Chapitre 8

Intégration numérique et résolution d'EDO (avec SciPy)

8.1 Intégration numérique

La bibliothèque `SciPy`, à travers le module `scipy.integrate`, fournit plusieurs routines d'intégration parmi lesquelles certaines portent des noms qui nous sont désormais familiers. On peut citer en particulier les méthodes `trapezoid` (formule du trapèze) et `simpson` (formule de Simpson). Mentionnons également la fonction `quad` qui permet de calculer très facilement et très efficacement une intégrale définie du type :

$$I = \int_a^b f(x)dx\,.$$

Selon la documentation, la routine `quad` utilise la bibliothèque QUADPACK. Elle admet pour arguments la fonction $f(x)$ à intégrer (l'intégrande), ainsi que les bornes inférieure et supérieure a et b. Elle retourne deux valeurs sous la forme d'un tuple :

- l'approximation numérique de l'intégrale ;
- un majorant de l'erreur commise.

A titre d'exemple simple, nous allons reprendre l'exercice 3 de la Série 22 d'Analyse I et calculer l'aire du domaine (fini) limité par les courbes d'équation :

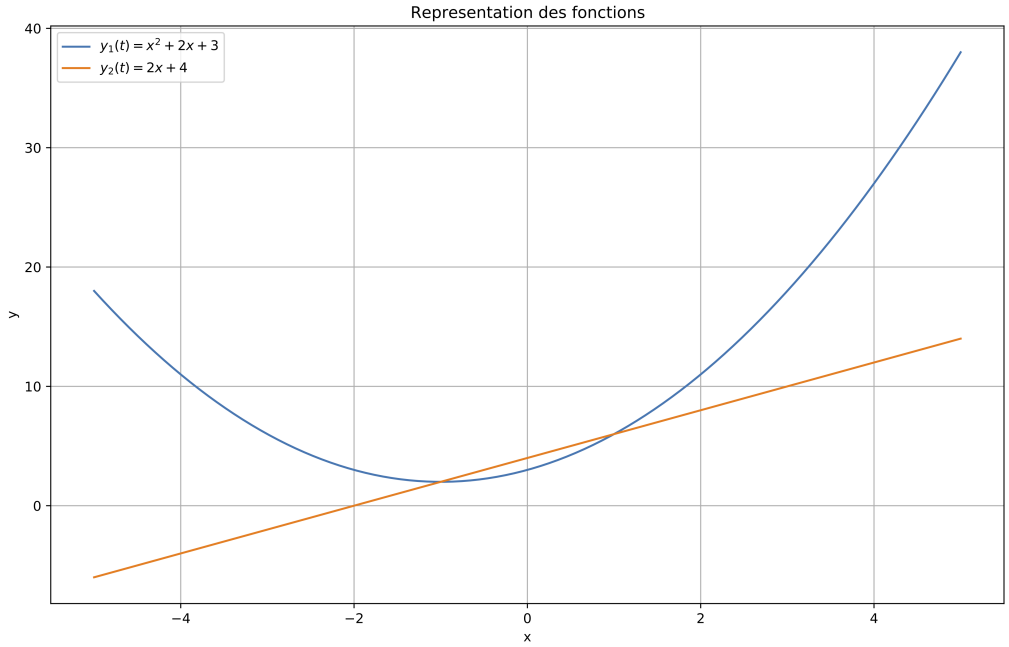
$$y_1 = x^2 + 2x + 3 \quad \text{et} \quad y_2 = 2x + 4\,.$$

Il est aisé de représenter ces courbes pour localiser le domaine qui nous intéresse :

```
import numpy as np
import matplotlib.pyplot as plt
def y_1(x):
    return x**2 + 2*x + 3
def y_2(x):
    return 2*x + 4
x = np.linspace(-5,5,100)
plt.figure(figsize=(12,8))
plt.plot(x,y_1(x), label='$y_1(t) = x^2 + 2x + 3$')
plt.plot(x,y_2(x), label='$y_2(t) = 2x + 4$')
```

```
plt.grid()
plt.xlabel('x') ; plt.ylabel('y') ; plt.legend(loc='best')
plt.title('Représentation des fonctions')
plt.show()
```

