INGÉNIERIE NUMÉRIQUE

B. Landelle

Table des matières

I Quadrature

1 Principe

Définition 1. *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 2. *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 $f(a_k)$.

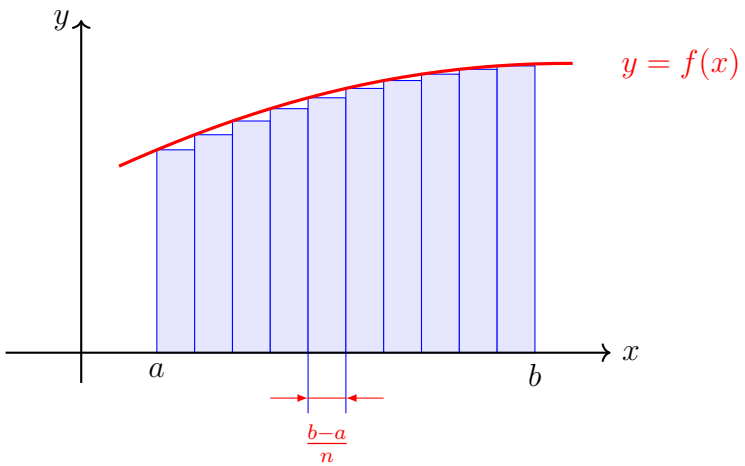

FIGURE 1 – Méthode des rectangles

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
```

Exercice : Écrire une implémentation de la méthode des *rectangles médians* qui consiste à effectuer le calcul $h\sum_{k=1}^{n} f\left(\frac{a_k + a_{k+1}}{2}\right)$.

Corrigé : On saisit :

```
def rect(f,a,b,n):
    """Méthode des rectangles médians"""
    res=0
    h=(b-a)/n
    c=a+h/2
    for k in range(n):
        res+=f(c)
        c+=h
    return res*h
```

II 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 $[t_0, \ldots, t_n]$ avec la syntaxe suivante :

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

où `x0` désigne la condition initiale à l'instant t_0, premier élément de la liste `t`. L'instruction renvoie un tableau $[x_0, \ldots, x_n]$, solution approchée de $[x(t_0), \ldots, x(t_n)]$.

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 [3], [4] et l'ouvrage [8]. L'instruction `integr.odeint` s'appuie sur la librairie FORTRAN intitulée ODEPACK (voir [1], [2]).

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()
```

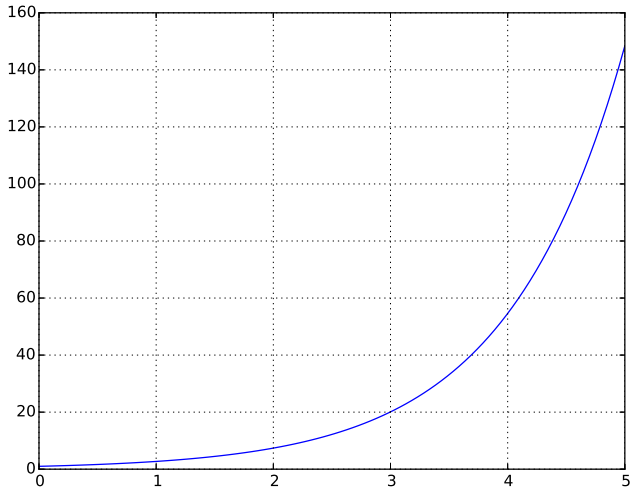

FIGURE 2 – Solution de $x' = x$ avec $x(0) = 1$

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()
```

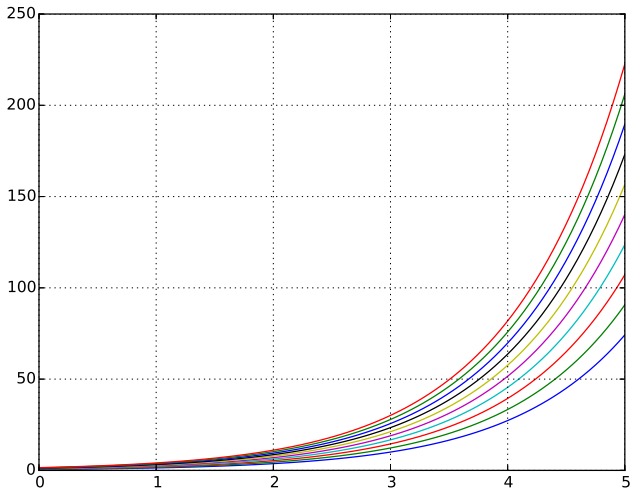

FIGURE 3 – Courbes intégrales

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

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

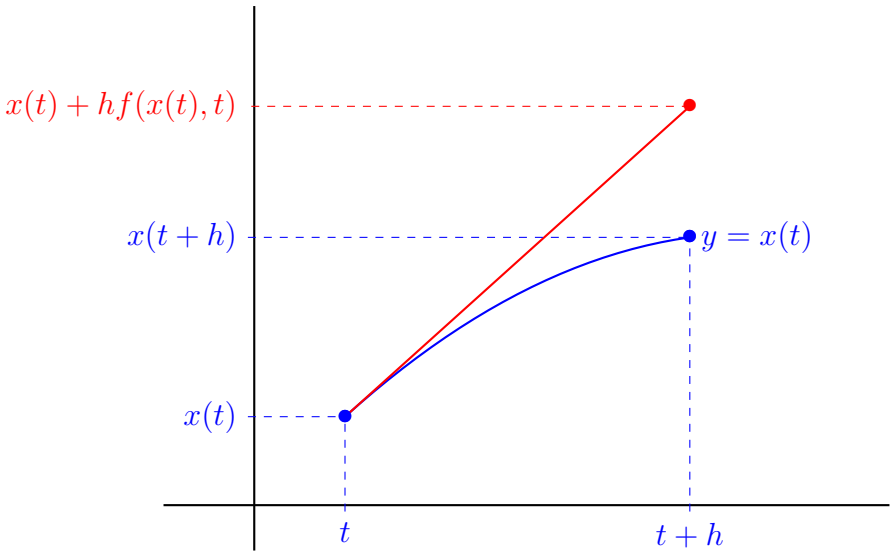

FIGURE 4 – Schéma d'Euler explicite

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 $(x_0,\dots,x_n)$ solution approchée de $(x(t_0),\dots,x(t_n))$ 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()
```

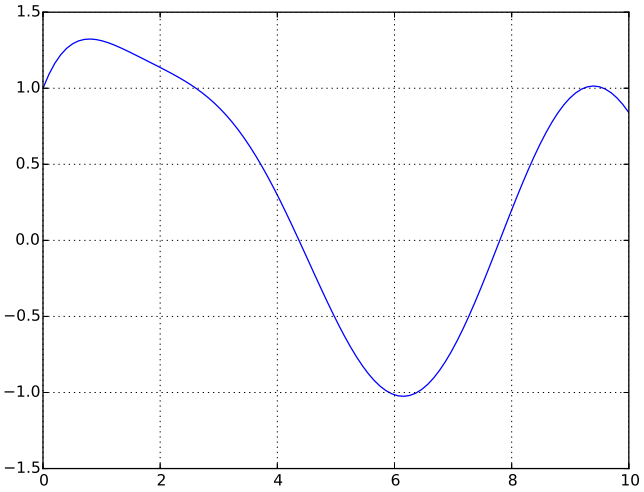

FIGURE 5 – Solution de $x''+x'+x=\sin t$ avec $(x(0),x'(0))=(1,1)$

III 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 de 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.

