GM3 - 2022/2023

SCHÉMAS DE DISCRÉTISATION À UN PAS

Omar CHOUKRANI

À l'attention de : Hasnaa ZIDANI

Table des matières

1 Introduction

Le but de ce TP est de résoudre numériquement des équations différentielles :

$$\begin{cases} x'(t) = f(t, x(t)) \quad \text{pour} \quad t \in [0, T] \\ x(0) = x_0 \end{cases}$$

avec $x_0 \in \mathbb{R}^d$ fixé et la fonction $f : [0, T] \times \mathbb{R}^d \longrightarrow \mathbb{R}^d$ une fonction Lipschitz. La solution exacte de cette équation sera notée $X(\cdot)$.

On va s'intéresser à trois schémas de déscritisation. Pour cela on définit d'abord une déscritisation de l'intervalle de temps :

$$0 = x_0 < t_1 < \cdots < t_N \text{ pour } N \geq 1$$

Ici, on va choisir une discrétisation uniforme avec un pas h défini par :

$$h = \frac{T}{N}$$

Dans la suite on notera `temps` le vecteur de taille $N+1$ qui contient les t_i pour $i = 0, \cdots, N$.

On va aussi noter y le tableau qui contient les approximations de X. En particulier, la i-ème colonne $y_i = y(:, i)$ désigne une approximation de $X(t_i)$:

$$y_i \approx X(t_i) \text{ pour } i = 0, \cdots, N$$

Pour chaque $i \in \{0, \cdots, N\}$, l'approximation y_i est un vecteur de $\mathbb{R}^d$ qui sera calculé par un schéma de discrétisation à un pas.

Schéma d'Euler Explicite (EE):

(EE) | On définit $y_0 = x_0$.
Pour $i \in \{0, \cdots, N-1\}$, on calcule $y_{i+1} = y_i + hf(t_i, y_i)$.

Schéma d'Euler Implicite (EI):

(EE) | On pose $y_0 = x_0$.
Pour $i \in \{0, \cdots, N-1\}$, on calcule $y_{i+1} = y_i + hf(t_i, y_{i+1})$.

Schéma du Point Milieu (PM):

(EE) | On pose $y_0 = x_0$.
Pour $i \in \{0, \cdots, N-1\}$, on calcule $y_{i+1} = y_i + hf(t_i + \frac{h}{2}, y_i + \frac{h}{2} f(t_i, y_i))$.

2 Implémentation des méthodes (EE) et (PM)

2.1 `euler_explicite`

Cette fonction prend en paramètre f la dynamique de l'équation différentielle, y_0 le vecteur initial et `temps` le vecteur contenant les temps où la solution est calculée. Elle retourne la solution numérique y à l'aide de la méthode de Euler Explicite.

```
def euler_explicite(f, y0, temps):
    d = len(y0)
    y = np.zeros((d, len(temps)))
    y[:,0] = y0
    for i in range(1, len(temps)):
        dt = temps[i] - temps[i-1]
        y[:,i] = y[:,i-1] + dt*f(temps[i-1],y[:,i-1])
    return y
```

2.2 `point_milieu`

Cette fonction prend en paramètre f la dynamique de l'équation différentielle, y_0 le vecteur initial et `temps` le vecteur contenant les temps où la solution est calculée. Elle retourne la solution numérique y à l'aide du de la méthode du Point Milieu.

```
def point_milieu(f, y0, temps):
    d = len(y0)
    y = np.zeros((d, len(temps)))
    y[:,0] = y0
    for i in range(1, len(temps)):
        dt = temps[i] - temps[i-1]
        y[:,i] = y[:,i-1] + dt*f(temps[i-1]+dt/2,y[:,i-1]
        + dt/2*f(temps[i-1],y[:,i-1]))
    return y
```

3 Exemple d'application

3.1 (EE) vs (PM)

Considérons l'équation différentielle suivante :

$$\begin{cases} x'(t) = f(t, x(t)) \quad \text{pour} \quad t \in [0, 2] \\ x(0) = 1/2 \end{cases}$$

Avec

$$f(t, x) = x - t^2 + 1$$

Remarquons que cette dynamique est continue (car polynômiale) et localement Lipschitzienne par rapport à la seconde variable. L'équation admet donc, d'après le théorème de Cauchy-Lipschitz, une unique solution maximale notée X. De plus, on a :

$$\forall t \geq 0, \; X(t) = (t+1)^2 - \frac{e^t}{2}$$

En effet, on a pour tout $t \geq 0$:

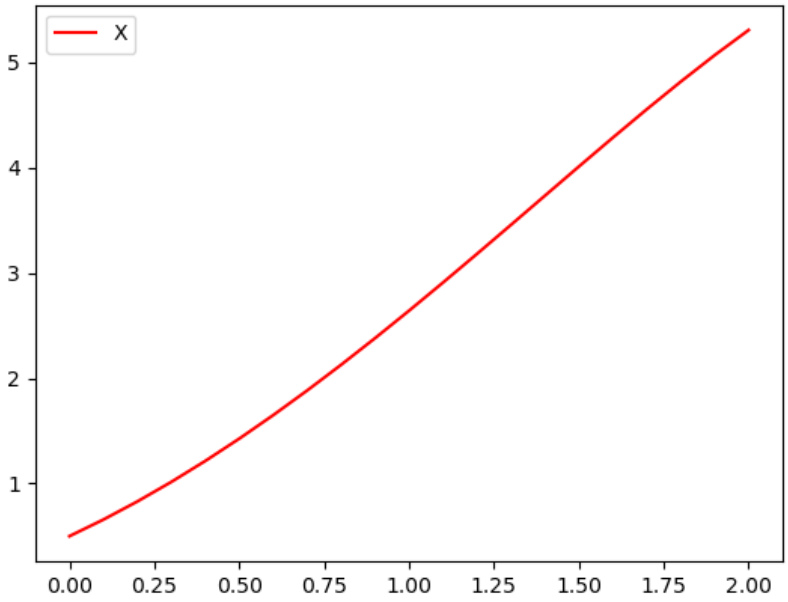
$$\begin{aligned} X'(t) &= 2t + 2 - \frac{e^t}{2} \\ &= t^2 + 2t + 1 - t^2 + 1 - \frac{e^t}{2} \\ &= (t+1)^2 - t^2 + 1 - \frac{e^t}{2} \\ &= X(t) - t^2 + 1 \\ &= f(t, X(t)) \end{aligned}$$

De plus :

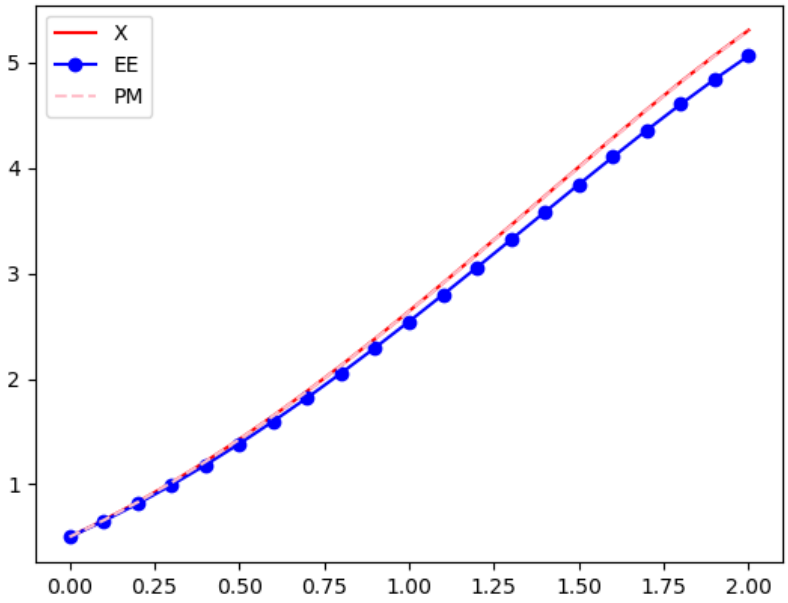
$$X(0) = 1 - \frac{1}{2} = \frac{1}{2}$$