Figure obtenue :



On peut déterminer les bornes a et b du domaine en définissant une fonction $f(x) = y_2(x) - y_1(x) = -x^2 + 1$ et en recherchant ses zéros (par exemple, par la méthode de la bissection, de la sécante ou de Newton) :

```
from scipy import optimize
def f(x):
    return y_2(x) - y_1(x)
a = optimize.bisect(f, -4, 0)
b_1 = optimize.newton(f,2)
def f_prime(x):
    return -2*x
b_2 = optimize.newton(f,2,fprime=f_prime)
print(a, b_1, b_2)
```

```
-1.0 1.0000000000000002 1.0
```

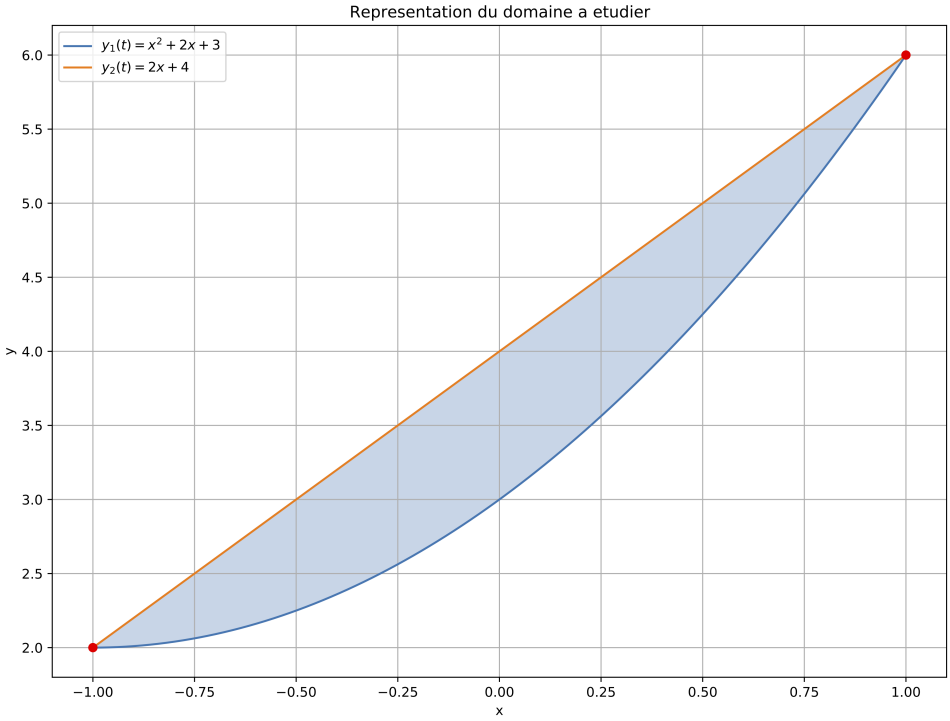
On en déduit que $a = -1$ et $b = 1$.
On peut alors préciser le domaine à considérer :

```
x = np.linspace(a,b_2,100)

plt.figure(figsize=(12,8))
plt.plot(x,y_1(x), label='$y_1(t) = x^2 + 2x + 3$')
plt.plot(x,y_2(x), label='$y_2(t) = 2x + 4$')

plt.fill_between(x, y_2(x), y_1(x), alpha=0.3)

plt.grid()
plt.plot([a],[y_1(a)],'ro')
plt.plot([b_2],[y_1(b_2)],'ro')
plt.xlabel('x')
plt.ylabel('y')
plt.legend(loc='best')
plt.title('Représentation du domaine à étudier')
plt.show()
```



