OUTILS NUMÉRIQUES

B. Landelle

Table des matières

I Tableaux **2**
1 Généralités 2
2 Arithmétique flottante 4

II Quadrature **5**
1 Principe 5
2 Méthode des rectangles 5

III Équations différentielles **6**
1 Problème de Cauchy 6
2 Premier ordre 7
3 Méthode d'Euler explicite 8
4 Deuxième ordre 9

IV Résolution numérique d'équations **10**
1 Résolution par dichotomie 10
2 Méthode de Newton 13

Pour tout ce qui suit, on importera le module `numpy` sous l'alias `np` :

```
import numpy as np
```

I Tableaux

1 Généralités

Définition 1. *Un tableau est un objet de type* `ndarray` *constitué d'une suite de variables de type entier, flottant ou complexe stockées dans des emplacements consécutifs de la mémoire.*

Proposition 1. *En langage python, on utilise les instructions ou méthodes suivantes sur les tableaux :*
- *l'instruction* `np.array` *pour construire un tableau ;*
- *la méthode* `shape` *pour avoir les dimensions d'un tableau ;*
- *la méthode* `dtype` *pour avoir le type des variables d'un tableau.*

Cette liste est loin d'être exhaustive !

Construction d'un tableau d'entiers à une dimension :

```
>>> a=np.array([1,2,3])
>>> type(a)
<class 'numpy.ndarray'>
>>> a.shape
(3,)
>>> a.dtype
dtype('int32')
```

Construction d'un tableau de flottants à deux dimensions :

```
>>> b=np.array([[1,2],[1/3,1/4],[0,1]])
>>> b
array([[ 1.        ,  2.        ],
       [ 0.33333333,  0.25      ],
       [ 0.        ,  1.        ]])
>>> b.shape
(3, 2)
>>> b.dtype
dtype('float64')
```

Proposition 2. *L'accès en lecture et en écriture à une case d'un tableau est de complexité temporelle en $O(1)$.*

Ces performances en lecture et écriture sont possibles grâce à l'organisation des données dans des emplacements consécutifs en mémoire.

L'accès aux composantes d'un tableau et le recours au slicing suit les mêmes règles que celles énoncées sur les listes :

```
>>> a=np.array([1,2,3,4,5])
>>> a[0]
1
>>> a[1:]
array([2, 3, 4, 5])
>>> a[::2]
array([1, 3, 5])
```

et sur un tableau à deux dimensions :

```
>>> b=np.array([[1,2],[1/3,1/4],[0,1]])
>>> b[0,:]
array([ 1.,  2.])
>>> b[-1,:]
array([ 0.,  1.])
>>> b[:,0]
array([ 1.        ,  0.33333333,  0.        ])
```

L'instruction `b[0,:]` renvoie la première ligne, l'instruction `b[-1,:]` renvoie la dernière ligne, l'instruction `b[:,0]` renvoie la première colonne.

Un tableau est un type mutable.

```
>>> a=np.array([3,2,1])
>>> b=a
>>> a[0]=0
>>> a
array([0, 2, 1])
>>> b
array([0, 2, 1])
```

Pour créer une copie indépendante, on effectue une *copie en profondeur* avec l'instruction `deepcopy` du module `copy` ou avec `np.array`.

```
>>> from copy import deepcopy
>>> a=np.array([3,2,1])
>>> b=deepcopy(a)
>>> c=np.array(a)
>>> a[0]=0
>>> a
array([0, 2, 1])
>>> b
array([3, 2, 1])
>>> c
array([3, 2, 1])
```

L'instruction `np.linspace(a,b,n)` génère un tableau de n valeurs régulièrement espacées de a à b et l'instruction `np.arange(a,b,h)` génère un tableau de valeurs régulièrement espacées de h à a inclus à b exclu.

```
>>> np.linspace(0,1,4)
array([ 0.        ,  0.33333333,  0.66666667,  1.        ])
>>> np.arange(0,1,.2)
array([ 0. ,  0.2,  0.4,  0.6,  0.8])
```

Les instructions `list` et `np.array` permettent les conversions respectivement vers le type liste ou le type tableau.

```
>>> np.array([2*k+3 for k in range(10)])
array([ 3,  5,  7,  9, 11, 13, 15, 17, 19, 21])
>>> list(np.arange(0,1,.2))
[0.0, 0.20000000000000001, 0.40000000000000002, 0.60000000000000009,
0.80000000000000004]
```

2 Arithmétique flottante

En langage python, les nombres sont codés au format flottant. Tester l'égalité entre deux flottants ou tester la nullité d'un flottant n'est pas pertinent du fait de l'imprécision liée au format. Il faut donc effectuer des tests avec un certain seuil de tolérance :

```
def floatnull(x):
    eps=1e-8
    return abs(x)<eps
```

Pour tester l'égalité entre deux tableaux à une dimension de même taille constitués de flottants, on peut coder :

```
def arrayfloateq(T1,T2):
    eps=1e-8
    for k in range(len(T1)):
        if abs(T1[k]-T2[k])>eps:
            return False
    return True
```

On peut utiliser un procédé de *seuillage* pour transformer les nombres proches de zéro (en valeur absolue) en zéro. Par exemple, sur un tableau à une dimension, on pourrait procéder ainsi :

```
def seuil(x):
    eps=1e-8
    return x*(abs(x)>eps) # True confondu avec 1 pour l'opération *

def round1(a):
    n=len(a)
    res=[]
    for k in range(n):
        res.append(seuil(a[k]))
    return res
```

On obtient :

```
>>> a=np.sin([k*np.pi/2 for k in range(6)])
>>> a
array([ 0.00000000e+00,   1.00000000e+00,   1.22464680e-16,
       -1.00000000e+00,  -2.44929360e-16,   1.00000000e+00])
>>> round1(a)
array([ 0.,  1.,  0., -1., -0.,  1.])
```

II Quadrature

1 Principe

Définition 2. *Une* méthode de quadrature *sur* $\mathrm{E} = \mathscr{C}^0([\,a\,;b\,],\mathbb{R})$ *consiste en le choix de* poids $\lambda_0, \ldots, \lambda_{p-1}$ *réels et de* nœuds $x_0, \ldots, x_{p-1}$ *dans* $[\,a\,;b\,]$ *et strictement ordonnés tels que, pour* $f \in \mathrm{E}$*, le calcul de la somme finie* $\sum_{i=0}^{p-1} \lambda_i f(x_i)$ *fournisse une valeur approchée de* $\int_a^b f(t)\,\mathrm{d}t$*, c'est-à-dire*

$$\int_a^b f(t)\,\mathrm{d}t \simeq \sum_{i=0}^{p-1} \lambda_i f(x_i)$$

Remarque : La définition peut sembler un peu creuse puisque le sens de *valeur approchée* n'est pas définie ...

2 Méthode des rectangles

Dans cette section, la méthode présentée s'applique avec une subdivision $(a_k)_{0\leqslant k\leqslant n}$ de $[\,a\,;b\,]$ régulièrement espacée :

$$\forall k \in [\![\,0\,;\,n\,]\!] \qquad a_k = a + kh \quad \text{avec} \quad h = \frac{b-a}{n}$$

Sur chaque intervalle $[\,a_k\,;a_{k+1}\,]$ avec $k \in [\![\,0\,;\,n-1\,]\!]$, on utilise une méthode de quadrature simple pour approcher $\int_{a_k}^{a_{k+1}} f(t)\,\mathrm{d}t$.

Définition 3. *Soit* $f \in \mathrm{E}$*. La* méthode des rectangles *consiste à approcher* $\int_a^b f(t)\,\mathrm{d}t$ *par la somme*

$$h\sum_{k=0}^{n-1} f(a_k) \simeq \int_a^b f(t)\,\mathrm{d}t$$

La quantité $h \times f(a_k)$ représente l'aire d'un rectangle de base $[\,a_k\,;a_{k+1}\,]$ et de hauteur $f(a_k)$.

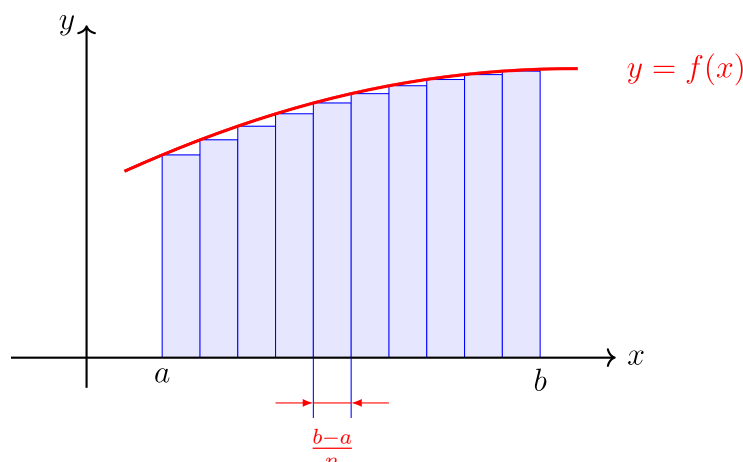

FIGURE 1 – Méthode des rectangles

Traditionnellement, la méthode des rectangles s'entend au sens des *rectangles à gauche* comme ci-dessus. La *méthode des rectangles à droite* consiste à effectuer le calcul $h\sum_{k=1}^{n} f(a_k)$.

```
def rect(f,a,b,n):
    """Méthode des rectangles à gauche"""
    res=0
    h=(b-a)/n
    c=a
    for k in range(n):
        res+=f(c)
        c+=h
    return res*h
```

III Équations différentielles

Pour des résolutions numériques d'équations différentielles, on importera le module `scipy.integrate` sous l'alias `integr` :

```
import scipy.integrate as integr
```

1 Problème de Cauchy

Un *problème de Cauchy* associé à une équation différentielle d'ordre 1 est un système de la forme

$$\begin{cases} x'(t) = f(x(t), t) \\ x(t_0) = x_0 \end{cases}$$

L'équation différentielle considérée est sous forme *normalisée* avec le terme $x'(t)$ explicite en fonction de $x(t)$ et t. Sous certaines hypothèses, le *théorème de Cauchy-Lipschitz* garantit qu'il existe une unique solution à ce problème. La détermination formelle de cette solution est souvent impossible et on privilégie donc la recherche d'une solution numérique approchée.

