informatique

DUT INFO 1 / 2015-2016

Mis à jour le 9 mai 2016 à 13:55

Au delà des réels: méthodes numériques en informatique

Guillaume CONNAN

IUT NANTES Université de Nantes

UNIVERSITÉ DE NANTES

TABLE DES MATIÈRES

CHAPITRE 1

IEEE 754-2008

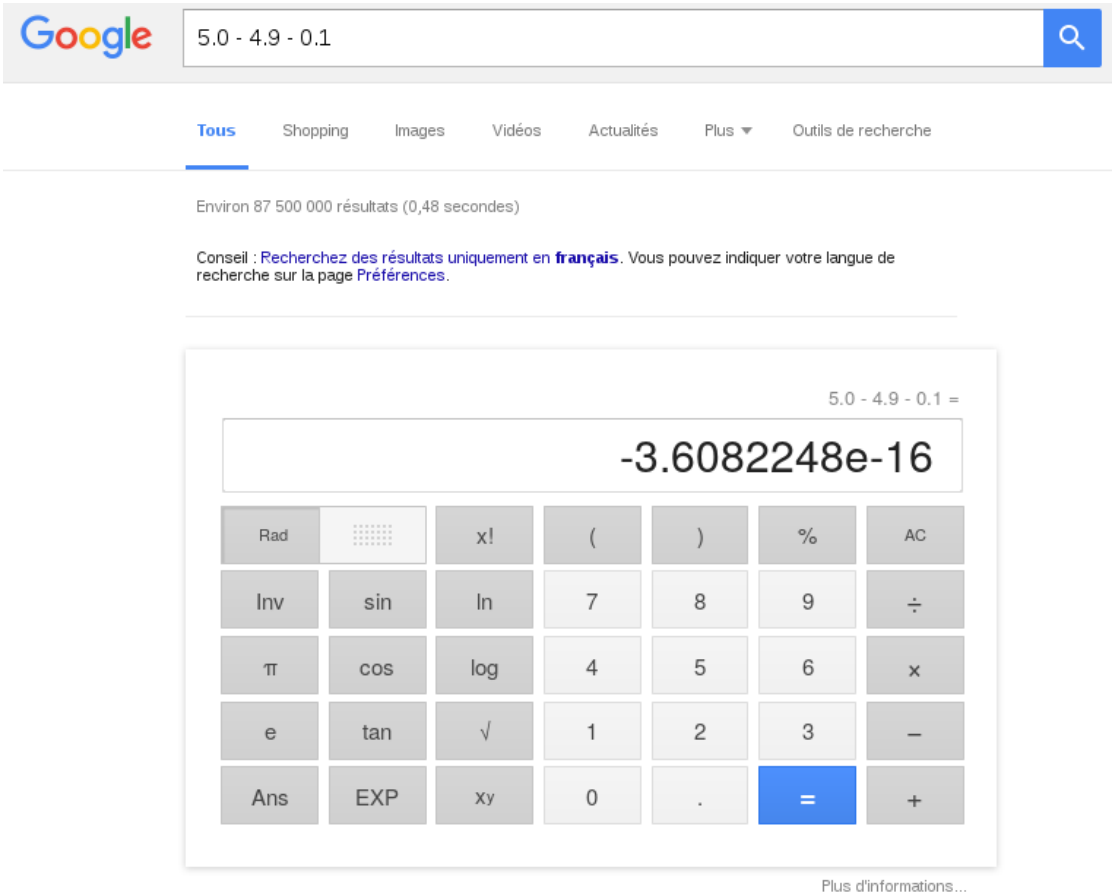
Des avions qui s'écrasent, des fusées qui s'écrasent, des missiles qui tuent les amis, les indices boursiers totalement faux, les résultats des élections totalement faux, les jeux qui plantent,...
La méconnaissance du traitement des nombres sur machine (et non pas le traitement lui-même !) peut être tragique ou au mieux catastrophique.
Nous allons initier notre exploration de ce monde mystérieux que vous utilisez chaque jour en plongeant dans la machine.

1 Les nombres à virgule flottante

Les problèmes

W.KAHAN (1933 -)

Wiliam KAHAN est un mathématicien et informaticien canadien, lauréat du prix TURING en 1989 et principal artisan de la norme IEEE 754 publiée pour la première fois en 1985.
Les problèmes liés à la manipulation des VF (nombres à virgule flottante, ou *floating point (FP)* in english of speciality), sont nombreux.
Citons notamment le célèbre cas du missile américain *Patriot* qui manqua sa cible, un missile *Scud* irakien, durant la première guerre du Golfe en février 1991. Le *Patriot* poursuit sa cible en mesurant le temps mis par les ondes radars pour revenir après rebond sur la cible. Or le système mesurait le temps en dizaines de seconde (drôle d'idée) puis multipliait ce temps par 0,1 pour obtenir le temps en secondes et utilisait un registre à 24 bits en virgule fixe.
Après cent heures d'utilisation (l'armée US ne chôme pas), un écart de 0,34 seconde est apparu (pourquoi ? cf en TD...). Un *Scud* volant à 1.676 m/s, le *Patriot* a visé 500 m à côté. Cette erreur causa la mort de 28 soldats tandis que 98 étaient blessés.
Il y a eu beaucoup d'autres cas célèbres, même chez les plus puissants :



Bon, ne soyons pas de mauvaise foi :

```
In [4]: 5 - 4.9 - 0.1
Out[4]: -3.608224830031759e-16
```

Même des logiciels spécialisés, très chers (Maple ©), se trompent :

```
> evalf(sin(2^100));
                              .4491999480
```

```
> evalf(sin(2^100),20);
                      -.58645356896925826300
> evalf(sin(2^100),30);
                   .199885621653625738215132811525
> evalf(sin(2^100),40);
              -.872183605418267309780719778213470559324 3
```

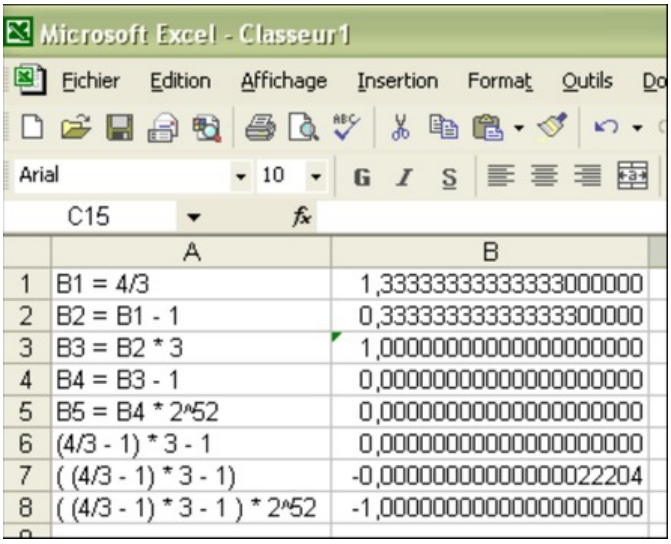
Heureusement, le libre sauve la mise...Giac/XCAS :

```
1>> evalf(sin(2^100))
-0.872183605418
```

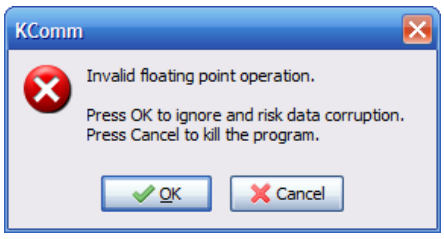
Python :

```
In [6]: sin(2**100)
Out[6]: -0.8721836054182673
```

Un exemple rigolo : les élections au Schleswig-Holstein en 1992 (merci à Paul Zimmermann).
Les partis ayant moins de 5% des voix n'ont pas de siège.
Au dépouillement, le parti écologiste obtient exactement 5%.
Après l'annonce officielle des résultats, on découvre que le parti écologiste n'a en fait que 4.97% des voix !
Le programme affichant les résultats arrondit à un seul chiffre après la virgule, et donc a arrondi 4.97 à 5.0.
L'erreur corrigée, le siège est retiré aux verts, et attribué au SPD, qui avec 45 sièges sur 89 obtient la majorité absolue.
Un autre exemple qui fait pitié (merci à William Kahan) :



Un dernier qui vous touchera plus :



Le problème des VF est apparu sur machine en 1941 : Konrad Zuse a proposé son Z3 avec une mantisse de 14 bits, un exposant de 7 bits, un bit de signe le tout en base 2.
En effet, les humains préfèrent compter en base 10...et n'arrêtent pas de faire des erreurs d'arrondis suite à leur manie (pourquoi ? cf en TD...). Certaines machines calculent donc en base

10 (les calculettes, Maple©, les machines dédiées à la Finance,...). La norme IEEE 754-2008 introduit aussi des calculs en base 10.

La machine pour l'instant est basée sur une logique à deux états et préfère donc la base 2 (ou 8 ou 16).

Николай Петрович Брусенцов(1925 -)

Certaines machines ont pourtant été conçues selon une logique à trois états (Vrai, Faux, j'en sais rien) comme le Сетунь soviétique mis au point en 1958 par une équipe dirigée par Николай Петрович Брусенцов et le célèbre mathématicien Сергей Львович Соболев. L'histoire de cette machine est édifiante : la production des Сетунь fut confiée à l'*usine des machines mathématiques* qui en fabriqua très peu car le Сетунь était trop bon marché ! De plus il a fonctionné des années durant sans aucune panne ni besoin de pièces de rechange ce qui n'est pas très bien vu dans le milieu informatique :-) Enfin le Сетунь réglait le problème crucial du *tiers-exclus* et obéissait aux canons les plus exigeants d'E. Dijkstra. Bref, les autorités soviétiques ont décidé de le découper en morceaux et de mettre les restes dans une décharge. De zélés informaticiens ont toutefois réussi à en sauvé un exemplaire en le cachant dans un grenier.

Vous utilisez vous-même une logique ternaire en SQL pour gérer `NULL`. Les machines quantiques peuvent travailler avec des *qtrits*. Vous verrez ça un jour, jeunes padawans.

Autre problème : le profusion d'appellations qui sèment la confusion ! En C (cf dinechin) :

- `char` est un entier codé sur 8 bits. C'est l'abréviation du nom commun *character* ;
- `int` est l'abréviation du non commun *integer* qui désigne un nombre entier comme en mathématiques mais cependant $2147483647 + 1 = -2147483648$...
- `short` et `long` sont des adjectifs désignant des entiers de taille différentes ;
- `float` est un verbe ;
- `single` ? Simple quoi ?
- `double` ? Mais le double de quoi ?
- `long double` : deux adjectifs à suivre ?

Alors il faut clarifier tout ça !

Quelle précision ?

En physique, on ne peut guère aller en deçà de grandeurs de l'ordre de 10^{-15} alors pourquoi s'embêter ?

```
In [9]: 3*0.1
Out[9]: 0.30000000000000004

In [10]: sum([0.1 for _ in range(1000000)])
Out[10]: 100000.00000133288
```

Il faut parfois aller bien plus loin que 10^{-15} pour avoir une précision de 10^{-15} : c'est bien là le problème.

On cherchera avec attention l'exercice Recherche 1 - 6 page 28

Pourtant, avec 50 bits on peut coder la distance Terre-Lune avec une erreur de l'ordre de la taille d'une bactérie !

Qu'est-ce qui se passe ? On ne peut plus faire de calcul ? On ne pourra pas fabriquer de jeux vidéos pour dégommer plein de méchants ?

Pas de panique : il faut juste avoir en tête que **LES NOMBRES RÉELS N'EXISTENT PAS : TOUT CE QUE VOUS AVEZ VU AU LYCÉE N'EST QU'ILLUSION !**

Oubliez donc les réels : les nombres que vous allez manipuler sont différents mais sont tout aussi maîtrisables. Il suffit de connaître leur arithmétique, **l'arithmétique des nombres à virgule flottante** qui obéissent à la norme IEEE 754.

Des problèmes subsistent néanmoins. Par exemple, on sait très bien ce que donnera un calcul selon un format donné...le problème est qu'on ne sait pas toujours quel format est utilisé...

2 La norme IEEE 754

En bref

Nous nous contenterons, sauf mention contraire, de travailler en base 2. Vous en avez parlé en archi : un VF est représenté par un bit de signe, une mantisse (ou significande en franglais) et un exposant :

$$v = (-1)^s \times m \times 2^E$$

Les formats habituels sont le *binary32* ou *single* avec $(\#E, \#m) = (8, 24)$ et le *binary64* ou *double* avec $(\#E, \#m) = (11, 53)$.

Pour simplifier nos présentations, on illustrera souvent nos propos avec des représentations *toy7* plus petites, comme par exemple $(\#E, \#m) = (3, 4)$.

La mantisse doit dans la mesure du possible vérifier $1 \leqslant m < 2$ ce qui minimise l'exposant et apporte plus d'informations. Par exemple, en *toy7* :

$$\begin{aligned} v &= 0,010\,01 \quad \times 2^0 \\ v &= 0,100\,1 \quad \times 2^{-1} \\ v &= 1,001 \quad \times 2^{-2} \end{aligned}$$

C'est la *forme normale* d'un VF. De plus, cela se permet de se passer du premier 1 qui est implicite. On notera

$$m = 1, f$$

On peut également gagner de la place en ne stockant pas le signe de l'exposant : il suffit de le translater de $2^{(\#E)-1} - 1$...POURQUOI ?

$$e = E + 2^{(\#E)-1} - 1$$

Exemple

Représentons $0,75_{10}$ en *toy7*.

$$0,75_{10} = \frac{3_{10}}{2^2_{10}} = 11_2 \times 2^{-2} = 1,100_2 \times 2^{-1}$$

L'exposant stocké e vérifie avec le décalage :

$$e - \left(2^{3-1} - 1\right) = E = -1 \leftrightarrow e = -1 + 3 = 2 = 10_2$$

Ainsi :

s (1 bit)	e (3 bits)			f (3 bits)		
0	0	1	0	1	0	0

On doit également s'occuper des exposants extrêmes qui ne correspondent pas à des formes normales (cf TD).

À retenir

Petit algo pour convertir un nombre de valeur absolue inférieure à 1 en base 2

Soit $x = a_1 \times 2^{-1} + a_2 \times 2^{-2} + a_3 \times 2^{-3} + \cdots + a_n \times 2^n$ l'écriture de x tel que $|x| < 1$ en base 2 avec les a_i égaux à 0 ou 1.

Alors $2x = a_1 + a_2 \times 2^{-1} + a_3 \times 2^{-2} + \cdots + a_n \times 2^{n-1}$: a_1 est donc la partie entière de $2x$.

Ensuite $2(2x - a_1) = a_2 + a_3 \times 2^{-1} + \cdots + a_n \times 2^{n-2}$: a_2 est alors la partie entière de $2(2x - a_1)$ und so weiter...

Les normaux, les sous-normaux et les paranormaux

Nous venons de voir que l'exposant E, qui est un entier signé, est stocké sur $\#E$ bits sous la forme $e = E + 2^{(\#E)-1} - 1$.
Comme e est un entier naturel stocké sur $\#E$ bits, il varie entre $e_{\min} = 0$ et $e_{\max} = 2^{\#E} - 1$.
Par exemple, en *toy7*, $\#E = 3$ donc $e_{\min} = 000$ et $e_{\max} = 111 = 1000 - 1$.
CEPENDANT les valeurs extrêmes de E sont réservées pour des cas spéciaux.
Cela donne :

$$E_{\min} = \left(e_{\min} - 2^{(\#E)-1} + 1\right) + 1 = 0 - 2^{3-1} + 1 + 1 = -2$$
$$E_{\max} = \left(e_{\max} - 2^{(\#E)-1} + 1\right) - 1 = (2^3 - 1) - 2^{3-1} + 1 - 1 = 7 - 4 + 1 - 1 = 3$$

En effet, des problèmes apparaissent avec les formes normales :

— on ne peut pas représenter zéro sous forme normale ;

— des nombres du type $0.\underbrace{000\ldots00}_{\#E}11 \times 2^0 = 1.1 \times 2^{-(\#E+1)}$ ne sont pas représentables sous forme normale car il faudrait un exposant plus petit que $E_{\min}$.

La norme IEEE 754 a donc introduit les nombres *sous-normaux* pour remplir le vide qui existerait entre 0 et le plus petit nombre normal (cf exercice Recherche 1 - 11 page 30).
Pour les signaler et les distinguer des normaux, la norme impose que leur exposant décalé e soit 0 : $e_{\min}$ est donc une valeur réservée.
Un cas particulier : si $e = e_{min}$ et $f = 0$, alors la norme ne considère ce nombre ni comme un nombre normal ni comme un nombre sous-normal. C'est la représentation de zéro.

À retenir

Il y a deux zéros ! En *toy7* :

+0 :

s (1 bit)	e (3 bits)			f (3 bits)		
0	0	0	0	0	0	0

−0 :

s (1 bit)	e (3 bits)			f (3 bits)		
1	0	0	0	0	0	0

Des tests `x == 0` pourraient alors échouer ! La norme impose donc que les deux zéros soient considérés comme égaux :

```
In [11]: x = 1 / 1e500

In [12]: x
Out[12]: 0.0

In [13]: x == 0
Out[13]: True

In [14]: y = -1 / 1e500
```

```
In [15]: y
Out[15]: -0.0

In [16]: y == 0
Out[16]: True

In [17]: x == y
Out[17]: True
```

Reprenons à présent un exemple classique : vous voulez calculer une norme $N = \sqrt{x^2 + y^2}$ avec $x = 1 \times 2^3$ et $y = 1,1 \times 2^3$ en *toy7*.
Les carrés de ces nombres posent problème : leur exposant est trop grand. On est dans un cas de *dépassement* ou d'*overflow* en english de spécialité.
Une première idée serait de remplacer x^2 et y^2 par le plus grand nombre disponible $1,111 \times 2^4$ soit :

s (1 bit)	e (3 bits)			f (3 bits)		
0	1	1	1	1	1	1

Alors x^2 puis y^2 puis $x^2 + y^2$ seraient remplacés par $1,111 \times 2^4$.
On aurait donc $N = \left(1,111 \times 2^4\right)^{1/2}$ arrondi à $1,010 \times 2^2$ ce qui est dramatiquement faux et pourraient envoyer des *Patriots* sur le toit du bâtiment B.
La norme introduit donc deux infinis codés en *toy7* :

$+\infty$:

s (1 bit)	e (3 bits)			f (3 bits)		
0	1	1	1	0	0	0

$-\infty$:

s (1 bit)	e (3 bits)			f (3 bits)		
1	1	1	1	0	0	0

Soit, plus généralement, $e = e_{\max}$ et $f = 0$.
On retrouve alors les limites habituelles vues au lycée (ah, quand même...) :

```
In [18]: 1 + 1e500
Out[18]: inf


In [19]: x = 1e500

In [20]: x
Out[20]: inf
```

```
In [22]: 1 + x
Out[22]: inf

In [23]: x**3
Out[23]: inf

In [24]: 1 / x
Out[24]: 0.0
```

Nous verrons bien d'autres exemples en exercice.

Vous avez aussi parlé au lycée des **FORMES INDÉTERMINÉES** lors de l'étude des limites. Par exemple, que dire de :

$$\lim_{x \to +\infty} x^2 - x$$

Vous êtes muets ? Demandons à la machine :

```
In [28]: x**2 - x
Out[28]: nan
```

Une manière exotique de régler le problème est d'étudier l'algorithme suivant :

http://www.food.com/recipe/madhur-jaffreys-naan-bread-446809

Sinon, il suffit de savoir que la Hitroizeu introduit le `NaN` (comme dans *Not a Number*) dans les cas où le calcul demandé est « indéfini ».

```
In [29]: x = 1e500

In [30]: x - x
Out[30]: nan

In [31]: x / x
Out[31]: nan
```

```
In [32]: x - x == x - x
Out[32]: False

In [33]: x**2 - x
Out[33]: nan

In [34]: x * (x - 1)
Out[34]: inf
```

Un `NaN` n'est pas comparable. Trier un tableau comportant un `NaN` ne pose donc pas de problème :

```
In [36]: y = 1e500 / 1e500

In [37]: y
Out[37]: nan

In [38]: y > 3
Out[38]: False

In [39]: y < 3
Out[39]: False

In [40]: y == 3
Out[40]: False

In [41]: y == y
Out[41]: False
```

Il faut penser à un problème célèbre dont fut cette fois victime la marine étasunienne.

Aparté

L'affaire est beaucoup moins dramatique mais plus croustillante que celle des *Patriots*. La Marine US s'équipe de PC montés avec...Windows NT 4.0 et en profite pour réduire le 10% le nombre de marins à bord du USS Yorktown. Mais, comme le dira un des experts civils après la panne : « *Using Windows NT, which is known to have some failure modes, on a warship is similar to hoping that luck will be in our favor* »
Que s'est-il passé le 21 septembre 1997 ? Un opérateur a tapé par erreur un zéro sur son clavier et tout le navire a été bloqué en pleine mer :-)
« *Your $2.95 calculator, for example, gives you a zero when you try to divide a number by zero, and does not stop executing the next set of instructions. It seems that the computers on the Yorktown were not designed to tolerate such a simple failure.* »
On espère que les missiles atomiques sont mieux programmés...

Revenons sur cette anecdote et transplantons là dans un environnement plus pacifique.
Vous cherchez les zéros de, disons, comme ça, au hasard, ax^2+bx+c. Comme vous avez de vagues souvenirs de votre jeunesse, vous programmez : retourne-moi $-b \pm \frac{\sqrt{\Delta}}{2a}$ avec $\Delta = b^2 - 4ac$.
On suppose que toutes les valeurs sont proches de zéro et les mesures difficiles à ce niveau de précision.

```
In [43]: a, b, c = 1e-6, 1e-4, 1e-10

In [44]: d = b**2 - 4*a*c


In [45]: d
Out[45]: 9.9999996e-09


In [48]: sqrt(d)
Out[48]: 9.999999799999999e-05
```

```
In [49]: a, b, c = 1e-6, 1e-10, 1e-4

In [50]: d = b**2 - 4*a*c

In [51]: d
Out[51]: -3.9999999999e-10

In [52]: sqrt(d)
---------------------------------
ValueError ----> 1 sqrt(d)

ValueError: math domain error
```

En fait Python ne suit pas jusqu'au bout la norme car le dernier résultat devrait être NaN.
Ainsi, on n'a plus besoin de surcharger le programme de conditionnelles. Le calcul continue.
Reprenons nos formes indéterminées :

```
In [53]: x = 1e500

In [54]: x**2 - x
Out[54]: nan
```

En effet, $\infty - \infty$ est une forme indéterminée. Que faisiez-vous au lycée pour lever cette indétermination ?

```
In [55]: x * (x - 1)
Out[55]: inf
```

C'est réglé... Autre curiosité :

```
In [56]: x - x
Out[56]: nan

In [57]: x - x == x - x
Out[57]: False
```

On peut débusquer un `NaN` en effectuant le test `x /= x`.
Selon la Hitroizeu, un `NaN` est représenté avec l'exposant maximum et une mantisse non nulle : il y a donc toute une famille de `NaN` ! En voici un exemple en *toy7* :

s (1 bit)	e (3 bits)			f (3 bits)		
1	1	1	1	0	1	0

Aparté : de l'importance pour un programmeur de bien penser son message d'erreur

Encore un drame...Le 1er juin 2009, le vol Air France 447 finit tragiquement : les 228 personnes à bord meurent englouties au large des côtes du Brésil.
Beaucoup de bruit a été fait autour de cet accident, beaucoup de gens ont été blâmés, en particulier les copilotes.
Voici le problème : pour économiser du carburant, les avions volent maintenant très haut et en limite de décrochage. Il faut un guidage informatique très fin pour le permettre. Ce soir-là, l'avion est pris dans une tempête et le froid extrême obstrue les capteurs qui ne peuvent plus mesurer la vitesse du vent.
Le pilote automatique envoie alors comme message à ses collègues humains des messages trop vagues :
Les messages de panne successivement affichés sur l'ECAM n'ont pas permis à l'équipage de faire un diagnostic rapide et efficace de la situation dans laquelle l'avion se trouvait, en particulier de l'obstruction des sondes Pitot. Il n'a jamais été en mesure de faire le lien entre les messages qui sont apparus et la procédure à appliquer, alors que la lecture de l'ECAM et des messages doit faciliter l'analyse de la situation et permettre d'organiser le traitement des pannes. Plusieurs systèmes avaient pourtant identifié l'origine du problème mais n'ont généré que des messages (extrait du rapport final du Bureau d'Enquêtes et d'Analyses pour la sécurité de l'aviation civile).
Donner les conséquences et pas les causes ! Sans indications, les pilotes perdent de précieuses secondes et ne peuvent effectuer la manœuvre habituelle permettant de contrer un décrochage.

Successeur d'un VF

Il n'est pas possible de définir un successeur d'un réel mais un VF, lui, en possède un.
Écrivons un flottant avec sa mantisse entière en posant

$$n = \#f$$

Remarque Dans la littérature, on considère souvent $p = n + 1$.

Par exemple, pour *toy7*, $n = 3$, pour *binary32*, $n = 23$, pour *binary64*, $n = 52$.

$$v = M(v) \times 2^{E(v)-n}$$

$M(v)$ étant un entier, il admet un successeur. Si l'on reste dans les limites des VF normaux du système :

$$\text{succ}(v) = (M(v) + 1) \times 2^{E(v)-n} = v + 2^{E(v)-n}$$

On note

$$\text{ulp}(v) = 2^{E(v)-n}$$

À retenir **ULP**

Par exemple, en *binary64*, soit x un réel et $y = RN(x)$.
On a $y = (-1)^s \times 1, b_1 b_2 ... b_{52} \times 2^E$.
Alors $\text{ulp}(y) = 0,000...001 \times 2^E = 2^{E-52}$

Reconnaissance des flottants

La machine traite une chaîne de bits | s | e | f | ainsi :

$e = 0\ldots0$	$f = 0$	$\rightarrow$	$v = (-1)^s \times 0.0$
$e = 0\ldots0$	$f \neq 0$	$\rightarrow$	$v = (-1)^s \times 0.f \times 2^{1-E_{\max}}$
$e = 1\ldots1$	$f = 0$	$\rightarrow$	$v = (-1)^s \times \infty$
$e = 1\ldots1$	$f \neq 0$	$\rightarrow$	NaN
$0\ldots0 < e < 1\ldots1$	$f \neq 0$	$\rightarrow$	$v = (-1)^s \times 1.f \times 2^{e-E_{\max}}$

Tableau récapitulatif

On notera $\varepsilon_m = 2^{-n}$ l'*epsilon* de la machine, c'est-à-dire le successeur de 1.
On notera λ le plus petit VF normal positif.
On notera μ le plus petit VF sous-normal positif.
On notera Ω le plus grand VF normal.

Recherche

Quellle relation existe-t-il entre μ, ε_m et λ ?

Format	$\#E$	$\#f$	$E_{\min}$	$E_{\max}$	ε_m	λ	μ	Ω
toy7	3	3	-2	$+3$	$2^{-3} = 1/8$	$2^{-2} = 1/4$	$2^{-5} = 1/32$	$1,111 \times 2^3 = 15$
binary32	8	23	-126	$+127$	$2^{-23} \approx 1,2 \times 10^{-7}$			
binary64	11	52	-1022					

3 Algèbre des nombres VF

L'ensemble des VF

Nous allons à présent travailler dans notre ensemble de flottants normaux et sous-normaux. Nous noterons par exemple $\mathbb{V}_{32}$ les normaux et sous-normaux de *binary32* et nous noterons $\overline{\mathbb{V}_{32}}$ ce même ensemble enrichi des deux infinis.
Dans le cas général, nous noterons tout simplement $\mathbb{V}_b$ voire $\mathbb{V}$.

À retenir

Ainsi $\overline{\mathbb{V}_b}$ est un ensemble FINI.

On va munir cet ensemble d'opérations usuelles et étudier les structures algébriques produites.

Comparaison

Recherche

Il est très simple et rapide de comparer deux VF : comment la machine procède-t-elle ? Quel est l'avantage de ce stockage des VF ?

Addition

On décompose l'action en trois étapes :

1. on commence par ramener les deux nombres au même exposant, en l'occurrence le plus grand des deux ;
2. on ajoute les deux mantisses *complètes* en tenant compte du signe ;
3. on renormalise le nombre obtenu.

Par exemple, en *toy7*, additionnons 1,1 et 0,0111 ou plutôt $1,1 \times 2^0$ et $1,11 \times 2^{-2}$ ou plutôt (en *toy7*, le décalage d'exposant est de 3) :

1,1 : | 0 | 0 | 1 | 1 | 1 | 0 | 0 |

0,0011 : | 0 | 0 | 0 | 1 | 1 | 1 | 0 |

On réécrit 0,00101 avec le même exposant que 1,1 mais attention à la précision de la mantisse !
1,100
0,001*11*
_ _ _ __

1,101*11*
On ne peut pas garder les deux derniers 1 du plus petit nombre car on n'a pas la place de les stocker !
Que faire ? Il faut arrondir en choisissant selon quatre méthodes usuelles :

Définition 1 - 1

Les arrondis

1. l'arrondi au plus proche (RN) qui arrondit au VF...le plus proche. En cas d'égalité, on choisit la valeur paire (donc qui se termine par un 0 en binaire) ;
2. l'arrondi vers 0 (RZ) qui arrondit à la valeur de plus petite valeur absolue : c'est la troncature ;
3. l'arrondi vers $+\infty$ (RU) qui arrondit à la valeur supérieure la plus petite ;
4. l'arrondi vers $-\infty$ (RD) qui arrondit à la valeur inférieure la plus grande.

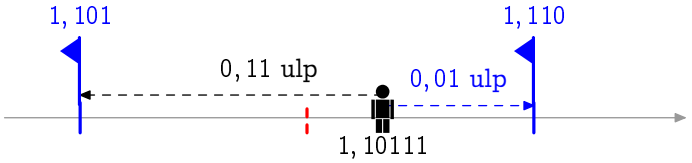
Le mode d'arrondi par défaut est le premier.

Posons $x = 1,10111$ (qui n'est pas un VF !) : il est compris entre les deux VF $1,101$ et $1,110$.

$$1,101\mathit{00} \leqslant \underbrace{1,101\mathit{11}}_{x} \leqslant 1,110\mathit{00}$$

$$\underbrace{1,100 + \mathit{1,00}\,ulp(x)}_{v} \leqslant \underbrace{1,100 + \mathit{1,11}\,ulp(x)}_{x} \leqslant \underbrace{1,100 + \mathit{10,00}\,ulp(x)}_{\mathrm{succ}(v)}$$

$|x - v| = 0,11ulp(x)$, $|x - \mathrm{succ}(v)| = 0,01ulp(x)$ donc $RN(x) = \mathrm{succ}(v)$



$$RN(1,1 + 0,001) = 1,110$$

Il y a énormément de choses à dire sur ces problèmes d'arrondis. Nous leur consacrerons une section.
Il y a beaucoup de choses à dire sur les sommes itérées. Nous leur consacrerons une section.

Recherche

Est-ce que l'addition des VF est associative ?
Est-on sûr de l'ordre dans le quel un compilateur calcule $a + b + c + d$?

Multiplication

On suit un ordre logique :

1. on « xore » les bits de signe ;
2. on additionne les exposants réels et on décale ou plutôt on additionne les exposants décalés et on retire la valeur d'un décalage ;
3. on multiplie les mantisses ;
4. on normalise.

Effectuons par exemple le produit de 101,1 par -10,01, i.e. :

101,1 :

0	1	0	1	0	1	1

-10,01 :

1	1	0	0	0	0	1

1. 0 xor 1 donne 1 : le bit de signe est 1 ;
2. 101 + 100 − 11 = 1001 − 11 = 110 : l'exposant décalé est 110 donc l'exposant est 3 ;
3. $1,011 \times 1,001 = 1,011 + 1,011 \times 0,001 = 1,011 + 0,001011 = 1,100011$;

4. le produit est donc $1,100011 \times 2^3$. Le passage à la forme normale va arrondir le produit. On a

$$1,100 < x < 1,101$$

avec $|x - 1,100| = 0,011ulp(x)$ et $|x - 1,101| = 0,101ulp(x)$ donc

$$RN(101,1 \times (-10,01)) = -1,100 \times 2^3$$

i.e.

1	1	1	0	1	0	0	

Ici encore, le VF ne correspond pas au réel...
Il est temps de prendre le problème à bras le corps, mais avant :

Recherche

Est-ce que la multiplication des VF est associative ?
Est-ce que la multiplication des VF est distributive sur l'addition ?

4 Réels, arrondis et flottants

Dans la suite de nos aventures, nous ne considérerons que le mode d'arrondi par défaut sauf mention contraire.

Un problème, dix problèmes, mille problèmes

Nous commençons à bien connaître le problème :

```
In [4]: 3 * 0.1 == 0.3
Out[4]: False
```

On pourrait alors se dire qu'il ne faut jamais comparer deux VF x et y avec == mais plutôt les considérer égaux s'ils vérifient $|x - y| < d$ avec d un seuil d'erreur. Mais résoudre ce problème ainsi va en créer de nombreux autres :

— comment fixer d ?

— si x et y ont des signes différents ?

— $x \equiv y \leftrightarrow |x - y| < d$ définit-elle une relation d'équivalence ?

— ...

Recherche

Répondez à la troisième question...

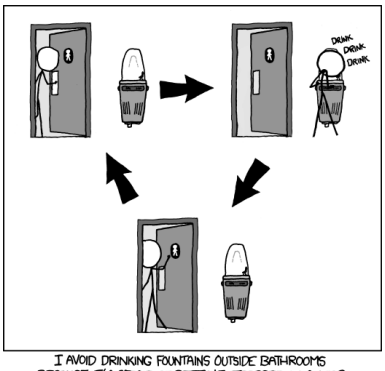
Il faut penser différemment !

Danger

Beaucoup de propriétés des réels ne sont plus vraies dans $\mathbb{V}$. Par exemple, le programme suivant, que nous allons étudier, serait très dangereux si nous travaillons dans $\mathbb{R}$ mais est très intéressant dans $\mathbb{V}$:

```
def f1(a) :
    return a if (a + 1.) - a != 1. else f1(2. * a)

def f2(a, b) :
    return b if (a + b) - a == b else f2(a, b + 1.)
```



Mais comme souvent en mathématique et en informatique, **un petit dessin aide à visualiser une situation complexe**.

Danger

Attention : tout VF est en fait un rationnel mais tout rationnel n'est pas un VF ! Par exemple...

La seule certitude est qu'un VF se représente correctement...lui-même.
Mais je vous rappelle que ce que vous tapez sur votre clavier n'est pas forcément un VF si vous tapez en base 10.
En particulier, si x est un VF, $x = RN(x)$.
La norme assure de plus que pour toute opération arithmétique standard,

$$RN(x) \top RN(y) == RN(x \top y)$$

Il faut aussi penser que rien n'est gratuit :

À retenir

S'il y a un *epsilon* dans votre calcul, c'est à vous de vous en occuper et de le contrôler : la norme vous donne les moyens de contrôler mais ce n'est pas la machine qui fait le boulot...c'est VOUS !
N'oubliez pas que vous avec un avantage sur la machine : vous connaissez la valeur réelle du réel que vous manipulez ou faites manipuler.

Voici une petite réflexion encore plus subtile...

Danger

$0,001 \times 2^0$ ce n'est pas tout à fait pareil que sa forme normalisée $1,000 \times 2^{-3}$.
Les 3 zéros rajoutés à droite n'ont sûrement aucune valeur !
Par exemple si on calcule $11,111 - 11,110$, mais sur 6 bits, on aurait peut-être un autre résultat qui dépend de bits ici inconnus...
Il y a donc plusieurs échelle de certitude !

Quand des nombres sont très proches, les soustraire peut engendrer ce genre de phénomène qu'on appelle *annulation catastrophique* : l'*erreur relative* est bien plus grande que l'*erreur absolue* (nous reparlerons de ces notions un peu plus loin).)
On pourra s'intéresser à l' Recherche 1 - 26 page 32 pour méditer sur le danger relatif de certaines annulations (cancellations in angliche) : toute annulation n'est pas catastrophique !

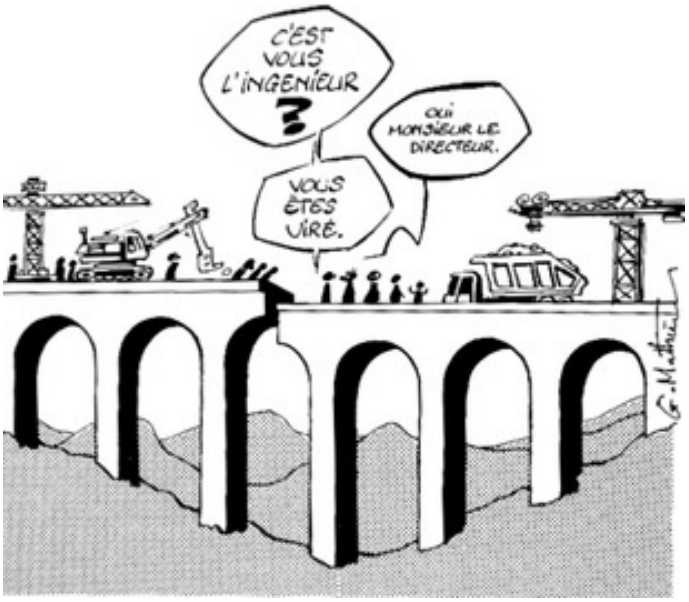
Danger

Les machines calculent vite, c'est un bien mais aussi un mal : si on effectue une somme avec une toute petite erreur, une grande somme peut nous faire perdre tous nos bits en une seconde car la somme de beaucoup de très petites erreurs peut créer une grosse erreur...

Mais le plus grand danger vient de :

95 % of the folks out there are completely clueless about floating-point

James Gosling (M. Java) - 28 février 1998



Maîtriser les erreurs

Tout d'abord, une déception : il n'existe pas de traitement automatique des erreurs sur les VF. Cependant un espoir : il existe des moyens de traiter les problèmes selon le contexte...mais qui nécessitent un bagage minimum en mathématique ;-)
D'abord un peu de vocabulaire. Il faudra bien distinguer :

Précision (*precision*) : c'est le nombre de bits utilisés pour représenter un nombre. La précision concerne donc le format utilisé pour écrire ou stocker ou arrondir un nombre. Par exemple `3`, `3.0`, `3.0000`, `3.0e0` n'ont pas la même précision, précision qui dépend en plus du langage.

Exactitude (*accuracy*) : c'est ce qui relie un nombre au contexte dans lequel il est employé. Connaître la distance qui sépare votre domicile de votre lieu de travail au mètre près peut être considéré comme très proche du résultat exact mais connaître votre taille au mètre près l'est moins...L'exactitude est liée à la mesure de l'erreur.

Nous définirons deux types de mesure d'erreur :

Définition 1 - 2

Erreurs
Soit x un nombre et $\widehat{x}$ le nombre qui le représente. On distingue :

l'erreur absolue $|x-\widehat{x}|$

l'erreur relative $\eta = \frac{x-\widehat{x}}{x}$ alors $\widehat{x} = x(1+\eta)$

commise en prenant $\widehat{x}$ à la place de x.

On prend souvent comme approximation de l'erreur relative le rapport $|\log(x) - \log(\widehat{x})|$. Nous justifierons cette approximation lors de l'étude du calcul différentiel.
Pour reprendre un exemple de W. Kahan et Darcy [1998], $3,177777777777777$ est une approximation plutôt précise (16 décimales) mais inexacte (2 décimales) de π.
Comme nous allons le voir dans de nombreux exemples (e.g. Recherche 1 - 26 page 32), analyser les erreurs avec un peu de mathématique permet de contrôler son programme.
On peut mesurer les erreurs à deux niveaux. Lorsqu'on étudie un algorithme qui donne un résultat r en fonction d'une donnée d, on peut considérer que r est une fonction de d :$r = f(d)$.
On peut analyser l'erreur commise sur r ou l'erreur commise sur d.

Définition 1 - 3

L'erreur *aval* (*forward error*) est la différence entre le résultat mesuré $\widehat{r}$ et le résultat théorique r.
L'erreur *amont*, ou erreur *inverse* (*backward error*) est le plus petit $|\Delta d|$ tel que $f(d+\Delta d) = \widehat{r}$.

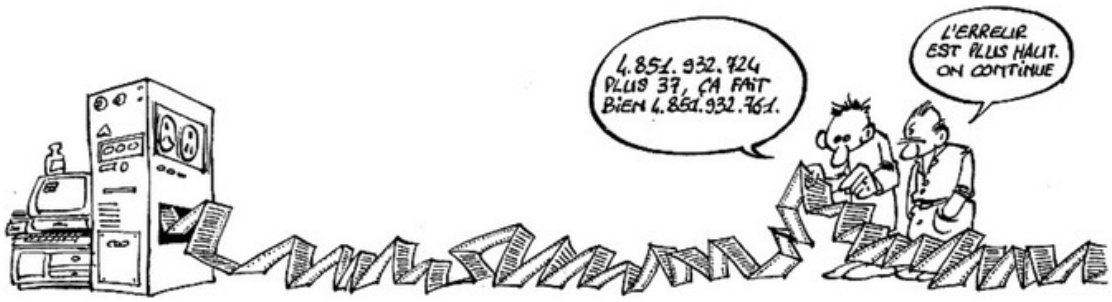
Feel nervous, but feel in control. It's not dark magic, it's science.

Florent DE DINECHIN

C'est ce peu de mathématique que nous allons découvrir dans ce cours...
Mais un point non négligeable que nous n'aborderons pas (nous l'évoquerons) est comment le processeur, l'OS, le langage (mal)traitent la norme IEEE 754...

À retenir

La norme ne vous transforme pas en Harry POTTER : il ne suffit pas de se dire qu'il y a une norme et que tout va bien se passer en appuyant sur son clavier-baguette magique !
La norme vous donne les moyens de maîtriser les forces du mal **mais ça peut demander un sacré effort de réflexion (c'est-à-dire de mathématique :-))** .



5 Que la force de l'erreur soit avec vous

Atelier Padawan # 1 : majorer l'erreur

Précision machine

Soit x un nombre. On peut l'encadrer par deux VF successifs :

$$M \times 2^{E-n} \leqslant x < (M+1) \times 2^{E-n} = M \times 2^{E-n} + 2^{E-n}$$

avec M un entier, n la longueur de la pseudo-mantisse.
L'approximation $\widehat{x}$ est donc la plus proche de ces deux bornes de cet intervalle de longueur 2^{E-n}.
Avec l'arrondi par défaut :

$$|x - \widehat{x}| \leqslant \frac{1}{2} 2^{E-n}$$

donc l'**erreur relative** peut être majorée

$$\left|\frac{x-\widehat{x}}{x}\right| \leqslant \left|\frac{x-\widehat{x}}{m \times 2^E}\right| \leqslant \frac{1}{2}\frac{2^{E-n}}{m \times 2^E} = \frac{1}{2}\frac{2^{-n}}{m} = \frac{1}{2}\frac{\varepsilon_M}{m}$$

avec m la mantisse normalisée.
Si $\widehat{x}$ n'est pas un VF sous-normal, alors $1 \leqslant m < 2$ donc

$$\left|\frac{x-\widehat{x}}{x}\right| \leqslant \frac{1}{2}\varepsilon_M$$

L'erreur relative commise est donc majorée par la moitié du successeur de 1, i.e. l'*epsilon* de la machine.
Si $\widehat{x}$ est sous-normal, alors l'**erreur absolue** est majorée :

$$|x - \widehat{x}| \leqslant \frac{1}{2} 2^{E_{\min}-1-n}$$

Ainsi, dans tous les cas :

À retenir

Précision
$\widehat{x} = x(1+\eta_1) + \eta_2$ avec :

- si $\widehat{x}$ est normal $|\eta_1| \leqslant \frac{1}{2}\varepsilon_M$ est l'erreur relative et $\eta_2 = 0$
- si $\widehat{x}$ est sous-normal $\eta_1 = 0$ et $|\eta_2| \leqslant \frac{1}{2} 2^{E_{\min}-1-n}$

On appelle $u = \frac{1}{2}\varepsilon_M$ l'unité d'arrondi (*unit roundoff*) ou la précision machine.

Par exemple, la précision machine en *binary64* vaut $2^{-52}/2 \approx 1.1 \times 10^{-16}$.

Danger

Précision ≠ exactitude !
$R((a\cdot b)\cdot c) = R(a\cdot b)\cdot R(c) = a\cdot b(1+\eta_{ab})\cdot c(1+\eta_c) = a\cdot b\cdot c(1+\eta_{ab})(1+\eta_c) \approx a\cdot b\cdot c(1+\eta_{ab}+\eta_c)$
La précision est toujours u mais l'exactitude est d'environ $2u$...

L'exactitude n'est pas assurée par la précison !

Remarque

On préfère étudier l'erreur relative car elle ne dépend pas du facteur d'échelle : si x et $\widehat{x}$ sont multipliés par un facteur k, l'erreur absolue aussi mais l'erreur relative reste inchangée.

Oui, bon, OK, mais cette majoration est parfois très pessimiste et ne permet pas d'aller très loin.

Éliminiation

L'élimination (*cancellation* in english of speciality) est le phénomène qui apparaît lorsqu'on soustrait deux nombres très proches.

Avec les notations habituelles, $\widehat{a} = a(1+\eta_a)$, $\widehat{b} = b(1+\eta_b)$, $x = a-b$ et $\widehat{x} == \widehat{\widehat{a}-\widehat{b}}$. Alors

$$\left|\frac{x-\widehat{x}}{x}\right| = \left|\frac{-a\eta_a + b\eta_b}{a-b}\right| \leqslant \max(|\eta_a|,|\eta_b|)\frac{|a|+|b|}{|a-b|}$$

On n'obtient ici qu'une majoration, très pessimiste d'ailleurs.

Cependant on constate que l'erreur est d'autant plus grande que $|a-b| \ll |a|+|b|$ et de plus cette erreur augmente les erreurs déjà présentes dans $\widehat{a}$ et $\widehat{b}$.

Ainsi, l'erreur causée par l'élimination remettent au premier plan de petites erreurs faites précédemment.

Parfois cette erreur n'est pas forcément dramatique. D'abord parce que les données initiales sont peut-être exactes et donc aucune erreur n'est amplifiée. Ensuite, l'erreur causée peut être « écrasée » par une valeur plus grande. Nous étudierons ce phénomène par exemple dans l' Recherche 1 - 26 page 32.

Atelier Padawan # 2 : quitter la réalité (un peu difficile...)

Dans l'atelier précédent, nous avons travaillé avec des outils « réels » : majorations, minorations avec des opérations définies sur $\mathbb{R}$ et nous avons oublié qu'en fait nous travaillons dans $\mathbb{V}$: il est temps d'enlever nos scaphandres pour être plus à l'aise...

Exceptionnellement dans cette section nous travaillerons en base β quelconque.

Trouver la base (algorithme de Malcolm-Gentleman)

Observez ce qui se passe et expliquez pourquoi cela se passe sachant que sur JavaScript, il n'y a qu'un seul type numérique : *binary64*...

```
Rhino 1.7 release 3

js> Math.pow(2, 53)
9007199254740992
js> Math.pow(2, 53) + 1
9007199254740992
js> Math.pow(2, 53) + 2
9007199254740994
```

```
js> Math.pow(2, 53) + 3
9007199254740996
js> Math.pow(2, 53) + 4
9007199254740996
js> Math.pow(2, 53) + 5
9007199254740996
js> Math.pow(2, 54)
18014398509481984
```

```
js> Math.pow(2, 54) + 2
18014398509481984
js> Math.pow(2, 54) + 3
18014398509481988
js> Math.pow(2, 55)
36028797018963970
js> Math.pow(2, 55) + 4
36028797018963970
```

Et maintenant, voici un algorithme en impératif qui, si on travaillait dans $\mathbb{R}$ ne terminerait pas mais que vous allez pouvoir comprendre après notre petite observation sur JavaScript : :

```
1 Algorithme de Malcolm-Gentleman
2 a ← 1.0, b ← 1.0
3 TantQue R(R(a + 1.0) − a) == 1.0 Faire
4 | a ← R(2 × a)
5 FinTantQue
6 TantQue R(R(a + b) − a) ≠ b Faire
7 | b ← b + 1
8 FinTantQue
9 Retourner b
```

Nous en avons déjà parlé 1.4.1 page 16

```
1 def f1(a) :
2     return a if (a + 1.) - a != 1. else f1(2. * a)
3
4 def f2(a, b) :
5     return b if (a + b) - a == b else f2(a, b + 1.)
6
7 f2(f1(1.), 1.)
```

Regardons la première boucle. On y construit une suite (a_i) vérifiant pour tout naturel i $a_{i+1} = 2a_i$ avec $a_0 = 1.0$. Une récurrence immédiate démontre que $a_i = 2^i$ pour tout entier naturel i.
...Mouais, c'est exact SI l'on travaille dans $\mathbb{R}$...Mais nous travaillons dans $\mathbb{V}_\beta$!

Dans quel espace tu travailles tu me diras et si ta proposition est vraie je saurai

Mathemator - 44 ap. GC

En fait notre espace de travail est $\mathbb{V}_\beta$ avec β notre base (qui est donc un entier supérieur ou égal à 2) dont la mantisse (pas la pseudo-mantisse...) est de longueur p.
Les lignes 4 à 6 construisent une suite $a_i = R(2a_{i-1})$ avec $a_0 = 1.0$ mais est-on sûr que l'on a toujours $a_i = 2^i$ dans ces conditions ?
On démontre d'abord par récurrence que $a_i = R(2^i) = 2^i$ tant que i vérifie $2^i \leqslant \beta^p - 1 : 2^i = 2 \times 2^{i-1}$ est un entier qui s'écrit avec moins de p chiffres de la base β et en deçà de $\beta^p - 1$, on peut passer d'un flottant à son successeur par des pas inférieurs à 1.
Alors $R(a_i + 1.0) = a_i + 1.0$: pas d'arrondi.
Puis $R(R(a_i + 1.0) - a_i) = R(a_i + 1.0 - a_i) = 1.0$: la condition de la première boucle est donc vérifiée tant que $2^i < \beta^p$. Que se passe-t-il après ?
Dès que $\beta^p - 1$ est atteint, les choses changent.
Considérons la première itération i_s telle que $2^{i_s} \geqslant \beta^p$.
On a

$$a_{i_s} = R(2 \times a_{i_s-1}) = R(2 \times 2^{i_s-1}) < R(2 \times \beta^p) \leqslant R(\beta \times \beta^p) = \beta^{p+1}$$

car d'une part $a_{i_s-1} = 2^{i_s-1}$, d'autre part $a_{i_s-1} \leqslant \beta^p - 1 < \beta^p$ et enfin $2 \leqslant \beta$.
Ainsi

$$\beta^p \leqslant a_{i_s} < \beta^{p+1}$$

L'exposant de la forme normale de a_{i_s} est donc p et d'après le résultat suivant démontré à la section 1.2.4 page 13, le successeur d'un VF v vérifie (en notant $n = p - 1$ la longueur de la pseudo-mantisse) :

$$\text{succ}(v) = v + \beta^{E(v)-n} = v + \beta^{E(v)-p+1}$$

Ici $\text{succ}(a_{i_s}) = a_{i_s} + \beta^{E(a_{i_s}-p+1)} = a_{i_s} + \beta^{p-p+1} = a_{i_s} + \beta$.
On en déduit que $a_{i_s} + 1.0$ est entre a_{i_s} et son successeur $a_{i_s} + \beta$.
Donc, selon l'arrondi, $R(a_{i_s} + 1.0)$ vaut a_{i_s} ou $a_{i_s} + \beta$ et finalement $R(R(a_{i_s} + 1.0) - a_{i_s})$ vaut 0 ou β mais en aucun cas 1.0 : on sort donc de la boucle.

On en déduit que `boucle1 1` *vaut* a_{i_s} *donc que* $\beta^p \leqslant$ `boucle1 1` $< \beta^{p+1}$

Notons a = `boucle1` 1. Son successeur est $a + \beta$.
Passons à la seconde boucle.

Tant que $b < \beta$...mais je vous laisse conclure la démonstration :-)

Moralité
Cette démonstration est plutôt technique et difficile en première lecture. Cependant elle est pleine d'enseignements :

À retenir

- elle nous montre bien que nous changeons de monde : un test du style `if (a + 1.0)- a /= 1.0` paraîtrait bien hors de propos si nous raisonnions avec des réels ;
- nous avons vu que l'on pouvait raisonner rigoureusement sur les VF ;
- nous pouvons malgré tout retenir la trame intuitive de la démonstration : tant que notre nombre entier est représentable avec p chiffres, on reste exact. Les problèmes arrivent lorsqu'on n'a plus assez de place pour stocker tous les chiffres : il y a de la perte d'information...
- Ce genre de raisonnement se retrouve très souvent pour travailler sur les erreurs commises : il faudra donc en retenir la substantifique moelle...

N'oubliez pas que certains langages ne travaillent pas en base 2, par exemple Maple© :

```
> a := 1.0:
> b := 1.0:
> while evalf(evalf(a + 1.0) - a) - 1.0 = 0.0 do
      a := 2.0 * a;
  end_while:
> while evalf(evalf(a + b) - a) - b <> 0.0 do
      b := b + 1.0
  end_while:
> b;
                                  10.0
```

Nouvelles opérations

Pour alléger les notations et bien comprendre que nous changeons de monde mais que la structure des calculs sur les flottants reste algébrique (mais des propriétés intéressantes comme l'associativité disparaissent), nous noterons :

Notations

Les opérations arithmétiques de base sont bien définies sur $\mathbb{V}$ donc nous noterons à présent, si l'arrondi n'a pas besoin d'être précisé :

- $R(x + y)$ sous la forme $x \oplus y$;
- $R(x - y)$ sous la forme $x \ominus y$;
- $R(x \times y)$ sous la forme $x \otimes y$;
- $R(x/y)$ sous la forme $x \oslash y$.

Ainsi on notera par exemple $a \oplus b = a + b + \mathrm{err}(a \oplus b)$.

Trouver la précision

Encore un algorithme de M.A. Malcolm proposé en 1972.
Cette fois, vous vous occuperez de la démonstration tout(e) seul(e)...

```
def precision() :
    def toPrec(a, i) :
        return i - 1 if (a + 1.) - a != 1. else toPrec(2 * a, i + 1)
    return toPrec(1., 0.)
```

Atelier Padawan # 3 : somme de flottants proches

Quelques lemmes utiles

Un lemme en mathématique est un résultat intermédiaire qui sert à prouver un résultat plus important. Un lemme peut être cependant très compliqué à démontrer : la démonstration du *lemme fondamental* a valu à Bao Châu Ngô d'obtenir en 2010 la médaille Fields...
Fini de jouer...Les démonstrations des algorithmes précédents sont intéressantes mais les résultats obtenus sont anecdotiques : on sait bien dans quelle base on travaille et avec quelle précision.
Passons à présent à des résultats beaucoup plus utiles maintenant que nous sommes au parfum des démonstrations sur les flottants.
Un des grands dangers de la manipulation des flottants est d'effectuer des soustractions car on peut perdre beaucoup d'information par annulation (*cancellation*).
Pour avoir un meilleur contrôle de la chose, nous allons étudier un lemme énoncé par Pat H. Sterbenz [1973] (1927 - 2008)...mais pour le démontrer, nous aurons besoin de quelques sous-lemmes ;-)

Danger

Convention
Pour éviter de déduire des lemmes suivants des théorèmes totalement faux, n'oubliez pas que **DANS CE QUI SUIT X ET Y SONT DES NOMBRES À VIRGULE FLOTTANTE !**

Lemme 1 - 1

Majoration de l'erreur d'une somme
Posons $x \oplus y = x + y + \text{err}(x \oplus y)$. Alors, s'il n'y a pas de dépassement de capacité,

$$|\text{err}(x \oplus y)| \leqslant \min(|x|, |y|)$$

On a bien sûr un résultat analogue pour la différence

On a $x \oplus y = x + y + \text{err}(x \oplus y)$ (on est dans $\mathbb{R}$...)
De plus $x = x + y + (-y)$ (on est toujours dans $\mathbb{R}$...)
x et $x \oplus y$ sont donc deux VF distants respectivement de $|y|$ et de $|\text{err}(x \oplus y)|$ de $x + y$ (que de de, de deux et de 2 : pffff....)
Mais la norme nous assure que c'est $x \oplus y$ le VF le plus proche de $x + y$ donc l'erreur commise en le choisissant est moindre que celle faite en choisissant $-y$.
On en déduit que $|\text{err}(x \oplus y)| \leqslant |y|$.
On montre de même que $|\text{err}(x \oplus y)| \leqslant |x|$.

Remarque

De l'importance de disposer avec la IEEE 754 de la **meilleure approximation !**

Corollaire 1 - 1

Møller, Knuth, Dekker
L'erreur commise $|\text{err}(x \oplus y)|$ peut être exprimée exactement sur p bits.

Ce n'est qu'un corollaire d'un sous-lemme, mais c'est un résultat très important !
Supposons, sans perdre de généralité, que $|x| \geqslant |y|$.
Comme x et y sont des VF, le plus petit bit significatif de $\text{err}(x \oplus y)$ est au moins de magnitude celle de $\text{ulp}(y)$.
De plus $|\text{err}(x \oplus y)| \leqslant |y|$, donc la mantisse entière de $\text{err}(x \oplus y)$ a une longueur inférieure à p bits.
Finalement le mantisse entière de $|\text{err}(x \oplus y)| \leqslant |y|$ est exactement exprimable sur p bits.

Lemme 1 - 2

Supposons que $|x + y| \leqslant \min(|x|, |y|)$, alors $x \oplus y = x + y$.
On obtient un résultat analogue pour la soustraction.

La démonstration va ressembler à la précédente.
Supposons, sans perdre de généralité, que $|x| \geqslant |y|$.
Le plus petit bit significatif de $x + y$ est au moins de magnitude celle de $\text{ulp}(y)$.
Or $|x + y| \leqslant |y|$ donc la mantisse entière de $x + y$ a une longueur inférieure à p bits.

Finalement le mantisse entière de $x+y$ est exactement exprimable sur p bits.

Danger

N'oubliez pas la convention !
Nous avons démontré nos lemmes en considérant des VF x et y.
Est-ce que le résultat suivant contredit notre dernier lemme ?

```
In [11]: 1 - 0.9
Out[11]: 0.09999999999999998
```

Lemme de Sterbenz

Lemme 1 - 3

Lemme de Sterbenz (1973)
Soit $(x,y) \in \mathbb{V}^2$ vérifiant $\frac{x}{2} \leqslant y \leqslant 2x$. On a

$$x \ominus y = x - y$$

La différence de deux VF suffisamment proches est donc exacte.

Une petite démonstration ?... Allez, le lemme précédent va sûrement être utile pour avoir un petit schéma de démonstration qui tient sur une ligne :-)
Pour une démonstration plus complète, on pourra par exemple se référer à Muller et coll. [2010] page 123.
On peut se restreindre à des nombres positifs.
On a deux cas :

1. $\boldsymbol{x < y}$: alors $x < y \leqslant 2x$ donc $0 < y - x \leqslant x \leqslant y$;
2. $\boldsymbol{x \geqslant y}$: alors $\frac{x}{2} \leqslant y \leqslant x$ donc $-\frac{x}{2} \leqslant y - x \leqslant 0$ et par suite $0 \leqslant x - y \leqslant \frac{x}{2} \leqslant y \leqslant x$.

Dans tous les cas $|x-y| \leqslant \min(|x|,|y|)$ et on peut utiliser le lemme 1 - 2 page précédente.

Remarques

Concrètement, à quoi correspond cette condition $\frac{x}{2} \leqslant y \leqslant 2x$?
Demandez-vous ce que l'on peut dire de l'écart maximum entre les exposants de x et y.
On admettra que **le résultat reste vrai même en cas de sous-capacité**.

Atelier Padawan # 4 : somme compensée de flottants quelconques

Les lemmes précédents ne permettent de s'assurer de l'exactitude de la somme que dans des cas restreints.
Que se passe-t-il quand on ne peut pas s'assurer à chaque instant que les opérandes sont suffisamment proches ?

Fast2Sum

Voici une première version officiellement due à T.J. Dekker en 1971 mais qui avait déjà été introduite par l'incontournable W. Kahan en 1965.

Théorème 1 - 1

Fast2Sum - Dekker & Kahan
On considère deux VF x et y tels que $|x| \geqslant |y|$ et l'algorithme suivant :

```
1  s   ← x ⊕ y
2  y_v ← s ⊖ x
3  d   ← y ⊖ y_v
4  Retourner (s, d)
```

Alors $x + y = s + d$ avec $s = x \oplus y$ et $d = \mathrm{err}(x \oplus y)$. De plus s et d ne se chevauchent pas.

ce qui donne dans notre langage favori :

```
1 def fast_two_sum(x, y) :
2     if abs(x) >= abs(y) :
3         s  = x + y
4         yv = s - x
5         d  = y - yv
6         return (s, d)
7     return fast_two_sum(y, x)
```

La ligne 1 donne $s = x + y + \mathrm{err}(x \oplus y)$.
De plus, comme $|x| \geqslant |y|$, c'est x « qui gagne » donc s et x ont le même signe.
La ligne 2 calcule un y virtuel (y_v) qui vaut $s \ominus x$: on aimerait bien que ça fasse $s - x$ c'est-à-dire $y + \mathrm{err}(x \oplus y)$.
Enfin la ligne 3 calcule $y \ominus y_v$: on aimerait bien aussi que ça fasse $y - y_v$ c'est-à-dire $-\mathrm{err}(x \oplus y)$ comme ça on récupérerait exactement l'erreur commise.
En plus, comme $d = -\mathrm{err}(x \oplus y) =$, on a $|d| \leqslant \frac{1}{2}\mathrm{ulp}(s)$ donc s et d ne se chevauchent pas dans leur somme.

On aimerait bien, on aimerait bien...En fait tout s'arrange car nous allons pouvoir utiliser le lemme de Sterbenz.
Il suffit de distinguer deux cas :

- **si x et y sont de même signe OU si $|y| \leqslant \frac{|x|}{2}$** : alors $\frac{x}{2} \leqslant s \leqslant 2x$ et on peut appliquer le lemme ;
- **sinon** : on a x et y de signes opposés ET $y > \frac{|x|}{2}$ alors si y est négatif $x/2 < -y < x$ et sinon $-x/2 < y < -x$. Dans les deux cas, d'après le lemme de Sterbenz, s est calculée exactement et alors $y_v = y$.

On illustrera cet algorithme par un exemple simple en TD dans l' Recherche 1 - 23 page 31
On illustrera encore mieux le problème par cette... illustration :

x	x_1	x_2		
$+y$		y_1	y_2	
$= s$	x_1	$x_2 + y_1$	y_2 est perdu	
$-x$		y_1	0	
$-y$			$-y_2$	0 on récupère l'erreur

Ainsi, $x + y = s + d$ se lit :

x	x_1	x_2		
$+y$		y_1	y_2	
$= s$	x_1	$x_2 + y_2$	y_2 est perdu	
$+d$			y_2	0 on récupère l'erreur

Danger s et d ne se chevauchent pas : $s \oplus d = s$!

2Sum

Il s'avère que sur les machines modernes, la comparaison préliminaire du Fast2Sum peut s'avérer plus coûteuse que les trois additions supplémentaires qui vont suivre.
De plus, le Fast2Sum ne fonctionne qu'en base 2 ou 3 alors que l'algorithme suivant fonctionne dans toutes les bases.

Mais il faut encore rentrer dans plus de détails comme l'éventualité d'un parallélisme (lignes 2 et 3), etc...vous verrez ça en Master...

Théorème 1 - 2

2Sum - Knuth

On considère deux VF x et y et l'algorithme suivant :

1 $s \leftarrow x \oplus y$
2 $y_v \leftarrow s \ominus x$, $x_v \leftarrow s \ominus y$
3 $e_y \leftarrow y \ominus y_v$, $e_x \leftarrow x \ominus x_v$
4 $d \leftarrow e_x \oplus e_y$
5 **Retourner** (s, d)

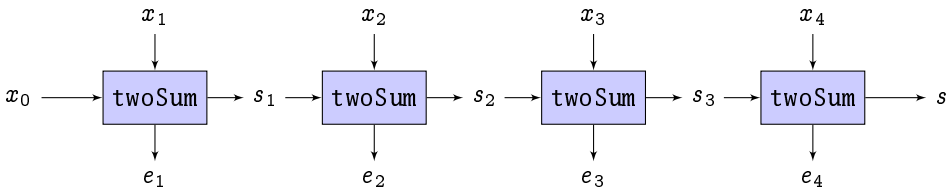
Alors $x + y = s + d$ avec $s = x \oplus y$ et $d = \mathrm{err}(x \oplus y)$. De plus s et d ne se chevauchent pas.

Somme compensée d'un nombre quelconque de flottants

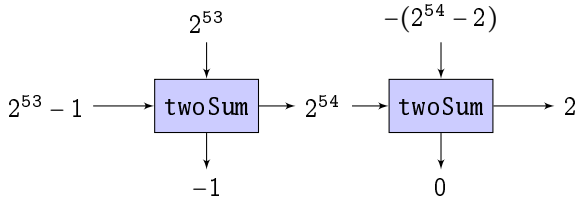
Voici qu'entre à nouveau dans l'arène W. KAHAN avec son algorithme *compensated summation* de 1965 (un peu en avance sur son temps...).
En Python :

```
1 def sommeKahan(xs) :
2     s, d = 0., 0.
3     for x in xs :
4         (s, d) = fast_two_sum(s, x + d)
5     return s
```

Sauriez-vous le traduire en fonctionnel avec un pliage (`reduce`) ? Il vaudrait mieux pour répondre aux questions de l' Recherche 1 - 25 page 32.
Est-ce optimum ? En 1972, Michèle PICHAT[Pichat, 1972] mais aussi Arnold NEUMAIER (mais l'article est en allemand) ont l'idée de sommer à part les erreurs et de soustraire le tout à la fin. Cet algorithme (dit de la *sommation en cascade*) est repris en 2008 par Siegfried M. RUMP, Takeshi OGITA et Shin'Ichi OISHI[Rump et coll., 2008] mais en introduisant le fastSum...faites de même !
Le schéma suivant vous inspirera sûrement :



Voyons tout de suite un exemple. On veut effectuer la somme en *binary64* de $x_0 = 2^{53} - 1$, $x_1 = 2^{53}$ et $x_2 = -(2^{54} - 2)$.
Que vaut la somme exacte (sans machine car la machine se trompe...) ?
Voyons maintenant le schéma du fonctionnement sur machine :



Recherche

Expliquez les résultats trouvés. Que va faire l'algorithme de somme compensée ? Et celui de somme en cascade ?

Recherche

Explorez la fonction `fsum` de la bibliothèque `math`.
https://docs.python.org/3.5/library/math.html#math.fsum

On note $u = \frac{\varepsilon_M}{2}$ et $\gamma_n = \frac{nu}{1-nu}$. Alors nos trois amis Rump et coll. [2008] ont démontré le théorème suivant :

Théorème 1 - 3

En appliquant l'algorithme de Pichat-Rump-Ogita-Oishi à une famille $(x_i)_{1 \leqslant i \leqslant n}$ de flottants et si $nu < 1$ alors, même en cas de sous-capacité mais sans sur-capacité, le résultat s retourné par l'algorithme vérifie :

$$\left| s - \sum_{i=1}^{n} x_i \right| \leqslant u \left| E \sum_{i=1}^{n} x_i \right| + \gamma_{n-1}^2 \sum_{i=1}^{n} |x_i|$$

Ça fait peur, hein ?

6 Et la multiplication ? Et la division ? Et le sinus ? Et...

Pour la multiplication, on procède à peu près de la même manière. On peut gagner du temps si le processeur utilise la FMA (Fused Multiple Add) mais ceci est une autre histoire...
Et la division ? On cherche l'inverse x de a : on est donc amené à résoudre l'équation $\frac{1}{x} - a = 0$ d'inconnue a qui n'est pas linéaire.
Et la racine carrée ? Là aussi, on quitte le linéaire.
Il nous faut donc explorer des outils un peu plus sophistiqués...ce que nous allons faire dans le chapitre suivant.

EXERCICES

Recherche 1 - 1

Comment est codé 2 en tant qu'entier 32 bits ? En tant que binary32 ? Comment est-il codé en mémoire sous forme hexadécimale ?

Recherche 1 - 2

Donnez au format *binary32* et *binary64* les représentations mémoires hexadécimales des nombres 125,25 puis 0,375 puis $-0,375$ puis 0,44.

Recherche 1 - 3

Donnez la représentation mémoire hexadécimale au format *binary32* de 2^{-130}.

Recherche 1 - 4

Comment traduiriez-vous par une égalité qu'un nombre $x \in [0,1[$ s'écrit $b_1 b_2 ... b_n$ dans une base β ?
Soit l'algorithme :

Fonction `multSuccessives(x:` *réel* `base:` *entier naturel* `):` *liste d'entiers naturels*
`nb` ← `x`
`dec = []`
TantQue `nb` ≠ `0` **Faire**
  `bi` ← `nb` × `base`
  `dec` ← `dec ++ liste(⌊bi⌋)`
  `nb` ← `{bi}`
FinTantQue
Retourner `dec`

On notera $\{x\}$ la partie fractionnaire de x.
Que pensez-vous de sa précondition ? De sa postcondition ? de sa terminaison ? De sa correction ? De sa complexité ?

Recherche 1 - 5

On suppose qu'on dispose d'une fonction **`FloatToIntBits`** qui renvoie l'écriture machine en base 2 sur 32 bits d'un réel et des fonctions bit à bit habituelles.
Donnez des fonctions renvoyant le signe, l'exposant, la mantisse d'un réel 32 bits.
Pour vous faire plaisir, vous pourrez coder tout ça en Java...Par exemple, voici une fonction qui donne la représentation mémoire hexadécimale d'un floattant 32 bits :

```java
static String memoire(float reel){
    int bits;
    bits = Float.floatToIntBits(reel);
    return (Integer.toHexString(bits));
}
```

Recherche 1 - 6

Que pensez-vous de ça :

```
from math import sin, pi
def der(f,h) :
    return lambda x: (f(x + h) - f(x)) / h

def opp(f) :
    return lambda x : -f(x)

cos_app   = lambda h : der(sin, h)
sin_app   = lambda h : opp(der(cos_app(h), h))
cos_app_2 = lambda h : der(sin_app(h), h)
sin_app_2 = lambda h : opp(der(cos_app_2(h), h))
```

Qui donne :

```
In [94]: sin(pi / 2)
Out[94]: 1.0

In [95]: sin_app(1e-7)(pi / 2)
Out[95]: 0.9992007221626409

In [96]: sin_app_2(1e-7)(pi / 2)
Out[96]: -1110223024625.1565
```

Recherche 1 - 7 Écriture en base 2 d'un nombre inférieur à 1

Déterminons l'écriture de 0,1 en base 2. Ce nombre est égal à 1/10 en base 10 c'est-à-dire 1/1010 en base 2. Posons la division :

```
 1       | 1010
         ------
 10      | 0,00011
 100     |
 1000    |
 10000   |
- 1010   |
------   |
   1100  |
-  1010  |
-------  |
     10  |
```

Ça ne tombe pas juste, alors que 0,125 en base 2 s'écrit 0,001 exactement car il est égal à $1/8 = 2^{-3}$.
Voici un petit algorithme permettant de convertir un nombre en base 10 de valeur absolue inférieure à 1 en base 2.
Soit $x = a_1 \times 2^{-1} + a_2 \times 2^{-2} + a_3 \times 2^{-3} + \cdots + a_n \times 2^n$ l'écriture de x tel que $|x| < 1$ en base 2 avec les a_i égaux à 0 ou 1.
Alors $2x = a_1 + a_2 \times 2^{-1} + a_3 \times 2^{-2} + \cdots + a_n \times 2^{n-1}$: a_1 est donc la partie entière de $2x$.
Puis $2(2x - a_1) = a_2 + a_3 \times 2^{-1} + \cdots + a_n \times 2^{n-2}$: a_2 est alors la partie entière de $2(2x - a_1)$,...
Déterminez une fonction Python donnant cette décomposition.

```
In [5]: petits2bits(0.125,50)
Out[5]: '0:001'

In [6]: petits2bits(0.1,50)
Out[6]: '0:00011001100110011001100110011001100110011001100110'
```

Comment la modifier pour obtenir l'arrondi sur 64 bits selon l'IEEE-754 avec l'exposant ?
On pourra utiliser à bon escient la bibliothèque **`bigfloat`** qui fait appel à la bibliothèque C **`MPFR`**.

```
In [194]: petits2bits(2, 0.1, 80)
Out[196]: '0:00011001100110011001100110011001100110011001100110011001100110011001100110011001'
```

Recherche 1 - 8

Pourquoi `3 * 0.1 != 0.3` ? Commentez le code suivant sachant que la fonction **`ieee(x, garde, l_pow, l_mant)`** renvoie l'écriture IEEE 754 du flottant **`x`** avec un certain nombre de bits de garde et un format de mantisse et d'exposant :

```
In [201]: ieee(0.1,3,11,52)
normal
ulp = 2**-56
les bits oubliés sont 10
Out[201]: '0 01111111011 1001100110011001100110011001100110011001100110011010'

In [202]: ieee(2*0.1,3,11,52)
normal
ulp = 2**-55
les bits oubliés sont 10
Out[202]: '0 01111111100 1001100110011001100110011001100110011001100110011010'

In [203]: ieee(3*0.1,3,11,52)
normal
ulp = 2**-54
les bits oubliés sont 00
Out[203]: '0 01111111101 0011001100110011001100110011001100110011001100110100'

In [204]: ieee(0.3,3,11,52)
normal
ulp = 2**-54
```

```
les bits oubliés sont 00
Out[204]: '0 01111111101 0011001100110011001100110011001100110011001100110011'

In [205]: ieee(4*0.1,3,11,52)
normal
ulp = 2**-54
les bits oubliés sont 10
Out[205]: '0 01111111101 1001100110011001100110011001100110011001100110011010'

In [206]: ieee(0.4,3,11,52)
normal
ulp = 2**-54
les bits oubliés sont 10
Out[206]: '0 01111111101 1001100110011001100110011001100110011001100110011010'

In [207]: ieee(0.5,3,11,52)
normal
ulp = 2**-53
les bits oubliés sont 00
Out[207]: '0 01111111110 0000000000000000000000000000000000000000000000000000'

In [208]: ieee(5*0.1,3,11,52)
normal
ulp = 2**-53
les bits oubliés sont 00
Out[208]: '0 01111111110 0000000000000000000000000000000000000000000000000000'
```

Recherche 1 - 9

Pourquoi le *Patriot* a raté son *Scud* de 600m ?
Représentez 1/10 avec une mantisse de 24 bits et tronquez le résultat.
Calculez l'erreur en seconde puis après 100 heures d'utilisation. Sachant qu'un Scud vole à $1676\,\mathrm{m\,s^{-1}}$, quelle est environ l'erreur commise en mètres ?
Et si on avait arrondi au plus proche ?

Recherche 1 - 10

Représentez 1,32 sur *toy16* : 1 bit de signe, 4 bits d'exposant et 11 bits pour f.

Recherche 1 - 11

En utilisant vos carreaux, représentez les VF normaux de *toy7*. Faites de même avec les sous-normaux.

Recherche 1 - 12

Quel est le « PGCD » de tous les *binary64* ?

Recherche 1 - 13

Représentez $1,0011$ et $-1,0011$ en *toy7* selon les quatre modes d'arrondi usuels.
Représentez également $0,00011$ en *toy7* puis $0,0000001$.

Recherche 1 - 14

Calculez $ulp(1)$ en *binary32*, *binary64* et *toy7*.

Recherche 1 - 15

Complétez le tableau page 14.

Recherche 1 - 16

Le recours au dixième peut être mortel sur machine. Il vaut mieux diviser par deux mais est-ce pour autant sans danger ?
Effectuez la division répétée de 1 par 2 en *toy7*. Que cela vous inspire-t-il ?

Recherche 1 - 17

1. Comment expliquer les résultats suivants obtenus avec un langage suivant la norme IEEE 754 :

```
> let f(x) = x^2 / sqrt(x^3 + 1)

```

```
> f(1e100)
1.0e50

> f(1e150)
0.0

> f(1e200)
NaN
```

2. Comment interprétez-vous les réponses pythonesques suivantes :

```
from math import sqrt

def f(x) :
    return x**2 / sqrt(x**3 + 1)
```

```
In [99]: f(1e100)
Out[99]: 1e+50

In [100]: f(1e150)
---------------------------------------------------------------------------
OverflowError                             Traceback (most recent call last)
<ipython-input-100-79775d85f31a> in <module>()
----> 1 f(1e150)

/home/moi/PROG/PYTHON/IntroIeee/ieee_poly.py in f(x)

OverflowError: (34, 'Numerical result out of range')

In [101]: f(1e200)
---------------------------------------------------------------------------
OverflowError                             Traceback (most recent call last)
<ipython-input-101-c1d077e9d28e> in <module>()
----> 1 f(1e200)

/home/moi/PROG/PYTHON/IntroIeee/ieee_poly.py in f(x)

OverflowError: (34, 'Numerical result out of range')
```

3. On pose `let f(x)= exp(x)/x`. Prévoyez ce que donnera `f(1e500)`, `f(1e100)`, `f(1e-100)`, `f(0)`.

Recherche 1 - 18
Petit jeu à 2 : effectuez toute sorte de multiplications et d'additions en *toy7* en comparant avec votre voisin(ne).

Recherche 1 - 19
En utilisant `floatToDigits` de la bibliothèque `Numeric` écrivez une fonction qui calcule la somme de deux flottants puis la multiplication de deux flottants en n'oubliant pas les cas particuliers...

Recherche 1 - 20
Soit a et b deux VF tels que $|a| \geqslant |b|$. Démontrez que $a+b$ est un multiple de 2^{E_b-p}.
Donnez alors une nouvelle démonstration du lemme 1 - 1 page 23

Recherche 1 - 21
Terminez la démonstration de la section 1.5.2.1 page 20 et démontrez le programme de la section 1.5.2.3 page 22

Recherche 1 - 22
Répondez à la question de l'encadré *Danger* page 24

Recherche 1 - 23
Appliquez l'algorithme Fast2Sum aux deux cas suivants en posant chaque opération sachant qu'on travaille avec un mantisse de 4 bits :

1. $x = 1111 \times 2^2$ et $y = 1001$;
2. $x = 1001 \times 2^1$ et $y = -1011$.

Recherche 1 - 24

Écrivez une fonction `twoSum` en Python qui calcule les nombres `s` et `d` de l'algorithme du théorème 1 - 2 page 26.

Recherche 1 - 25

1. Traduisez la fonction `sommeKahan` page 26 en impératif en pseudo-spaghetti puis par exemple en C ou (et) en Java.
2. Répondez aux questions de l'encadré de la section 1.5.4.3 page 26.
3. On voudrait calculer $\sum_{k=1}^{100\,000} RN\left(\frac{1}{k}\right)$.
   i. Utilisez une somme récursive standard.
   ii. Utilisez `sommeKahan`.
   iii. Voici les premières bonnes décimales : 12.090 146 129 863 428 0. Estimez les erreurs commises en les exprimant en ulps.
   iv. Et $\sum_{k=1}^{100\,000} \frac{1}{k}$ dans tout ça ?

Testez en utilisant des fonctions `somme_kahan`, `somme_pichat` ainsi que les fonctions Python `sum` et `math.fsum`.

Recherche 1 - 26

J'ose espérer que vous savez résoudre une équation du type $ax^2 + bx + c = 0$ avec $a \neq 0$...
Les racines, si elles existent, sont données par une formule bien connue dépendant de a, b et c :

$$r = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{-b \pm \sqrt{\Delta}}{2a} \quad \text{avec} \quad \Delta = b^2 - 4ac$$

Où peuvent se cacher d'éventuelles annulations catastrophiques ? Étudiez ces cas avec attention, voyez si vous pouvez éviter les éliminations catastrophiques en réécrivant les formules un peu dans l'esprit de la « levée d'indétermination ».

1. Que se passe-t-il lorsque $b^2 \gg |4ac|$? En quoi la formule $\frac{-\left(b+\text{signe}(b)\sqrt{\Delta}\right)}{2a}$ peut aider ?
2. Que se passe-t-il lorsque $b^2 \approx 4ac$? Peut-on y remédier ? Que peut-on dire de Δ par rapport à b^2 ? Y a-t-il élimination catastrophique ?
3. Que se passe-t-il dans le cas de l'équation $10^{200}x^2 - 3 \times 10^{200}x + 2 \times 10^{200} = 0$?
4. Et dans le cas de $10^{-200}x^2 - 3x + 2 \times 10^{200} = 0$?
5. Moralité ?

Recherche 1 - 27 Bits de garde, d'arrondi, collant

Soustrayez $1,101 \times 2^{-2}$ puis $1,111 \times 2^{-2}$ à 1×2^0 en *toy7* en gardant tous les bits, même ceux qui dépassent le format de la mantisse puis refaites le calcul en faisant disparaître les bits qui dépassent de la mantisse.
Pourquoi parle-ton de *bit de garde* ?
En fait, il y a toujours trois bits à droite de la mantisse : G le bit de garde (*guard*), R le bit d'arrondi (*rounding*) et S le bit collant (*sticky*).
Les deux premiers sont comme des paniers qui recueillent les bits translatés (*shifted*). Le dernier est un panier absorbant qui est un ou inclusif des bits translatés au-delà de R.

Recherche 1 - 28 Calcul de exp 1 et puissances de 10

On veut calculer une approximation de e (= exp(1)) comme limite au voisinage de l'infini de $\left(1 + \frac{1}{n}\right)^n$.

1. Rappel : f est dérivable au voisinage de x si, et seulement si, $\lim_{t \to x} \frac{f(t) - f(x)}{t - x} \in \mathbb{R}$. Dans ce cas on note $f'(x)$ cette valeur.
2. Utiliser ce rappel pour calculer $\lim_{t \to 0} \frac{\ln(1+t)}{t}$.
3. Déduisez-en $\lim_{n \to +\infty} n \ln\left(1 + \frac{1}{n}\right)$ puis $\lim_{n \to +\infty} \left(1 + \frac{1}{n}\right)^n$.
4. On note `expn n = (1 + 1/n)**n`. Comparez et commentez :

```
>>> [abs(exp(1) - expn(10**k)) for k in range(20)]
>>> [abs(exp(1) - expn(2**(3*k))) for k in range(20)]
```

Recherche 1 - 29 0,1 en base 2

Nous démontrerons dans le chapitre suivant que, pour tout réel q vérifiant $|q| < 1$ on a

$$\sum_{k=k_0}^{+\infty} q^k = \frac{q^{k_0}}{1-q}$$

1. Démontrez que $\sum_{i=1}^{+\infty} \left(2^{-4i} + 2^{-4i-1}\right) = \frac{1}{10}$.
2. Déduisez-en le développement de 1/10 en base 2.
3. Montrez que, en *binary32*, $\frac{\widehat{x}-x}{x} = -\frac{\varepsilon_M}{8}$.

Recherche 1 - 30 Max IEEE

On peut calculer le maximum de deux réels ainsi :

Fonction max(x, y : *flottants*) : *flottant*
Si $x > y$ **Alors**
  max = x
Sinon
  max = y
FinSi

Est-ce adapté à la recherche du maximum de deux flottants en arithmétique IEEE ?

Recherche 1 - 31 Options GCC et double arrondi

Voici un petit code C pour effectuer une multiplication [Muller et coll., 2010] :

```c
#include <stdio.h>

int
main(void)
{
  double a = 1848874847.0;
  double b = 19954562207.0;
  double c;  c = a * b;
  printf("c = %20.19e\n", c);
  return 0;
}
```

et voici les résultats obtenus en utilisant deux options de compilation différentes :

— Avec `mfpmath=387` :

```
$ gcc -mfpmath=387 multi.c -o multi_387
$ ./multi_387
c = 3.6893488147419103232e+19
```

— Avec `mfpmath=sse` :

```
$ gcc -mfpmath=sse multi.c -o multi_sse
$ ./multi_sse
c = 3.6893488147419111424e+19
```

Que se passe-t-il sachant qu'on trouve dans `man gcc` les lignes suivantes :

```
-mfpmath=unit
         Generate floating-point arithmetic for selected unit unit.  The choices for unit are:

```

```
387 Use the standard 387 floating-point coprocessor present on the majority of chips and emulated
    → otherwise.  Code compiled with this option runs almost everywhere.  The temporary
    → results are computed in 80-bit precision  instead of the precision specified by the
    → type, resulting in slightly different results compared to most of other chips.  This
    → is the default choice for i386 compiler.

sse Use scalar floating-point instructions present in the SSE instruction set.  This instruction
    → set is supported by Pentium3 and newer chips, in the AMD line by Athlon-4, Athlon-xp
    → and Athlon-mp chips.  The earlier version of SSE instruction set supports only
    → single-precision arithmetic, thus the double and extended-precision arithmetic are
    → still done using 387.  A later version, present only in Pentium4 and the future
    → AMDx86-64 chips, supports double-precision arithmetic too.
```

et qu'en précision infinie on trouve :

```
>>>  1848874847 * 19954562207
36893488147419107329
>>> 36893488147419107329 - 36893488147419111424
-4095
>>>  36893488147419107329 -  36893488147419103232
4097
```

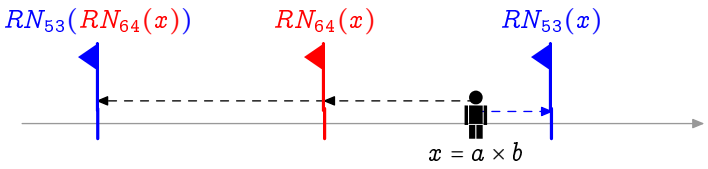
et que la représentation binaire exacte du produit est :

$$\overbrace{\underbrace{10000000000000000000000000000000000000000000000000000}_{53\text{ bits}}10000000000}^{64\text{ bits}}01$$

et que le calcul effectué avec l'arrondi au plus près en *binary64* est

```
*Main Numeric> floatToDigits 2 (1848874847 * 19954562207)
([1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    0,0,1],66)
```

On pourra méditer sur le schéma suivant :



Recherche 1 - 32 Double arrondi

Nous montrerons dans le chapitre suivant que $\sqrt{1-u} \approx 1 - \frac{1}{2}u - \frac{1}{8}u^2$ pour de petites valeurs de u.
Le *double étendu* est un format sur 80 bits avec 15 bits d'exposant et 64 bits de pseudo-mantisse.
Quel est le résultat de $\sqrt{1-2^{-53}}$ en *binary64* avec ou sans double arrondi depuis un format étendu ?
On pourra écrire le développement en binaire sur 110 bits de $1 - \frac{1}{2}u - \frac{1}{8}u^2$ et arrondir selon le mode 387 ou selon le mode SSE comme cela a été illustré dans l'exercice précédent..

Recherche 1 - 33 MPFR

La bibliothèque MPFR qui est une extension GNU depuis 2009 a été fondée au départ en 1998 par Guillaume Hanrot, Jean-Michel Muller, Joris van der Hoeven et Paul Zimmermann.
Elle a d'abord été conçue pour permettre d'obtenir des résultats corrects, portables et reproductible, parfois au dépends de la vitesse mais elle reste très efficace.
Par exemple :

```
#include <stdio.h>
#include <mpfr.h>

int main (void)
```

```c
{
  mpfr_t un , dix , trois, zero_1, zero_3, trois_fois_zero_1, diff ; /* Déclarations simultanées */
  int test;

  mpfr_set_default_prec (53) ; /* Fixer la precision par defaut */
  mpfr_set_emin ( -1073) ; /* Fixer l' exposant emin ( en realite , emin - precision +2) */
  mpfr_set_emax (1024) ; /* Fixer l' exposant emax ( en realite , emax +1) */
  mpfr_inits (zero_1,zero_3, trois_fois_zero_1, (mpfr_ptr) 0); /* Précision par defaut 53 bits */
  mpfr_inits2 (128, un, trois, dix , diff,  (mpfr_ptr) 0) ; /* Précision de 128 bits exactement */
  mpfr_set_d (un , 1.0 , MPFR_RNDN ); /* MPFR_RND Division en arrondi au plus près */
  mpfr_set_d (trois , 3.0 , MPFR_RNDN );
  mpfr_set_d (dix , 10. , MPFR_RNDN );

  mpfr_div (zero_1 , un , dix , MPFR_RNDN );
  mpfr_div (zero_3 , trois , dix , MPFR_RNDN );
  mpfr_mul (trois_fois_zero_1, trois, zero_1, MPFR_RNDN  );
  mpfr_sub (diff, trois_fois_zero_1,  zero_3, MPFR_RNDN  );

  test = mpfr_equal_p (trois_fois_zero_1, zero_3);

  mpfr_printf (" Avec %3d bits de précision , 1/10     ~ %RNe \n" , mpfr_get_prec (zero_1) , zero_1);
  mpfr_printf (" Avec %3d bits de précision , 1/10     ~ %.38RNe \n" , mpfr_get_prec (zero_1) , zero_1);
  mpfr_printf (" Avec %3d bits de précision , 1/10     ~ %.38Rg \n" , mpfr_get_prec (zero_1) , zero_1);
  mpfr_printf (" Avec %3d bits de précision , 3/10     ~ %.38RNe \n" , mpfr_get_prec (zero_3) , zero_3);
  mpfr_printf (" Avec %3d bits de précision , 3 * 1/10 ~ %.38RNe \n" , mpfr_get_prec (trois_fois_zero_1) ,
        ↪  trois_fois_zero_1);
  printf (" 3 * 1/10 == 3/10 ? (0 si faux, /= 0 si vrai) :  %u \n" , test);
  mpfr_printf (" Avec %3d bits de précision , la différence 3*1/10 - 0.3 vaut  ~ %.38RNe \n" , mpfr_get_prec
        ↪  (diff) , diff);

  mpfr_clears ( un , dix , trois, zero_1, zero_3, trois_fois_zero_1, diff, (mpfr_ptr) 0);
  return 0;
}
```

renvoie

```
$ gcc -o mpfr_essai mpfr_essai.c -lmpfr -lgmp
$ ./mpfr_essai

 Avec  53 bits de precision , 1/10     ~ 1.0000000000000001e-01
 Avec  53 bits de precision , 1/10     ~ 1.0000000000000000555111512312578270211 8e-01
 Avec  53 bits de precision , 1/10     ~ 0.1000000000000000055511151231257827021 2
 Avec  53 bits de precision , 3/10     ~ 2.9999999999999998889776975374843459576e-01
 Avec  53 bits de precision , 3 * 1/10 ~ 3.0000000000000004440892098500626161694 5e-01
 3 * 1/10 == 3/10 ? (0 si faux, /= 0 si vrai) :  0
 Avec 128 bits de precision , la différence 3*1/10 - 0.3 vaut  ~
        ↪  5.5511151231257827021181583404541015625 0e-17
```

Créez alors une fonction qui calcule la somme des inverses des entiers de 1 à 100 000 en prenant la précision en argument.
Sur Sage, on peut utiliser la bibliothèque MPFR à l'aide de la fonction `RealField(p)` avec p la précision :

```
sage:  RealField(128)(RealField(53)(1)/ RealField(53)(10))
0.10000000000000000055511151231257827021 2
sage:  RealField(128)(RealField(53)(3)/ RealField(53)(10))
0.29999999999999999888977697537484345957 6
sage:  RealField(128)(RealField(53)(3) *  RealField(53)(0.1))
0.30000000000000004440892098500626161695
```

Recherche 1 - 34 6 points

Répondez aux questions suivantes en justifiant vos réponses :

1. Est-ce que l'addition des VF est associative ?

2. Est-ce que la multiplication des VF est associative ?
3. Est-ce que la multiplication est distributive sur l'addition ?

Recherche 1 - 35 6 points

1. Comment expliquer les résultats suivants :

Haskell
```
*Main> let f(x) = x^2 / sqrt(x^3 + 1)
*Main> f(1e100)
1.0e50
```

Haskell
```
*Main> f(1e150)
0.0
*Main> f(1e200)
NaN
```

2. On pose `f(x)= (x**2 - x)/ (2 * (sqrt (x**4 - x**2)))`. Prévoyez ce que donnera `f(1e500)`, `f(1e100)`, `f(1e-100)`, `f(0)`.
3. Comment améliorer les calculs faits par la machine ?

Recherche 1 - 36 4 points

On considère la fonction suivante :

```
1 def toP(a, cpt) :
2     return cpt - 1 if (a + 1) - a != 1. else toP(2.*a, cpt + 1)
3
4 p = toP(1., 0.)
```

Imaginons que Python travaille uniquement en *toy7* (1.0 est donc un VF codé sur 7 bits). Que vaut alors p ? Justifiez votre réponse.

Recherche 1 - 37

Exemple dû à Jean-Michel Muller (in Muller et coll. [2010])
M. X a récemment été à sa banque (Chaotic Bank Society), pour connaître les nouvelles offres proposées aux meilleurs clients. Son banquier lui propose l'offre suivante : « vous déposez tout d'abord e - 1 euros sur votre compte (où e = 2.7182818... est la base du logarithme népérien). La première année, nous prenons 1 euro sur votre compte de frais de gestion. Par contre, la deuxième année est plus intéressante pour vous, car nous multiplions votre capital restant par 2 et prenons 1 euro de frais de gestion. La troisième année est encore plus intéressante, car nous multiplions votre capital par 3 et prenons 1 euro de frais de gestion. Et ainsi de suite : la n-ième année, nous multiplions votre capital par n et prenons 1 euro de frais de gestion. Intéressant, non ? » Pour prendre sa décision, M. X décide de demander l'aide d'un informaticien et se retourne vers vous.

Recherche 1 - 38 Bac

Le sujet du Bac S des Centres Étrangers 2012 fait étudier la suite :

$$I_{n+2} = \frac{1}{2}\mathrm{e} - \frac{n+1}{2} I_n, \qquad I_1 = \frac{1}{2}(\mathrm{e}-1)$$

Calculez I_{59} à la machine...

Recherche 1 - 39 DS 2015

Voici une capture d'un interpréteur OCaml :

Pseudo-interpréteur Ocaml
```
# let a = 2.0 ;;
val a : float = 2.
# let n = 54.0 ;;
val n : float = 54.
# a ** n ;;
- : float = 18014398509481984.
# a ** n  +. 1.0 ;;
- : float = 18014398509481984.
```

Pseudo-interpréteur Ocaml
```
# a ** n  +. 2.0 ;;
- : float = 18014398509481984.
# a ** n  +. 3.0 ;;
- : float = 18014398509481988.
# a ** n  +. 4.0 ;;
- : float = 18014398509481988.
# a ** n  +. 5.0 ;;
- : float = 18014398509481988.
```

Expliquez les résultats obtenus. Quel sera le résultat affiché suite à la commande suivante :

Pseudo-interpréteur Ocaml

```
# a ** n  +. 10.0 ;;
- : float = ????????????????
```

Recherche 1 - 40 DS 2015

On veut calculer une approximation de e (= exp(1)) comme limite au voisinage de l'infini de $\left(1+\frac{1}{n}\right)^n$. On rappelle : *$f$ est dérivable au voisinage de x si, et seulement si,* $\lim_{t\to x}\frac{f(t)-f(x)}{t-x}\in\mathbb{R}$. *Dans ce cas on note $f'(x)$ cette valeur.*

1. Utilisez ce rappel pour calculer $\lim_{t\to 0}\frac{\ln(1+t)}{t}$.
2. On rappelle que $\lim_{n\to+\infty}\frac{1}{n}=0$, que $\lim_{X\to 1}\exp(X)=\mathrm{e}$ et que $\frac{1}{\frac{1}{n}}=n$.

   Déduisez-en $\lim_{n\to+\infty} n\ln\left(1+\frac{1}{n}\right)$ puis $\lim_{n\to+\infty}\left(1+\frac{1}{n}\right)^n$.
3. Expliquez et commentez les résultats observés sur l'interpréteur d'OCaml :

Pseudo-interpréteur Ocaml

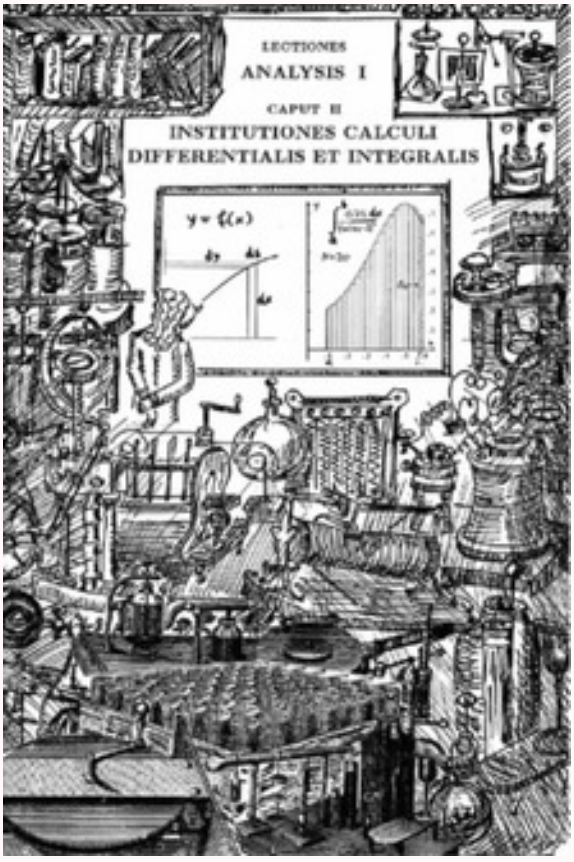
```
# exp 1.0 ;;
- : float = 2.71828182845904509
# let expn n = (1.0 +. 1.0 /. n) ** n ;;
val expn : float -> float = <fun>
# expn (10.0 ** 10.0) ;;
- : float = 2.71828205323478755
# expn (2.0 ** 33.0) ;;
- : float = 2.71828182830082055
```

Pseudo-interpréteur Ocaml

```
# expn (10.0 ** 15.0) ;;
- : float = 3.03503520654926184
# expn (2.0 ** 52.0) ;;
- : float = 2.71828182845904509
# expn (10.0 ** 16.0) ;;
- : float = 1.
# expn (2.0 ** 53.0) ;;
- : float = 1.
```

CHAPITRE 2

Un soupçon de calcul différentiel



La notion d'*infiniment petit* a crée des guerres entre les savants pendant deux siècles. Elle n'a été formalisée rigoureusement qu'à la fin du XIX^{e} siècle. Cependant, nous allons rester fidèle au cheminement de l'esprit humain et nous attacher à cette notion qui est pédagogiquement plus intéressante et se rapproche de l'arithmétique des flottants sur machine Cette approche historique et intuitive du calcul a été magistralement mise en œuvre dans Hairer et Wanner [2001].

1 Le calcul infinitésimal

Infiniment petits

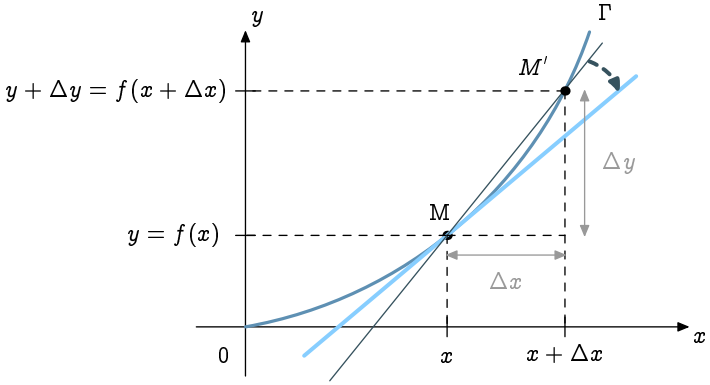
La notion de dérivée est née bien avant celle de limite : dès le XVII[e] