Cours Info - 14

Résolution numérique d'une équation différentielle ordinaire

D.Malka

MPSI 2015-2016

Sommaire

Sommaire

Equations différentielles

Equations différentielles ordinaires (EOD)

$$y' = ay$$

$$y' = \frac{y}{a.t + y}$$

$$\begin{cases} x' = ax - bxy \\ y' = cy + dyx \end{cases}$$

Forme générale (pour une fonction dépend d'une seule variable t) :

$$\begin{cases} y' = f(y(t), t) \\ y(a) = y_0 \end{cases}$$

Existence de solutions

Existence de solutions

La majorité des équations différentielles n'admet pas de solution analytique.

Mais des preuves non constructives peuvent assurer qu'elles admettent une solution.
Exemple : le théorème de Cauchy-Lipschitz.

Résolution approchée

On détermine alors la solution de façon approchée par des algorithmes en discrétisant l'équation.

La fonction solution de l'équation différentielle est alors une liste de valeurs discrètes prises par la fonction pour différents antécédents.

Il faut s'assurer de la stabilité du problème ainsi que de la convergence de la stabilité numérique de l'algorithme. Il faut contrôler et estimer l'erreur commise par rapport à la solution exacte.

Sommaire

Intérêt

Intérêt de la résolution numérique d'une équation :

- Résoudre des équations n'ayant pas de solution analytique.
- Observer facilement l'influence d'un paramètre sur la solution d'une équation, éventuellement analytique.

Méthode d'Euler explicite

- On cherche à résoudre sur $[a, b]$ l'équation de la forme :

$$y' = f(y(t), t) \quad \text{avec} \quad y(a) = y_0$$

- Si $y(t)$ est de classe $\mathcal{C}_1$ alors, connaissant $y(t_i)$, on peut évaluer de façon approchée, $y(t_{i+1})$ par son développement de Taylor à l'ordre 1. *Schéma numérique explicite :*

$$y(t_{i+1}) = y(t_i) + y'(t_i)(t_{i+1} - t_i)$$

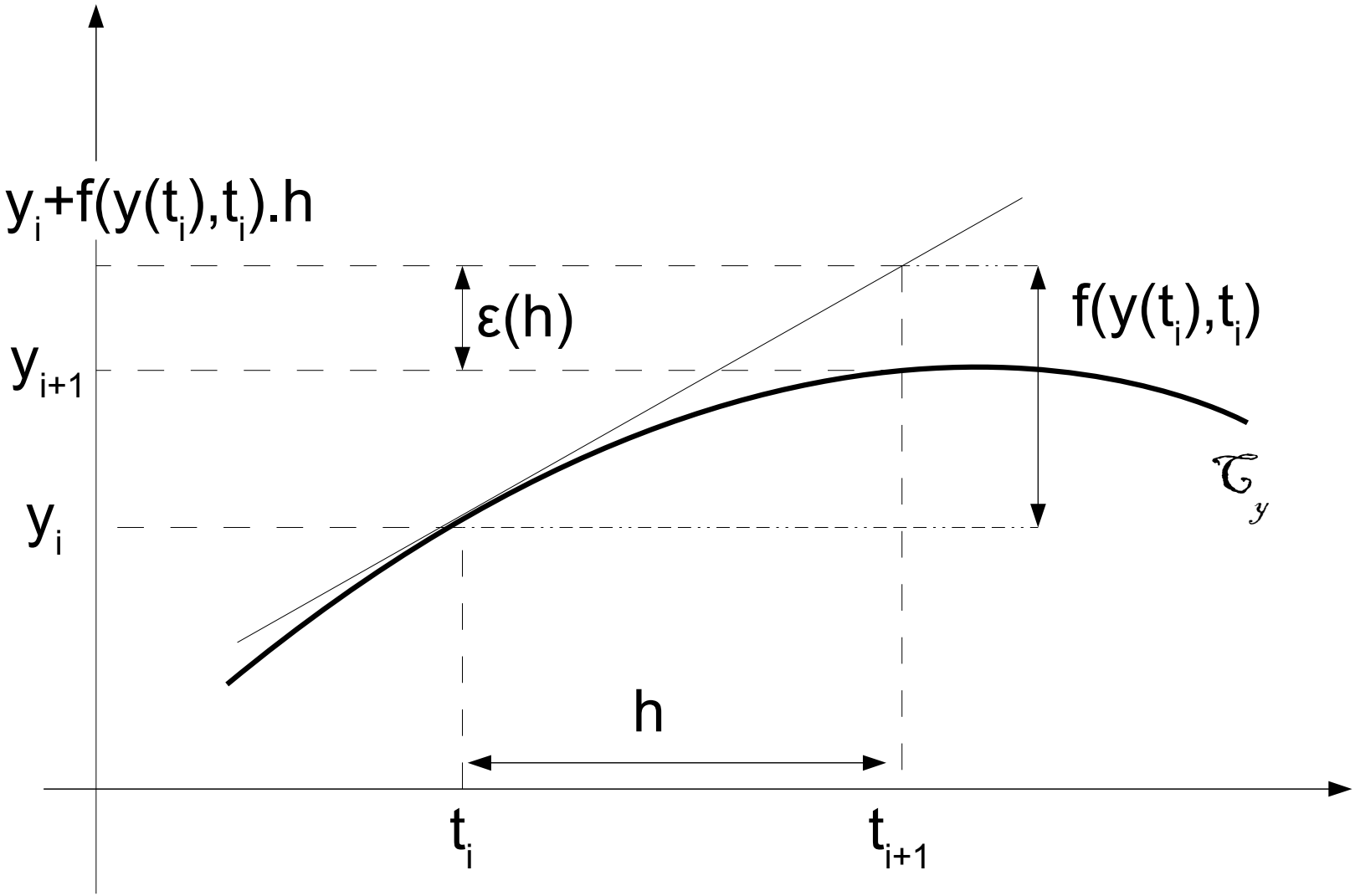
- Soit en posant $h = t_{i+1} - t_i$ et avec $y'(t_i) = f(y(t_i), t_i)$:

$$\boxed{y(t_{i+1}) = y(t_i) + f(y(t_i), t_i).h}$$

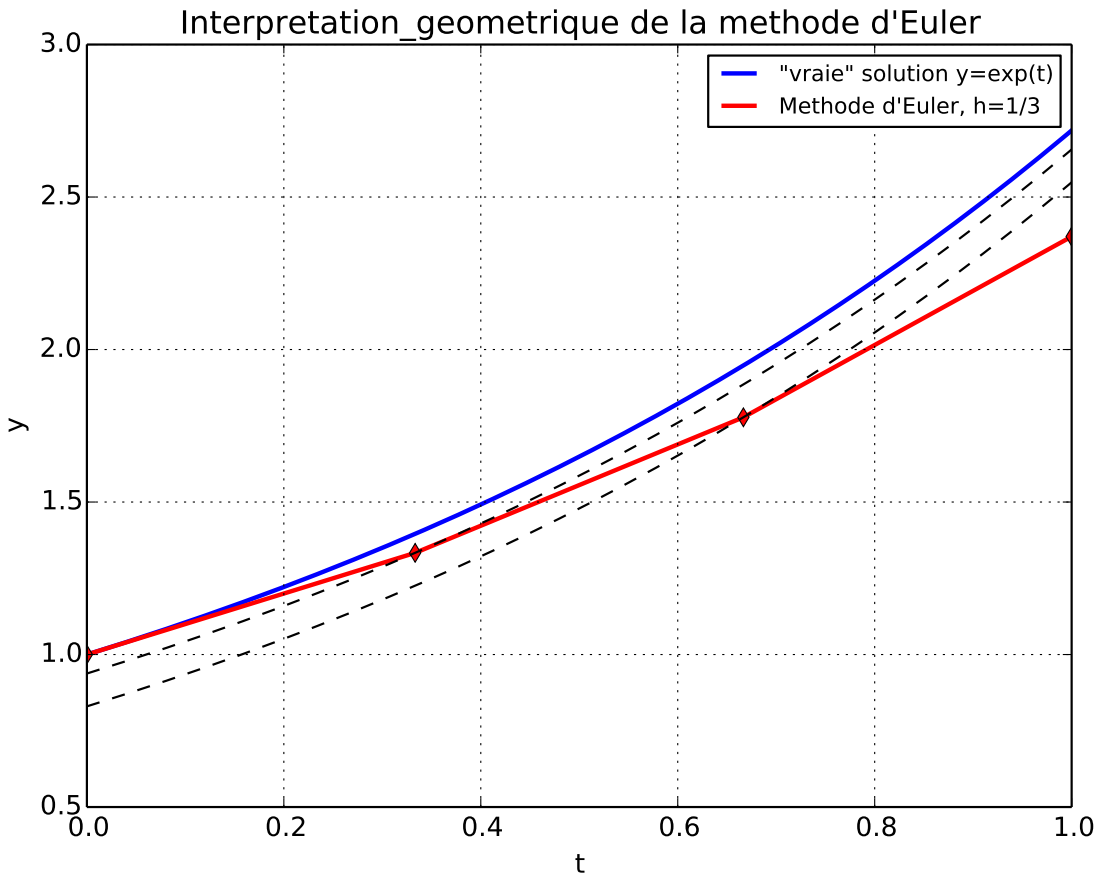
Connaissant $y(a)$, on évalue par itérations successives les valeurs approchées de la fonction $y(t)$:

$$[y(a), y(a+h), y(a+2h), y(a+3h) \ldots y(b-h), y(b)]$$

Interprétation géométrique



Interprétation géométrique

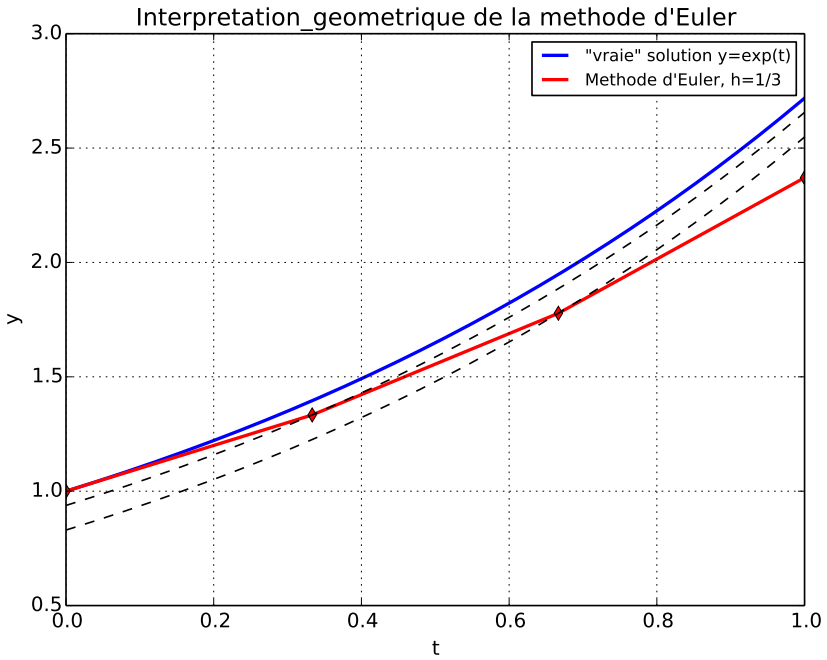


La courbe de la fonction est approchée par les tangentes successives à la courbe.

Implémentation en Python

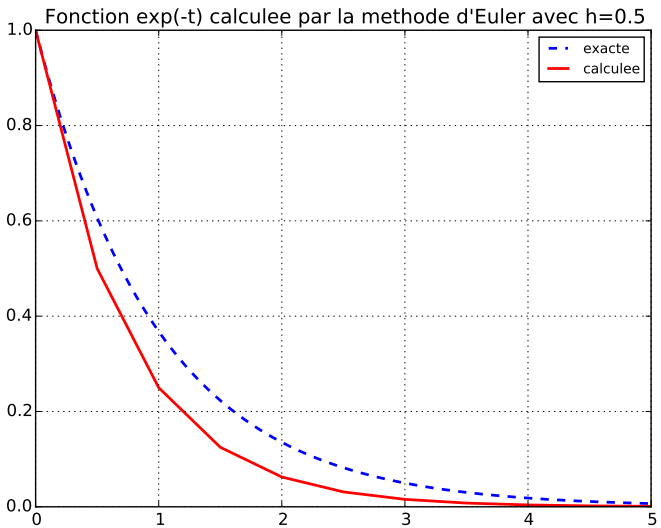
```
1  def Euler(f,y0,a,b,n):
2      '''
3      Integre l'equation y'(t)=f(y(t),t) avec y(a)=y0 selon le schema
           explicite d'Euler
4      t,y : list of floats
5      f : function
6      y0,a,b :floats. [a,b] : intervalle de resolition
7      n : int, nombre de points. Pas de discretisation : h=(b-a)/(n-1)
8      '''
9      h=(b-a)/(n-1)
10     t=[0]*n
11     for i in range(0,n):
12         t[i]=a+i*h
13     y=[0]*n;y[0]=y0
14     for i in range(0,n-1):
15         y[i+1]=y[i]+f(y[i],t[i])*h
16     return t,y
```

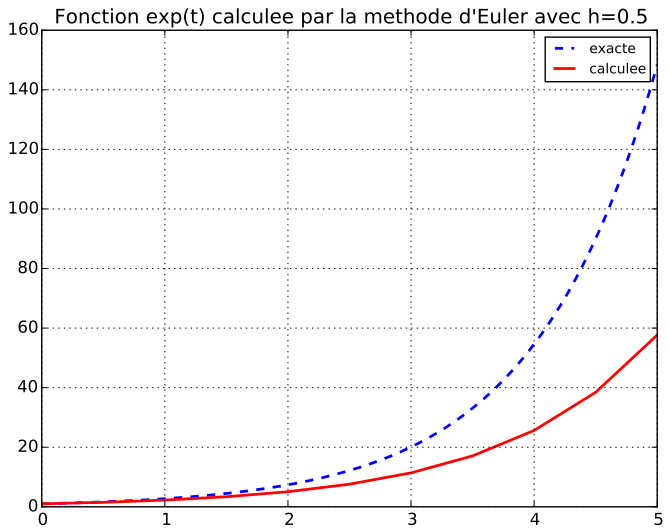
Erreur



- Erreur locale $e_i \sim \frac{1}{2} f''(x_i) h^2$.
- Erreur globale E qui provient de l'accumulation des erreurs des itérations précédentes.

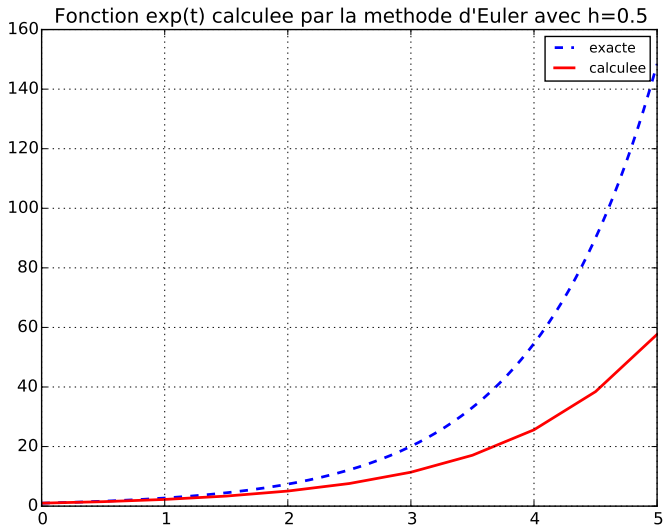
Influence du pas discrétisation

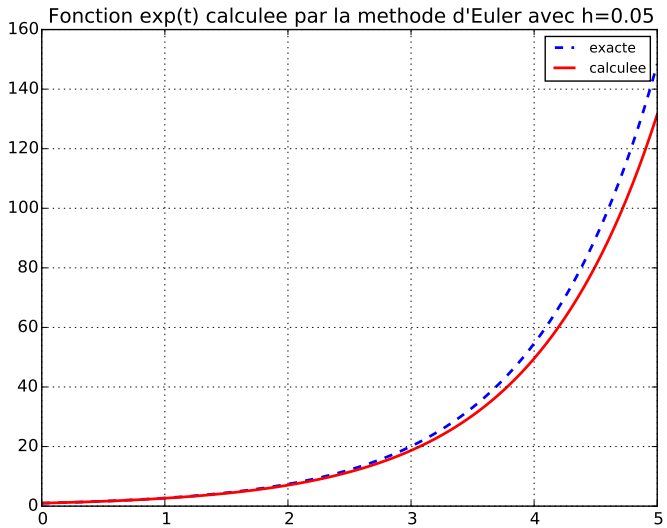




- toujours instable si l'EOD est instable lui-même ($y' - y = 0$) i.e l'erreur globale diverge.
- stable si l'EOD stable et si h suffisamment petit ($y' + y = 0$) i.e l'erreur globale est bornée.

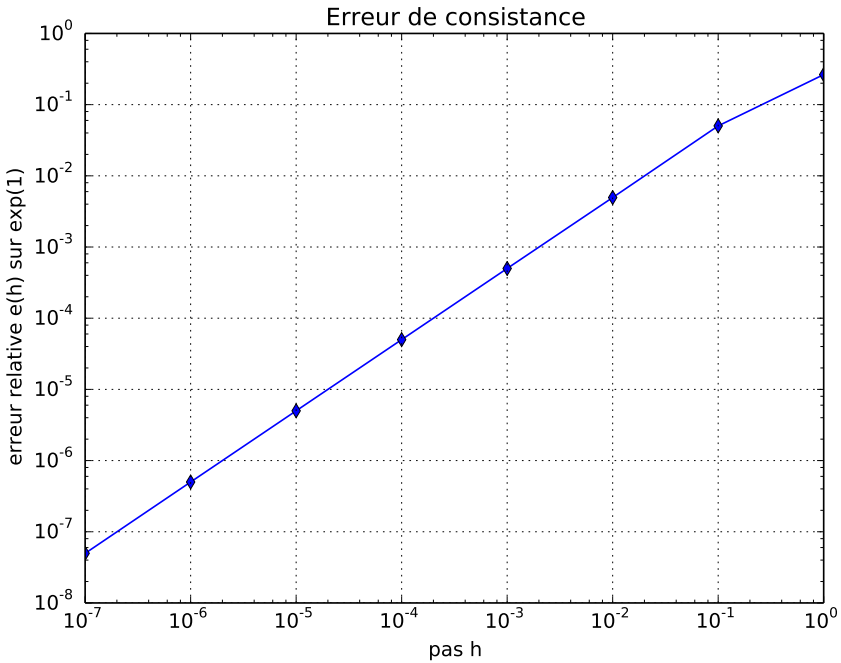
Influence du pas discrétisation





- Plus h est petit, plus la solution approchée sera consistante ...
- ... jusqu'à un certain point. En deçà les erreurs numériques explosent !
- **Complexité de la méthode d'Euler** : $O(n)$ avec $n = \dfrac{b-a}{h}$. Plus h est petit, plus le temps de calcul est élevé.
- Il faut trouver un compromis.