On utilise l'instruction `integr.odeint` pour effectuer une résolution numérique de ce problème de Cauchy. On résout l'équation sur un intervalle de temps discrétisé sous la forme d'un tableau ou d'une liste $[t_0, \dots, t_n]$ avec la syntaxe suivante :

```
integr.odeint(f,x0,tt)
```

où `x0` désigne la condition initiale à l'instant t_0, premier élément de la liste `tt`. L'instruction renvoie un tableau $[x_0, \dots, x_n]$, solution approchée de $[x(t_0), \dots, x(t_n)]$.

La précision des solutions approchées fournies par `integr.odeint` dépend de la liste des temps discrétisés. Dans l'ensemble, cette précision est remarquable. L'instruction s'appuie sur les méthodes d'Adams-Moulton et BDF (*Backward Differentiation Formula*). Le lecteur curieux pourra consulter les articles [1] et [2]. L'instruction `integr.odeint` s'appuie sur la librairie Fortran intitulée Odepack (voir [3], [4]).

2 Premier ordre

Pour résoudre numériquement le problème de Cauchy

$$\begin{cases} x'(t) = x(t) \\ x(0) = 1 \end{cases}$$

sur l'intervalle $[\,0\,;5\,]$, on saisit :

```
def f(x,t):
    return x

tt=np.linspace(0,5,100);x0=1    # intervalle discrétisé, condition initiale
tx=integr.odeint(f,x0,tt)       # résolution numérique de l'équation
plt.plot(tt,tx);plt.grid();plt.show()
```

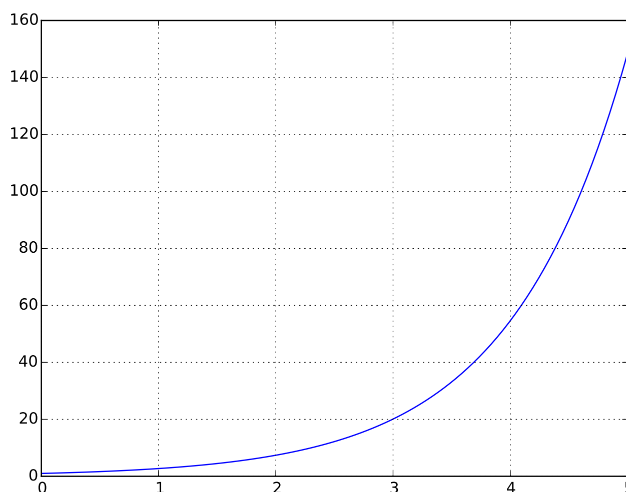

Figure 2 – Solution de $x' = x$ avec $x(0) = 1$

Pour tracer plusieurs courbes intégrales correspondant à des conditions initiales distinctes, on saisit :

```
for x0 in np.linspace(.5,1.5,10):
    sol=integr.odeint(f,x0,tt)
    plt.plot(tt,sol)
plt.grid();plt.show()
```

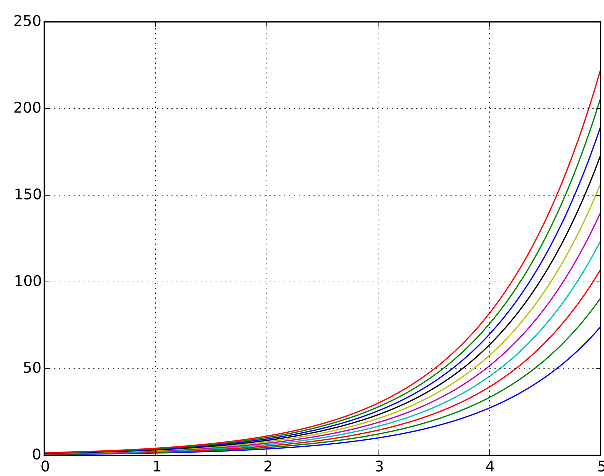

FIGURE 3 – Courbes intégrales

Ces courbes ne se rencontrent pas, conformément à ce qu'annonce le théorème de Cauchy linéaire.

