Les entiers, comme tout autre objet, sont représentés dans la machine par des 0 et des 1 (ou plutôt tension haute/basse). Nous allons voir dans ce chapitre comment cela se passe exactement, et quelles sont les limites de ces représentations.

I. Écriture en base b

Depuis le moyen âge on représente les nombres entiers par la numération décimale de position : lorsqu'on écrit 548 on veut dire : $5 \times 10^2 + 4 \times 10^1 + 8 \times 10^0$. La numération par position est très pratique car elle limite le nombre de symboles (en base 10 les chiffres de 0 à 9) ce qui est plus pratique que la numération en chiffres romains.
L'écriture de position permet aussi de poser les opérations (addition, soustraction, multiplication, divison) très facilement avec le principe de la retenue (cf. primaire).
La base 10 est la plus répandue car nous avons 10 doigts. Mais nous sommes aussi habitués à compter en base 60 (pour le temps) et en France en base 20 pour la lecture de certains nombres (96 se lit quatre-vingt seize et pas nonante six).
Dans un ordinateur les circuits et mémoires sont composés de transistors ne pouvant prendre que deux états. L'information est donc stockée sous forme de chaîne de *bits* (*binary digit* ou chiffres binaires 0 ou 1). Le choix de la base 2 est donc naturel.

1 - Définition

Définition :

Si $b \in \mathbb{N}$, tout entier naturel non nul n s'écrit de manière unique

$$n = \sum_{k=0}^{N} a_k b^k = a_0 + a_1 b + a_2 b^2 + ... + a_N b^N$$

Remarque 1 :

La base naturelle est la base 10 (décimale). En informatique, les bases usuelles seront la base 2 (binaire, dont les chiffres sont 0 et 1 appelés bits), la base 8 (octal, dont les chiffres sont 0, 1, 2, 3, 4, 5, 6, 7) et la base 16 (hexadécimal, dont les chiffres sont 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F) : une unité hexadécimale représente 4 bits soit 1/2 octets.

Un entier n s'écrit en base 2 sous la forme $b_k b_{k-1} \cdots b_0$ si les b_i valent 0 ou 1 et vérifient :

$$n = \sum_{i=0}^{k-1} b_i 2^i$$

Ainsi $21 = 16 + 4 + 1 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^1$. Il s'écrit donc en base 2 : 10101.
Pour déterminer l'écriture binaire d'un entier n on procède à des divisions par deux successives :

- 11 divisé par 2 donne 5 il reste 1 ;
- 5 divisé par 2 donne 2 il reste 1 ;
- 2 divisé par 2 donne 1 il reste 0 ;
- et enfin 1 divisé par 2 donne 0 il reste 1.

L'écriture en binaire de 11 est donc `[1,0,1,1]` (attention à l'ordre !). C'est-à-dire que $11 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$.

Définition :

Le dernier bit codant pour un entier en base 2 est appelé le **le bit de poids faible**. (cf. steganographie). Le premier bit codant pour un entier en base 2 est appelé le **le bit de poids fort**.

Exemple 1 :

Proposer un code python permettant d'obtenir une liste codant un nombre en base 2

Dans l'exemple ci-dessous il faut bien tenir compte que le bit de poids faible est obtenu en premier. Il faut donc retourner la liste pour l'obtenir dans le bon sens.

```
1 def base2(n):
2     L=[]
3     while n>0:
4         L.append(n%2)
5         n = n//2
6     return L[::-1]
```

Propriété :

La parité est directement donnée par le bit de poids faible

Exemple 2 :

Conversion en base de 2 pour 42 ?

$$42 = 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2 + 0 \times 1$$

[1, 0, 1, 0, 1, 0]

Remarque 2 :

Le contrôle de parité (appelé parfois VRC, pour Vertical Redundancy Check ou Vertical Redundancy Checking) est un des systèmes de contrôle les plus simples. à coder sur le bit de poids fort le nombre total de bits. Il consiste à ajouter un 1 si le nombre de bits du mot de code est impair, 0 dans le cas contraire.

Prenons l'exemple suivant : `L=[1,0,0,0,0,1,0,1]`

Dans cet exemple, le nombre de bits de données à 1 est pair, le bit de parité est donc positionné à 0. Dans l'exemple suivant, par contre, les bits de données étant en nombre impair, le bit de parité est à 1 :`L=[1,0,0,0,0,1,1,1]`

Imaginons désormais qu'après transmission le bit de poids faible (le bit situé à droite) de l'octet précédent soit victime d'une interférence : `L=[1,0,0,0,0,1,1,0]` Le bit de parité ne correspond alors plus à la parité de l'octet : une erreur est détectée.

2 - Évaluation d'un nombre écrit dans une base b ,

Il est assez simple de calculer la valeur d'un nombre en connaissant son écriture en base b. $[a_N,..a_0]$

```
1  def eval(L, b):
2      L = L[::-1]
3      b_puiss = 1
4      resultat = 0
5      for chiffre in L:
6          resultat += chiffre * b_puiss
7          b_puiss *= b
8      return resultat
```

Remarque 3 :

Pourquoi utiliser une variable `b_puiss` plutôt que de calculer bk à chaque étape ?

La méthode de Horner permet de calculer rapidement un nombre :

$$n = a_0 + 2 \times (a_1 + 2 \times (a_2 + 2...)$$

```
1  def eval_horner(L, b):
2      resultat = 0
3      for chiffre in L[::-1]:
4          resultat *= b
5          resultat += chiffre
6      return resultat
```

3 - Nombres de chiffres

Propriété :

Nombre de chiffres en base b : n possède k chiffres en base b si et seulement si

$$b^{k-1} \leq n < b^k$$

donc : $k = \lfloor \log_b n \rfloor + 1$

Exemple 3 :

Combien de bit sont nécessaire pour écrire 42 ?

$$N = \lfloor \log(42)/\log(2) \rfloor + 1 = 6$$

Dans la plupart des langages, les entiers sont mémorisés en machine dans une liste de k *bits* l'entier k étant imposé (souvent une puissance de 2 : $8bits$, $16bits$, etc.).

Pour un codage sur un octet, l'écriture en binaire de 11 est donc `[0,0,0,0,1,0,1,1]`.

II. Opérations sur les entiers

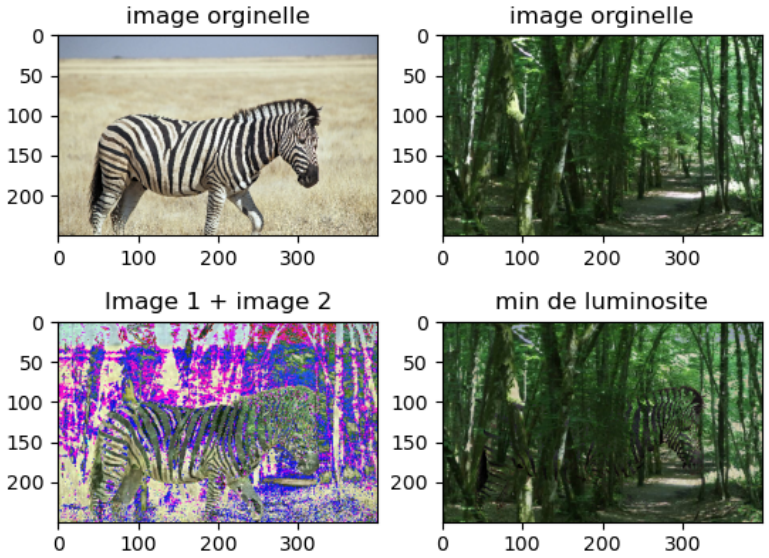
1 - L'addition

L'addition des entiers en binaire se fait comme celle des entier en base 10 :

	1	1	0	1	13
+	1	0	0	1	9
1			1		retenue
1	0	1	1	0	

Remarque 4 :

Lorsque les entiers sont codés sur un octet, une somme peut conduire à dépassement de capacité.



2 - Les entiers relatifs

a) bit de signe

Une première manière de représenter les entiers relatifs est de réserver un *bit* à gauche pour le signe égal à 0 si l'entier est positif et à 1 s'il est négatif. Si l'on veut coder nos entiers sur k *bits* il ne reste donc que $k-1$ *bits* pour représenter $|n|$. Ainsi, sur 8 bits on ne peut représenter que les entiers de -127 à 127. Par exemple 98 est représenté par 01100010 et -98 par 11100010.

Ex. 1 :

Écrire une fonction `base2_rel(n)` qui associe aux entiers n et k la liste correspondant à l'écriture binaire de l'entier relatif n sur k *bits*. On pourra faire appel à la fonction `base2_octet`.

Cette méthode d'écriture des entiers relatifs a plusieurs inconvénients :

- l'entier 0 a deux representations (puisque $0 = -0$).
- il faut deux algorithmes différents pour additionner les entiers de même signe et ceux de signes opposés.

b) Complément à 2

Pratiquement pour coder un entier négatif $-n$ sur k *bits* on prend l'écriture de n sur $k-1$ *bits*, on inverse chaque *bit* (0 devient 1 et 1 devient 0) puis on ajoute 1 à gauche.

0	1	1	1	1	1	1	1	=	127
0								=	1
0	0	0	0	0	0	1	0	=	2
0	0	0	0	0	0	0	1	=	1
0	0	0	0	0	0	0	0	=	0
1	1	1	1	1	1	1	1	=	-1
1	1	1	1	1	1	1	0	=	-2
1								=	-...
1	0	0	0	0	0	0	1	=	-127
1	0	0	0	0	0	0	0	=	-128

Les propriétés de cette représentation :

- 0 n'a qu'une représentation ;
- les entiers représentables sur k *bits* sont les entiers compris entre -2^{k-1} et $2^{k-1}-1$;

- le premier *bit* (appelé *bit* de poids fort) représente le signe (0 pour positif, 1 pour négatif) ;
- l'addition des entiers est codée selon le même algorithme qu'ils soient de même signe ou non.

Représentation des nombres réels

3 - Rappel sur la notation scientifique, la norme IEE-754

Remarque 5 :
L'ordinateur ne sais pas gérer parfaitement les nombres réels. Il ne peut que les approcher par des **nombres décimaux**. (i.e. nombres ayant un nombre fini de chiffres après la virgule). Nous utiliserons abusivement le terme « nombre réel » pour désigner ces nombres.

L'écriture en virgule flottante d'un nombre décimal est de la forme :

$$x = s \times m \times b^e$$

b est la base de numération (pour nous : 10 ou 2). s désigne le signe, m la **mantisse** et e l'exposant. L'écriture est **normalisée** si $m \in [1,b[$.
Ainsi, $-53{,}4$ peut s'écrire $-53{,}4 \times 10^0$ ou $-5{,}34 \times 10^1$ ou encore $-534{,}0 \times 10^{-1}$. L'écriture normalisée est $-5{,}34 \times 10^1$.

Définition :
Un réel n peut être approché par un nombre de la forme :

$$n = (-1)^s x{,}xxxxx\cdots \times 10^i \tag{1}$$

où x représente symboliquement chaque chiffre décimal, $s \in \{0;1\}$ représente le signe et $i \in \mathbb{Z}$ est l'exposant.

$$n = (-1)^s \times \sum_k (x\frac{1}{10^k}) \times 10^i$$

Définition :
L'écriture normalisée de n dans la base binaire est :

$$n = (-1)^s d{,}ddddd\cdots \times 2^j \tag{2}$$

où d représente chaque chiffre binaire, $s \in \{0;1\}$ représente encore le signe et $j \in \mathbb{Z}$ est l'exposant.

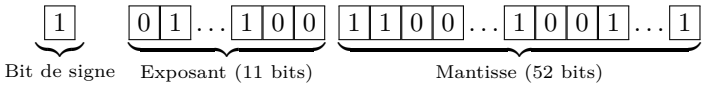
$$XX = (-1)^s \times \sum (\frac{1}{2^d}) \times 2^j$$

On remarquera qu'en binaire, le premier chiffre est **forcément** un 1, car sinon, cela signifie que l'on pourrait encore décaler la virgule. Ce chiffre n'a donc pas besoin d'être enregistré dans la mémoire.

4 - Codage des flottants

Pour coder un nombre réel, il suffit de mettre bout à bout :

— **un bit de signe :** 0 (positif) ou 1 (négatif).
— **l'exposant :** sa taille dépend du code utilisé. Avec Python nous la coderons sur 11 bits de -1022 à 1023.
— **la mantisse :** sa taille dépend du code utilisé. Avec Python, nous la coderons sur 52 bits.



(3)

Définition :
Dans la notation scientifique binaire ci-dessus, on appelle :
— **Mantisse :** les chiffres binaires situés après la virgule (sans le premier chiffre qui est forcément 1) : (1),*ddddddd*.
— **Exposant :** le nombre j de la puissance de 2. On notera que j est un nombre **signé** (on peut avoir des puissances négatives).

Le plus grand flottant s'écrit donc en binaire :

$$1,\underbrace{11\cdots1}_{52\text{ termes}} \times 2^{1023} = (1 + 2^{-1} + \cdots + 2^{-52}) \times 2^{1023} \simeq 1{,}79 \times 10^{308}$$

Cela a pour conséquence que les flottants ne sont pas du tout répartis de la même manière entre deux bornes :

- l'écart entre deux flottants consécutifs compris entre 1 et 2 est 2^{-52}. Il y a donc environ 2^{52} flottants entre 1 et 2 ;
- il n'y a aucun flottant entre 2^{52} et 2^{53}.

5 - Précision des nombres

Définition :

On appelle **précision** les valeurs (absolues) minimum et maximum que notre type réel est capable de représenter.

La précision dépend du type et du langage de programmation utilisé. En général, on différenciera :

— **les réels « simple précision »** (*float*) (8 bits d'exposant et 23 bits de mantisse).
Leur valeur est comprise entre $1 \times 2^{-128} \approx 3 \times 10^{-39}$ et $2 \times 2^{127} \approx 3.4 \times 10^{38}$

— **les réels « double précision »** (*double*) (11 bits d'exposant et 52 bits de mantisse)
Leur valeur est comprise entre $1 \times 2^{-1024} \approx 5.6 \times 10^{-309}$ et $2 \times 2^{1023} \approx 1.8 \times 10^{308}$

Le nombre de chiffre significatif est donné par :

```
>>> 0.5**23 # simple prec
1.1920928955078125e-07
>>> 0.5**52 # double prec
2.220446049250313e-16
```

On retiendra qu'en 64 python (utilisé par Python), il y a 16 chiffres significatifs en double précision.

6 - Erreur d'arrondi

Les calculs dans le type `float` sont des calculs **approchés** :

Exemple 4 :

Justifier les cas suivants

```
>>> 0.1 * 3
0.30000000000000004
>>> 0.1*3==.3
False
>>>1+1e-16==1
True
```

Le calcul dans le type `float` ont d'autres limites :

- L'addition des flottants n'est pas associative :

```
>>> a = -10**30 ; b = -a ; c = 1.
>>> (a + b) + c
>>> a + (b + c)
```

Le même calcul sur les entiers (exact) et sur les flottants (approché) peut apporter des résultats différents :

```
>>> 2.0 == 2
True
>>> 2.**53 + 1 == 2**53 +1
False
```

Exemple 5 :

Justifier les cas suivants

```
>>> int (1000000000000000000 / 3)
333333333333333312
>>> 1000000000000000000 // 3
333333333333333333
```

- On l'a vu dans le TP précédent, la capacité des entiers est illimitée. Ce n'est pas le cas des flottants.

```
>>> 1.e309
inf
>>> 2.0**1024
OverflowError
```

- Dans une boucle les erreurs d'arrondis peuvent s'accumuler jusqu'à obtenir un résultat totalement faux. De plus il est difficile de prévoir l'erreur.

III. Intégration numérique

1 - Méthode des rectangles

a) Représentation numérique d'une fonction

Pour une fonction quelconque, la représentation numérique d'une fonction est associée à un domaine de définition du type $I = [a;b]$. On évalue cette fonction sur une « grille » de N valeurs de x prise entre a et b. On utilise à ce titre le plus souvent la fonction `linspace` (avec le package Numpy ou pylab)

```
from numpy import *
a=0   # borne inf
b=2   # borne sup
N=100 # nombre de points
x=linspace(a,b,N)
```

Cette fonction permet de créer une liste de valeurs de x. Le pas de discrétisation en x est donné par

$$\Delta x = \frac{b-a}{N-1}$$

Remarque 6 :

Il faudra être très vigilant sur le choix du nombre de points par rapport au nombre d'intervalles ! Pour N points il y a N-1 intervalles, pour N intervalles, il faut N+1 points !

La représentation d'une fonction mathématique, même continue est une succession de coordonnées de points $(x_i,y_i = f(x_i))$. Cette représentation discrète (point par point) est utilisée par tous les instruments numériques traitant de signaux (capteur de vitesse, de température...).

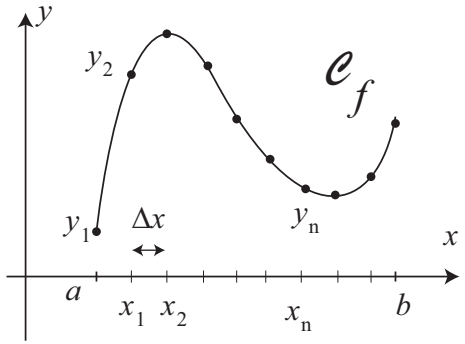


FIGURE 1 – Représentation numérique d'une fonction

Un signal analogique noté $s(t)$ est un signal **continu**. La représentation numérique d'un signal analogique est constitué de deux liste T et S. En général, T est de la forme $k \times T_e$ où T_e est la période d'échantillonnage du signal.

On a alors la représentation suivante :

$$s(T[k]) = S_k$$

En présence d'accéléromètre, il peut être intéressant d'intégrer le signal fourni par l'accéléromètre pour obtenir la vitesse puis la position.

b) Principe

La méthode des rectangles consiste à approximer la fonction f par une fonction en escalier. On considère un entier n et un pas de subdivision $\frac{b-a}{n}$. Pour tout entier k de $[\![0,n]\!]$, on pose $a_k = a + k\frac{b-a}{n}$. Sur l'intervalle $[a_k,a_{k+1}]$, on approxime f par la fonction constante égale à $f(a_k)$ (respectivement $f(a_{k+1})$).

On prend comme valeur approchée de l'intégrale de f sur $[a,b]$ l'intégrale de la fonction en escalier ainsi construite, c'est-à-dire la somme des aires des rectangles qui vaut S_n (respectivement S'_n) : cf figures ci-dessous.

Définition : *sommes de Riemann*

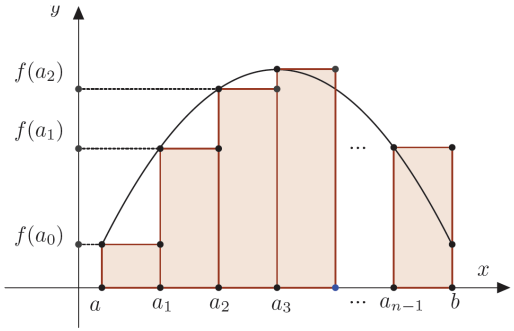
Soit $f : [a,b] \to \mathbb{R}$ continue sur $[a,b]$.

On appelle sommes de Riemann les suites (S_n) et (S'_n) définies pour tout $n \in \mathbb{N}^*$ par :

$$S_n = \frac{b-a}{n}\sum_{k=0}^{n-1} f\left(a + k\frac{b-a}{n}\right) = \tfrac{b-a}{n}f(a) + \tfrac{b-a}{n}f\left(a + \tfrac{b-a}{n}\right) + \ldots + \tfrac{b-a}{n}f\left(a + (n-1)\tfrac{b-a}{n}\right)$$

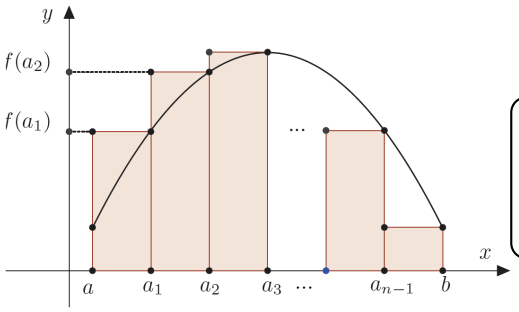
et $$S'_n = \frac{b-a}{n}\sum_{k=1}^{n} f\left(a + k\frac{b-a}{n}\right) = \tfrac{b-a}{n}f\left(a + \tfrac{b-a}{n}\right) + \tfrac{b-a}{n}f\left(a + 2\tfrac{b-a}{n}\right) + \ldots + \tfrac{b-a}{n}f(b)$$

interprétation graphique de S_n :



La somme des aires des rectangles est S_n.
avec $a_k = a + k\frac{b-a}{n}$ pour tout $k \in [\![0, n-1]\!]$

interprétation graphique de S'_n :



La somme des aires des rectangles est S'_n.
avec $a_k = a + k\frac{b-a}{n}$ pour tout $k \in [\![0, n-1]\!]$

Remarque 7 :

Pour une intégration « à gauche » (en prenant le premier point), la méthode des rectangles minore (resp. croissante) l'intégrale vraie pour une courbe croissante (resp. décroissante)

Propriété :

Soit $f : [a,b] \to \mathbb{R}$ continue sur $[a,b]$.

Alors les sommes de Riemann (S_n) et (S'_n) convergent vers $\displaystyle\int_a^b f(t)\,dt$.

Propriété : *vitesse de convergence et majoration de l'erreur*

Soit $f \in \mathcal{C}^1([a,b],\mathbb{R})$. Posons $M = \max_{x\in[a,b]} |f'(x)|$.

Alors $\left|\displaystyle\int_a^b f(t)\,dt - S_n\right| \leq \frac{M(b-a)^2}{2n}$ ce qui implique $\displaystyle\int_a^b f(t)\,dt - S_n = O_{+\infty}\left(\frac{1}{n}\right)$.

Remarque 8 :

La dernière inégalité permet d'obtenir une majoration de l'erreur commise en approximant $\int_a^b f(t)\,dt$ par S_n. Ce résultat reste valable si on remplace S_n par S'_n.

Exemple 6 :

Voici un code Python donnant une valeur approchée de $\displaystyle\int_0^\pi \sin t\,dt$ avec 100 rectangles. On suppose que le module pylab a été importé.

```
1  N=100
2  dt=pi/N
3  t=linspace(0,pi,N+1)
4  S=0
5  for k in range(N):
6      S+=sin(t[k])*dt
7  #ou S+=sin(t[k+1])*dt
```

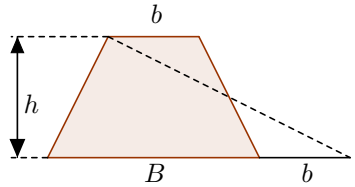
2 - Méthode des trapèzes

a) Principe

La méthode des trapèzes consiste à approximer la fonction f par une fonction continue affine par morceaux. On considère un entier n et un pas de subdivision $\frac{b-a}{n}$. Pour tout entier k de $[\![0,n]\!]$, on pose $a_k = a+k\frac{b-a}{n}$. Sur l'intervalle $[a_k,a_{k+1}]$, on approxime f par la fonction affine qui joint les points $(a_k,f(a_k))$ et $(a_{k+1},f(a_{k+1}))$.

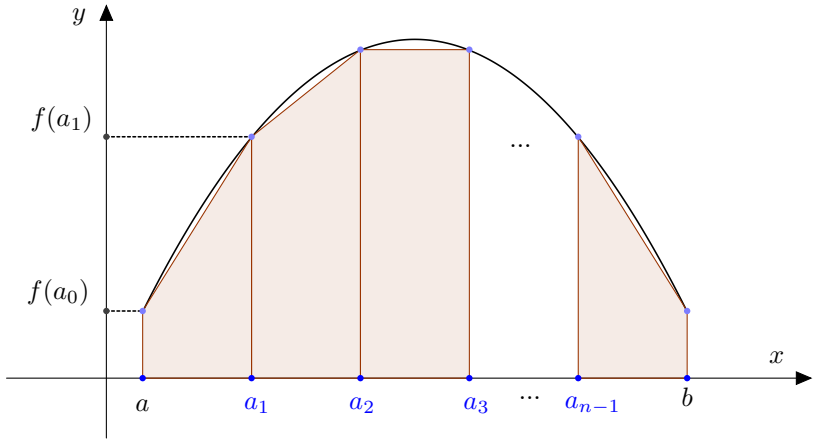
On prend comme valeur approchée de l'intégrale de f sur $[a,b]$ l'intégrale de la fonction continue affine par morceaux, c'est-à-dire une somme d'aires de trapèzes notée T_n.

L'aire d'un trapèze de petite base b, de grande base B et de hauteur h est .....



interprétation graphique de T_n :
Au lieu d'une somme de rectangles, on effectue la somme T_n d'aires de trapèzes de sommets $(a_k,0)$, $(a_{k+1},0)$, $(a_k,f(a_k))$ et $(a_{k+1},f(a_{k+1})$
avec $a_k = a + k\frac{b-a}{n}$ pour tout $k \in [\![0, n-1]\!]$

Chaque trapèze a pour aire : $(a_{k+1}-a_k)\,\dfrac{f(a_k)+f(a_{k+1})}{2} = \dfrac{(b-a)}{2n}\,(f(a_k)+f(a_{k+1}))$



Remarque 9 :
La méthode des trapèzes minore (resp. majore) l'intégrale vraie si la fonction est concave (resp. convexe)

Propriété :
Soit $f : [a,b] \to \mathbb{R}$ continue sur $[a,b]$.
Soit $n \in \mathbb{N}^*$. On pose $a_k = a + k\frac{b-a}{n}$ pour tout $k \in [\![0, n-1]\!]$.
La somme des aires des trapèzes est :

$$T_n = \frac{b-a}{n}\sum_{k=0}^{n-1}\frac{f(a_k)+f(a_{k+1})}{2}$$

La suite (T_n) converge vers $\displaystyle\int_a^b f(t)\,dt$.

Exemple 7 :
Voici un code Python donnant une valeur approchée de $\displaystyle\int_0^\pi \sin t\,dt$ avec 100 trapèzes. On suppose que le module pylab a été importé.

```
1  N=100
2  dt=pi/N
3  t=linspace(0,pi,N+1)
4  S=0
5  for k in range(N):
6     S+=(sin(t[k])+sin(t[k+1]))*dt/2
```

b) Convergence

Propriété : *vitesse de convergence et majoration de l'erreur*

Soit $f \in \mathcal{C}^2$ ([a,b],$\mathbb{R}$). Posons $M_2 = \max_{x\in[a,b]} |f''(x)|$.

Alors $\left|\int_a^b f(t)\,dt - T_n\right| \leq \frac{M_2(b-a)^3}{12n^2}$ ce qui implique $\int_a^b f(t)\,dt - T_n = O_{+\infty}\left(\frac{1}{n^2}\right)$. La méthode des trapèzes converge beaucoup plus vite que la méthode des rectangles.