Afin de tester les méthodes (EE) et (PM) sur l'exemple ci-dessus, on a tracé la courbe de X à l'aide des bibliothèques `numpy` et `matplotlib` :

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

def solution1(t):
  return (t+1)*(t+1) - np.exp(t) / 2

temps = np.linspace(0, 2, 20)
y_ex = solution1(temps)

plt.plot(temps, y_ex, label="X")

plt.legend()
plt.show()
```



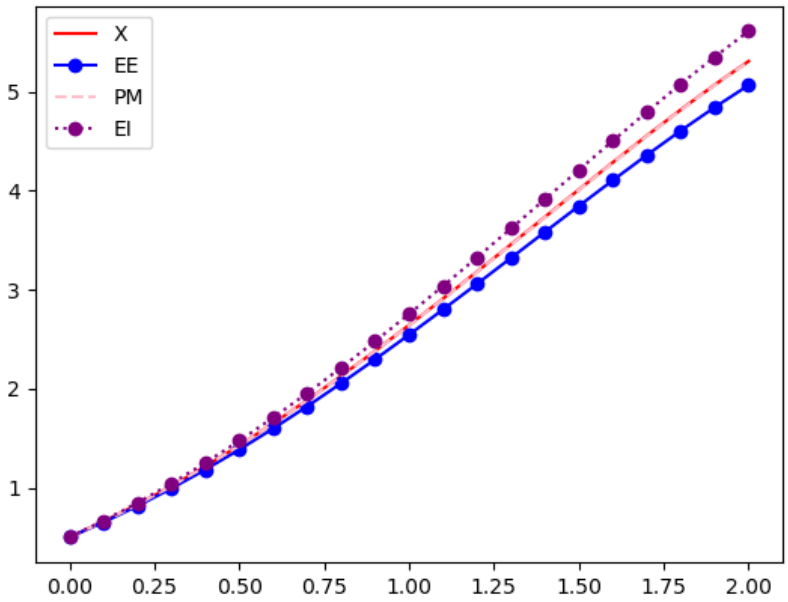
Ensuite, on a tracé les courbes des solutions numériques :



On voit bien que les solutions numériques sont relativement “proches” de la soltion exacte.

3.2 Implémentation de la méthode de Euler Implicite

```
d = 1
y_ei = np.zeros((d, len(temps)))
y_ei[:,0] = 1/2
for i in range(1, len(temps)):
  dt = temps[i] - temps[i-1]
  y_ei[:,i] = (1/(1-dt))*(y_ei[:,i-1]+dt*(1-temps[i]*temps[i]))
```



3.3 `euler_implicite`

Notons qu'on peut implémenter cette méthode pour n'importe quelle fonction à l'aide du module `scipy.optimize` qui permet de résoudre l'équation suivante à chaque itération :

$$x = y_{i-1} + hf(t_{i-1}, x)$$

Ceci en utilisant la fonction `fsolve` qui permet de résoudre numériquement des équations non linéaires en utilisant des méthodes d'optimisation. Cette dernière prend en entrée deux arguments principaux : une fonction qui représente l'équation à résoudre et une estimation initiale de la solution (y_{i-1} dans notre cas). Voici un exemple de programmation de la fonction `euler_implicite` :

