée `uu` et `ww` et pour finir une chaîne de caractères `s` qui contient le nom du schéma.

NOTA BENE : on a juste fixé le nombre de points mais nous n'avons pas fixé la taille du domaine. On regardera la solution exacte pour choisir un T final significatif.

```
def affichage(tt,yy,zz,uu,ww,s):

    plt.figure(figsize=(12,5))

    plt.subplot(1,2,1)
    plt.plot(tt,uu,'o--',tt,ww,'d--')
    plt.plot(tt,yy,tt,zz)
    plt.xlabel('t')
    plt.legend([r'$u(t)$ approchée',r'$w(t)$ approchée','$y(t)$ exacte','$z(t)$ exacte'])
    plt.title(f'{s} - y(t) et z(t)')
    plt.grid()

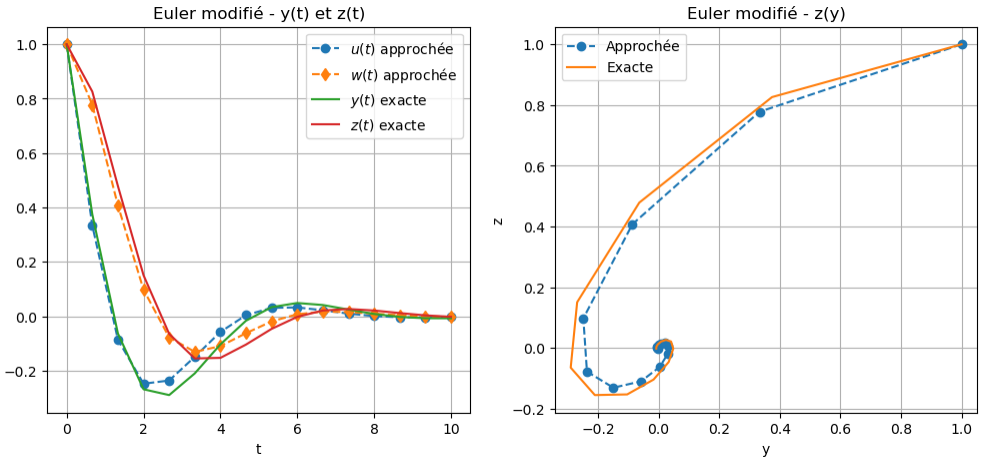
    plt.subplot(1,2,2)
    plt.plot(uu,ww,'o--')
    plt.plot(yy,zz)
    plt.xlabel('y')
    plt.ylabel('z')
    plt.legend(['Approchée','Exacte'])
    plt.title(f'{s} - z(y)')
    plt.grid()
    plt.axis('equal');
```

Q2 [3 points]
Calculer la solution approchée obtenue par la méthode d'**Euler modifié**.
Afficher $t \mapsto y(t)$, $t \mapsto z(t)$ et $y \mapsto z(y)$ en comparant solution exacte et solution approchée.

$$(\text{EM}) \begin{cases} u_0 = 1, \\ w_0 = 0, \\ \tilde{u} = u_n + \frac{h}{2}\varphi_1(t_n, u_n, w_n), \\ \tilde{w} = w_n + \frac{h}{2}\varphi_2(t_n, u_n, w_n), \\ u_{n+1} = u_n + h\varphi_1\left(t_n + \frac{h}{2}, \tilde{u}, \tilde{w}\right), \\ w_{n+1} = w_n + h\varphi_2\left(t_n + \frac{h}{2}, \tilde{u}, \tilde{w}\right). \end{cases}$$

```
# x0, y0, h variables globales
def EM(phi1,phi2,tt):
    uu, ww = np.zeros_like(tt), np.zeros_like(tt)
    uu[0], ww[0] = y0, z0
    for n in range(len(tt)-1):
        uu_tmp = uu[n]+h/2*phi1(tt[n],uu[n],ww[n])
        ww_tmp = ww[n]+h/2*phi2(tt[n],uu[n],ww[n])
        uu[n+1] = uu[n]+h*phi1(tt[n]+h/2,uu_tmp,ww_tmp)
        ww[n+1] = ww[n]+h*phi2(tt[n]+h/2,uu_tmp,ww_tmp)
    return [uu,ww]

[uu, ww] = EM(phi1,phi2,tt)
affichage(tt,yy,zz,uu,ww,"Euler modifié")
```



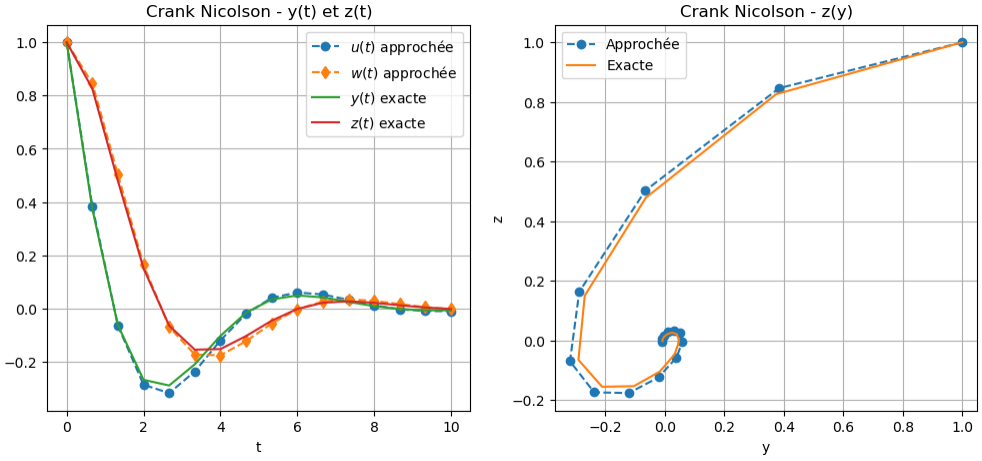
Q3 [3 points]
Calculer la solution approchée obtenue par la méthode de **Crank-Nicolson**.
Afficher $t \mapsto y(t)$, $t \mapsto z(t)$ et $y \mapsto z(y)$ en comparant solution exacte et solution approchée.

$$(\text{CN}) \begin{cases} u_0 = 1, \\ w_0 = 0, \\ u_{n+1} = u_n + \frac{h}{2}\left(\varphi_1(t_{n+1}, u_{n+1}, w_{n+1}) + \varphi_1(t_n, u_n, w_n)\right), \\ w_{n+1} = w_n + \frac{h}{2}\left(\varphi_2(t_{n+1}, u_{n+1}, w_{n+1}) + \varphi_2(t_n, u_n, w_n)\right). \end{cases}$$

```
from scipy.optimize import fsolve

# x0,y0, h variables globales
def CN(phi1,phi2,tt):
    uu, ww = np.zeros_like(tt), np.zeros_like(tt)
    uu[0], ww[0] = y0, z0
    for n in range(len(tt)-1):
        sys = lambda z : [ -z[0]+uu[n]+h/2*phi1(tt[n],uu[n],ww[n])+h/2*phi1(tt[n+1],z[0],z[1]) ,
 ↪  -z[1]+ww[n]+h/2*phi2(tt[n],uu[n],ww[n])+h/2*phi2(tt[n+1],z[0],z[1]) ]
        uu[n+1], ww[n+1] = fsolve( sys , (uu[n],ww[n]) )
    return [uu,ww]

[uu, ww] = CN(phi1,phi2,tt)
affichage(tt,yy,zz,uu,ww,"Crank Nicolson")
```



Dans ce cas particulier il est possible d'expliciter cette récurrence en exploitant la linéarité des φ_1 et φ_2 :

$$(\mathrm{CN})\ \begin{cases} u_0 = 1, \\ w_0 = 0, \\ u_{n+1} = u_n + \frac{h}{2}\eta\left(w_{n+1} + w_n\right), \\ w_{n+1} = w_n + \frac{h}{2}\left(u_{n+1} + u_n\right) + \frac{h}{2}\eta\left(w_{n+1} + \eta w_n\right) \end{cases}$$

ce qui donne le schéma explicite calculé ci-dessous par sympy :

```
t_n = sp.Symbol('t_n')
dt = sp.Symbol('h')
t_np1 = t_n+dt
u_n = sp.Symbol(r'u_{n}')
u_np1 = sp.Symbol(r'u_{n+1}')
w_n = sp.Symbol(r'w_{n}')
w_np1 = sp.Symbol(r'w_{n+1}')

Eq1 = sp.Eq (u_np1, u_n+dt/2*(phi1(t_n,u_n,w_n)+phi1(t_np1,u_np1,w_np1)) )
Eq2 = sp.Eq (w_np1, w_n+dt/2*(phi2(t_n,u_n,w_n)+phi2(t_np1,u_np1,w_np1)) )
# prlat(Eq1)
# prlat(Eq2)
sol = sp.solve([Eq1,Eq2],(u_np1,w_np1))

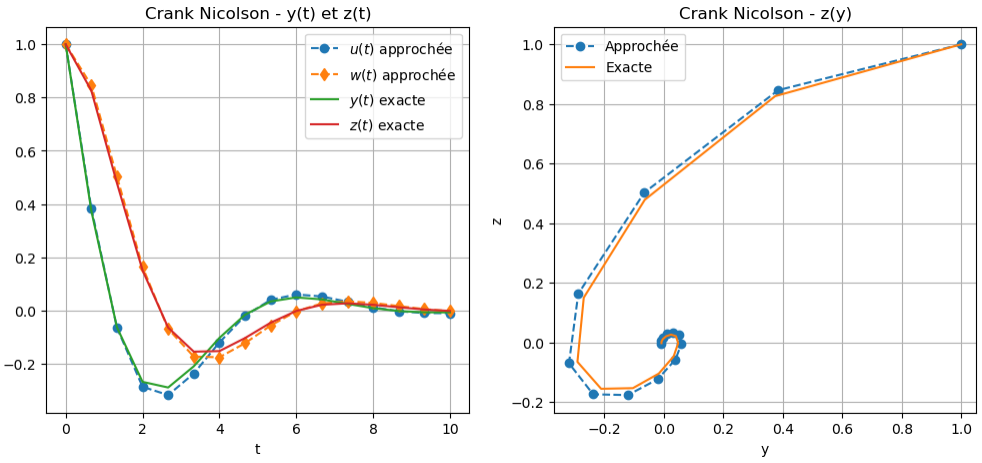
prlat(u_np1,"=",sol[u_np1].factor(u_n))
prlat(w_np1,"=",sol[w_np1].factor(w_n))
```

$$u_{n+1} = -\frac{4hw_n + u_n\left(h^2 - 2h - 4\right)}{h^2 + 2h + 4}$$

$$w_{n+1} = -\frac{-4hu_n + w_n\left(h^2 + 2h - 4\right)}{h^2 + 2h + 4}$$

```
# x0,y0, h variables globales
def CN(tt):
    uu, ww = np.zeros_like(tt), np.zeros_like(tt)
    uu[0], ww[0] = y0, z0
    for n in range(len(tt)-1):
        uu[n+1] = -((h**2-2*h-4)*uu[n]+4*h*ww[n])/(h**2+2*h+4)
        ww[n+1] = -((h**2+2*h-4)*ww[n]-4*h*uu[n])/(h**2+2*h+4)
    return [uu,ww]

[uu, ww] = CN(tt)
affichage(tt,yy,zz,uu,ww,"Crank Nicolson")
```

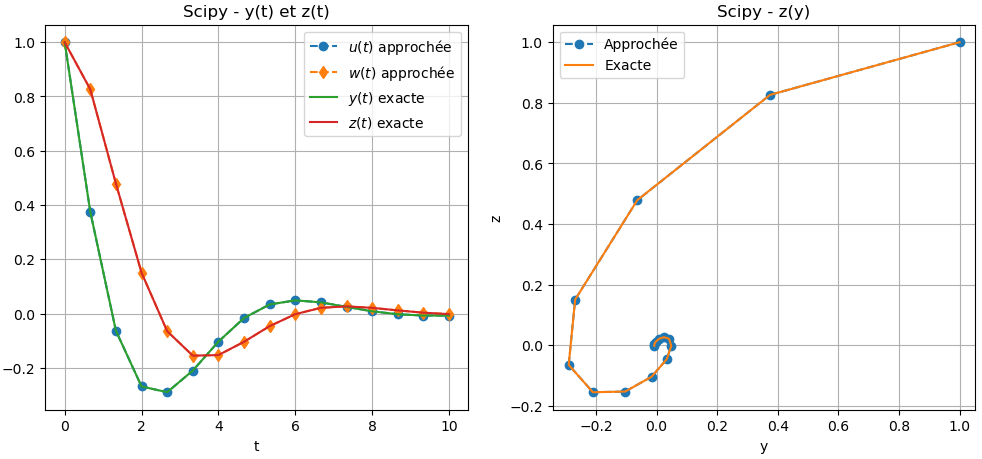


Q5 [1 point]
Calculer la solution approchée obtenue par la fonction `odeint` du module `scipy`.
Afficher $t \mapsto y(t)$, $t \mapsto z(t)$ et $y \mapsto z(y)$ en comparant solution exacte et solution approchée.

```
from scipy.integrate import odeint

pphi = lambda yy,t : [ phi1(t,yy[0],yy[1]), phi2(t,yy[0],yy[1])  ]

yy0  = [y0,z0]
uu,ww = odeint(pphi,yy0,tt).T
affichage(tt,yy,zz,uu,ww,"Scipy")
```



30.3. EXERCICE 3 : étude de la A-stabilité d'un schéma

Soit le schéma

$$\begin{cases} u_0 = y(t_0) = y_0, \\ \tilde{u}_{n+1/2} = u_n + \frac{h}{4}\left(\varphi(t_n, u_n) + \varphi(t_n + \frac{h}{2}, \tilde{u}_{n+1/2})\right), \\ u_{n+1} = u_n + h\varphi\left(t_n + \frac{h}{2}, \tilde{u}_{n+1/2}\right) \end{cases} \qquad n = 0, 1, 2, \dots N-1$$

1. Étudier théoriquement la A-stabilité du schéma.
2. Approcher la solution du problème de Cauchy suivant avec le schéma donné d'abord avec 120 points puis avec 130 points. Commenter le résultat.

$$\begin{cases} y'(t) = -50y(t), \quad t \in [0; 10] \\ y(0) = 1 \end{cases}$$

Q1 [3 points] Étudier théoriquement la A-stabilité du schéma.

Soit $\beta > 0$ un nombre réel positif et considérons le problème de Cauchy $y'(t) = -\beta y(t)$ pour $t > 0$ avec $y(0) = 1$. Sa solution est $y(t) = e^{-\beta t}$ donc $\lim\limits_{t \to +\infty} |y(t)| = 0$.

Le schéma appliqué à ce problème de Cauchy s'écrit

$$\begin{cases} u_0 = y(t_0) = y_0, \\ \tilde{u}_{n+1/2} = u_n - \frac{\beta h}{4} u_n - \frac{\beta h}{4} \tilde{u}_{n+1/2}, \\ u_{n+1} = u_n - \beta h \tilde{u}_{n+1/2} \end{cases} \qquad n = 0, 1, 2, \dots N-1$$

d'où

$$u_{n+1} = \left(1 - \beta h \frac{4-\beta h}{4+\beta h}\right) u_n$$

Vérifions ces calculs :

```
# pour ne pas effacer l'affectation de "h", ici je vais l'appeler "dt" mais afficher "h"
dt = sp.Symbol('h')
beta = sp.Symbol(r'\beta')
u_np1 = sp.Symbol(r'u_{n+1}')
sp.var('u_n,x')

u_tilde = sp.solve (-x + u_n - beta*dt/4*u_n-beta*dt/4*x, x)[0]
display(Math(r'\tilde u='+sp.latex(u_tilde)))

u_np1 = (u_n-beta*dt*u_tilde).factor()
display(Math('u_{n+1}='+sp.latex(u_np1)))
```

$$\tilde{u} = \frac{u_n\,(-\beta h + 4)}{\beta h + 4}$$

$$u_{n+1} = \frac{u_n\,(\beta^2 h^2 - 3\beta h + 4)}{\beta h + 4}$$

On note $x = \beta h$. Le schéma est A-stable ssi $|q(x)| < 1$. On étudie alors la fonction $q\colon \mathbb{R}^+ \to \mathbb{R}$ définie par $q(x) = \dfrac{x^2 - 3x + 4}{4 + x}$.

```
sp.var('x',nonnegative=True)

display(Markdown(r'Dans $\mathbb{R}^+$ on étudie la fonction'))
q = (x**2-3*x+4)/(x+4)
display(Math('q(x)='+sp.latex(q)))


display(Markdown('Limites aux bornes du domaine de definition:'))
display(Math('q(0)='+sp.latex(q.subs(x,0))))
lim=sp.Limit(q,x,sp.oo)
display(Math(sp.latex(lim)+'='+sp.latex(lim.doit())))

display(Markdown(r'On cherche les points stationnaires dans $\mathbb{R}^+$ :'))
dq=(q.diff(x)).factor()
display(Math("q'(x)="+sp.latex(dq)))
display(Markdown("On remarque que le numerateur de q' est une parabole et on a"))
sol=sp.solve(dq)
display(Math(r"q'(x)=0 \iff x\in"+sp.latex(sol)))
display(Markdown("De plus $q'(x)>0$ ssi $x$ est supérieur à cette valeur, donc"))
minimum=sol[0]
display(Math("x="+sp.latex(minimum)+"\\text{ est un minimum et}"))
qmin=q.subs(x,sol[0])
display(Math("q("+sp.latex(minimum)+")="+sp.latex(qmin)+"="+str(float(qmin))+">0"))
display(Markdown("Donc $q(x)>0$ pour tout $x$."))
display(Markdown("De plus, on a $q(x)<1$ ssi"))
display(sp.solve(q<1))
display(Markdown("Conclusion: $-1<q(x)<1$ pour tout $x<4$."))
sp.plot(q,1,-1,xlim=[-1,15],ylim=[-1.5,1.5]);
```

Dans $\mathbb{R}^+$ on étudie la fonction

$$q(x) = \frac{x^2 - 3x + 4}{x + 4}$$

Limites aux bornes du domaine de definition :

$q(0) = 1$

$\lim_{x\to\infty} \left(\frac{x^2 - 3x + 4}{x + 4}\right) = \infty$

On cherche les points stationnaires dans $\mathbb{R}^+$:

$q'(x) = \frac{x^2 + 8x - 16}{(x+4)^2}$

On remarque que le numerateur de q' est une parabole et on a

$q'(x) = 0 \iff x \in \left[-4 + 4\sqrt{2}\right]$

De plus $q'(x) > 0$ ssi x est supérieur à cette valeur, donc

$x = -4 + 4\sqrt{2}$ est un minimum et

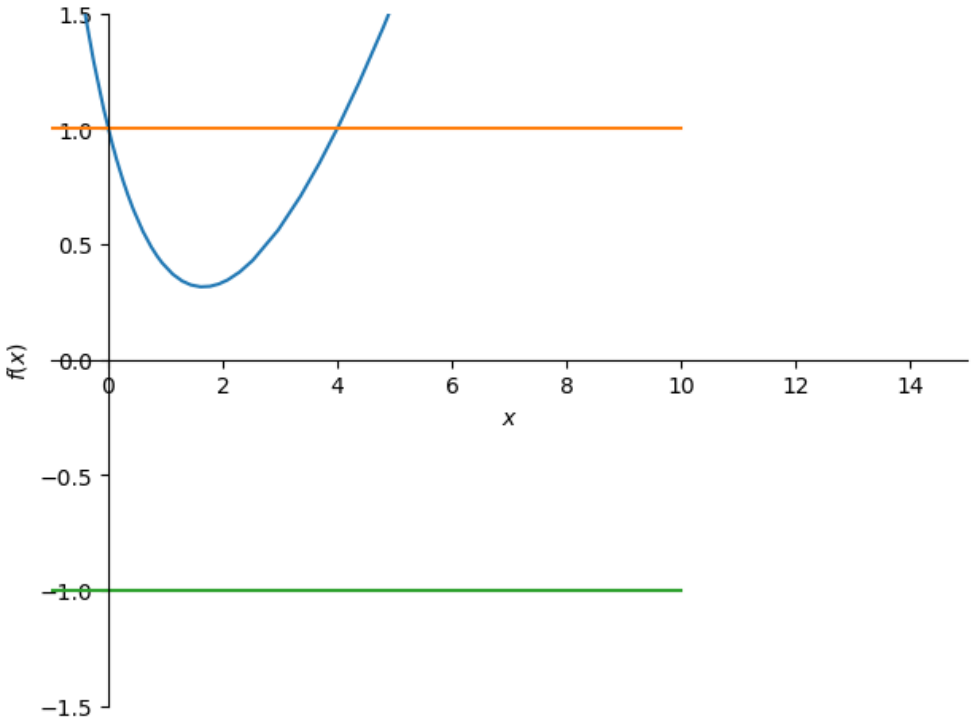
$q(-4 + 4\sqrt{2}) = \frac{\sqrt{2}\left(-12\sqrt{2} + \left(-4 + 4\sqrt{2}\right)^2 + 16\right)}{8} = 0.3137084989847604 > 0$

Donc $q(x) > 0$ pour tout x.

De plus, on a $q(x) < 1$ ssi

$0 < x \wedge x < 4$

Conclusion : $-1 < q(x) < 1$ pour tout $x < 4$.



Q2 [2 points] Approcher la solution du problème de Cauchy suivant avec le schéma donné d'abord avec 120 points puis avec 130 points. Commenter le résultat.

$$\begin{cases} y'(t) = -50y(t), \quad t \in [0; 10] \\ y(0) = 1 \end{cases}$$

Pour la A-stabilité il faut que $h = \frac{10-0}{N}$ soit $< \frac{4}{\beta}$ donc $N > \frac{(10-0)\beta}{4} = \frac{10\times 50}{4} = 125$. De plus, le problème est stiff.

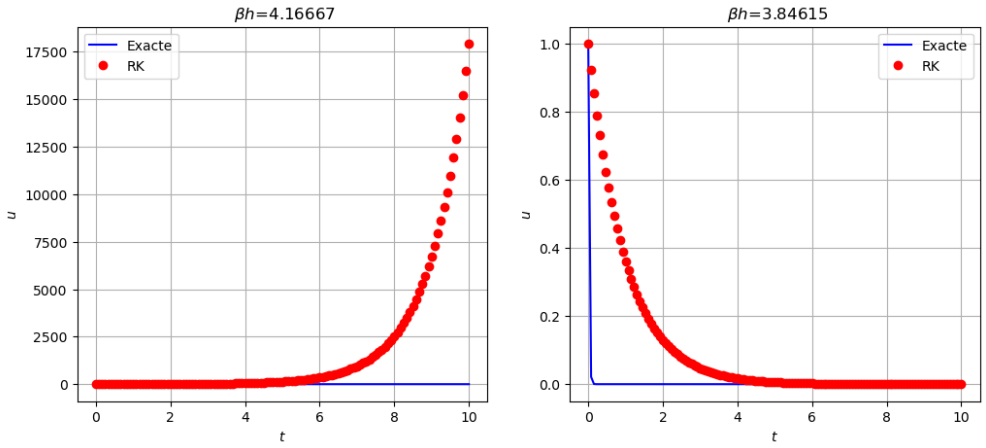
```
def schema(tt,phi,y0):
    uu = np.zeros_like(tt)
    uu[0] = y0
    h = tt[1]-tt[0]
    for n in range(len(tt)-1):
        u_tilde = fsolve ( lambda x : -x+uu[n]+h/4*phi(tt[n],uu[n])+h/4*phi(tt[n+1],x) , uu[n] )[0]
        uu[n+1] = uu[n]+h*phi(tt[n]+h/2,u_tilde)
    return uu


t0, y0, tfinal = 0, 1, 10
beta = 50

phi = lambda t,y : -beta*y
sol_exacte = lambda t : np.exp(-beta*t)

fig, ax = plt.subplots(nrows=1, ncols=2,figsize=(12,5))

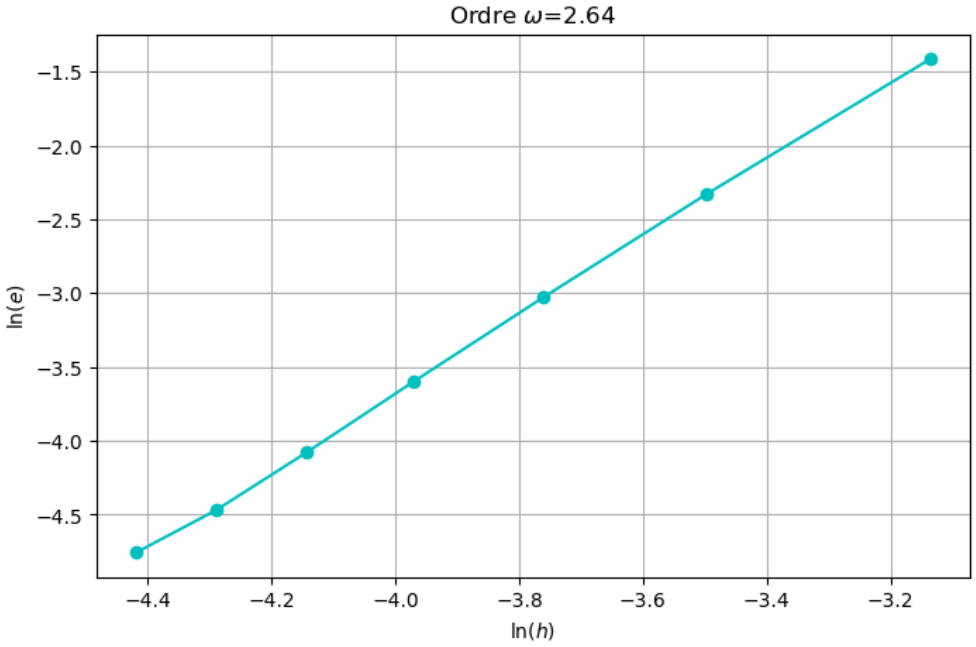
for i,N in enumerate([120, 130]) :
    tt = np.linspace(t0,tfinal,N+1)
    h  = tt[1]-tt[0]
    yy = sol_exacte(tt)
    uu = schema(tt,phi,y0)
    ax[i].plot(tt,yy,'b-',label=("Exacte"))
    ax[i].plot(tt,uu,'ro',label=("RK"))
    ax[i].set_title(rf' $\beta h$={beta*h:g}')
    ax[i].set_xlabel('$t$')
    ax[i].set_ylabel('$u$')
    ax[i].legend()
    ax[i].grid(True);
```




```
H = []
err = []
N = 130
for k in range(7):
    N += 100
    tt = np.linspace(t0, tfinal, N + 1)
    h = tt[1] - tt[0]
    yy = sol_exacte(tt)
    uu = schema(tt,phi,y0)
    H.append(h)
    err.append(np.linalg.norm(uu-yy,np.inf))


omega = np.polyfit(np.log(H),np.log(err), 1)[0]
```

```
plt.figure(figsize=(8,5));
plt.plot(np.log(H), np.log(err), 'c-o')
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
plt.title(rf'Ordre $\omega$={omega:1.2f}')
plt.grid(True);
```



Chapitre 31.

2022 Contrôle Terminal session 1

- 2H30
- Deposer votre notebook .ipynb sur Moodle (dépôt de devoir)

Dans le sujet il y a plusieurs paramètres. Pour les fixer, remplacez mon nom et prénom par les votres dans les cellules indiquées et vous obtiendrez un choix pour chaque paramètre.

```
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp
from scipy.optimize import fsolve
from scipy.integrate import solve_ivp, odeint
from IPython.display import display, Markdown, Math
```

31.1. EXERCICE 1 : Méthode d'Euler exponentielle

On considère le problème de Cauchy

$$\begin{cases} y'(t) = ay(t) + b(t), \quad t \in [0;2] \\ y(t_0) = y_0 \end{cases}$$

La méthode d'Euler exponentielle appliquée à ce problème de Cauchy s'écrit comme suit :

$$\text{(EEx)} \quad \begin{cases} u_0 = y_0, \\ u_{n+1} = e^{ha}u_n + hg(ha)b(t_n + h/2), \quad n = 0, \dots, N-1 \end{cases}$$

où $g(z) = \dfrac{e^z - 1}{z}$.

1. Comparer la solution exacte, la solution approchée obtenue par la méthode d'Euler explicite (EE) et la solution obtenue avec la méthode d'Euler exponentielle (EEx) avec $N + 1$ points.
2. Comparer ensuite empiriquement les ordres de convergence.

La donnée initiale y_0, la constante a, la fonction $t \mapsto b(t)$ et le nombre de points N sont donnés dans la cellule ci-dessous (modifiér nom et prénom).

```
prlat = lambda *args: display(Math(''.join( sp.latex(arg) for arg in args )))


nom = "Faccanoni"
prenom = "Gloria"

print("Paramètres de l'exercice 1")
S = sum([ord(c) for c in nom+prenom])
L = list(range(1,5))
idx = S%len(L)
y0 = L[idx]
prlat("y_0 = ",y0)
```

```
L = [ (-10,1), (-10,3), (-10,5), (-5,1), (-5,3) ]
idx = S%len(L)
a,d = L[idx]
c,N = -a, 1-a
prlat("a = ",a)
prlat("b(t) = ",c,"t+",d)
prlat("N = ",N)
```

```
Paramètres de l'exercice 1
```

y_0 = 1

a = −5

b(t) = 5t+3

N = 6

Q1 [3 points] Comparer la solution exacte, la solution approchée obtenue par la méthode d'Euler explicite (EE) et la solution obtenue avec la méthode d'Euler exponentielle (EEx) avec $N+1$ points.

```
# ========================================================
# variables globales
# ========================================================
t0     = 0
tfinal = 2

# ========================================================
# solution exacte symbolique
# ========================================================
t = sp.Symbol('t')
y = sp.Function('y')

b = lambda t :  c*t+d
phi = lambda t,y :  a*y+b(t)

edo = sp.Eq( sp.diff(y(t),t) , phi(t,y(t)) )
display(edo)
solpar = sp.dsolve(edo, y(t), ics={y(t0):y0})
display(solpar.simplify())

# ========================================================
# solution exacte numérique
# ========================================================
print(f'Choix: {a=}, b(t)={c}t+{d} donc la solution exacte est')

sol_exacte = sp.lambdify(t,solpar.rhs,'numpy')

# ========================================================
# SCHÉMAS
# ========================================================
schemas = ['EEx','EE']
Nb_schemas = len(schemas)

def EEx(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.zeros_like(tt)
    uu[0] = y0
    for n in range(len(tt)-1):
        K1 = (np.exp(a*h)-1)/(a*h)
        uu[n+1] = np.exp(a*h)*uu[n]+h*K1*b(tt[n]+h/2)
    return uu
```

```
def EE(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.zeros_like(tt)
    uu[0] = y0
    for n in range(len(tt)-1):
        uu[n+1] = uu[n]+h*phi(tt[n],uu[n])
    return uu

# =====================================================
# CALCUL SOLUTION APPROCHEE
# =====================================================
tt = np.linspace(t0,tfinal,N+1)
yy = sol_exacte(tt)

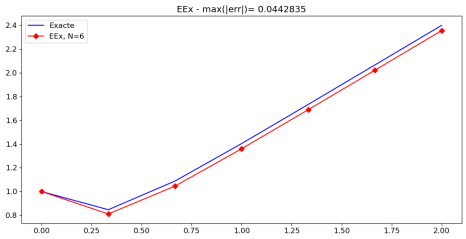
uu  = { schemas[s] : eval(schemas[s])(phi,tt,y0) for s in range(Nb_schemas) }
err = { schemas[s] : np.linalg.norm(uu[schemas[s]]-yy,np.inf) for s in range(Nb_schemas) }

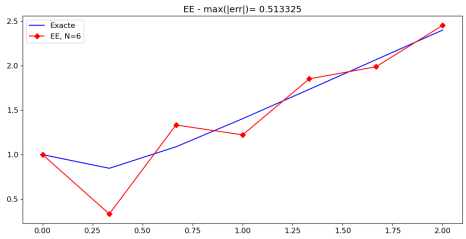
plt.figure(figsize=(28,6))
idx = 0
for key in uu:
    plt.subplot(1,Nb_schemas,idx+1)
    plt.plot(tt,yy,'b-',tt,uu[key],'r-D')
    plt.legend(['Exacte', f'{key}, {N=}'])
    plt.title( f'{key} - max(|err|)= {err[key]:g}' )
    idx += 1
```

$$\frac{d}{dt}y(t) = 5t - 5y(t) + 3$$

$$y(t) = t + \frac{2}{5} + \frac{3e^{-5t}}{5}$$

Choix: a=-5, b(t)=5t+3 donc la solution exacte est





Q2 [2 points] Comparer ensuite empiriquement les ordres de convergence.

```
# =====================================================
# CALCUL ORDRE
# =====================================================
H = []
err= { schemas[s] : [] for s in range(Nb_schemas) }

plt.figure(figsize=(12,6))
ax = []
for i in range(Nb_schemas):
    ax.append(plt.subplot(1,Nb_schemas,i+1))

for k in range(6):
    N += 20
    h = (tfinal-t0)/N
    H.append(h)
    tt = np.linspace(t0,tfinal,N+1)
```

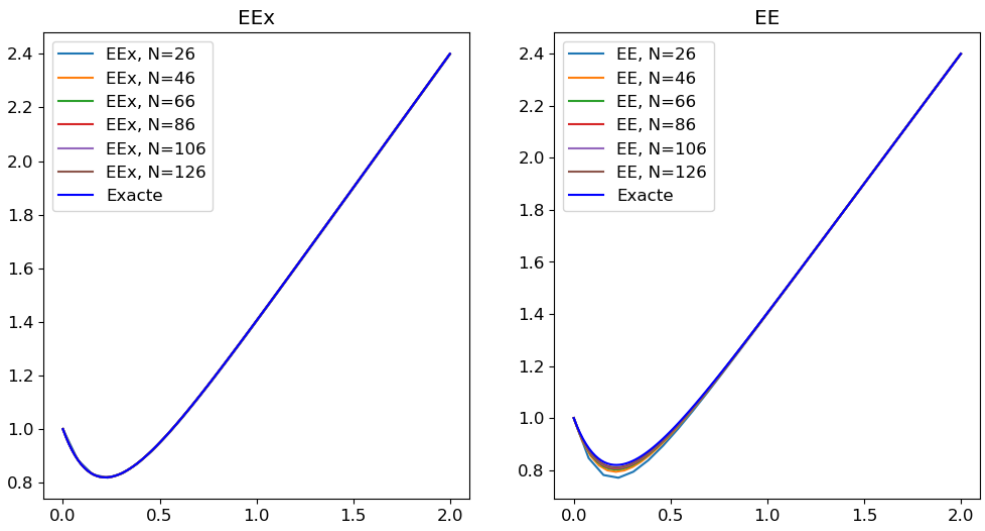
```
    yy = sol_exacte(tt)
    uu = { schemas[s] : eval(schemas[s])(phi,tt,y0) for s in range(Nb_schemas) }
    idx = 0
    for key in uu:
        err[key].append( np.linalg.norm(uu[key]-yy,np.inf) )
        ax[idx].plot(tt,uu[key],label=f'{key}, {N=}')
        idx += 1

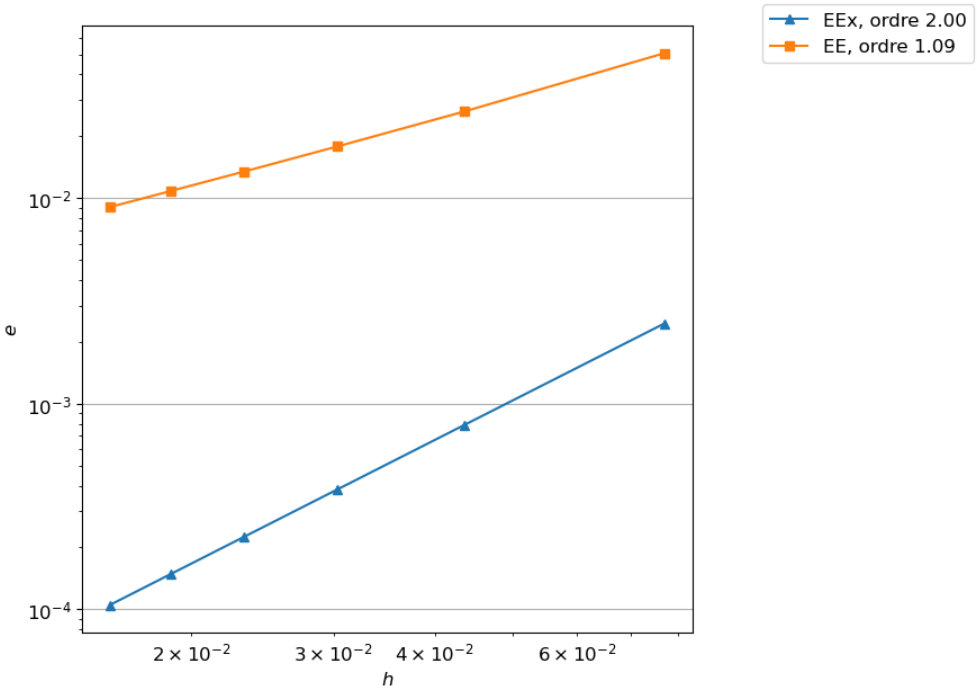
for i in range(Nb_schemas):
    ax[i].plot(tt,yy,'b-',label='Exacte')
    ax[i].legend()
    ax[i].set_title(schemas[i])

ordre = {  key : np.polyfit(np.log(H),np.log(err[key]),1)[0] for key in err  }

plt.figure(figsize=(7,7))
markers = ['^', 's', 'p', 'h', '8', 'D', '>', '<', '*','+','o']

for s in range(Nb_schemas):
    plt.loglog(H,err[schemas[s]], label=f'{schemas[s]}, ordre
 ↪  {ordre[schemas[s]]:1.2f}',marker=markers[s])
plt.xlabel('$h$')
plt.ylabel('$e$')
plt.legend(bbox_to_anchor=(1.1, 1.05))
plt.grid(True)
```





31.2. EXERCICE 2 : étude d'un schéma Runge-Kutta

Considérons le schéma dont la matrice de Butcher est donnée ci-dessous :

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{5}{24} & \frac{1}{3} & \frac{-1}{24} \\ 1 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{array}$$

1. Écrire la suite définie par réccurence associée à ce schéma.
2. Étudier théoriquement l'ordre du schéma.
3. Étudier théoriquement la A-stabilité du schéma.

4. Considérons le problème de Cauchy $y'(t) = -50y(t)$ pour $t \in]0, 1]$ et $y(0) = 1$. Vérifier empiriquement qu'avec $N = 8$ (i.e. 9 points de discrétisation) le schéma est instable et avec $N = 21$ il est stable.

Cf. exercice 7.4 page 86 polycopié Christophe Besse
https://www.math.univ-toulouse.fr/~cbesse/mesassets/teaching/L3Mapi3/EDONum/Cours_EDO_C_Besse.pdf

Q1 [1 points] Écrire la suite définie par réccurence associée à ce schéma.

$$\begin{cases} u_0 = y_0 \\ K_1 = \varphi(t_n, u_n) \\ K_2 = \varphi\left(t_n + \frac{h}{2}, u_n + h\left(\frac{5}{24}K_1 + \frac{1}{3}K_2 - \frac{1}{24}K_3\right)\right) \\ K_3 = \varphi(t_{n+1}, u_n + hK_2) \\ u_{n+1} = u_n + \frac{h}{6}(K_1 + 4K_2 + K_3) \qquad\qquad n = 0, 1, \dots N-1 \end{cases}$$

Q2 [3 points] Étudier théoriquement l'ordre du schéma.

Soit ω l'ordre de la méthode.

C'est un schéma implicite à 3 étages ($s = 3$) donc $\omega \leq 2s = 6$

```
prlat= lambda *args: display(Math(''.join( sp.latex(arg) for arg in args )))

def ordre_RK(s, A=None, b=None, c=None):

    j = sp.symbols('j')

    if A is None:
        A = sp.MatrixSymbol('a', s, s)
    else:
        A = sp.Matrix(A)
    if c is None:
        c = sp.symbols(f'c_0:{s}')
    if b is None:
        b = sp.symbols(f'b_0:{s}')


    display(Markdown("**Matrice de Butcher**"))
    But = matrice_Butcher(s, A, b, c)

    Eqs = []

    # display(Markdown(f"**On a {s+1} conditions pour avoir consistance = pour être d'ordre 1**"))
    # ordre_1(s, A, b, c, j)

    display(Markdown("**On doit ajouter 1 condition pour être d'ordre 2**"))
    Eq_2 = ordre_2(s, A, b, c, j)
    Eqs.append(Eq_2)

    display(Markdown("**On doit ajouter 2 conditions pour être d'ordre 3**"))
    Eq_3 = ordre_3(s, A, b, c, j)
    Eqs.extend(Eq_3)

    display(Markdown("**On doit ajouter 4 conditions pour être d'ordre 4**"))
    Eq_4 = ordre_4(s, A, b, c, j)
    Eqs.extend(Eq_4)

    return Eqs


def matrice_Butcher(s, A, b, c):
    But = sp.Matrix(A)
    But = But.col_insert(0, sp.Matrix(c))
    last = [sp.Symbol(" ")]
    last.extend(b)
    But = But.row_insert(s,sp.Matrix(last).T)
    display(Math(sp.latex(sp.Matrix(But))))
    return But


def ordre_1(s, A, b, c, j):
    texte = r"\sum_{j=1}^s b_j =" + f"{sum(b).simplify()}"
    texte += r"\text{ doit être égale à }1"
    display(Math(texte))
    for i in range(s):
        somma = sp.summation(A[i,j],(j,0,s-1)).simplify()
        texte = r'\sum_{j=1}^s a_{'+str(i)+r'j}=' + sp.latex( somma )
        texte += r"\text{ doit être égale à }"+sp.latex(c[i])
        display(Math( texte ))
    return None


def ordre_2(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j=' +sp.latex(sum([b[i]*c[i] for i in range(s)]).simplify())
```

```
    texte += r"\text{ doit être égale à }\frac{1}{2}"
    display(Math(texte))
    Eq_2 = sp.Eq( sum([b[i]*c[i] for i in range(s)]).simplify(), sp.Rational(1,2) )
    return Eq_2


def ordre_3(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j^2='
    texte += sp.latex( sum([b[i]*c[i]**2 for i in range(s)]).simplify() )
    texte += r"\text{ doit être égale à }\frac{1}{3}"
    display(Math(texte))
    Eq_3_1 = sp.Eq( sum([b[i]*c[i]**2 for i in range(s)]).simplify(), sp.Rational(1,3) )
    texte = r'\sum_{i,j=1}^s b_i a_{ij} c_j='
    somma = sum([b[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{6}"
    display(Math(texte))
    Eq_3_2 = sp.Eq( sum([b[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify(),
 ↪  sp.Rational(1,6) )
    return [Eq_3_1, Eq_3_2]


def ordre_4(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j^3='
    texte += sp.latex( sum([b[i]*c[i]**3 for i in range(s)]).simplify() )
    texte += r"\text{ doit être égale à }\frac{1}{4}"
    display(Math(texte))
    Eq_4_1 = sp.Eq( sum([b[i]*c[i]**3 for i in range(s)]).simplify(), sp.Rational(1,4) )
    texte = r'\sum_{i,j=1}^s b_i c_i a_{ij} c_j='
    somma = sum([b[i]*c[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{8}"
    display(Math(texte))
    Eq_4_2 = sp.Eq( sum([b[i]*c[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify(),
 ↪  sp.Rational(1,8) )
    texte = r'\sum_{i,j=1}^s b_i a_{ij} c_j^2='
    somma = sum([b[i]*A[i,j]*c[j]**2 for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{12}"
    display(Math(texte))
    Eq_4_3 = sp.Eq( sum([b[i]*A[i,j]*c[j]**2 for j in range(s) for i in range(s)]).simplify(),
 ↪  sp.Rational(1,12) )
    texte = r'\sum_{i,j,k=1}^s b_i a_{ij} a_{jk} c_k='
    somma = sum([b[i]*A[i,j]*A[j,k]*c[k] for k in range(s) for j in range(s) for i in
 ↪  range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{24}"
    display(Math(texte))
    Eq_4_4 = sp.Eq( sum([b[i]*A[i,j]*A[j,k]*c[k] for k in range(s) for j in range(s) for i in
 ↪  range(s)]).simplify(), sp.Rational(1,24) )
    return [Eq_4_1, Eq_4_2, Eq_4_3, Eq_4_4]



##################################################
# Conditions avec s étages
##################################################

c = [0,sp.Rational(1,2),1]
b = [sp.Rational(1,6),sp.Rational(2,3),sp.Rational(1,6)]
A = [ [0,0,0],
      [sp.Rational(5,24),sp.Rational(1,3),-sp.Rational(1,24)],
      [0,1,0]]
```

```
s = len(c)

Eqs = ordre_RK(s, A, b, c)

# for eq in Eqs:
#     display(eq)
#
```

Matrice de Butcher

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{5}{24} & \frac{1}{3} & -\frac{1}{24} \\ 1 & 0 & 1 & 0 \\ & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{bmatrix}$$

On doit ajouter 1 condition pour être d'ordre 2

$\sum_{j=1}^{s} b_j c_j = \frac{1}{2}$ doit être égale à $\frac{1}{2}$

On doit ajouter 2 conditions pour être d'ordre 3

$\sum_{j=1}^{s} b_j c_j^2 = \frac{1}{3}$ doit être égale à $\frac{1}{3}$

$\sum_{i,j=1}^{s} b_i a_{ij} c_j = \frac{1}{6}$ doit être égale à $\frac{1}{6}$

On doit ajouter 4 conditions pour être d'ordre 4

$\sum_{j=1}^{s} b_j c_j^3 = \frac{1}{4}$ doit être égale à $\frac{1}{4}$

$\sum_{i,j=1}^{s} b_i c_i a_{ij} c_j = \frac{1}{8}$ doit être égale à $\frac{1}{8}$

$\sum_{i,j=1}^{s} b_i a_{ij} c_j^2 = \frac{5}{72}$ doit être égale à $\frac{1}{12}$

$\sum_{i,j,k=1}^{s} b_i a_{ij} a_{jk} c_k = \frac{5}{144}$ doit être égale à $\frac{1}{24}$

D'après ces résultats, le schéma est d'ordre 3.

Q3 [4 points] Étudier théoriquement la A-stabilité du schéma.