Consistance de la méthode d'Euler



Erreur sur relative sur e par calcul itératif pour différents pas h avec la méthode d'Euler : pour h petit, $\log(e) = \alpha \log(h) + b$ soit $e = k.h^{\alpha}$.

Autres méthodes en bref

Quelques autres méthodes (utilisée dans les fonctions prédéfinies de `scipy`).

- Idée : pousser le développement de Taylor plus loin.
  - Méthode d'Heun : ordre 2
  - Méthode de Runge-Kutta : ordre 4
- Méthode de pas adaptatif

Fonction prédéfinies en Python

Module `integrate` de la bibliotheque `scipy` :

```
import scipy.integrate as integ
```

Fonction `odeint`.

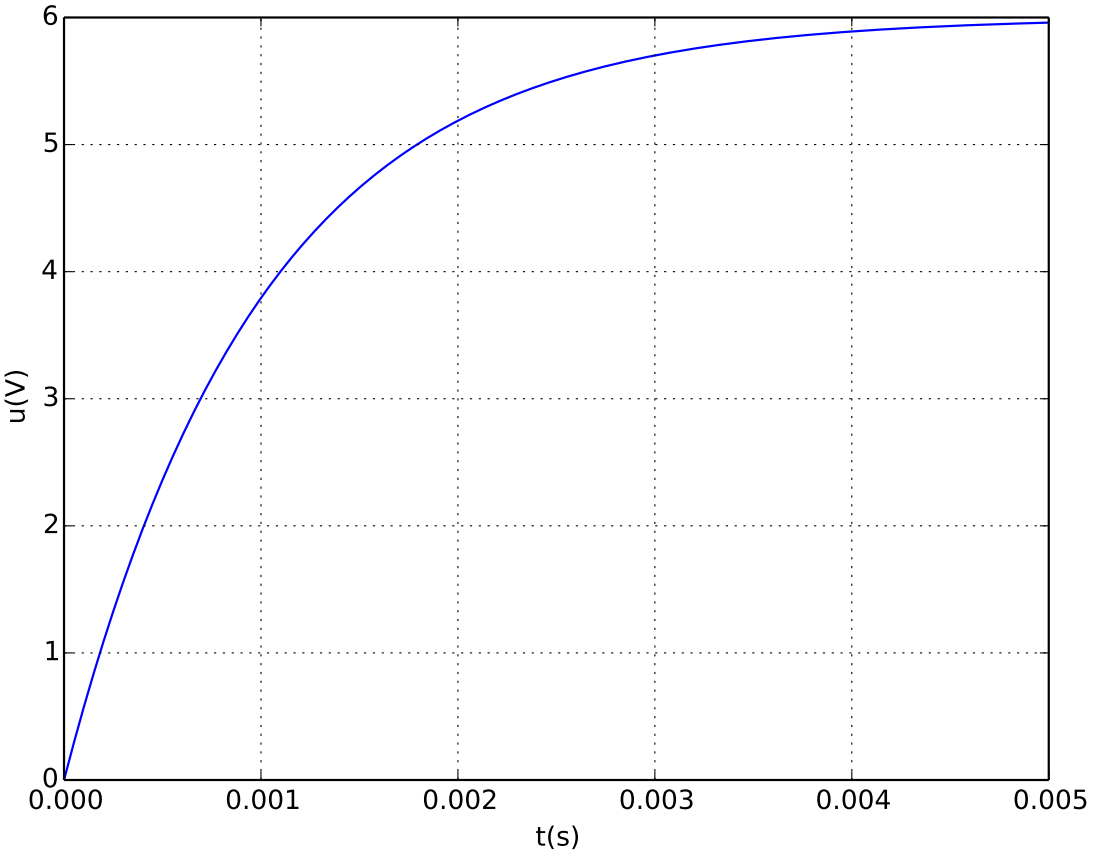
```
1 import scipy.integrate as integ
2 integ.odeint(f,init,t)
3 #f fonction telle que y'=f(y,t)
4 # t=[a...b], instants t auxquels y(t) est calculee
5 #y(a)=init
```

Charge d'un condensateur

$$\frac{dU}{dt} + \frac{u}{\tau} = \frac{E}{\tau}$$

```
1  C=1e-6
2  R=1e3
3  tau=R*C
4  E=6
5
6  def f(u,t):
7      rhs=-u/tau+E/tau
8      return rhs
9
10 u0=0
11 t=np.linspace(0,5*tau,1000)
12 u=integ.odeint(f,u0,t)
```