```
def euler_implicite(f, y0, temps):
    d = len(y0)
    y = np.zeros((d, len(temps)))
    y[:,0] = y0
    for i in range(1, len(temps)):
        dt = temps[i] - temps[i-1]
        y[:,i] = fsolve(lambda x: x-y[:,i-1]-dt*fonction1(temps[i],x),
   y[:,i-1])
    return y
```

4 (EE) vs (EI) vs (PM)

Dans cette partie, on va comparer les trois méthodes. Pour ce faire, on va effectuer des calculs pour $N = 20, 40, 80, 160, 320, 1600$ pour chacune des trois méthodes. On notera :

- $E_c(N) = \sum_{i=0}^{N} |y_i - X(t_i)|$ l'erreur cumulée
- $E_s(N) = \max_i |y_i - X(t_i)|$ l'écart maximal

4.1 Euler Explicite

- $E_c(20) = 1.9903665947425253$
- $E_c(40) = 2.133268889830414$
- $E_c(80) = 2.210410989493962$
- $E_c(160) = 2.2505487325870224$
- $E_c(320) = 2.2710291951593287$
- $E_c(640) = 2.2813749471838234$

- $E_s(20) = 0.24197192013003654$
- $E_s(40) = 0.12746574220323126$
- $E_s(80) = 0.06549505405515976$
- $E_s(160) = 0.03320765415716842$
- $E_s(320) = 0.016721431162730838$
- $E_s(640) = 0.00839044634582109$

On remarque que l'erreur cumulée augmente avec N, ce qui semble un peu contre-intuitif alors que c'est tout à fait logique puisque l'on augmente le nombre de points d'évaluation et donc les petites erreurs absolues contribuant à la somme.

4.2 Euler Implicite

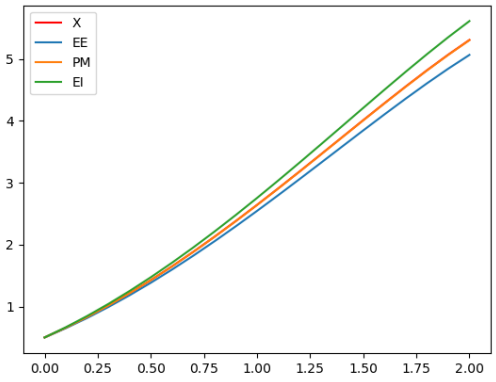
- $E_c(20) = 2.3636850773192206$
- $E_c(40) = 2.3259360052732347$
- $E_c(80) = 2.308448279538847$
- $E_c(160) = 2.30002021505833$
- $E_c(320) = 2.2958816297868117$
- $E_c(640) = 2.29383078144358$

- $E_s(20) = 0.30442271347734096$
- $E_s(40) = 0.1429137895565109$
- $E_s(80) = 0.06934689890669521$
- $E_s(160) = 0.03416998184312181$
- $E_s(320) = 0.016961973518519358$
- $E_s(640) = 0.0084505794623837$

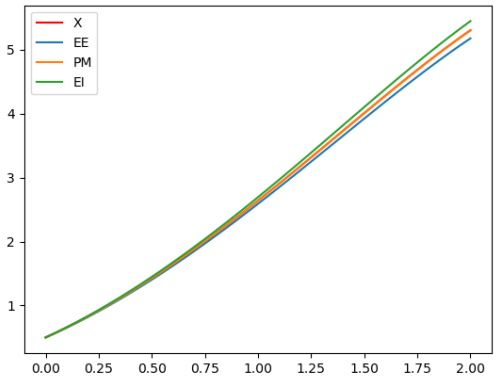
4.3 Point Milieu

- $E_c(20) = 0.03751092704981718$
- $E_c(40) = 0.019359177056069043$
- $E_c(80) = 0.009820799462889829$
- $E_c(160) = 0.004944303526922655$
- $E_c(320) = 0.002480440668269801$
- $E_c(640) = 0.0012422683979776972$

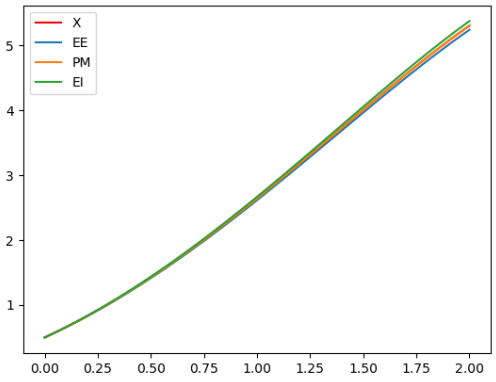
- $E_s(20) = 0.003747073502069931$
- $E_s(40) = 0.0009277142152619433$
- $E_s(80) = 0.0002304036640010665$
- $E_s(160) = 5.738409553845969e-05$
- $E_s(320) = 1.431723125655537e-05$
- $E_s(640) = 3.575601474459233e-06$



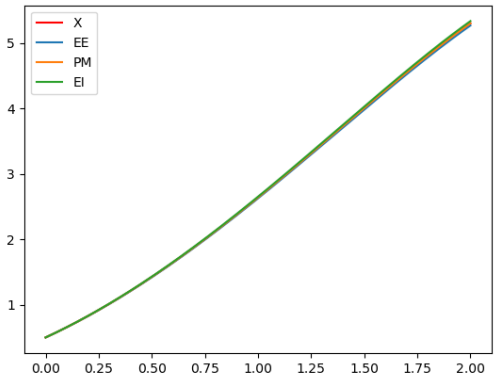
N=20



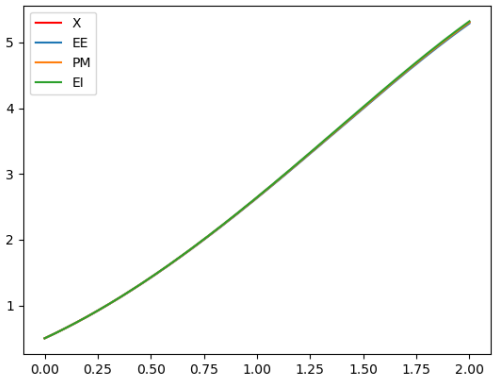
N=40



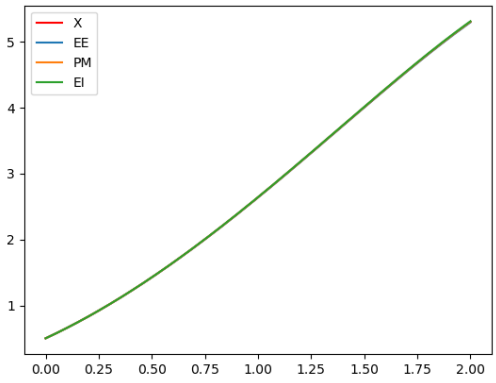
N=80



N=160



N=320

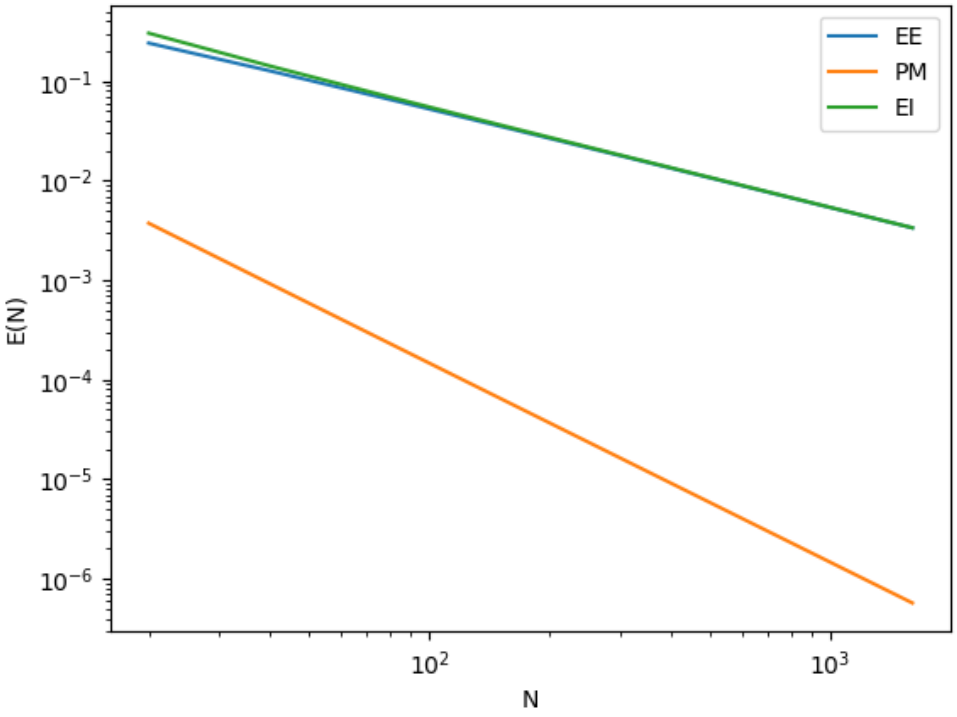


N=640

4.4 L'erreur E(N)

Pour donner une idée sur l'efficacité de trois méthodes, on va représenter l'erreur E en fonction de N, où :

$$E(N) = \sup_{0 \leq i \leq N} |y_i - X(t_i)|$$



4.5 L'ordre numérique

L'erreur est de la forme :

$$E(N) = ch^p$$

Avec $c > 0$, $h = \frac{T}{N}$ et p l'ordre numérique que nous voulons déterminer.

Puisque E est inversement proportionnel a h, on a :

$$E(2N) = c\left(\frac{h}{2}\right)^p$$

D'où :

$$p = \frac{\ln\left(\frac{E(N)}{E(2N)}\right)}{\ln(2)}$$

Or, d'après les valeurs enrigistrées des erreurs, on remarque que pour les méthodes d'Euler (EE et EI), p convergeait vers 1. En revanche, l'ordre de la méthode du point milieu convergerait vers 2. Ce qui est cohérant avec le fait que plus l'ordre est élevé, plus la méthode est précise.

5 Modèle SIR

On considère ici le système différentiel suivant :

$$\begin{cases} S'(t) = -bS(t)I(t), \\ I'(t) = bS(t)I(t) - gI(t), \\ I(0) = I_0 > 0, \quad S(0) = S_0 > 0. \end{cases}$$

On peut mq la solution $\{(S(t), I(t)) \mid t > 0\}$ est contenue dans la courbe paramétrée :

$$x_1 + x_2 - \frac{g}{b}\ln(x_1) = c_0, \quad x = (x_1, x_2) \in \mathbb{R}^2$$
$$c_0 := I(0) + S(0) - \frac{g}{b}\ln(S(0))$$

Pour les tests numériques, on va prendre :

$$S_0 = 0.9, \quad I_0 = 0.1, \quad R_0 = 0, \quad t_0 = 0, \quad T = 100.$$

5.1 La dynamique `fSIR`

Commençons par écrire le système sous sa forme "canonique". Remarquons que :

$$X'(t) = \begin{pmatrix} S'(t) \\ I'(t) \end{pmatrix} = \begin{pmatrix} -bS(t)I(t) \\ bS(t)I(t) - gI(t) \end{pmatrix} = f(t, X(t))$$

Avec :

$$\begin{aligned} f : \quad \mathbb{R} \times \mathbb{R}^2 &\longrightarrow \mathbb{R}^2 \\ (t, (x_1(t), x_2(t))) &\longmapsto \begin{pmatrix} -bx_1(t)x_2(t) \\ bx_1(t)x_2(t) - gx_2(t) \end{pmatrix} \end{aligned}$$

On peut ainsi exprimer le système comme suit :

$$\begin{cases} (S'(t), I'(t)) = f(t, (S(t), I(t)), \\ I(0) = I_0 > 0, \quad S(0) = S_0 > 0. \end{cases}$$

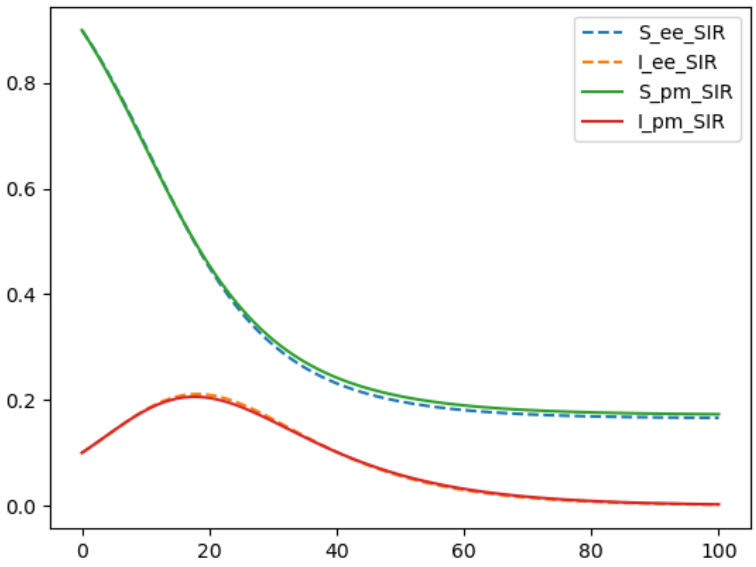
Voici un exemple d'implémentation de cette dynamique :