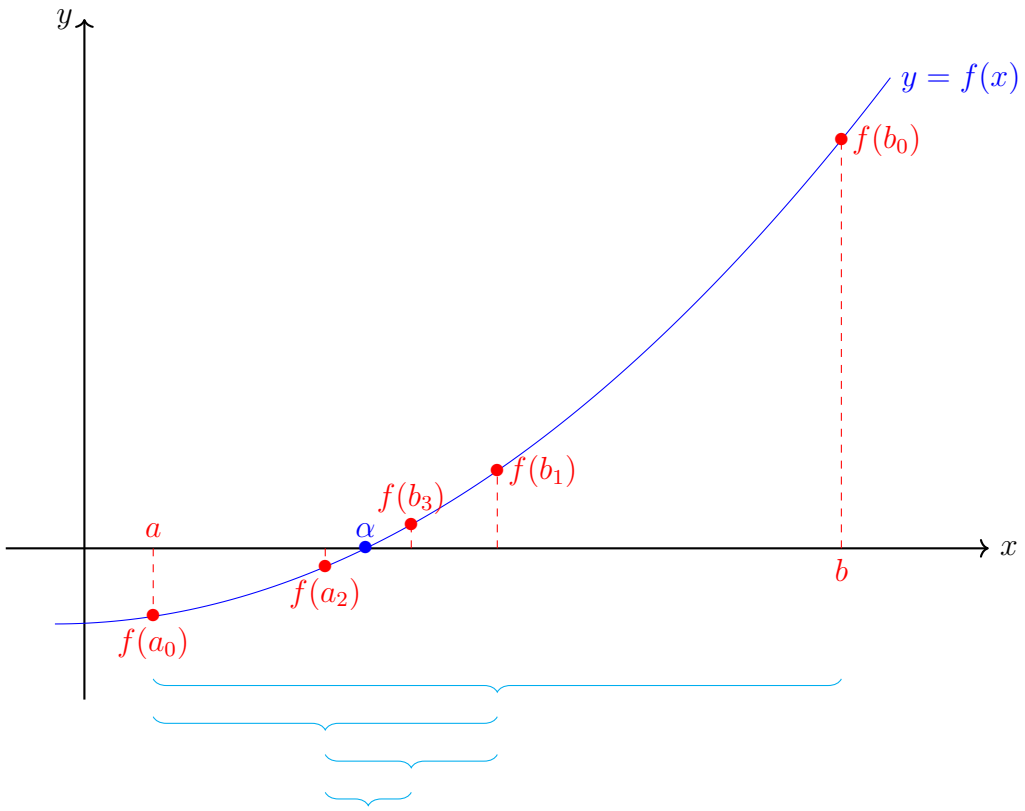

FIGURE 6 – Dichotomie

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 3. *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 1. *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 2. *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

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

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.

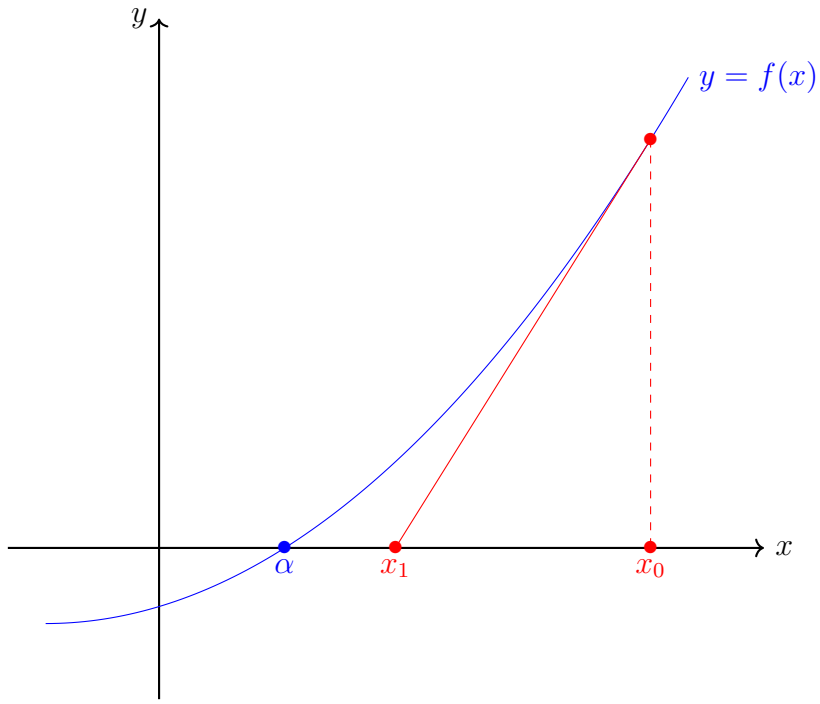

FIGURE 7 – Descente le long de la tangente

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

$$0 = f'(x_0)(x - x_0) + f(x_0) \iff x = x_0 - \frac{f(x_0)}{f'(x_0)}$$

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 4. *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.

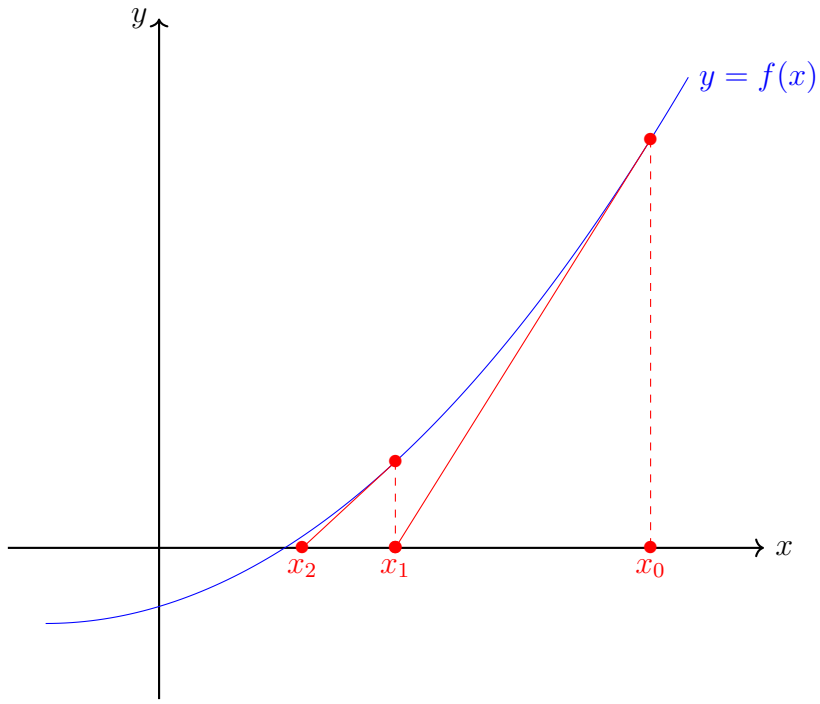

FIGURE 8 – Méthode de Newton

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*

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

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.