La méthode appliquée à l'EDO $y'(t) = -\beta y(t)$ avec $\beta > 0$ s'écrit :

$$\begin{cases} u_0 = y_0 \\ K_1 = -\beta u_n \\ K_2 = -\beta\left(u_n + \frac{h}{24}\left(5K_1 + 8K_2 - K_3\right)\right) \\ K_3 = -\beta\left(u_n + hK_2\right) \\ u_{n+1} = u_n + \frac{h}{6}\left(K_1 + 4K_2 + K_3\right) \qquad n = 0, 1, \dots N-1 \end{cases}$$

En effet, si on note $x = \beta h$, par induction on obtient

$$u_n = \left(-\frac{x^3 - 5x^2 + 16x - 24}{x^2 + 8x + 24}\right)^n u_0.$$

Par conséquent, $\lim_{n \to +\infty} u_n = 0$ si et seulement si

$$\left|\frac{x^3 - 5x^2 + 16x - 24}{x^2 + 8x + 24}\right| < 1.$$

L'étude de la fonction $x \mapsto \frac{x^3-5x^2+16x-24}{x^2+8x+24}$ pour $x>0$ montre que le schéma est A-stable ssi $h<\frac{6}{\beta}$.

```
u_n, h, beta, K1, K2, K3 = sp.symbols(r'u_n h \beta K_1 K_2 K_3')

g = lambda y : -beta*y

eq1 = sp.Eq( K1 , g(u_n))
eq2 = sp.Eq( K2 , g(u_n+h*5/24*K1+h/3*K2-h/24*K3) )
eq3 = sp.Eq( K3 , g(u_n+h*K2) )
sol = sp.solve([eq1,eq2,eq3],[K1,K2,K3])
for key in sol:
    texte = sp.latex(key) + "=" + sp.latex(sol[key])
    display(Math(texte))

RHS = u_n+h/6*(K1+2*K2+K3)
RHS = RHS.subs(sol).simplify()
texte = 'u_{n+1}=' + sp.latex(RHS)
display(Math(texte))

display(Markdown(r"Notons $x=h\beta>0$. On trouve donc"))
x = sp.Symbol('x',positive=True)
RHS = RHS.subs(h*beta,x).simplify()
texte = 'u_{n+1}=' + sp.latex(RHS)
display(Math(texte))
texte = r"\text{La méthode est A-stable ssi }|q(x)|<1\text{ avec }q(x)=\frac{u_{n+1}}{u_n}="
texte += sp.latex(RHS/u_n)
display(Math(texte))
sp.solve([-1<RHS/u_n,RHS/u_n<1])
```

$$K_1 = -\beta u_n$$

$$K_2 = \frac{4\beta^2 h u_n - 24\beta u_n}{\beta^2 h^2 + 8\beta h + 24}$$

$$K_3 = \frac{-5\beta^3 h^2 u_n + 16\beta^2 h u_n - 24\beta u_n}{\beta^2 h^2 + 8\beta h + 24}$$

$$u_{n+1} = \frac{u_n\left(-3\beta^3 h^3 + 11\beta^2 h^2 - 24\beta h + 72\right)}{3\left(\beta^2 h^2 + 8\beta h + 24\right)}$$

Notons $x = h\beta > 0$. On trouve donc

$$u_{n+1} = \frac{u_n\left(-3x^3 + 11x^2 - 24x + 72\right)}{3\left(x^2 + 8x + 24\right)}$$

La méthode est A-stable ssi $|q(x)|<1$ avec $q(x) = \frac{u_{n+1}}{u_n} = \frac{-3x^3 + 11x^2 - 24x + 72}{3\left(x^2 + 8x + 24\right)}$

$x<6$

```
q = -RHS/u_n
display( Math (r'-\frac{u_{n+1}}{u_n}=' + sp.latex(q) + "=:q(x)") )
display( Math (r'q(0)=' + sp.latex(q.subs(x,0)) ) )
display( Math (r'\lim_{x\to+\infty}q(x)=' + sp.latex(q.limit(x,sp.oo)) ) )
dq = sp.diff(q,x).simplify()
display( Math ( r"q'(x)=" + sp.latex(dq) ) )
display( Math ( r"q'(x)>0 \text{ ? } " + sp.latex(sp.solve(dq>0)) ) )
display( Math ( r"q(x)=1 \iff x= " + sp.latex(sp.solve(q-1)) ) )
# prlat( r"q(x)=0 \iff x= " ,[ xi.evalf() for xi in sp.solve(q)] )

xx = np.linspace(0,8,100)
yy = (xx**3-5*xx**2+16*xx-24)/(xx**2+8*xx+24)
plt.plot(xx,yy)
plt.plot([xx[0],xx[-1]],[-1,-1])
```

```
plt.plot([xx[0],xx[-1]],[1,1])
plt.grid();
```

$$-\frac{u_{n+1}}{u_n} = -\frac{-3x^3 + 11x^2 - 24x + 72}{3\left(x^2 + 8x + 24\right)} =: q(x)$$

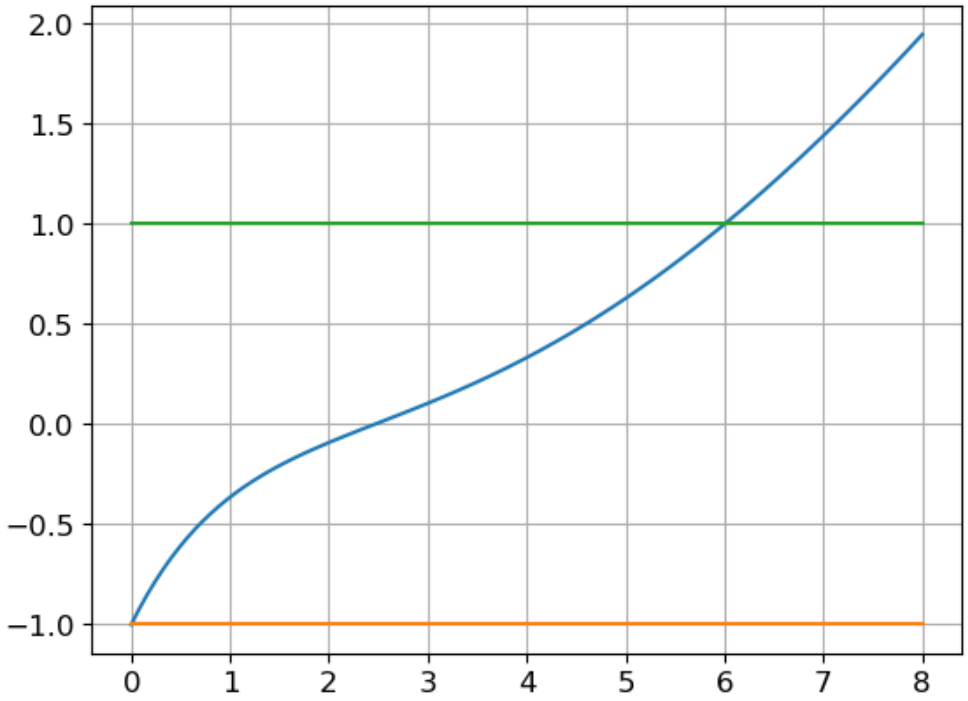
$q(0) = -1$

$\lim_{x \to +\infty} q(x) = \infty$

$$q'(x) = \frac{x^4 + 16x^3 + \frac{104x^2}{3} - 128x + 384}{x^4 + 16x^3 + 112x^2 + 384x + 576}$$

$q'(x) > 0$? True

$q(x) = 1 \iff x = [6]$



Elle est strictement croissante sur $[0, +\infty[$, $q(0) = -1$ et $\lim_{x \to +\infty} q(x) = +\infty$. Il existe donc un et un seul x_0 tel que $q(x_0) = 1$. En résolvant l'équation on trouve $x_0 = 6$.

La condition de stabilité est donc satisfaite ssi $0 < x < 6$, donc ssi

$$h < \frac{6}{\beta}.$$

Q4 [3 points] Considérons le problème de Cauchy $y'(t) = -50y(t)$ pour $t \in]0, 1]$ et $y(0) = 1$.
Vérifier empiriquement qu'avec $N = 8$ (i.e. 9 points de discrétisation) le schéma est instable et avec $N = 21$ il est stable.

```
# variables globales
t0     = 0
tfinal = 1
y0     = 1
beta      = 50

phi = lambda t,y: -beta*y

################################################
```

```
# solution exacte
################################################
sol_exacte = lambda t : y0*np.exp(-beta*t)

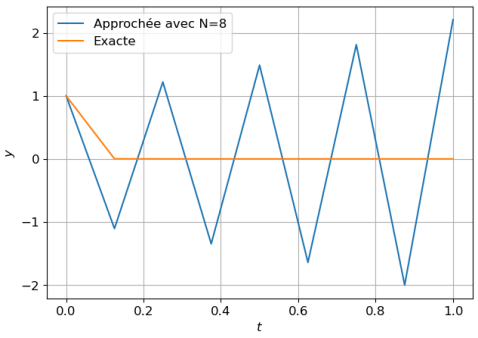
################################################
# solution approchée
################################################
def RK(tt,phi,y0):
    h = tt[1]-tt[0]
    uu = np.zeros_like(tt)
    uu[0] = y0
    for n in range(len(tt)-1):
        K1 = phi(tt[n],uu[n])
        sys = lambda X : [-X[0]+phi(tt[n]+h/2,uu[n]+h*(5/24*K1+X[0]/3-X[1]/24)) ,
                          -X[1]+phi(tt[n+1],uu[n]+h*X[0])]
        K2, K3 = fsolve( sys , [K1,K1] )
        uu[n+1] = uu[n]+h/6*(K1+4*K2+K3)
    return uu

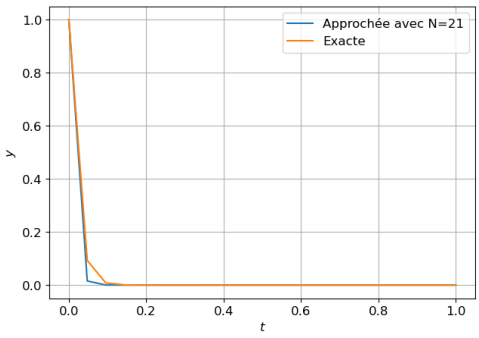
################################################
# ordre
################################################
# 6/beta>h=(tfinal-t0)/N
# N>(tfinal-t0)*beta/6

plt.figure(figsize=(16,5))

plt.subplot(1,2,1)
N = int((tfinal-t0)*beta/6)
tt = np.linspace(t0, tfinal, N + 1)
h  = tt[1] - tt[0]
yy = sol_exacte(tt)
uu = RK(tt, phi, y0)
err = np.linalg.norm(uu-yy,np.inf)
plt.plot(tt,uu,label=f'Approchée avec N={N}')
plt.plot(tt,yy,label='Exacte')
plt.xlabel('$t$')
plt.ylabel('$y$')
plt.grid(True)
plt.legend();

plt.subplot(1,2,2)
N = int(int((tfinal-t0)*beta/6)*2.46115761806205+2)
tt = np.linspace(t0, tfinal, N + 1)
h  = tt[1] - tt[0]
yy = sol_exacte(tt)
uu = RK(tt, phi, y0)
err = np.linalg.norm(uu-yy,np.inf)
plt.plot(tt,uu,label=f'Approchée avec N={N}')
plt.plot(tt,yy,label='Exacte')
plt.xlabel('$t$')
plt.ylabel('$y$')
plt.grid(True)
plt.legend();
```





Annexe : calculons la suite géométrique pour étudier la A-stabilité dans le cas générale d'un schéma dont la matrice de Butcher est

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ c_2 & a_{21} & a_{22} & a_{23} \\ 1 & 0 & 1 & 0 \\ \hline & b_1 & b_2 & b_3 \end{array}$$

```
u_n, h, beta, K1, K2, K3 = sp.symbols(r'u_n h \beta K_1 K_2 K_3')
b_1, b_2, b_3, a_21, a_22, a_23 = sp.symbols(r'b_1 b_2 b_3 a_{21} a_{22} a_{23}')
g = lambda y : -beta*y
eq1 = sp.Eq( K1 , g(u_n))
eq2 = sp.Eq( K2 , g(u_n+h*a_21*K1+h*a_22*K2+h*a_23*K3) )
eq3 = sp.Eq( K3 , g(u_n+h*K2) )
sol = sp.solve([eq1,eq2,eq3],[K1,K2,K3])
for key in sol:
    texte = sp.latex(key) + "=" + sp.latex(sol[key])
    display(Math(texte))
RHS = u_n+h*(b_1*K1+b_2*K2+b_3*K3)
RHS = RHS.subs(sol).simplify()
display(Math( 'u_{n+1}=' + sp.latex(RHS) ))
RHS = RHS.subs(h,x/beta).simplify()
display(Math( 'u_{n+1}=' + sp.latex(RHS) ))
print("Avec nos paramètres on obtient bien:")
RHS =
 ↪  RHS.subs({a_21:5/sp.S(24),a_22:1/sp.S(3),a_23:-1/sp.S(24),b_1:1/sp.S(6),b_2:4/sp.S(6),b_3:1/sp.S(6)}).simplif
display(Math( 'u_{n+1}=' + sp.latex(RHS) ))
```

$$K_1 = -\beta u_n$$

$$K_2 = \frac{-\beta^2 a_{21} h u_n - \beta^2 a_{23} h u_n + \beta u_n}{\beta^2 a_{23} h^2 - \beta a_{22} h - 1}$$

$$K_3 = \frac{\beta^3 a_{21} h^2 u_n + \beta^2 a_{22} h u_n - \beta^2 h u_n + \beta u_n}{\beta^2 a_{23} h^2 - \beta a_{22} h - 1}$$

$$u_{n+1} = \frac{u_n \left(-\beta^2 a_{23} h^2 + \beta a_{22} h - \beta h \left(b_1 \left(-\beta^2 a_{23} h^2 + \beta a_{22} h + 1\right) - b_2 \left(\beta a_{21} h + \beta a_{23} h - 1\right) + b_3 \left(\beta^2 a_{21} h^2 + \beta a_{22} h - \beta h + 1\right.\right.\right.}{-\beta^2 a_{23} h^2 + \beta a_{22} h + 1}$$

$$u_{n+1} = \frac{u_n \left(a_{22} x - a_{23} x^2 - x \left(b_1 \left(a_{22} x - a_{23} x^2 + 1\right) - b_2 \left(a_{21} x + a_{23} x - 1\right) + b_3 \left(a_{21} x^2 + a_{22} x - x + 1\right)\right) + 1\right)}{a_{22} x - a_{23} x^2 + 1}$$

Avec nos paramètres on obtient bien:

$$u_{n+1} = \frac{u_n \left(-x^3 + 5x^2 - 16x + 24\right)}{x^2 + 8x + 24}$$

31.3. EXERCICE 3 : étude d'un système Hamiltonien

On s'intèresse au problème de Cauchy

$$\begin{cases} y'(t) = \varphi_1(t, y(t), z(t)), \\ z'(t) = \varphi_2(t, y(t), z(t)), \\ y(0) = y_0, \\ z(0) = z_0 \end{cases}$$

Soit $H\colon (y,z) \mapsto H(y,z)$ une fonction de deux variables régulière et posons $\varphi_1(t,y,z) = \dfrac{\partial H}{\partial z}$ et $\varphi_2(t,y,z) = -\dfrac{\partial H}{\partial y}$.

1. Notons $\mathcal{H}\colon t \mapsto H(y(t), z(t))$. Montrer que $\mathcal{H}$ est constante.
2. Posons $H(y,z) = \dfrac{(ay)^2 + (bz)^2}{2}$ où a et b sont des constantes fixées dans la cellule précédente.
   1. Calculer la solution exacte (les données initiales sont fixées dans la cellule précédente).
   2. Dans la suite on posera $T = 4\pi/(ab)$ et, pour les affichages, on chosira $N+1$ points avec $N = 48$. Dans trois repères différents afficher
      - $t \mapsto y(t)$ et $t \mapsto z(t)$ dans le même répère,
      - $y \mapsto z(y)$,
      - $t \mapsto \mathcal{H}(t)$.
3. Dans la suite on notera $u_n \approx y(t_n)$, $w_n \approx z(t_n)$ et $J_n \approx \mathcal{H}(t_n)$. Calculer la solution approchée obtenue par la méthode suivante appelée méthode d'**Euler symplectique**.

$$\text{(ES)} \quad \begin{cases} u_0 = y_0, \\ w_0 = z_0, \\ u_{n+1} = u_n + h\varphi_1\left(t_n, u_n, w_n\right), \\ w_{n+1} = w_n + h\varphi_2\left(t_{n+1}, u_{n+1}, w_{n+1}\right). \end{cases}$$

   Nota bene : le choix particulier de φ_1 et φ_2 permet de réécrire le schéma sous forme explicite.
   Dans trois répère différents afficher, en comparant solution exacte et solution approchée,
   - $\{(t_i, u_i)\}$ et $\{(t_i, w_i)\}$ dans un même répère,
   - $\{(u_i, w_i)\}$,
   - $\{(t_i, J_i)\}$ où $J_i = H(u_i, w_i)$.

Q1 [1 point] Notons $\mathcal{H}\colon t \mapsto H(y(t), z(t))$. Montrer que $\mathcal{H}$ est constante.

On a

$$\mathcal{H}'(t) = \frac{d}{dt} H(y(t), z(t)) = \frac{\partial H}{\partial y} y'(t) + \frac{\partial H}{\partial z} z'(t) = -z'(t)y'(t) + y'(t)z'(t) = 0.$$

```
prlat= lambda *args: display(Math(''.join( sp.latex(arg) for arg in args )))

nom = "Faccanoni"
prenom = "Gloria"

print("Paramètres de l'exercice 3")
L = [ (-1,0),(1,0),(0,1),(0,-1) ]
idx = sum([ord(c) for c in nom+prenom])%len(L)
y0,z0 = L[idx]
prlat("y_0 = ",y0)
prlat("z_0 = ",z0)

L = [ (1,2),(2,1),(3,2),(2,3),(3,1),(1,3) ]
```

```
idx = sum([ord(c) for c in nom+prenom])%len(L)
(a,b) = L[idx]
prlat("a = ",a)
prlat("b = ",b)
```

```
Paramètres de l'exercice 3
```

y_0 = −1

z_0 = 0

a = 3

b = 1

Q2 [2 point] Posons $H(y,z) = \dfrac{(ay)^2+(bz)^2}{2}$ où a et b sont des constantes fixées dans la cellule précédente.

1. Calculer la solution exacte (les données initiales sont fixées dans la cellule précédente).

2. Dans la suite on posera $T = 4\pi/(ab)$ et, pour les affichages, on chosira $N+1$ points avec $N = 48$. Dans trois répères différents afficher

- $t \mapsto y(t)$ et $t \mapsto z(t)$ dans le même répère,
- $y \mapsto z(y)$,
- $t \mapsto \mathcal{H}(t)$.

Le problème de Cauchy devient

$$\begin{cases} y'(t) = b^2 z(t), \\ z'(t) = -a^2 y(t), \\ y(0) = y_0, \\ z(0) = z_0 \end{cases}$$

```
t0 = 0

tfinal = 4*np.pi/(a*b)
N = 48

phi1 = lambda t,y,z : b**2*z
phi2 = lambda t,y,z : -a**2*y
Ham = lambda y,z : ((a*y)**2+(b*z)**2)/2
```

La solution exacte est

```
_h = sp.Symbol('h',positive=True)

_phi1 = lambda t,y,z :  sp.diff(Ham(y,z),z)
_phi2 = lambda t,y,z : -sp.diff(Ham(y,z),y)

t = sp.Symbol('t')
y = sp.Function('y')
z = sp.Function('z')

edo1 = sp.Eq( sp.diff(y(t),t) , _phi1(t,y(t),z(t)) )
edo2 = sp.Eq( sp.diff(z(t),t) , _phi2(t,y(t),z(t)) )
# display(edo1)
# display(edo2)
solpar_1, solpar_2 = sp.dsolve([edo1,edo2],[y(t),z(t)], ics={y(t0):y0, z(t0):z0})
```

```
display(solpar_1,solpar_2)
```

$y(t) = -\cos(3t)$

$z(t) = 3\sin(3t)$

Les trois graphes sont

```
func_1 = sp.lambdify(t,solpar_1.rhs,'numpy')
func_2 = sp.lambdify(t,solpar_2.rhs,'numpy')

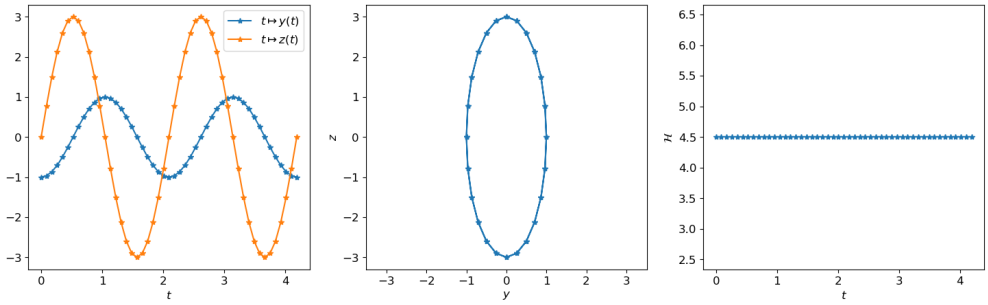
tt = np.linspace(t0,tfinal,N+1)
h  = tt[1]-tt[0]

plt.figure(figsize=(18,5))
yy = func_1(tt)
zz = func_2(tt)
HH = [ Ham(y,z) for (y,z) in zip(yy,zz) ]

plt.subplot(1,3,1)
plt.plot(tt,yy,'-*',tt,zz,'-*')
plt.legend([r'$t\mapsto y(t)$',r'$t\mapsto z(t)$'])
plt.xlabel(r'$t$')

plt.subplot(1,3,2)
plt.plot(yy,zz,'-*')
plt.xlabel(r'$y$')
plt.ylabel(r'$z$')
plt.axis('equal')

plt.subplot(1,3,3)
plt.plot(tt,HH,'-*')
plt.xlabel(r'$t$')
plt.ylabel(r'$\mathcal{H}$')
plt.axis('equal');
```



Q3 [4 points]
Dans la suite on notera $u_n \approx y(t_n)$, $w_n \approx z(t_n)$ et $J_n \approx \mathcal{H}(t_n)$.

Calculer la solution approchée obtenue par la méthode suivante appelée méthode d'**Euler symplectique**.

$$(\text{ES}) \begin{cases} u_0 = y_0, \\ w_0 = z_0, \\ u_{n+1} = u_n + h\varphi_1(t_n, u_n, w_n), \\ w_{n+1} = w_n + h\varphi_2(t_{n+1}, u_{n+1}, w_{n+1}). \end{cases}$$

Nota bene : le choix particulier de φ_1 et φ_2 permet de réécrire le schéma sous forme explicite.

Dans trois répère différents afficher, en comparant solution exacte et solution approchée,

- $\{(t_i, u_i)\}$ et $\{(t_i, w_i)\}$ dans un même répère,
- $\{(u_i, w_i)\}$,
- $\{(t_i, J_i)\}$ où $J_i = H(u_i, w_i)$.

Pour l'affichage des solutions exactes VS approchées, on peut écrire une fonction qui prend en paramètres la discrétisation `tt`, les deux listes solution exacte `yy` et `zz`, les deux listes solution approchée `uu` et `ww`, puis une chaîne de caractères `s` qui contient le nom du schéma, et pour finir `HH` la liste qui contient le calcul de l'Hamiltonien exact et `HH_num` celui approché.

```
def affichage(tt,yy,zz,uu,ww,s,HH,HH_num):

    plt.subplot(1,3,1)
    plt.plot(tt,uu,'o--',tt,ww,'d--')
    plt.plot(tt,yy,tt,zz)
    plt.xlabel('t')
    plt.legend([r'$u(t)$ approchée',r'$w(t)$ approchée','$y(t)$ exacte','$z(t)$ exacte'])
    plt.title(f'{s} - y(t) et z(t)')
    plt.grid()

    plt.subplot(1,3,2)
    plt.plot(uu,ww,'o--')
    plt.plot(yy,zz)
    plt.xlabel('y')
    plt.ylabel('z')
    plt.legend(['Approchée','Exacte'])
    plt.title(f'{s} - z(y)')
    plt.grid()
    plt.axis('equal')

    plt.subplot(1,3,3)
    plt.plot(tt,HH_num,'o--')
    plt.plot(tt,HH)
    plt.xlabel('t')
    plt.ylabel(r'$\mathcal{H}$')
    plt.legend(['Approchée','Exacte'])
    plt.title(f'{s} - '+r'$\mathcal{H}(t)$')
    plt.grid()
    plt.axis('equal');
```

Le schéma se réécrit

$$(\text{ES}) \quad \begin{cases} u_0 = y_0, \\ w_0 = z_0, \\ u_{n+1} = u_n + hb^2 w_n, \\ w_{n+1} = -a^2 h u_n + (1-(abh)^2) w_n. \end{cases}$$

et

$$J_{n+1} = \frac{(au_{n+1})^2 + (bw_{n+1})^2}{2}$$

Vérifions nos calculs :

```
_h = sp.Symbol('h')
t_n = sp.Symbol('t_n')
u_n = sp.Symbol('u_n')
w_n = sp.Symbol('w_n')
t_np1 = t_n+h
u_np1 = sp.Symbol('u_{n+1}')
w_np1 = sp.Symbol('w_{n+1}')
```

```
eq1 = sp.Eq( u_np1 , u_n+_h*_phi1(t_n,u_n,w_n))
eq2 = sp.Eq( w_np1 , w_n+_h*_phi2(t_np1,u_np1,w_np1))
sol = sp.solve([eq1,eq2],[u_np1,w_np1])
prlat( 'u_{n+1}=' , sol[u_np1] )
prlat( 'w_{n+1}=' , sol[w_np1].factor(w_n) )
# prlat( 'H_{n+1}=' , _Ham(sol[u_np1],sol[w_np1]).expand().simplify() )
```

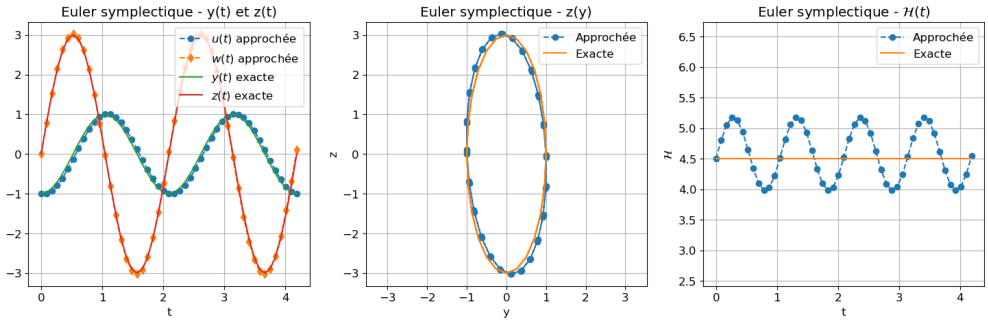
$$u_\{n+1\}=hw_n + u_n$$

$$w_\{n+1\}= - 9hu_n - w_n \left(9h^2 - 1\right)$$

```
# x0, y0, h variables globales
def ES(phi1,phi2,tt):
    h = tt[1]-tt[0]
    uu = np.zeros_like(tt)
    ww = np.zeros_like(tt)
    uu[0] = y0
    ww[0] = z0
    for n in range(len(tt)-1):
        # uu[n+1] = uu[n]+h*phi1(tt[n],uu[n],ww[n])
        # ww[n+1] = ww[n]+h*phi2(tt[n+1],uu[n+1],None)
        # idem que
        uu[n+1] = uu[n]+h*b**2*ww[n]
        ww[n+1] = -a**2*h*uu[n]+(1-(a*b*h)**2)*ww[n]
    return [uu,ww]

[uu, ww] = ES(phi1,phi2,tt)
HH_num = Ham(uu,ww)

plt.figure(figsize=(18,5))
affichage(tt,yy,zz,uu,ww,"Euler symplectique",HH,HH_num)
```



Chapitre 32.

2022 Contrôle Terminal session 2

- 2H30
- Deposer votre notebook .ipynb sur Moodle (dépôt de devoir)

```
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp
from scipy.optimize import fsolve
from scipy.integrate import solve_ivp, odeint
from IPython.display import display, Markdown, Math

prlat= lambda *args: display(Math(''.join( sp.latex(arg) for arg in args )))
```

32.1. EXERCICE 1 : Méthode d'Euler exponentielle et problèmes raides

Soient $b, g > 0$ deux constantes et considérons le problème de Cauchy

$$\begin{cases} y'(t) = -by(t) + g, & t \in [0;1] \\ y(0) = 1 + 10^{-8}. \end{cases}$$

1. Calculer et tracer le graphe de la solution exacte $t \mapsto y(t)$ d'abord pour $b = g = 10$ puis pour $b = g = 100$.
2. Méthode d'**Euler exponentielle** La méthode d'Euler exponentielle appliquée à ce problème de Cauchy s'écrit comme suit :

$$\text{(EEx)} \quad \begin{cases} u_0 = y_0, \\ u_{n+1} = e^{-bh} u_n + ghf(-bh), & n = 0, \dots, N-1 \end{cases}$$

où $f(z) = \frac{e^z - 1}{z}$.

- Vérifier que cette méthode génère une suite arithmético-géométrique et montrer analytiquement qu'elle calcule la solution exacte discrète.
- Soit $b = g = 100$. Tracer la solution exacte et les solutions approchées obtenues avec $N = 10$, $N = 50$, $N = 100$.

Q1 [2 points] Solution exacte
Calculer et tracer le graphe de la solution exacte $t \mapsto y(t)$ d'abord pour $b = g = 10$ puis pour $b = g = 100$.

Il s'agit d'une edo linéaire d'ordre 1 dont la solution est

$$y(t) = \left(y_0 - \frac{g}{b}\right) e^{-bt} + \frac{g}{b} \xrightarrow[t \to +\infty]{} y(t) = \frac{g}{b} \text{ pour tout } y(0) = y_0 \in \mathbb{R}$$

Le problème est numériquement bien posé, c'est-à-dire que l'erreur sur la solution sera au pire de l'ordre de l'erreur sur la condition initiale. Cependant, lorsque b devient de plus en plus grand, la solution devient de plus en plus raide : il s'agit d'un problème *stiff*.

Vérifions ces calculs avec `sympy` :

```
t = sp.symbols('t')
y = sp.Function('y')
b = sp.symbols('b', positive=True)
g = sp.symbols('g', positive=True)
y0 = sp.symbols('y_0')
edo = sp.Eq(y(t).diff(t), -b*y(t)+g)
IC = {y(0): y0}
display(edo)
display(sp.Eq(y(0), y0))
sol = sp.dsolve(edo, y(t), ics=IC)
display(sol)
lim = sp.limit(sol.rhs, t, sp.oo)
display(sp.Eq(sp.Limit(y(t), t, sp.oo), lim))
```

$$\frac{d}{dt}y(t) = -by(t) + g$$

$$y(0) = y_0$$

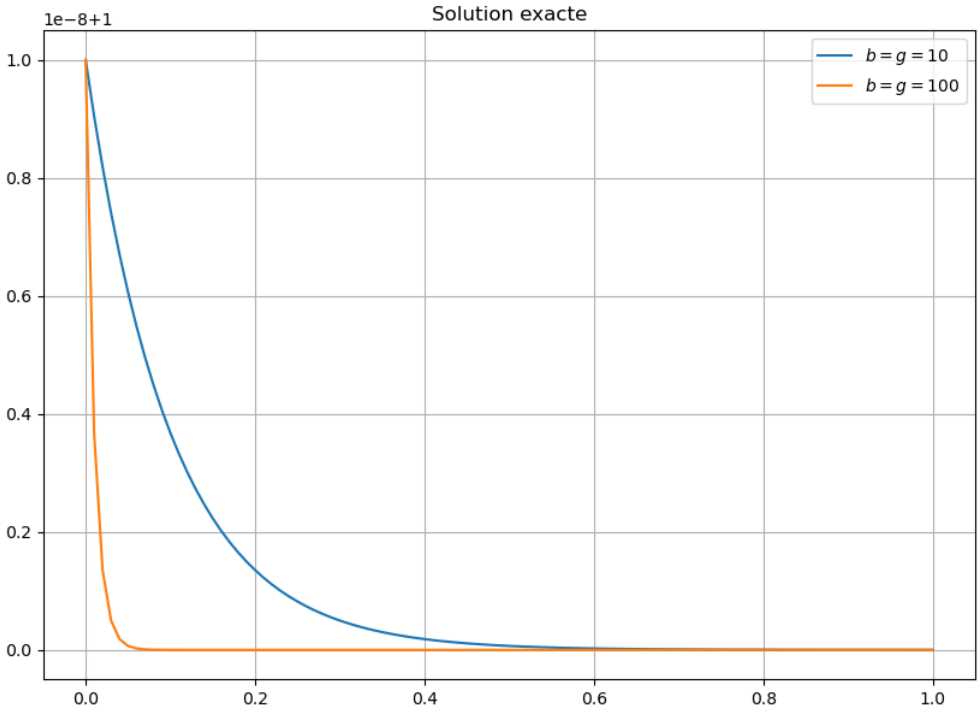
$$y(t) = \frac{g}{b} + \frac{(by_0 - g)\, e^{-bt}}{b}$$

$$\lim_{t \to \infty} y(t) = \frac{g}{b}$$

```
# ========================================================
# solution exacte
# ========================================================
exacte = sp.lambdify((t,b,g,y0), sol.rhs, 'numpy')
# exacte = lambda t,b,g,y0 : (y0-g/b)*np.exp(-b*t)+g/b

# ========================================================
# variables globales
# ========================================================
t0 = 0
tfinal = 1
y0 = 1+1.e-8

# ========================================================
# affichage
# ========================================================
tt = np.linspace(t0,tfinal,101)
plt.figure(figsize=(10,7))
for b in [10,100] :
    plt.plot( tt , exacte(tt,b,b,y0), label=rf'$b=g={b}$');
plt.legend()
plt.grid()
plt.title('Solution exacte');
```



Q2 [**4 points**] Méthode d'**Euler exponentielle** La méthode d'Euler exponentielle appliquée à ce problème de Cauchy s'écrit comme suit :

$$\text{(EEx)} \quad \begin{cases} u_0 = y_0, \\ u_{n+1} = e^{-bh} u_n + ghf(-bh), \quad n = 0, \dots, N-1 \end{cases}$$

où $f(z) = \frac{e^z - 1}{z}$.
1. Vérifier que cette méthode génère une suite arithmético-géométrique et montrer analytiquement qu'elle calcule la solution exacte discrète. 1. Soit $b = g = 100$. Tracer la solution exacte et les solutions approchées obtenues avec $N = 10$, $N = 50$, $N = 100$.

La méthode d'Euler exponentielle donne la suite définie par récurrence

$$\begin{cases} u_0 = y_0, \\ u_{n+1} = Au_n + B \end{cases} \qquad \text{avec } A = \exp(-bh),\ B = (\exp(-bh) - 1)\frac{g}{b} = (A-1)\frac{g}{b}$$

Donc

$$u_{n+1} = Au_n + (A-1)\frac{g}{b} = A\left(u_n + \frac{g}{b}\right) - \frac{g}{b}$$

soit encore

$$u_{n+1} + \frac{g}{b} = A\left(u_n + \frac{g}{b}\right).$$

Si on note $w_n = u_n + \frac{g}{b}$, on a

$$w_{n+1} = Aw_n = A^{n+1} w_0$$

ainsi

$$u_n - \frac{g}{b} = (\exp(-hb))^n \left(y_0 - \frac{g}{b}\right).$$

Comparons la solution exacte et la solution approchée :

$$y_n = \frac{g}{b} + \left(y_0 - \frac{g}{b}\right) e^{-bt_n}$$
$$u_n = \frac{g}{b} + \left(y_0 - \frac{g}{b}\right) e^{-bhn}$$

Comme $t_n = t_0 + nh = nh$, la solution approchée est exacte.

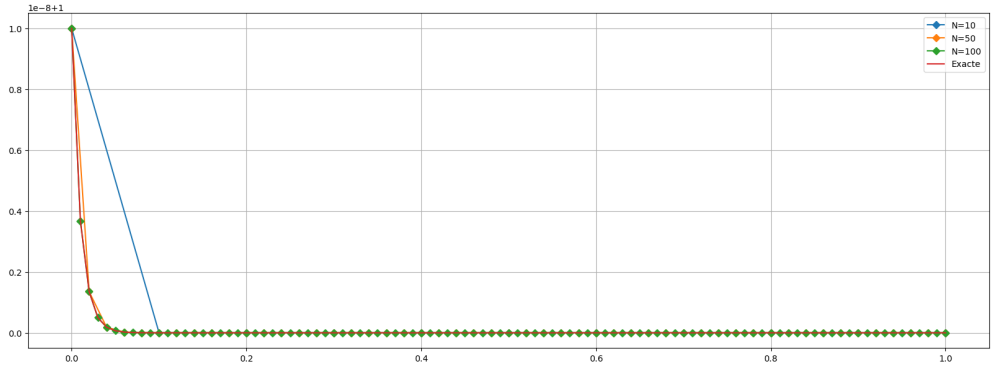
```
b = 100
g = b

# ======================================================
# SCHÉMA
# ======================================================

def EEx(tt,y0):
    uu = np.zeros_like(tt)
    uu[0] = y0
    h = tt[1]-tt[0]
    for n in range(len(tt)-1):
        K1 = (np.exp(-b*h)-1)/(-b*h)
        uu[n+1] = np.exp(-b*h)*uu[n]+g*h*K1
    return uu


# ======================================================
# CALCUL SOLUTION APPROCHEE
# ======================================================
plt. figure(1,figsize=(20,7))
for N in [10,50,100]:
    tt = np.linspace(t0,tfinal,N+1)
    uu = EEx(tt,y0)
    plt.plot(tt,uu,'-D',label=f'{N=}')

yy = exacte(tt,b,b,1+1.e-8)
plt.plot(tt,yy,'-',label='Exacte')
plt.legend()
plt.grid()
```



32.2. EXERCICE 2 : étude d'un schéma Runge-Kutta

Considérons le schéma dont la matrice de Butcher est donnée ci-dessous :

$$\begin{array}{c|cccc} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{2}{3} & \frac{1}{6} & \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & \vartheta & 0 \\ 1 & \frac{3}{2} & -\frac{3}{2} & \frac{1}{2} & \frac{1}{2} \\ \hline & \frac{3}{2} & \gamma & \frac{1}{2} & \frac{1}{2} \end{array}$$

1. Calculer γ et ϑ pour que le schéma soit consistant. Dans la suite on posera γ et ϑ égales à ces valeurs.
2. Étudier empiriquement l'ordre de convergence sur le problème de Cauchy

$$\begin{cases} y'(t) = -\dfrac{y(t)}{2}\left(1 - \dfrac{y(t)}{3}\right), & \forall t \in I = [0, 10], \\ y(0) = 1, \end{cases}$$

3. Étudier théoriquement l'ordre de convergence du schéma.

Q1 [1 points] Calculer γ et ϑ pour que le schéma soit consistant. Dans la suite on posera γ et ϑ égales à ces valeurs.

C'est un schéma semi-implicite à 4 étages ($s = 4$).

Pour qu'il soit consistant il faut que

$$\begin{cases} \displaystyle\sum_{j=1}^{s} b_i = 1 \\ c_i = \displaystyle\sum_{j=1}^{s} a_{ij} \quad i = 1, \dots, s \end{cases}$$

- Comme $\displaystyle\sum_{j=1}^{s} b_i = \frac{3}{2} + \gamma + \frac{1}{2} + \frac{1}{2} = \frac{5}{2} + \gamma$, il faut poser $\gamma = -\dfrac{3}{2}$.
- La deuxième condition est satisfaite ssi $\dfrac{1}{2} = -\dfrac{1}{2} + \dfrac{1}{2} + \vartheta$, il faut donc poser $\vartheta = \dfrac{1}{2}$.

Q2 [5 points] Étudier empiriquement l'ordre de convergence sur le problème de Cauchy

$$\begin{cases} y'(t) = -\dfrac{y(t)}{2}\left(1 - \dfrac{y(t)}{3}\right), & \forall t \in I = [0, 10], \\ y(0) = 1, \end{cases}$$

```
# variables globales
t0     = 0
tfinal = 10
y0     = 1

phi = lambda t,y: -y/2*(1-y/3)

################################################
# solution exacte
################################################
t   = sp.Symbol('t')
y   = sp.Function('y')
edo = sp.Eq( sp.diff(y(t),t) , phi(t,y(t)) )
solpar = sp.dsolve(edo, y(t), ics={y(t0):y0}).simplify()
display(solpar)

sol_exacte = sp.lambdify(t,solpar.rhs,'numpy')

################################################
# solution approchée
################################################
def RK(tt,phi,y0):
    h = tt[1]-tt[0]
    uu =np.zeros_like(tt)
    uu[0] = y0
    for n in range(len(tt)-1):
        sys = lambda X : [ -X[0]+phi( tt[n]+h/2   ,uu[n]+h/2*X[0]                       ) ,
                           -X[1]+phi( tt[n]+h*2/3 ,uu[n]+h/6*(X[0]+3*X[1])              ),
                           -X[2]+phi( tt[n]+h/2   ,uu[n]+h/2*(-X[0]+X[1]+X[2])          ),
                           -X[3]+phi( tt[n+1]     ,uu[n]+h/2*(3*X[0]-3*X[1]+X[2]+X[3]) )
```

```
                            ]
        K_start = phi(tt[n],uu[n])
        K1, K2, K3, K4 = fsolve( sys , [K_start,K_start,K_start,K_start] )
        uu[n+1] = uu[n]+h/2*(3*K1-3*K2+K3+K4)
    return uu


##############################################
# ordre
##############################################
H = []
err = []
N = 10

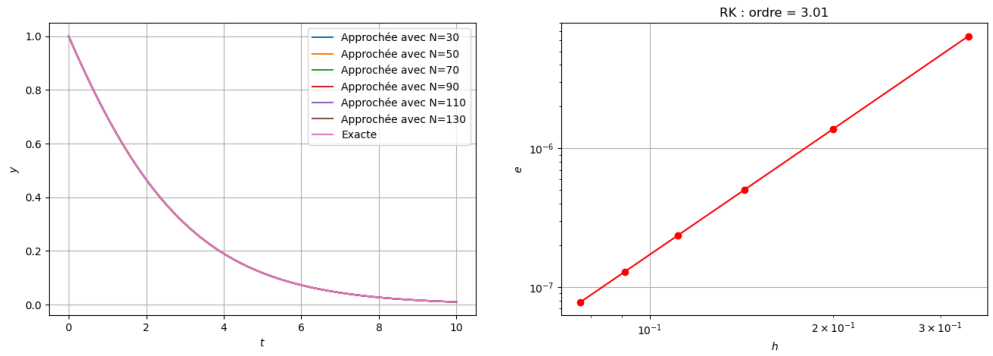
_, (ax1,ax2) = plt.subplots(nrows=1,ncols=2,figsize=(16,5))

for k in range(6):
    N += 20
    tt = np.linspace(t0, tfinal, N + 1)
    H.append( tt[1] - tt[0] )
    yy = sol_exacte(tt)
    uu = RK(tt,phi,y0)
    err.append( np.linalg.norm(uu-yy,np.inf) )
    ax1.plot(tt,uu,label=f'Approchée avec N={N}')

ax1.plot(tt,yy,label='Exacte')
ax1.set_xlabel('$t$')
ax1.set_ylabel('$y$')
ax1.grid(True)
ax1.legend()

ax2.loglog( H, err, 'r-o')
ax2.set_xlabel('$h$')
ax2.set_ylabel('$e$')
ax2.set_title(f'RK : ordre = {np.polyfit(np.log(H),np.log(err),1)[0]:1.2f}')
ax2.grid(True)
```

$$y(t) = \frac{3 \cdot \left(2\sqrt{e^t} - 1\right)}{4e^t - 1}$$



Q3 [3 points] Étudier théoriquement l'ordre de convergence du schéma.

Soit ω l'ordre de la méthode.

C'est un schéma semi-implicite à $s = 4$ étages donc $\omega \leq 2s = 8$.

Pour $\vartheta = \frac{1}{2}$ et $\gamma = \frac{3}{2}$, $\omega \geq 1$.

Si, de plus, $\sum_{j=1}^{s} b_j c_j = \frac{1}{2}$ alors $\omega \geq 2$

Si, de plus,

$$\begin{cases} \sum_{j=1}^{s} b_j c_j^2 = \frac{1}{3} \\ \sum_{i=1}^{s} \sum_{j=1}^{s} b_i a_{ij} c_j = \frac{1}{6} \end{cases}$$

alors $\omega \geq 3$

Si, de plus,

$$\begin{cases} \sum_{j=1}^{s} b_j c_j^3 = \frac{1}{4} \\ \sum_{i=1}^{s} \sum_{j=1}^{s} b_i c_i a_{ij} c_j = \frac{1}{8} \\ \sum_{i=1}^{s} \sum_{j=1}^{s} b_i a_{ij} c_j^2 = \frac{1}{12} \\ \sum_{i=1}^{s} \sum_{j=1}^{s} \sum_{k=1}^{s} b_i a_{ij} a_{jk} c_k = \frac{1}{24} \end{cases}$$

alors $\omega \geq 4$

Calculons donc toutes ces sommes :

```
def ordre_RK(s, A=None, b=None, c=None):

    j = sp.symbols('j')

    if A is None:  A = sp.MatrixSymbol('a', s, s)
    else:          A = sp.Matrix(A)
    if c is None:  c = sp.symbols(f'c_0:{s}')
    if b is None:  b = sp.symbols(f'b_0:{s}')


    display(Markdown("**Matrice de Butcher**"))
    But = matrice_Butcher(s, A, b, c)

    Eqs = []

    # display(Markdown(f"**On a {s+1} conditions pour avoir consistance = pour être d'ordre 1**"))
    # ordre_1(s, A, b, c, j)

    display(Markdown("**On doit ajouter 1 condition pour être d'ordre 2**"))
    Eq_2 = ordre_2(s, A, b, c, j)
    Eqs.append(Eq_2)

    display(Markdown("**On doit ajouter 2 conditions pour être d'ordre 3**"))
    Eq_3 = ordre_3(s, A, b, c, j)
    Eqs.extend(Eq_3)

    display(Markdown("**On doit ajouter 4 conditions pour être d'ordre 4**"))
    Eq_4 = ordre_4(s, A, b, c, j)
    Eqs.extend(Eq_4)

    return Eqs


def matrice_Butcher(s, A, b, c):
    But = sp.Matrix(A)
    But = But.col_insert(0, sp.Matrix(c))
    last = [sp.Symbol(" ")]
    last.extend(b)
    But = But.row_insert(s,sp.Matrix(last).T)
    display(Math(sp.latex(sp.Matrix(But))))
```

```
    return But


def ordre_1(s, A, b, c, j):
    texte = r"\sum_{j=1}^s b_j =" + f"{sum(b).simplify()}"
    texte += r"\text{ doit être égale à }1"
    display(Math(texte))
    for i in range(s):
        somma = sp.summation(A[i,j],(j,0,s-1)).simplify()
        texte = r'\sum_{j=1}^s a_{'+str(i)+r'j}=' + sp.latex( somma )
        texte += r"\text{ doit être égale à }"+sp.latex(c[i])
        display(Math( texte ))
    return None


def ordre_2(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j=' +sp.latex(sum([b[i]*c[i] for i in range(s)]).simplify())
    texte += r"\text{ doit être égale à }\frac{1}{2}"
    display(Math(texte))
    Eq_2 = sp.Eq( sum([b[i]*c[i] for i in range(s)]).simplify(), sp.Rational(1,2) )
    return Eq_2


def ordre_3(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j^2='
    texte += sp.latex( sum([b[i]*c[i]**2 for i in range(s)]).simplify() )
    texte += r"\text{ doit être égale à }\frac{1}{3}"
    display(Math(texte))
    Eq_3_1 = sp.Eq( sum([b[i]*c[i]**2 for i in range(s)]).simplify(), sp.Rational(1,3) )
    texte = r'\sum_{i,j=1}^s b_i a_{ij} c_j='
    somma = sum([b[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{6}"
    display(Math(texte))
    Eq_3_2 = sp.Eq( sum([b[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,6) )
    return [Eq_3_1, Eq_3_2]


def ordre_4(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j^3='
    texte += sp.latex( sum([b[i]*c[i]**3 for i in range(s)]).simplify() )
    texte += r"\text{ doit être égale à }\frac{1}{4}"
    display(Math(texte))
    Eq_4_1 = sp.Eq( sum([b[i]*c[i]**3 for i in range(s)]).simplify(), sp.Rational(1,4) )
    texte = r'\sum_{i,j=1}^s b_i c_i a_{ij} c_j='
    somma = sum([b[i]*c[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{8}"
    display(Math(texte))
    Eq_4_2 = sp.Eq( sum([b[i]*c[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,8) )
    texte = r'\sum_{i,j=1}^s b_i a_{ij} c_j^2='
    somma = sum([b[i]*A[i,j]*c[j]**2 for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{12}"
    display(Math(texte))
    Eq_4_3 = sp.Eq( sum([b[i]*A[i,j]*c[j]**2 for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,12) )
    texte = r'\sum_{i,j,k=1}^s b_i a_{ij} a_{jk} c_k='
    somma = sum([b[i]*A[i,j]*A[j,k]*c[k] for k in range(s) for j in range(s) for i in
↪   range(s)]).simplify()
    texte = texte + sp.latex(somma)
```

```
    texte += r"\text{ doit être égale à }\frac{1}{24}"
    display(Math(texte))
    Eq_4_4 = sp.Eq( sum([b[i]*A[i,j]*A[j,k]*c[k] for k in range(s) for j in range(s) for i in
↪   range(s)]).simplify(), sp.Rational(1,24) )
    return [Eq_4_1, Eq_4_2, Eq_4_3, Eq_4_4]



##################################################
# Conditions avec s étages
##################################################

c = [sp.Rational(1,2), sp.Rational(2,3), sp.Rational(1,2), 1]
b = [ sp.Rational(3,2), -sp.Rational(3,2), sp.Rational(1,2), sp.Rational(1,2)]
A = [ [sp.Rational(1,2), 0, 0, 0],
      [sp.Rational(1,6), sp.Rational(1,2), 0, 0],
      [-sp.Rational(1,2), sp.Rational(1,2), sp.Rational(1,2), 0],
      [sp.Rational(3,2), -sp.Rational(3,2), sp.Rational(1,2), sp.Rational(1,2)]]
s = len(c)


Eqs = ordre_RK(s, A, b, c)

# for eq in Eqs:
#     display(eq)
```

Matrice de Butcher

$$\begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{2}{3} & \frac{1}{6} & \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & \frac{1}{2} & 0 \\ 1 & \frac{3}{2} & -\frac{3}{2} & \frac{1}{2} & \frac{1}{2} \\ & \frac{3}{2} & -\frac{3}{2} & \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

On doit ajouter 1 condition pour être d'ordre 2

$$\sum_{j=1}^{s} b_j c_j = \frac{1}{2} \text{ doit être égale à } \frac{1}{2}$$

On doit ajouter 2 conditions pour être d'ordre 3

$$\sum_{j=1}^{s} b_j c_j^2 = \frac{1}{3} \text{ doit être égale à } \frac{1}{3}$$

$$\sum_{i,j=1}^{s} b_i a_{ij} c_j = \frac{1}{6} \text{ doit être égale à } \frac{1}{6}$$

On doit ajouter 4 conditions pour être d'ordre 4

$$\sum_{j=1}^{s} b_j c_j^3 = \frac{11}{36} \text{ doit être égale à } \frac{1}{4}$$

$$\sum_{i,j=1}^{s} b_i c_i a_{ij} c_j = \frac{5}{48} \text{ doit être égale à } \frac{1}{8}$$

$$\sum_{i,j=1}^{s} b_i a_{ij} c_j^2 = \frac{5}{72} \text{ doit être égale à } \frac{1}{12}$$

$$\sum_{i,j,k=1}^{s} b_i a_{ij} a_{jk} c_k = \frac{1}{48} \text{ doit être égale à } \frac{1}{24}$$

D'après ces résultats, le schéma est d'ordre 3.