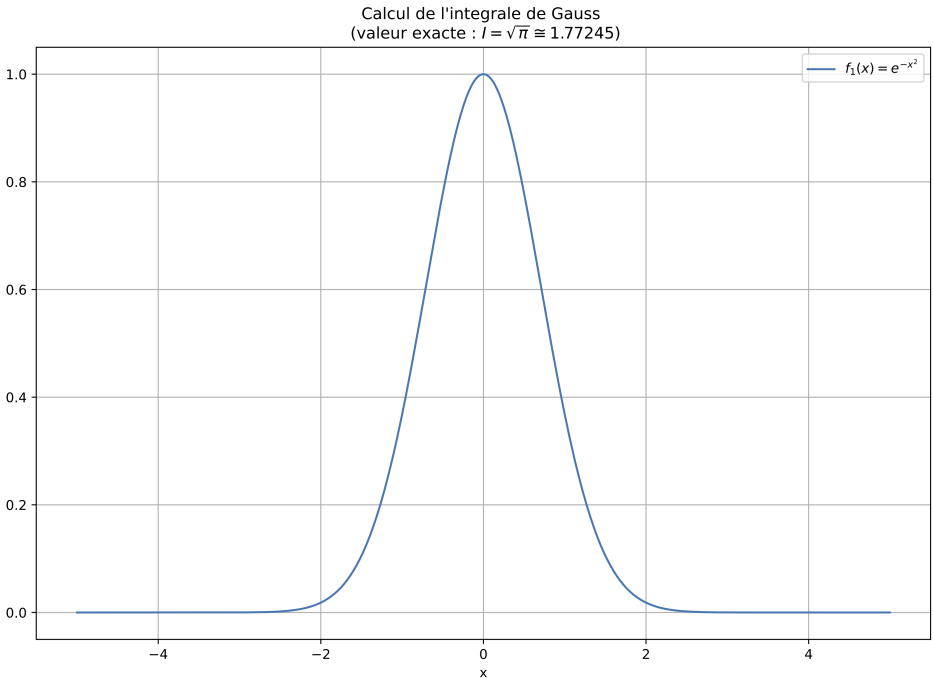
. . . et intégrer :

```
from scipy import integrate
I, erreur = integrate.quad(f, a, b_2)
print(I, erreur)
```

```
1.3333333333333335 1.4802973661668755e-14
```

Remarquons que la fonction `quad` permet également d'effectuer des intégrations sur un intervalle non borné. Ainsi, il est par exemple possible d'obtenir une bonne approximation d'une intégrale de Gauss :

$$I = \int_{-\infty}^{\infty} e^{-x^2} dx \,.$$



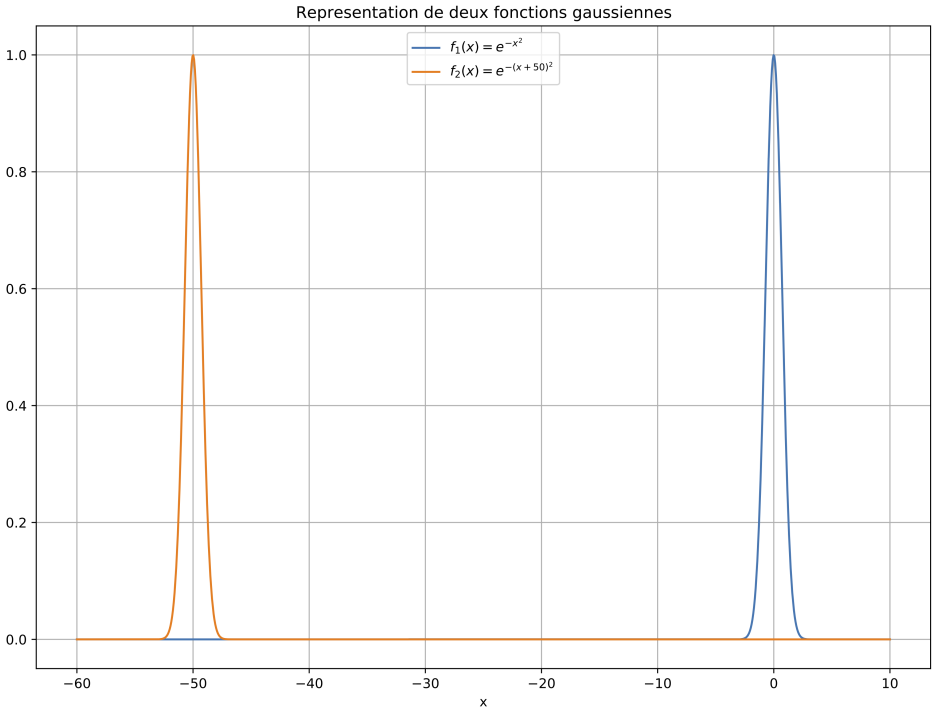

```
import numpy as np
import matplotlib.pyplot as plt
def f_1(x): return np.exp(-x**2)

x = np.linspace(-5,5,1000)
plt.figure(figsize=(12,8))
plt.plot(x,f_1(x), label='$f_1(x) = e^{-x^2}$')
plt.grid()
plt.xlabel('x') ; plt.legend(loc='best')
plt.title("Calcul de l'intégrale de Gauss \n (valeur exacte :
 ↪  $I=\sqrt{\pi}\cong 1.77245$)")
plt.show()

from scipy import integrate
I, erreur = integrate.quad(f_1, -np.inf, np.inf)
print(I, erreur)
```

```
1.7724538509055159 1.4202636780944923e-08
```