```
def fSIR(t, x):
    x1, x2 = x[0], x[1]
    return np.array([-b*x1*x2, b*x1*x2-g*x2])
```

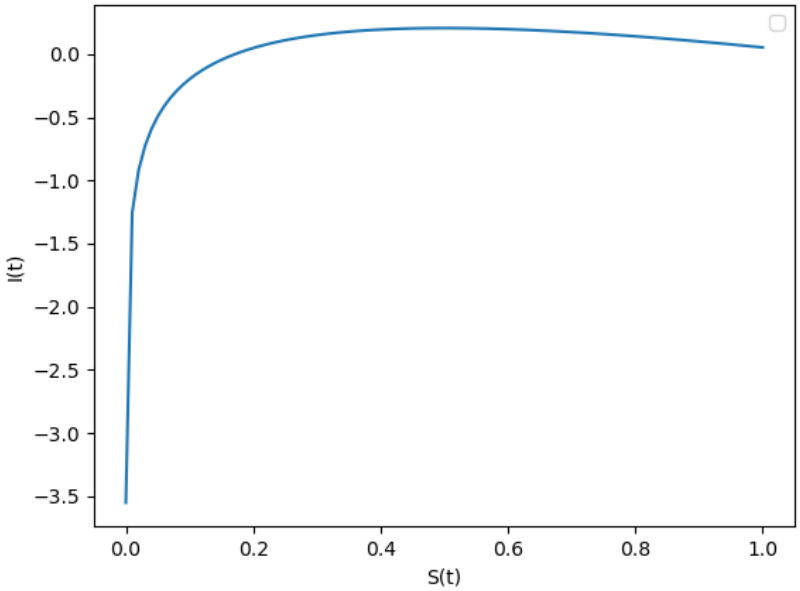
5.2 Résolution numérique

Pour une descritisation de $N = 101$ points et des valeurs de $b = 0.2$ et $g = 0.1$, on a :

- Les solutions numériques :



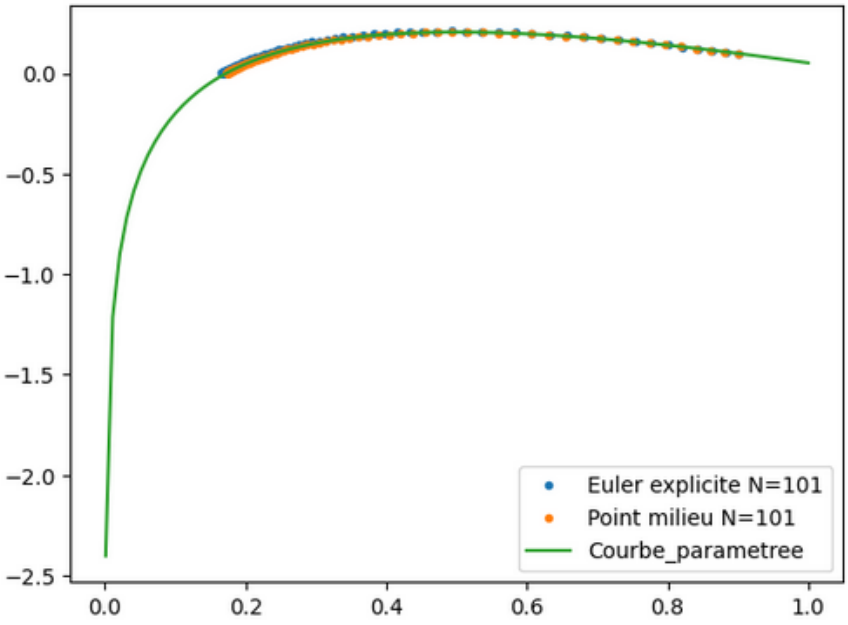
- La courbe paramétrée correspondante à la solution exacte :



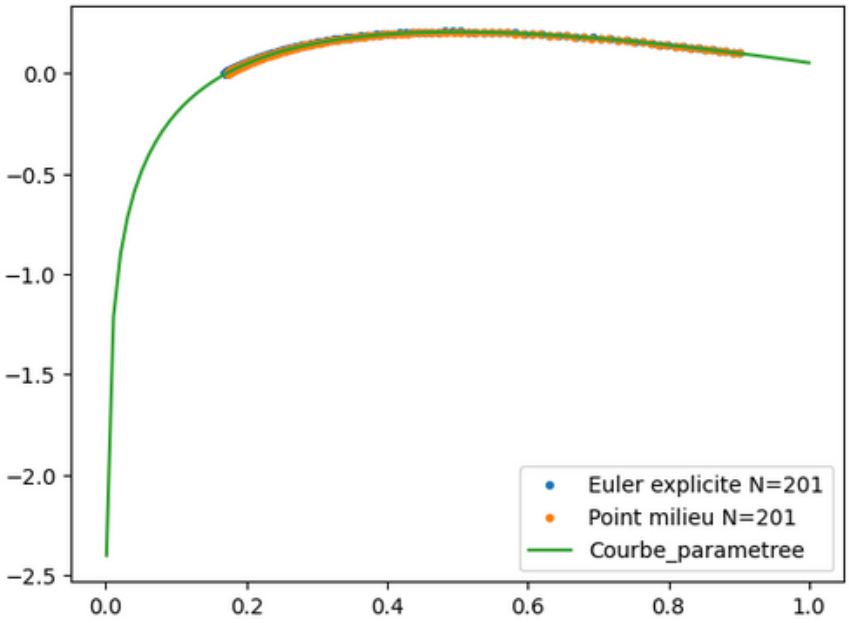
Traçons maintenant la courbe des points $\{(y_i(1), y_i(2)), i = 0, \cdots, N\}$ pour différents N :

```
for N in [100, 200, 1000]:

    temps_N = np.linspace(0, T, N+1)