32.3. EXERCICE 3 : étude d'un schéma multipas

On notera $\varphi_k \overset{\text{déf}}{=} \varphi(t_k, u_k)$.
Considérons les deux méthodes multipas

$$u_{n+1} = \frac{18}{11}u_n - \frac{9}{11}u_{n-1} + \frac{2}{11}u_{n-2} + h\frac{6}{11}\varphi_{n+1}$$

$$u_{n+1} = \frac{18}{11}u_n - \frac{9}{11}u_{n-1} + \frac{3}{11}u_{n-2} + h\frac{6}{11}\varphi_{n+1}$$

1. Parmi ces deux méthode, indiquer laquelle est consistente (justifier la réponse). Dans la suite on n'étudiera que cette méthode.
2. Étudier la zéro-stabilité de la méthode.
3. Étudier théoriquement l'ordre de la méthode.
4. Vérifier empiriquement l'ordre de convergence sur le problème de Cauchy

$$\begin{cases} y'(t) = -\dfrac{y(t)}{2}\left(1 - \dfrac{y(t)}{3}\right), \quad \forall t \in I = [0, 10], \\ y(0) = 1, \end{cases}$$

Q1 [1 points] Choisir la méthode consistente (justifier la réponse). Dans la suite on n'étudiera que cette méthode.

- On a $p = 2$: c'est des méthodes à $q = p + 1 = 3$ pas.
- Les méthodes sont implicites car $b_{-1} = \frac{6}{11} \neq 0$.

Pour que la méthode soit consistante il faut que

$$\begin{cases} \xi(0) \overset{\text{def}}{=} \displaystyle\sum_{j=0}^{p} a_j = a_0 + a_1 = 1, \\ \xi(1) \overset{\text{def}}{=} \displaystyle\sum_{j=0}^{p} (-j)a_j + \sum_{j=-1}^{p} b_j = 1. \end{cases}$$

Seul la première des méthodes est consistante car

$$\frac{18}{11} - \frac{9}{11} + \frac{2}{11} = 1$$

tandis que

$$\frac{18}{11} - \frac{9}{11} + \frac{3}{11} = \frac{12}{11} \neq 1.$$

Q2 [2 points] Étudier la zéro-stabilité.

Le premier polynôme caractéristique est

$$\varrho(r) = r^{p+1} - \sum_{j=0}^{p} a_j r^{p-j} = r^3 - \frac{18}{11}r^2 + \frac{9}{11}r - \frac{2}{11} = \frac{1}{11}(r-1)(11r^2 - 7r + 2).$$

Calculons ses racines avec `sympy`.

```
aa = [sp.Rational(18,11), sp.Rational(-9,11), sp.Rational(2,11)]
q = len(aa)

r = sp.Symbol('r')
rho = sp.Function(r'\varrho')
p = q-1
rho_gen =  sp.poly(r**(p+1) - sum( aa[j] * r**(p-j) for j in range(p+1)), r)
eq = sp.Eq( rho(r), rho_gen.as_expr() )

display(Markdown(f"Le premier polynome caractéristique est :"))
display(Math( sp.latex(eq,order='old') ) )

display(Markdown(f"Ses racines sont :"))
racines = sp.solve(rho_gen,r)
display(racines)
display(Markdown(f"dont les modules valent :"))
display([abs(rac).evalf() for rac in racines])
```

Le premier polynome caractéristique est :

$$\varrho(r) = -\frac{2}{11} + \frac{9r}{11} - \frac{18r^2}{11} + r^3$$

Ses racines sont :

```
[1, 7/22 - sqrt(39)*I/22, 7/22 + sqrt(39)*I/22]
```

dont les modules valent :

```
[1.00000000000000, 0.426401432711221, 0.426401432711221]
```

Les racines sont

$$r_0 = 1, \quad r_{1,2} = \frac{7 \pm \sqrt{-39}}{22}.$$

La méthode est donc zéro-stable car

$$\begin{cases} |r_j| \leq 1 \quad \text{pour tout } j = 0, 1, 2 \\ \varrho'(r_j) \neq 0 \text{ si } |r_j| = 1. \end{cases}$$

Q3 [3 points] Étudier théoriquement l'ordre.

La première barrière de Dahlquist affirme qu'un schéma implicite à $q = 3$ pas consistante et zéro-stable peut être au mieux d'ordre $\omega = q + 1 = 4$.

Pour que la méthode soit - au moins d'ordre 2 il faut qu'elle soit consistante et que $\sum_{j=0}^{p}(-j)^2 a_j + 2\sum_{j=-1}^{p}(-j)^1 b_j = 1$, - au moins d'ordre 3 il faut qu'elle soit au moins d'ordre 2 et que $\sum_{j=0}^{p}(-j)^3 a_j + 3\sum_{j=-1}^{p}(-j)^2 b_j = 1$, - au moins d'ordre 4 il faut qu'elle soit au moins d'ordre 2 et que $\sum_{j=0}^{p}(-j)^4 a_j + 4\sum_{j=-1}^{p}(-j)^3 b_j = 1$.

Dans notre cas - la méthode est consistante ; - pour que la méthode sois d'ordre au moins 2 il faut de plus $1 = a_1 + 4a_2 - 2(b_1 + 2b_2 - b_{-1})$ ce qui est satisfait ; - pour que la méthode sois d'ordre au moins 3 il faut de plus $1 = -a_1 - 8a_2 + 3(b_1 + 4b_2 + b_{-1})$ ce qui est satisfait ; - pour que la méthode sois d'ordre au moins 4 il faut de plus $1 = a_1 + 16a_2 - 4(b_1 + 8b_2 - b_{-1})$ ce qui n'est pas satisfait.

La méthode est donc d'ordre 3 (cf. validation avec `sympy` ci-dessous).

```
# j = 0...p
# q = p+1 # nb de pas
#   # ordre de la méthode

CAS_GENERAL = True
p = 2

q = p+1
ordre_max = q+2 if q%2==0 else q+1

print(f"{' '*70}\nC'est une méthode à q = {q} pas d'ordre    {ordre_max}")


if CAS_GENERAL:
    aa = sp.symbols(f'a_0:{q}')
    bb = sp.symbols(f'b_0:{q}')
    bm1 = sp.Symbol('b_{-1}')
else : # cas particulier
    aa = [sp.Rational(18,11), -sp.Rational(9,11),sp.Rational(2,11)]
    bb = [0,0,0]
    bm1 = sp.Rational(6,11)

i=1
sa=sum( [-j*aa[j] for j in range(len(aa))] )
sb=bm1+sum( [bb[j] for j in range(len(bb))] )
print(f"{' '*70}\nLa méthode est d'ordre    {i} (= consistante) si ")
prlat(sum(aa).factor(),"=1" )
prlat((sa).factor()+(i*sb).factor(),"=1")

for i in range(2,ordre_max+1):
    sa=sum( [(-j)**i*aa[j] for j in range(len(aa))] )
    sb=bm1+sum( [(-j)**(i-1)*bb[j] for j in range(1,len(bb))] )
    print(f"{' '*70}\nLa méthode est d'ordre    {i} si elle est d'ordre {i-1} et, de plus, ")
    prlat((sa).factor()+(i*sb).factor(),"=1" )
```

```
C'est une méthode à q = 3 pas d'ordre    4

La méthode est d'ordre    1 (= consistante) si
```

$$a_0 + a_1 + a_2 = 1$$
$$-a_1 - 2a_2 + b_0 + b_1 + b_2 + b_{-1} = 1$$

```
La méthode est d'ordre    2 si elle est d'ordre 1 et, de plus,
```

$$a_1 + 4a_2 - 2\left(b_1 + 2b_2 - b_{-1}\right) = 1$$

```
La méthode est d'ordre    3 si elle est d'ordre 2 et, de plus,
```

$$-a_1 - 8a_2 + 3\left(b_1 + 4b_2 + b_{-1}\right) = 1$$

```
La méthode est d'ordre    4 si elle est d'ordre 3 et, de plus,
```

$$a_1 + 16a_2 - 4\left(b_1 + 8b_2 - b_{-1}\right) = 1$$

Q4 [3 points] Vérifier empiriquement l'ordre de convergence sur le problème de Cauchy

$$\begin{cases} y'(t) = -\dfrac{y(t)}{2}\left(1 - \dfrac{y(t)}{3}\right), & \forall t \in I = [0, 10], \\ y(0) = 1, \end{cases}$$

```
# variables globales
t0     = 0
tfinal = 10
y0     = 1

phi = lambda t,y: -y/2*(1-y/3) #-y-t**2

################################################
# solution exacte
################################################
t   = sp.Symbol('t')
y   = sp.Function('y')
edo = sp.Eq( sp.diff(y(t),t) , phi(t,y(t)) )
solpar = sp.dsolve(edo, y(t), ics={y(t0):y0})
display(solpar)

sol_exacte = sp.lambdify(t,solpar.rhs,'numpy')

################################################
# schéma (initialisation avec sol exacte pour ordre de convergence)
################################################
def multipas(phi, tt, sol_exacte):
    h  = tt[1] - tt[0]
    uu = [sol_exacte(tt[i]) for i in range(3)]
    for i in range(2,len(tt) - 1):
        eq = lambda x : -x+18/11*uu[i]-9/11*uu[i-1]+2/11*uu[i-2]+h*6/11*phi(tt[i+1],x)
        temp = fsolve( eq ,uu[i])
        uu.append( temp[0] )
    return uu

################################################
# ordre
################################################
H = []
err = []
N = 10

plt.figure(figsize=(16,5))
ax1 = plt.subplot(1,2,1)

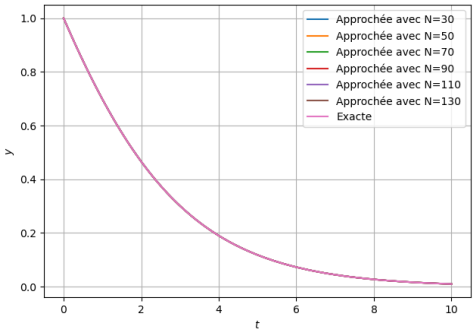
for k in range(6):
    N += 20
    tt = np.linspace(t0, tfinal, N + 1)
    H.append( tt[1] - tt[0] )
    yy = sol_exacte(tt)
    uu = multipas(phi, tt, sol_exacte)
    err.append( np.linalg.norm(uu-yy,np.inf) )
    ax1.plot(tt,uu,label=f'Approchée avec N={N}')

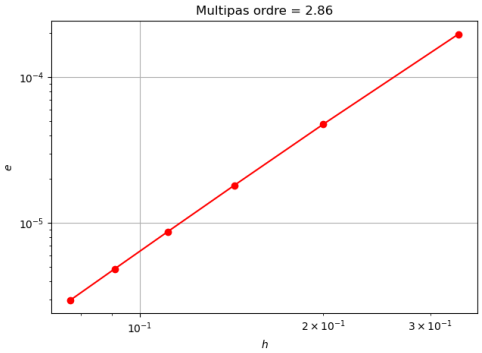
ax1.plot(tt,yy,label='Exacte')
plt.xlabel('$t$')
plt.ylabel('$y$')
ax1.grid(True)
ax1.legend()

ax2 = plt.subplot(1,2,2)
ax2.loglog( H, err, 'r-o')
plt.xlabel('$h$')
```

```
plt.ylabel('$e$')
plt.title(f'Multipas ordre = {np.polyfit(np.log(H),np.log(err),1)[0]:1.2f}')
ax2.grid(True)
```

$$y(t) = \frac{3 \cdot \left(\frac{1}{4} - \frac{\sqrt{e^t}}{2}\right)}{\frac{1}{4} - e^t}$$





Chapitre 33.

2021 Contrôle Continu

```
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp
from scipy.optimize import fsolve
from scipy.integrate import solve_ivp, odeint
from IPython.display import display, Markdown, Math
```

33.1. EXERCICE 1 - sujet A : implémentation d'un schéma pour un système

Le modèle S.Z.R., proposée par Munz, Hudea, Imad et Smith en 2009, est une variation du modèle S.I.R pour modéliser une apocalypse zombie.
À chaque instant on décide de diviser la population en trois catégories (qu'on appelle « compartiments » dans le langage de l'épidémiologie) : - les individus « Susceptibles » ou « Sains » (S) : ceux qui n'ont jamais eu la maladie et peuvent la contracter ; - les individus « Zombies » (Z) ; - les individus « Rétablis » (R, comme « Recovered » en anglais) : les personnes décédées qui ont la faculté de revenir à la vie (terme ζR) mais aussi les individus débarassés de leur infection (suite à un coup de hache bien placé).
On travaillera avec des variables normalisées : leur valeur est comprise entre 0 et 1 et représente une fraction de la population totale.
Le modèle s'écrit

$$\begin{cases} S'(t) = -\beta S(t)Z(t) \\ Z'(t) = \beta S(t)Z(t) + \zeta R(t) - \alpha S(t)Z(t) \\ R'(t) = \alpha S(t)Z(t) - \zeta R(t) \end{cases}$$

1. Soit $P(t) = S(t) + Z(t) + R(t)$. Montrer que P est un invariant.
2. Calculer la solution approchée obtenue par la méthode d'Euler Progressif avec 61 points. Afficher ensuite sur le même repère $t \mapsto S(t)$, $t \mapsto Z(t)$, $t \mapsto R(t)$ et $t \mapsto P(t)$.
3. Même exercice avec la méthode d'Euler Régressif.
4. Même exercice avec la fonction `odeint` du module scipy.

Testons ce modèle avec les coefficients
- $\alpha = 1$ [humains sains moyennement violents]
- $\beta = 0.5$ [zombies peu agressifs]
- $\zeta = 0.25$ [résurrection des zombies non négligeable]

Dans la suite, considérons les données initiales :
- $S(0) = 0.95$
- $Z(0) = 0.05$
- $R(0) = 0$

On simulera pour $t \in [0; 60]$ jours.

```
alpha = 1
beta  = 0.5
zeta  = 0.25

# yy=[S,I,R]
```

```
phi0 = lambda S,Z,R,t :  -beta*S*Z
phi1 = lambda S,Z,R,t :   (beta-alpha)*S*Z+zeta*R
phi2 = lambda S,Z,R,t :   alpha*S*Z-zeta*R

S0 = 0.95
Z0 = 1-S0
R0 = 0

tt = np.linspace(0,60,61) # en jours
h = tt[1]-tt[0]
```

$P'(t) = S'(t) + Z'(t) + R'(t) = -\beta S(t)Z(t) + \beta S(t)Z(t) + \zeta R(t) - \alpha S(t)Z(t) + \alpha S(t)Z(t) - \zeta R(t) = 0.$

On notera $S_n \approx S(t_n)$, $Z_n \approx Z(t_n)$ et $R_n \approx R(t_n)$.

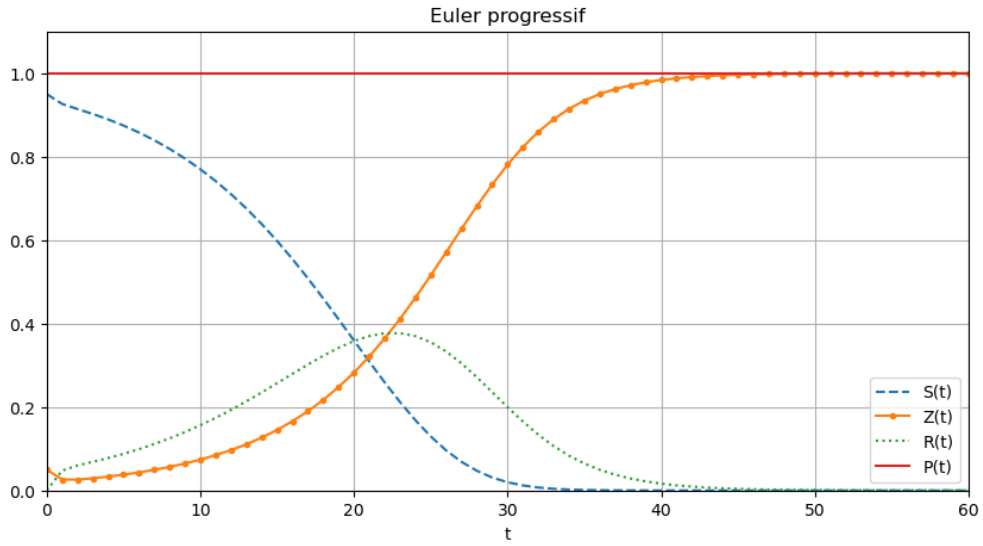
Euler explicite

$$\begin{cases} S_{n+1} = S_n + h\varphi_0(S_n, Z_n, R_n, t_n), \\ Z_{n+1} = Z_n + h\varphi_1(S_n, Z_n, R_n, t_n), \\ R_{n+1} = R_n + h\varphi_2(S_n, Z_n, R_n, t_n). \end{cases}$$

```
# S0, Z0, R0, h variables globales
def euler_progressif(phi0,phi1,phi2,tt):
    SS = [S0]
    ZZ = [Z0]
    RR = [R0]
    for i in range(len(tt)-1):
        SS.append(SS[i]+h*phi0(SS[i],ZZ[i],RR[i],tt[i]))
        ZZ.append(ZZ[i]+h*phi1(SS[i],ZZ[i],RR[i],tt[i]))
        RR.append(RR[i]+h*phi2(SS[i],ZZ[i],RR[i],tt[i]))
    return [SS,ZZ,RR]

[SS_ep, ZZ_ep, RR_ep] = euler_progressif(phi0,phi1,phi2,tt)

plt.figure(figsize=(10,5))
PP_ep = [SS_ep[i]+ZZ_ep[i]+RR_ep[i] for i in range(len(tt))]
plt.plot(tt,SS_ep,'--',tt,ZZ_ep,'.-',tt,RR_ep,':',tt,PP_ep)
plt.xlabel('t')
plt.legend(['S(t)','Z(t)','R(t)','P(t)'])
plt.title('Euler progressif')
plt.grid()
plt.axis([0,60,0,1.1]);
```



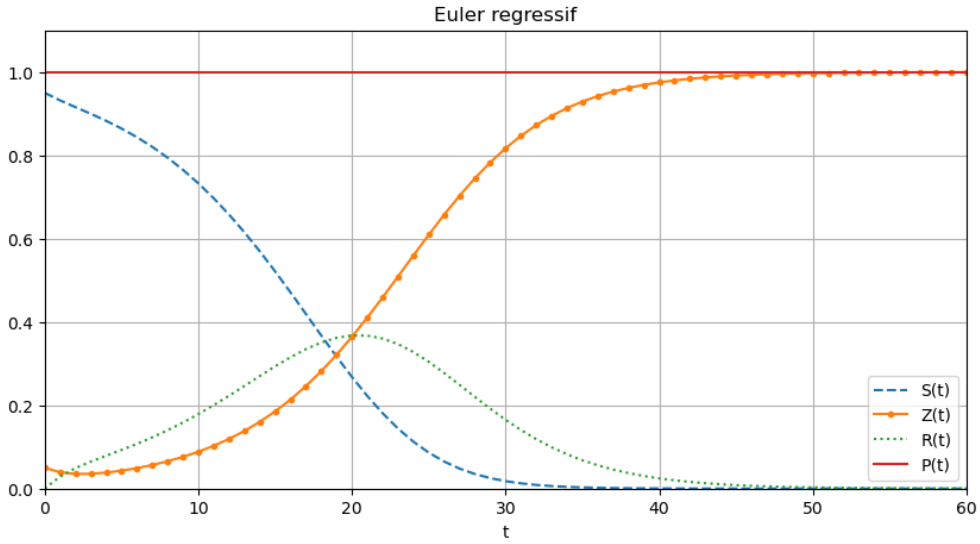
Euler implicite

$$\begin{cases} S_{n+1} = S_n + h\varphi_0(S_{n+1}, Z_{n+1}, R_{n+1}, t_{n+1}), \\ Z_{n+1} = Z_n + h\varphi_2(S_{n+1}, Z_{n+1}, R_{n+1}, t_{n+1}), \\ R_{n+1} = R_n + h\varphi_3(S_{n+1}, Z_{n+1}, R_{n+1}, t_{n+1}). \end{cases}$$

```
# S0, Z0, R0, h variables globales
def euler_regressif(phi0,phi1,phi2,tt):
    SS = [S0]
    ZZ = [Z0]
    RR = [R0]
    for i in range(len(tt)-1):
        sys = lambda z : [-z[0]+SS[i]+h*phi0(z[0],z[1],z[2],tt[i+1]) ,
                          -z[1]+ZZ[i]+h*phi1(z[0],z[1],z[2],tt[i+1]) ,
                          -z[2]+RR[i]+h*phi2(z[0],z[1],z[2],tt[i+1]) ]
        Stemp,Ztemp,Rtemp = fsolve( sys , (SS[i],ZZ[i],RR[i]) )
        SS.append(Stemp)
        ZZ.append(Ztemp)
        RR.append(Rtemp)
    return [SS,ZZ,RR]

[SS_er, ZZ_er, RR_er] = euler_regressif(phi0,phi1,phi2,tt)

plt.figure(figsize=(10,5))
PP_er=[SS_er[i]+ZZ_er[i]+RR_er[i] for i in range(len(tt))]
plt.plot(tt,SS_er,'--',tt,ZZ_er,'.-',tt,RR_er,':',tt,PP_er)
plt.xlabel('t')
plt.legend(['S(t)','Z(t)','R(t)','P(t)'])
plt.title('Euler regressif')
plt.grid()
plt.axis([tt[0],tt[-1],0,1.1]);
```



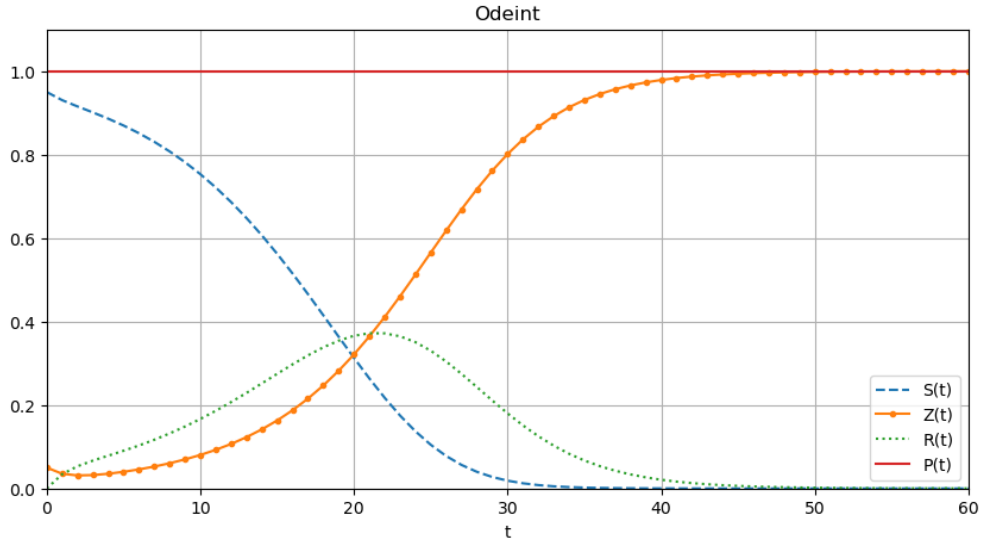
odeint

```
pphi = lambda yy,t : [ phi0(yy[0],yy[1],yy[2],t), phi1(yy[0],yy[1],yy[2],t), phi2(yy[0],yy[1],yy[2],t) ]
CI   = [S0,Z0,R0]
sol  = odeint(pphi,CI,tt)

SS,ZZ,RR = sol[:,0],sol[:,1],sol[:,2]

plt.figure(figsize=(10,5))
PP = [SS[i]+ZZ[i]+RR[i] for i in range(len(tt))]
plt.plot(tt,SS,'--',tt,ZZ,'.-',tt,RR,':',tt,PP)
```

```
plt.xlabel('t')
plt.legend(['S(t)','Z(t)','R(t)','P(t)'])
plt.title('Odeint')
plt.grid()
plt.axis([0,60,0,1.1]);
```



33.2. EXERCICE 1 - sujet B : implémentation d'un schéma pour un système

Le modèle S.I.R. a été présenté pour la première fois par KERMACK & McKENDRICK à Londres et Cambridge en 1927 pour expliquer a posteriori l'évolution de l'épidémie de peste à Bombay en 1905-1906.
À chaque instant on décide de diviser la population en trois catégories (qu'on appelle « compartiments » dans le langage de l'épidémiologie) : - les individus « Susceptibles » ou « Sains » (S) : ceux qui n'ont jamais eu la maladie et peuvent la contracter ; - les individus « Infectés » (I) : les malades, ce sont aussi les contagieux (c'est une hypothèse de ce modèle) ; - les individus « Rétablis » (R, comme « Recovered » en anglais) : ceux qui ont déjà eu la maladie et sont désormais immunisés contre cette maladie. Dans ce modèle on inclut dans ce groupe les personnes décédées (puisqu'elles ne peuvent plus contracter la maladie).
On travaillera avec des variables normalisées : leur valeur est comprise entre 0 et 1 et représente une fraction de la population totale (P).
Le modèle s'écrit

$$\begin{cases} S'(t) = -\beta S(t)I(t) \\ I'(t) = \beta S(t)I(t) - \nu I(t) \\ R'(t) = \nu I(t) \end{cases}$$

1. Soit $P(t) = S(t) + I(t) + R(t)$. Montrer que P est un invariant.
2. Calculer la solution approchée obtenue par la méthode d'Euler Progressif avec 91 points. Afficher ensuite sur le même repère $t \mapsto S(t)$, $t \mapsto I(t)$, $t \mapsto R(t)$ et $t \mapsto P(t)$.
3. Même exercice avec la méthode d'Euler Régressif.
4. Même exercice avec la fonction `odeint` du module scipy.

Dans la suite, considérons les données initiales :

- $S(0) = 0.99$
- $I(0) = 0.01$
- $R(0) = 0$

Pour la rougeole on prendra

- $\beta = 0.8$ = taux de transmission (plus il est grand, plus la maladie est infectieuse)
- $\nu = 0.05$ = taux de rémission ou de décès du virus (= 1 / (duréé moyenne de l'infection) ; plus ν est petit, plus le temps durant lequel le malade est contagieux est grand)

On simulera pour $t \in [0; 90]$ jours.

```
beta = 0.8
nu   = 0.05

# yy=[S,I,R]
phi0 = lambda S,I,R,t :  -beta*S*I
phi1 = lambda S,I,R,t :   beta*S*I-nu*I
phi2 = lambda S,I,R,t :   nu*I

S0=0.99
I0=1-S0
R0=0

tt = np.linspace(0,90,91) # en jours
h = tt[1]-tt[0]
```

$$P'(t) = S'(t) + I'(t) + R'(t) = -\beta S(t)I(t) + \beta S(t)I(t) - \nu I(t) + \nu I(t) = 0.$$

On notera $S_n \approx S(t_n)$, $I_n \approx I(t_n)$ et $R_n \approx R(t_n)$.

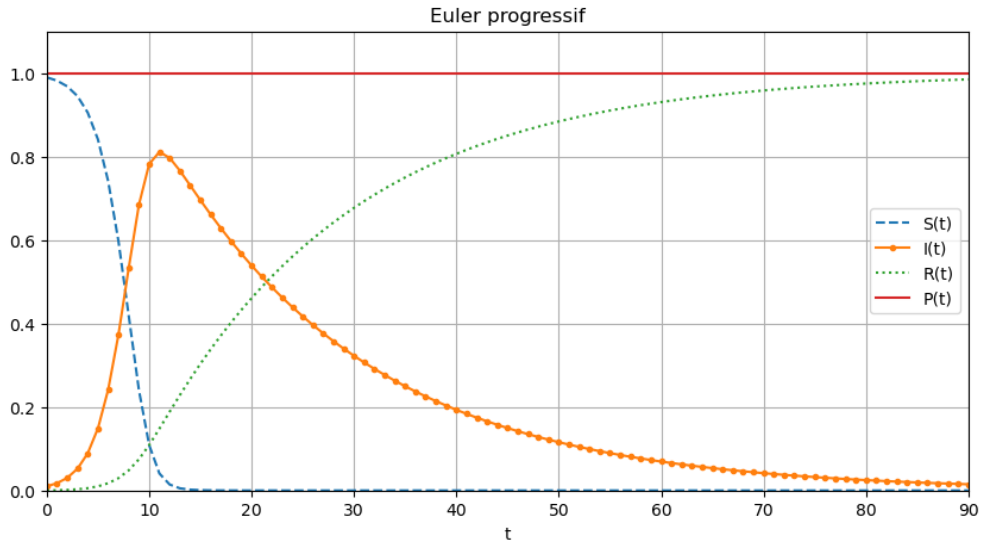
Euler explicite

$$\begin{cases} S_{n+1} = S_n + h\varphi_0(S_n, I_n, R_n, t_n), \\ I_{n+1} = I_n + h\varphi_1(S_n, I_n, R_n, t_n), \\ R_{n+1} = R_n + h\varphi_2(S_n, I_n, R_n, t_n). \end{cases}$$

```
# S0, I0, R0, h variables globales
def euler_progressif(phi0,phi1,phi2,tt):
    SS = [S0]
    II = [I0]
    RR = [R0]
    for i in range(len(tt)-1):
        SS.append(SS[i]+h*phi0(SS[i],II[i],RR[i],tt[i]))
        II.append(II[i]+h*phi1(SS[i],II[i],RR[i],tt[i]))
        RR.append(RR[i]+h*phi2(SS[i],II[i],RR[i],tt[i]))
    return [SS,II,RR]

[SS_ep, II_ep, RR_ep] = euler_progressif(phi0,phi1,phi2,tt)

plt.figure(figsize=(10,5))
PP_ep=[SS_ep[i]+II_ep[i]+RR_ep[i] for i in range(len(tt))]
plt.plot(tt,SS_ep,'--',tt,II_ep,'.-',tt,RR_ep,':',tt,PP_ep)
plt.xlabel('t')
plt.legend(['S(t)','I(t)','R(t)','P(t)'])
plt.title('Euler progressif')
plt.grid()
plt.axis([tt[0],tt[-1],0,1.1]);
```



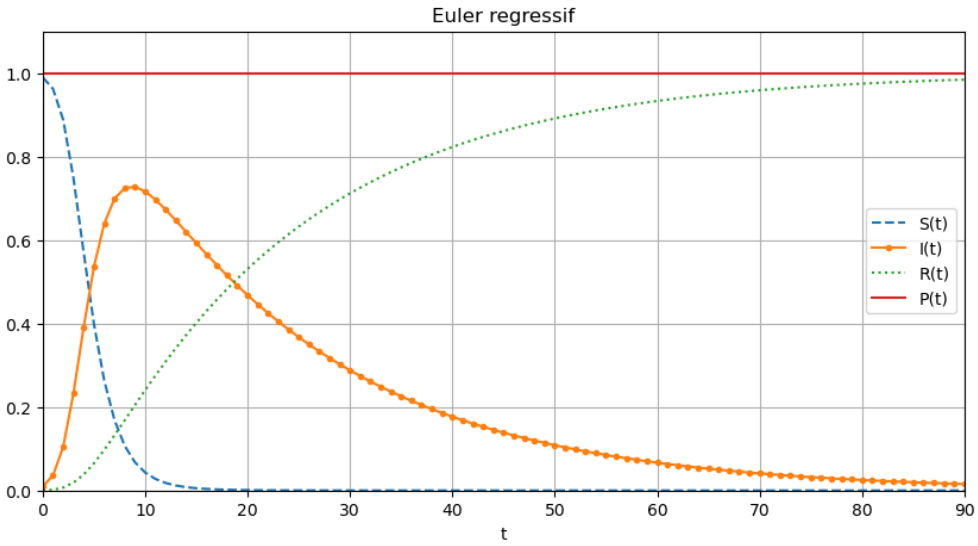
Euler implicite

$$\begin{cases} S_{n+1} = S_n + h\varphi_0(S_{n+1}, I_{n+1}, R_{n+1}, t_{n+1}), \\ I_{n+1} = I_n + h\varphi_1(S_{n+1}, I_{n+1}, R_{n+1}, t_{n+1}), \\ R_{n+1} = R_n + h\varphi_2(S_{n+1}, I_{n+1}, R_{n+1}, t_{n+1}). \end{cases}$$

```
from scipy.optimize import fsolve
# S0, I0, R0, h variables globales
def euler_regressif(phi0,phi1,phi2,tt):
    SS = [S0]
    II = [I0]
    RR = [R0]
    for i in range(len(tt)-1):
        sys = lambda z : [ -z[0]+SS[i]+h*phi0(z[0],z[1],z[2],tt[i+1]) ,
 ↪  -z[1]+II[i]+h*phi1(z[0],z[1],z[2],tt[i+1]) , -z[2]+RR[i]+h*phi2(z[0],z[1],z[2],tt[i+1]) ]
        Stemp,Itemp, Rtemp = fsolve( sys , (SS[i],II[i],RR[i]) )
        SS.append(Stemp)
        II.append(Itemp)
        RR.append(Rtemp)
    return [SS,II,RR]

[SS_er, II_er, RR_er] = euler_regressif(phi0,phi1,phi2,tt)

plt.figure(figsize=(10,5))
PP_er=[SS_er[i]+II_er[i]+RR_er[i] for i in range(len(tt))]
plt.plot(tt,SS_er,'--',tt,II_er,'.-',tt,RR_er,':',tt,PP_er)
plt.xlabel('t')
plt.legend(['S(t)','I(t)','R(t)','P(t)'])
plt.title('Euler regressif')
plt.grid()
plt.axis([tt[0],tt[-1],0,1.1]);
```

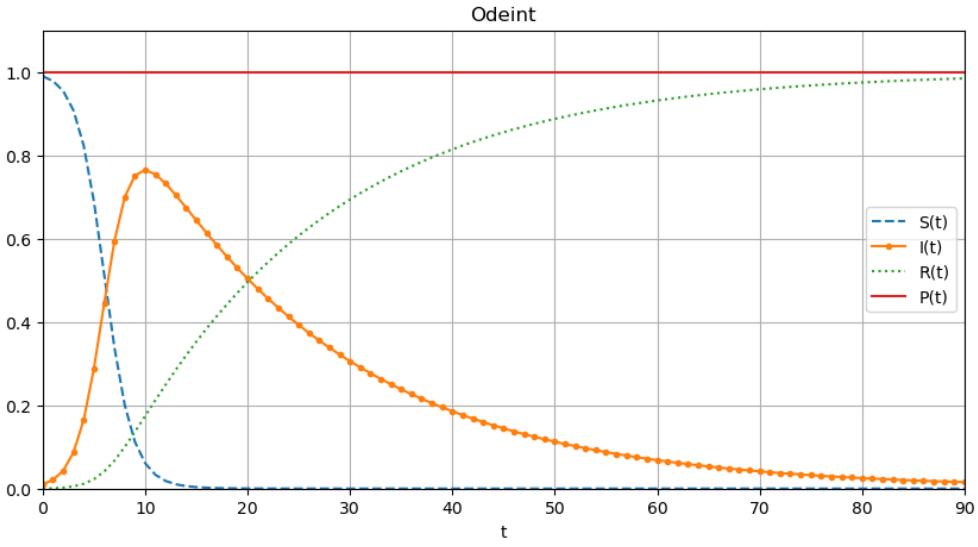


odeint

```
pphi = lambda yy,t : [ phi0(yy[0],yy[1],yy[2],t), phi1(yy[0],yy[1],yy[2],t), phi2(yy[0],yy[1],yy[2],t) ]
CI   = [S0,I0,R0]
sol  = odeint(pphi,CI,tt)

SS,II,RR = sol[:,0],sol[:,1],sol[:,2]

plt.figure(figsize=(10,5))
PP = [SS[i]+II[i]+RR[i] for i in range(len(tt))]
plt.plot(tt,SS,'--',tt,II,'.-',tt,RR,':',tt,PP)
plt.xlabel('t')
plt.legend(['S(t)','I(t)','R(t)','P(t)'])
plt.title('Odeint')
plt.grid()
plt.axis([tt[0],tt[-1],0,1.1]);
```



33.3. EXERCICE 2 - sujet A : étude d'un schéma multipas

On notera $\varphi_k \overset{\text{déf}}{=} \varphi(t_k, u_k)$. Soit la méthode multipas

$$u_{n+1} = 2\gamma u_n + (1-2\gamma)\, u_{n-1} + h\left(\gamma\varphi_{n+1} + (2-3\gamma)\,\varphi_n\right).$$

1. Pour quelles valeurs de $ $ est-elle consistante ?
2. Pour quelles valeurs de γ est-elle convergente ?
3. Quel ordre de convergente a-t-elle ?
4. Pour deux valeurs différentes de γ pour lesquelles le schéma est convergent (dont une qui donne l'ordre de convergence maximale), vérifier empiriquement l'ordre de convergence sur le problème de Cauchy

$$\begin{cases} y'(t) = y(t)(1-y(t)), \quad \forall t \in I = [0,1], \\ y(0) = 2, \end{cases}$$

après avoir calculé la solution exacte.

$p = 1$: c'est une méthode à $q = p + 1 = 2$: - $a_0=2 $ et $a_1=1-2 $ - $b_0=2-3 $ et $b_1 = 0$ - $b_{-1}= $

La méthode est implicite si $\gamma \neq 0$.

La méthode est consistante pour tout γ car

$$\begin{cases} \displaystyle\sum_{j=0}^{p} a_j = a_0 + a_1 = 2\gamma + (1-2\gamma) = 1, \\ \displaystyle -\sum_{j=0}^{p} j a_j + \sum_{j=-1}^{p} b_j = -(0a_0 + 1a_1) + (b_{-1} + b_0 + b_1) = -(1-2\gamma) + (\gamma + 2 - 3\gamma) = 1 \end{cases}$$

Vérifions ces calculs ci-dessous :

```
sp.var('gamma')
p=1
a0=2*gamma
a1=1-2*gamma
b0=2-3*gamma
b1=0
bm1=gamma

cond11=a0+a1
cond12=-(0*a0+1*a1)+(bm1+b0+b1)
display(cond11)
display(cond12.factor())
display(sp.solve([sp.Eq(cond11,1),sp.Eq(cond12,1)],gamma))
```

1

1

[]

Le premier polynôme caractéristique est

$$\varrho(r) = r^{p+1} - \sum_{j=0}^{p} a_j r^{p-j} = r^2 - a_0 r - a_1 = r^2 - 2\gamma r + (2\gamma - 1)$$

dont les racines sont

$$r_0 = 1, \quad r_1 = 2\gamma - 1.$$

```
sp.var('r')
rho=r**2-a0*r-a1
display(sp.factor(rho))
sol=sp.solve(rho,r)
display(sol)
```

$(r-1)\left(-2\gamma + r + 1\right)$

```
[1, 2*gamma - 1]
```

La méthode est zéro-stable ssi

$$\begin{cases} |r_j| \leq 1 \quad \text{pour tout } j = 0, \dots, p \\ \varrho'(r_j) \neq 0 \text{ si } |r_j| = 1, \end{cases}$$

donc ssi $0 \leq \gamma < 1$.

```
cond=sp.solve([abs(c)<=1 for c in sol],gamma)
print('La première condition impose')
display(cond)
```

```
La première condition impose
```

$0 \leq \gamma \wedge \gamma \leq 1$

Conclusion : la méthode est convergente ssi elle est consistante et zéro-stable donc ssi $0 \leq \gamma < 1$.

La méthode est d'ordre au moins 2 car

$$\sum_{j=0}^{p} (-j)^2 a_j + 2 \sum_{j=-1}^{p} (-j)^1 b_j = (0^2 a_0 + 1^2 a_1) + 2(-(-1)b_{-1} + 0b_0 + 1b_1) = (1 - 2\gamma) + 2(\gamma) = 1$$

```
cond2=(0*a0+1*a1)+2*(-(-1)*bm1+0*b0+1*b1)
display(cond2.factor())
display(sp.solve(sp.Eq(cond2,1),gamma))
```

1

```
[]
```

La méthode est d'ordre 3 ssi $\gamma = \frac{2}{5}$ car

$$\sum_{j=0}^{p} (-j)^3 a_j + 3 \sum_{j=-1}^{p} (-j)^2 b_j = -a_1 + 3(b_{-1} + b_1) = -(1 - 2\gamma) + 3(\gamma) = 5\gamma - 1 = 1 \iff \gamma = \frac{2}{5}$$

```
cond3=(0*a0-1*a1)+3*(1*bm1+0*b0+1*b1)
display(cond3.factor())
display(sp.solve(sp.Eq(cond3,1),gamma))
```

$5\gamma - 1$

```
[2/5]
```

Conclusion

- le schéma est convergent au moins d'ordre 2 pour tout $\gamma \in [0, 1[$
- le schéma est convergent à l'ordre 3 pour $\gamma = \frac{2}{5}$ (et on peut montrer qu'il n'est pas d'ordre 4)

On définit la solution exacte (calculée en utilisant le module `sympy`) pour estimer les erreurs :

```
# variables globales
t0     = 0
tfinal = 1
y0     = 2

phi = lambda t,y : y*(1-y)


t  = sp.Symbol('t')
y  = sp.Function('y')
edo = sp.Eq( sp.diff(y(t),t) , phi(t,y(t)) )
solpar = sp.dsolve(edo,y(t),ics={y(t0):y0})
display(Markdown(f'La solution exacte est : $${sp.latex(solpar)}$$'))

sol_exacte = sp.lambdify(t,solpar.rhs,'numpy')
```

La solution exacte est :

$$y(t) = \frac{1}{1 - \frac{e^{-t}}{2}}$$

On calcule la solution approchée pour différentes valeurs de N :

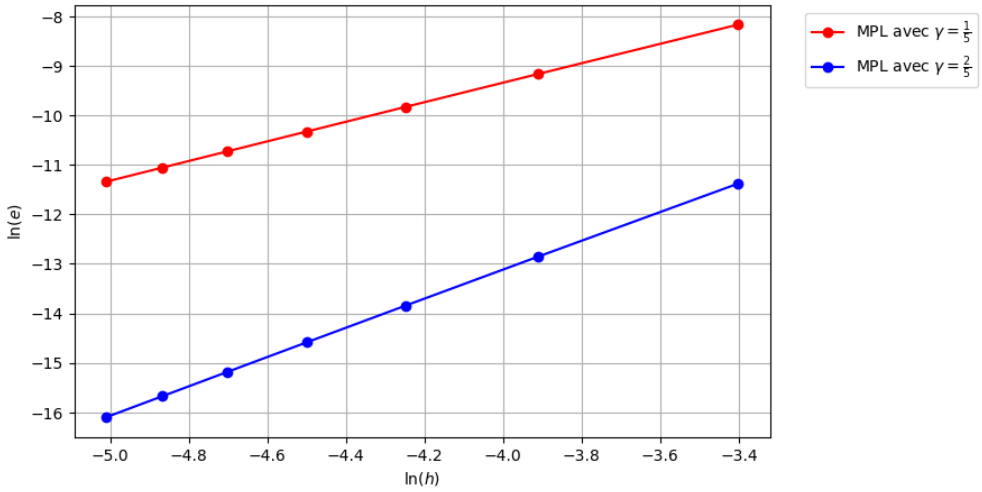
```
def multipas(phi, tt, gamma):
    h  = tt[1] - tt[0]
    uu = [y0]
    uu.append(sol_exacte(tt[1]))
    for i in range(1,len(tt) - 1):
        temp = fsolve( lambda x :
↪   -x+2*gamma*uu[i]+(1-2*gamma)*uu[i-1]+h*(gamma*phi(tt[i+1],x)+(2-3*gamma)*phi(tt[i],uu[i]))   ,uu[i])
        uu.append( temp[0] )
    return uu


H = np.array([])
err_mp_2 = np.array([])
err_mp_3 = np.array([])
N = 10
for k in range(7):
    N += 20
    tt = np.linspace(t0, tfinal, N + 1)
    h = tt[1] - tt[0]
    yy = sol_exacte(tt)
    uu_mp_2 = multipas(phi, tt, 1/5)
    uu_mp_3 = multipas(phi, tt, 2/5)
    H = np.append(H,h)
    err_mp_2 = np.append(err_mp_2,np.linalg.norm(uu_mp_2-yy,np.inf))
    err_mp_3 = np.append(err_mp_3,np.linalg.norm(uu_mp_3-yy,np.inf))

ordre_mp_2 = np.polyfit(np.log(H),np.log(err_mp_2), 1)[0]
ordre_mp_3 = np.polyfit(np.log(H),np.log(err_mp_3), 1)[0]
print ('Multipas gamma=1/5 ordre=%1.2f' %(ordre_mp_2))
print ('Multipas gamma=2/5 ordre=%1.2f' %(ordre_mp_3))


plt.figure(figsize=(8,5))
plt.plot(np.log(H), np.log(err_mp_2), 'r-o', label=r'MPL avec $\gamma=\frac{1}{5}$')
plt.plot(np.log(H), np.log(err_mp_3), 'b-o', label=r'MPL avec $\gamma=\frac{2}{5}$')
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
plt.legend(bbox_to_anchor=(1.04, 1), loc='upper left')
plt.grid(True)
plt.show()
```

```
Multipas gamma=1/5 ordre=1.97
Multipas gamma=2/5 ordre=2.94
```



33.4. EXERCICE 2 - sujet B : étude d'un schéma multipas

On notera $\varphi_k \overset{\text{déf}}{=} \varphi(t_k, u_k)$. Soit la méthode multipas

$$u_{n+1} = 2(1-\gamma)u_n + (2\gamma-1)\,u_{n-1} + h(1-\gamma)\varphi_{n+1} + h\,(3\gamma-1)\,\varphi_n.$$

1. Pour quelles valeurs de $ $ est-elle consistante ?
2. Pour quelles valeurs de γ est-elle convergente ?
3. Quel ordre de convergente a-t-elle ?
4. Pour deux valeurs différentes de γ pour lesquelles le schéma est convergent (dont une qui donne l'ordre de convergence maximale), vérifier empiriquement l'ordre de convergence sur le problème de Cauchy

$$\begin{cases} y'(t) = y(t)(1-y(t)), \quad \forall t \in I = [0,1], \\ y(0) = 2, \end{cases}$$

après avoir calculé la solution exacte.

$p = 1$: c'est une méthode à $q = p+1 = 2$: - $a_0=2(1-) $ et $a_1 = 2\gamma - 1$ - $b_0=3 -1 $ et $b_1 = 0$ - $b_{-1}=1- $

La méthode est implicite si $\gamma \neq 1$.