IV. Résolution EDO 1° ordre

1 - Problème de Cauchy

En analyse, un problème de Cauchy est un problème constitué d'une équation différentielle dont on recherche une solution vérifiant une certaine condition initiale. Le théorème de Cauchy-Lipschitz assure l'existence et l'unicité d'une solution au problème de Cauchy.

Définition :

Les équations différentielles linéaires d'ordre 1 de la forme

$$y' + a(x)y = \delta(x)$$

où a et δ sont des fonctions continues sur I.

Ces équations se retrouvent très fréquemment en physique : chute avec frottement, charge ou décharge d'un condensateur, d'une bobine, évolution thermique...

Exemple 8 :

Résoudre sur $\mathbb{R}$: $y' + ay = 0$, avec $y(0) = 1$. a est une constante réelle.

2 - Approximation numérique

Le principe de la méthode d'Euler repose sur une résolution pas à pas de l'équation différentielle à partir de la condition initiale. Tentons de résoudre numériquement l'équation suivante :

$$y' + ay = 0$$

avec $y(0)$ connu.

Utilisons l'approximation du nombre dérivée pour les points $x_0 = 0$ et $x_1 = 0 + \Delta x$. En remplaçant dans l'équation différentielle, on obtient :

$$y'(0) = -ay(0) \approx \frac{y(0+\Delta x) - y(0)}{\Delta x}$$

Il suffit maintenant d'isoler $y(\Delta x)$ pour obtenir :

$$y(\Delta x) = y(0) + \Delta x \times \big(-ay(0)\big)$$

Il est important de remarquer que la valeur $y(\Delta x)$ n'est obtenue qu'avec des informations sur les valeurs "antérieures" en $x = 0$. Une fois connue la valeur de y en $x = \Delta x$, il est possible de déterminer le terme en $x = 2\Delta x$ et de proche en proche de déterminer les points de la fonction :

$$y(2\Delta x) = y(\Delta x) + \Delta x \times \big(-ay(\Delta x)\big)$$
$$y(3\Delta x) = y(2\Delta x) + \Delta x \times \big(-ay(2\Delta x)\big)$$
$$\ldots$$

La structure de la solution approchée est celle d'une suite avec $u_n = y(n\Delta x)$:

$$u_0 = y(0)$$
$$u_{n+1} = u_n + \Delta x \times (-au_n)$$

Exemple 9 :

Écrire un code python permettant de résoudre l'équation différentielle $y' + y = 0$, avec $y(0) = 1$ pour $x \in [0,5]$ avec $N = 100$ points.

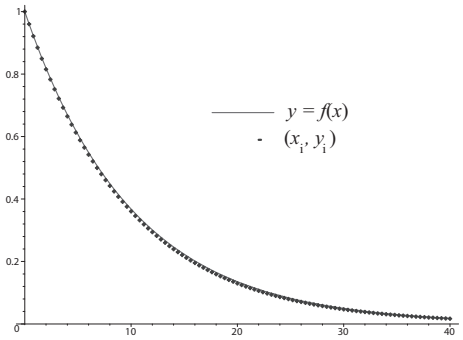


FIGURE 2 – Résolution numérique de $y' + 0{,}1 \times y = 0$, avec $y(0) = 1$

```
xmin,xmax=0,5
N=100
X=linspace(xmin,xmax,N)
h=X[1]-X[0]
Y=N*[0]
for k in range(0,N-1):
    Y[k+1]=Y[k]+h*(-Y[k])
```

3 - Convergence de la solution

a) Ecart à la solution réelle

L'écart entre la solution numérique et la « vraie » solution dépend du nombre de points dans l'intervalle de résolution.

Définition :
On peut montrer que l'erreur est proportionnelle au pas Δx, on parle alors de méthode du **premier ordre**.

Il existe des méthodes quantitatives pour s'assurer qu'une solution numérique soit proche d'une solution analytique. On parle alors de solution **convergée**. L'intérêt de la méthode d'Euler est de résoudre des cas non-analytique, c'est à dire dont nous ne connaissons pas la solution. Afin de s'assurer que le nombre de points choisi est suffisant, nous effectuerons donc un test « à l'oeil » : c'est à dire que nous considérerons que le nombre de points est suffisant lorsque l'allure de la solution n'est plus modifiée :

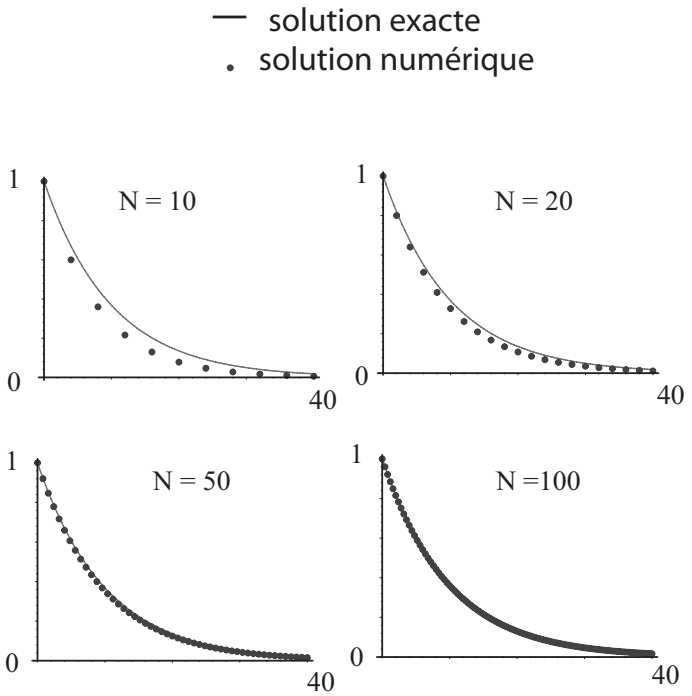


FIGURE 3 – Convergence de la solution numérique

Définition :

On dit qu'une équation différentielle discréditée est consistante par rapport à l'équation différentielle réelle si l'écart des solutions tend vers 0 lorsque le pas de discrétisation tend vers 0

b) Influence de l'équation différentielle

Avec un même pas, deux équations différentes peuvent présenter des écarts importants selon que la solution finale varie rapidement ou non. En effet, la méthode d'Euler utilisant une approximation du nombre dérivée, plus celui-ci est important, plus les écarts seront importants. Résolvons l'équation différentielle $y' = -a \times y$ avec $y(0) = 1$. L'écart à la solution numériques augmente rapidement avec la valeur de a.

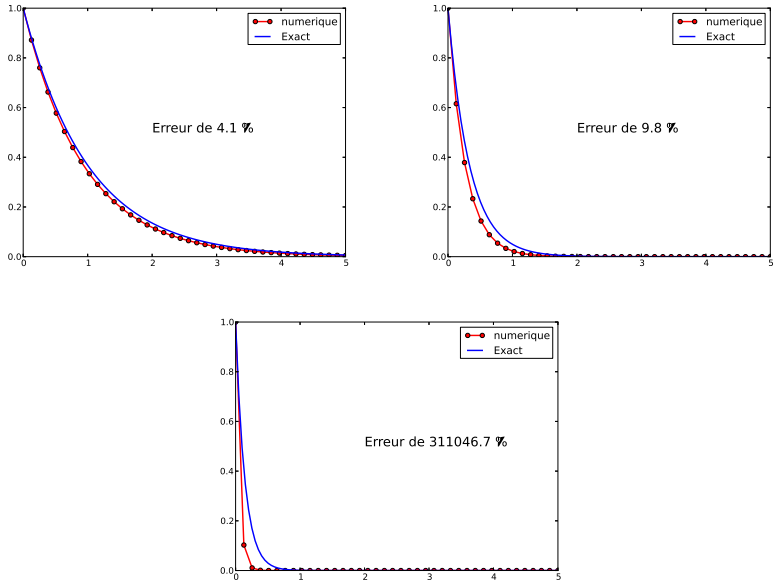


FIGURE 4 – Convergence de la solution numérique de l'équation $y' + ay = 0$ pour $a = 1$, $a = 3$ et $a = 7$.

c) instabilité

Si le pas est trop faible, la solution peut même diverger. C'est parfois le cas lorsque la solution doit présenter des oscillations.

$$y'(x) = -y(x)$$

On remarque que la relation de récurrence, pour une équation de ce type est de la forme :

$$Y[k+1] = Y[k] + h * (-aY[k]) = Y[k](1 - ah)$$

Les éléments $Y[k]$ suivent une relation de récurrence d'une suite géométrique de raison $(1 - ah)$. On en déduit que la solution diverge si $|1 - ah| > 1$.

Définition :

Pour des équations du premier ordre, la solution est stable si l'on peut trouver une constante V telle que

$$-1 \leq \frac{X[k+1] - V}{X[k] - V} \leq 1$$

4 - Application en physique non linéaire

a) Présentation

Les problèmes du prix du millénaire comptent sept défis mathématiques réputés insurmontables posés par le Clay Mathematical Institute en 2000. La résolution de chacun des problèmes est dotée d'un prix d'un million de dollars américains offert par l'institut. À ce jour, six des sept problèmes demeurent non résolus. L'un deux concerne la mécanique des fluides avec l'équation de Navier Stokes :

$$\rho \left[\frac{\partial \vec{v}}{\partial t} + \left(\vec{v} \cdot \vec{\nabla} \right) \vec{v} \right] = -\vec{\nabla} p + \mu \left[\nabla^2 \vec{v} + \frac{1}{3} \vec{\nabla} \left(\vec{\nabla} \cdot \vec{v} \right) \right] + \rho \vec{f}$$

La difficulté de cette équation est l'apparition de produit de dérivée avec d'autres fonctions. Ces équations sont fondamentales pour expliquer le comportement des fluides. Il existe des solutions partielles, mais aucune solution générale n'est encore proposée.

b) Chute à haute vitesse

Considérons la chute d'un parachutiste, ce dernier est soumis à son poids $\vec{P} = m\vec{g}$ et à une force de frottement de l'air. Une force de frottement dans l'air est modélisée par une force proportionnelle au carré de la vitesse :

$$\vec{f} = -av^2\frac{\vec{v}}{|v|}$$

La 2e loi de Newton donne

$$m\vec{a} = m\vec{g} + \vec{f}$$

En projetant selon l'axe Oz, on en déduit que

$$m\frac{dv}{dt} = mg - av^2$$

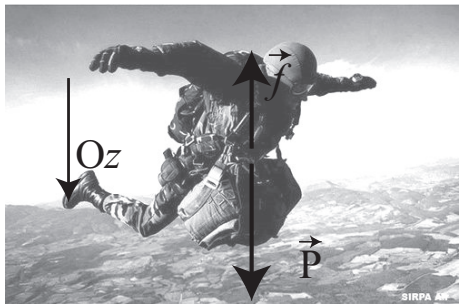


FIGURE 5 – Bilan des forces sur un parachutiste

Exemple 10 :

En décomposant le temps sur une grille de pas Δt, écrire la relation de récurrence entre $u_{n+1} = v\big(n\Delta t\big)$ et $u_{n+1} = v\big((n+1)\Delta t\big)$

Remarque 10 :

Dans le cas présenté avec une chute parfaitement verticale, les puristes pourront montrer que la solution de cette équation est analytique et fait intervenir la fonction `arctanh`...