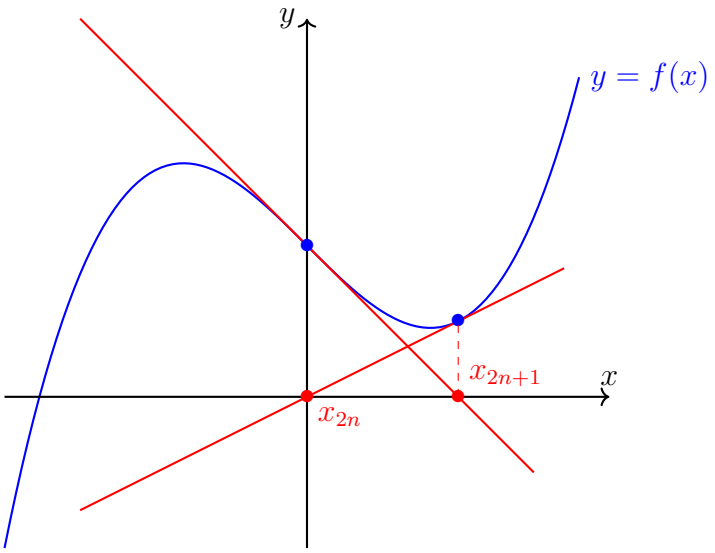

FIGURE 9 – Graphe de f et des tangentes en 0 et 1

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.

Le lecteur désirant approfondir l'étude des schémas numériques de résolution d'équations différentielles pourra consulter les ouvrages [6], [7] et [8].

IV Tableaux

1 Généralités

Définition 5. *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 3. *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 4. *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=deepcopy(a)
    for k in range(n):
        res[k]=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.])
```

Le lecteur désireux d'approfondir sa connaissance de l'arithmétique flottante pourra consulter l'ouvrage [5].

V Matrices

Le module indispensable pour manipuler des matrices est `numpy.linalg`. On recommande également l'importation de `numpy.random` pour la génération de matrices aléatoires.

Soient n et p entiers non nul. Les vecteurs de $\mathbb{R}^n$ sont codés par des tableaux à une dimension et les matrices de $\mathscr{M}_{n,p}(\mathbb{R})$ sont codées par des tableaux à deux dimensions, saisis par ligne. Par exemple, le vecteur $a = (1, 2, 3)$ et la matrice $\mathrm{B} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ sont saisis respectivement par

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

Dans ce qui suit, on parlera de matrices au sens large, les vecteurs pouvant être assimilés à des matrices lignes ou colonnes, au choix selon le point de vue considéré.

1 Génération de matrices

Les instructions `np.zeros` et `np.ones` génèrent des matrices constituées respectivement de 0 ou de 1. L'instruction `np.zeros(n)` génère le vecteur nul de $\mathbb{R}^n$ et l'instruction `np.zeros((n,p))` génère la matrice nulle de $\mathscr{M}_{n,p}(\mathbb{R})$.

```
>>> np.zeros(5)
array([ 0.,  0.,  0.,  0.,  0.])
>>> np.zeros((3,4))
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

Sur le même principe, on génère un vecteur ou une matrice constituée de 1.

```
>>> np.ones(5)
array([ 1.,  1.,  1.,  1.,  1.])
>>> np.ones((3,4))
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
```

Les instructions `np.zeros` et `np.ones` génèrent par défaut des matrices de flottants. On peut forcer l'utilisation d'un type entier en le spécifiant en option :

```
>>> a=np.ones((3,3),dtype='int')
>>> a
array([[1, 1, 1],
       [1, 1, 1],
       [1, 1, 1]])
```

L'instruction `np.diag` permet de construire une matrice diagonale de diagonale donnée ou au contraire d'extraire la diagonale d'une matrice.

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

L'instruction `np.eye(n)` génère la matrice identité d'ordre n.

```
>>> np.eye(4)
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
```

La fonction `A(n)` d'argument `n` entier non nul renvoie la matrice

$$\mathrm{A} = \begin{pmatrix} 0 & 1 & 0 & \dots & 0 \\ 1 & \ddots & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & 1 \\ 0 & \dots & 0 & 1 & 0 \end{pmatrix} \in \mathscr{M}_n(\mathbb{R})$$

```
def A(n):
    res=np.zeros((n,n))
    for i in range(1,n):
        res[i,i-1]=1
        res[i-1,i]=1
    return res
```

Exercice : Écrire une fonction `B(n)` d'argument `n` entier et qui renvoie la matrice

$$\mathrm{B} = \begin{pmatrix} 0 & 1 & 0 & \dots & 0 \\ n & 0 & 2 & \ddots & \vdots \\ 0 & n-1 & 0 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & n \\ 0 & \dots & 0 & 1 & 0 \end{pmatrix} \in \mathscr{M}_{n+1}(\mathbb{R})$$

Corrigé : On saisit :

```
def B(n):
    res=np.zeros((n+1,n+1))
    for i in range(n):
        res[i+1,i]=n-i
        res[i,i+1]=i+1
    return res
```

Pour tester des fonctions sur des vecteurs ou des matrices, il peut s'avérer très confortable de générer ceux-ci aléatoirement. L'instruction `rd.rand` génère un vecteur ou une matrice dont les coordonnées sont tirées aléatoirement et indépendamment dans $[\,0\,;1\,]$.

```
>>> rd.rand(5)
array([ 0.94525138,  0.10025385,  0.17437674,  0.6036298 ,  0.32027945])
>>> rd.rand(3,4)
array([[ 0.72529463,  0.4485876 ,  0.04743872,  0.79269754],
       [ 0.7819368 ,  0.70206414,  0.22630832,  0.68197435],
       [ 0.08252628,  0.13326646,  0.47804099,  0.08423917]])
```

Pour `n` entier non nul, on peut démontrer que le résultat du tirage `rd.rand(n,n)` est une matrice d'ordre n dont la probabilité qu'elle soit inversible est égale à 1. En pratique, on réalise donc un tirage d'une matrice dans $\mathrm{GL}_n(\mathbb{R})$.