Comme le montrent les résultats ci-dessous, il convient toutefois d'utiliser une "boîte noire" numérique telle que la fonction `quad` avec tout l'esprit critique nécessaire :



$$I = \int_{-\infty}^{\infty} e^{-x^2} dx = \int_{-\infty}^{\infty} e^{-(x+50)^2} dx = \sqrt{\pi}\,.$$

```python
import numpy as np
import matplotlib.pyplot as plt
def f_1(x):
    return np.exp(-x**2)
def f_2(x):
    return np.exp(-(x+50)**2)

x = np.linspace(-60,10,1000)

plt.figure(figsize=(12,8))
plt.plot(x,f_1(x), label='$f_1(x) = e^{-x^2}$')
plt.plot(x,f_2(x), label='$f_2(x) = e^{-(x+50)^2}$')
plt.grid()
plt.xlabel('x') ; plt.legend(loc='best')
plt.title('Représentation de deux fonctions gaussiennes')
```

```
plt.show()

from scipy import integrate
I_1, erreur_1 = integrate.quad(f_1, -np.inf, np.inf)
I_2, erreur_2 = integrate.quad(f_2, -np.inf, np.inf)
print(I_1, erreur_1)
print(I_2, erreur_2)
```

```
1.7724538509055159 1.4202636780944923e-08
2.9480983632192556e-198 0.0
```

Avant toute intégration numérique, il convient d'étudier ou/et de représenter la fonction à intégrer de manière à s'assurer que son comportement (présence de singularités, d'importantes variations locales, etc.) ne met pas en défaut la méthode numérique envisagée.

8.2 Résolution d'un système d'équations différentielles ordinaires

La fonction `odeint` permet de résoudre numériquement un système d'équations différentielles ordinaires (EDO) du premier ordre de la forme :

$$\left\{ \begin{array}{rcl} \dot{\vec{y}}(t) & = & \vec{f}(\vec{y}(t), t)\,, \\ \vec{y}(t_0) & = & \vec{y}_0\,, \end{array} \right.$$

et d'obtenir une approximation $\vec{y}(t)$ de l'évolution du système pour des temps t supérieurs à t_0. Ainsi, il est par exemple possible de résoudre l'équation (vectorielle) du pendule simple que nous étudions dans la série 24 en considérant le système :

$$\left\{ \begin{array}{rcl} \begin{pmatrix} \dot{\theta} \\ \dot{v} \end{pmatrix} & = & \begin{pmatrix} v \\ -\omega_0^2 \sin\theta \end{pmatrix}\,, \\ \begin{pmatrix} \theta(0) \\ v(0) \end{pmatrix} & = & \begin{pmatrix} \theta_0 \\ v_0 \end{pmatrix}\,. \end{array} \right.$$

Les constantes θ_0 et v_0 correspondent aux deux conditions initiales et $\omega_0^2 = g/L$ (où L est la longueur du fil). La fonction `odeint` retourne alors un vecteur solution :

$$\vec{y}(t) = \begin{pmatrix} \theta(t) \\ v(t) \end{pmatrix}\,.$$

Ainsi, le code permettant de résoudre l'EDO du pendule simple pourrait débuter de la manière suivante :