3 Méthode d'Euler explicite

La relation entre les états aux instants t et $t+h$ est

$$x(t+h) = x(t) + \int_t^{t+h} x'(s)\,\mathrm{d}s = x(t) + \int_t^{t+h} f(x(s),s)\,\mathrm{d}s$$

Si h est « petit », la variation de $t \mapsto f(x(t),t)$ est faible et le principe de la méthode d'Euler explicite consiste à réaliser l'approximation

$$\forall s \in [\,t\,;t+h\,] \qquad f(x(s),s) \simeq f(x(t),t)$$

d'où

$$\boxed{x(t+h) \simeq x(t) + hf(x(t),t)}$$

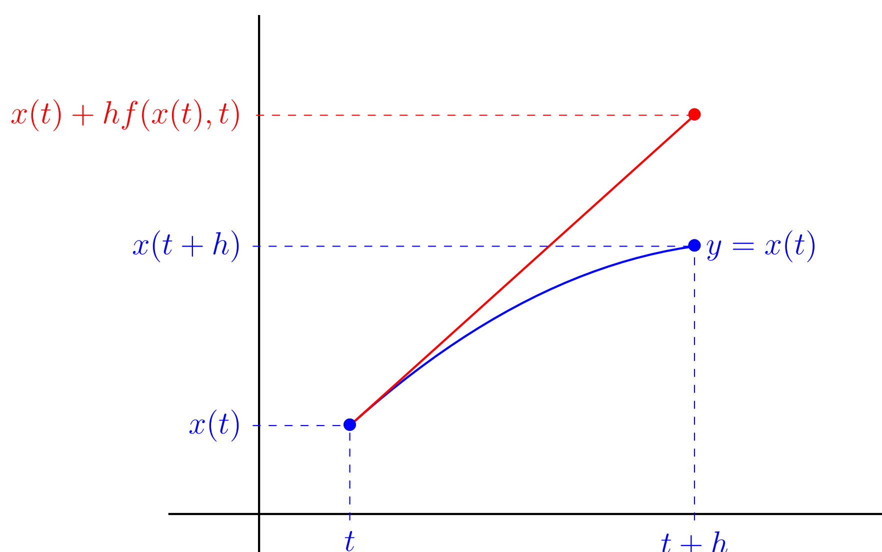

FIGURE 4 – Schéma d'Euler explicite

Remarque : La méthode d'Euler explicite consiste à approcher l'intégrale $\displaystyle\int_t^{t+h} f(x(s),s)\,\mathrm{d}s$ par la méthode des rectangles à gauche.

La suite $(x_0,\ldots,x_n)$ solution approchée de $(x(t_0),\ldots,x(t_n))$ par la méthode d'Euler explicite est définie par

$$\forall k \in [\![\, 1\,;\, n \,]\!] \qquad x_k = x_{k-1} + h_{k-1} f(x_{k-1},t_{k-1}) \quad \text{avec} \quad h_{k-1} = t_k - t_{k-1}$$

```
def Euler(f,x0,t):
    x=[x0]
    for k in range(1,len(t)):
        h=t[k]-t[k-1]
        x.append(x[k-1]+h*f(x[k-1],t[k-1]))
    return x
```

Remarques : (1) Cette méthode est dite *explicite* car la quantité $f(x_{k-1},t_{k-1})$ à calculer à la k-ème itération est complètement explicite : t_{k-1} est une donnée du problème et x_{k-1} a été obtenu à l'étape précédente.

(2) On peut démontrer que sous certaines hypothèses sur la fonction f (continue, lipschitzienne en la première variable qui peut même être assouplie en localement lipschitzienne), le schéma d'Euler est convergent.

4 Deuxième ordre

Pour des équations différentielles d'ordre supérieur à 1, l'écriture matricielle permet de se ramener à un système différentielle d'ordre 1. Dans le cas du problème de Cauchy suivant

$$\begin{cases} x''(t) = \mathrm{F}(x'(t),x(t),t) \\ (x(t_0),x'(t_0)) = (x_0,v_0) \end{cases}$$

On pose
$$\mathrm{X}(t) = \begin{pmatrix} x(t) \\ x'(t) \end{pmatrix} \quad \text{et} \quad \mathrm{X}_0 = \begin{pmatrix} x_0 \\ v_0 \end{pmatrix}$$

puis on trouve
$$\mathrm{X}'(t) = \begin{pmatrix} x'(t) \\ x''(t) \end{pmatrix} = \begin{pmatrix} x'(t) \\ \mathrm{F}(x'(t),x(t),t) \end{pmatrix} = f(\mathrm{X}(t),t)$$

et on peut alors utiliser `integr.odeint` pour traiter le problème de Cauchy associé à une équation différentielle matricielle d'ordre 1 résolue suivant :

$$\begin{cases} \mathrm{X}'(t) = f(\mathrm{X}(t),t) \\ \mathrm{X}(t_0) = \mathrm{X}_0 \end{cases}$$

Par exemple, pour résoudre numériquement le problème de Cauchy

$$\begin{cases} x''(t) + x'(t) + x(t) = \sin t \\ (x(0),x'(0)) = (1,1) \end{cases} \iff \begin{cases} (x'(t),x''(t)) = (x'(t),-x(t)-x'(t)+\sin t) \\ (x(0),x'(0)) = (1,1) \end{cases}$$

sur l'intervalle $[\,0\,;10\,]$, on saisit :

```
def f(X,t):
    return [X[1],-X[0]-X[1]+np.sin(t)]

tt=np.linspace(0,10,100);X0=[1,1]
tX=integr.odeint(f,X0,tt)
plt.plot(tt,tX[:,0])    # tracé de t->x(t) première coordonnée de X
plt.grid();plt.show()
```