Charge d'un condensateur

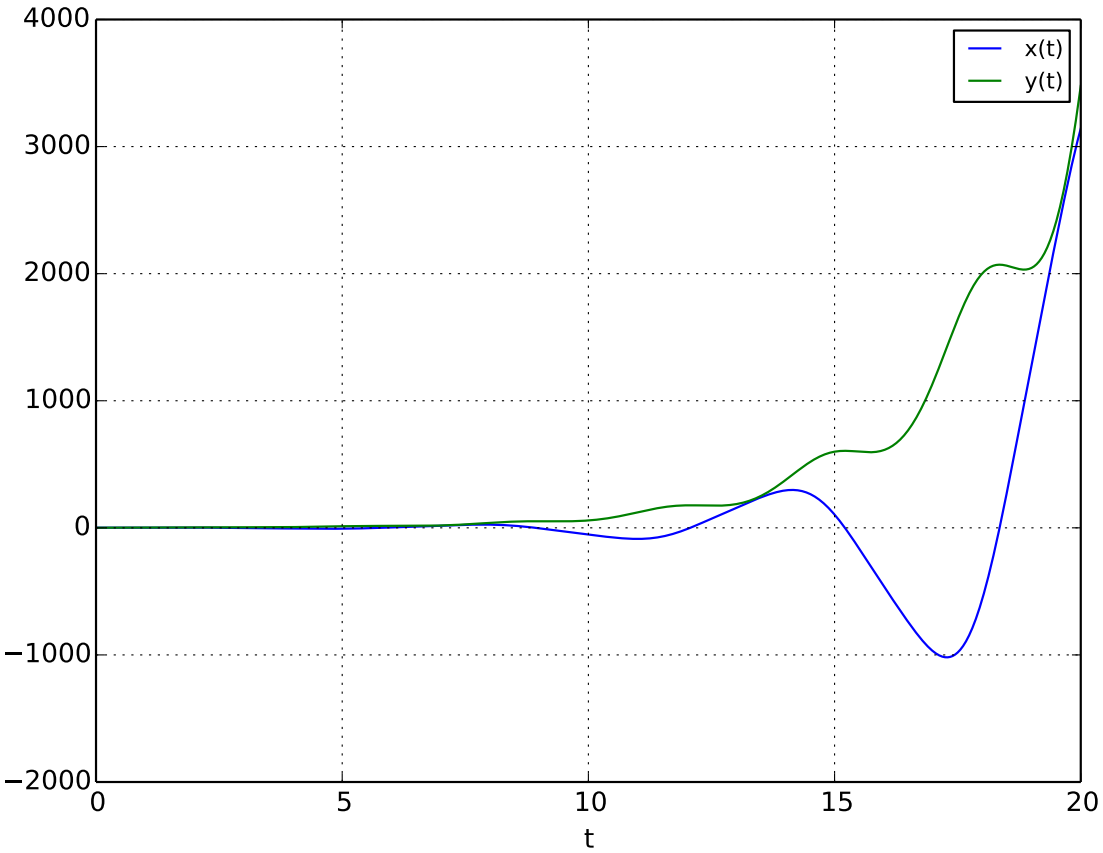


Système non linéaire

$$\begin{cases} x' = \cos(t)y \\ y' = \sin(t)x \end{cases}$$

```
1  def f(sol,t):
2      x=sol[0]
3      y=sol[1]
4
5      rhs_x=np.cos(t)*y
6      rhs_y=np.sin(t)*x
7      return [rhs_x,rhs_y]
8
9  init=[1,1]
10
11 t=np.linspace(0,20,1000000)
12 sol=integ.odeint(f,init,t)
13
14 print(sol)
```