La méthode est consistante pour tout γ car

$$\begin{cases} \displaystyle\sum_{j=0}^{p} a_j = a_0 + a_1 = 2(1-\gamma) + (2\gamma-1) = 1, \\ \displaystyle -\sum_{j=0}^{p} j a_j + \sum_{j=-1}^{p} b_j = -(0a_0 + 1a_1) + (b_{-1} + b_0 + b_1) = -(2\gamma-1) + (1-\gamma+3\gamma-1) = 1 \end{cases}$$

Vérifions ces calculs ci-dessous :

```
sp.var('gamma')
p=1
a0=2*(1-gamma)
a1=2*gamma-1
b0=3*gamma-1
b1=0
bm1=1-gamma
```

```
cond11=a0+a1
cond12=-(0*a0+1*a1)+(bm1+b0+b1)
display(cond11)
display(cond12.factor())
display(sp.solve([sp.Eq(cond11,1),sp.Eq(cond12,1)],gamma))
```

1

1

```
[]
```

Le premier polynôme caractéristique est

$$\varrho(r) = r^{p+1} - \sum_{j=0}^{p} a_j r^{p-j} = r^2 - a_0 r - a_1 = r^2 - 2(1-\gamma)r - (2\gamma - 1)$$

dont les racines sont

$$r_0 = 1, \quad r_1 = 1 - 2\gamma.$$

```
sp.var('r')
rho=r**2-a0*r-a1
display(sp.factor(rho))
sol=sp.solve(rho,r)
display(sol)
```

$(r-1)\,(2\gamma + r - 1)$

```
[1, 1 - 2*gamma]
```

La méthode est zéro-stable ssi

$$\begin{cases} |r_j| \leq 1 \quad \text{pour tout } j = 0, \dots, p \\ \varrho'(r_j) \neq 0 \text{ si } |r_j| = 1, \end{cases}$$

donc ssi $0 < \gamma \leq 1$.

```
cond=sp.solve([abs(c)<=1 for c in sol],gamma)
display(cond)
```

$0 \leq \gamma \wedge \gamma \leq 1$

Conclusion : la méthode est convergente ssi elle est consistante et zéro-stable donc ssi $0 < \gamma \leq 1$.

La méthode est d'ordre au moins 2 car

$$\sum_{j=0}^{p} (-j)^2 a_j + 2 \sum_{j=-1}^{p} (-j)^1 b_j = (0^2 a_0 + 1^2 a_1) + 2(-(-1)b_{-1} + 0b_0 + 1b_1) = (2\gamma - 1) + 2(1-\gamma) = 1$$

```
cond2=(0*a0+1*a1)+2*(-(-1)*bm1+0*b0+1*b1)
display(cond2.factor())
display(sp.solve(sp.Eq(cond2,1),gamma))
```

1

```
[]
```

La méthode est d'ordre 3 ssi $\gamma = \frac{2}{5}$ car

$$\sum_{j=0}^{p}(-j)^3 a_j + 3\sum_{j=-1}^{p}(-j)^2 b_j = -a_1 + 3(b_{-1}+b_1) = -(2\gamma-1)+3(1-\gamma) = 4-5\gamma = 1 \iff \gamma = \frac{3}{5}$$

```
cond3=(0*a0-1*a1)+3*(1*bm1+0*b0+1*b1)
display(cond3.factor())
display(sp.solve(sp.Eq(cond3,1),gamma))
```

$4-5\gamma$

[3/5]

Conclusion

- le schéma est convergent au moins d'ordre 2 pour tout $\gamma \in]0,1]$
- le schéma est convergent à l'ordre 3 pour $\gamma = \frac{3}{5}$ (et on peut montrer qu'il n'est pas d'ordre 4)

On définit la solution exacte (calculée en utilisant le module `sympy`) pour estimer les erreurs :

```
# variables globales
t0     = 0
tfinal = 1
y0     = 2

phi = lambda t,y : y*(1-y)


t  = sp.Symbol('t')
y  = sp.Function('y')
edo= sp.Eq( sp.diff(y(t),t) , phi(t,y(t)) )
solpar = sp.dsolve(edo, y(t), ics={y(t0):y0})
display(Markdown(f'La solution exacte est : $${sp.latex(solpar)}$$'))

sol_exacte = sp.lambdify(t,solpar.rhs,'numpy')
```

La solution exacte est :

$$y(t) = \frac{1}{1-\frac{e^{-t}}{2}}$$

On calcule la solution approchée pour différentes valeurs de N :

```
def multipas(phi, tt, gamma):
    h  = tt[1] - tt[0]
    uu = np.empty_like(tt)
    uu[0] = y0
    uu[1] = sol_exacte(tt[1])
    for i in range(1,len(tt) - 1):
        temp = fsolve( lambda x :
 ↪  -x+2*(1-gamma)*uu[i]+(2*gamma-1)*uu[i-1]+h*(1-gamma)*phi(tt[i+1],x)+h*(3*gamma-1)*phi(tt[i],uu[i])
 ↪  ,uu[i])
        uu[i+1] = temp[0]
    return uu

H = np.array([])
err_mp_2 = np.array([])
err_mp_3 = np.array([])
N = 10
for k in range(7):
    N += 20
```

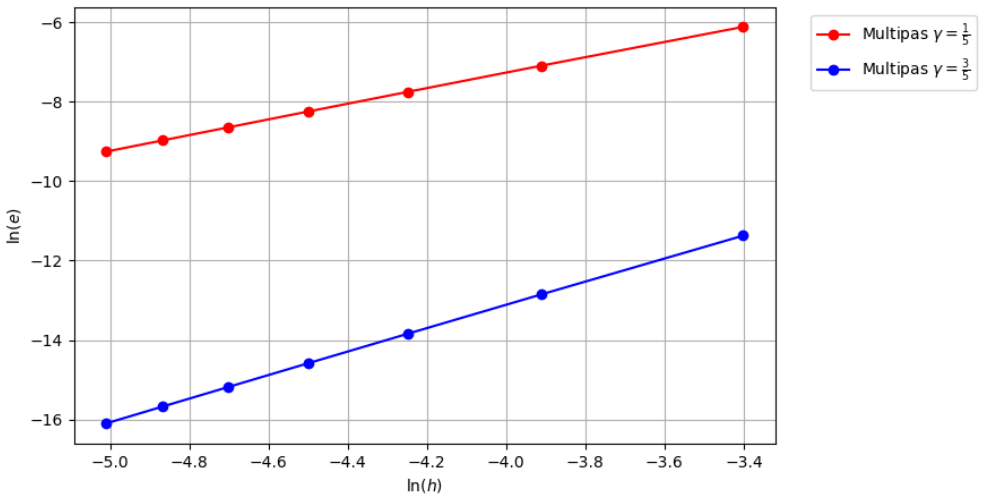
```
    tt = np.linspace(t0, tfinal, N + 1)
    h = tt[1] - tt[0]
    yy = sol_exacte(tt)
    uu_mp_2 = multipas(phi, tt, 1/5)
    uu_mp_3 = multipas(phi, tt, 3/5)
    H = np.append(H,h)
    err_mp_2 = np.append(err_mp_2,np.linalg.norm(uu_mp_2-yy,np.inf))
    err_mp_3 = np.append(err_mp_3,np.linalg.norm(uu_mp_3-yy,np.inf))

ordre_mp_2 = np.polyfit(np.log(H),np.log(err_mp_2), 1)[0]
ordre_mp_3 = np.polyfit(np.log(H),np.log(err_mp_3), 1)[0]

print (f'MPL avec gamma=1/5 ordre={ordre_mp_2:.2f}')
print (f'MPL avec gamma=3/5 ordre={ordre_mp_3:.2f}')

plt.figure(figsize=(8,5))
plt.plot(np.log(H), np.log(err_mp_2), 'r-o', label=r'Multipas $\gamma=\frac{1}{5}$')
plt.plot(np.log(H), np.log(err_mp_3), 'b-o', label=r'Multipas $\gamma=\frac{3}{5}$')
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
plt.legend(bbox_to_anchor=(1.04, 1), loc='upper left')
plt.grid(True)
plt.show()
```

```
MPL avec gamma=1/5 ordre=1.95
MPL avec gamma=3/5 ordre=2.94
```



Chapitre 34.

2021 - Contrôle Terminal session 1

```
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp
from scipy.optimize import fsolve
from scipy.integrate import solve_ivp, odeint
from IPython.display import display, Markdown, Math

prlat= lambda *args: display(Math(''.join( sp.latex(arg) for arg in args )))
```

34.1. EXERCICE 1 : étude d'un schéma RK

Soit le schéma de Runge-Kutta dont la matrice de Butcher est

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \frac{\eta}{4} & \frac{\eta}{4} & 0 \\ \hline & \frac{3-\mu}{3} & \frac{\mu}{3} \end{array}$$

1. Écrire le schéma sous la forme d'une suite définie par récurrence. Le schéma est explicite, semi-implicite ou implicite ?
2. Sans faire de calculs, indiquer quel est l'ordre maximale du schéma.
3. Étudier théoriquement l'ordre du schéma en fonction de η et μ.
4. Étudier théoriquement la A-stabilité du schéma pour les valeurs de η et μ qui donnent l'ordre maximal.
5. Implémenter le schéma avec les valeurs de η et μ trouvées avec vos nom et prénom et approcher la solution du problème de Cauchy suivant avec $N+1$ points (quelles valers de N peut-on choisir ? justifier la réponse)

$$\begin{cases} y'(t) = -50y(t), \quad t \in [0;1] \\ y(0) = 1 \end{cases}$$

Vérifier ensuite empiriquement l'ordre de convergence.

Q1
Écrire le schéma sous la forme d'une suite définie par récurrence.
Le schéma est explicite, semi-implicite ou implicite ?

Le schéma associé à cette matrice est explicite (car la matrice $\mathbb{A}$ est triangulaire inférieure à diagonale nulle).
On peut calculer u_{n+1} à partir de u_n par la formule de récurrence

$$\begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n, u_n\right) \\ K_2 = \varphi\left(t_n + \frac{\eta}{4}h, u_n + \frac{\eta}{4}hK_1\right) \\ u_{n+1} = u_n + \frac{h}{3}\left((3-\mu)K_1 + \mu K_2\right) \quad n = 0, 1, \ldots N-1 \end{cases}$$

Q2
Sans faire de calculs, indiquer quel est l'ordre maximale du schéma.

Soit ω l'ordre de la méthode et $s = 2$ le nombre d'étages. Étant un schéma explicite, $\omega \leq s = 2$.

Q3
Étudier théoriquement l'ordre du schéma en fonction de η et μ.

Si

$$\begin{cases} \sum_{j=1}^{s} b_i = 1 \\ c_i = \sum_{j=1}^{s} a_{ij} \quad i = 1, \dots, s \end{cases}$$

alors $\omega \geq 1$

Si $\omega \geq 1$ et $\sum_{j=1}^{s} b_j c_j = \frac{1}{2}$ alors $\omega \geq 2$

D'après les calculs ci-dessous il est toujours au moins d'ordre 1 et il est d'ordre 2 ssi $\eta\mu = 6$.

```
def ordre_RK(s, A=None, b=None, c=None):

    j = sp.symbols('j')

    if A is None:
        A = sp.MatrixSymbol('a', s, s)
    else:
        A = sp.Matrix(A)
    if c is None:
        c = sp.symbols(f'c_0:{s}')
    if b is None:
        b = sp.symbols(f'b_0:{s}')


    display(Markdown("**Matrice de Butcher**"))
    But = matrice_Butcher(s, A, b, c)

    Eqs = []

    display(Markdown(f"**On a {s+1} conditions pour avoir consistance = pour être d'ordre 1**"))
    ordre_1(s, A, b, c, j)

    display(Markdown("**On doit ajouter 1 condition pour être d'ordre 2**"))
    Eq_2 = ordre_2(s, A, b, c, j)
    Eqs.append(Eq_2)

    return Eqs


def matrice_Butcher(s, A, b, c):
    But = sp.Matrix(A)
    But = But.col_insert(0, sp.Matrix(c))
    last = [sp.Symbol(" ")]
    last.extend(b)
    But = But.row_insert(s,sp.Matrix(last).T)
    display(Math(sp.latex(sp.Matrix(But))))
    return But

def ordre_1(s, A, b, c, j):
    texte = r"\sum_{j=1}^s b_j =" + f"{sum(b).simplify()}"
    texte += r"\text{ doit être égale à }1"
    display(Math(texte))
```

```
    for i in range(s):
        somma = sp.summation(A[i,j],(j,0,s-1)).simplify()
        texte = r'\sum_{j=1}^s a_{'+str(i)+r'j}=' + sp.latex( somma )
        texte += r"\text{ doit être égale à }"+sp.latex(c[i])
        display(Math( texte ))
    return None

def ordre_2(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j=' +sp.latex(sum([b[i]*c[i] for i in range(s)]).simplify())
    texte += r"\text{ doit être égale à }\frac{1}{2}"
    display(Math(texte))
    Eq_2 = sp.Eq( sum([b[i]*c[i] for i in range(s)]).simplify(), sp.Rational(1,2) )
    return Eq_2




##################################################
# Conditions avec s étages
##################################################

eta = sp.Symbol(r'\eta')
mu = sp.Symbol(r'\mu')

c = [0,sp.S(eta)/4]
b = [sp.S(3-mu)/3,sp.S(mu)/3]
A = [[0,0],[sp.S(eta)/4,0]]
s = len(c)


Eqs = ordre_RK(s, A, b, c)

display(Markdown("**Solutions**"))

sol = sp.solve(Eqs)[0]
display(Math(f"{sp.latex(list(sol.keys())[0])}={sp.latex(list(sol.values())[0])}"))
# for eq in Eqs:
#     display(eq)
```

Matrice de Butcher

$$\begin{bmatrix} 0 & 0 & 0 \\ \frac{\eta}{4} & \frac{\eta}{4} & 0 \\ & 1 - \frac{\mu}{3} & \frac{\mu}{3} \end{bmatrix}$$

On a 3 conditions pour avoir consistance = pour être d'ordre 1

$$\sum_{j=1}^{s} b_j = 1 \text{ doit être égale à } 1$$

$$\sum_{j=1}^{s} a_{0j} = 0 \text{ doit être égale à } 0$$

$$\sum_{j=1}^{s} a_{1j} = \frac{\eta}{4} \text{ doit être égale à } \frac{\eta}{4}$$

On doit ajouter 1 condition pour être d'ordre 2

$$\sum_{j=1}^{s} b_j c_j = \frac{\eta\mu}{12} \text{ doit être égale à } \frac{1}{2}$$

Solutions

$$\eta = \frac{6}{\mu}$$

Q4
Étudier théoriquement la A-stabilité du schéma pour les valeurs de η et μ qui donnent l'ordre maximal.

Soit $\beta > 0$ un nombre réel positif et considérons le problème de Cauchy $y'(t) = -\beta y(t)$ pour $t > 0$ et $y(0) = 1$. Sa solution est $y(t) = e^{-\beta t}$ donc $\lim_{t\to+\infty} |y(t)| = 0$.

Le schéma appliqué à ce problème de Cauchy s'écrit

$$\begin{cases} u_0 = y_0 \\ K_1 = -\beta u_n \\ K_2 = -\beta\left(u_n + \dfrac{\eta}{4} h K_1\right) \\ u_{n+1} = u_n + \dfrac{h}{3}\left((3-\mu)K_1 + \mu K_2\right) \quad n = 0, 1, \dots N-1 \end{cases}$$

Dans le cas optimal on a $\eta\mu = 6$, ainsi

$$u_{n+1} = \left(\frac{1}{2}(\beta h)^2 - (\beta h) + 1\right) u_n$$

```
u_n = sp.Symbol('u_n')
dt = sp.Symbol('h') # je change le nom de la variable "h" en "dt" pour ne pas effacer l'affectation de
 ↪  "h"
beta = sp.Symbol(r'\beta')

K1 = -beta*u_n
display(Math('K_1='+sp.latex(K1)))
K2 = -beta*(u_n+sp.S(eta)/4*dt*K1).factor()
display(Math('K_2='+sp.latex(K2)))
u_np1 = (u_n+dt*(sp.S(3-mu)/3*K1+sp.S(mu)/3*K2)).factor()
display(Math('u_{n+1}='+sp.latex(u_np1)))

display(Markdown("Sachant que"))
display(Math(f"{sp.latex(list(sol.keys())[0])}={sp.latex(list(sol.values())[0])}"))
display(Markdown("on a"))
u_np1 = u_np1.subs(sol)
display(Math('u_{n+1}='+sp.latex(u_np1)))
```

$K_1 = -\beta u_n$

$K_2 = \dfrac{\beta u_n \left(\beta\eta h - 4\right)}{4}$

$u_{n+1} = \dfrac{u_n \left(\beta^2 \eta\mu h^2 - 12\beta h + 12\right)}{12}$

Sachant que

$\eta = \dfrac{6}{\mu}$

on a

$u_{n+1} = \dfrac{u_n \left(6\beta^2 h^2 - 12\beta h + 12\right)}{12}$

On note $x = \beta h$ et on étudie la fonction $q\colon \mathbb{R}^+ \to \mathbb{R}$ définie par $q(x) = \frac{1}{2}x^2 - x + 1$ car le schéma est A-stable ssi $|q(x)| < 1$.
Il s'agit d'une parabole convexe de sommet en $x_s = 1$ et $q(x_s) = \frac{1}{2} > -1$ donc $q(x) < 1$ pour $0 < x < 2$ et $q(x) > 0$ pour tout x. On conclut que le schéma est A-stable pour tout $h < \dfrac{2}{\beta}$.

On peut vérifier les calculs avec `sympy` :

```
x = sp.Symbol('x',nonnegative=True)

q = (u_np1/u_n).subs(beta*dt,x)
display(Math('q(x)='+sp.latex(q)))

display(Markdown("$|q(x)|<1$ pour"))
display(sp.solve(abs(q)<1))

sp.plot(q,xlim=[0,3],ylim=[-1.5,1.5]);

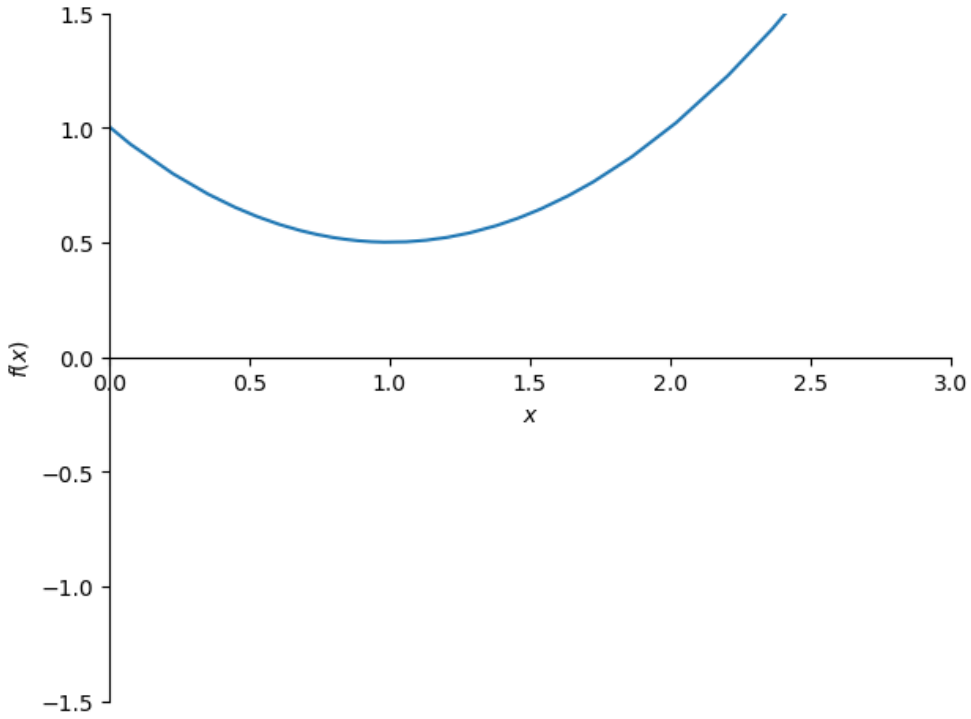

# display(Math('q(0)='+sp.latex(q.subs(x,0))))
# lim=sp.Limit(q,x,sp.oo)
# display(Math(sp.latex(lim)+'='+sp.latex(lim.doit())))

# print('On cherche les points stationnaires dans R^+')
# dq=(q.diff(x)).factor()
# display(Math("q'(x)="+sp.latex(dq)))
# print("q est une parabole dont le sommet est")
# sol=sp.solve(dq)
# display(Math("q'(x)=0 \iff x\in"+sp.latex(sol)))
# minimum=sol[0]
# display(Math("x="+sp.latex(minimum)+"\\text{ est un minimum et}"))
# qmin=q.subs(x,sol[0])
# display(Math("q("+sp.latex(minimum)+")="+sp.latex(qmin)))
# sp.plot(q,1,-1,xlim=[-1,15],ylim=[-1.5,1.5]);
# print("Conclusion: -1<q(x)<1 pour tout x<2. Vérifions ce calcul:")
# print("q(x)<1")
# display(sp.solve(q<1))
# print("q(x)>-1")
# display(sp.solve(q>-1))
# print("Notons que q(x)>0 pour tout 0<x<2")
# #display(sp.solve(q>0))
```

$$q(x) = \frac{x^2}{2} - x + 1$$

$|q(x)| < 1$ pour

$$0 < x \wedge x < 2$$



Pour la question 5 on va fixer η et μ. Pour cela, écrire votre nom et prénom dans la cellule ci-dessous et vous obtiendrez un choix pour les paramètres.

```
nom = "Faccanoni"
prenom = "Gloria"

L = [1,2,3,6]
idx = sum([ord(c) for c in nom+prenom])%len(L)
display(Math(r'\eta='+sp.latex(L[idx])))
display(Math(r'\mu='+sp.latex(sp.S(6)/L[idx])))

eta, mu = L[idx], 6/L[idx]
```

$\eta = 1$

$\mu = 6$

Q5
Implémenter le schéma avec les valeurs de η et μ trouvées avec vos nom et prénom et approcher la solution du problème de Cauchy suivant avec $N+1$ points (quelles valers de N peut-on choisir ? justifier la réponse)

$$\begin{cases} y'(t) = -50y(t), & t \in [0;1] \\ y(0) = 1 \end{cases}$$

Vérifier ensuite empiriquement l'ordre de convergence.

```
# ================================================
# SCHEMA DE RUNGE-KUTTA
# ================================================

# def RK(tt,phi,y0): # comme suite geometrique
#     x = beta*h
#     q = x**2/2-x+1
#     uu = y0*q**np.arange(len(tt))
#     return uu
```

```
def RK(tt,phi,y0):
    uu = np.zeros_like(tt)
    uu[0] = y0
    h = tt[1]-tt[0]
    for n in range(len(tt)-1):
        K1 = phi(tt[n],uu[n])
        K2 = phi(tt[n]+eta/4*h,uu[n]+eta/4*h*K1)
        uu[n+1] = uu[n]+h/3*((3-mu)*K1+mu*K2)
    return uu


# ================================================
# EDO
# ================================================
t0, y0, tfinal = 0, 1, 1
beta = 50

phi = lambda t,y : -beta*y
sol_exacte = lambda t : np.exp(-beta*t)


# ================================================
# COMPARAISON POUR N DONNE
# ================================================
plt.figure(figsize=(20,5))

plt.subplot(1,2,1)

N = 50
tt = np.linspace(t0,tfinal,N+1)
h = tt[1]-tt[0]

if h>=2/beta:
    print('Attention: h > limite de stabilité')

yy = sol_exacte(tt)
# uu = RK(tt)
uu = RK(tt,phi,y0)
plt.plot(tt,yy,'b-',label=("Exacte"))
plt.plot(tt,uu,'ro',label=("RK"))
plt.title(f'${N=}$, ${h=} < {2/beta}$')
plt.xlabel('$t$')
plt.ylabel('$u$')
plt.legend()
plt.grid(True)



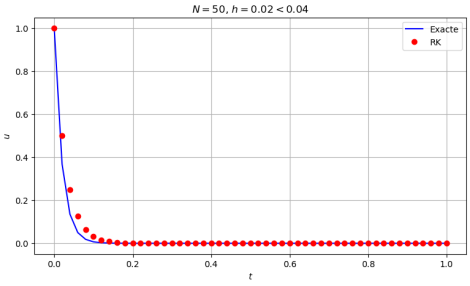
# ================================================
# ORDRE DE CONVERGENCE
# ================================================

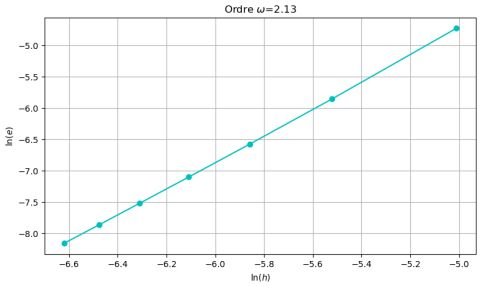
plt.subplot(1,2,2)

H = []
err = []
N = 50
for k in range(7):
    N += 100
    tt = np.linspace(t0, tfinal, N + 1)
    h = tt[1] - tt[0]
    yy = sol_exacte(tt)
```

```
    uu = RK(tt,phi,y0)
    H.append(h)
    err.append(np.linalg.norm(yy-uu, np.inf))

omega = np.polyfit(np.log(H),np.log(err), 1)[0]
plt.plot(np.log(H), np.log(err), 'c-o')
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
plt.title(rf'Ordre $\omega$={omega:1.2f}')
plt.grid(True);
```





34.2. EXERCICE 2 : choix d'un schéma

Soit le problème de Cauchy

$$\begin{cases} y'(t) = -10^2\Big(y(t) - \cos(t)\Big), \quad t \in [0;2] \\ y(0) = 0 \end{cases}$$

1. Calculer la solution exacte.
2. Choisir le schéma le plus adapté parmi les suivants (en justifiant la réponse)
   - EE (Euler Explicite),
   - EI (Euler Implicite),
   - EM (Euler Modifié),
   - H (Heun).
3. Approcher ensuite la solution du problème donné avec $N = 40$ avec le schéma choisi (on affichera la solution exacte et la solution approchée sur le même repère).

- On définit la solution exacte (calculée en utilisant le module `sympy`) :

```
t  = sp.Symbol('t')
y  = sp.Function('y')

t0     = 0
tfinal = 2
y0     = 0

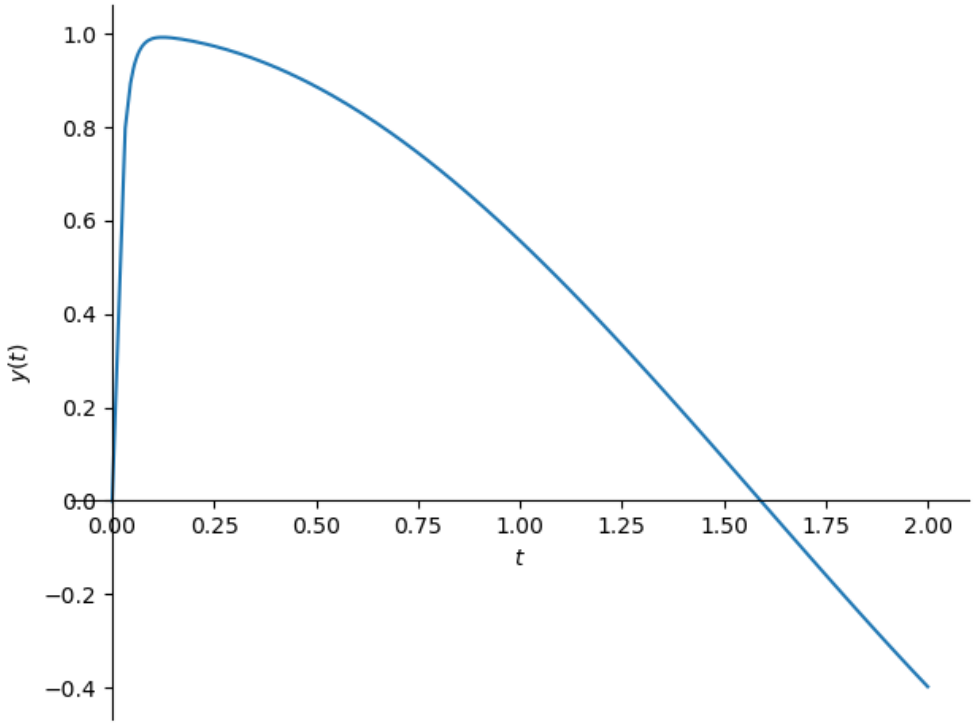
# NB il faut utiliser le cos du module sympy et non de numpy
edo = sp.Eq( sp.diff(y(t),t) , -50*(y(t)-sp.cos(t)) )
display(edo)
solpar = sp.dsolve(edo, y(t), ics={y(t0):y0})
display(solpar)

sp.plot(solpar.rhs, (t, t0, tfinal), ylabel='$y(t)$', xlabel='$t$');

sol_exacte = sp.lambdify(t,solpar.rhs,'numpy')
```

$$\frac{d}{dt}y(t) = -50y(t) + 50\cos(t)$$

$$y(t) = \frac{50\sin(t)}{2501} + \frac{2500\cos(t)}{2501} - \frac{2500e^{-50t}}{2501}$$



- Il s'agit d'une EDO stiff (le terme en exponentielle décroît très rapidement) : on ne peut utiliser ni les schémas EE et EM ni le schéma H car la **condition de stabilité** est trop contraignante. Le schéma EI, en revanche, est inconditionnellement **A-stable** : on peut donc choisir le pas h sans contrainte de stabilité.
- On calcule la solution approchée

```
def EE(phi, tt, y0):
    h = tt[1] - tt[0]
    uu =np.zeros_like(tt)
    uu[0] = y0
    for n in range(len(tt) - 1):
        uu[n+1] = uu[n] + h * phi(tt[n], uu[n])
    return uu

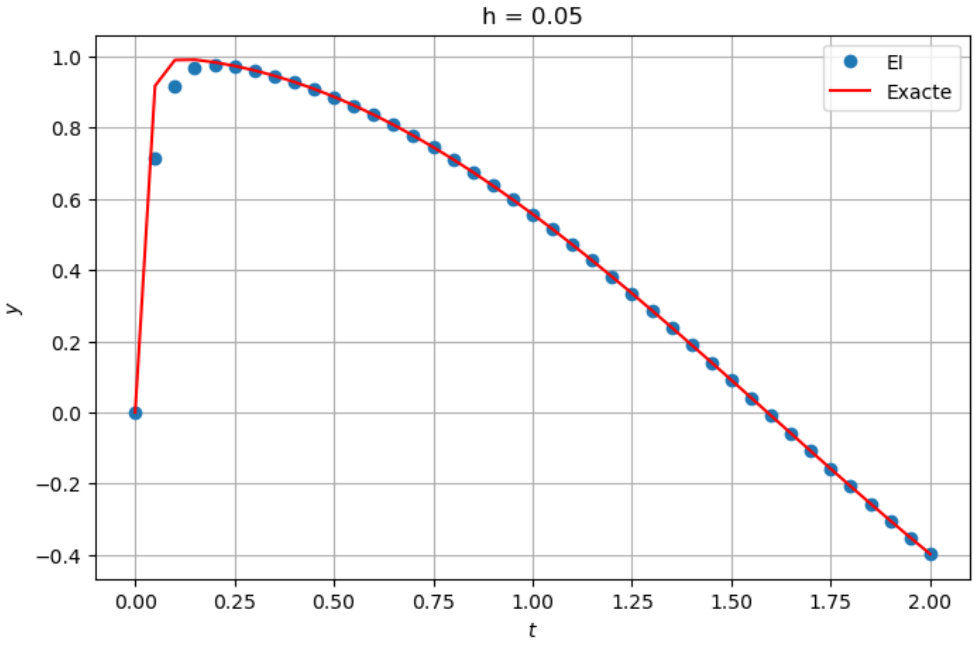
def EI(phi, tt, y0):
    h = tt[1] - tt[0]
    uu =np.zeros_like(tt)
    uu[0] = y0
    for n in range(len(tt) - 1):
        temp = fsolve(lambda x: -x + uu[n] + h * phi(tt[n + 1], x), uu[n])
        uu[n+1] = temp[0]
    return uu


phi = lambda t,y : -50*(y-np.cos(t))

N = 40
tt = np.linspace(t0, tfinal, N + 1)
h = tt[1] - tt[0]
yy = sol_exacte(tt)

# uu = EE(phi, tt, y0) # pour montrer les pb de stabilité avec le même nombre de points
uu = EI(phi, tt, y0)
err = np.linalg.norm(yy-uu, np.inf)
```

```
plt.figure(figsize=(8,5))
plt.plot(tt, uu, 'o', label='EI')
plt.plot(tt, yy, 'r-', label='Exacte')
plt.xlabel('$t$')
plt.ylabel('$y$')
plt.grid(True)
plt.legend()
plt.title(f'{h = }')
plt.show();
```



34.3. EXERCICE 3 : choix d'un schéma

Soit le problème de Cauchy

$$\begin{cases} y'(t) = \alpha(y(t) - t), \quad t \in [0; 8] \\ y(0) = \dfrac{1}{\alpha} + \varepsilon \end{cases}$$

Pour fixer α, écrivez votre nom et prénom dans la cellule ci-dessous et vous obtiendrez un choix pour ce paramètre.

1. Calculer la solution exacte.
2. Soit $\varepsilon = 0$. Parmi les schémas suivants, choisir le plus adapté (plusieurs réponses possibles, choix à justifier) :
   - EE (Euler Explicite),
   - EI (Euler Implicite),
   - EM (Euler Modifié),
   - H (Heun).
3. Soit $\varepsilon = 0$. Approcher la solution du problème donné avec un des schémas choisis (on affichera la solution exacte et la solution approchée sur le même repère).

```
nom = "Faccanoni"
prenom = "Gloria"

L = [2,3,4,5]
idx = sum([ord(c) for c in nom+prenom])%len(L)
```

```
alpha = L[idx]
display(Math(r'\alpha='+sp.latex(alpha)))

# alpha = 5
```

$\alpha = 2$

- On calcule la solution exacte (calculée en utilisant le module `sympy`) :

```
t       = sp.Symbol('t')
epsilon = sp.Symbol(r'\varepsilon')
y       = sp.Function('y')

t0 = 0
y0 = sp.Rational(1,alpha)+epsilon
display(Math('y(0)='+sp.latex(y0)))

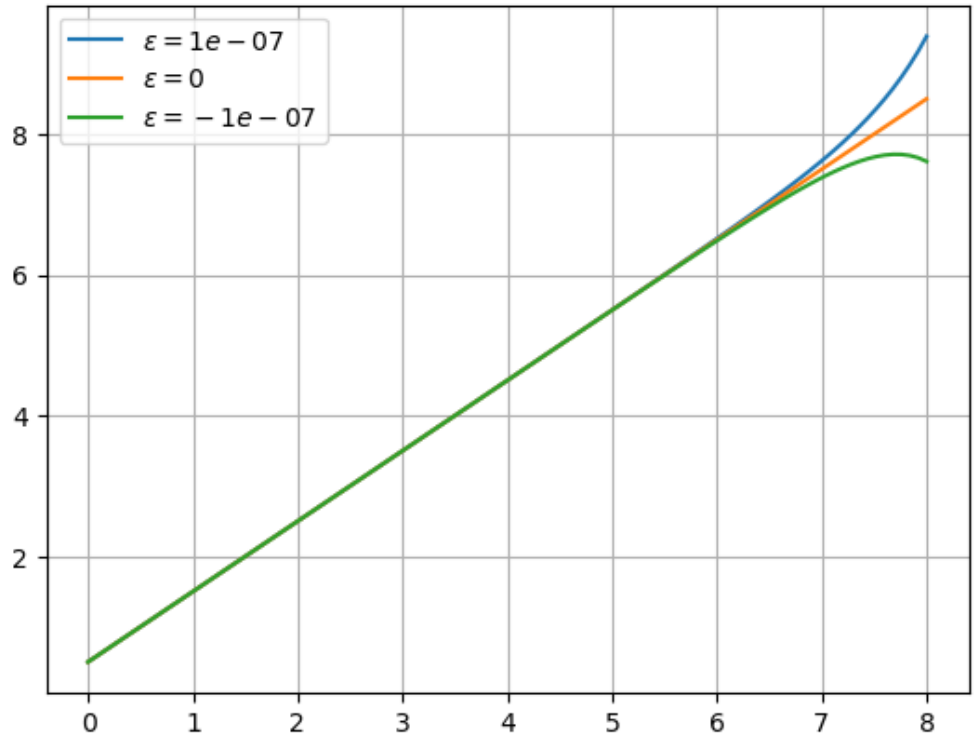
edo = sp.Eq( sp.diff(y(t),t) , alpha*(y(t)-t) )
display(edo)
solpar = sp.dsolve(edo, y(t), ics={y(t0):y0})
display(solpar)

tt = np.linspace(0,8,101)
plt.grid()
for val in [1.e-7,0,-1.e-7]:
    sol_exacte = sp.lambdify(t,solpar.rhs.subs(epsilon,val),'numpy')
    plt.plot(tt,sol_exacte(tt), label=f'$\\epsilon={val}$')
plt.legend();
```

$y(0) = \varepsilon + \frac{1}{2}$

$\frac{d}{dt}y(t) = -2t + 2y(t)$

$y(t) = \varepsilon e^{2t} + t + \frac{1}{2}$



- On remarque que c'est un problème de Cauchy numériquement mal posé (si $\varepsilon = 0$ la fonction est linéaire, avec $\varepsilon > 0$ elle croît comme une exponentielle et avec $\varepsilon < 0$ elle decroît comme une exponentielle) : on doit donc choisir le schéma le plus précis parmi les schémas proposés, à savoir le schéma de Heun ou le schéma d'Euler Modifié qui sont d'ordre 2.

```
phi = lambda t,y : alpha*(y-t)

t0     = 0
tfinal = 8
y0     = 1/alpha

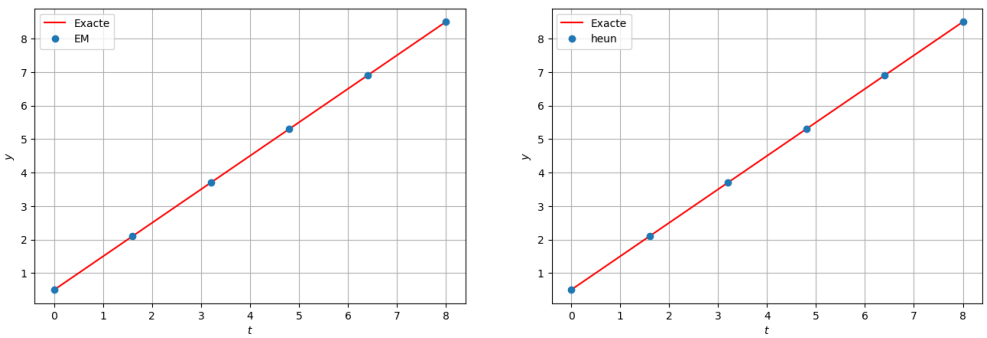
N = 5
tt = np.linspace(t0, tfinal, N + 1)

sol_exacte = sp.lambdify(t,solpar.rhs.subs(epsilon,0),'numpy')
yy = sol_exacte(tt)
```

```
def EM(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.zeros_like(tt)
    uu[0] = y0
    for n in range(len(tt)-1):
        k1 = phi( tt[n], uu[n] )
        uu[n+1] = uu[n]+h*phi(tt[n]+h/2,uu[n]+k1*h/2)
    return uu

def heun(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.zeros_like(tt)
    uu[0] = y0
    for n in range(len(tt)-1):
        k1 = phi( tt[n], uu[n] )
        k2 = phi( tt[n+1], uu[n] + h*k1  )
        uu[n+1] = uu[n] + (k1+k2)*h/2
    return uu


fig,ax = plt.subplots(nrows=1,ncols=2,figsize=(16,5))
ax[0].plot(tt, yy, 'r-', label='Exacte')
ax[1].plot(tt, yy, 'r-', label='Exacte')
for i,schema in enumerate([EM, heun]):
    uu = schema(phi, tt, y0)
    ax[i].plot(tt, uu, 'o', label=schema.__name__)
    ax[i].set_xlabel('$t$')
    ax[i].set_ylabel('$y$')
    ax[i].grid(True)
    ax[i].legend();
```

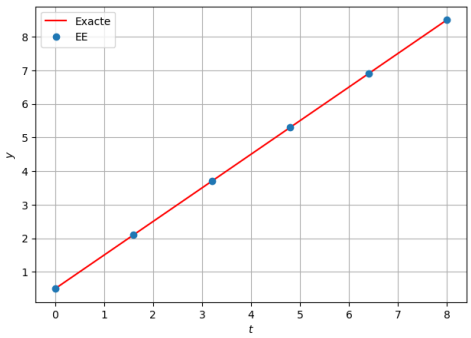


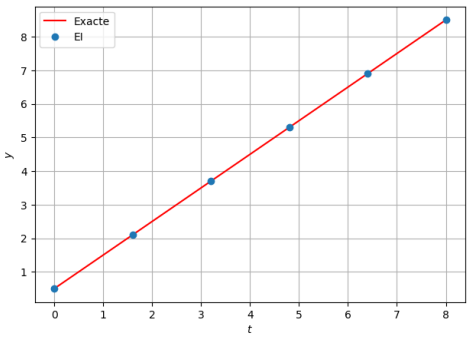
En effet, si on utilise les schémas d'ordre 1 proposés avec le même nombre de points de discrétisation on obtient

```
def EE(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.zeros_like(tt)
    uu[0] = y0
    for n in range(len(tt)-1):
        k1 = phi( tt[n], uu[n] )
        uu[n+1] = uu[n]+h*k1
    return uu

def EI(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.zeros_like(tt)
    uu[0] = y0
    for n in range(len(tt)-1):
         uu[n+1] = fsolve(lambda x: -x+uu[n]+h*phi(tt[n+1],x) , uu[n] )[0]
    return uu


fig,ax = plt.subplots(nrows=1,ncols=2,figsize=(16,5))
ax[0].plot(tt, yy, 'r-', label='Exacte')
ax[1].plot(tt, yy, 'r-', label='Exacte')
for i,schema in enumerate([EE, EI]):
    uu = schema(phi, tt, y0)
    ax[i].plot(tt, uu, 'o', label=schema.__name__)
    ax[i].set_xlabel('$t$')
    ax[i].set_ylabel('$y$')
    ax[i].grid(True)
    ax[i].legend();
```





Chapitre 35.

2021 Contrôle Terminal session 2

```
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp
from scipy.optimize import fsolve
from scipy.integrate import solve_ivp, odeint
from IPython.display import display, Markdown, Math

prlat= lambda *args: display(Math(''.join( sp.latex(arg) for arg in args )))
```

35.1. EXERCICE 1 : étude d'un schéma

On étudie le schéma de résolution du problème de Cauchy $y'(t) = \varphi(t, y(t))$ et $y(t_0) = y_0$ défini par la formule de récurrence

$$\begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n, u_n\right) \\ K_2 = \varphi\left(t_n + \frac{1}{2}h, u_n + \frac{1}{2}hK_1\right) \\ K_3 = \varphi\left(t_n + h, u_n + hK_1\right) \\ u_{n+1} = u_n + h\left(aK_1 + bK_2 + cK_3\right) \quad n = 0, 1, \ldots N-1 \end{cases}$$

où $a, b, c \in [0; 1]$.

1. Pour quelles valeurs du triplet (a, b, c) retrouve-t-on
   - la méthode d'Euler explicite (EE) ?
   - la méthode d'Euler modifiée (EM) ?
   - la méthode de Heun (H) ?
2. Le schéma peut être vu comme une méthode de type Runge-Kutta. Écrire la matrice de Butcher associée. Le schéma est explicite, semi-implicite ou implicite ?
3. Sans faire de calculs, indiquer quel est l'ordre maximale du schéma.
4. Étudier théoriquement l'ordre du schéma en fonction de a, b, c.
5. Étudier théoriquement la A-stabilité du schéma pour les valeurs du triplet (a, b, c) qui donnent l'ordre maximal.
6. **Pour la question 6 on va fixer (a, b, c). Pour cela, écrivez votre nom et prénom dans la cellule ci-dessous et vous obtiendrez un choix pour les paramètres.** Implémenter le schéma avec les valeurs a, b, c trouvées avec vos nom et prénom et approcher la solution du problème de Cauchy suivant avec $N+1$ points (quelles valers de N peut-on choisir ? justifier la réponse) Vérifier ensuite empiriquement l'ordre de convergence.

$$\begin{cases} y'(t) = -50y(t), \quad t \in [0; 1] \\ y(0) = 1 \end{cases}$$

Q1 [3 points]
Pour quelles valeurs du triplet (a, b, c) retrouve-t-on

- la méthode d'Euler explicite (EE) ?
- la méthode d'Euler modifiée (EM) ?
- la méthode de Heun (H) ?

- Si $(a, b, c) = (1, 0, 0)$ on retrouve la méthode d'Euler explicite (EE) :

$$u_{n+1} = u_n + hK_1 = u_n + h\varphi\left(t_n, u_n\right)$$

- Si $(a, b, c) = (0, 1, 0)$ on retrouve la méthode d'Euler modifiée (EM) :

$$u_{n+1} = u_n + hK_2 = u_n + h\varphi\left(t_n + \frac{1}{2}h, u_n + \frac{1}{2}h\varphi\left(t_n, u_n\right)\right)$$

- Si $(a, b, c) = (\frac{1}{2}, 0, \frac{1}{2})$ on retrouve la méthode de Heun (H) :

$$u_{n+1} = u_n + \frac{h}{2}(K_1 + K_3) = u_n + \frac{h}{2}\left(\varphi\left(t_n, u_n\right) + \varphi\left(t_n + h, u_n + h\varphi\left(t_n, u_n\right)\right)\right)$$

Q2 [2 points]
Le schéma peut être vu comme une méthode de type Runge-Kutta. Écrire la matrice de Butcher associée. Le schéma est explicite, semi-implicite ou implicite ?

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 1 & 1 & 0 & 0 \\ \hline & a & b & c \end{array}$$

Le schéma est explicite (la matrice $\mathbb{A}$ est triangulaire inférieure à diagonale nulle).

Q3 [1 points]
Sans faire de calculs, indiquer quel est l'ordre maximale du schéma.

Soit ω l'ordre de la méthode et $s = 3$ le nombre d'étages. Étant un schéma explicite, $\omega \leq s = 3$.

Q4 [3 points]
Étudier théoriquement l'ordre du schéma en fonction de a, b, c.

- Si $\begin{cases} \sum_{j=1}^{s} b_i = 1 \\ c_i = \sum_{j=1}^{s} a_{ij} \quad i = 1, \ldots, s \end{cases}$ alors $\omega \geq 1$
- Si $\omega \geq 1$ et $\sum_{j=1}^{s} b_j c_j = \frac{1}{2}$ alors $\omega \geq 2$
- Si $\omega \geq 2$ et $\begin{cases} \sum_{j=1}^{s} b_j c_j^2 = \frac{1}{3} \\ \sum_{i=1}^{s}\sum_{j=1}^{s} b_i a_{ij} c_j = \frac{1}{6} \end{cases}$ alors $\omega \geq 3$

Il est donc

- au moins d'ordre 1 ssi $a = 1 - b - c$
- il est d'ordre 2 ssi, de plus, $\frac{b}{2} + c = \frac{1}{2}$ donc ssi $\begin{cases} a = c \\ b = 1 - 2c \end{cases}$
- il ne peut pas être d'ordre 3 car la deuxième condition n'est jamais satisfaite.