Exercice : Écrire une fonction `V(L)` d'argument une liste de flottants $[x_1, \dots, x_n]$ qui renvoie la matrice *de Vandermonde* de cette liste définie par

$$\mathrm{V} = \begin{pmatrix} 1 & 1 & \dots & 1 \\ x_1 & x_2 & \dots & x_n \\ x_1^2 & x_2^2 & \dots & x_n^2 \\ \vdots & \vdots & & \vdots \\ x_1^{n-1} & x_2^{n-1} & \dots & x_n^{n-1} \end{pmatrix}$$

Corrigé : On saisit :

```
def V(L):
    n=len(L)
    res=np.zeros((n,n))
    for i in range(n):
        for j in range(n):
            res[i,j]=L[j]**i
    return res
```

Remarque : Une version sans exponentiation est possible :

```
def V(L):
    n=len(L)
    res=np.ones((n,n))
    res[1,:]=np.array(L)
    for i in range(2,n):
        res[i,:]=res[1,:]*res[i-1,:]
    return res
```

2 Opérations matricielles

On utilisera des matrices générées aléatoirement pour illustrer les opérations sur des exemples.

Une matrice peut être multipliée par un scalaire (produit *extérieur* sur un $\mathbb{K}$-ev), des matrices de dimensions compatibles peuvent être additionnées :

```
>>> a=rd.rand(3)
>>> a
array([ 0.57999144,  0.57460622,  0.79592186])
>>> b=rd.rand(3)
>>> b
array([ 0.77425788,  0.75196289,  0.11517891])
>>> a+b
array([ 1.35424932,  1.32656912,  0.91110077])
>>> 2*a
array([ 1.15998289,  1.14921245,  1.59184372])
>>> A=rd.rand(2,2)
>>> A
array([[ 0.49867413,  0.63919241],
       [ 0.76025963,  0.37326753]])
>>> B=rd.rand(2,2)
>>> B
array([[ 0.04274343,  0.57188152],
       [ 0.92958428,  0.93376888]])
>>> A+B
array([[ 0.54141756,  1.21107393],
       [ 1.68984391,  1.30703641]])
```

L'instruction `np.transpose` et la méthode `T` effectuent la transposition :

```
>>> A=rd.rand(2,3)
>>> np.transpose(A)
array([[ 0.79360088,  0.92593801],
       [ 0.0744836 ,  0.52320879],
       [ 0.79010506,  0.83641115]])
>>> A.T
array([[ 0.79360088,  0.92593801],
       [ 0.0744836 ,  0.52320879],
       [ 0.79010506,  0.83641115]])
```

L'instruction `np.trace` calcule la trace :

```
>>> A=rd.rand(3,3)
>>> A
array([[ 0.66657305,  0.84646752,  0.52250037],
       [ 0.99327682,  0.55750006,  0.2041305 ],
       [ 0.67305521,  0.34774888,  0.98803663]])
>>> np.trace(A)
2.2121097330222317
```

L'instruction `np.dot` et la méthode `dot` réalisent le produit matriciel :

```
>>> A=rd.rand(2,2)
>>> B=rd.rand(2,2)
>>> np.dot(A,B)
array([[ 0.5746594 ,  0.7904693 ],
       [ 0.48632516,  0.65721293]])
>>> A.dot(B)
array([[ 0.5746594 ,  0.7904693 ],
       [ 0.48632516,  0.65721293]])
```

Pour effectuer une exponentiation matricielle, on utilise l'instruction `alg.matrix_power` :

```
>>> A=rd.rand(3,3)
>>> alg.matrix_power(A,10)
array([[ 278.56566527,  349.41967131,  255.36858979],
       [ 422.80901928,  530.35190296,  387.60033548],
       [ 451.8667134 ,  566.80051128,  414.23830039]])
```

L'instruction `alg.det` calcule le déterminant et l'instruction `alg.inv` calcule l'inverse matricielle. Comme dit précédemment, l'exécution de l'instruction `rd.rand(n,n)` génère aléatoirement une matrice de $\mathrm{GL}_n(\mathbb{K})$.

```
>>> A=rd.rand(3,3)
>>> alg.det(A)
0.080804707847183632
>>> alg.inv(A)
array([[-2.49498297, -5.39252509,  6.78645757],
       [ 2.11390228,  1.17609491, -1.74598544],
       [ 1.76079039,  6.33667569, -6.31434742]])
```

Soient $\mathrm{A} = \begin{pmatrix} 1 & -1 & 1 \\ -2 & 1 & 2 \\ -2 & -1 & 4 \end{pmatrix}$ et $\mathrm{P} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$

Le calcul de $\mathrm{P}^{-1}\mathrm{AP}$ donne :

```
>>> A=np.array([[1,-1,1],[-2,1,2],[-2,-1,4]])
>>> P=np.array([[1,1,0],[1,0,1],[1,1,1]])
>>> alg.inv(P).dot(A.dot(P))
array([[ 1.,  0.,  0.],
       [ 0.,  2.,  0.],
       [ 0.,  0.,  3.]])
```

ou aussi :

```
>>> np.dot(alg.inv(P),np.dot(A,P))
array([[ 1.,  0.,  0.],
       [ 0.,  2.,  0.],
       [ 0.,  0.,  3.]])
```

3 Résolution

Soit $\mathrm{A} \in \mathrm{GL}_n(\mathbb{K})$ et $\mathrm{B} \in \mathscr{M}_{n,1}(\mathbb{K})$. Pour résoudre le système linéaire $\mathrm{AX} = \mathrm{B}$ d'inconnue $\mathrm{X} \in \mathscr{M}_{n,1}(\mathbb{K})$, on peut utiliser au choix l'instruction `alg.solve` ou l'inversion et le produit matriciel puisque

$$\mathrm{AX} = \mathrm{B} \iff \mathrm{X} = \mathrm{A}^{-1}\mathrm{B}$$

Par exemple, pour résoudre le système

$$\begin{cases} 10x + 7y + 8z + 7t &= 32 \\ 7x + 5y + 6z + 5t &= 23 \\ 8x + 6y + 10z + 9t &= 33 \\ 7x + 5y + 9z + 10t &= 31 \end{cases}$$

on saisit :

```
>>> A=np.array([[10,7,8,7],[7,5,6,4],[8,6,10,9],[7,5,9,10]])
>>> B=np.array([32,23,33,31])
```

puis

```
>>> alg.solve(A,B)
array([ 5.55555556, -6.55555556,  2.88888889, -0.11111111])
```

ou

```
>>> alg.inv(A).dot(B)
array([ 5.55555556, -6.55555556,  2.88888889, -0.11111111])
```

VI Algèbre bilinéaire

1 Produit scalaire, norme

Soit n entier non nul. L'instruction `np.dot` entre deux vecteurs de $\mathbb{R}^n$ renvoie leur produit scalaire canonique. On peut aussi l'utiliser comme méthode directement sur l'un des vecteurs concernés.