Système non linéaire

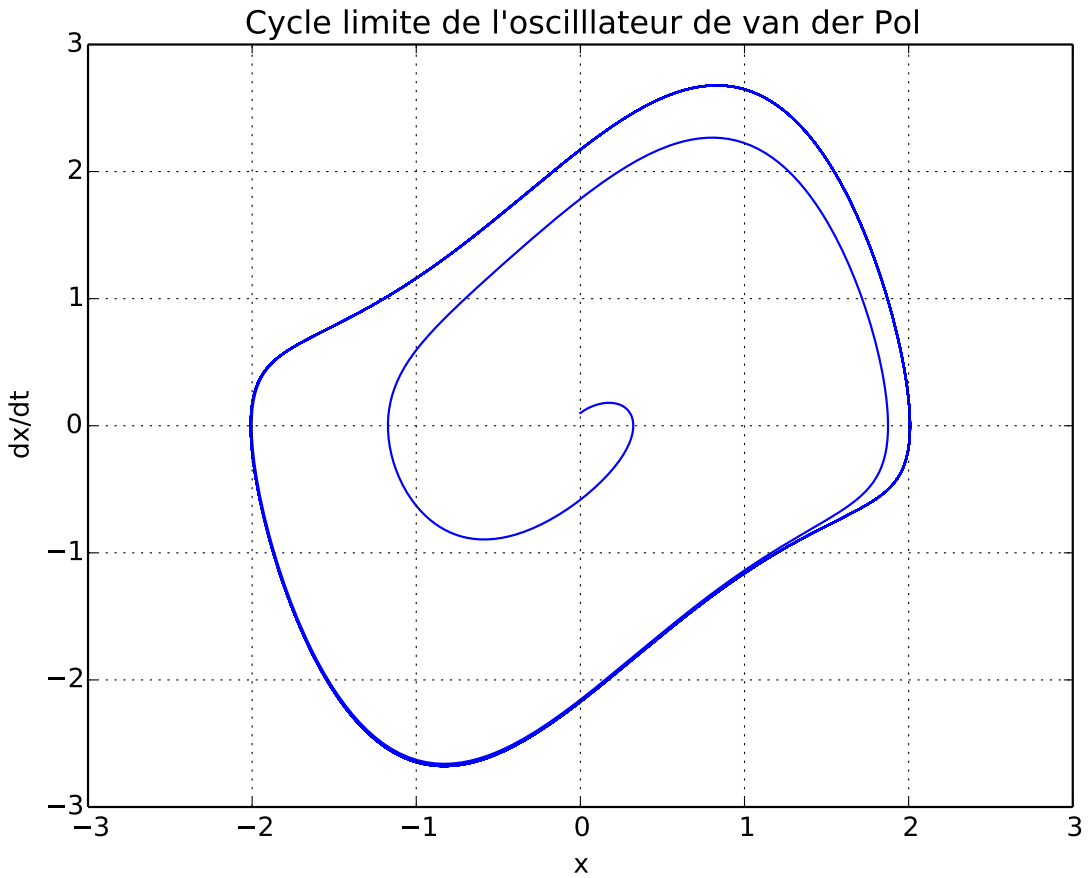


Oscillateur de Van der Pol

$$\ddot{x} = \mu(1 - x^2)\dot{x} - x$$

```
1  def f(sol,t):
2      x=sol[0]
3      xp=sol[1]
4      rhs_x=xp
5      rhs_xp=mu*(1-x**2)*xp-x
6      return [rhs_x,rhs_xp]
7  #RESOLUTION
8  init=[0,0.1]
9  t=np.linspace(0,100,10000)
10 sol=integ.odeint(f,init,t)
11 x=sol[:,0]
12 xp=sol[:,1]
```