    X_ee_SIR = euler_explicite(fSIR, X0, temps_N)
    X_pm_SIR = point_milieu(fSIR, X0, temps_N)

    plt.plot(X_ee_SIR[0,:], X_ee_SIR[1,:],".", label=f'Euler explicite N={N+1}')
    plt.plot(X_pm_SIR[0,:], X_pm_SIR[1,:],".", label=f'Point milieu N={N+1}')
    plt.plot(S_SIR, I_SIR, label="Courbe_parametree")

    plt.legend()
    plt.show()
```

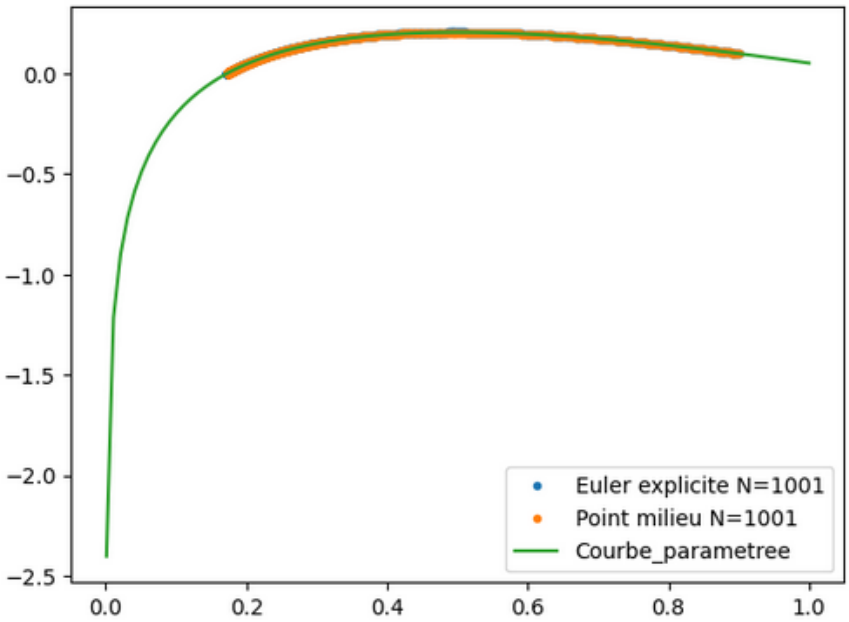
- Pour $N = 101$



• Pour $N = 201$



• Pour $N = 1001$



On en conclut que plus N est grand, plus le résultat est pertinent.

6 Ordre élevé

Dans la suite, on s'intéresse aux schémas de type Runge-Kutta suivants :

Schémas Runge-Kutta (RK2$_\alpha$):

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \alpha & \alpha & 0 \\ \hline & 1-\frac{1}{2\alpha} & \frac{1}{2\alpha} \end{array} \quad \text{pour} \quad \alpha \in [0,1].$$

Schémas Runge-Kutta (RK4):

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{2}{6} & \frac{2}{6} & \frac{1}{6} \end{array}$$

6.1 Implémentation des méthodes (EE) et (PM)

6.1.1 (RK2$_\alpha$) et (RK4)

Montrons que (RK2) est d'ordre 2 et (RK4) est d'ordre 4.

On a $f \in \mathcal{C}^2$ avec la continuité des partielles de F.

On a aussi :

$F(t,y,h) = \sum_{j=1}^{2} c_j f(t+\theta_j h, y_{ij}(t,y,h))$

$F(t,y,0) = \sum_{j=1}^{2} c_j f(t,y) = (1 - 1/2\alpha + 1/2\alpha) f(t,y) = f(t,y)$

$\frac{\partial F}{\partial h}(t,y,0) = \sum_{j=1}^{2} c_j \theta_j f'(t,y) = 0 \times (1 - 1/2\alpha) + \alpha \times 1/2\alpha = \frac{1}{2} f'(t,y)$

D'où le résultat.

De même pour (RKR4), on montre qu'elle est d'ordre 4.

6.1.2 $RK2_1$

```
def RK2(f, y0, temps):

    d = len(y0)
    y = np.zeros((d, len(temps)))
    y[:,0] = y0

    for i in range(1, len(temps)):

        dt = temps[i] - temps[i-1]

        S1 = temps[i-1]
        K1 = f(S1, y[:,i-1])

        S2 = temps[i]
        K2 = f(S2, y[:,i-1] + dt*K1)

        y[:,i] = y[:,i-1] + 0.5 * dt * (K1 + K2)

    return y
```

6.1.3 RK4

```
def RK4(f, y0, temps):

    d = len(y0)
    y = np.zeros((d, len(temps)))
    y[:, 0] = y0

    for i in range(1, len(temps)):

        dt = temps[i] - temps[i-1]
        t = temps[i-1]

        k1 = dt * f(y[:, i-1], t)
        k2 = dt * f(t + 0.5 * dt, y[:, i-1] + 0.5 * k1)
        k3 = dt * f(t + 0.5 * dt, y[:, i-1] + 0.5 * k2)
        k4 = dt * f(t + dt, y[:, i-1] + k3)

        y[:, i] = y[:, i-1] + (1/6) * (k1 + 2*k2 + 2*k3 + k4)

    return y
```

6.2 Modèle linéaire

On va s'intéresser dans un premier temps à un problème linéaire :

$$\begin{cases} x'(t) = Mx(t) \quad \text{pour} \quad t \in [0,T] \\ x(t_0) = x_0 \end{cases}$$

où M est une matrice de $\mathcal{M}_d(\mathbb{R})$.

6.2.1 `euler_implicite(M, x0, temps)`

```
def euler_implicite(M, x0, temps):

    d = len(x0)
    x = np.zeros((d, len(temps)))
    x[:, 0] = x0

    for i in range(1, len(temps)):
        dt = temps[i] - temps[i-1]
        t = temps[i]
        x[:, i] = np.linalg.solve(np.eye(d) - dt * M, x[:, i-1])