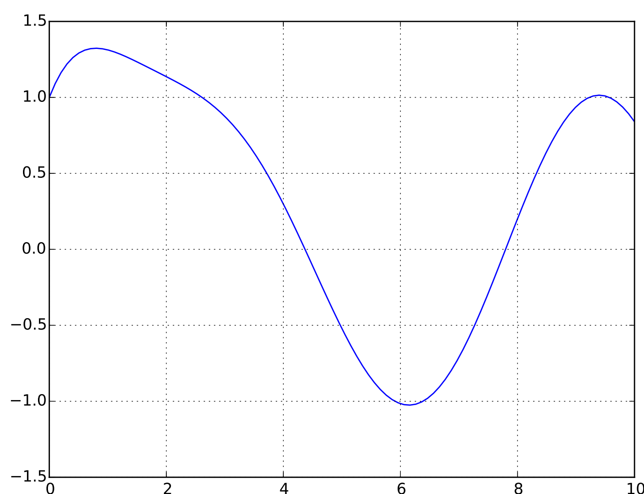

FIGURE 5 – Solution de $x'' + x' + x = \sin t$ avec $(x(0), x'(0)) = (1,1)$

IV Résolution numérique d'équations

Dans cette section, on présente différente méthodes de résolution numérique de l'équation $f(x) = 0$.

1 Résolution par dichotomie

Soit $f \in \mathscr{C}^0([\,a\,;b\,],\mathbb{R})$ avec $f(a)f(b) \leqslant 0$ ce qui garantit l'existence d'un réel $\alpha \in [\,a\,;b\,]$ tel que $f(\alpha) = 0$. L'algorithme consiste à regarder la valeur de f en le milieu $c = \dfrac{a+b}{2}$ et en fonction du signe de $f(c)$, à considérer comme nouvel intervalle $[\,a\,;c\,]$ ou $[\,c\,;b\,]$ puis à répéter cette démarche. Ainsi, à chaque itération, la longueur de l'intervalle est divisée par deux et va donc encadrer de plus en plus finement la valeur d'une racine.

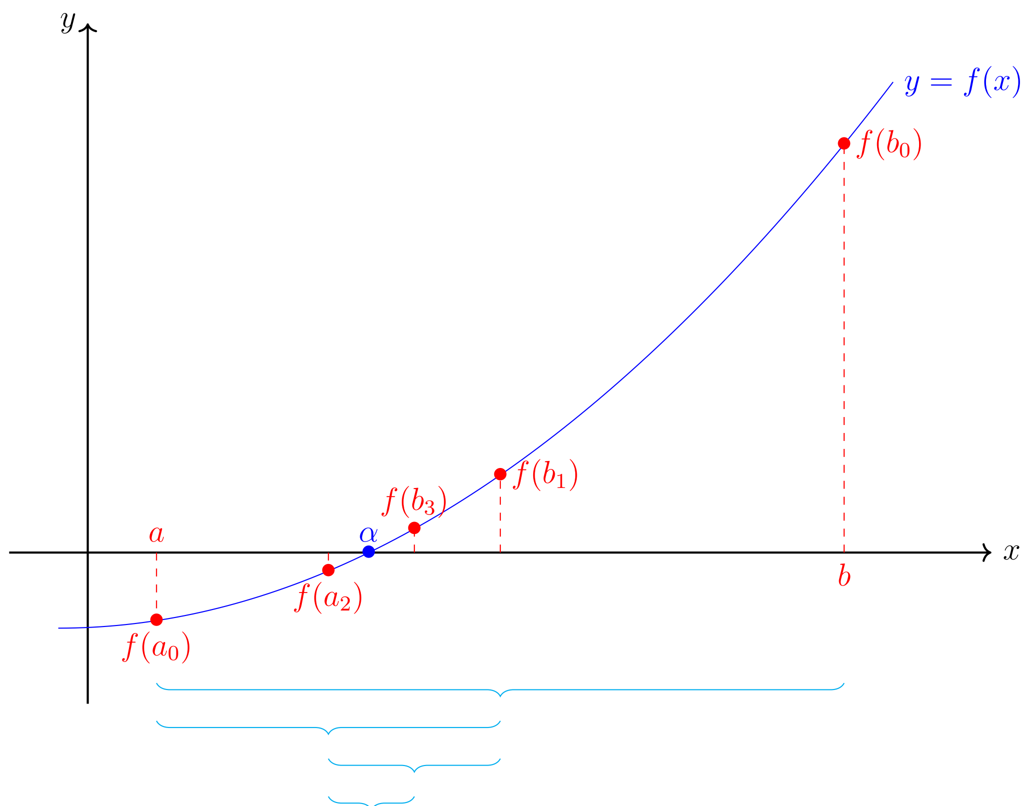

Figure 6 – Dichotomie

Code :

```
def dicho(f,a,b,eps):
    deb,fin=a,b
    milieu=(deb+fin)/2
    while fin-deb>eps:
        if f(milieu)*f(deb)<=0:
            fin=milieu
        else:
            deb=milieu
        milieu=(deb+fin)/2
    return milieu
```

ou récursivement :

```
def dicho(f,a,b,eps):
    c=(a+b)/2
    if b-a<eps:
        return c
    else:
        if f(a)*f(c)<=0:
            return dicho(f,a,c,eps)
        else:
            return dicho(f,c,b,eps)
```