On peut vérifier nos calculs comme ci-dessous :

```
def ordre_RK(s, A=None, b=None, c=None):

    j = sp.symbols('j')

    if A is None:
        A = sp.MatrixSymbol('a', s, s)
    else:
        A = sp.Matrix(A)
    if c is None:
        c = sp.symbols(f'c_0:{s}')
    if b is None:
        b = sp.symbols(f'b_0:{s}')


    display(Markdown("**Matrice de Butcher**"))
    But = matrice_Butcher(s, A, b, c)

    Eqs = []

    display(Markdown(f"**On a {s+1} conditions pour avoir consistance = pour être d'ordre 1**"))
    Eq_1 = ordre_1(s, A, b, c, j)
    Eqs.append(Eq_1)

    display(Markdown("**On doit ajouter 1 condition pour être d'ordre 2**"))
    Eq_2 = ordre_2(s, A, b, c, j)
    Eqs.append(Eq_2)

    display(Markdown("**On doit ajouter 2 conditions pour être d'ordre 3**"))
    Eq_3 = ordre_3(s, A, b, c, j)
    Eqs.extend(Eq_3)

    return Eqs


def matrice_Butcher(s, A, b, c):
    But = sp.Matrix(A)
    But = But.col_insert(0, sp.Matrix(c))
    last = [sp.Symbol(" ")]
    last.extend(b)
    But = But.row_insert(s,sp.Matrix(last).T)
    display(Math(sp.latex(sp.Matrix(But))))
    return But


def ordre_1(s, A, b, c, j):
    texte = r"\sum_{j=1}^s b_j =" + f"{sum(b).simplify()}"
    texte += r"\text{ doit être égale à }1"
    display(Math(texte))
    Eq_1 = sp.Eq( sum(b).simplify(), 1 )
    for i in range(s):
        somma = sp.summation(A[i,j],(j,0,s-1)).simplify()
        texte = r'\sum_{j=1}^s a_{'+str(i)+r'j}=' + sp.latex( somma )
        texte += r"\text{ doit être égale à }"+sp.latex(c[i])
        display(Math( texte ))
        Eq_1 = Eq_1 & sp.Eq( somma, c[i] )
    return Eq_1


def ordre_2(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j=' +sp.latex(sum([b[i]*c[i] for i in range(s)]).simplify())
    texte += r"\text{ doit être égale à }\frac{1}{2}"
    display(Math(texte))
    Eq_2 = sp.Eq( sum([b[i]*c[i] for i in range(s)]).simplify(), sp.Rational(1,2) )
```

```
    return Eq_2


def ordre_3(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j^2='
    texte += sp.latex( sum([b[i]*c[i]**2 for i in range(s)]).simplify() )
    texte += r"\text{ doit être égale à }\frac{1}{3}"
    display(Math(texte))
    Eq_3_1 = sp.Eq( sum([b[i]*c[i]**2 for i in range(s)]).simplify(), sp.Rational(1,3) )
    texte = r'\sum_{i,j=1}^s b_i a_{ij} c_j='
    somma = sum([b[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{6}"
    display(Math(texte))
    Eq_3_2 = sp.Eq( sum([b[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,6) )
    return [Eq_3_1, Eq_3_2]


################################################
# Conditions avec s étages
################################################

a,b,c = sp.symbols('a, b, c')

cc = [0, sp.Rational(1,2), 1]
bb = [a, b, c]
A = [ [cc[0], 0, 0],
      [cc[1], 0, 0],
      [cc[2], 0, 0]]
s = len(cc)

Eqs = ordre_RK(s, A, bb, cc)

display(Markdown("**Solution pour être d'ordre 2 :**"))
sol = sp.solve(Eqs[:3])
sol
```

Matrice de Butcher

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 1 & 1 & 0 & 0 \\ & a & b & c \end{bmatrix}$$

On a 4 conditions pour avoir consistance = pour être d'ordre 1

$\sum_{j=1}^{s} b_j = a + b + c$ doit être égale à 1

$\sum_{j=1}^{s} a_{0j} = 0$ doit être égale à 0

$\sum_{j=1}^{s} a_{1j} = \frac{1}{2}$ doit être égale à $\frac{1}{2}$

$\sum_{j=1}^{s} a_{2j} = 1$ doit être égale à 1

On doit ajouter 1 condition pour être d'ordre 2

$\sum_{j=1}^{s} b_j c_j = \frac{b}{2} + c$ doit être égale à $\frac{1}{2}$

On doit ajouter 2 conditions pour être d'ordre 3

$\sum_{j=1}^{s} b_j c_j^2 = \frac{b}{4} + c$ doit être égale à $\frac{1}{3}$

$\sum_{i,j=1}^{s} b_i a_{ij} c_j = 0$ doit être égale à $\frac{1}{6}$

Solution pour être d'ordre 2 :

```
{a: 1/6, b: 2/3, c: 1/6}
```

> **Q5 [2 points]**
> Étudier théoriquement la A-stabilité du schéma pour les valeurs du triplet (a, b, c) qui donnent l'ordre maximal.

Soit $\beta > 0$ un nombre réel positif et considérons le problème de Cauchy

$$\begin{cases} y'(t) = -\beta y(t), & \text{pour } t > 0, \\ y(0) = 1. \end{cases}$$

Sa solution est $y(t) = e^{-\beta t}$ donc

$$\lim_{t \to +\infty} |y(t)| = 0.$$

Le schéma appliqué à ce problème de Cauchy s'écrit

$$\begin{cases} u_0 = y_0 \\ K_1 = -\beta u_n \\ K_2 = -\beta\left(u_n + \frac{1}{2} h K_1\right) = -\beta\left(u_n - \frac{1}{2}\beta h u_n\right) \\ K_3 = -\beta\left(u_n + h K_1\right) = -\beta\left(u_n - \beta h u_n\right) \\ u_{n+1} = u_n + h\left(a K_1 + b K_2 + c K_3\right) \qquad\qquad n = 0, 1, \ldots N-1 \end{cases}$$

d'où

$$u_{n+1} = \left(1 - (a+b+c)\beta h + \left(\frac{b}{2} + c\right)(\beta h)^2\right) u_n$$

L'ordre maximal (i.e. l'ordre 2) est obtenu pour $a + b + c = 1$ et $b = 1 - 2c$, on trouve alors

$$u_{n+1} = \left(\frac{1}{2}(\beta h)^2 - (\beta h) + 1\right) u_n$$

Vérifions ces calculs :

```
u_n = sp.Symbol('u_n')
beta = sp.Symbol(r'\beta', positive=True)
dt = sp.Symbol('h', positive=True) # pour ne pas effacer l'affectation de "h", ici je vais l'appeler "dt"

K1 = -beta*u_n
display(Math('K_1='+sp.latex(K1)))
K2 = -beta*(u_n+sp.Rational(1,2)*dt*K1).factor()
display(Math('K_2='+sp.latex(K2)))
K3 = -beta*(u_n+dt*K1).factor()
display(Math('K_3='+sp.latex(K3)))
u_np1 = (u_n+dt*(a*K1+b*K2+c*K3)).factor()
display(Math('u_{n+1}='+sp.latex(u_np1)))
display(Markdown("Avec les paramètres qui donnent l'ordre 2 on obtient"))
u_np1= u_np1.subs(sol).simplify()
display(Math('u_{n+1}='+sp.latex(u_np1)))
```

$K_1 = -\beta u_n$

$K_2 = \dfrac{\beta u_n \left(\beta h - 2\right)}{2}$

$K_3 = \beta u_n \left(\beta h - 1\right)$

$u_{n+1} = -\dfrac{u_n \left(-\beta^2 b h^2 - 2\beta^2 c h^2 + 2\beta a h + 2\beta b h + 2\beta c h - 2\right)}{2}$

Avec les paramètres qui donnent l'ordre 2 on obtient

$u_{n+1} = \dfrac{u_n \left(\beta^2 h^2 - 2\beta h + 2\right)}{2}$

On note $x = \beta h$ et on étudie la fonction $q\colon \mathbb{R}^+ \to \mathbb{R}$ définie par $q(x) = \frac{1}{2}x^2 - x + 1$ car le schéma est A-stable ssi $|q(x)| < 1$.
Il s'agit d'une parabole convexe de sommet en $x_s = 1$ et $q(x_s) = \frac{1}{2} > -1$ donc $q(x) < 1$ pour $0 < x < 2$ et $q(x) > 0$ pour tout x. On conclut que le schéma est A-stable pour tout $h < \dfrac{2}{\beta}$.

On peut vérifier les calculs avec `sympy` :

```
x = sp.Symbol('x', positive=True)

q = (u_np1/u_n).simplify().subs(beta*dt,x)
display(Math(sp.latex(sp.Eq(sp.Symbol('q(x)'),q.simplify()))))

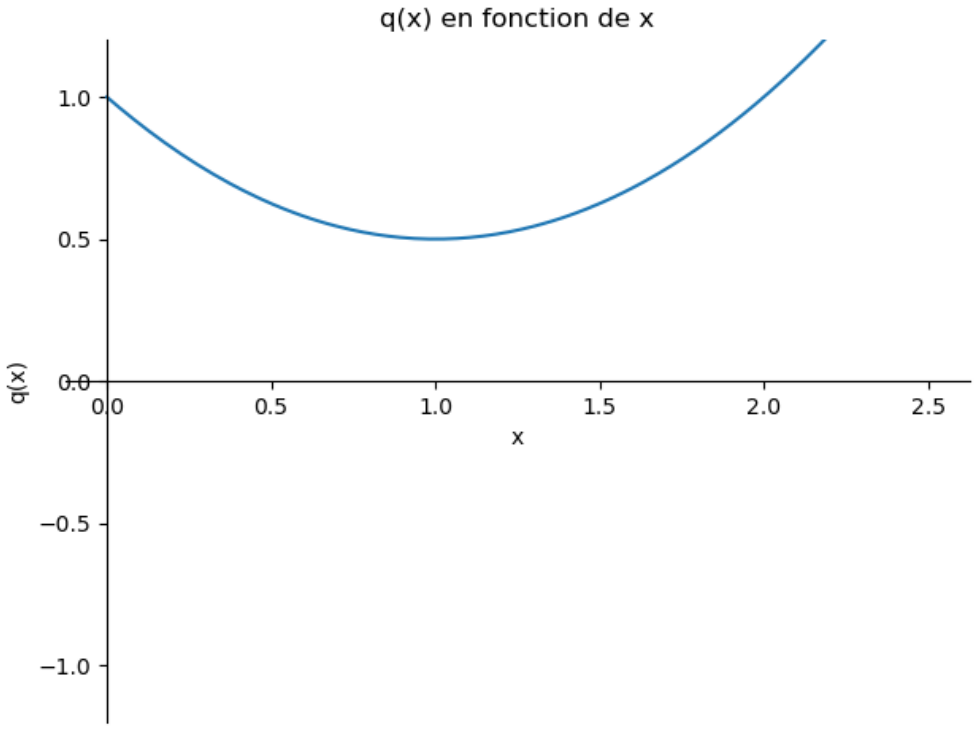
display(Markdown("$|q(x)|<1$ ssi"))
display(Math(sp.latex(sp.solve(sp.Abs(q)<1))))

sp.plot(q, (x,0,2.5), ylim=(-1.2,1.2),
            ylabel='q(x)', xlabel='x', title='q(x) en fonction de x');
```

$q(x) = \dfrac{x^2}{2} - x + 1$

$|q(x)| < 1$ ssi

$x < 2$



Pour la question 6 on va fixer (a, b, c). Pour cela, écrivez votre nom et prénom dans la cellule ci-dessous et vous obtiendrez un choix pour les paramètres.

```
nom = "Faccanoni"
prenom = "Gloria"

DEN = list(range(3,11))
idx = sum([ord(c) for c in nom+prenom])%len(DEN)
my_c = sp.Rational(1,DEN[idx])
my_b = 1-2*my_c
display(Math(r'a='+sp.latex(my_c)+',\\quad b='+sp.latex(my_c)+',\\quad  c='+sp.latex(my_b)))


a, b, c = float(my_c), float(my_c), float(my_b)
```

$$a = \frac{1}{3}, \quad b = \frac{1}{3}, \quad c = \frac{1}{3}$$

Q6 [2 points]
Implémenter le schéma avec les valeurs a, b, c trouvées avec vos nom et prénom et approcher la solution du problème de Cauchy suivant avec $N + 1$ points (quelles valers de N peut-on choisir ? justifier la réponse)

$$\begin{cases} y'(t) = -50y(t), & t \in [0; 1] \\ y(0) = 1 \end{cases}$$

Vérifier ensuite empiriquement l'ordre de convergence.

Pour la A-stabilité il faut que h soit $< \frac{2}{50}$ donc $N > \frac{1-0}{h} = \frac{50}{2}$

```
# ================================================
# SCHÉMA RK
# ================================================

# def RK(tt,phi,y0): # comme suite geometrique
#     h = tt[1]-tt[0]
#     x = beta*h
#     q = x**2/2-x+1
#     uu = y0*q**np.arange(len(tt))
#     return uu

def RK(tt,phi,y0):
    uu = np.empty_like(tt)
    uu[0] = y0
    h = tt[1]-tt[0]
    for n in range(len(tt)-1):
        K1 = phi(tt[n],uu[n])
        K2 = phi(tt[n]+h/2,uu[n]+h/2*K1)
        K3 = phi(tt[n+1],uu[n]+h*K1)
        uu[n+1] = uu[n]+h*(a*K1+b*K2+c*K3)
    return uu


# ================================================
# EDO
# ================================================
t0, y0, tfinal = 0, 1, 1
beta = 5

phi = lambda t,y : -beta*y
sol_exacte = lambda t : np.exp(-beta*t)

_, ax = plt.subplots(nrows=1,ncols=2,figsize=(15,5))
```

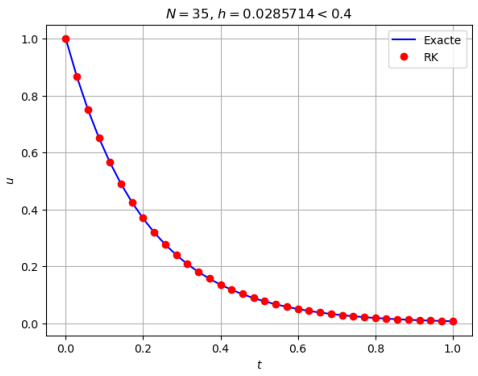
```
# ================================================
# UN TEST
# ================================================
N = int(50/2)+10
tt = np.linspace(t0,tfinal,N+1)
h = tt[1]-tt[0]

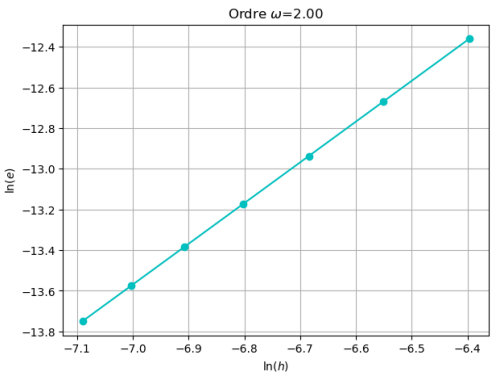
if h>=2/beta:
    print('Attention: h > limite de stabilité')

yy = sol_exacte(tt)
uu = RK(tt,phi,y0)
ax[0].plot(tt,yy,'b-',label=("Exacte"))
ax[0].plot(tt,uu,'ro',label=("RK"))
ax[0].set_title(f'${N=}$, ${h=:g}<{2/beta:g}$')
ax[0].set_xlabel('$t$')
ax[0].set_ylabel('$u$')
ax[0].legend()
ax[0].grid(True)

# ================================================
# CONVERGENCE
# ================================================
H = []
err = []
N = 500
for k in range(7):
    N += 100
    tt = np.linspace(t0, tfinal, N + 1)
    h = tt[1] - tt[0]
    yy = sol_exacte(tt)
    uu = RK(tt,phi,y0)
    H.append(h)
    err.append(np.linalg.norm(yy-uu,np.inf))

omega = np.polyfit(np.log(H),np.log(err), 1)[0]
ax[1].plot(np.log(H), np.log(err), 'c-o')
ax[1].set_xlabel(r'$\ln(h)$')
ax[1].set_ylabel(r'$\ln(e)$')
ax[1].set_title(rf'Ordre $\omega$={omega:1.2f}')
ax[1].grid(True);
```





35.2. EXERCICE 2 : étude d'un schéma multipas

On notera $\varphi_k \overset{\text{déf}}{=} \varphi(t_k, u_k)$.
Soit la méthode multipas

$$u_{n+1} = u_{n-1} + h\left(b_{-1}\varphi_{n+1} + b_0\varphi_n + b_1\varphi_{n-1}\right).$$

1. Pour quelles valeurs des paramètres b_0, b_1, b_2 est-elle zéro-stable ?
2. Pour quelles valeurs des paramètres la méthode est convergente d'ordre 4 ?
3. Vérifier empiriquement l'ordre de convergence sur le problème de Cauchy suivant après en avoir calculé la solution exacte :
$$\begin{cases} y'(t) = y(t)\ln(1+y(t)), \quad \forall t \in I = [0,1], \\ y(0) = 2, \end{cases}$$

Q1 [2 points]
Pour quelles valeurs des paramètres b_0, b_1, b_2 est-elle zéro-stable ?

Le premier polynôme caractéristique est

$$\varrho(r) = r^{p+1} - \sum_{j=0}^{p} a_j r^{p-j} = r^2 - a_0 r - a_1 = r^2 - 1 = (r-1)(r+1)$$

dont les racines sont

$$r_0 = 1, \quad r_1 = -1.$$

La méthode est donc zéro-stable pour tout choix des paramètres car

$$\begin{cases} |r_j| \leq 1 \quad \text{pour tout } j = 0, \dots, p \\ \varrho'(r_j) \neq 0 \text{ si } |r_j| = 1. \end{cases}$$

Q2 [3 points]
Pour quelles valeurs des paramètres la méthode est convergente d'ordre 4 ?

- Une méthode consistante d'ordre $\omega \geq 1$ et zéro-stable est convergente d'ordre ω.
- On a $p = 1$: c'est une méthode à $q = p + 1 = 2$ pas avec $a_0 = 0$ et $a_1 = 1$.
- La méthode est implicite si $b_{-1} \neq 0$.

La première barrière de Dahlquist affirme qu'un schéma implicite à $q = 2$ étapes consistante et zéro-stable peut être au mieux d'ordre $\omega = q + 2 = 4$.

Pour que la méthode soit consistante d'ordre 4 il faut que

$$\begin{cases} \xi(0) = a_0 + a_1 = 1 \\ \xi(1) = b_0 + b_1 + b_{-1} - 1 = 1 \\ \xi(2) = 1 - 2\left(b_1 - b_{-1}\right) = 1 \\ \xi(3) = 3\left(b_1 + b_{-1}\right) - 1 = 1 \\ \xi(4) = 1 - 4\left(b_1 - b_{-1}\right) = 1 \end{cases}$$

Dans notre cas cela reviens à imposer

$$\begin{cases} 0 + 1 = 1 \\ -1 + (b_{-1} + b_0 + b_1) = 1 \\ 1 - 2(-b_{-1} + b_1) = 1 \\ -1 + 3(b_{-1} + b_1) = 1 \\ 1 - 4(-b_{-1} + b_1) = 1 \end{cases}$$

On cherche donc b_{-1}, b_0 et b_1 tels que

$$\begin{pmatrix} 1 & 1 & 1 \\ 2 & 0 & -2 \\ 3 & 0 & 3 \\ 4 & 0 & -4 \end{pmatrix} \begin{pmatrix} b_{-1} \\ b_0 \\ b_1 \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \\ 2 \\ 0 \end{pmatrix}$$

Il s'agit d'un système surdeterminé qui admet l'unique solution

$$b_{-1} = b_1 = \frac{1}{3}, \qquad b_0 = \frac{4}{3}$$

```
q = 2

def xi(i, q):
    sa = sum((-j)**i * aa[j] for j in range(q))
    sb = b_m1 + sum((-j)**(i-1) * bb[j] for j in range(1, q))
    if i == 1:
        sb += bb[0]
    return (sa) + (i * sb).factor()

# Définition des symboles
b_0, b_1  = sp.symbols('b_0 b_1 ', real=True)
b_m1 = sp.symbols('b_{-1}', real=True)
aa = [0, 1]
bb = [b_0, b_1]

systeme = [sp.Eq(xi(i, q),1) for i in range(2*q+1)]
display(Markdown(r"$\begin{cases}"+' '.join([ fr"\xi({i})="  + sp.latex(s)+r"\\" for i,s in
 ↪  enumerate(systeme)]) + r"\end{cases}$"))

sol = sp.solve(systeme)
display(sol)
```

$$\begin{cases}\xi(0) = \text{True}\\ \xi(1) = b_{0} + b_{1} + b_{-1} - 1 = 1\\ \xi(2) = 1 - 2\left(b_{1} - b_{-1}\right) = 1\\ \xi(3) = 3\left(b_{1} + b_{-1}\right) - 1 = 1\\ \xi(4) = 1 - 4\left(b_{1} - b_{-1}\right) = 1\end{cases}$$

```
{b_0: 4/3, b_1: 1/3, b_{-1}: 1/3}
```

Q3 [3 points]
Vérifier empiriquement l'ordre de convergence sur le problème de Cauchy

$$\begin{cases}y'(t) = y(t)\ln(1+y(t)), \quad \forall t \in I = [0,1],\\ y(0) = 2,\end{cases}$$

après en avoir calculé la solution exacte.

- On définit la solution exacte (calculée en utilisant le module `sympy`) pour estimer les erreurs :

```
t0     = 0
tfinal = 1
y0     = 2

phi = lambda t,y : y*np.log(1/y)
```

```
t  = sp.Symbol('t')
y  = sp.Function('y')
edo = sp.Eq( sp.diff(y(t),t) , y(t)*sp.log(1/y(t)) )
display(edo)
solpar = sp.dsolve(edo, y(t), ics={y(t0):y0})
display(solpar)

sol_exacte = sp.lambdify(t,solpar.rhs,'numpy')
```

$$\frac{d}{dt}y(t) = y(t)\log\left(\frac{1}{y(t)}\right)$$

$$y(t) = e^{e^{-t}\log(2)}$$

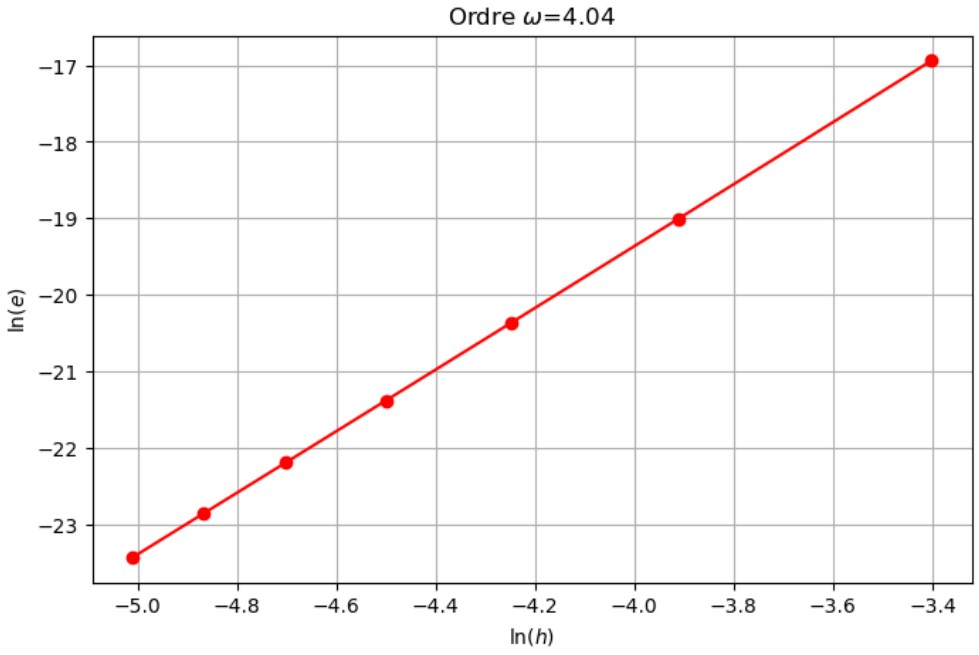
- On calcule la solution approchée pour différentes valeurs de N :

```
def multipas(phi, tt, y0):
    h  = tt[1] - tt[0]
    uu = np.empty_like(tt)
    uu[0] = y0
    uu[:q] = sol_exacte(tt[:q])
    for n in range(q-1,len(tt) - 1):
        temp = fsolve( lambda x : -x+uu[n-1]+h/3*(phi(tt[n+1],x)+4*phi(tt[n],uu[n])+phi(tt[n-1],uu[n-1]))
 ↪  ,uu[n])
        uu[n+1] = temp[0]
    return uu

H = []
err_mp = []
N = 10
for k in range(7):
    N += 20
    tt = np.linspace(t0, tfinal, N + 1)
    h = tt[1] - tt[0]
    yy = sol_exacte(tt)
    uu_mp = multipas(phi, tt, y0)
    H.append(h)
    err_mp.append(np.linalg.norm(yy-uu_mp,np.inf))

omega = np.polyfit(np.log(H),np.log(err_mp), 1)[0]

plt.figure(figsize=(8,5))
plt.plot(np.log(H), np.log(err_mp), 'r-o')
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
plt.grid(True)
plt.title(rf'Ordre $\omega$={omega:1.2f}');
```



Chapitre 36.

2020 Devoir Maison

```
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp
from scipy.optimize import fsolve
from IPython.display import display, Markdown, Math
```

36.1. EXERCICE 1 : implémentation et comparaison de schémas

Le déplacement $x(t)$ d'un système oscillant composé d'une masse et d'un ressort, soumis à une force de frottement proportionnelle à la vitesse, est décrit par l'équation différentielle du second ordre

$$x''(t) + 5x'(t) + 6x(t) = 0,$$

avec $x(0) = 1$ et $x'(0) = 0$, pour $t \in [0, 5]$.

1. Écrire le système sous forme matricielle. Calculer ensuite la **solution exacte** avec `sympy`. Afficher $t \mapsto x(t)$, $t \mapsto x'(t)$ et $x \mapsto x'(x)$.
2. Calculer la solution approchée obtenue par la méthode d'**Euler Explicite** avec 301 points. Afficher $t \mapsto x(t)$, $t \mapsto x'(t)$ et $x \mapsto x'(x)$ en comparant solution exacte et solution approchée.
3. Même exercice avec la méthode d'**Euler Implicite**.
4. Même exercice avec la fonction `odeint` du module `scipy`.

Q1
Écrire le système sous forme matricielle. Calculer ensuite la solution exacte avec `sympy`. Afficher $t \mapsto x(t)$, $t \mapsto x'(t)$ et $x \mapsto x'(x)$.

Si on note $y(t) = x(t)$ et $z(t) = x'(t)$ on a $y(0) = 1$, $z(0) = 0$ et

$$\begin{cases} y'(t) = z(t), \\ z'(t) = -6y(t) - 5z(t) \end{cases} \qquad \text{i.e.} \quad \begin{pmatrix} y \\ z \end{pmatrix}' (t) = \begin{pmatrix} 0 & 1 \\ -6 & -5 \end{pmatrix} \begin{pmatrix} y \\ z \end{pmatrix} (t)$$

Autrement dit :

$$\begin{aligned} \varphi &: \mathbb{R} \times \mathbb{R}^2 \to \mathbb{R} \\ \Big(t, (y, z)\Big) &\mapsto \begin{pmatrix} \varphi_1(t, (y, z)) = -z \\ \varphi_2(t, (y, z)) = -6y - 5z \end{pmatrix} \end{aligned}$$

```
t0 = 0
tfinal = 5

# Notation vectorielle
# ====================

# CI (2 conditions)
```

```
zz0 = np.array([1,0])

# EDO (2 equations)
# phi : RxR^2 -> R^2
phi = lambda t,zz : np.array( [ zz[1], -6*zz[0]-5*zz[1] ] )
```

Pour simplifier les affichages, on définit une fonction `affichage` qui affiche les graphes de y_n et z_n en fonction de t_n et le portrait de phase (y_n, z_n). Si une solution approchée est passé en argument, elle est aussi affichée pour comparaison.

Il suffira alors d'appeler cette fonction pour chaque schéma.

```
# Discrétisation
# ==============
N = 300
tt = np.linspace(t0,tfinal,N+1)


# FONCTION UTILE

def affichage(tt, zz_exacte, zz=None, s=""):

    # zz_exacte[:, 0] = y(t) = x(t)
    # zz_exacte[:, 1] = z(t) = x'(t)

    # zz[:, 0] = approx de zz_exacte[:, 0]
    # zz[:, 1] = approx de zz_exacte[:, 1]

    plt.figure(figsize=(17,7))

    # Graphe de y(t) et z(t)
    plt.subplot(1,2,1)
    if zz is not None:
        plt.plot(tt,zz[:, 0],'--',label=r'$u(t) \approx y(t)$',color="red")
        plt.plot(tt,zz[:, 1],'--',label=r'$w(t) \approx z(t)$',color="blue")
    plt.plot(tt,zz_exacte[:, 0],label=r"$y(t)=x(t)$",color="red")
    plt.plot(tt,zz_exacte[:, 1],label=r"$z(t)=x'(t)$",color="blue")
    plt.xlabel('t')
    plt.legend()
    plt.title(f'{s} - $y(t)$ et $z(t)$')
    plt.grid()

    # Diagramme de phase z(y)
    plt.subplot(1,2,2)
    if zz is not None:
        plt.plot(zz[:, 0], zz[:, 1],'--',label='Approchée')
    plt.plot(zz_exacte[:, 0], zz_exacte[:, 1],label='Exacte')
    plt.xlabel('y')
    plt.ylabel('z')
    plt.legend()
    if s!="":
        plt.title(f'{s} - z(y)')
    plt.grid()
    plt.axis('equal');
```

La solution exacte est

$$x(t) = y(t) = 3e^{-2t} - 2e^{-3t},$$
$$x'(t) = z(t) = -6e^{-2t} + 6e^{-3t}.$$

Vérifions-le avec le module `sympy` :

```
# Stratégie 1 : comme 1 EDO d'ordre 2
# ====================================
t = sp.Symbol('t')
x = sp.Function('x')(t)
t0 = 0
edo = sp.Eq( x.diff(t,t) + 5*x.diff(t) + 6*x , 0 )
sol = sp.dsolve(edo, x, ics={x.subs(t,t0):1,
                             x.diff(t).subs(t,t0):0})
display(sol)

func_1 = sp.lambdify(t,sol.rhs,'numpy')
func_2 = sp.lambdify(t,sol.rhs.diff(t),'numpy')

# Stratégie 2 : comme 2 EDO d'ordre 1
# ====================================
# t = sp.Symbol('t')
# y = sp.Function('y')
# z = sp.Function('z')

# t0 = 0
# y0 = zz0[0]
# z0 = zz0[1]

# phi1 = lambda t,y,z : phi(t,[y,z])[0]
# phi2 = lambda t,y,z : phi(t,[y,z])[1]

# edo1 = sp.Eq( sp.diff(y(t),t) , phi1(t,y(t),z(t)) )
# edo2 = sp.Eq( sp.diff(z(t),t) , phi2(t,y(t),z(t)) )
# display(edo1)
# display(edo2)

# solpar_1, solpar_2 = sp.dsolve([edo1,edo2],
#                                [y(t),z(t)],
#                                ics={y(t0):y0, z(t0):z0})
# display(solpar_1)
# display(solpar_2)

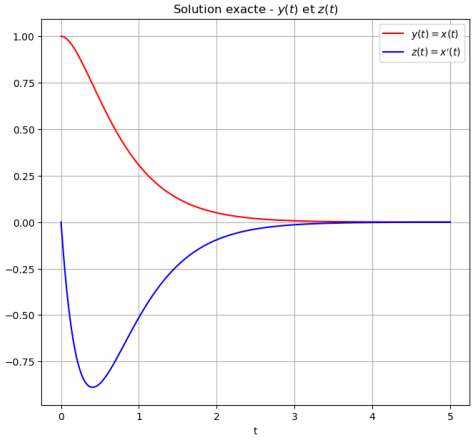
# func_1 = sp.lambdify(t,solpar_1.rhs,'numpy')
# func_2 = sp.lambdify(t,solpar_2.rhs,'numpy')

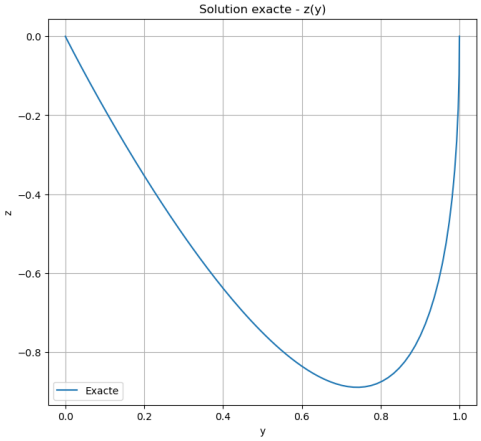


# EVALUATION ET AFFICHAGE

y_vec = func_1(tt)
z_vec = func_2(tt)
zz_exacte = np.array([y_vec,z_vec]).T
affichage(tt,zz_exacte,s="Solution exacte")
```

$x(t) = (3 - 2e^{-t})\, e^{-2t}$





Q2
Calculer la solution approchée obtenue par la méthode d'Euler Progressif avec 301 points. Afficher $t \mapsto x(t)$, $t \mapsto x'(t)$ et $x \mapsto x'(x)$ en comparant solution exacte et solution approchée.

On notera $u_n \approx y_n = x(t_n)$ et $w_n \approx z(t_n) = x'(t_n)$.

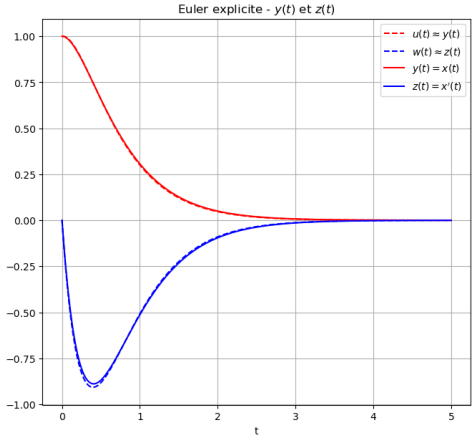
Euler explicite

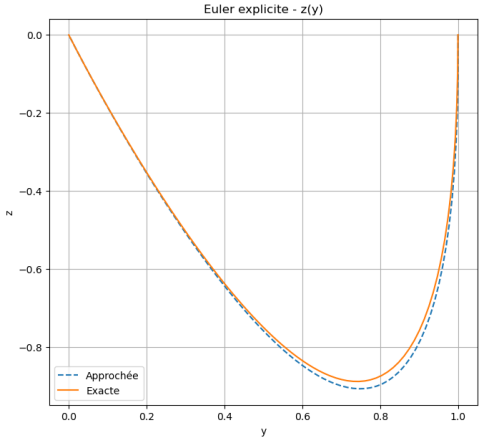
$$\begin{cases} u_0 = 1, \\ w_0 = 0, \\ u_{n+1} = u_n + h\varphi_1(t_n, u_n, w_n), \\ w_{n+1} = w_n + h\varphi_2(t_n, u_n, w_n). \end{cases}$$

```
def EE(phi,tt,zz0):
    nb_eq = len(zz0) # nombre d'équations
    N = len(tt)-1 # N+1 points, donc N intervalles
    zz = np.zeros((N+1,nb_eq)) # colonnes = variables, lignes = temps
    zz[0] = zz0
    h = tt[1]-tt[0]
    for n in range(N):
        zz[n+1] = zz[n]+h*phi(tt[n],zz[n])
    return zz


# Version qui retourne un tableau 1D si zz0 est scalaire
# def EE(phi, tt, zz0):
#     zz0 = np.atleast_1d(zz0)  # Convertir zz0 en tableau (même si scalaire)
#     nb_eq = len(zz0)  # Nombre d'équations
#     N = len(tt) - 1  # N+1 points, donc N intervalles
#     zz = np.zeros((N + 1, nb_eq))  # Colonnes = variables, lignes = temps
#     zz[0] = zz0
#     h = tt[1] - tt[0]
#     for n in range(N):
#         zz[n + 1] = zz[n] + h * phi(tt[n], zz[n])
#     if nb_eq == 1:
#         return zz.flatten()  # Retourner un tableau 1D si scalaire
#     return zz

zz_ep = EE(phi,tt,zz0)
affichage(tt,zz_exacte,zz=zz_ep,s="Euler explicite");
```





Q3
Calculer la solution approchée obtenue par la méthode d'Euler Regressif avec 301 points. Afficher $t \mapsto x(t)$, $t \mapsto x'(t)$ et $x \mapsto x'(x)$ en comparant solution exacte et solution approchée.

Euler implicite

$$\begin{cases} u_0 = 1, \\ w_0 = 0, \\ u_{n+1} = u_n + h\varphi_1(t_{n+1}, u_{n+1}, w_{n+1}), \\ w_{n+1} = w_n + h\varphi_2(t_{n+1}, u_{n+1}, w_{n+1}), \end{cases}$$

qu'on peut rendre explicite par un calcul élementaire :

$$w_{n+1} = w_n + h(-6u_{n+1} - 5w_{n+1}) = w_n + h(-6(u_n + hw_{n+1}) - 5w_{n+1}) \quad \Longrightarrow \quad (1 + 5h + 6h^2)w_{n+1} = w_n - 6hu_n$$

ainsi

$$\begin{cases} u_0 = 1, \\ w_0 = 0, \\ u_{n+1} = u_n + h\dfrac{w_n - 6hu_n}{1 + 5h + 6h^2}, \\ w_{n+1} = \dfrac{w_n - 6hu_n}{1 + 5h + 6h^2}. \end{cases}$$

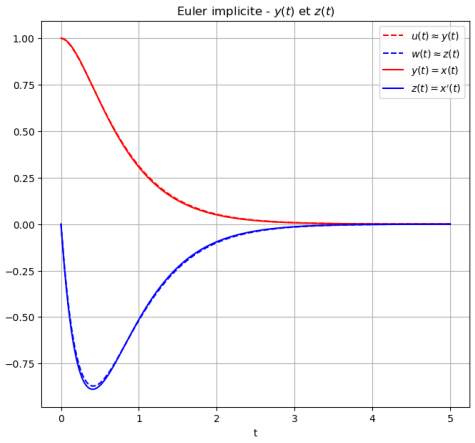
```
# VERSION explicite
def EI(phi,tt,zz0):
    h = tt[1]-tt[0]
    uu, ww = np.zeros(len(tt)), np.zeros(len(tt))
    uu[0], ww[0] = zz0
    for n in range(len(tt)-1):
        uu[n+1] = uu[n] + h*(ww[n]-6*h*uu[n])/(1+5*h+6*h**2)
        ww[n+1] = (ww[n]-6*h*uu[n])/(1+5*h+6*h**2)
    return np.array([uu,ww]).T

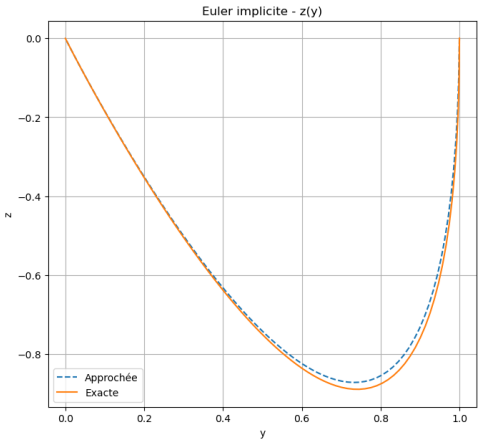

#  VERSION AVEC fsolve
# from scipy.optimize import fsolve

# def EI(phi,tt,zz0):
#    nb_eq = len(zz0) # nombre d'équations
#    N = len(tt)-1 # N+1 points, donc N intervalles
#    zz = np.zeros((N+1,nb_eq))
#    zz[0] = zz0
#    h = tt[1]-tt[0]
#    for n in range(N):
#        system = lambda z : z - zz[n] - h*phi(tt[n+1],z)
#        zz[n+1] = fsolve( system , zz[n] )
#    return zz
```

```
# Version qui retourne un tableau 1D si zz0 est scalaire
# def EI(phi, tt, zz0):
#     zz0 = np.atleast_1d(zz0)  # Convertir zz0 en tableau (même si scalaire)
#     nb_eq = len(zz0)  # Nombre d'équations
#     N = len(tt) - 1  # N+1 points, donc N intervalles
#     zz = np.zeros((N + 1, nb_eq))
#     zz[0] = zz0
#     h = tt[1] - tt[0]
#     for n in range(N):
#         system = lambda z: z - zz[n] - h * phi(tt[n + 1], z)
#         zz[n + 1] = fsolve(system, zz[n])
#     if nb_eq == 1:
#         return zz.flatten()  # Retourner un tableau 1D si scalaire
#     return zz

zz_ei = EI(phi,tt,zz0)
affichage(tt,zz_exacte,zz=zz_ei,s="Euler implicite");
```



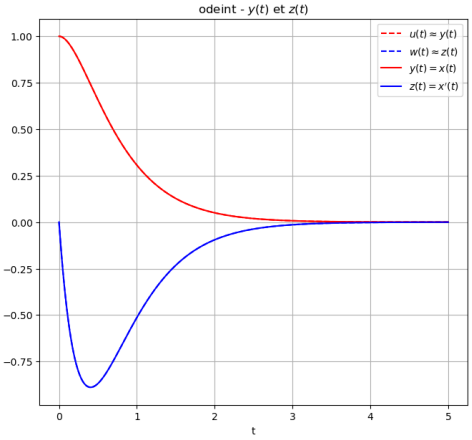


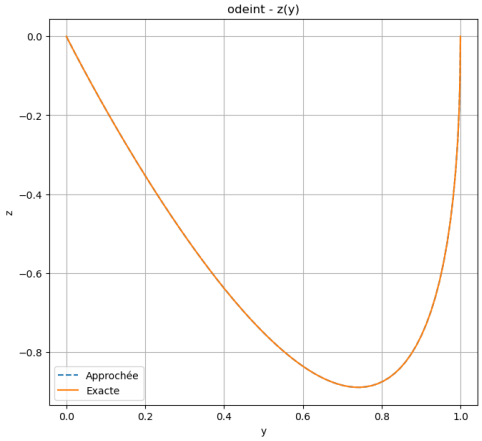
Q4
Calculer la solution approchée obtenue par la fonction `odeint` du module `scipy`. Afficher $t \mapsto x(t)$, $t \mapsto x'(t)$ et $x \mapsto x'(x)$ en comparant solution exacte et solution approchée.

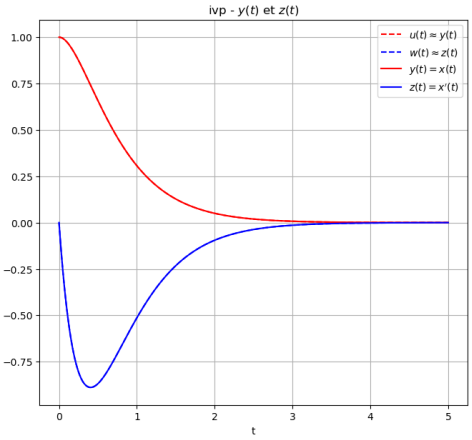
```
# OLD
from scipy.integrate import odeint

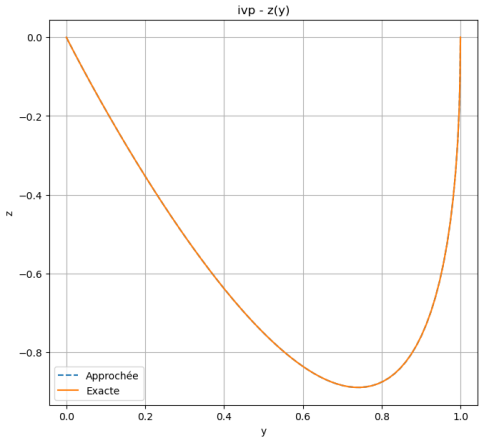
zz_odeint = odeint(phi,zz0,tt,tfirst=True)
affichage(tt,zz_exacte,zz=zz_odeint,s="odeint")

# NEW
from scipy.integrate import solve_ivp
zz_ivp = solve_ivp(phi,[t0,tfinal],zz0, t_eval=tt,
                   method='RK45'
                   # method='BDF'
                   ).y.T
affichage(tt,zz_exacte,zz=zz_ivp,s="ivp")
```









36.2. EXERCICE 2 : étude d'un schéma RK semi-implicite à 2 étages

Soit le schéma de Runge-Kutta dont la matrice de Butcher est

$$\begin{array}{c|cc} \frac{1}{3} & \frac{1}{3} & 0 \\ 1 & 1 & 0 \\ \hline & \frac{3}{4} & \frac{1}{4} \end{array}$$

1. Écrire le schéma sous la forme d'une **suite définie par récurrence**.
2. Étudier **théoriquement** l'**ordre** de ce schéma.
3. Étudier **théoriquement** la **A-stabilité** de ce schéma.
4. **Implémenter** ce schéma et le **tester** sur le problème de Cauchy suivant avec $N+1$ points et $N=8$:

$$\begin{cases} y'(t) = -6y(t), & t \in [0;1] \\ y(0) = 1 \end{cases}$$

Q5 Écrire le schéma sous la forme d'une suite définie par récurrence.

Le schéma associé à cette matrice est semi-implicite (K_0 dépend de lui même) et permet de calculer u_{n+1} à partir de u_n par la formule de récurrence

$$\begin{cases} u_0 = y_0 \\ K_0 = \varphi\left(t_n + \frac{h}{3}, u_n + \frac{h}{3}K_0\right) \\ K_1 = \varphi\left(t_{n+1}, u_n + hK_0\right) \\ u_{n+1} = u_n + \frac{h}{4}\left(3K_0 + K_1\right) \end{cases} \quad n = 0, 1, \dots N-1$$

Q6 Étudier théoriquement l'ordre du schéma.

Soit ω l'ordre de la méthode.