V. Résolution d'une équation différentielle du second ordre

1 - Présentation

Définition :

En mathématiques : les équations différentielles linéaires d'ordre 2 à coefficients constants

$$ay'' + by' + cy = \delta(x),$$

où a,b,c sont des éléments de $\mathbb{R}$, $a \neq 0$, et δ une fonction continue sur I.

En physique, on rencontre fréquemment ces équations pour des systèmes oscillants avec ou sans amortissement. L'équation différentielle canonique est alors donnée par :

$$\frac{d^2x}{dt^2} + \frac{\omega_o}{Q}\frac{dx}{dt} + \omega_0^2 x = 0$$

On remarquera que, par rapport aux mathématiques, il y a simplement :

$$\omega_0/Q = b/a \quad \text{et} \quad \omega_0^2 = c/a$$

2 - Résolution numérique

Exemple 11 :

Résoudre par la méthode d'Euler l'équation de l'oscillateur harmonique :

$$\ddot{x} + \omega_0^2 x = 0$$

, avec $\omega_0 = 1\,rad.s^{-1}$, pendant 10 s avec 100 points.

On peut donc décomposer l'équation différentielle précédente du 2° ordre en deux équations du 1° ordre :

$$\frac{dx}{dt} = \dot{x}$$

$$\frac{d\dot{x}}{dt} = -\omega_0^2 x$$

```
Tmin=0
Tmax=10
N=100
T=linspace(Tmin,Tmax,N)
h=(Tmax-Tmin)/(N-1)
X=N*[0]
V=N=[0]
X[0]=1
V[0]=0
wo=1
for k in range(0,N-1):
    X[k+1]=X[k]+h*V[k]
    V[k+1]=V[k]+h*(- wo**2 X[k])
```

On peut donc décomposer l'équation différentielle précédente du 2° ordre en deux équations du 1° ordre :

$$\frac{dx}{dt} = \dot{x}$$

$$\frac{d\dot{x}}{dt} = -\frac{\omega_0}{Q}\dot{x} - \omega_0^2 x$$

Exemple 12 :

Résoudre par la méthode d'Euler l'équation de l'oscillateur harmonique :

$$\ddot{x} + \frac{\omega_0}{Q}\dot{x} + \omega_0^2 x = 0$$

, avec $\omega_0 = 1\, rad.s^{-1}$, $Q = 5$, pendant 4 s avec 100 points. Les conditions initiales, sont $x(0) = 1$, et $\dot{x}(0) = 0$.

Afin de résoudre ces équations, on considère deux listes dont les relations de récurrence entre les différents éléments consistent en des suites imbriquées définies par la position et la vitesse :

$$X[k+1] = X[k] + h * V[k]$$

$$V[k+1] = V[k] + h * (-wo/Q * V[k] - wo * *2X[k])$$

```
Tmin=0
Tmax=4
N=100
T=linspace(Tmin,Tmax,N)
h=(Tmax-Tmin)/(N-1)
X=N*[0]
V=N=[0]
X[0]=1
V[0]=0
wo=1
Q=5
for k in range(0,N-1):
    X[k+1]=X[k]+h*V[k]
    V[k+1]=V[k]+h*(-wo/Q*V[k] - wo**2 X[k])
```

3 - Pas fixe

```
Tmax=10
h=0.1
X[0]=[1]
V[0]=[0]
t=0
T=[t]
k=0
while t<Tmax:
    X.append(X[k]+h*V[k])
    V.append(V[k]+h*(-wo/Q*V[k] - wo**2 X[k]))
    k+=1
    t+=h
    T.append(t)
```

4 - Résolution d'une équation différentielle 2D

a) Équations

La trajectoire d'un corps dans l'espace peut souvent être ramené à un plan. Le principe fondamental de la dynamique donne les équations différentielles régissant l'évolution de la vitesse en fonction des forces en présence. Pour une chute libre, sans frottement, les équations sont :

$$\begin{cases} \dot{v}_x = 0 \\ \dot{v}_y = -g \end{cases} \tag{4}$$

avec $v_x(0) = 0$ et $v_y(0) = V_0$.

Le découplage des équations en vue d'une résolution numérique conduit à un système de 4 équations :

$$\begin{cases} \dot{v}_x = & 0 \\ \dot{v}_y = & -g \\ \dot{x} = & v_x \\ \dot{y} = & v_y \end{cases} \tag{5}$$

b) Code Python correspondant

```
Tmin=0
Tmax=10
N=100
T=linspace(Tmin,Tmax,N)
h=(Tmax-Tmin)/(N-1)
X,Y,Vx,Vy=N*[0],N*[0],N*[0],N*[0]
X[0],Y[0],Vx[0],Vy[0]=0,0,1,1
for k in range(0,N-1):
    Vx[k+1]=Vx[k]+0
    Vy[k+1]=Vx[k]+h*(-g)
    X[k+1]=X[k]+h*Vx[k]
    Y[k+1]=Y[k]+h*Vy[k]
```