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

L'instruction `alg.norm` calcule la norme euclidienne d'un vecteur de $\mathbb{R}^n$.

```
>>> alg.norm(a)
3.7416573867739413
```

2 Orthonormalisation de Gram-Schmidt

Étant donné une famille libre $(u_1, \dots, u_p)$ de vecteurs de $\mathbb{R}^n$, il existe une famille orthonormée $(v_1, \dots, v_p)$ de vecteurs de $\mathbb{R}^p$ telle que

$$\forall k \in [\![1\,;\, p]\!] \qquad \text{Vect}\,(u_1, \dots, u_k) = \text{Vect}\,(v_1, \dots, v_k)$$

L'algorithme d'orthonormalisation de Gram-Schmidt permet de construire une telle famille orthonormée :

Algorithme 1 : Orthonormalisation

Entrées : $[u_1, \dots, u_p]$ libre
Résultat : $[v_1, \dots, v_p]$ orthonormée
$res \leftarrow [\ u_1/\|u_1\|\ ]$
pour $k \in [\![2\,;\, p]\!]$ **faire**
  $z \leftarrow u_k - \sum_{i=1}^{k-1} \langle u_k, v_i \rangle\, v_i$
  $res \leftarrow res + [\ z/\|z\|\ ]$
retourner res

Théorème 2. *Soit* $\mathrm{E} = \mathbb{R}^n$ *et* $(v_1, \dots, v_p)$ *une famille libre de vecteurs de* E. *La famille* $(v_1, \dots, v_p)$ *obtenue par orthonormalisation de Gram-Schmidt est une famille orthonormée vérifiant*

$$\forall k \in [\![1\,;\, p]\!] \qquad \text{Vect}\,(u_1, \dots, u_k) = \text{Vect}\,(v_1, \dots, v_k)$$

Dans l'algorithme de Gram-Schmidt, l'étape itérative consiste à construire $z_k = u_k - p_{\mathrm{F}_k}(u_k)$ où $\mathrm{F}_k = \text{Vect}\,(u_1, \dots, u_{k-1}) = \text{Vect}\,(v_1, \dots, v_{k-1})$. Ainsi, on a

$$v_k = \frac{z_k}{\|z_k\|} \in \mathrm{F}_k^{\perp} \quad \text{d'où} \quad v_k \perp v_i \quad \forall i \in [\![1\,;\, k-1]\!]$$

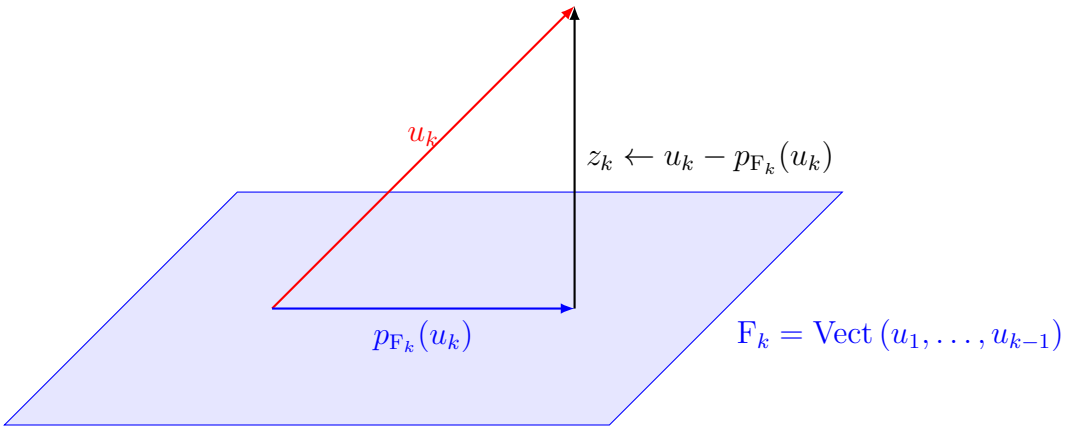

FIGURE 10 – Étape itérative de l'algorithme d'orthonormalisation

Une implémentation de l'algorithme est donnée par :

```
def norm(u):
    # normalise le vecteur u
    return u/alg.norm(u)

def ortho(A):
    # Renvoie la matrice constituée des colonnes orthonormalisées
    # à partir des colonnes de la matrice A
    p=A.shape[1]
    v=[norm(A[:,0])]
    for k in range(1,p):
        u=A[:,k]
        z=u-sum([np.dot(v[i],u)*v[i] for i in range(k)])
        v.append(norm(z))
    return np.array(v).T
```

On remarque que l'implémentation d'une fonction de *normalisation* qui prend un vecteur en entrée et le normalise rend la fonction `ortho` très lisible avec une écriture assez légère.

```
>>> A=rd.rand(4,3)
>>> A
array([[ 0.04565402,  0.24376409,  0.44048029],
       [ 0.26116379,  0.66024369,  0.59413365],
       [ 0.600649  ,  0.4420384 ,  0.77978355],
       [ 0.82817965,  0.14609593,  0.44261169]])
>>> B=ortho(A)
>>> B
array([[ 0.04319782,  0.33972245,  0.7314721 ],
       [ 0.24711311,  0.81200547, -0.50851586],
       [ 0.56833393,  0.20877085,  0.4162726 ],
       [ 0.78362338, -0.42620478, -0.18187169]])
```

Si $p = n$, la famille orthonormalisée $(v_1, \ldots, v_n)$ est une base orthonormée de $\mathbb{R}^n$. Par conséquent, la matrice M obtenue dont les colonnes sont les coordonnées des vecteurs v_i est une

matrice orthogonale, *i.e.* vérifiant $M^\top M = I_n$ ou de manière équivalente $MM^\top = I_n$. Pour tester l'orthogonalité d'une matrice, on saisit :

```
def ps(A,B):
    return np.trace(A.T.dot(B))

def isortho(A):
    n=len(A)
    eps=1e-8
    B=A.T.dot(A)-np.eye(n)
    return ps(B,B)<eps
```

On peut facilement générer aléatoirement une matrice orthogonale par orthonormalisation d'une matrice tirée aléatoirement dans $GL_n(\mathbb{R})$:

```
>>> A=rd.rand(3,3)
>>> B=ortho(A)
>>> np.dot(B.T,B)
array([[  1.00000000e+00,   5.05754733e-15,  -8.89543056e-15],
       [  5.05754733e-15,   1.00000000e+00,  -1.03875092e-15],
       [ -8.92318613e-15,  -1.03875092e-15,   1.00000000e+00]])
>>> isortho(B)
True
```

VII Probabilités

La technologie informatique permet de générer des nombres *pseudo-aléatoires* dont les tirages ressemblent à des tirages réellement faits au hasard. L'algorithme *Mersenne Twister*, utilisé par python, est l'un des plus réputés pour la qualité de son pseudo-hasard et sa rapidité d'exécution. Pour effectuer des simulations aléatoires, on importera le module `numpy.random` sous l'alias `rd` :

```
import numpy.random as rd
```

1 Quelques expérimentations convaincantes

On présente des tirages selon $\mathscr{U}_{[0;1]^2}$.

```
n=1000
tx=rd.random(n)
ty=rd.random(n)
plt.plot(tx,ty,'bo')
plt.show()
```