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
```

```
4
5  g = 9.81 ; L = 2 ; omegazerocarre = g/L # constantes
6  y_0 = np.array([np.pi/6, 0.0]) # vecteur conditions initiales
7  # ou : y_0 = [np.pi/6, 0.0]
```

On définit ensuite le membre de droite de l'EDO, `f(t,y)`, sous forme vectorielle :

```
1  def f(t, y):
2      theta, v = y
3      return [v, -omegazerocarre*np.sin(theta)]
```

Cette fonction retourne un tableau de la même forme (`shape`) que `y` .
Il est également nécessaire de construire un vecteur temps dont les éléments correspondent aux instants t_n pour lesquels nous souhaitons obtenir une valeur approchée de la solution y (la première entrée de ce tableau devant être la valeur initiale t_0) :

```
1  t_0 = 0 ; T = 10 ; N = 200 # N est le nombre de sous-intervalles
2  t = np.linspace(t_0, T, N+1) # vecteur contenant N+1 elements t_n
```

Il convient de noter que la partition choisie au moment de la construction de `t` ne sera pas prise en compte par `odeint` pour la résolution numérique du problème. En effet, `odeint` va adapter le pas h utilisé en fonction de deux valeurs (qui correspondent à des tolérances relative et absolue) fournissant une majoration de l'erreur locale. Ces valeurs `rtol` et `atol` sont des `keyword arguments` valant par défaut environ $1.5 \cdot 10^{-8}$. La solution approchée aux instants donnés par `t` est ensuite obtenue par interpolation.
La ligne suivante fournit un tableau solution de dimension 2 de forme `(N+1,2)` (à savoir ici `(201,2)`) :

```
1  sol = odeint(f, y_0, t, tfirst=True)
```

Le paramètre optionnel `tfirst` permet de préciser la position de la variable `t` dans la fonction `f` : `f(y,t)` ou `f(t,y)` .
Les solutions peuvent alors être "décompactées" :

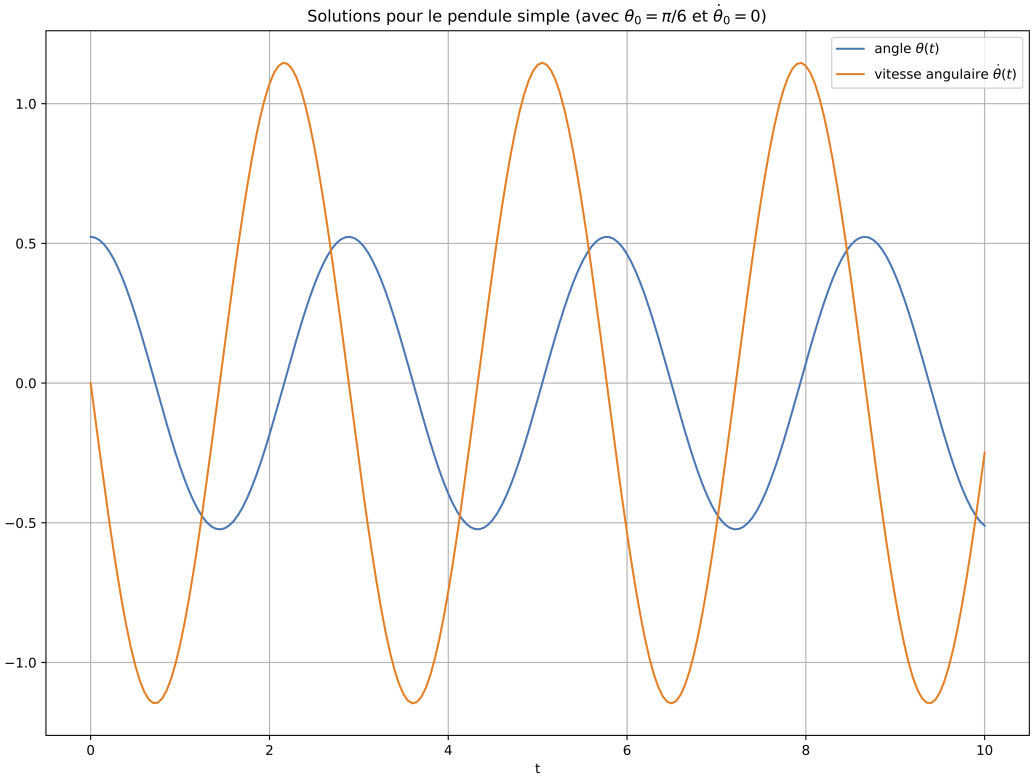
```
1  angle = sol[:,0]
2  vitesse_angulaire = sol[:,1]
```

...et visualisées de manière habituelle :