C'est un schéma semi-implicite à 2 étages ($s = 2$) donc $\omega \leq 2s = 4$

```
# ========================================================================
# Fonctions pour calculer les conditions de consistance pour un schéma RK.
#
# La fonction ordre_RK calcule les conditions pour les ordres 1, 2, 3 et 4 (même si théoriquement cela
 ↪ n'est pas possible)
# - affiche la matrice de Butcher
# - affiche les conditions de consistance obtenues pour chaque ordre par la fonction ordre_i
# - renvoie une liste d'équations pour une éventuelle résolution
#
# Chaque fonction ordre_i calcule les conditions pour être d'ordre i
# - peut afficher les conditions de consistance [actuellement en commentaire]
# - renvoie la liste d'équations
#
# ========================================================================

prlat = lambda *args : display(Math(''.join( sp.latex(arg) for arg in args )))

def ordre_RK(s, type="implicite", A=None, b=None, c=None):

    j = sp.symbols('j')

    if A is None:
        if type == "implicite":
            A = sp.MatrixSymbol('a', s, s)
        elif type=="semi-implicite":
            A = sp.Matrix(s, s, lambda i,j: sp.Symbol('a{}{}'.format(i, j)) if i >= j else 0)
        elif type=="explicite":
            A = sp.Matrix(s, s, lambda i,j: sp.Symbol('a{}{}'.format(i, j)) if i > j else 0)

    else:
        A = sp.Matrix(A)
    if c is None:
        c = sp.symbols(f'c_0:{s}')
    if b is None:
        b = sp.symbols(f'b_0:{s}')



    display(Markdown("**Matrice de Butcher**"))
    But = matrice_Butcher(s, A, b, c)

    Eqs = []

    display(Markdown(f"**On a {s+1} conditions pour avoir consistance (= pour être d'ordre 1)**"))
    Eq_1 = ordre_1(s, A, b, c, j)
    Eqs.append(Eq_1)
    display(*Eq_1)

    display(Markdown("**On doit ajouter 1 condition pour être d'ordre 2**"))
    Eq_2 = ordre_2(s, A, b, c, j)
    Eqs.append([Eq_2])
    display(Eq_2)

    display(Markdown("**On doit ajouter 2 conditions pour être d'ordre 3**"))
    Eq_3 = ordre_3(s, A, b, c, j)
    Eqs.append(Eq_3)
    display(*Eq_3)

    display(Markdown("**On doit ajouter 4 conditions pour être d'ordre 4**"))
    Eq_4 = ordre_4(s, A, b, c, j)
```

```
    Eqs.append(Eq_4)
    display(*Eq_4)

    return Eqs


def matrice_Butcher(s, A, b, c):
    But = sp.Matrix(A)
    But = But.col_insert(0, sp.Matrix(c))
    last = [sp.Symbol(" ")]
    last.extend(b)
    But = But.row_insert(s,sp.Matrix(last).T)
    display(Math(sp.latex(sp.Matrix(But))))
    return But


def ordre_1(s, A, b, c, j):
    texte = r"\sum_{j=1}^s b_j =" + f"{sum(b).simplify()}"
    texte += r"\text{ doit être égale à }1"
    # display(Math(texte))
    Eq_1 = [sp.Eq( sum(b).simplify(), 1 )]
    for i in range(s):
        somma = sp.summation(A[i,j],(j,0,s-1)).simplify()
        texte = r'\sum_{j=1}^s a_{'+str(i)+r'j}=' + sp.latex( somma )
        texte += r"\text{ doit être égale à }"+sp.latex(c[i])
        # display(Math( texte ))
        Eq_1.append(sp.Eq( somma, c[i] ))
    return Eq_1


def ordre_2(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j=' +sp.latex(sum([b[i]*c[i] for i in range(s)]).simplify())
    texte += r"\text{ doit être égale à }\frac{1}{2}"
    # display(Math(texte))
    Eq_2 = sp.Eq( sum([b[i]*c[i] for i in range(s)]).simplify(), sp.Rational(1,2) )
    return Eq_2


def ordre_3(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j^2='
    texte += sp.latex( sum([b[i]*c[i]**2 for i in range(s)]).simplify() )
    texte += r"\text{ doit être égale à }\frac{1}{3}"
    # display(Math(texte))
    Eq_3_1 = sp.Eq( sum([b[i]*c[i]**2 for i in range(s)]).simplify(), sp.Rational(1,3) )
    texte = r'\sum_{i,j=1}^s b_i a_{ij} c_j='
    somma = sum([b[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{6}"
    # display(Math(texte))
    Eq_3_2 = sp.Eq( sum([b[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,6) )
    return [Eq_3_1, Eq_3_2]


def ordre_4(s, A, b, c, j):
    texte = r'\sum_{j=1}^s b_j c_j^3='
    texte += sp.latex( sum([b[i]*c[i]**3 for i in range(s)]).simplify() )
    texte += r"\text{ doit être égale à }\frac{1}{4}"
    # display(Math(texte))
    Eq_4_1 = sp.Eq( sum([b[i]*c[i]**3 for i in range(s)]).simplify(), sp.Rational(1,4) )
    texte = r'\sum_{i,j=1}^s b_i c_i a_{ij} c_j='
    somma = sum([b[i]*c[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
```

```
    texte += r"\text{ doit être égale à }\frac{1}{8}"
    # display(Math(texte))
    Eq_4_2 = sp.Eq( sum([b[i]*c[i]*A[i,j]*c[j] for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,8) )
    texte = r'\sum_{i,j=1}^s b_i a_{ij} c_j^2='
    somma = sum([b[i]*A[i,j]*c[j]**2 for j in range(s) for i in range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{12}"
    # display(Math(texte))
    Eq_4_3 = sp.Eq( sum([b[i]*A[i,j]*c[j]**2 for j in range(s) for i in range(s)]).simplify(),
↪   sp.Rational(1,12) )
    texte = r'\sum_{i,j,k=1}^s b_i a_{ij} a_{jk} c_k='
    somma = sum([b[i]*A[i,j]*A[j,k]*c[k] for k in range(s) for j in range(s) for i in
↪   range(s)]).simplify()
    texte = texte + sp.latex(somma)
    texte += r"\text{ doit être égale à }\frac{1}{24}"
    # display(Math(texte))
    Eq_4_4 = sp.Eq( sum([b[i]*A[i,j]*A[j,k]*c[k] for k in range(s) for j in range(s) for i in
↪   range(s)]).simplify(), sp.Rational(1,24) )
    return [Eq_4_1, Eq_4_2, Eq_4_3, Eq_4_4]


# ============================================================================
# Calcul des conditions de consistance pour un schéma RK
# On doit indiquer le nombre d'étages s et le type de schéma
# parmi implicite, semi-implicite ou explicite
# ============================================================================
s = 2
type = "semi-implicite" # implicite, semi-implicite, explicite

c = [sp.Rational(1,3),1]
b = [sp.Rational(3,4),sp.Rational(1,4)]
A = [[sp.Rational(1,3),0],[1,0]]

display(Markdown(f"Conditions pour un schéma {type} avec s = {s} étages "))
Eqs = ordre_RK(s=s, type=type)
display(Markdown(f"Pour notre exemple on trouve "))
Eqs = ordre_RK(s=s, type=type, A=A, b=b, c=c)
```

Conditions pour un schéma semi-implicite avec s = 2 étages

Matrice de Butcher

$$\begin{bmatrix} c_0 & a_{00} & 0 \\ c_1 & a_{10} & a_{11} \\ & b_0 & b_1 \end{bmatrix}$$

On a 3 conditions pour avoir consistance (= pour être d'ordre 1)

$b_0 + b_1 = 1$

$a_{00} = c_0$

$a_{10} + a_{11} = c_1$

On doit ajouter 1 condition pour être d'ordre 2

$$b_0 c_0 + b_1 c_1 = \frac{1}{2}$$

On doit ajouter 2 conditions pour être d'ordre 3

$$b_0 c_0^2 + b_1 c_1^2 = \frac{1}{3}$$

$$a_{00} b_0 c_0 + a_{10} b_1 c_0 + a_{11} b_1 c_1 = \frac{1}{6}$$

On doit ajouter 4 conditions pour être d'ordre 4

$b_0c_0^3 + b_1c_1^3 = \frac{1}{4}$

$a_{00}b_0c_0^2 + a_{10}b_1c_0c_1 + a_{11}b_1c_1^2 = \frac{1}{8}$

$a_{00}b_0c_0^2 + a_{10}b_1c_0^2 + a_{11}b_1c_1^2 = \frac{1}{12}$

$a_{00}^2b_0c_0 + a_{00}a_{10}b_1c_0 + a_{10}a_{11}b_1c_0 + a_{11}^2b_1c_1 = \frac{1}{24}$

Pour notre exemple on trouve

Matrice de Butcher

$$\begin{bmatrix} \frac{1}{3} & \frac{1}{3} & 0 \\ 1 & 1 & 0 \\ & \frac{3}{4} & \frac{1}{4} \end{bmatrix}$$

On a 3 conditions pour avoir consistance (= pour être d'ordre 1)

True

True

True

On doit ajouter 1 condition pour être d'ordre 2

True

On doit ajouter 2 conditions pour être d'ordre 3

True

True

On doit ajouter 4 conditions pour être d'ordre 4

False

False

False

False

D'après ces résultats, le schéma est d'ordre 3.

Q7 Étudier théoriquement la A-stabilité du schéma.

Soit $\beta > 0$ un nombre réel positif et considérons le problème de Cauchy

$$\begin{cases} y'(t) = -\beta y(t), & \text{pour } t > 0, \\ y(0) = 1. \end{cases}$$

Sa solution est $y(t) = e^{-\beta t}$ donc

$$\lim_{t \to +\infty} y(t) = 0.$$

Le schéma appliqué à ce problème de Cauchy s'écrit

$$\begin{cases} u_0 = y_0 \\ K_0 = -\beta\left(u_n + \frac{h}{3}K_0\right) \\ K_1 = -\beta\left(u_n + hK_0\right) \\ u_{n+1} = u_n + \frac{h}{4}\left(3K_0 + K_1\right) \quad n = 0, 1, \ldots N-1 \end{cases}$$

L'équation $K_0 = -\beta\left(u_n + \frac{h}{3}K_0\right)$ donne $K_0 = \frac{-3\beta}{\beta h+3}u_n$ ainsi

$$u_{n+1} = u_n + \frac{h}{4}\left(3K_0 + K_1\right) = u_n + \frac{h}{4}\left(3K_0 - \beta u_n - \beta h K_0\right) = \left(1 - \frac{\beta h}{4}\right)u_n + \frac{h(3-\beta h)}{4}K_0$$

et enfin

$$u_{n+1} = \left(1 - \frac{\beta h}{4}\right)u_n + \frac{h(3-\beta h)}{4}K_0 = \left(1 - \frac{\beta h}{4}\right)u_n + \frac{h(3-\beta h)}{4}\frac{-3\beta}{\beta h+3}u_n = \left(1 - \frac{\beta h}{4} - \frac{3\beta h(3-\beta h)}{4(3+\beta h)}\right)u_n = \frac{(\beta h)^2 - 4\beta h}{2(\beta h+3}$$

Vérifions ce calcul :

```
# pour ne pas effacer l'affectation de "h", ici on vas l'appeler "dt" mais on affiche "h"
dt = sp.Symbol('h')
u_np1 = sp.Symbol('u_{n+1}')
beta = sp.Symbol(r'\beta')
sp.var('u_n,x')

K0 = sp.solve(-x-beta*(u_n+dt/3*x),x)[0]
display(Math('K_0='+sp.latex(K0)))

K1 = -beta*(u_n+dt*K0).factor()
display(Math('K_1='+sp.latex(K1)))

u_np1 = (u_n+dt/4*(3*K0+K1)).factor()
display(Math('u_{n+1}='+sp.latex(u_np1)))
```

$$K_0 = -\frac{3\beta u_n}{\beta h+3}$$

$$K_1 = \frac{\beta u_n\left(2\beta h-3\right)}{\beta h+3}$$

$$u_{n+1} = \frac{u_n\left(\beta^2h^2-4\beta h+6\right)}{2\left(\beta h+3\right)}$$

On note $x = \beta h$ et on étudie la fonction $q\colon \mathbb{R}^+ \to \mathbb{R}$ définie par $q(x) = \frac{x^2-4x+6}{2(x+3)} = \frac{1}{2}\left(x - 7 + \frac{27}{x+3}\right)$. Le schéma est A-stable ssi $|q(x)| < 1$.

L'étude montre que la fonction q est continue et décroissante sur $[0, x_m]$ puis croissante sur $[x_m, +\infty[$ avec $q(0) = 1$ et $0 < q(x_m) < 1$. Comme $\lim_{x\to+\infty} q(x) = +\infty$, il existe un et un seul $\widehat{x} > x_m$ tel que $q(\widehat{x}) = 1$. On trouve $\widehat{x} = 6$.

D'après les calculs ci-dessous on conclut que le schéma est A-stable ssi $h < \frac{6}{\beta}$.

```
sp.var('x',nonnegative=True)

q = (u_np1/u_n).subs(beta*dt,x)
display(Math('q(x)='+sp.latex(q)))

display(Math('q(0)='+sp.latex(q.subs(x,0))))
lim=sp.Limit(q,x,sp.oo)
display(Math(sp.latex(lim)+'='+sp.latex(lim.doit())))

print('On cherche les points stationnaires dans R^+')
dq=(q.diff(x)).factor()
display(Math("q'(x)="+sp.latex(dq)))
print("Le signe de q' est le signe du numérateur qui est une parabole. On cherche où elle s'annulle pour
 ↪  x>0")
sol=sp.solve(dq)
display(Math("q'(x)=0 \\iff x\\in"+sp.latex(sol)))
minimum=sol[0]
```

```
display(Math("x="+sp.latex(minimum)+"\\text{ est un minimum et}"))
qmin=q.subs(x,sol[0])
display(Math("q("+sp.latex(minimum)+")="+sp.latex(qmin)+"\\approx"+sp.latex(sp.N(qmin))))
print("Conclusion: q(x)>0 pour tout x>0. Vérifions ce calcul:")
print("q(x)>0")
display(sp.solve(q>0))
print("On resout maintenant q(x)=1:")
display(sp.solve(q-1))
print('On trouve q(0)=q(6)=1')
print("Conclusion: q(x)<1 pour x<6. Vérifions ce calcul:")
print("q(x)<1")
display(sp.solve(q<1))


sp.plot(q,1,-1,xlim=[-1,15],ylim=[-1.5,1.5]);
```

$$q(x) = \frac{x^2 - 4x + 6}{2x + 6}$$

$$q(0) = 1$$

$$\lim_{x \to \infty} \left(\frac{x^2 - 4x + 6}{2x + 6} \right) = \infty$$

```
On cherche les points stationnaires dans R^+
```

$$q'(x) = \frac{x^2 + 6x - 18}{2\left(x + 3\right)^2}$$

```
Le signe de q' est le signe du numérateur qui est une parabole. On cherche où elle s'annulle pour x>0
```

$$q'(x) = 0 \iff x \in \left[-3 + 3\sqrt{3}\right]$$

$x = -3 + 3\sqrt{3}$ est un minimum et

$$q(-3 + 3\sqrt{3}) = \frac{\sqrt{3}\left(-12\sqrt{3} + \left(-3 + 3\sqrt{3}\right)^2 + 18\right)}{18} \approx 0.196152422706632$$

```
Conclusion: q(x)>0 pour tout x>0. Vérifions ce calcul:
q(x)>0
```

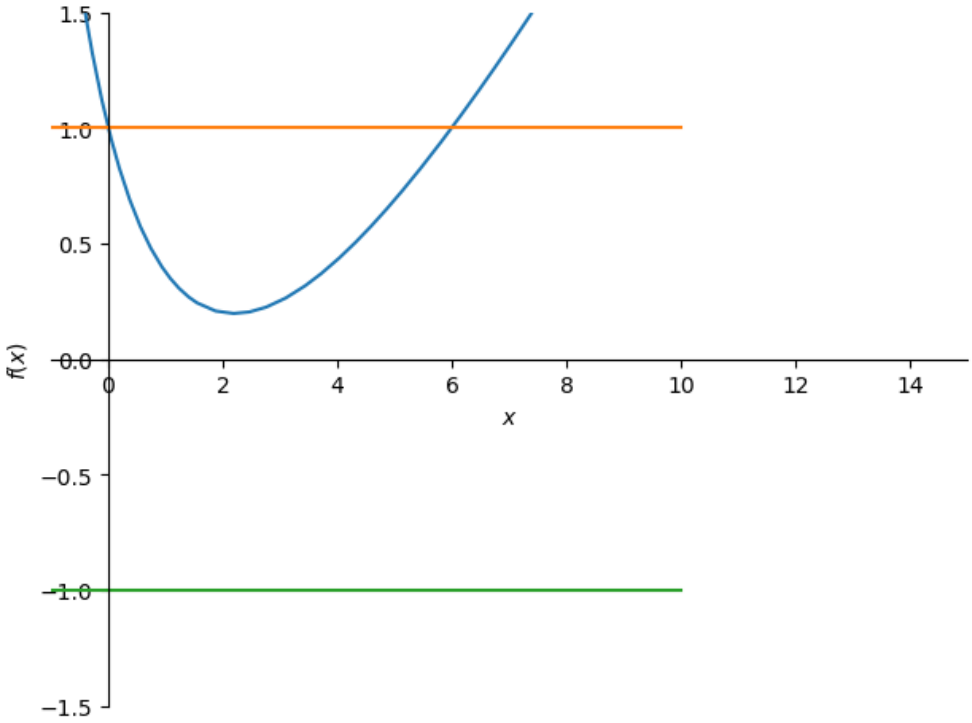
True

```
On resout maintenant q(x)=1:
```

```
[0, 6]
```

```
On trouve q(0)=q(6)=1
Conclusion: q(x)<1 pour x<6. Vérifions ce calcul:
q(x)<1
```

$0 < x \wedge x < 6$



Q8 Implémenter le schéma et approcher la solution du problème de Cauchy suivant avec $N+1$ points et $N = 8$.

$$\begin{cases} y'(t) = -6y(t), & t \in [0;1] \\ y(0) = 1 \end{cases}$$

Dans chaque point t_i, il faut approcher $(K_0)_i$ en resolvant une équation. Si on utilise la fonction `fsolve` du module `scipy.optimize`, il faut initialiser `fsolve` avec une approximation de $(K_0)_i$. On choisira donc $\varphi(t_i, u_i)$:

```
# version implicite
def RK(phi,tt):
    h = tt[1]-tt[0]
    uu = np.zeros(len(tt))
    uu[0] = y0
    for n in range(len(tt)-1):
        k0 = fsolve( lambda x : -x+phi( tt[n]+h/3, uu[n]+h/3*x ) , phi(tt[n],uu[n]) )[0]
        k1 = phi( tt[n+1], uu[n]+h*k0 )
        uu[n+1] = uu[n] + h/4*(3*k0+k1 )
    return uu

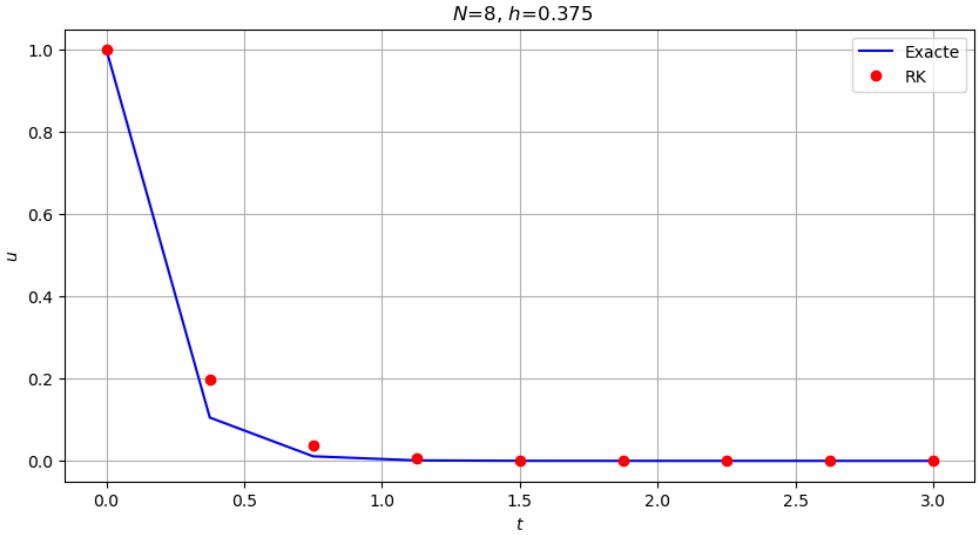
# version explicite car phi = - beta h
# def RK(beta,tt):
#     h = tt[1]-tt[0]
#     x = beta*h
#     q = lambda x : (x**2-4*x+6)/(2*x+6)
#     uu = [y0*q(x)**n for n in range(len(tt))]
#     return uu




t0, y0, tfinal = 0, 1, 3
beta = 6
sol_exacte = lambda t : np.exp(-beta*t)
phi = lambda t,y : -beta*y
```

```
print('A-stable ssi h <',6/beta)
plt.figure(figsize=(10,5))
N = 8
tt = np.linspace(t0,tfinal,N+1)
h = tt[1]-tt[0]
yy = sol_exacte(tt)
uu = RK(phi,tt)  # version avec fsolve
# uu = RK(beta,tt) # version explicite
plt.plot(tt,yy,'b-',label=("Exacte"))
plt.plot(tt,uu,'ro',label=("RK"))
plt.title(r' $N$={}, $h$={}'.format(N,h))
plt.xlabel('$t$')
plt.ylabel('$u$')
plt.legend()
plt.grid(True)
```

```
A-stable ssi h < 1.0
```



36.3. EXERCICE 3 : étude d'un schéma Predictor-Corrector

Considérons les deux méthodes multi-pas linéaires suivantes :

$$u_{n+1} = u_{n-3} + \frac{4h}{3}\left(2\varphi_n - \varphi_{n-1} + 2\varphi_{n-2}\right) \quad \text{[P]}$$

$$u_{n+1} = u_{n-1} + \frac{h}{3}\left(\varphi_{n+1} + 4\varphi_n + \varphi_{n-1}\right) \quad \text{[C]}$$

ayant noté $\varphi_k \overset{\text{déf}}{=} \varphi(t_k, u_k)$.

1. **Étudier l'ordre de convergence théorique** de chacune des deux méthodes.
2. **Écrire le schéma predictor-corrector** associé à ces deux méthodes et en déterminer l'ordre de convergence théorique.
3. **Estimer empiriquement l'ordre de convergence** des méthodes [P] et [C], ainsi que de la méthode Predictor-Corrector associée, en les appliquant au problème de Cauchy

$$\begin{cases} y'(t) = 1 + (t - y(t))^2, & \forall t \in I = [2,3], \\ y(2) = 1. \end{cases}$$

```
def xi(i, q, aa=None, bb=None, bm1=None):
    if aa is None:
        aa = sp.symbols(f'a_0:{q}')
    if bb is None:
        bb = sp.symbols(f'b_0:{q}')
    if bm1 is None:
        bm1 = sp.symbols('b_{-1}')
    p = len(aa)
    sa = sum((-j)**i * aa[j] for j in range(p))
    sb = bm1 + sum((-j)**(i-1) * bb[j] for j in range(1, p))
    # ??? si j=0 et i=0, il calcule (-j)**i =1 et on a bien a_0 dans la somme
    # mais si j=0 et i=1 il ne calcule pas b_0 et il faut l'ajouter à la main
    if i == 1:
        sb += bb[0]
    return (sa) + (i * sb)

def affichage_xi(q, aa, bb, bm1):
    p = q-1
    omega = q + 2 if q%2==0 else q + 1 # i = 0, ..., omega
    display(Markdown(f"Pour une méthode à q={q} pas, on affiche $\\xi(i)$ pour i = 0, ..., {q+2}"))
    for i in range(q+3):
        display(Markdown(f"$\\xi({i}) = {sp.latex(xi(i, q, None, None, None))} = {sp.latex(xi(i, q, aa,
↪ bb, bm1))}$"))
```

Q9 Étudier la méthode P.

[P] est une méthode à $q = 4$ pas explicite :

- $p = 3$
- $b_{-1} = 0$
- $a_0 = a_1 = a_2 = 0$ et $a_3 = 1$
- $b_0 = \frac{8}{3}$, $b_1 = \frac{-4}{3}$, $b_2 = \frac{8}{3}$ et $b_3 = 0$

La méthode est explicite, donc son ordre de consistance maximal est q. On vérifie qu'elle est effectivement consistante et son ordre de consistance est bien $\omega = 4$. En effet, $\xi(0) = \xi(1) = \xi(2) = \xi(3) = \xi(4) = 1$.

Vérifions ces calculs ci-dessous :

```
# Paramètres
q = 3

a0 = 0; a1 = 0; a2 = 0; a3 = 1
b0 = sp.Rational(8,3); b1 = sp.Rational(-4,3); b2 = sp.Rational(8,3); b3 = 0
bm1 = 0

aa = [a0,a1,a2,a3]
bb = [b0,b1,b2,b3]
affichage_xi(q, aa, bb, bm1)
```

Pour une méthode à q=3 pas, on affiche $\xi(i)$ pour i = 0, ..., 5

$\xi(0) = a_0 + a_1 + a_2 = 1$

$\xi(1) = -a_1 - 2a_2 + b_0 + b_1 + b_2 + b_{-1} = 1$

$\xi(2) = a_1 + 4a_2 - 2b_1 - 4b_2 + 2b_{-1} = 1$

$\xi(3) = -a_1 - 8a_2 + 3b_1 + 12b_2 + 3b_{-1} = 1$

$\xi(4) = a_1 + 16a_2 - 4b_1 - 32b_2 + 4b_{-1} = 1$

$\xi(5) = -a_1 - 32a_2 + 5b_1 + 80b_2 + 5b_{-1} = -\frac{109}{3}$

Le premier polynôme caractéristique est

$$\varrho(r) = r^{p+1} - \sum_{j=0}^{p} a_j r^{p-j} = r^4 - a_0 r^3 - a_1 r^2 - a_2 r - a_3 = r^4 - 1$$

dont les racines sont

$$r_0 = 1,\ r_1 = -1,\ r_2 = i,\ r_3 = -i$$

La méthode est zéro-stable ssi

$$\begin{cases} |r_j| \leq 1 \quad \text{pour tout } j = 0, \dots, p \\ r_j \text{ a multiplicité} > 1 \quad \Longrightarrow \quad |r_j| < 1, \end{cases}$$

ce qui est bien vérifié.

Conclusion : la méthode est convergente car consistante et zéro-stable. Elle est d'ordre 4.

Q10 Étudier la méthode C.

[C] est une méthode à $q = 2$ pas implicite :

- $p = 1$
- $b_{-1} = \frac{1}{3}$
- $a_0 = 0$ et $a_1 = 1$
- $b_0 = \frac{4}{3}$ et $b_1 = \frac{1}{3}$

La méthode est implicite avec un nombre pair de pas, donc son ordre de consistance maximal est $q + 2$. On vérifie qu'elle est effectivement consistante et son ordre de consistance est bien $\omega = 4$. En effet, $\xi(0) = \xi(1) = \xi(2) = \xi(3) = \xi(4) = 1$.

Vérifions ces calculs ci-dessous :

```
# Paramètres
q = 2

a0 = 0; a1 = 1
b0 = sp.Rational(4,3); b1 = sp.Rational(1,3)
bm1 = sp.Rational(1,3)

aa = [a0,a1]
bb = [b0,b1]
affichage_xi(q, aa, bb, bm1)
```

Pour une méthode à q=2 pas, on affiche $\xi(i)$ pour i = 0, ..., 4

$\xi(0) = a_0 + a_1 = 1$

$\xi(1) = -a_1 + b_0 + b_1 + b_{-1} = 1$

$\xi(2) = a_1 - 2b_1 + 2b_{-1} = 1$

$\xi(3) = -a_1 + 3b_1 + 3b_{-1} = 1$

$\xi(4) = a_1 - 4b_1 + 4b_{-1} = 1$

Le premier polynôme caractéristique est

$$\varrho(r) = r^{p+1} - \sum_{j=0}^{p} a_j r^{p-j} = r^2 - a_0 r - a_1 = r^2 - 1$$

dont les racines sont

$$r_0 = 1,\ r_1 = -1$$

La méthode est zéro-stable ssi

$$\begin{cases} |r_j| \leq 1 \quad \text{pour tout } j = 0, \dots, p \\ r_j \text{ a multiplicité} > 1 \quad \Longrightarrow \quad |r_j| < 1, \end{cases}$$

ce qui est bien vérifié.

Conclusion : la méthode est convergente car consistante et zéro-stable. Elle est d'ordre 4.

Q11 Estimation empirique de l'ordre de convergence des méthodes P et C ainsi que de la méthode Predictor-Corrector associée aux deux méthodes P et C sur le problème de Cauchy

$$\begin{cases} y'(t) = 1 + (t - y(t))^2, \quad \forall t \in I = [2, 3], \\ y(2) = 1. \end{cases}$$

Le schéma predictor-corrector est défini par

$$\begin{aligned} \tilde{u}_{n+1} &= u_{n-3} + \frac{4h}{3}\left(2\varphi_n - \varphi_{n-1} + 2\varphi_{n-2}\right) \\ u_{n+1} &= u_{n-1} + \frac{h}{3}\left(\varphi(t_{n+1}, \tilde{u}_{n+1}) + 4\varphi_n + \varphi_{n-1}\right) \end{aligned}$$

Ce schéma est d'ordre 4 car le prédictor est d'ordre 4, le corrector d'ordre 4 et l'on a

$$\omega_{[PC]} = \min\left\{\omega_{[C]}, \omega_{[P]} + 1\right\}$$

On définit la solution exacte pour estimer les erreurs et on calcule la solution approchée pour différentes valeurs de N pour les trois schémas :

```
t0     = 2
tfinal = 3
y0     = 1

phi = lambda t,y : 1+(t-y)**2

# Solution exacte
# ===============

t = sp.Symbol('t')
y = sp.Function('y')
edo = sp.Eq( sp.diff(y(t),t) , phi(t,y(t)) )
display(edo)
solpar = sp.dsolve( edo, y(t), ics={y(t0):y0})
display(solpar)

# Conversion en fonction numpy
# ============================

sol_exacte = sp.lambdify(t,solpar.rhs,'numpy')
# sol_exacte = lambda t : t+1/(1-t)



# Schémas
# =======

def predictor(phi, tt):
    h  = tt[1] - tt[0]
    uu = np.zeros_like(tt)
```

```
    uu[:4] = sol_exacte(tt[:4])
    for n in range(3,len(tt) - 1):
        k0 = phi(tt[n],uu[n])
        k1 = phi(tt[n-1],uu[n-1])
        k2 = phi(tt[n-2],uu[n-2])
        uu[n+1] = uu[n-3] + 4*h/3*( 2*k0 - k1 + 2*k2 )
    return uu


from scipy.optimize import fsolve
def corrector(phi, tt):
    h  = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    uu[:2] = sol_exacte(tt[:2])
    for n in range(1,len(tt) - 1):
        k0 = phi(tt[n],uu[n])
        k1 = phi(tt[n-1],uu[n-1])
        temp = fsolve ( lambda x : -x + uu[n-1] + h/3*( phi(tt[n+1],x) + 4*k0 + k1 ), uu[n])
        uu[n+1] = temp[0]
    return uu

def PC(phi, tt):
    h  = tt[1] - tt[0]
    uu = np.zeros_like(tt)
    uu[:4] = sol_exacte(tt[:4])
    for n in range(3,len(tt) - 1):
        k0 = phi(tt[n],uu[n])
        k1 = phi(tt[n-1],uu[n-1])
        k2 = phi(tt[n-2],uu[n-2])
        u_tilde = uu[n-3]+ 4*h/3*( 2*k0 - k1 + 2*k2 )
        uu[n+1] = uu[n-1]+h/3* ( phi(tt[n+1],u_tilde) + 4*k0 + k1 )
    return uu


# Calcul de l'ordre de convergence
# ================================

H = []
err_p = []
err_c = []
err_pc = []

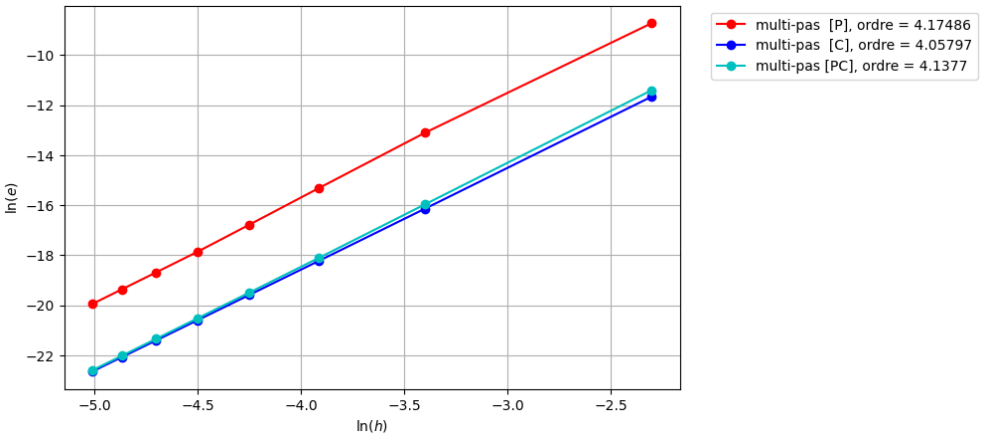
for N in range(10,160,20):
    tt = np.linspace(t0, tfinal, N + 1)
    h = tt[1] - tt[0]
    yy = sol_exacte(tt)
    uu_p = predictor(phi, tt)
    uu_c = corrector(phi, tt)
    uu_pc = PC(phi, tt)
    H.append(h)
    err_p.append(max([abs(uu_p[i] - yy[i]) for i in range(len(yy))]))
    err_c.append(max([abs(uu_c[i] - yy[i]) for i in range(len(yy))]))
    err_pc.append(max([abs(uu_pc[i] - yy[i]) for i in range(len(yy))]))

plt.figure(figsize=(8,5))
plt.plot(np.log(H), np.log(err_p), 'r-o', label=f'multi-pas  [P], ordre =
 ↪  {np.polyfit(np.log(H),np.log(err_p ), 1)[0]:g}')
plt.plot(np.log(H), np.log(err_c), 'b-o', label=f'multi-pas  [C], ordre =
 ↪  {np.polyfit(np.log(H),np.log(err_c ), 1)[0]:g}')
plt.plot(np.log(H), np.log(err_pc),'c-o', label=f'multi-pas [PC], ordre =
 ↪  {np.polyfit(np.log(H),np.log(err_pc), 1)[0]:g}')
plt.xlabel(r'$\ln(h)$')
plt.ylabel(r'$\ln(e)$')
```

```
plt.legend(bbox_to_anchor=(1.04, 1), loc='upper left')
plt.grid(True);
```

$$\frac{d}{dt}y(t) = (t - y(t))^2 + 1$$

$$y(t) = \frac{t^2 - t - 1}{t - 1}$$



On a bien

- ordre du corrector $\omega_{[C]} = 4$
- ordre du predictor $\omega_{[P]} = 4$

et donc

- ordre du predictor-corrector $\omega_{[PC]} = \min\{\omega_{[C]}, \omega_{[P]} + 1\} = 4$

Chapitre 37.

2019 Contrôle Continu

```
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp

from IPython.display import display, Markdown, Math
```

37.1. EXERCICE : schéma de Runge-Kutta

Soit le schéma de Runge-Kutta dont la *matrice de Butcher* est

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \alpha & \alpha & 0 \\ \hline & \frac{2\alpha-1}{2\alpha} & \frac{1}{2\alpha} \end{array}$$

Remarque : chaque sujet correspond à une valeur de α choisie parmi les suivantes :

$$\left\{\frac{1}{6}, \frac{1}{4}, \frac{1}{3}, \frac{5}{12}, \frac{7}{12}, \frac{2}{3}, \frac{3}{4}, \frac{5}{6}, \frac{11}{12}\right\}$$

1. Écrire le schéma sous la forme d'une suite définie par récurrence.
2. Étudier théoriquement la A-stabilité.
3. Implémenter le schéma et le tester avec le problème de Cauchy $y'(t) = \beta y(t)$, $y(0) = 1$ sur l'intervalle $[0; 1]$. On choisira le nombre minimal de points pour avoir A-stabilité. *Remarque :* chaque sujet correspond à une valeur de β choisie parmi les suivantes : $\{50, 60, 70, 80, 90, 100\}$
4. Étudier théoriquement l'ordre du schéma.
5. Calculer la solution exacte du problème de Cauchy

$$\begin{cases} y'(t) = -2ty^2(t), & t \in [0; 3] \\ y(0) = 2 \end{cases}$$

6. Vérifier empiriquement l'ordre de convergence en affichant la courbe de convergence et en estimant la pente sur ce problème de Cauchy.

```
ALPHA = [sp.Rational(2,12), sp.Rational(3,12), sp.Rational(4,12), sp.Rational(5,12),
         sp.Rational(7,12), sp.Rational(8,12), sp.Rational(9,12), sp.Rational(10,12), sp.Rational(11,12)]
for alpha in ALPHA:
    display(Markdown(f"$$c_2 = a_{{21}} = {sp.latex(alpha)} \\qquad
 ↪  b_1={sp.latex(1-sp.Rational(1,2*alpha))} \\qquad b_2={sp.latex(sp.Rational(1,2*alpha))}$$"))
```

$$c_2 = a_{21} = \frac{1}{6} \qquad b_1 = -2 \qquad b_2 = 3$$

$$c_2 = a_{21} = \frac{1}{4} \qquad b_1 = -1 \qquad b_2 = 2$$

$$c_2 = a_{21} = \frac{1}{3} \qquad b_1 = -\frac{1}{2} \qquad b_2 = \frac{3}{2}$$

$$c_2 = a_{21} = \frac{5}{12} \qquad b_1 = -\frac{1}{5} \qquad b_2 = \frac{6}{5}$$

$$c_2 = a_{21} = \frac{7}{12} \qquad b_1 = \frac{1}{7} \qquad b_2 = \frac{6}{7}$$

$$c_2 = a_{21} = \frac{2}{3} \qquad b_1 = \frac{1}{4} \qquad b_2 = \frac{3}{4}$$

$$c_2 = a_{21} = \frac{3}{4} \qquad b_1 = \frac{1}{3} \qquad b_2 = \frac{2}{3}$$

$$c_2 = a_{21} = \frac{5}{6} \qquad b_1 = \frac{2}{5} \qquad b_2 = \frac{3}{5}$$

$$c_2 = a_{21} = \frac{11}{12} \qquad b_1 = \frac{5}{11} \qquad b_2 = \frac{6}{11}$$

37.1.1. Correction : écriture schéma

Considérons le problème de Cauchy

trouver une fonction $y\colon I \subset \mathbb{R} \to \mathbb{R}$ définie sur un intervalle $I = [t_0, T]$ telle que

$$\begin{cases} y'(t) = \varphi(t, y(t)), \quad \forall t \in I = [t_0, T], \\ y(t_0) = y_0, \end{cases}$$

avec y_0 une valeur donnée et supposons que l'on ait montré l'existence et l'unicité d'une solution y pour $t \in I$.

Pour $h > 0$ soit $t_n \overset{\text{déf.}}{=} t_0 + nh$ avec $n = 0, 1, 2, \dots, N$ une suite de $N+1$ nœuds de I induisant une discrétisation de I en N sous-intervalles $I_n = [t_n; t_{n+1}]$ chacun de longueur $h > 0$ (appelé le *pas de discrétisation*).

Pour chaque nœud t_n, on cherche la valeur inconnue u_n qui approche la valeur exacte $y_n \equiv y(t_n)$.

- L'ensemble de $N+1$ valeurs $\{t_0, t_1 = t_0 + h, ..., t_N = T\}$ représente les points de la discrétisation.
- L'ensemble de $N+1$ valeurs $\{y_0, y_1, ..., y_N\}$ représente la solution exacte.
- L'ensemble de $N+1$ valeurs $\{u_0 = y_0, u_1, ..., u_N\}$ représente la solution numérique. Cette solution approchée sera obtenue en construisant une suite récurrente.

Le schéma qu'on va construire permet de calculer explicitement u_{n+1} à partir de u_n et il est donc possible de calculer successivement u_1, u_2,..., en partant de u_0 par la formule de récurrence

$$\begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n, u_n\right) \\ K_2 = \varphi\left(t_n + \alpha h, u_n + \alpha h K_1\right) \\ u_{n+1} = u_n + \frac{h}{2\alpha}\left((2\alpha - 1)K_1 + K_2\right) \quad n = 0, 1, \dots N-1 \end{cases}$$

37.1.2. Correction : étude de la A-stabilité

Définition de A-stabilité

Soit $\beta > 0$ un nombre réel positif et considérons le problème de Cauchy

$$\begin{cases} y'(t) = -\beta y(t), \quad \text{pour } t > 0, \\ y(0) = y_0 \end{cases}$$

où $y_0 \neq 0$ est une valeur donnée. Sa solution est $y(t) = y_0 e^{-\beta t}$ donc

$$\lim_{t \to +\infty} y(t) = 0.$$

Si, sous d'éventuelles conditions sur h, on a

$$\lim_{n \to +\infty} u_n = 0,$$

alors on dit que **le schéma est A-stable.**

Le schéma donné appliqué à cette équation devient

```
h = sp.symbols('h')
Alpha = sp.symbols(r'\Alpha')
beta = sp.symbols(r'\beta')
u_n = sp.symbols('u_n')
u_np1 = sp.symbols('u_{n+1}')

Phi = lambda y : -beta*y

K_1 = Phi(u_n)
display(sp.Eq(sp.Symbol('K_1'), K_1))

K_2 = Phi(u_n+Alpha*h*K_1)
display(sp.Eq(sp.Symbol('K_2'), K_2.expand().factor(u_n)))

display(sp.Eq(u_np1,(u_n+h/(2*Alpha)*((2*Alpha-1)*K_1+K_2  )).factor()))
```

$K_1 = -\beta u_n$

$K_2 = \beta u_n \left(\mathrm{A}\beta h - 1\right)$

$u_{n+1} = \dfrac{u_n \left(\beta^2 h^2 - 2\beta h + 2\right)}{2}$

c'est-à-dire

$$\begin{cases} u_0 = y_0 \\ u_{n+1} = \dfrac{2 - 2(\beta h) + (\beta h)^2}{2} u_n \quad n = 0, 1, \ldots N-1 \end{cases}$$

Par induction on obtient

$$u_n = \left(1 - (\beta h) + \frac{1}{2}(\beta h)^2\right)^n u_0.$$

Notons que u_n ne dépend pas du choix de α.

Étant une suite géométrique, $\lim_{n \to +\infty} u_n = 0$ si et seulement si

$$\left|1 - (\beta h) + \frac{1}{2}(\beta h)^2\right| < 1.$$

Notons x le produit βh et q le polynôme $q(x) = \frac{1}{2}x^2 - x + 1$.

```
# from sympy.plotting import plot

x = sp.symbols('x', positive=True)
q = 1 - 1*x + x**2/2
display(Markdown(f"$$q(x)={sp.latex(q)}$$"))

display(Markdown("Calculons les valeurs de $x$ pour lesquelles $|q(x)|<1$:"))
display(sp.solve(sp.Abs(q)<1))

display(Markdown("Calculons la dérivée de $q$ pour trouver les extremums locaux:"))
dq = sp.diff(q,x)
display(sp.Eq(sp.Symbol("q'(x)"), dq))

display(Markdown("Calculons les valeurs de $x$ pour lesquelles $q'(x)=0$:"))
x_sommet = sp.solve(dq)[0]
display(x_sommet)

display(Markdown("Calculons la valeur de $q$ au sommet:"))
y_sommet=q.subs(x,x_sommet)
display(y_sommet)

display(Markdown("Comme $y_{sommet}=q(x_{sommet})>0$, on en déduit que $q(x)>0$ pour tout $x$ et que
↪  $q(x)<1$ ssi $x<2x_{sommet}$:"))
display(Markdown(f"ainsi $q(x)>0$ pour tout $x$ et $q(x)<1$ ssi $x<{sp.latex(2*x_sommet)}$"))
sp.plot(q,1,-1,(x,0,x_sommet*2));
```

$$q(x) = \frac{x^2}{2} - x + 1$$

Calculons les valeurs de x pour lesquelles $|q(x)| < 1$:

$x < 2$

Calculons la dérivée de q pour trouver les extremums locaux :

$q'(x) = x - 1$

Calculons les valeurs de x pour lesquelles $q'(x) = 0$:

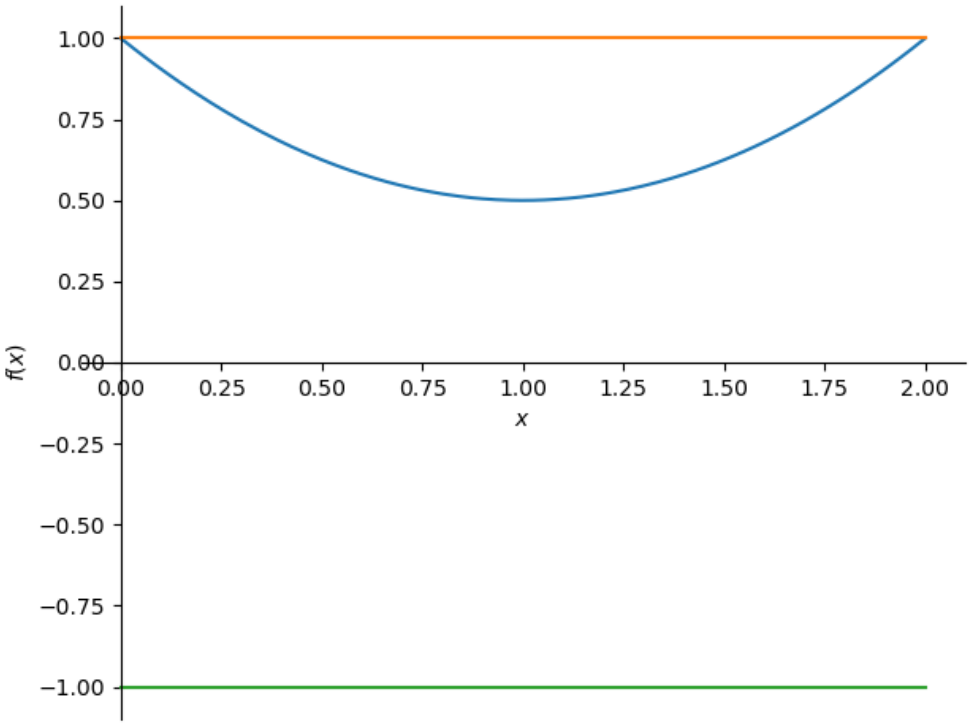
1

Calculons la valeur de q au sommet :

$\frac{1}{2}$

Comme $y_{sommet} = q(x_{sommet}) > 0$, on en déduit que $q(x) > 0$ pour tout x et que $q(x) < 1$ ssi $x < 2x_{sommet}$:

ainsi $q(x) > 0$ pour tout x et $q(x) < 1$ ssi $x < 2$



Il s'agit de la parabole convexe de sommet $\left(1, \frac{1}{2}\right)$.
Nous avons $|q(x)| < 1$ si et seulement si $x < 2$. La relation $\lim_{n \to +\infty} u_n = 0$ est donc satisfaite si et seulement si

$$h < \frac{2}{\beta}.$$

Cette condition de stabilité limite le pas h d'avance en t lorsqu'on utilise le schéma.

Nous avons $q(x) > 0$ pour tout x donc cette convergence est monotone.

37.1.3. Correction : implémentation et test

$h < \frac{2}{\beta}$ sur l'intervalle $[0; 1]$ impose $N > \frac{\beta}{2}$.

```
def RKalpha(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = [y0]
    for i in range(len(tt)-1):
        k1 = phi( tt[i], uu[i] )
        k2 = phi( tt[i]+h*alpha , uu[i]+h*alpha*k1 )
        uu.append( uu[i] + h*((2*alpha-1)*k1+k2 )/(2*alpha))
    return uu
```

```
BETA = np.arange(50,101,10)

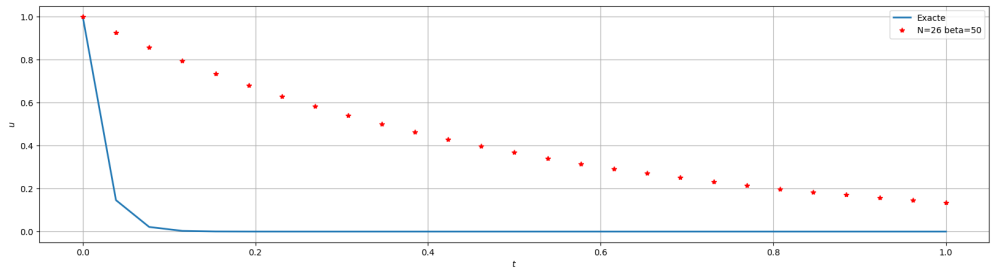
t0, y0, tfinal = 0, 1, 1

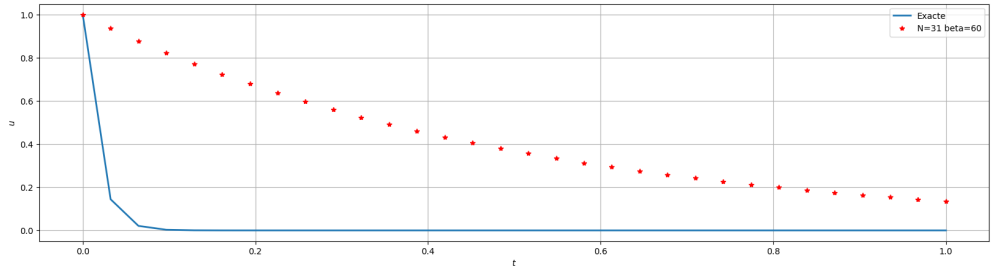
sol_exacte = lambda t : y0*np.exp(-beta*t)
phi = lambda t,y : -beta*y

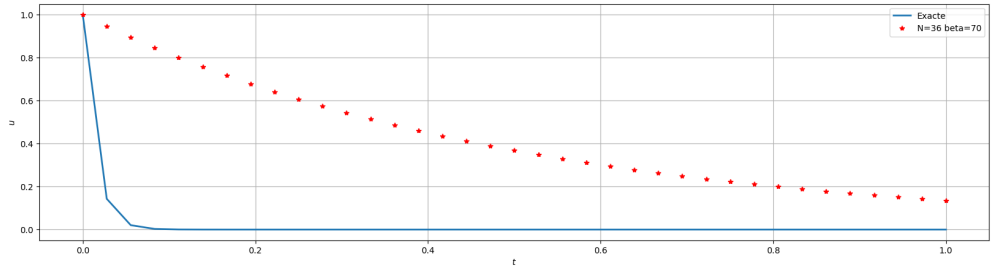
i = 0

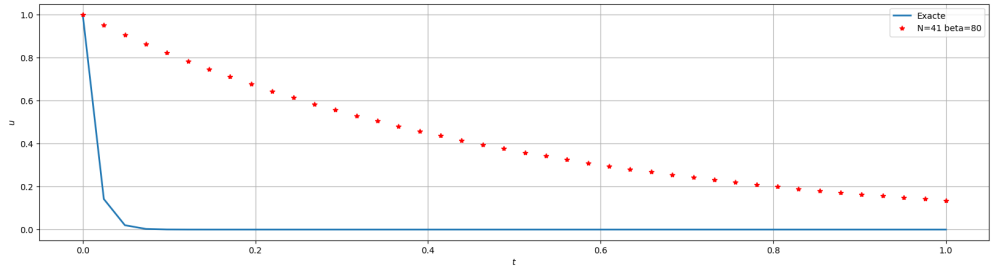

for beta in BETA:
    N=(int(beta/2)+1)
```

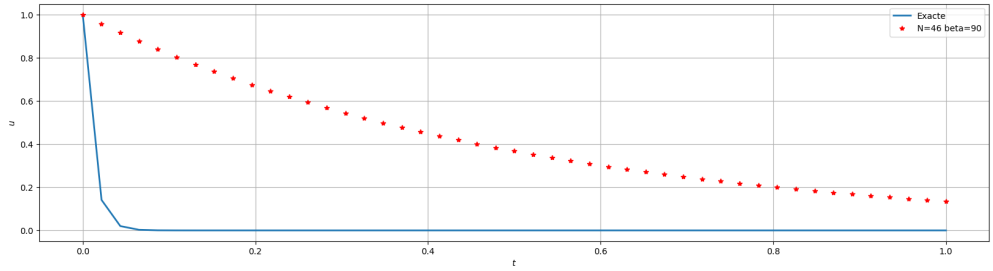
```
    tt = np.linspace(t0,tfinal,N+1)
    yy = [sol_exacte(t) for t in tt]
    alpha=ALPHA[0]
    uu = RKalpha(phi,tt,y0)
    i+=1
    plt.figure(i,figsize=(20,5))
    plt.plot(tt,yy,lw=2,label=("Exacte"))
    plt.plot(tt,uu,'r*',label=("N="+str(N)+" beta="+str(beta)))
    plt.xlabel('$t$')
    plt.ylabel('$u$')
    plt.legend()
    plt.grid(True)
```

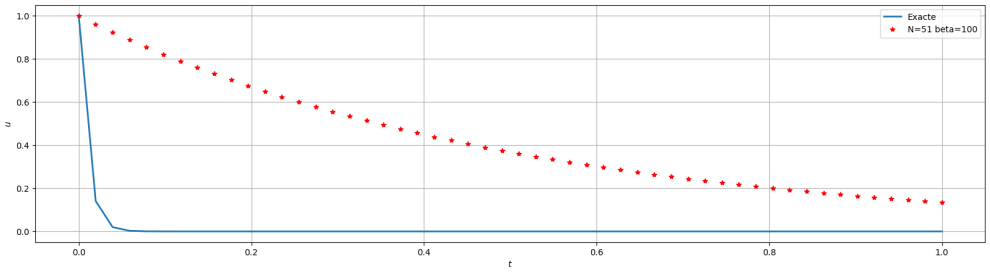












37.1.4. Correction : étude de l'ordre

- C'est un schéma explicite à deux étage : l'ordre est au plus 2.
- $c_1 = a_{11} + a_{12}$, $c_2 = a_{21} + a_{22}$ et $1 = b_1 + b_2 \implies$ il est consistante
- $c_1 b_1 + c_2 b_2 = \frac{1}{2} \implies$ il est d'ordre 2

```
for alpha in ALPHA:
    print("alpha = ",alpha)
    c = [0,alpha]
    b = [1-1/(2*alpha),1/(2*alpha)]
    A = [[0,0],[alpha,0]]
    s = len(c)
    ordre = []
    ordre.append(sum(b)==1)
    for i in range(s):
        ordre.append(sum(A[i])==c[i])
    ordre.append(sum([b[i]*c[i] for i in range(s)])==sp.Rational(1,2))
    print(set(ordre))
    print("----------------------")
```

```
alpha =  1/6
{True}
----------------------
alpha =  1/4
{True}
----------------------
alpha =  1/3
{True}
----------------------
alpha =  5/12
{True}
----------------------
alpha =  7/12
{True}
----------------------
alpha =  2/3
{True}
----------------------
alpha =  3/4
{True}
----------------------
alpha =  5/6
{True}
----------------------
alpha =  11/12
{True}
----------------------
```

37.1.5. Correction : solution exacte

On peut soit calculer la solution “à la main” soit utiliser `sympy`.