Expérimentation : On présente l'utilisation de `dicho` pour la résolution de l'équation

$$x^2 - 2 = 0 \quad \text{avec} \quad x \in [\,0\,;2\,]$$

```
>>> dicho(lambda t:t**2-2,0,2,1e-10)
1.414213562355144
>>> np.sqrt(2)
1.4142135623730951
```

La commande `bisect` du module `scipy.optimize`, habituellement importé sous l'alias `resol`, est une implémentation de la méthode de résolution par dichotomie :

```
>>> import scipy.optimize as resol
>>> f=lambda x : x**2-2
>>> resol.bisect(f,0,2)
1.4142135623715149
```

Définition 4. *Soit $f \in \mathscr{C}^0([\,a\,;b\,],\mathbb{R})$ avec $f(a)f(b) \leqslant 0$. L'algorithme de* dichotomie *consiste en la construction des suites $(a_n)_n$, $(b_n)_n$ et $(c_n)_n$ avec $a_0 = a$, $b_0 = b$ et pour tout entier n*

$$c_n = \frac{a_n + b_n}{2} \qquad (a_{n+1}, b_{n+1}) = \begin{cases} (a_n, c_n) & \text{si } f(a_n)f(c_n) \leqslant 0 \\ (c_n, b_n) & \text{sinon} \end{cases}$$

Proposition 3. *Soit $f \in \mathscr{C}^0([\,a\,;b\,],\mathbb{R})$ avec $f(a)f(b) \leqslant 0$ et $(a_n)_n$, $(b_n)_n$ et $(c_n)_n$ les suites de l'algorithme de dichotomie. On a l'invariant de boucle suivant :*

$$\forall n \in \mathbb{N} \qquad f(a_n)f(b_n) \leqslant 0$$

Démonstration. Récurrence immédiate. □

Commentaire : Ceci garantit pour tout n entier, l'existence d'une racine de f dans $[\,a_n\,;b_n\,]$. Pour un seuil $\varepsilon > 0$, on retourne c_n lorsque $b_n - a_n \leqslant \varepsilon$. Ainsi, on est assuré d'avoir une racine α de f telle que

$$|c_n - \alpha| \leqslant b_n - a_n \leqslant \varepsilon$$

Proposition 4. *Soit $f \in \mathscr{C}^0([\,a\,;b\,],\mathbb{R})$ vérifiant $f(a)f(b) \leqslant 0$ et $(a_n)_n$, $(b_n)_n$ les suites de l'algorithme de dichotomie. On a*

$$\forall n \in \mathbb{N} \qquad b_n - a_n = \frac{b-a}{2^n}$$

et

$$n \geqslant \log_2\left(\frac{b-a}{\varepsilon}\right) \implies b_n - a_n \leqslant \varepsilon$$

Démonstration. L'égalité sur $b_n - a_n$ se montre par récurrence et disjonction de cas. Le reste suit sans difficulté. □

2 Méthode de Newton

Soit une fonction $f : \mathrm{I} \to \mathbb{R}$ de classe $\mathscr{C}^1$ avec I un intervalle non vide de $\mathbb{R}$ sur lequel f s'annule et une valeur initiale x_0. L'idée de la méthode de Newton consiste à « descendre » le long de la tangente, autrement dit à approcher la courbe par sa tangente et considérer sa racine.