```
1  plt.figure(figsize=(12,8))
2  plt.plot(t, angle, label=r"angle ${\theta(t)}$")
3  plt.plot(t, vitesse_angulaire, label=r"vitesse angulaire "\
4  r"$\dot\theta(t)$")
5  plt.grid()
6  plt.title("Solutions pour le pendule simple "\
7  r"(avec $\theta_0 = {\pi/6}$ et $\dot\theta_0 = {0}$)")
8  plt.xlabel('t')
```

```
plt.legend(loc='best')
plt.show()
```

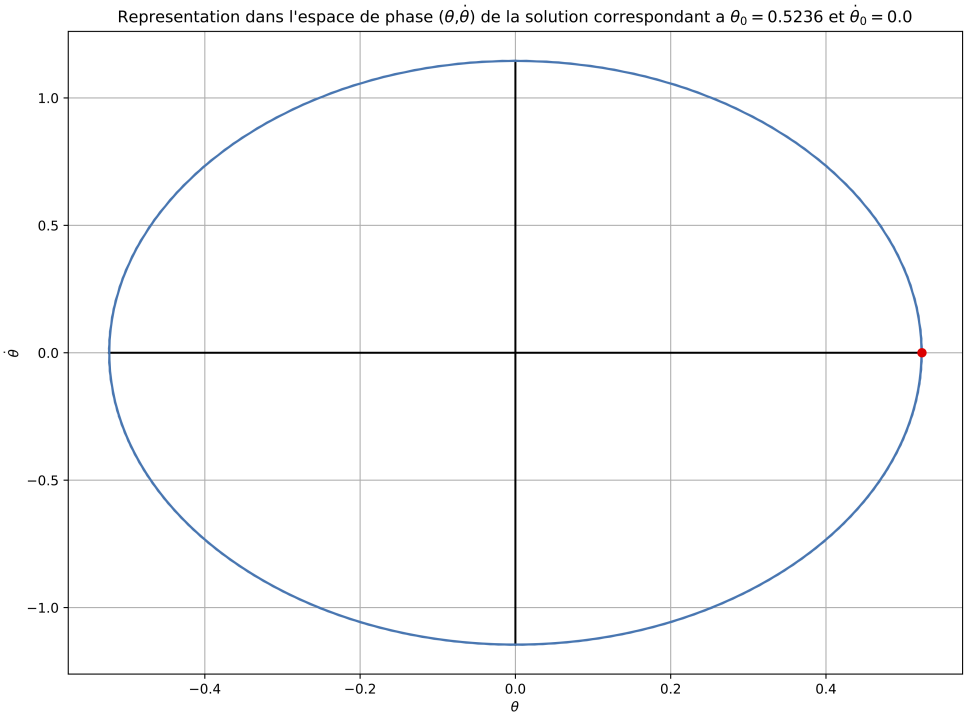
Figure générée :



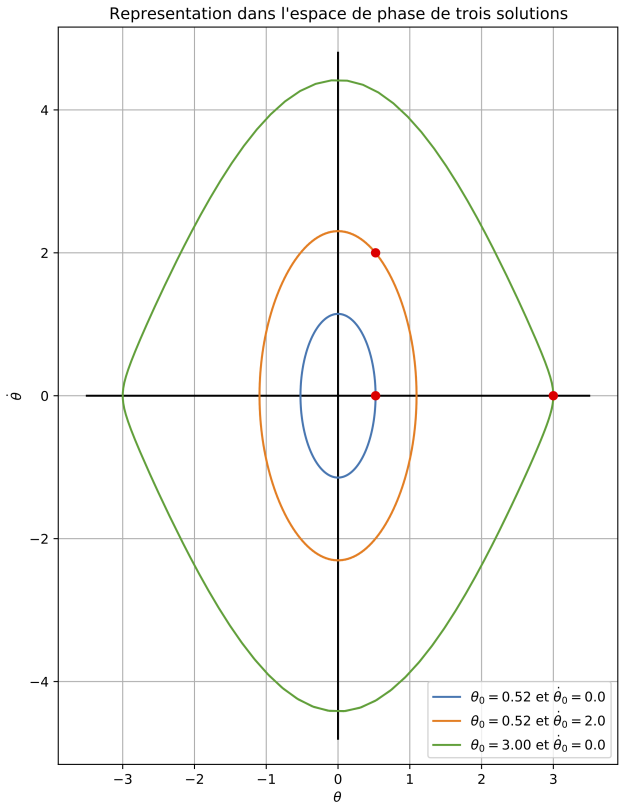
Le mouvement d'un pendule simple peut également être illustré en représentant la solution dans un espace particulier, appelé **espace de phase** avec θ en abscisse et $\dot{\theta}$ en ordonnée :