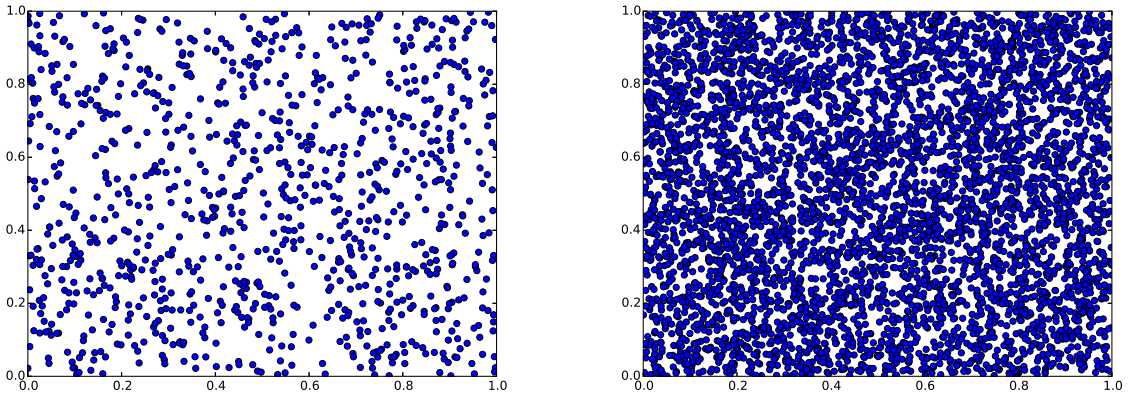

FIGURE 11 – Tirages selon la loi uniforme sur $[\,0\,;1\,]^2$

On peut aussi observer des résultats précis comme par exemple le *théorème de la limite centrée* et en particulier sa version spécialisée qu'est le *théorème de Moivre-Laplace* facilement implémentable.

Dans $(\Omega, \mathscr{A}, \mathbb{P})$ espace probabilisé, pour X une variable aléatoire réelle, on rappelle que le fonction de répartition de X notée $\mathrm{F_X}$ est définie par $\mathrm{F_X}(x) = \mathbb{P}(\mathrm{X} \leqslant x)$ pour x réel.

Définition 6. *Soit $(\Omega, \mathscr{A}, \mathbb{P})$ un espace probabilisé, X et $(\mathrm{X}_n)_{n\geqslant 1}$ des variables aléatoires réelles. On dit que $(\mathrm{X}_n)_n$* converge en loi *vers X que l'on note $\mathrm{X}_n \rightsquigarrow \mathrm{X}$ si, pour tout x point de continuité de $\mathrm{F_X}$, on a*

$$\mathrm{F}_{\mathrm{X}_n}(x) \xrightarrow[n\to\infty]{} \mathrm{F_X}(x)$$

Commentaire : Pour n grand, la loi de X_n ressemble à la loi de X.

Soit $(\Omega, \mathscr{A}, \mathbb{P})$ un espace probabilisé, $(\mathrm{X}_n)_{n\geqslant 1}$ une suite de variables aléatoires indépendantes de même loi $\mathscr{B}(p)$ avec $p \in \,]\,0\,;1\,[$ et $\mathrm{S}_n = \sum_{i=1}^{n} \mathrm{X}_i$ pour n entier. D'après le théorème de Moivre-Laplace, on a

$$\frac{\mathrm{S}_n - np}{\sqrt{np(1-p)}} \rightsquigarrow \mathrm{U} \quad \text{avec} \quad \mathrm{U} \sim \mathcal{N}(0,1)$$

où $\mathcal{N}(0,1)$ désigne la loi normale centrée réduite, *i.e.*

$$\forall x \in \mathbb{R} \qquad \mathbb{P}(\mathrm{U} \leqslant x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} \mathrm{e}^{-\frac{t^2}{2}}\, \mathrm{d}t$$

```
n=2000
N=20000
tS=[]
for k in range(N):
    tX=2*rd.randint(0,2,n)-1
    tS.append(sum(tX)/np.sqrt(n))
plt.hist(tS,30,normed=True)
f=lambda x:1/np.sqrt(2*np.pi)*np.exp(-x**2/2)
tx=np.linspace(-5,5,100)
```

```
tf=[f(x) for x in tx]
plt.plot(tx,tf,'r',linewidth=2)
plt.grid();plt.show()
```

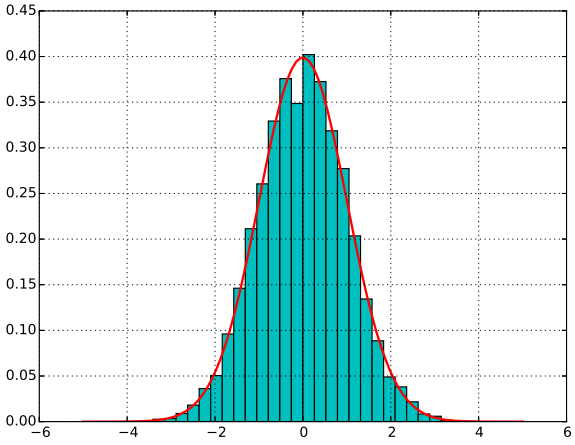

FIGURE 12 – Convergence en loi vers la loi normale centrée réduite

2 Simulation de lois

Même si elle est officiellement hors-programme, il est prudent d'avoir quelques connaissances sur la loi uniforme sur le segment $[\,0\,;1\,]$. Soit $(\Omega,\mathscr{A},\mathbb{P})$ un espace probabilisé. Une variable aléatoire U suit la *loi uniforme* sur le segment $[\,0\,;1\,]$ notée $\mathrm{U}\sim\mathscr{U}_{[\,0\,;1\,]}$ si $\mathrm{U}(\Omega)=[\,0\,;1\,]$ et

$$\forall\,[\,a\,;b\,[\,\subset[\,0\,;1\,]\qquad \mathbb{P}\left(\mathrm{U}\in[\,a\,;b\,[\right)=\int_0^1 \mathbb{1}_{[\,a\,;b\,[}(t)\,\mathrm{d}t=b-a$$

L'instruction `rd.random()` renvoie une réalisation de cette loi.

Soit $p\in[\,0\,;1\,]$. Une variable aléatoire X suivant la loi de Bernoulli de paramètre p vérifie

$$\mathrm{X}(\Omega)=\{0,1\}\quad\text{et}\quad\mathbb{P}(\mathrm{X}=1)=p\qquad\mathbb{P}(\mathrm{X}=0)=1-p$$

Pour simuler une loi de Bernoulli de paramètre p, on code le résultat d'une indicatrice d'un événement de probabilité p. On peut considérer $\mathbb{1}_{\mathrm{U}<p}$ avec $\mathrm{U}\sim\mathscr{U}_{[\,0\,;1\,]}$ (loi uniforme sur $[\,0\,;1\,]$, officiellement hors-programme).

```
ber=lambda p: rd.random()<p
```

Soit n entier. Une variable aléatoire Y suivant la loi binomiale de paramètres n,p vérifie

$$\mathrm{Y}(\Omega)=[\![\,0\,;\,n\,]\!]\quad\text{et}\quad\forall k\in[\![\,0\,;\,n\,]\!]\qquad\mathbb{P}(\mathrm{Y}=k)=\binom{n}{k}p^k(1-p)^{n-k}$$

Exercice : Proposer une implémentation `binomiale(n,p)` qui simule une réalisation de la loi $\mathscr{B}(n,p)$.