    return x
```

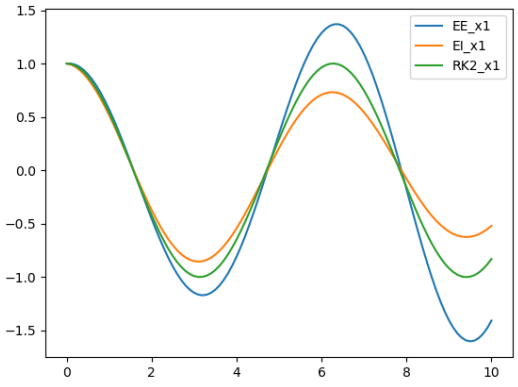
6.2.2 Exemple d'application

On considère le système d'équations différentielles :

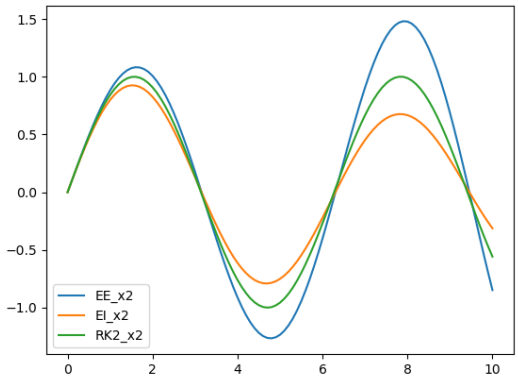
$$\begin{cases} x_1'(t) = -x_2(t), \\ x_2'(t) = x_1(t), \\ x_1(0) = 1, \quad x_1(0) = 0. \end{cases}$$

On note $X(t) = \begin{pmatrix} x_1(t) \\ x_2(t) \end{pmatrix} \in \mathbb{R}^2$. On a donc $X'(t) = MX(t)$ avec $M = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$.

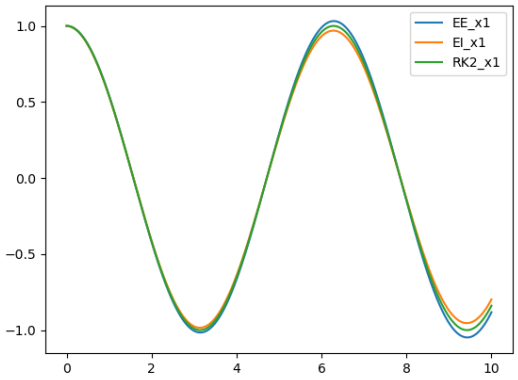
Solutions numériques



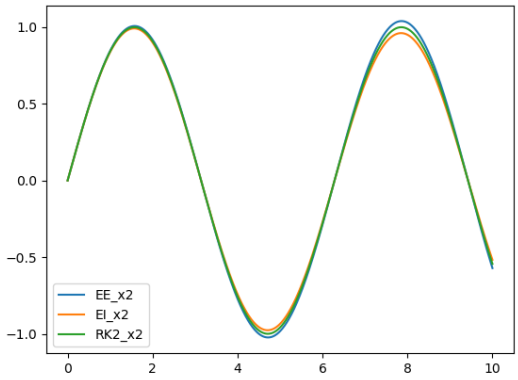
x1 - - N=100



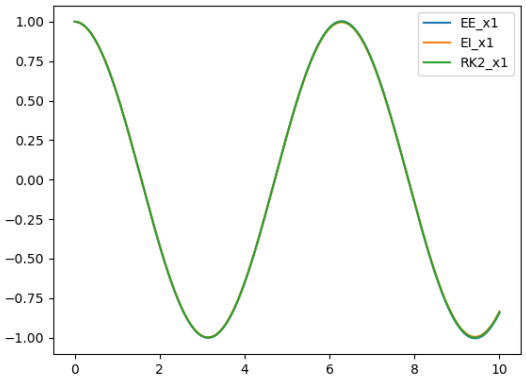
x2 - - N=100



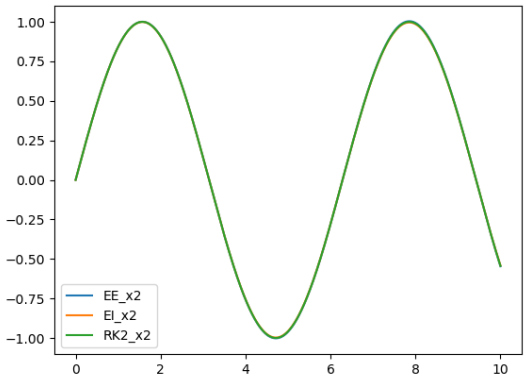
x1 - - N=1000



x2 - - N=1000



x1 - - N=10000



x2 - - N=10000

Solution explicite

Le système étant autonome, la solution unique de ce système est la suivante :

$$X(t) = e^{tM}X_0 \quad \text{avec} \quad X_0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Or M est diagonalisable dans $\mathbb{C}$ et $Sp(M) = -i, i$. En effet $\chi_M(X) = X^2+1 = (X-i)(X+i)$

Donc :

$$M = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} -i & i \\ 1 & 1 \end{pmatrix}\begin{pmatrix} -i & 0 \\ 0 & i \end{pmatrix}\begin{pmatrix} i/2 & 1/2 \\ -i/2 & 1/2 \end{pmatrix} = PDP^{-1}$$

Donc :

$$e^{tM} = Pe^{tD}P^{-1} = \begin{pmatrix} -i & i \\ 1 & 1 \end{pmatrix}\begin{pmatrix} e^{-it} & 0 \\ 0 & e^{it} \end{pmatrix}\begin{pmatrix} i/2 & 1/2 \\ -i/2 & 1/2 \end{pmatrix} = \begin{pmatrix} \cos(t) & -\sin(t) \\ \sin(t) & \cos(t) \end{pmatrix}$$

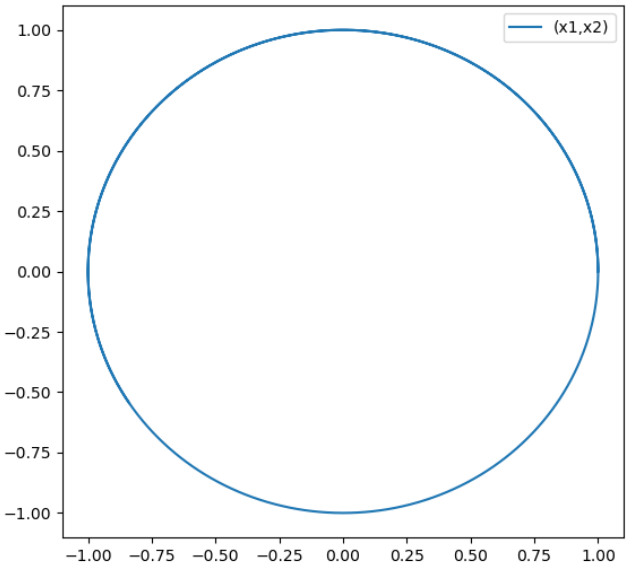
Donc :

$$X(t) = e^{tM}X_0 = \begin{pmatrix} \cos(t) & -\sin(t) \\ \sin(t) & \cos(t) \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} \cos(t) \\ \sin(t) \end{pmatrix}$$

D'où

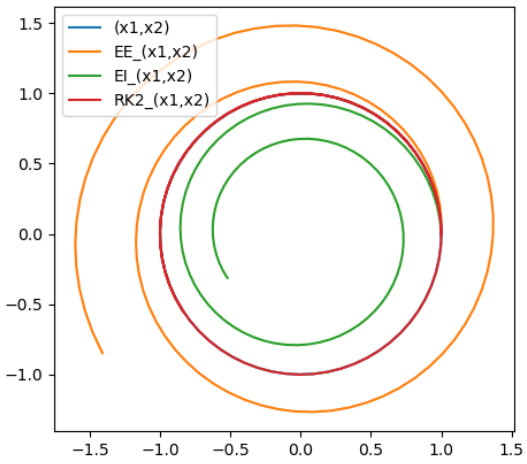
$$\forall t \in [0, T], \quad (x_1(t), x_2(t)) = (\cos(t), \sin(t))$$

qui a pour courbe le cercle unité :

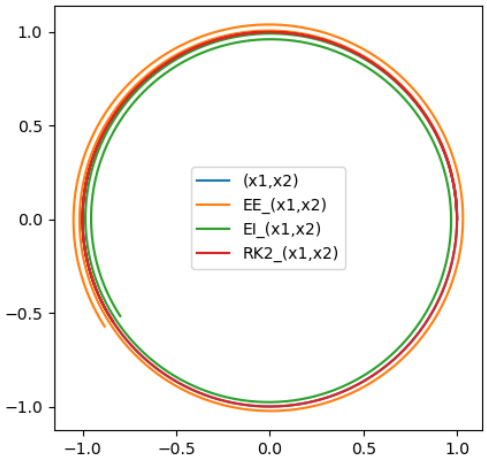


Voyons maintenant les différentes courbes $(x_1(t), x_2(t))$ pour les différentes méthodes :

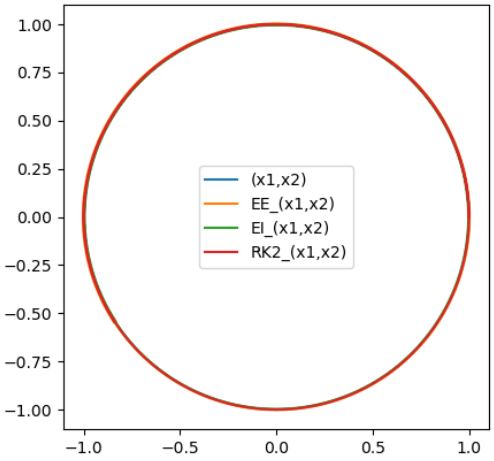
- Pour $N = 100$



- Pour $N = 1000$



- Pour $N = 10000$



Erreur $e(N)$

Pour déterminer la méthode la plus précise, on se propse de calculer les erreurs $e(N)$ pour les différents schémas et les afficher en fonction de N :

$$e(N) = \max_{i=0,\cdots,N} \|X(t_i) - y_i\|_\infty$$

Cette fonction prendera dans notre boucle (pour chaque N), deux tableaux (X et Y) et retournera une erreur $e(N)$. On l'implémentera comme suit :