Il s'agit d'une EDO à variables séparables. La fonction $y(t) = 0$ pour tout t est solution de l'EDO mais elle ne vérifie pas la CI. Toute autre solution de l'EDO sera non nulle et se trouve formellement comme suit :

$$y'(t) = -2ty^2(t) \quad \Longrightarrow \quad \frac{y'(t)}{y^2(t)} = -2t \quad \Longrightarrow \quad \int y^{-2}\mathrm{d}y = -2\int t\mathrm{d}t \quad \Longrightarrow \quad y(t) = \frac{1}{t^2+c},\ c \in \mathbb{R}.$$

En imposant la CI on obtient $2 = 1/C$ d'où l'unique solution du problème de Cauchy : $y(t) = \frac{2}{2t^2+1}$.

```
t = sp.symbols('t')
u = sp.Function('u')

edo = sp.Eq(sp.diff(u(t),t),-2*t*(u(t))**2)
display(edo)

t0 = 0
u0 = 2
IC = sp.Eq(u(t0),u0)
display(IC)

sol = sp.dsolve(edo,u(t),ics={u(t0):u0})
print('Solution :')
display(sol)

sp.plot(sol.rhs,(t,0,3))

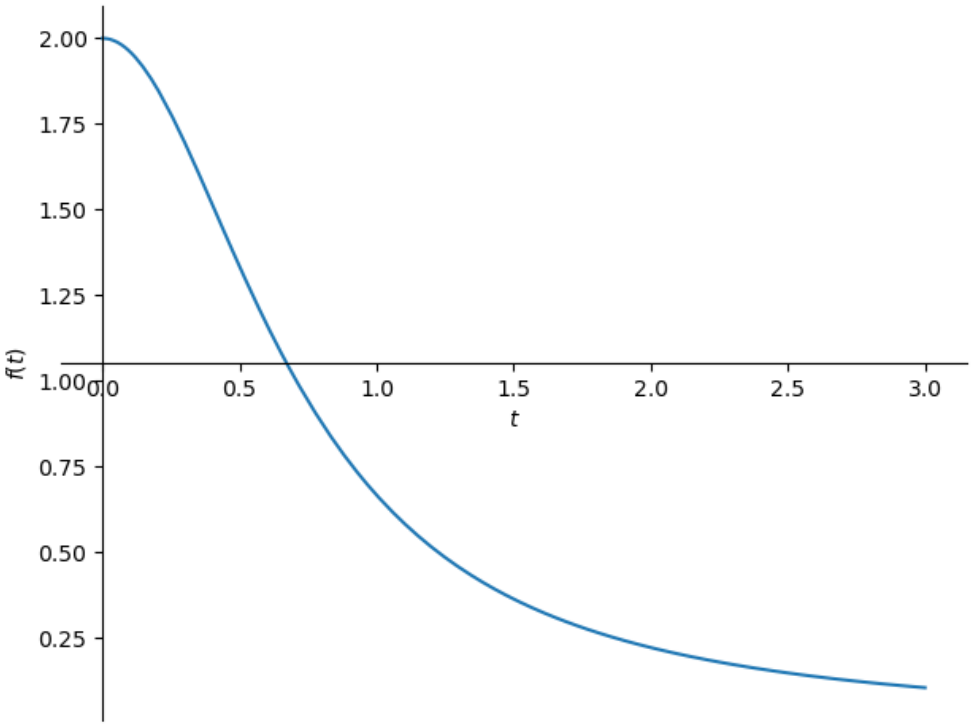
sol_exacte = sp.lambdify(t,sol.rhs, 'numpy')
```

$$\frac{d}{dt}u(t) = -2tu^2(t)$$

$$u(0) = 2$$

```
Solution :
```

$$u(t) = \frac{1}{t^2 + \frac{1}{2}}$$



37.1.6. Correction : étude empirique de l'ordre de convergence

On calcule la solution approchée avec différentes valeurs de $h_k = 1/N_k$, à savoir $N_k = 2^3$, … 2^{10}. On sauvegarde les valeurs de h_k dans le vecteur `H`.

Pour chaque valeur de h_k, on calcule le maximum de la valeur absolue de l'erreur et on sauvegarde toutes ces erreurs dans le vecteur `err` de sort que `err[k]` contient $e_k = \max_{i=0,\dots,N_k} |y(t_i) - u_i|$ avec $N_k = 2^{k+1}$.

Pour afficher l'ordre de convergence on utilise une échelle logarithmique : on représente $\ln(h)$ sur l'axe des abscisses et $\ln(\text{err})$ sur l'axe des ordonnées ainsi, si $\text{err} = Ch^p$, alors $\ln(\text{err}) = \ln(C) + p\ln(h)$. En échelle logarithmique, p représente donc la pente de la ligne droite $\ln(\text{err})$.

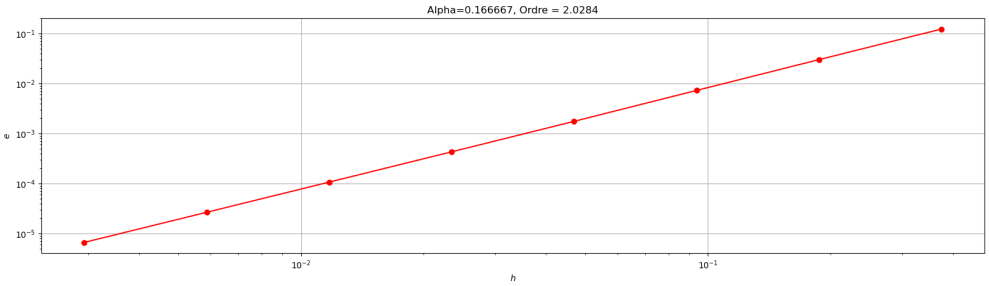
Pour estimer l'ordre de convergence on estime la pente de la droite qui relie l'erreur au pas k à l'erreur au pas $k+1$ en echelle logarithmique en utilisant la fonction `polyfit` basée sur la régression linéaire.

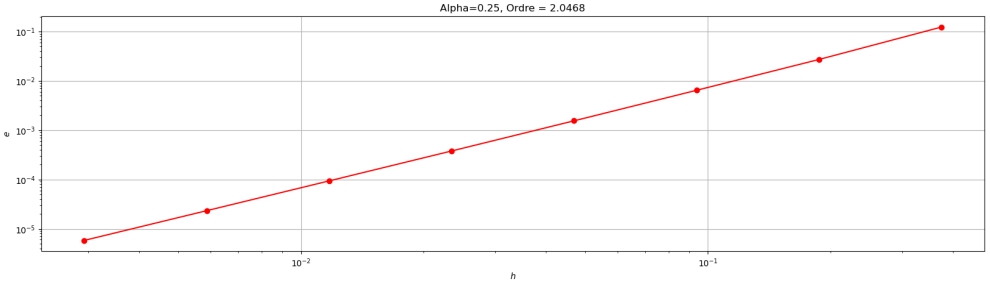
```
t0, y0, tfinal = 0, 2, 3
phi = lambda t,y : -2*t*y**2

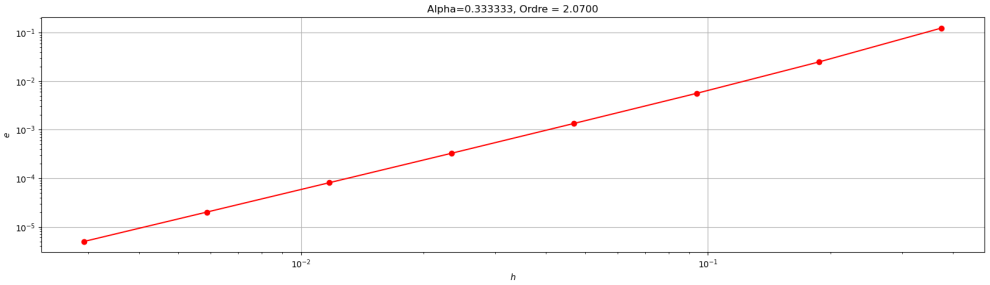
i = 0
for alpha in ALPHA:
    alpha = float(alpha)
    HH = []
    err = []
    for k in range(8):
        N = 2**(k+3)
        tt = np.linspace(t0,tfinal,N+1)
        HH.append( tt[1]-tt[0] )
        yy = sol_exacte(tt)
        uu = RKalpha(phi,tt,y0)
        err.append(np.linalg.norm(uu-yy, np.inf))

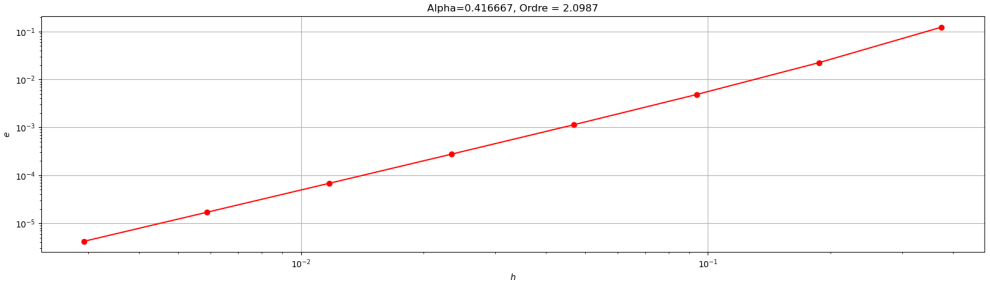
    i += 1
    plt.figure(i,figsize=(20,5))
    plt.loglog(HH,err, 'r-o')
    pente = np.polyfit(np.log(HH), np.log(err), 1)[0]
    plt.title(f'Alpha={alpha:g}, Ordre = {pente:.4f}')
```

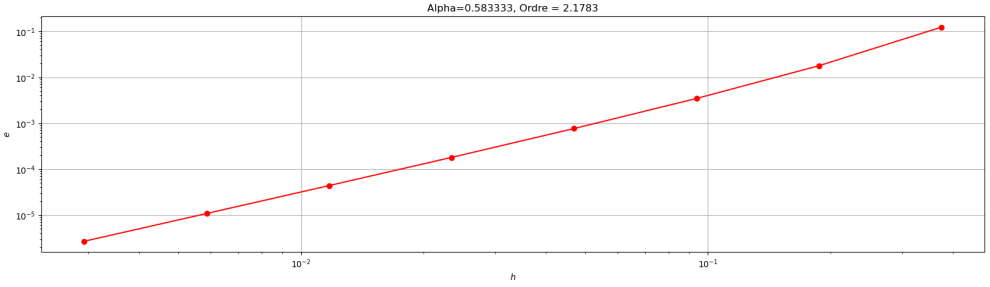
```
    plt.xlabel('$h$')
    plt.ylabel('$e$')
    #legend()
    plt.grid(True)
```

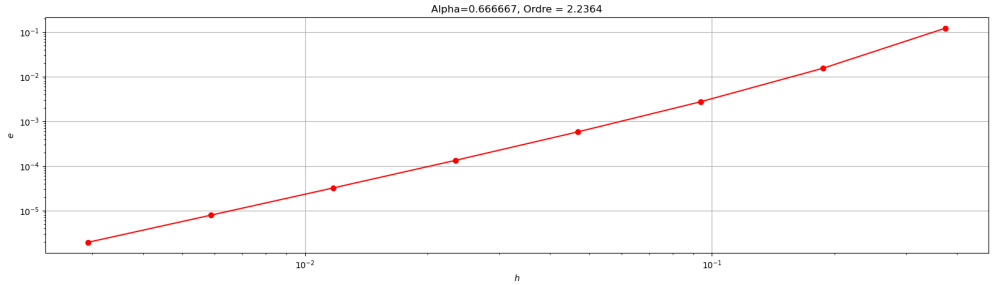


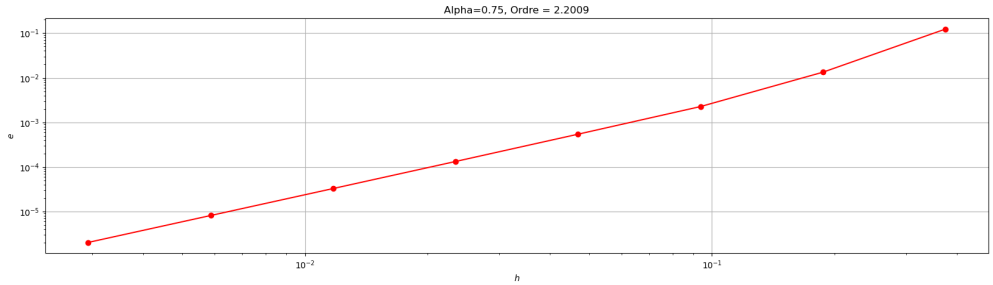


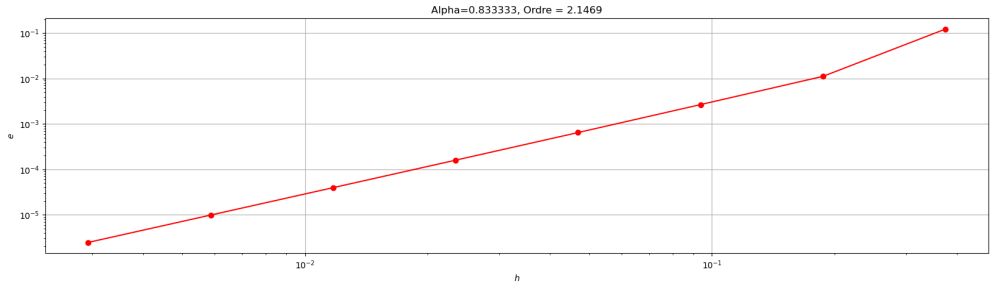


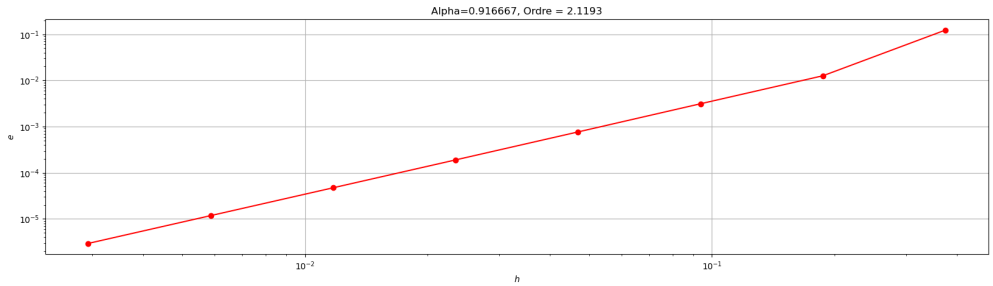





Alpha=0.666667, Ordre = 2.2364
e
h


Alpha=0.75, Ordre = 2.2009
e
h


Alpha=0.833333, Ordre = 2.1469
e
h


Alpha=0.916667, Ordre = 2.1193
e
h

Chapitre 38.

2019 Contrôle Terminale session 1

```
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp
from scipy.optimize import fsolve
from IPython.display import display, Markdown, Math
```

38.1. EXERCICE 1 : choix d'un schéma

Soit un problème de Cauchy **mathématiquement bien posé** mais **numériquement mal posé** (i.e. sensible aux perturbations sur la donnée initiale). Parmi les schémas EE (Euler Explicite), EI (Euler Implicite) et H (Heun), lequel faut-il choisir ? Justifier la réponse.

Toute méthode numérique induisant une erreur sur la solution, cette erreur va s'amplifier avec les itérations successives puisque le problème est **numériquement mal posé**. La seule issue pour traiter ce genre de problème est d'**utiliser des schémas d'ordre élevé** (ici donc le schéma de Heun qui est d'ordre 2 tandis que les schémas d'Euler sont d'ordre 1) avec un pas h petit (donc ce n'est pas la condition de A-stabilité qui limite le pas).

38.2. EXERCICE 2 : choix d'un schéma

Soit le problème de Cauchy

$$\begin{cases} y'(t) = -10^5 y(t), \quad t \in [0;100] \\ y(0) = 1 \end{cases}$$

Parmi les schémas EE (Euler Explicite), EI (Euler Implicite) et H (Heun), lequel faut-il choisir ? Justifier la réponse.

Le problème est **bien posé mathématiquement et numériquement**.
On ne peut pas utiliser ni le schéma EE ni le schéma H car la **condition de stabilité** est trop contraignante : dans les deux cas il faut un pas $h < \frac{2}{10^5}$, i.e. une discrétisation avec $N > \frac{100}{h} = \frac{10^7}{2}$ points.

Le schéma EI, en revanche, est inconditionnellement **A-stable** : on peut donc choisir le pas h sans contrainte de stabilité.

38.3. EXERCICE 3 : schéma predictor-corrector

Soit le problème de Cauchy

$$\begin{cases} y'(t) = 1 + \vartheta\left(t - y(t)\right), \quad t \in [0;2] \\ y(0) = 1 \end{cases}$$

1. Calculer la solution exacte de ce problème de Cauchy.

2. Soit les deux schémas

$$\begin{cases} u_0 = y_0, \\ u_1 = u_0 + h\varphi(t_1, u_1), \\ u_{n+1} = \frac{4}{3}u_n - \frac{1}{3}u_{n-1} + \frac{2}{3}h\varphi(t_{n+1}, u_{n+1}) \quad n = 1, 2, 3, \dots N-1 \end{cases}$$

$$\begin{cases} u_0 = y_0, \\ u_1 = u_0 + h\varphi(t_0, u_0), \\ u_{n+1} = u_n + \frac{h}{2}\Big(3\varphi(t_n, u_n) - \varphi(t_{n-1}, u_{n-1})\Big) \quad n = 1, 2, 3, 4, 5, \dots N-1 \end{cases}$$

   Écrire le schéma predictor-corrector associé à ce couple de schémas sous la forme d'une suite définie par récurrence.
3. Implémenter le schéma predictor-corrector de la question précédente et le tester avec le problème de Cauchy. On choisira $N = 1 + 5\vartheta$ points de discrétisation et on affichera la solution approchée et la solution exacte sur le même repère.
4. Vérifier empiriquement l'ordre de convergence du schéma predictor-corrector de la question précédente en affichant la courbe de convergence et en estimant sa pente avec le problème de Cauchy.

Chaque sujet correspond à une valeur de ϑ parmi les suivantes :

```
THETA = list(range(10,54,5))
THETA
```

```
[10, 15, 20, 25, 30, 35, 40, 45, 50]
```

Correction

On peut calculer la solution à la main :

$$a(t)y'(t) + b(t)y(t) = g(t)$$

avec $a(t) = 1$, $b(t) = \vartheta$ et $g(t) = 1 + \vartheta t$ donc $+ A(t) = \int \frac{b(t)}{a(t)}\mathrm{d}t = \int \vartheta \mathrm{d}t = \vartheta t$, $+ B(t) = \int \frac{g(t)}{a(t)}e^{A(t)}\mathrm{d}t = \int (1+\vartheta t)e^{\vartheta t}\mathrm{d}t = te^{\vartheta t}$,

d'où $y(t) = ce^{-\vartheta t} + t$.

En imposant la condition $1 = y(0)$ on trouve l'unique solution du problème de Cauchy donné :

$$\boxed{y(t) = e^{-\vartheta t} + t.}$$

Sinon, on peut utiliser le module de calcul symbolique.

```
t = sp.symbols('t')
y = sp.Function('y')

f = 1 + sp.symbols('theta')*(t-y(t))
edo = sp.Eq(sp.diff(y(t),t),f)
display(edo)

t0 = 0
y0 = 1
IC = sp.Eq(y(t0),y0)
display(IC)

sol = sp.dsolve(edo,y(t),ics={y(t0):y0})
display(sol)

sol_exacte = sp.lambdify((t, sp.symbols('theta')), sol.rhs, 'numpy')
```

$$\frac{d}{dt}y(t) = \theta\,(t - y(t)) + 1$$

$$y(0) = 1$$

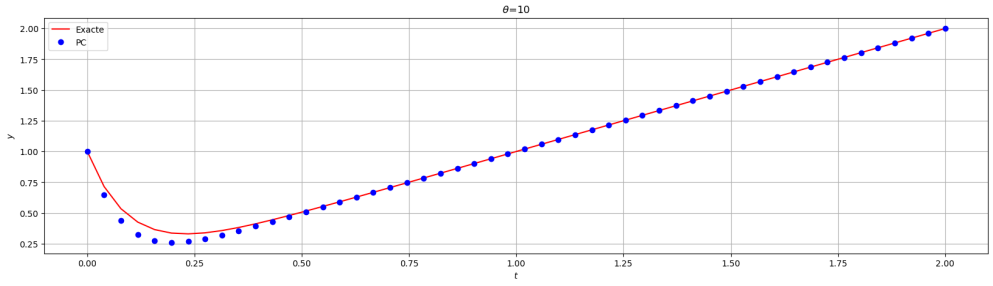
$$y(t) = t + e^{-t\theta}$$

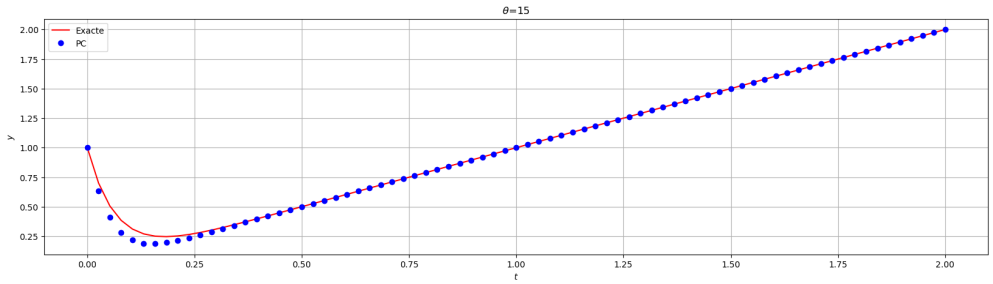
$$\begin{cases} u_0 = y_0, \\ \tilde{u}_1 = u_0 + h\varphi(t_0, u_0), \\ u_1 = u_0 + h\varphi(t_1, \tilde{u}_1), \\ \tilde{u}_{n+1} = u_n + \frac{h}{2}\Big(3\varphi(t_n, u_n) - \varphi(t_{n-1}, u_{n-1})\Big) \\ u_{n+1} = \frac{4}{3}u_n - \frac{1}{3}u_{n-1} + \frac{2}{3}h\varphi(t_{n+1}, \tilde{u}_{n+1}) \qquad n = 1, 2, 3, \ldots N-1 \end{cases}$$

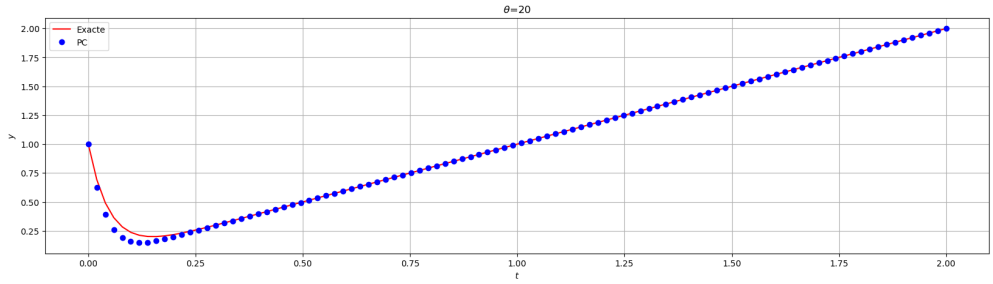
```
def PC(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = [y0]
    uu.append(uu[0]+h*phi(tt[0],uu[0]))
    for i in range(1,len(tt)-1):
        pred = uu[i] + h/2*( 3*phi(tt[i],uu[i]) - phi(tt[i-1],uu[i-1]) )
        uu.append( 4/3*uu[i] - uu[i-1]/3 + 2/3*h*phi(tt[i+1],pred) )
    return uu
```

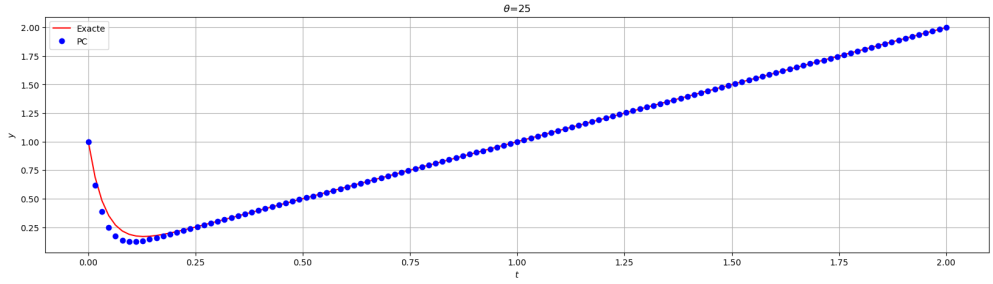
```
t0, y0, tfinal = 0, 1, 2

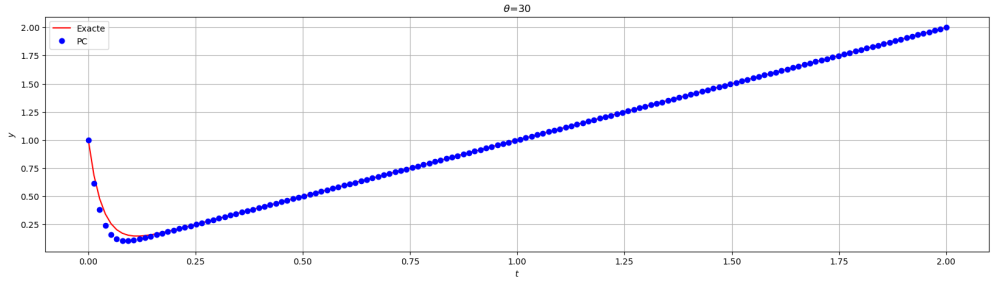
i = 0
for theta in THETA:
    N = 1+5*theta
    tt = np.linspace(t0,tfinal,N+1)
    phi = lambda t,y : 1+theta*(t-y)
    yy = sol_exacte(tt,theta)
    uu = PC(phi,tt,y0)
    i += 1
    plt.figure(i,figsize=(20,5))
    plt.plot(tt,yy,'r-',label=('Exacte'))
    plt.plot(tt,uu,'bo',label=('PC'))
    plt.title('$\\theta$={}'.format(theta))
    plt.xlabel('$t$')
    plt.ylabel('$y$')
    plt.legend()
    plt.grid(True)
```

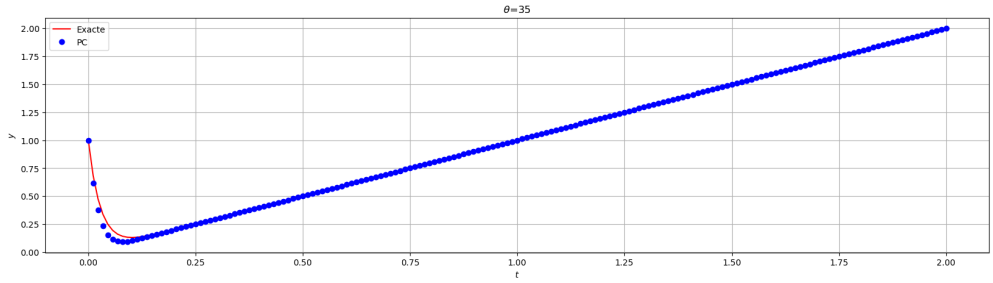


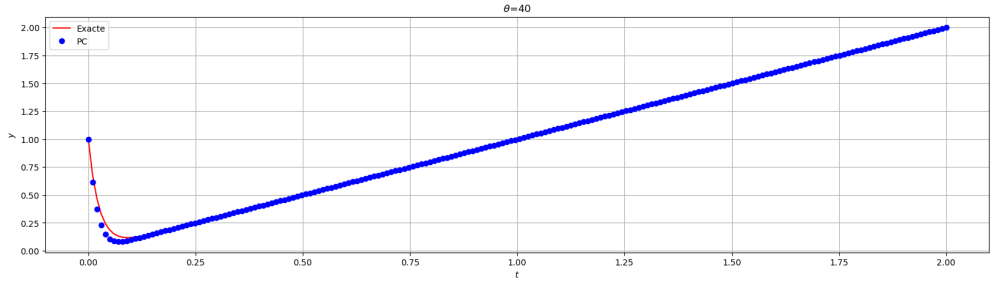


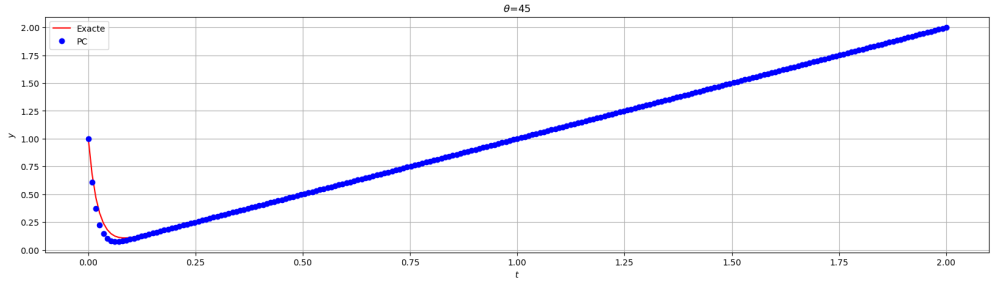

θ=20
Exacte
PC
y
t

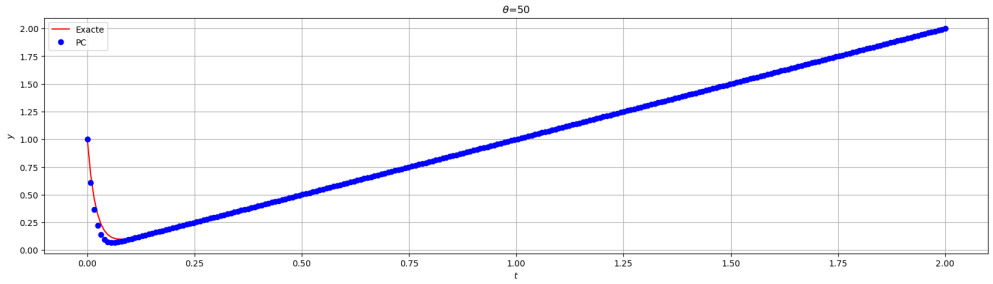

θ=25
Exacte
PC
y
t


θ=30
Exacte
PC
y
t


θ=35
Exacte
PC
y
t


θ=40
Exacte
PC
y
t


θ=45
Exacte
PC
y
t



On calcule la solution approchée avec différentes valeurs de $h_k = 2/N_k$, à savoir $N_k = (1 + 5\vartheta)2^k$, $k = 0, ...7$. On sauvegarde les valeurs de h_k dans le vecteur `H`.

Pour chaque valeur de h_k, on calcule le maximum de la valeur absolue de l'erreur et on sauvegarde toutes ces erreurs dans le vecteur `err` de sort que `err[k]` contient $e_k = \max_{i=0,\dots,N_k} |y(t_i) - u_i|$.

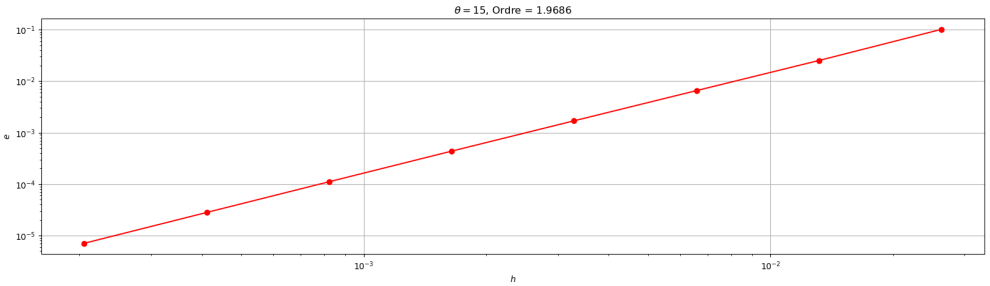
Pour afficher l'ordre de convergence on utilise une échelle logarithmique : on représente $\ln(h)$ sur l'axe des abscisses et $\ln(\text{err})$ sur l'axe des ordonnées ainsi, si $\text{err} = Ch^p$, alors $\ln(\text{err}) = \ln(C) + p\ln(h)$. En échelle logarithmique, p représente donc la pente de la ligne droite $\ln(\text{err})$.

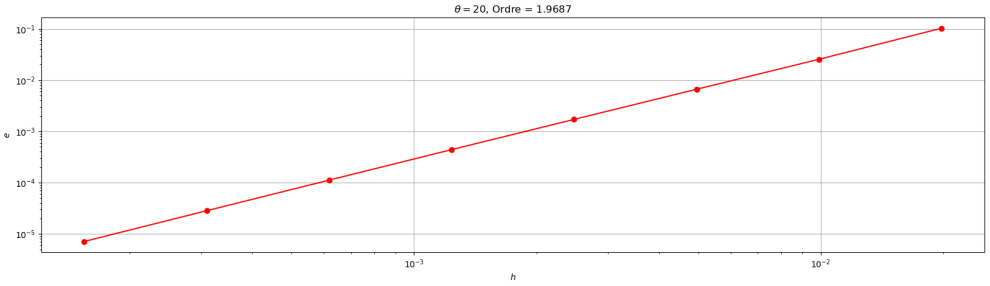
Pour estimer l'ordre de convergence on estime la pente de la droite qui relie l'erreur au pas k à l'erreur au pas $k+1$ en echelle logarithmique en utilisant la fonction `polyfit` basée sur la régression linéaire.

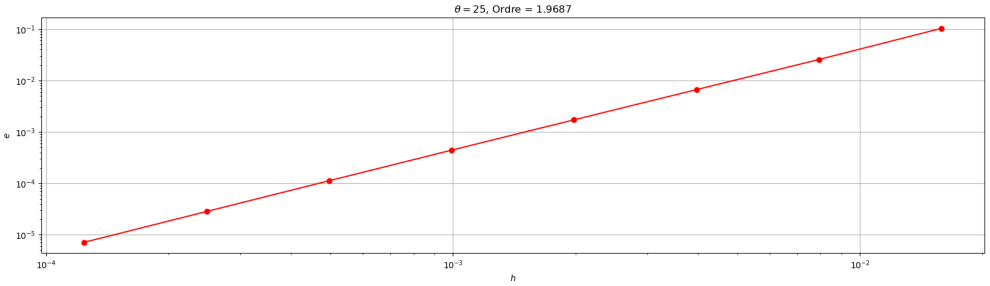
```
t0, y0, tfinal = 0, 1, 2

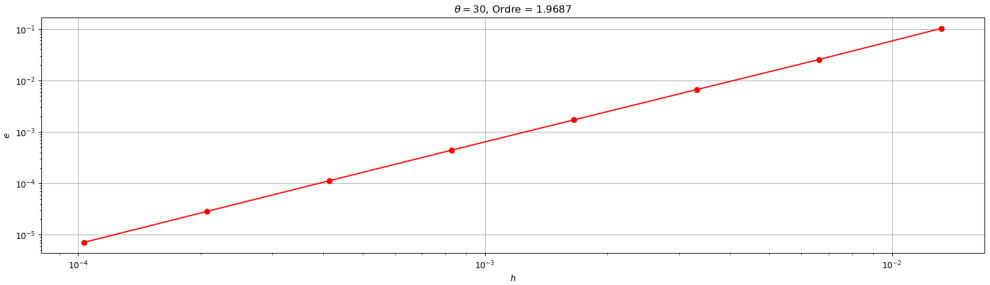
i = 0
for theta in THETA:
    H = []
    err = []
    for k in range(8):
        N = (1+5*theta)*2**k
        tt = np.linspace(t0,tfinal,N+1)
        h = tt[1]-tt[0]
        H.append(h)
        phi = lambda t,y : 1+theta*(t-y)
        yy = sol_exacte(tt,theta)
        uu = PC(phi,tt,y0)
        err.append(np.linalg.norm(np.array(uu)-np.array(yy),np.inf))
    i += 1
    plt.figure(i,figsize=(20,5))
    plt.loglog(H,err, 'r-o')
    pente = np.polyfit(np.log(H), np.log(err), 1)[0]
    plt.title('$\\theta=${}, Ordre = {:1.4f}'.format(theta,pente))
    plt.xlabel('$h$')
    plt.ylabel('$e$')
    plt.grid(True)
```

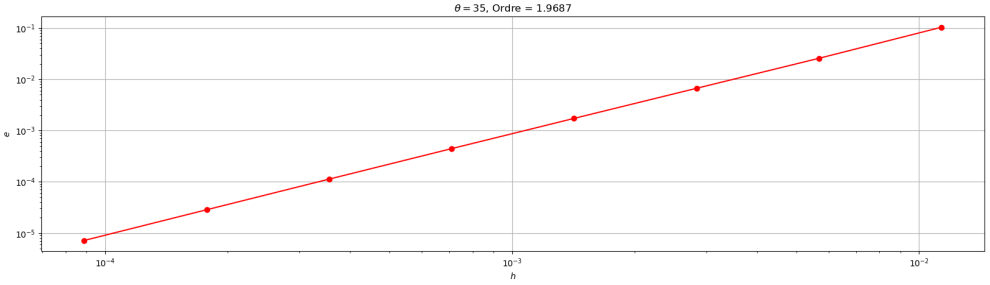


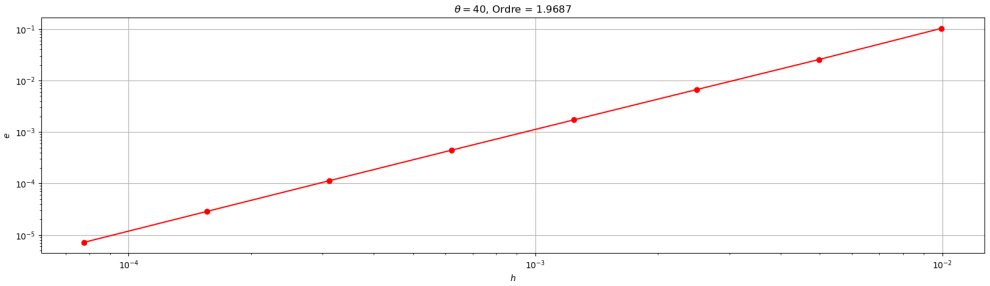

$\theta = 15$, Ordre = 1.9686
e
h

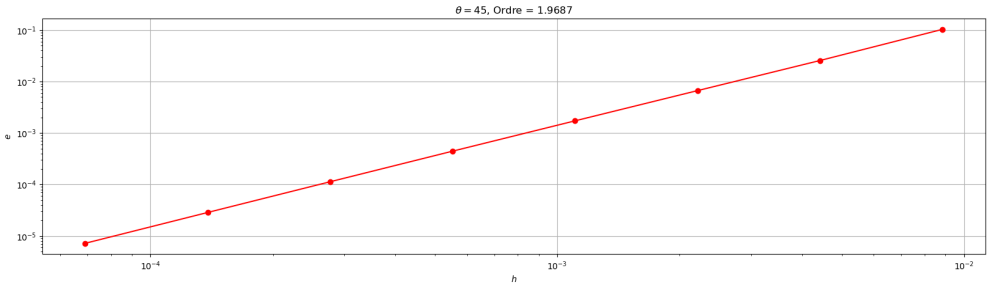

$\theta = 20$, Ordre = 1.9687
e
h

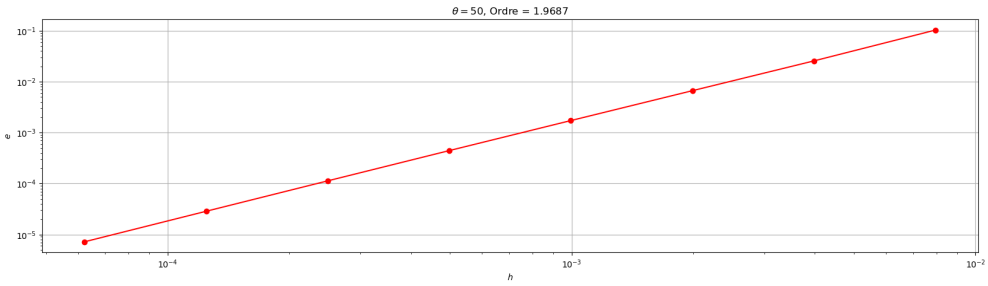

$\theta = 25$, Ordre = 1.9687
e
h


$\theta = 30$, Ordre = 1.9687
e
h


$\theta = 35$, Ordre = 1.9687
e
h


$\theta = 40$, Ordre = 1.9687
e
h





38.4. EXERCICE 4 : Schéma Runge-Kutta

Soit le schéma de Runge-Kutta dont la *matrice de Butcher* est

$$\begin{array}{c|cc} 1 & 1 & 0 \\ \alpha & \alpha & 0 \\ \hline & (1-\gamma) & \gamma \end{array}$$

1. Écrire le schéma sous la forme d'une suite définie par récurrence.
2. Étudier théoriquement l'ordre du schéma.
3. Implémenter le schéma et le tester avec le même problème de Cauchy qu'à l'exercice précedent. On choisira $N = 1 + 2\vartheta$ points de discrétisation et on affichera la solution approchée et la solution exacte sur le même repère. Attention : le schéma est semi-implicite, la fonction `fsolve` du module `scipy.optimize` est votre amie...

Chaque sujet correspond à un couple (α, γ) choisi parmi les suivants :

```
zeta = 1
GAMMA = [sp.Rational(13,24),sp.Rational(14,24),sp.Rational(15,24),sp.Rational(16,24),sp.Rational(17,24),
 ↪  sp.Rational(18,24),sp.Rational(19,24),sp.Rational(20,24),sp.Rational(21,24),sp.Rational(22,24),sp.Rational(23,24)]
for gamma in GAMMA:
    alpha = sp.Rational(sp.Rational(1,2)+zeta*(gamma-1),gamma)
    display(Markdown(rf"$$\alpha = {sp.latex(alpha)},\qquad \gamma = {sp.latex(gamma)}$$"))
```

$$\alpha = \frac{1}{13}, \qquad \gamma = \frac{13}{24}$$

$$\alpha = \frac{1}{7}, \qquad \gamma = \frac{7}{12}$$

$$\alpha = \frac{1}{5}, \qquad \gamma = \frac{5}{8}$$

$$\alpha = \frac{1}{4}, \qquad \gamma = \frac{2}{3}$$

$$\alpha = \frac{5}{17}, \qquad \gamma = \frac{17}{24}$$

$$\alpha = \frac{1}{3}, \qquad \gamma = \frac{3}{4}$$

$$\alpha = \frac{7}{19}, \qquad \gamma = \frac{19}{24}$$

$$\alpha = \frac{2}{5}, \qquad \gamma = \frac{5}{6}$$

$$\alpha = \frac{3}{7}, \qquad \gamma = \frac{7}{8}$$

$$\alpha = \frac{5}{11}, \qquad \gamma = \frac{11}{12}$$

$$\alpha = \frac{11}{23}, \qquad \gamma = \frac{23}{24}$$

Correction

Considérons le problème de Cauchy

$$\begin{cases} y'(t) = \varphi(t, y(t)), & \forall t \in I = [t_0, T], \\ y(t_0) = y_0, \end{cases}$$

avec y_0 une valeur donnée et supposons que l'on ait montré l'existence et l'unicité d'une solution y pour $t \in I$.

Pour $h > 0$ soit $t_n \overset{\text{déf.}}{=} t_0 + nh$ avec $n = 0, 1, 2, \dots, N$ une suite de $N + 1$ nœuds de I induisant une discrétisation de I en N sous-intervalles $I_n = [t_n; t_{n+1}]$ chacun de longueur $h > 0$ (appelé le *pas de discrétisation*).

Pour chaque nœud t_n, on cherche la valeur inconnue u_n qui approche la valeur exacte $y_n \equiv y(t_n)$.
- L'ensemble de $N + 1$ valeurs $\{t_0, t_1 = t_0 + h, ..., t_N = T\}$ représente les points de la discrétisation.
- L'ensemble de $N + 1$ valeurs $\{y_0, y_1, ..., y_N\}$ représente la solution exacte.
- L'ensemble de $N + 1$ valeurs $\{u_0 = y_0, u_1, ..., u_N\}$ représente la solution numérique. Cette solution approchée sera obtenue en construisant une suite récurrente.