Corrigé : Une somme de n variables aléatoires indépendantes de même loi de Bernoulli de paramètre p suit la loi binomiale $\mathscr{B}(n,p)$. On peut donc réaliser cette loi binomiale avec

```
binomiale=lambda n,p:sum([ber(p) for k in range(n)])
```

Soit $(\Omega, \mathscr{A}, \mathbb{P})$ un espace probabilisé et $p \in]\,0\,;1\,[$. On rappelle qu'une variable aléatoire suivant une loi géométrique de paramètre p modélise le nombre de répétitions d'une suite d'expériences aléatoires succès/échec, succès avec probabilité p, jusqu'à obtention du premier succès.

Exercice : Écrire une fonction python `geom(p)` qui renvoie une réalisation de la loi géométrique de paramètre $p \in]\,0\,;1\,[$.

Corrigé : On saisit :

```
def geom(p):
    res=1
    while ber(p)==0:
        res+=1
    return res
```

3 Méthodes de Monte-Carlo

Définition 7. *Soit $(\Omega, \mathscr{A}, \mathbb{P})$ un espace probabilisé et X une variable aléatoires réelle discrète. On suppose disposer d'une suite de variables aléatoires discrètes $(\mathrm{X}_i)_{i\geqslant 1}$ indépendantes et de même loi que X. Le principe général d'une* méthode de Monte-Carlo *est d'approcher la valeur de l'espérance $\mathbb{E}(\mathrm{X})$ par la* moyenne empirique $\dfrac{1}{n}\displaystyle\sum_{i=1}^{n}\mathrm{X}_i(\omega)$ *pour une réalisation $\omega \in \Omega$. Autrement dit, pour un n choisi par l'expérimentateur, on effectue l'approximation*

$$\frac{1}{n}\sum_{i=1}^{n}\mathrm{X}_i(\omega) \simeq \mathbb{E}(\mathrm{X}) \quad \textit{avec} \quad \omega \in \Omega$$

Remarque : La justification du bien-fondé des méthodes de Monte-Carlo est *la loi des grands nombres*. Si les X_i sont dans L^2, on dispose du résultat intitulé *loi faible des grands nombres*

$$\forall \varepsilon > 0 \qquad \mathbb{P}\left(\left|\frac{1}{n}\sum_{i=1}^{n}\mathrm{X}_i - \mathbb{E}(\mathrm{X})\right| \geqslant \varepsilon\right) \xrightarrow[n\to\infty]{} 0$$

Notant $\bar{\mathrm{X}}_n = \dfrac{1}{n}\displaystyle\sum_{i=1}^{n}\mathrm{X}_i$ la moyenne empirique, on dit que $\left(\bar{\mathrm{X}}_n\right)_n$ *converge en probabilité* vers $\mathbb{E}(\mathrm{X})$ et l'on note $\bar{\mathrm{X}}_n \xrightarrow[n\to\infty]{\mathbb{P}} \mathbb{E}(\mathrm{X})$. Il existe un résultat plus puissant appelé *loi forte des grands nombres* qui dit que pour une suite de variables aléatoires indépendantes de même loi et d'espérance finie, la moyenne empirique converge presque sûrement vers l'espérance, c'est-à-dire

$$\bar{\mathrm{X}}_n \xrightarrow[n\to\infty]{} \mathbb{E}(\mathrm{X}) \quad \text{p.s.}$$

Comme les aléas pour lesquels la convergence n'a pas lieu sont dans un ensemble négligeable, l'usage des méthodes de Monte-Carlo est pertinent.

Exemple : On lance une pièce équilibrée 1000 fois et on effectue la moyenne du nombre de fois où l'on obtient pile. On peut simuler cette expérience en réalisant 1000 tirages indépendants d'une loi de Bernoulli $\mathscr{B}(1/2)$. L'instruction `rd.randint(a,b)` renvoie une réalisation de loi uniforme sur $[\![a\,;\, b-1]\!]$. Des appels successifs fournissent des réalisations indépendantes. On peut aussi exécuter `rd.randint(a,b,n)` qui renvoie n réalisations indépendantes de loi uniforme sur $[\![a\,;\, b-1]\!]$.

Si, pour une réalisation $\omega \in \Omega$ donnée, on souhaite visualiser la convergence de la suite $\left(\frac{1}{n}\sum_{i=1}^{n} \mathrm{X}_i(\omega)\right)_{n\geqslant 1}$, on saisit :

```
n=1000
tX=rd.randint(0,2,n)
tmoy=[]
s=0
tn=range(1,n+1)
for i in tn:
    s+=tX[i-1]
    tmoy.append(s/i)
plt.plot(tn,tmoy)
plt.grid();plt.show()
```

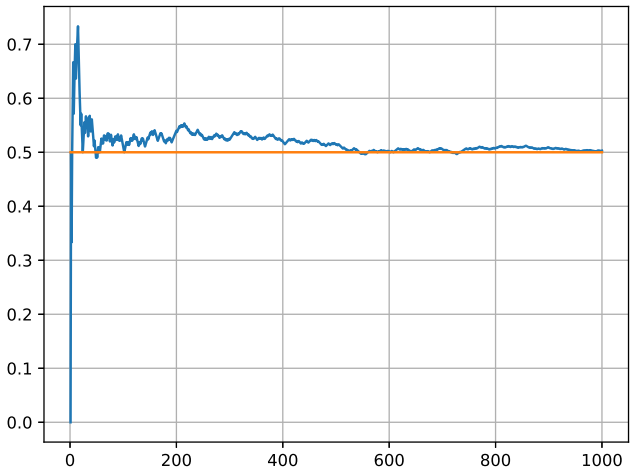

FIGURE 13 – Convergence de la moyenne empirique du nombre de « pile »

En affichant simultanément plusieurs réalisations, on perçoit la dispersion des résultats, dispersion étroitement liées à la variance de la variable aléatoire d'intérêt. Par exemple, pour des lois uniformes $\mathscr{U}_{[\![3\,;7]\!]}$ et $\mathscr{U}_{[\![0\,;10]\!]}$, on a même espérance mais une variance plus élevée pour la deuxième loi. On rappelle que pour $\mathrm{X} \sim \mathscr{U}_{[\![a\,;b]\!]}$, on a

$$\mathbb{E}(\mathrm{X}) = \frac{a+b}{2} \qquad \mathbb{V}(\mathrm{X}) = \frac{(b-a)(2+b-a)}{12}$$