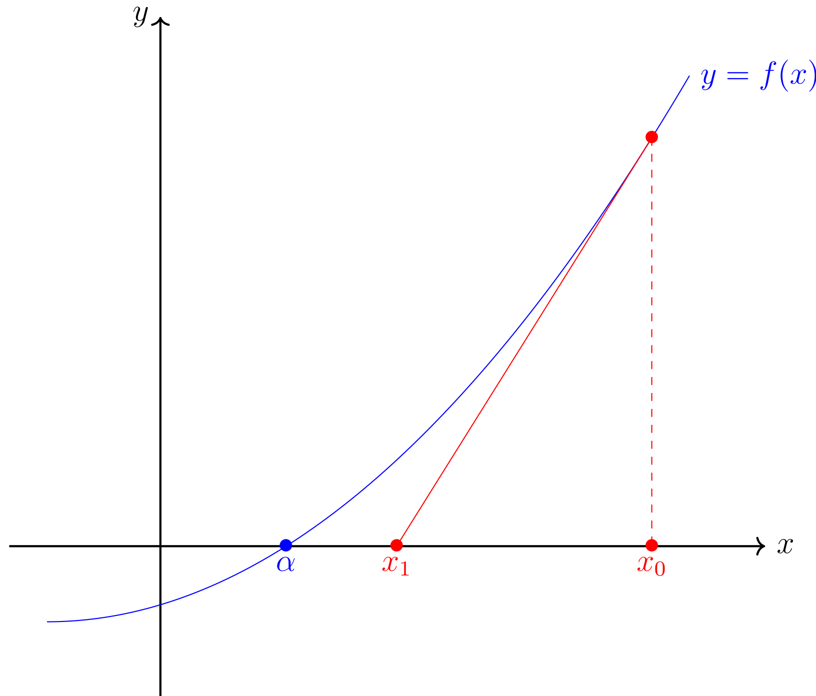

FIGURE 7 – Descente le long de la tangente

L'équation de la tangente à la courbe a pour équation $y = f'(x_0)(x - x_0) + f(x_0)$ qui admet pour racine

$$y = f'(x_0)(x - x_0) + f(x_0) \iff x = x_0 - \frac{f(x_0)}{f'(x_0)}$$

En itérant ce procédé, on définit la suite $(x_n)_{n\in\mathbb{N}}$ par

$$\boxed{\forall n \in \mathbb{N} \qquad x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}}$$

Définition 5. *Soit $f \in \mathscr{C}^1(\mathrm{I}, \mathbb{R})$ telle que f' ne s'annule pas sur* I *et $x_0 \in \mathrm{I}$. La suite $(x_n)_n$ définie par*

$$\forall n \in \mathbb{N} \qquad x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

est appelée suite de la méthode de Newton.

Sous certaines conditions, notamment si la condition initiale x_0 n'est pas trop éloignée de la racine recherchée, la suite converge vers cette racine. On prend comme condition d'arrêt au calcul itératif de la suite $(x_n)_{n\in\mathbb{N}}$ le test $|x_{n+1} - x_n| \leqslant \varepsilon$ où $\varepsilon > 0$ désigne un seuil choisi par l'utilisateur.

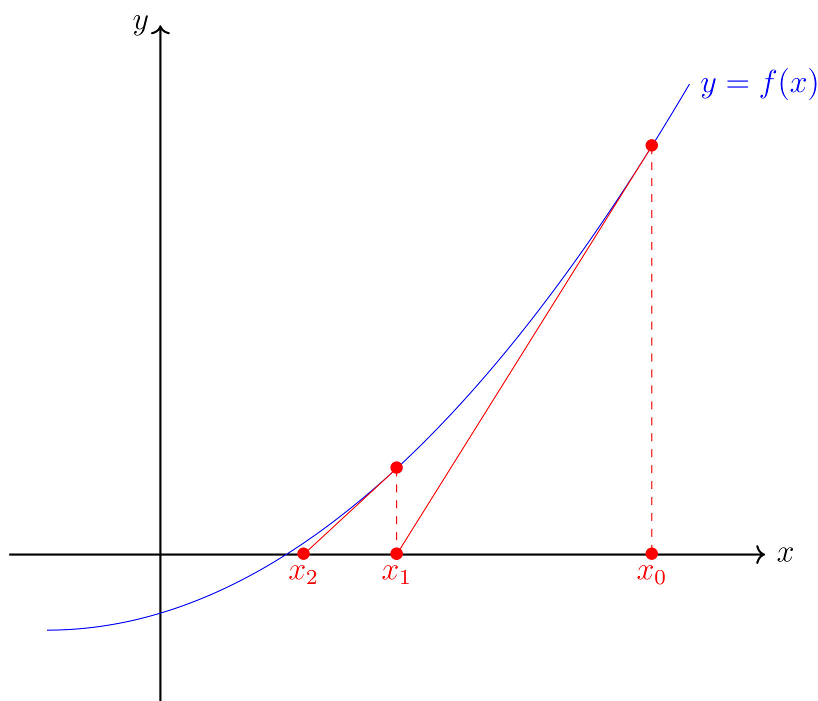

FIGURE 8 – Méthode de Newton

Code :

```
def newton(x0,f,df,eps):
    a,b=x0,x0-f(x0)/df(x0)
    while abs(b-a)>eps:
        a,b=b,b-f(b)/df(b)
    return b
```

Expérimentation : On présente l'utilisation de `newton` pour la résolution de l'équation

$$x^2 - 2 = 0 \quad \text{avec} \quad x_0 = 2$$

```
>>> newton(2,lambda t:t**2-2,lambda t:2*t,1e-10)
1.4142135623730951
>>> np.sqrt(2)
1.4142135623730951
```