```
plt.figure(figsize=(12,8))
plt.plot([min(angle),max(angle)],[0, 0],c='k')
plt.plot([0,0],[min(vitesse_angulaire),max(vitesse_angulaire)],c='k')
plt.plot(angle,vitesse_angulaire)
plt.plot([y_0[0]],[y_0[1]],'or')
plt.grid()
plt.title(r"Représentation dans l'espace de phase "\
rf"($\theta$,$\dot\theta$) de la solution correspondant "\
rf"à $\theta_0 = {y_0[0]:.4f}$ et $\dot\theta_0 = {y_0[1]}$")
plt.xlabel(r'$\theta$')
plt.ylabel(r'$\dot\theta$')
plt.show()
```

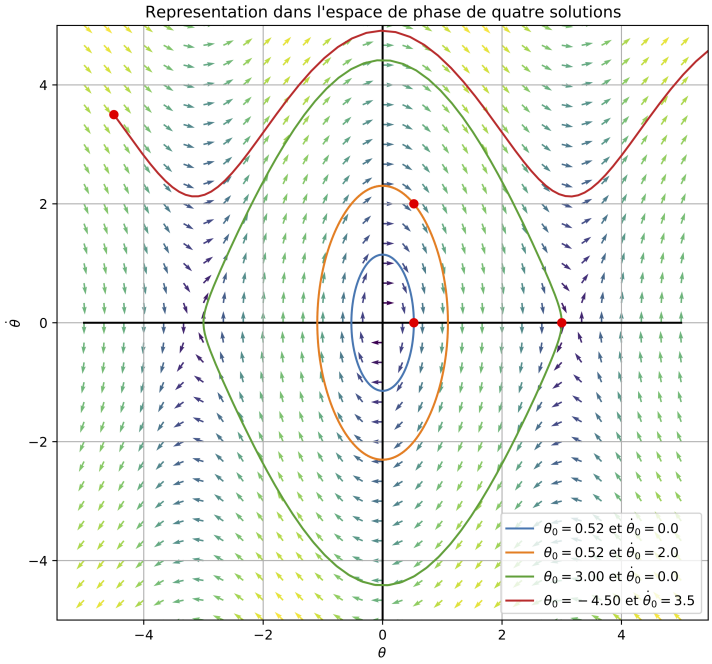
Figure générée :



Trois couples différents de conditions initiales (points rouges) fournissent trois courbes différentes :



Trajectoires et champ de directions :


Representation dans l'espace de phase de quatre solutions
$\dot{\theta}$
θ
4
2
0
−2
−4
−4
−2
0
2
4
$\theta_0 = 0.52$ et $\dot{\theta}_0 = 0.0$
$\theta_0 = 0.52$ et $\dot{\theta}_0 = 2.0$
$\theta_0 = 3.00$ et $\dot{\theta}_0 = 0.0$
$\theta_0 = -4.50$ et $\dot{\theta}_0 = 3.5$