Cette dispersion apparaît clairement sur les simulations suivantes :

```
n=1000
for j in range(10):
    tX=rd.randint(3,8,n)
    tmoy,s=[],0
    tn=range(1,n+1)
    for i in tn:
        s+=tX[i-1]
        tmoy.append(s/i)
    plt.plot(tn,tmoy)
plt.axis([0,n,0,10]);plt.grid();plt.show()


n=1000
for j in range(10):
    tX=rd.randint(0,11,n)
    tmoy,s=[],0
    tn=range(1,n+1)
    for i in tn:
        s+=tX[i-1]
        tmoy.append(s/i)
    plt.plot(tn,tmoy)
plt.axis([0,n,0,10]);plt.grid();plt.show()
```

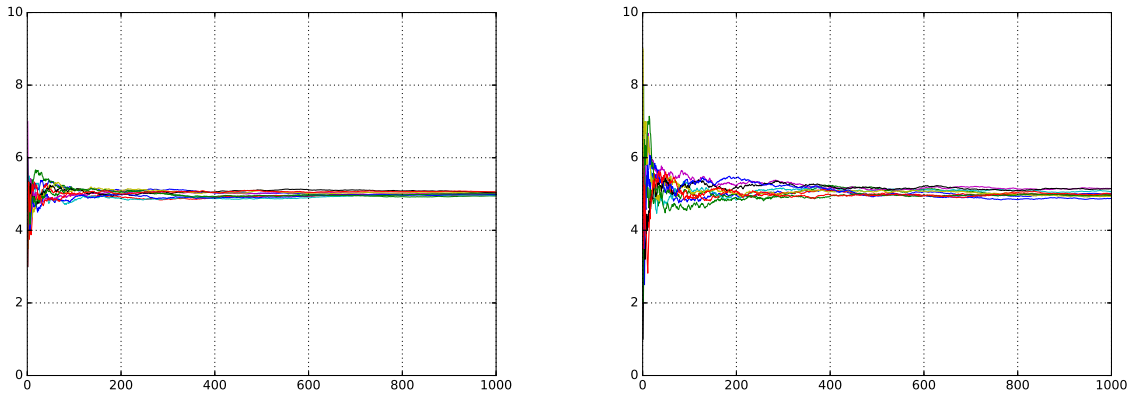

FIGURE 14 – Dispersion des simulations

4 Marches aléatoires dans $\mathbb{Z}$ et $\mathbb{Z}^2$

Définition 8. *Soit* $(\Omega, \mathscr{A}, \mathbb{P})$ *un espace probabilisé. On appelle* marche aléatoire dans $\mathbb{Z}$ *la suite* $(S_n)_{n\geqslant 1}$ *avec* $S_n = \sum_{i=1}^{n} X_i$ *pour* n *entier non nul et* $(X_i)_{i\geqslant 1}$ *une suite de variables aléatoires indépendantes de loi uniforme sur* $\{-1, 1\}$.

Remarque : Voir feuille d'exercices pour une étude théorique sommaire de la marche aléatoire dans $\mathbb{Z}$.

```
n=1000
tn=range(n+1)
for j in range(4):
    tX=2*rd.randint(0,2,n)-1
    tS=[0]
    for X in tX:
        tS.append(tS[-1]+X)
    plt.plot(tn,tS)
plt.grid();plt.show()
```

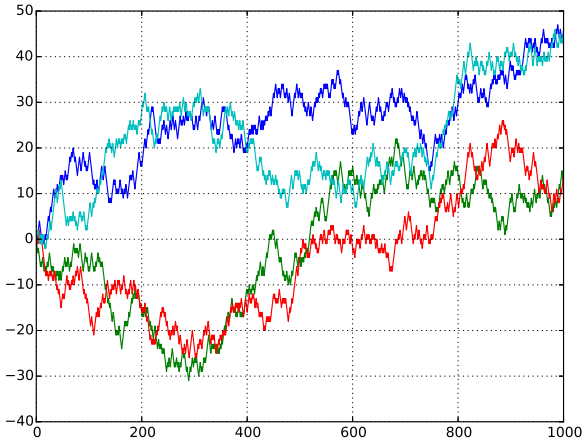

FIGURE 15 – Marches aléatoires dans $\mathbb{Z}$

Définition 9. *Soit* $(\Omega, \mathscr{A}, \mathbb{P})$ *un espace probabilisé. On appelle* marche aléatoire dans $\mathbb{Z}^2$ *la suite* $(S_n)_{n\geqslant 1}$ *avec* $S_n = \sum_{i=1}^{n} X_i$ *pour* n *entier non nul et* $(X_i)_{i\geqslant 1}$ *une suite de variables aléatoires indépendantes de loi uniforme sur* $\{(1,0),(-1,0),(0,1),(0,-1)\}$.

```
n=10000
tn=range(n)
depl=[[1,0],[-1,0],[0,1],[0,-1]]

tind=rd.randint(0,4,n)
tSX=[0]
tSY=[0]
for i in tind:
    tSX.append(tSX[-1]+depl[i][0])
    tSY.append(tSY[-1]+depl[i][1])
plt.plot(tSX,tSY)
plt.axis('equal')
plt.grid();plt.show()
```

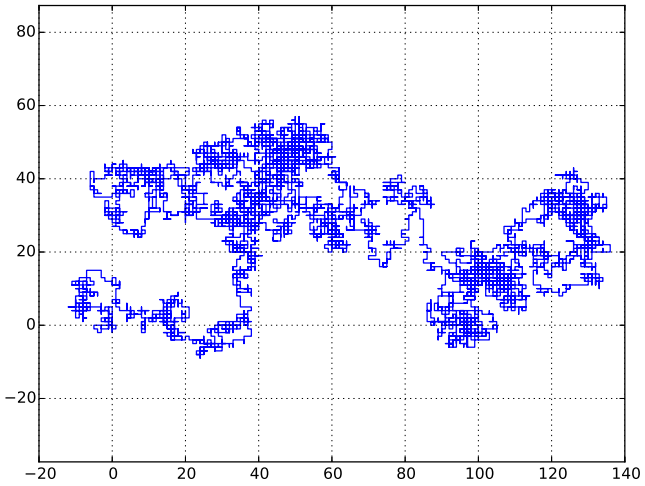

FIGURE 16 – Marches aléatoires dans $\mathbb{Z}^2$

Le lecteur désireux d'approfondir ce sujet pourra consulter les ouvrages [10] et [11].

Références

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

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

[3] 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

[4] 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

[5] Vincent Lefèvre, Paul Zimmermann, *Arithmétique flottante*, RR-5104, INRIA, 2004

[6] Jean-Pierre Demailly, *Analyse numérique et équations différentielles* EDP Sciences, 2006

[7] Michelle Schatzman, *Numerical Analysis*, Clarendon Press, 2002

[8] Catherine Bolley, *Analyse numérique*, École d'ingénieur, Nantes, cel-01066570, 2012

[9] Nicolas Bouleau, *Probabilités de l'ingénieur - Variables aléatoires et simulation*, Hermann, 2002

[10] Nicolas Bouleau, *Probabilités de l'ingénieur - Variables aléatoires et simulation*, Hermann, 2002

[11] G.S. Fishman, *Monte Carlo - Concepts, Algorithms and applications* , Springer Series in Operations Research, Springer-Verlag, 1996