Théorème 1. *Soit $f \in \mathscr{C}^2(\mathrm{I},\mathbb{R})$ et $\alpha \in \mathrm{I}$ tel que $f(\alpha) = 0$ et $f'(\alpha) \neq 0$. Alors, il existe un voisinage $\mathscr{V}$ de α tel que pour $x_0 \in \mathscr{V}$, la suite $(x_n)_n$ de la méthode de Newton converge vers α à vitesse quadratique, à savoir*

$$\forall n \in \mathbb{N} \qquad \mathrm{C}\,|x_n - \alpha| \leqslant (\mathrm{C}\,|x_0 - \alpha|)^{2^n} \qquad \textit{avec} \quad 0 \leqslant \mathrm{C}\,|x_0 - \alpha| < 1$$

Remarque : La méthode de Newton présente une vitesse de convergence exceptionnelle, bien meilleure que celle de la méthode de dichotomie. Avec $\delta_n = -\log_{10}(e_n)$ où $e_n = \dfrac{1}{\mathrm{C}}\,(\mathrm{C}\,|x_0 - \alpha|)^{2^n}$, on a $\delta_{n+1} \simeq 2\delta_n$ ce qu'on interprète (abusivement) comme le doublement du nombre de décimales communes entre x_n et α à chaque itération.

Corollaire 1. *Soit $f \in \mathscr{C}^2(\mathrm{I}, \mathbb{R})$, convexe avec f' ne s'annulant pas et $\alpha \in \mathrm{I}$ tel que $f(\alpha) = 0$. La suite $(x_n)_n$ de la méthode de Newton converge vers α avec une vitesse quadratique à partir d'un certain rang.*

Piège cyclique : Considérons la fonction f définie par

$$\forall x \in \mathbb{R} \qquad f(x) = x^3 - 2x + 2$$

et une valeur initiale $x_0 = 0$. La méthode de Newton boucle indéfiniment sur cette configuration. La tangente en 0 est $y = -2x + 2$ d'où $x_1 = 1$. Puis la tangente en 1 est $y = (x - 1) + 1 = x$ d'où $x_2 = 0$. Par récurrence immédiate, on obtient donc

$$\forall n \in \mathbb{N} \qquad x_{2n} = 0 \qquad x_{2n+1} = 1$$

L'unique racine réelle de f est $\alpha \simeq -1.77$. La convergence de la méthode de Newton est un résultat local pour une valeur initiale proche de la racine. Quand cette condition n'est pas satisfaite comme c'est le cas ici, la convergence n'est plus assurée.

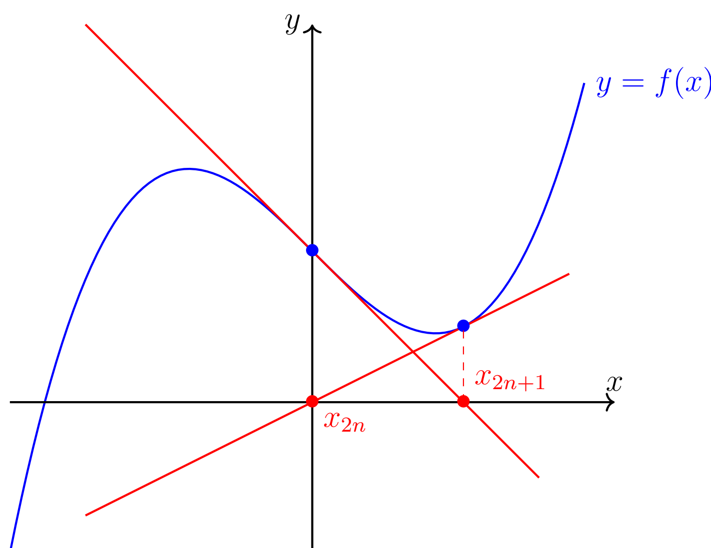

FIGURE 9 – Graphe de f et des tangentes en 0 et 1

Pour éviter ce problème lié au choix de la condition initiale, il est courant d'associer la méthode de Newton à une méthode d'encadrement comme la dichotomie par exemple.

Un exemple célèbre : La méthode de Héron (premier siècle après JC).
Pour calculer une valeur approchée de $\sqrt{a}$ avec $a > 0$, la méthode de Héron consiste à utiliser la suite $(x_n)_n$ définie par

$$x_0 = a \quad \text{et} \quad \forall n \in \mathbb{N} \qquad x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

La méthode de Héron est un cas particulier d'utilisation de la méthode de Newton avec une convergence globale quadratique.

Références

[1] A. C. Hindmarsh, *ODEPACK, A Systematized Collection of ODE Solvers*, IMACS Transactions on Scientific Computation, vol.1 pp. 55-64, R. S. Stepleman et al., 1983

[2] K. Radhakrishnan and A. C. Hindmarsh, *Description and Use of LSODE, the Livermore Solver for Ordinary Differential Equations*, LLNL report UCRL-ID-113855, December 1993

[3] Netlib repository of numerical software, `http://www.netlib.org`

[4] Hans Petter Langtangen, *Python Scripting for Computational Science*, Texts in Computational Science and Engineering, Springer-Verlag, 2005