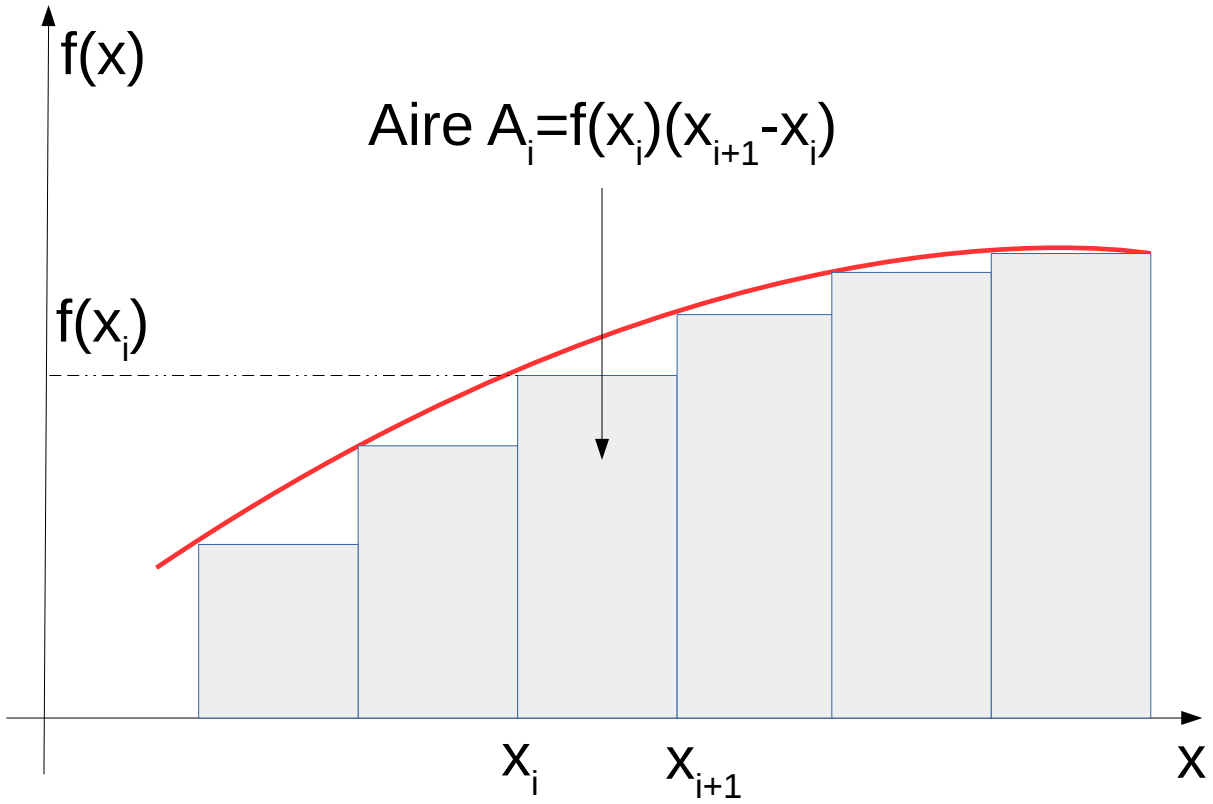
Oscillateur de Van der Pol



Sommaire

Intégration entre deux bornes

Méthodes des rectangles



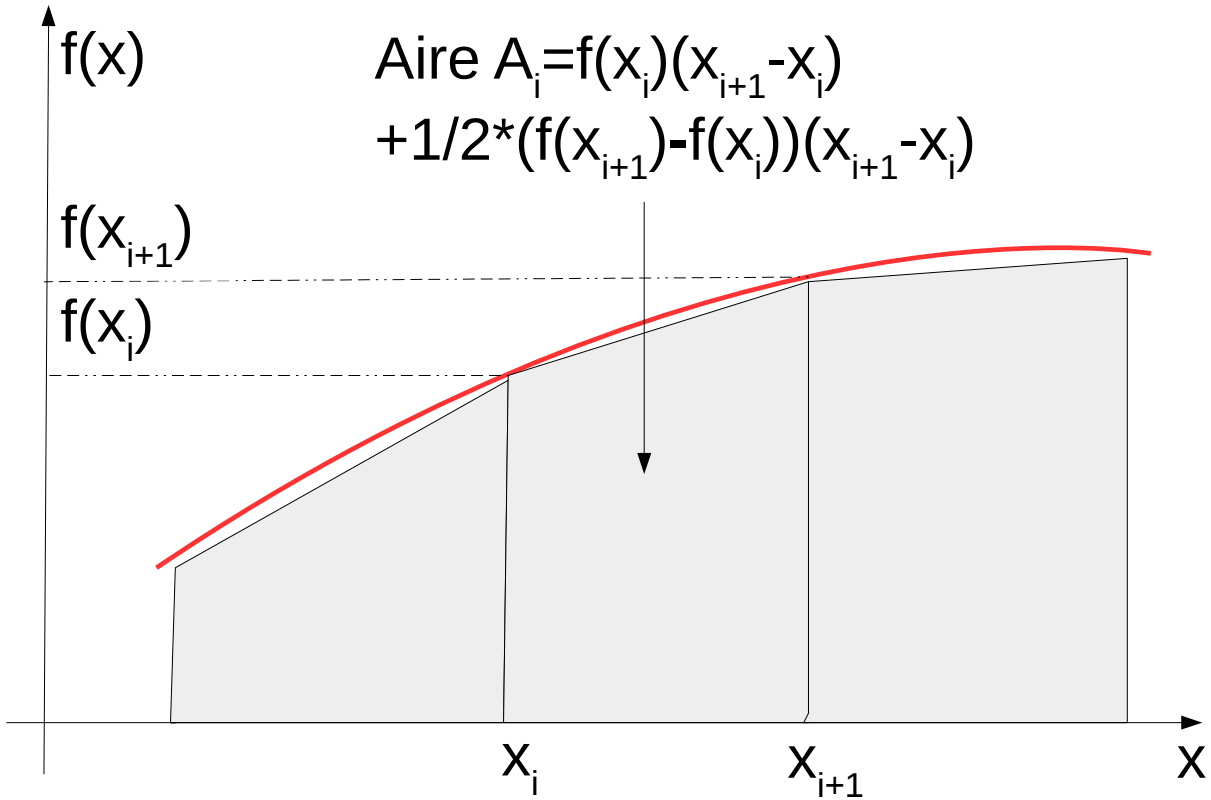
Intégration entre deux bornes

Méthodes des rectangles

```
1  def rect_integ(f,a,b,n):#ordre 0
2      """
3      Methode des rectangles
4      """
5      h=(b-a)/n
6      I=0
7      for i in range(n):
8          I=I+f(a+i*h)*h
9      return I
```

Intégration entre deux bornes

Méthodes des trapèzes



Intégration entre deux bornes

Méthodes des trapèzes

```
1  def trap_integ(f,a,b,n):#ordre 1
2      """
3      Methode des trapezes
4      """
5      h=(b-a)/n
6      I=0
7      for i in range(n):
8          I=I+f(a+i*h)*h+1/2*(f(a+(i+1)*h)-f(a+i*h))*h
9      return I
```

Intégration entre deux bornes

Fonction prédéfinie

Fonction `quad(f,a,b)`

```
1 quad(lambda x:x**2,0,1)
```

Comparatif : `quad` >> `trapeze` >> `rectangle`