Le schéma qu'on va construire permet de calculer implicitement u_{n+1} à partir de u_n par la formule de récurrence

$$\begin{cases} u_0 = y_0 \\ K_1 = \varphi\left(t_n + h, u_n + hK_1\right) \\ K_2 = \varphi\left(t_n + \alpha h, u_n + \alpha h K_1\right) \\ u_{n+1} = u_n + h\left((1-\gamma)K_1 + \gamma K_2\right) \quad n = 0, 1, \dots N-1 \end{cases}$$

- C'est un schéma semi-implicite à deux étages : l'ordre est au plus 4.
- $c_1 = a_{11} + a_{12}$, $c_2 = a_{21} + a_{22}$ et $1 = b_1 + b_2 \implies$ il est consistante
- $c_1 b_1 + c_2 b_2 = \frac{1}{2} \implies$ il est d'ordre 2
- $b_1 c_1^2 + b_2 c_2^2 = \frac{1}{2\gamma}$ et $b_1 a_{11} c_1 + b_2 a_{12} c_2 = \frac{1}{2} \implies$ il n'est pas d'ordre 3

On peut vérifier ces calculs avec `sympy` :

```
gamma = sp.Symbol(r'\gamma')
alpha = (gamma-sp.S(1)/2)/gamma
c = [1,alpha]
b = [1-gamma,gamma]
A = [[1,0],[alpha,0]]
s = len(c)


display(Markdown(  r'$$\sum_{j=1}^s b_j='+sp.latex(sum(b))+'$$'    ))

for i in range(s):
    display(Markdown(
        rf"$$i = {i} \quad \sum_{{j=1}}^{{{s}}} a_{{ij}} - c_i = {sp.latex(sum(A[i]) - c[i])}$$"
    ))

display(Markdown(
    rf"$$\sum_{{j=1}}^{{{s}}} b_j c_j = {sp.latex(sum(b[i]*c[i] for i in range(s)))}$$"
))
display(Markdown(
    rf"$$\sum_{{j=1}}^{{{s}}} b_j c_j^2 = {sp.latex(sum(b[i]*c[i]**2 for i in range(s)).simplify())}$$"
))
display(Markdown(
    rf"$$\sum_{{i,j=1}}^{{{s}}} b_i a_{{ij}} c_j = {sp.latex(sum(b[i]*A[i][j]*c[j] for i in range(s) for
    ↪  j in range(s)))}$$"
))
```

$$\sum_{j=1}^{s} b_j = 1$$

$$i = 0 \quad \sum_{j=1}^{2} a_{ij} - c_i = 0$$

$$i = 1 \quad \sum_{j=1}^{2} a_{ij} - c_i = 0$$

$$\sum_{j=1}^{2} b_j c_j = \frac{1}{2}$$

$$\sum_{j=1}^{2} b_j c_j^2 = \frac{1}{4\gamma}$$

$$\sum_{i,j=1}^{2} b_i a_{ij} c_j = \frac{1}{2}$$

Dans chaque point t_i, il faut approcher $(K_1)_i$ en resolvant une équation. Si on utilise la fonction `fsolve` du module `scipy.optimize`, il faut initialiser `fsolve` avec une approximation de $(K_1)_i$. On choisira donc $\varphi(t_i, u_i)$:

```
def RK(phi,tt,y0):
    h = tt[1]-tt[0]
    uu = np.empty_like(tt)
    uu[0] = y0
    for n in range(len(tt)-1):
        k1 = fsolve( lambda x : x - phi( tt[n+1], uu[n]+h*x ) , phi(tt[n],uu[n]) )[0]
        k2 = phi( tt[n]+alpha*h , uu[n]+alpha*h*k1 )
        uu[n+1] = uu[n] + h*((1-gamma)*k1+gamma*k2)
    return uu
```

Comparons solution exacte et solution approchée (pour trouver votre sujet vous devez chercher les valeurs de ϑ et β correspondants) :

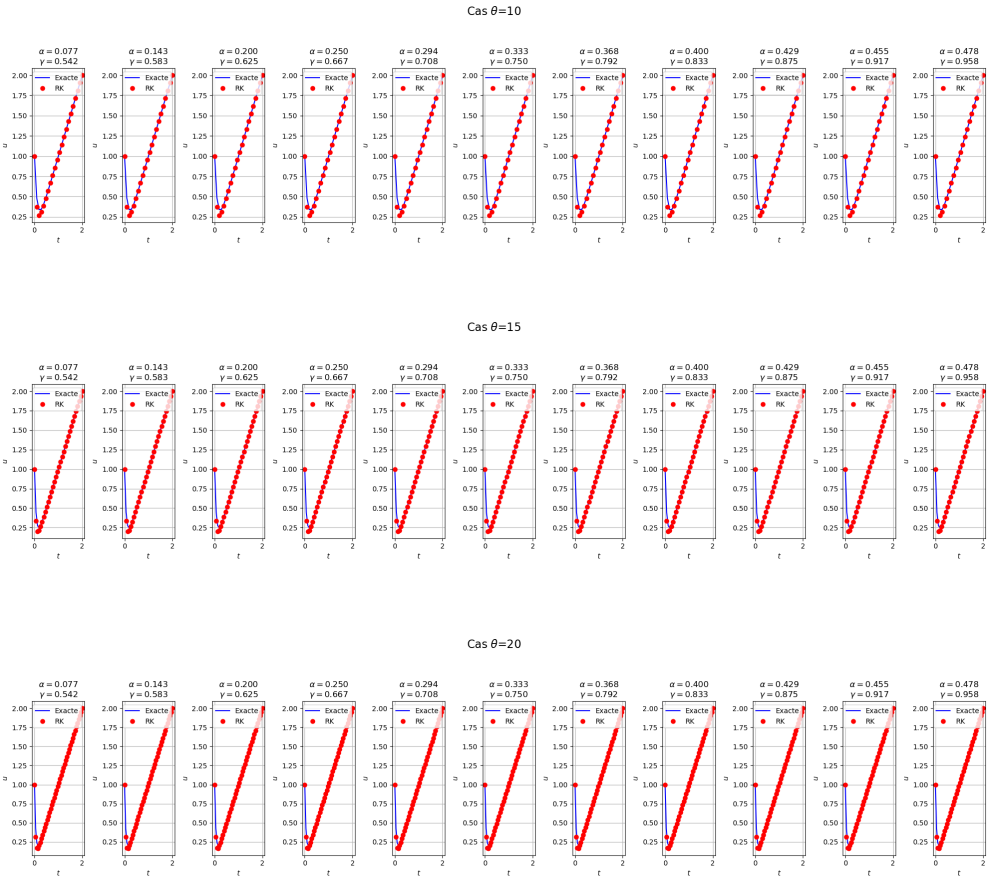
```
t0, y0, tfinal = 0, 1, 2

for theta in THETA:

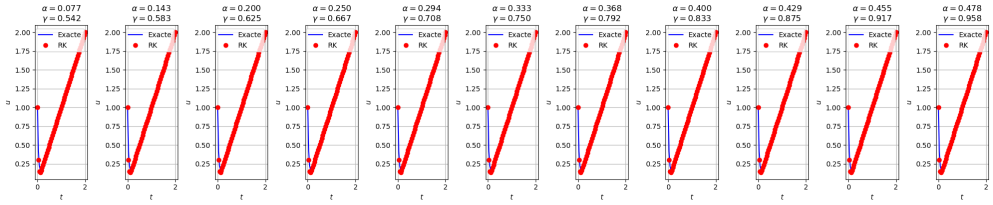
    N = 1+2*theta
    tt = np.linspace(t0,tfinal,N+1)
    sol_exacte = lambda t : t+np.exp(-theta*t)
    phi = lambda t,y : 1+theta*(t-y)
    yy = sol_exacte(tt)
    fig = plt.figure(theta,figsize=(20,5))
    fig.suptitle(rf'Cas $\theta$={theta}', fontsize=16)

    g = len(GAMMA)
    for i, gamma in enumerate(GAMMA, start=1):
        alpha = float(sp.Rational(sp.Rational(1,2)+zeta*(gamma-1),gamma))
        uu = RK(phi,tt,y0)
        plt.subplot(1,g,i)
        plt.plot(tt,yy,'b-',label=("Exacte"))
        plt.plot(tt,uu,'ro',label=("RK"))
        plt.title(rf'$\alpha={alpha:.3f}$'+'\n'+rf'$\gamma={float(gamma):.3f}$')
        plt.xlabel('$t$')
        plt.ylabel('$u$')
        plt.legend()
        plt.grid(True)

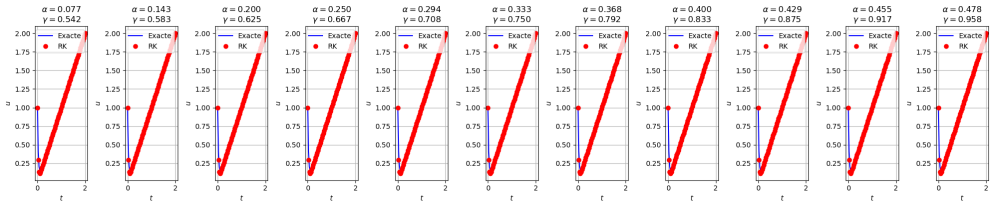
    fig.tight_layout(rect=[0, 0, 1, 0.92])
```



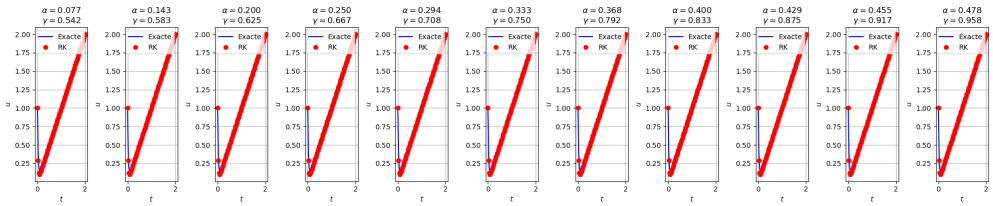
Cas θ=25



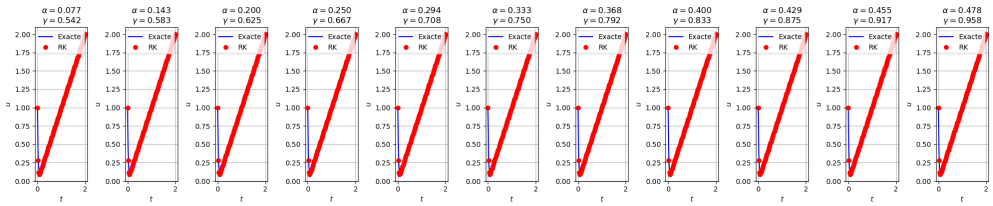
Cas θ=30



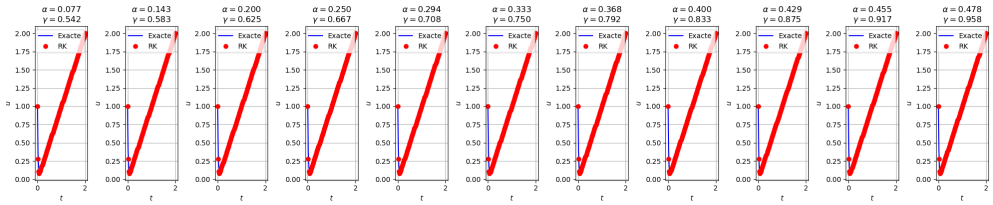
Cas θ=35



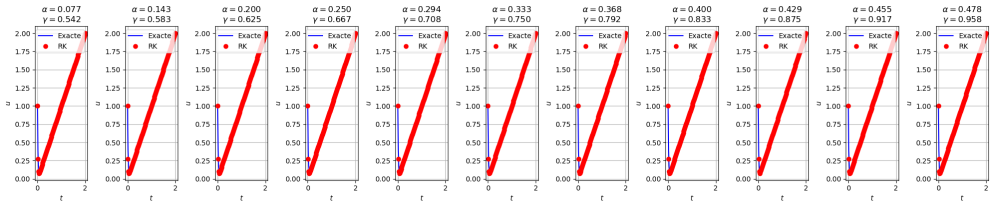
Cas θ=40



Cas θ=45



Cas θ=50



Chapitre 39.

2019 Contrôle Terminale session 2

```
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp

from IPython.display import display, Markdown, Math
```

39.1. EXERCICE 1 : Choix d'un schéma

Considérons le problème de Cauchy

$$\begin{cases} y'(t) = y(t) - 2e^{-t}, \\ y(0) = a \end{cases}$$

1. Calculer la solution exacte de ce problème de Cauchy en fonction de a pour $t > 0$. 2. Tracer son graphe sur l'intervalle $[0; 4]$ pour $a = 1$, $a = 1.1$ et $a = 0.9$. 3. Soit $a = 1$. Parmi les schémas EE (Euler Explicite), EI (Euler Implicite) et EM (Euler modifié), lequel faut-il choisir ? Justifier la réponse en 5 lignes au plus.

Correction

Il s'agit d'une EDO linéaire d'ordre 1, on peut donc calculer la solution à la main :

$$a(t)y'(t) + b(t)y(t) = g(t)$$

avec $a(t) = 1$, $b(t) = -1$ et $g(t) = -2e^{-t}$ donc $+ A(t) = \int \frac{b(t)}{a(t)} \mathrm{d}t = \int -1 \mathrm{d}t = -t$, $+ B(t) = \int \frac{g(t)}{a(t)} e^{A(t)} \mathrm{d}t = \int -2e^{-t}e^{-t} \mathrm{d}t = e^{-2t}$,

d'où $y(t) = ce^{-A(t)} + B(t)e^{-A(t)} = ce^{t} + e^{-t}$.

En imposant la condition $a = y(0)$ on trouve l'unique solution du problème de Cauchy donné :

$$\boxed{y(t) = (a-1)e^{t} + e^{-t}.}$$

Sinon, on peut utiliser le module de calcul symbolique.

```
sp.var('t,a')
u = sp.Function('u')
f = u(t)-2*sp.exp(-t)
edo = sp.Eq(sp.diff(u(t),t),f)
print("EDO:")
display(edo)

solgen = sp.dsolve(edo,u(t)).simplify()
print("Solution générale:")
display(solgen)

t0 = 0
u0 = a
```

```
consts = sp.solve( [solgen.rhs.subs(t,t0)-u0 ], dict=True)[0]
print(f"Prise en compte de la condition initiale u({t0})={u0}:")
display(consts)

solpar = solgen.subs(consts)
print("Solution du problème de Cauchy:")
display(solpar)

print("Affichage de trois solutions")
p = sp.plot(solpar.rhs.subs({a:1}), (t,0,4), line_color='b', legend = True, show=False)
p.extend(sp.plot(solpar.rhs.subs({a:1.1}), (t,0,4), line_color='m', legend = True, show=False))
p.extend(sp.plot(solpar.rhs.subs({a:0.9}), (t,0,4), line_color='r', legend = True, show=False))
p[0].label="a=1"
p[1].label="a=1.1"
p[2].label="a=0.9"
p.show();
```

EDO:

$$\frac{d}{dt}u(t) = u(t) - 2e^{-t}$$

Solution générale:

$$u(t) = C_1 e^{t} + e^{-t}$$

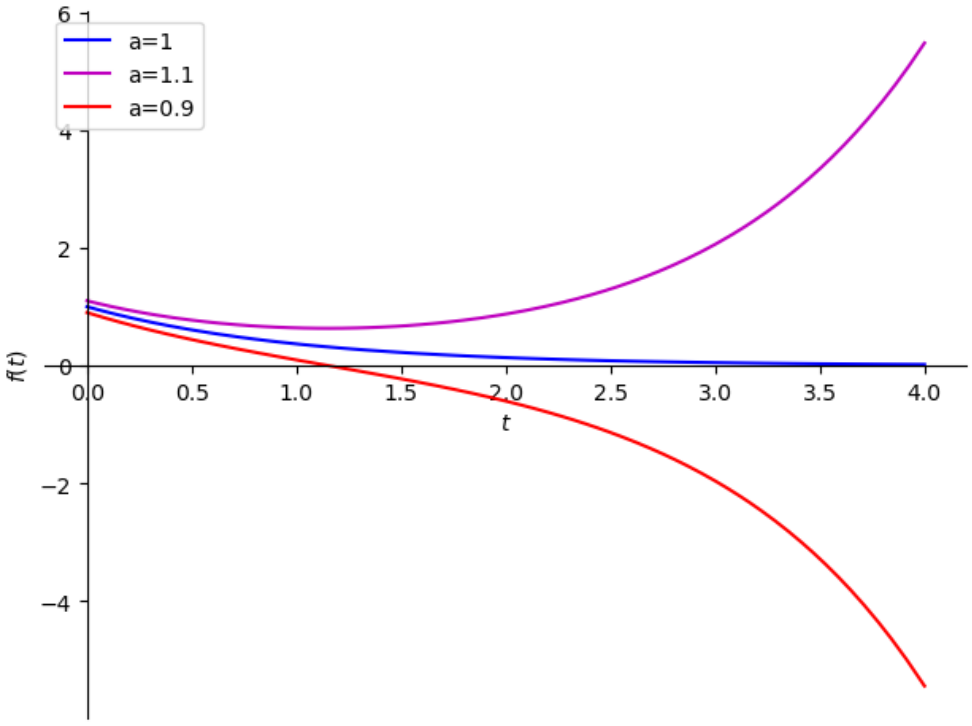
Prise en compte de la condition initiale u(0)=a:

{C1: a - 1}

Solution du problème de Cauchy:

$$u(t) = (a-1)\, e^{t} + e^{-t}$$

Affichage de trois solutions



Lorsque $a = 1$, le problème est **numériquement mal posé**, i.e. sensible aux perturbations sur la donnée initiale. Toute méthode numérique induisant une erreur sur la solution, cette erreur va s'amplifier avec les itérations successives. La seule issue pour traiter ce genre de problème est d'**utiliser des schémas d'ordre élévé** (ici donc le schéma EM qui est d'ordre 2 tandis que les schémas EE et EI sont d'ordre 1) avec un pas h petit.

39.2. EXERCICE 2 : Circuit LC

Un circuit LC est un circuit électrique contenant une bobine (L) et un condensateur (Capacité). C'est ainsi qu'on obtient le phénomène de résonance électrique.

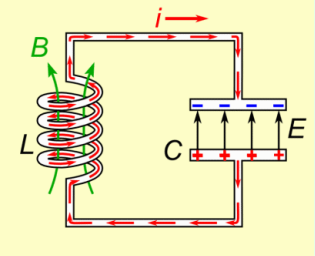


FIGURE 39.1. – LC_circuit.png

Dans un circuit électrique de type LC en série, le courant I, en ampères (A), et la charge électrique du condensateur q, en coulombs (C), évoluent avec le temps selon le système d'équations différentielles

$$\begin{cases} q'(t) = I(t) \\ LI'(t) + \dfrac{1}{C} q(t) = 0 \end{cases}$$

avec

- $L > 0$ l'inductance de la bobine, en henrys (H) ;
- $C > 0$ la capacité électrique du condensateur, en farads (F) ;
- t le temps, en secondes (s).

Pour nos calculs, on prendra les valeurs suivantes :

- $L = 2$ H,
- $C = 0.5$ F,
- $q(0) = 1$ C,
- $i(0) = 2$ A.

---

1. Écrire le système sous forme matricielle ainsi que sous la forme d'une EDO d'ordre 2 en l'inconnue $t \mapsto q(t)$. Calculer la solution exacte et afficher $t \mapsto q(t)$, $t \mapsto i(t)$ et $q \mapsto i(q)$.
2. Afficher $t \mapsto q(t)$, $t \mapsto i(t)$ et $q \mapsto i(q)$ en comparant solution exacte et solution approchée obtenue avec la méthode d'Euler Explicite pour $t \in [0; 7]$ avec $N = 101$ points.
3. Même exercice pour la méthode de Heun.
4. Même exercice pour la méthode de Euler-Cromer. Cette méthode, appelée Euler-Cromer ou Euler symplectique ou Newton-Störmer-Verlet, s'écrit comme suit :

$$\begin{cases} I_{n+1} = I_n + h\varphi_2(t_n, q_n, I_n) \\ q_{n+1} = q_n + h\varphi_1(t_{n+1}, q_{n+1}, I_{n+1}), \end{cases} \quad \leftarrow \text{ce calcul est explicite car } \varphi_1 \text{ ne dépend pas de } q$$

Correction Q1

On a l'écriture matricielle

$$\begin{pmatrix} q \\ I \end{pmatrix}' (t) = \begin{pmatrix} 0 & 1 \\ -\frac{1}{LC} & 0 \end{pmatrix} \begin{pmatrix} q \\ I \end{pmatrix} (t)$$

soit encore

$$\begin{cases} q'(t) = \varphi_1(t, q(t), I(t)) \\ I'(t) = \varphi_2(t, q(t), I(t)) \end{cases} \qquad \text{avec} \quad \begin{cases} \varphi_1(t, q(t), I(t)) = I(t) \\ \varphi_2(t, q(t), I(t)) = -\frac{1}{LC} q(t) \end{cases}$$

Transformons le système en une EDO d'ordre 2 :

$$\begin{cases} LI'(t) + \frac{1}{C} q(t) = E(t) \\ q'(t) = I(t) \implies q''(t) = I'(t) \implies Lq''(t) = LI'(t) \end{cases}$$

donc

$$Lq''(t) = LI'(t) = -\frac{1}{C} q(t) + E(t) = -\frac{1}{C} q(t) + E(t)$$

ainsi

$$\boxed{Lq''(t) + \frac{1}{C} q(t) = 0} \text{ et } \boxed{I(t) = q'(t)}$$

avec

$$\begin{cases} q(0) = q_0, \\ q'(0) = i_0. \end{cases}$$

On peut calculer la solution exacte à la main :

$$\begin{cases} q''(t) + \frac{1}{LC} q(t) = 0 \\ q(0) = q_0, \\ q'(0) = i_0 \end{cases}$$

Il s'agit d'une EDO d'ordre 2 linéaire à coefficients constants et homogène. Le polynôme caractéristique est $\lambda^2 + \frac{1}{LC} = 0$ et a les deux racines complexes conjugués $\lambda_1 = -i\frac{1}{\sqrt{LC}}$ et $\lambda_2 = i\frac{1}{\sqrt{LC}}$ ainsi la solution générale est

$$q(t) = C_1 \cos\left(\frac{1}{\sqrt{LC}} t\right) + C_2 \sin\left(\frac{1}{\sqrt{LC}} t\right), \qquad C_1, C_2 \in \mathbb{R}.$$

Sinon, on peut utiliser le module de calcul symbolique.

```
sp.var('t,t_0,q_0,i_0')
L = sp.Symbol('L',positive=True)
C = sp.Symbol('C',positive=True)

q = sp.Function('q')
edo = sp.Eq(sp.diff(q(t),t,t) , -q(t)/(L*C) )
display(edo)

solgen = sp.dsolve(edo,q(t)).simplify()
display(solgen)
```

$$\frac{d^2}{dt^2} q(t) = -\frac{q(t)}{CL}$$

$$q(t) = C_1 \sin\left(\frac{t}{\sqrt{C}\sqrt{L}}\right) + C_2 \cos\left(\frac{t}{\sqrt{C}\sqrt{L}}\right)$$

Avec les valeurs données :

```
pb = { L:2 , C:sp.S(1)/2 }
solgenLC = solgen.subs(pb)
display(solgenLC)
```

$$q(t) = C_1 \sin(t) + C_2 \cos(t)$$

```
t_0 = 0
q_0 = 1
i_0 = 2

consts = sp.solve( [solgenLC.rhs.subs(t,t_0)-q_0 , solgenLC.rhs.diff(t).subs(t,t_0)- i_0],  dict=True)[0]
display(consts)

solparLC = solgenLC.subs(consts)
display(solparLC)

I = sp.Function('I')
I = solparLC.rhs.diff(t).simplify()
display(Math(r"I(t)="+sp.latex( I.subs(pb) )))
```

```
{C1: 2, C2: 1}
```

$q(t) = 2\sin(t) + \cos(t)$

$I(t) = -\sin(t) + 2\cos(t)$

On transforme les deux fonctions `sympy` en des fonctions `numpy` pour l'évaluation et l'affichage :

```
Qfunc = sp.lambdify(t,solparLC.rhs,'numpy')
Ifunc = sp.lambdify(t,I,'numpy')

tt = np.linspace(0,7,101)
yyQ = Qfunc(tt)
yyI = Ifunc(tt)

plt.figure(1,figsize=(17,12))

plt.subplot(2,3,1)
plt.plot(tt,yyQ);
plt.xlabel('$t$ (s)')
plt.ylabel('$q$ (C)')
plt.grid(True)

plt.subplot(2,3,2)
plt.plot(tt,yyI);
plt.xlabel('$t$ (s)')
plt.ylabel('$I$ (A)')
plt.grid(True)

plt.subplot(2,3,3)
plt.plot(yyQ,yyI);
plt.xlabel('$q$ (C)')
plt.ylabel('$I$ (A)')
plt.grid(True)

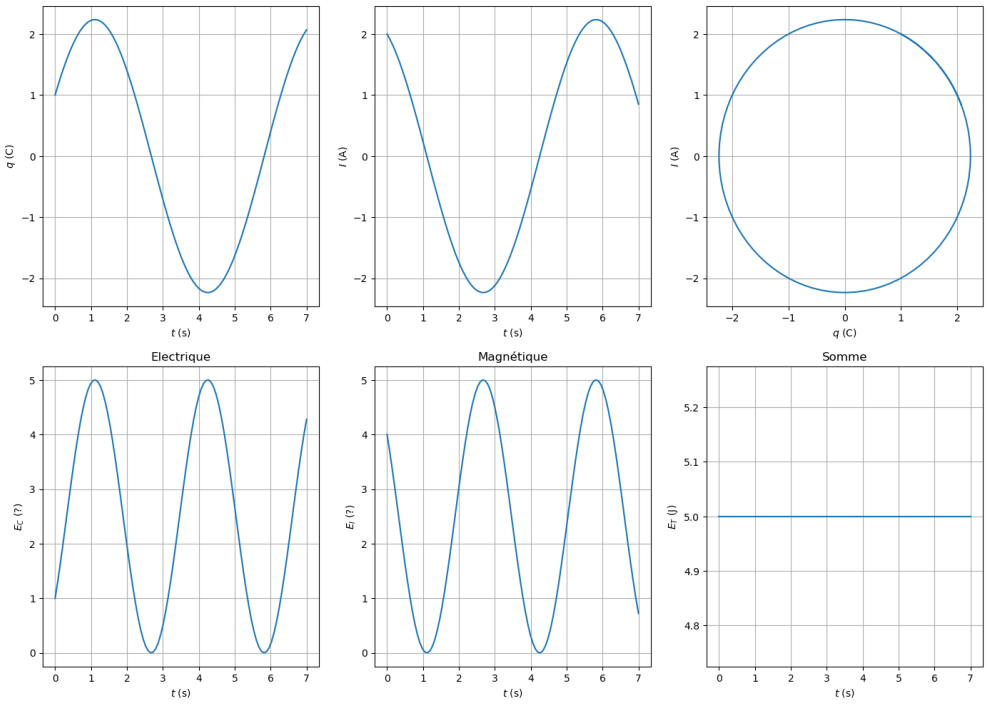
L = 2
C = 1/2

Ee=[0.5*(yyQ[i])**2/C for i in range(len(yyQ))]
plt.subplot(2,3,4)
plt.plot(tt,Ee);
plt.xlabel('$t$ (s)')
plt.ylabel('$E_C$ (?)')
plt.title("Electrique")
plt.grid(True)

Em=[0.5*L*(yyI[i])**2 for i in range(len(yyQ))]
plt.subplot(2,3,5)
```

```
plt.plot(tt,Em);
plt.xlabel('$t$ (s)')
plt.ylabel('$E_I$ (?)')
plt.title("Magnétique")
plt.grid(True)


Et=[Em[i]+Ee[i] for i in range(len(yyQ))]
plt.subplot(2,3,6)
plt.plot(tt,Et);
plt.xlabel('$t$ (s)')
plt.ylabel('$E_T$ (J)')
plt.title("Somme")
plt.grid(True)
```



Correction Q2

```
t0, tfinal = 0, 7
N = 51
q_0,i_0=1,2

phi1 = lambda t,q,qp : qp
phi2 = lambda t,q,qp : -q/(L*C)

tt = np. linspace(t0,tfinal,N)
yy1 = [Qfunc(t) for t in tt]
yy2 = [Ifunc(t) for t in tt]
```

$$\begin{cases} q_{n+1} = q_n + h\varphi_1(t_n, q_n, I_n) \\ I_{n+1} = I_n + h\varphi_2(t_n, q_n, I_n) \end{cases}$$

```
def EE(phi1,phi2,tt,q_0,i_0):
    h = tt[1]-tt[0]
    uu1 = [q_0]
    uu2 = [i_0]
```

```
    for i in range(len(tt)-1):
        uu1.append( uu1[i] + h*phi1(tt[i],uu1[i],uu2[i]) )
        uu2.append( uu2[i] + h*phi2(tt[i],uu1[i],uu2[i]) )
    return [uu1,uu2]
```

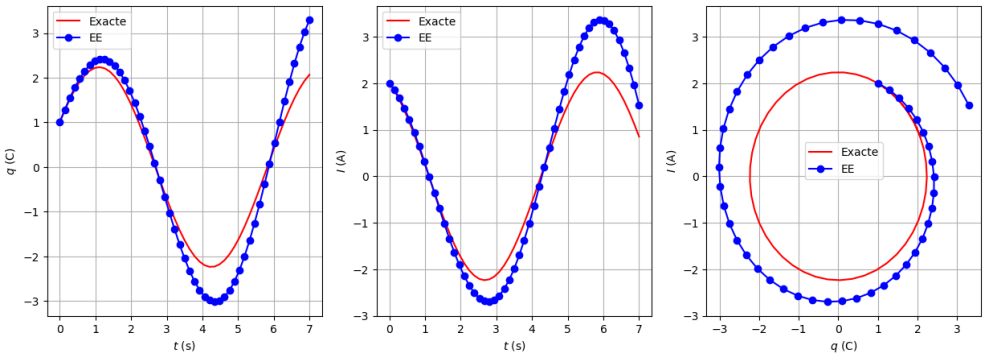
```
uu1E,uu2E = EE(phi1,phi2,tt,q_0,i_0)

plt.figure(1,figsize=(15,5))

plt.subplot(1,3,1)
plt.plot(tt,yy1,'r-',label=('Exacte'))
plt.plot(tt,uu1E,'b-o',label=('EE'))
plt.xlabel('$t$ (s)')
plt.ylabel('$q$ (C)')
plt.legend()
plt.grid(True)

plt.subplot(1,3,2)
plt.plot(tt,yy2,'r-',label=('Exacte'))
plt.plot(tt,uu2E,'b-o',label=('EE'))
plt.xlabel('$t$ (s)')
plt.ylabel('$I$ (A)')
plt.legend()
plt.grid(True)

plt.subplot(1,3,3)
plt.plot(yy1,yy2,'r-',label=('Exacte'))
plt.plot(uu1E,uu2E,'b-o',label=('EE'))
plt.xlabel('$q$ (C)')
plt.ylabel('$I$ (A)')
plt.legend()
plt.grid(True)
```



Correction Q3

$$\begin{cases} \tilde{q} = q_n + h\varphi_1(t_n, q_n, I_n), \\ \tilde{I} = I_n + h\varphi_2(t_n, q_n, I_n), \\ q_{n+1} = q_n + \frac{h}{2}\Big(\varphi_1(t_n, q_n, I_n) + \varphi_1(t_{n+1}, \tilde{q}, \tilde{I})\Big) \\ I_{n+1} = I_n + \frac{h}{2}\Big(\varphi_2(t_n, q_n, I_n) + \varphi_2(t_{n+1}, \tilde{q}, \tilde{I})\Big) \end{cases}$$

```
def Heun(phi1,phi2,tt,q_0,i_0):
    h = tt[1]-tt[0]
    uu1 = [q_0]
    uu2 = [i_0]
    for i in range(len(tt)-1):
        uu1pred = uu1[i] + h*phi1(tt[i],uu1[i],uu2[i])
        uu2pred = uu2[i] + h*phi2(tt[i],uu1[i],uu2[i])
```

```
        uu1.append( uu1[i] + h/2*( phi1(tt[i],uu1[i],uu2[i])+phi1(tt[i],uu1pred,uu2pred) ) )
        uu2.append( uu2[i] + h/2*( phi2(tt[i],uu1[i],uu2[i])+phi2(tt[i],uu1pred,uu2pred) ) )
    return [uu1,uu2]
```

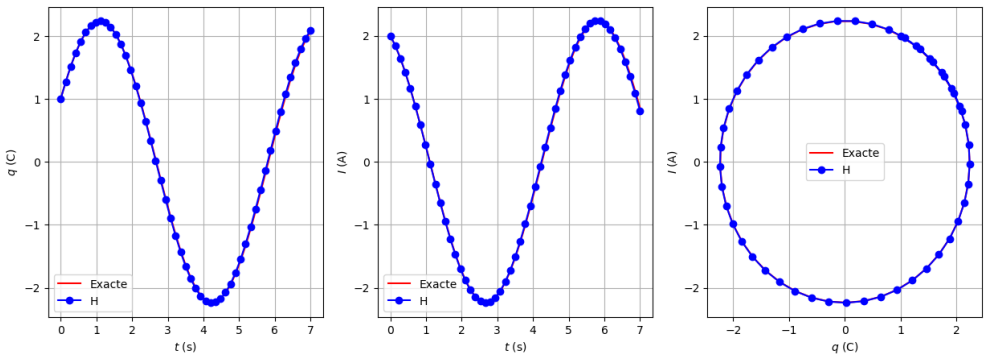
```
uu1H,uu2H = Heun(phi1,phi2,tt,q_0,i_0)

plt.figure(1,figsize=(15,5))

plt.subplot(1,3,1)
plt.plot(tt,yy1,'r-',label=('Exacte'))
plt.plot(tt,uu1H,'b-o',label=('H'))
plt.xlabel('$t$ (s)')
plt.ylabel('$q$ (C)')
plt.legend()
plt.grid(True)

plt.subplot(1,3,2)
plt.plot(tt,yy2,'r-',label=('Exacte'))
plt.plot(tt,uu2H,'b-o',label=('H'))
plt.xlabel('$t$ (s)')
plt.ylabel('$I$ (A)')
plt.legend()
plt.grid(True)

plt.subplot(1,3,3)
plt.plot(yy1,yy2,'r-',label=('Exacte'))
plt.plot(uu1H,uu2H,'b-o',label=('H'))
plt.xlabel('$q$ (C)')
plt.ylabel('$I$ (A)')
plt.legend()
plt.grid(True)
```



Correction Q4

```
# Euler-Cromer, see fdm-book-4print.pdf page 47
# The scheme goes under several names: forward-backward scheme, semi-implicit Euler method,
# semi-explicit Euler, symplectic Euler, Newton-Störmer-Verlet, and Euler-Cromer.
def EC(phi1,phi2,tt,q_0,i_0):
    h = tt[1]-tt[0]
    uu1 = [q_0]
    uu2 = [i_0]
    for i in range(len(tt)-1):
        uu2.append( uu2[i] + h*phi2(tt[i],uu1[i],uu2[i]) )
        uu1.append( uu1[i] + h*phi1(tt[i+1],uu1[i],uu2[i+1]) ) # en theorie uu1[i+1] mais n'intervient
↪   pas dans phi1
    return [uu1,uu2]
```

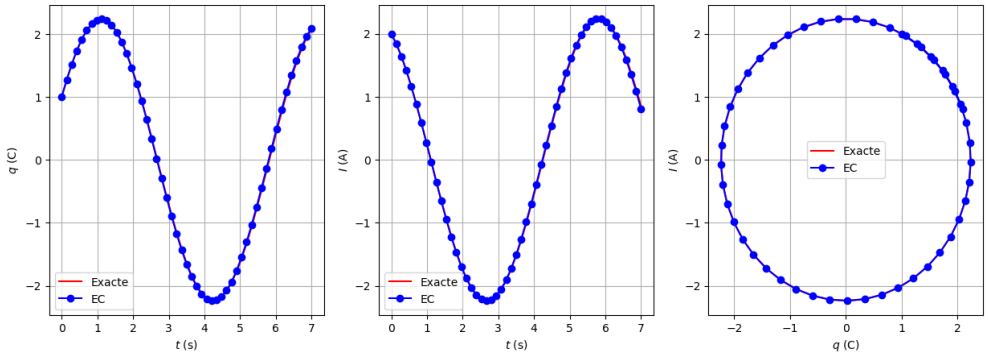
```
uu1EC,uu2EC = Heun(phi1,phi2,tt,q_0,i_0)

plt.figure(1,figsize=(15,5))

plt.subplot(1,3,1)
plt.plot(tt,yy1,'r-',label=('Exacte'))
plt.plot(tt,uu1EC,'b-o',label=('EC'))
plt.xlabel('$t$ (s)')
plt.ylabel('$q$ (C)')
plt.legend()
plt.grid(True)

plt.subplot(1,3,2)
plt.plot(tt,yy2,'r-',label=('Exacte'))
plt.plot(tt,uu2EC,'b-o',label=('EC'))
plt.xlabel('$t$ (s)')
plt.ylabel('$I$ (A)')
plt.legend()
plt.grid(True)

plt.subplot(1,3,3)
plt.plot(yy1,yy2,'r-',label=('Exacte'))
plt.plot(uu1EC,uu2EC,'b-o',label=('EC'))
plt.xlabel('$q$ (C)')
plt.ylabel('$I$ (A)')
plt.legend()
plt.grid(True)
```



Sixième partie

Downloads

Chapitre 40.

Notebook Originaux à télécharger

Dans cette partie, vous trouverez les notebooks originaux utilisés dans le cours et les exercices. Ils sont disponibles en téléchargement pour que vous puissiez les exécuter sur votre propre machine.

Pour **télécharger un notebook**, cliquez sur le bouton `Download` correspondant ci-dessous.

40.1. Cours

- **Introduction à l'approximation numérique d'EDO**
  - Solution exacte, solution exacte discrète, solution approchée.
  - Schémas d'Euler explicite (EE) et implicite (EI) : construction et implémentation.
  - Erreur de troncature, convergence.
  - Étude théorique et empirique de l'ordre de convergence des schéma EE et EI.
- **Schémas classiques à un pas**

  Construction, par interpolation-intégration, des schémas suivants :

  - Euler explicite (EE)
  - Euler implicite (EI)
  - Crank-Nicolson (CN)
  - Heun (H) [exemple de schéma predictor-corrector]
  - Euler modifié (EM) [exemple de schéma de type Runge-Kutta]
  - Runge-Kutta RK1_M [exemple de schéma de type Runge-Kutta]
- **Propriétés d'un problème de Cauchy**
  - Problèmes mal/bien posés mathématiquement,
  - Problèmes mal/bien posés numériquement,
  - Problèmes mal/bien conditionné (problèmes raides)
- **Construction de schémas multipas**
  - Schémas d'Adam (AB et AM)
  - Schémas de Nyström et Milne-Simpson
  - Schémas BDF
  - Construction de schémas predictor-corrector
- **Propriétés d'un schéma sur un intervalle borné : consistance, zéro-stabilité, convergence**
  - Consistance, zéro-stabilité, convergence, ordre de convergence
- **Propriétés d'un schéma sur un intervalle non borné : A-stabilité**
  - A-stabilité
  - Étude théorique de la A-stabilité des schémas EE, EI, EM, CN, Heun, AB2, BDF2
- **Étude de schémas multipas linéaires**
  - Premier polynôme caractéristique et zéro-stabilité (condition des racines)
  - Consistance
  - Convergence et ordre de convergence (conditions et première barrière de Dahlquist)
  - A-stabilité
  - Exemple d'étude avec les schémas EE et EI.
- **Construction et étude de schémas de Runge-Kutta**

- Construction et matrice de Butcher
- Ordre de convergence (conditions jusqu'à 4 + barrières de Butcher)
- A-stabilité
- Étude des schémas à 1 étage : construction, ordre et A-stabilité
- Étude des schémas à 2 étages : construction (tous), ordre (explicites) et A-stabilité (explicites)
- Exemples de schémas à 3, 4, 5, 6 étages explicites

40.2. Travaux Pratiques (énoncés)

- **TP1** – Calcul exact avec sympy et calcul approché avec scipy
- **TP2** – Implémentation des schémas classiques (EE, EI, EM, RK1_M, CN, Heun) et étude empirique de la convergence
- **TP3** – Implémentation des schémas multipas, predictor-corrector et Runge-Kutta (explicites et implicites)
- **TP4** – Étude empirique de la convergence des schémas du TP3 (attention : initialisation par la solution exacte)
- **TP5** – Devoir Maison 2020

40.3. Travaux Pratiques (avec corrections)

- **TP1 - partie A** – Calcul exact avec sympy
- **TP1 - partie B** – Calcul approché avec scipy
- **TP2** – Implémentation des schémas classiques (EE, EI, EM, RK1_M, CN, Heun) et étude empirique de la convergence
- **TP3** – Implémentation des schémas multipas, predictor-corrector et Runge-Kutta (explicites et implicites)
- **TP4** – Étude empirique de la convergence des schémas du TP3 (attention : initialisation par la solution exacte)

40.4. Travaux Dirigés (avec corrections)

- **TD1**
  - Schémas à un pas : construction, étude de la consistance, de la A-stabilité, de l'ordre de convergence, implémentation et étude empirique de la convergence.
- **TD2**
  - Schémas multi-pas linéaires : aide-mémoire (consistance, zéro stabilité, ordre de convergence ; A-stabilité si 1 ou 2 pas) et exercices avec paramètres.
  - Schémas prédicteur-correcteur basés sur des schémas multi-pas.
- **TD3**
  - Schémas RK : aide-mémoire (matrice de Butcher, consistance, ordre de convergence (jusqu'à 4) ; A-stabilité).
- **TD4**
  - Étude de problèmes numériquement mal posés ou de problèmes raides.
  - A-stabilité pour un système.
  - Schémas pour un système ou une EDO d'ordre supérieur à 1.

40.5. Annales (avec corrections)

Dans cette partie, vous trouverez les notebooks des CC, CT session 1 et CT session 2 depuis 2019.

40.5.1. A.A. 2025

- 2025 CC (2 versions de chaque exercice)
  - Étude d'un système d'EDO avec le schéma de Heun ou d'Euler Modifié. Étude empirique de l'ordre de convergence à l'aide d'un invariant.
  - Étude théorique de la convergence d'un schéma à 3 pas linéaire.
  - Étude d'un schéma RK à 2 étages semi-implicite (étude théorique et empirique de l'ordre de convergence)
- 2025 CT session 1
  - Détermination d'un schéma linéaire explicite à 2 pas d'ordre 2 (calcul des coefficients, étude empirique de l'ordre de convergence).
  - Étude de la A-stabilité d'un schéma RK à 3 étages explicite (étude théorique de la A-stabilité, vérification empirique).
  - Implementation et étude empirique de l'ordre de convergence d'un schéma predictor-corrector.
- 2025 CT session 2
  - Étude de l'ordre et de la A-stabilité d'un schéma RK à 3 étages explicite (étude théorique de l'ordre et de la A-stabilité, vérification empirique de l'ordre).
  - Détermination d'un schéma linéaire explicite à 3 pas d'ordre 2 (calcul des coefficients, étude empirique de l'ordre de convergence).

40.5.2. A.A. 2024

- 2024 CC
  - Choix d'un schéma.
  - Étude d'un schéma RK à 4 étages explicite avec paramètres (étude théorique de l'ordre de convergence et de la A-stabilité, étude empirique de l'ordre de convergence)
  - Étude d'un schéma à 2 pas linéaire (étude théorique de la consistance, la zéro-stabilité et l'ordre de convergence ; étude empirique de l'ordre de convergence).
- 2024 CT session 1
  - Choix d'un schéma.
  - Étude d'un schéma à 3 pas linéaire (montrer que s'il est consistant d'ordre 6 alors il n'est pas zéro-stable, étude théorique de l'ordre de convergence, calcul des paramètrès pour avoir l'ordre maximal dans le cas explicite ; étude empirique de l'ordre de convergence).
  - Étude d'un schéma RK à 2 étages implicite avec paramètres (étude théorique de l'ordre de convergence en fonction du paramètre, étude empirique de l'ordre de convergence, étude de la A-stabilité)
- 2024 CT session 2
  - Étude d'un système d'EDO avec le schéma de Heun.
  - Étude d'un schéma à 2 pas linéaire avec deux paramètres (étude théorique de la consistance, la zéro-stabilité et l'ordre de convergence ; étude empirique de l'ordre de convergence).
  - Étude d'un schéma RK à 3 étages explicite avec un paramètre (étude théorique de l'ordre de convergence, étude empirique de l'ordre de convergence, étude de la A-stabilité)

40.5.3. A.A. 2023

- 2023 CC
  - Choix d'un schéma.
  - Étude d'un schéma multipas à 2 pas (étude théorique de la consistance, la zéro-stabilité et l'ordre de convergence ; étude empirique de l'ordre de convergence).
  - Étude d'un schéma RK à deux étages semi-implicite avec paramètres (étude théorique de l'ordre de convergence et de la A-stabilité, étude empirique de l'ordre de convergence)
- 2023 CT session 1
  - Résolution d'un système avec les schémas EE et CN et avec odeint (scipy).
  - Étude d'un schéma multipas à 2 implicite (étude théorique de la consistance, la zéro-stabilité et l'ordre de convergence ; étude empirique de l'ordre de convergence).

- Étude d'un schéma RK à 3 étages explicite avec paramètres (étude théorique de l'ordre de convergence et de la A-stabilité, étude empirique de l'ordre de convergence)
- 2023 CT session 2
  - Choix d'un schéma pour un problème de Cauchy numériquement mal posé.
  - Étude d'un schéma RK à 3 étages semi-implicite (étude théorique de l'ordre de convergence et de la A-stabilité, étude empirique de l'ordre de convergence)

40.5.4. A.A. 2022

- 2022 CC
  - Étude d'un schéma multipas à 2 pas avec paramètres (étude théorique de la consistance, la zéro-stabilité et l'ordre de convergence ; étude empirique de l'ordre de convergence).
  - Résolution d'un système avec les schémas EM et CN, puis avec odeint (scipy) et dsolve (sympy).
  - Étude de la A-stabilité d'un schéma RK (étude théorique, étude empirique sur un problème stiff)
- 2022 CT session 1
  - Étude du schéma d'Euler exponentiel (étude empirique de l'ordre de convergence).
  - Étude d'un schéma RK à 3 étages implicite (étude théorique de l'ordre de convergence et de la A-stabilité ; étude empirique de l'ordre de convergence et de la A-stabilité)
  - Résolution d'un système Hamiltonien avec dsolve (sympy) et approximation par le schéma d'Euler symplectique.
- 2022 CT session 2
  - Étude du schéma d'Euler exponentiel pour un problème stiff linéaire (la solution discrète est exacte).
  - Étude d'un schéma RK à 4 étages semi-implicite (étude empirique de l'ordre de convergence ; étude théorique de l'ordre de convergence)
  - Étude d'un schéma multipas à 3 pas (étude théorique de la consistance, la zéro-stabilité et l'ordre de convergence ; étude empirique de l'ordre de convergence).

40.5.5. A.A. 2021

- 2021 CC
  - Résolution d'un système avec les schémas EE et EI et avec odeint (scipy).
  - Étude d'un schéma multipas à 2 pas avec paramètres (étude théorique de la consistance, la zéro-stabilité et l'ordre de convergence ; étude empirique de l'ordre de convergence).
- 2021 CT session 1
  - Étude d'un schéma RK à 2 étages explicite avec paramètres (étude théorique de l'ordre de convergence et de la A-stabilité ; étude empirique de l'ordre de convergence et de la A-stabilité).
  - Choix d'un schéma pour un problème raide et implémentation
  - Choix d'un schéma pour un problème numériquement mal posé et implémentation
- 2021 CT session 2
  - Étude d'un schéma RK à 3 étages explicite avec paramètres (étude théorique de l'ordre et de la A-stabilité, étude empirique de l'ordre et de la A-stabilité)
  - Étude d'un schéma multipas à 2 pas avec paramètres (étude théorique de la consistance, de la zéro-stabilité et de l'ordre de convergence, étude empirique de l'ordre)

40.5.6. A.A. 2020

- 2020 CC (DM)
  - Résolution d'une EDO d'ordre 2 (donc d'un système) avec les schémas EE et EI, avec odeint (scipy) et avec dsolve (sympy).
  - Étude d'un schéma RK à 2 étages semi-implicite (étude théorique de l'ordre de convergence et de la A-stabilité ; étude empirique de l'ordre de convergence et de la A-stabilité).
  - Étude d'un schéma Predictor-Corrector :

- étude du predictor : schéma multipas à 3 pas (étude théorique de la consistance, de la zéro-stabilité et de l'ordre de convergence, étude empirique de l'ordre)
- étude du corrector : schéma multipas à 2 pas (étude théorique de la consistance, de la zéro-stabilité et de l'ordre de convergence, étude empirique de l'ordre)
- étude empirique de l'ordre du schéma predictor-corrector.

40.5.7. A.A. 2019

- 2019 CC
  - Étude d'un schéma RK à 2 étages explicite avec paramètres (étude théorique de l'ordre de convergence et de la A-stabilité ; étude empirique de l'ordre de convergence et de la A-stabilité).
- 2019 CT session 1
  - Choix d'un schéma pour un problème numériquement mal posé
  - Choix d'un schéma pour un problème raide
  - Calcul de la solution exacte d'un problème de Cauchy
  - Étude empirique de la convergence d'un schéma Predictor-Corrector
  - Étude d'un schéma RK à 2 étages semi-implicite (ordre théorique, implémentation)
- 2019 CT session 2
  - Choix d'un schéma (après calcul de la solution exacte on vérifie qu'il s'agit d'un problème numériquement mal posé)
  - Résolution d'un système (circuit LC) avec les schémas EE, Heun, Euler-Cromer et avec dsolve (sympy).