```
def max_abs_diff(X, Y):

    abs_diff = np.abs(X - Y)

    max_abs_diff_per_row = np.max(abs_diff, axis=1)

    max_abs_diff_overall = np.max(max_abs_diff_per_row)

    return max_abs_diff_overall
```

Tout d'abord, on commence par la création de ($N \leq 100$) :

```
erreur_ee=[]
erreur_ei=[]
erreur_RK2=[]
Nx=np.linspace(10, 100, 100-10+1)
```

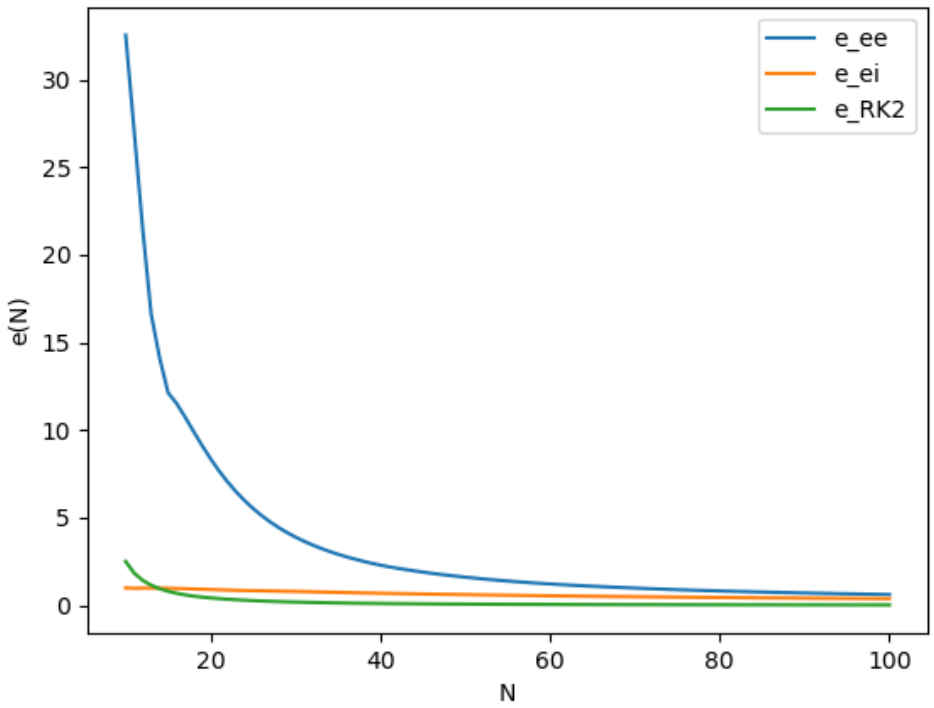
Pour afficher les erreurs (qui se trouvent dans le fichier `e(N).txt`), on utilise :

```
for N in range(10,101):

  temps_N = np.linspace(0, 10, N+1)

  Y = np.array([np.cos(temps_N), np.sin(temps_N)])
  X1 = euler_explicite(fM, x0, temps_N)
  X2 = euler_implicite(M, x0, temps_N)
  X3 = RK2(fM, x0, temps_N)

  e_ee = max_abs_diff(X1, Y)
  e_ei = max_abs_diff(X2, Y)
  e_RK2 = max_abs_diff(X3, Y)

  print("Pour N = ", N, ": \n")
  print("L'erreur associe a EE = ", e_ee)
  erreur_ee.append(e_ee)
  print("L'erreur associe a EI = ", e_ei)
  erreur_ei.append(e_ei)
  print("L'erreur associz a RK2 = ", e_RK2)
  erreur_RK2.append(e_RK2)

  print("\n")
```

Et pour tracer les erreurs en fonction de N :

```
plt.plot(Nx, erreur_ee, label="e_ee")
plt.plot(Nx, erreur_ei, label="e_ei")
plt.plot(Nx, erreur_RK2, label="e_RK2")

plt.legend()
plt.show()
```

Ce qui donne finalement les courbes suivantes :



Ceci nous permet de constater que le schéma de **RUNGE-KUTTA (RK2)** est généralement le schéma **le plus précis** de ceux étuidés ci-dessus. Néanmoins, Chacune de ces méthodes présente des avantages et des inconvénients spécifiques :

Euler Explicite

Avantages :

- Simplicité : Il ne nécessite qu'un pas de calcul simple pour obtenir la nouvelle valeur de la solution.
- Vitesse de calcul : Il ne nécessite qu'une seule évaluation de la fonction dérivée par itération.

Inconvénients :

- Stabilité : Il peut être instable pour des problèmes raides ou des pas de temps trop grands, ce qui peut entraîner une solution numérique incorrecte.

⋆ Précision : Il peut être moins précis car il utilise une approximation linéaire simple pour estimer la pente de la solution.

Euler Implicite

Avantages :

⋆ Stabilité : Il est plus stable pour des problèmes raides ou des pas de temps plus grands, car il utilise une approximation plus robuste pour estimer la pente de la solution.

⋆ Précision : Il utilise une estimation de la pente basée sur la valeur future de la solution.

Inconvénients :

⋆ Complexité : Il est légèrement plus compliqué à mettre en œuvre que l'Euler explicite car il nécessite la résolution d'une équation non linéaire à chaque itération.

⋆ Vitesse de calcul : Il peut être légèrement plus lent à calculer que l'Euler explicite pour la même raison d'avant.

Runge-Kutta 2

Avantages :

⋆ Précision : Il est plus précis que les schémas d'Euler pour un pas de temps donné, car il utilise une estimation de la pente basée sur la moyenne des pentes aux deux extrémités de l'intervalle de temps.

⋆ Stabilité : Il utilise une approximation plus précise de la pente de la solution.

Inconvénients :

⋆ Complexité : Il nécessite plusieurs évaluations de la fonction dérivée à chaque itération.

⋆ Vitesse de calcul : Il nécessite plusieurs évaluations de la fonction dérivée à chaque itération.

6.3 Modèle non linéaire

On considère le système différentiel :

$$\begin{cases} x_1'(t) = 2x_2(t) \\ x_2'(t) = -3x_1(t)^2 - 12x_1(t) \\ x(0) = (-3, 0)^T \end{cases}$$

On peut montrer que la solution exacte $\{(x_1(t), x_2(t)) \mid t \geq 0\}$ est contenue dans la courbe paramétrée :

$$x_1^3 + 6x_1^2 + x_2^2 = c_0, \quad x = (x_1, x_2) \in \mathbb{R}^2$$
$$c_0 = x_1(0)^3 + 6x_1(0)^2 + x_2(0)^2$$

On peut aussi écrire le système ci-dessus sous la forme :

$$\begin{cases} X'(t) = f(t, X(t)) \\ x(0) = (-3, 0)^T \end{cases}$$

Avec :

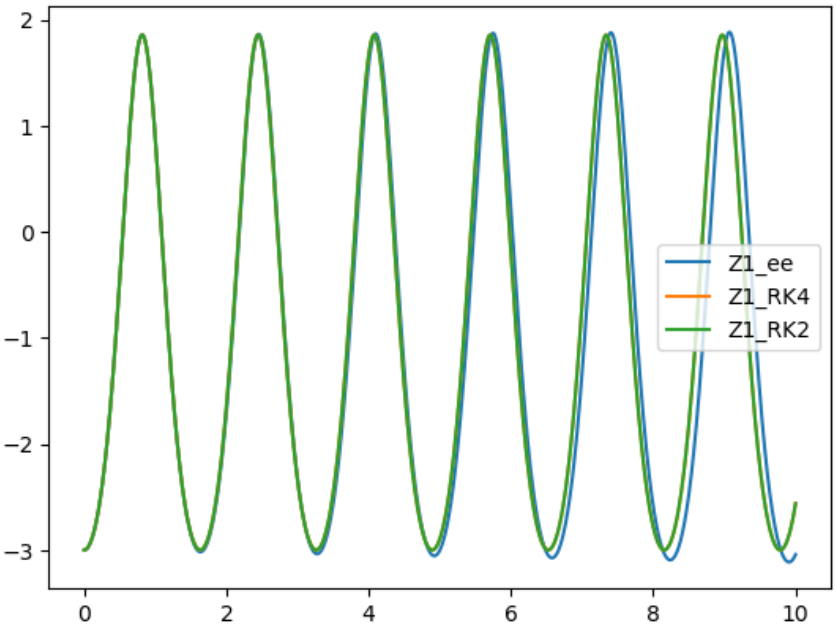
$$\begin{array}{rccc} f: & \mathbb{R} \times \mathbb{R}^2 & \longrightarrow & \mathbb{R}^2 \\ & (t, (x_1(t), x_2(t))) & \longmapsto & \begin{pmatrix} 2x_2(t) \\ -3x_1(t)^2 - 12x_1(t) \end{pmatrix} \end{array}$$

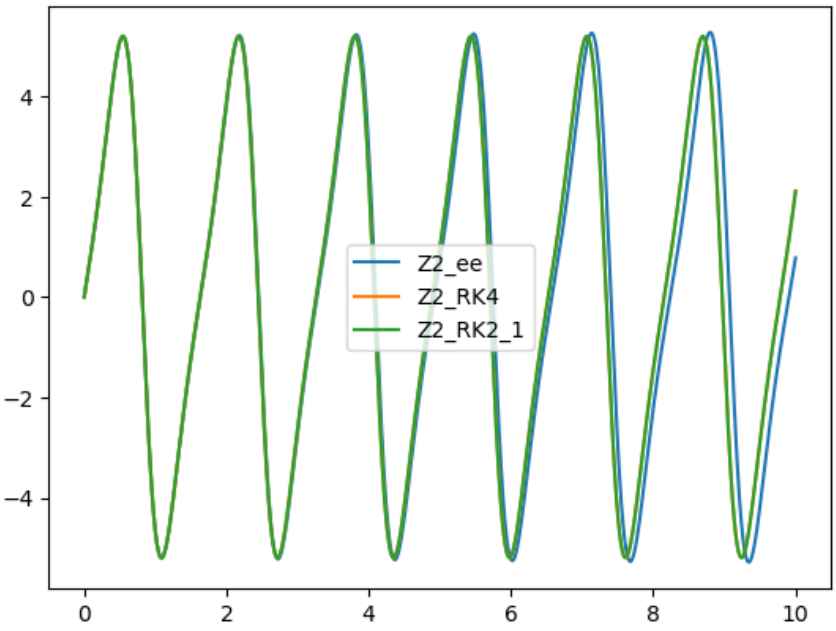
Qui peut être implémentée comme suit :

```
def fnl(t, x):
    x1, x2 = x[0], x[1]
    return np.array([2*x2, -3*x1*x1-12*x1])
```

Ensuite, pour simuler numériquement ce système différentiel, on fait appel aux fonctions `euler_explicite`, `RK2` et `RK4` :

```
tempsnl=np.linspace(0,T,10000)

Z1=euler_explicite(fnl, x0, tempsnl)
Z2=RK4(fnl, x0, tempsnl)
Z3=RK2(fnl, x0, tempsnl)

plt.plot(tempsnl, Z1[0,:], label="Z1_ee")
plt.plot(tempsnl, Z2[0,:], label="Z1_RK4")
plt.plot(tempsnl,Z3[0,:], label="Z1_RK2_1")
plt.legend()
plt.show()

plt.plot(tempsnl, Z1[1,:], label="Z2_ee")
plt.plot(tempsnl,Z2[1,:], label="Z2_RK4")
plt.plot(tempsnl,Z3[1,:], label="Z2_RK2_1")
plt.legend()
plt.show()
```

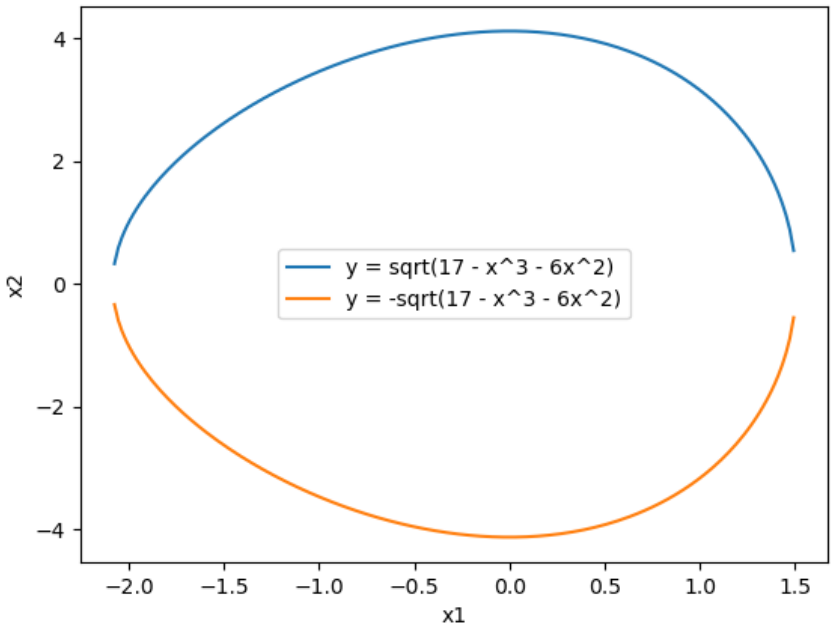




Pour tracer la courbe paramétrée correspondante à la solution exacte, on peut prendre :

```
x = np.linspace(-5, 5, 500)
y = np.sqrt(17 - x**3 - 6*x**2)


plt.plot(x, y, label='y = sqrt(17 - x^3 - 6x^2)')
plt.plot(x, -y, label='y = -sqrt(17 - x^3 - 6x^2)')
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend()
plt.show()
```

Ce qui donne :



Le vide que vous constatez près de l'axe des abscisses est causé par la résolution limitée de notre grille de points x1 en comparaison de la complexité du programme. Pour combler ces lacunes, il serait nécessaire d'augmenter le nombre de points dans notre grille x1 en augmentant la valeur de N. Toutefois, cela pourrait également entraîner une augmentation du temps de calcul et de la charge de travail pour le système.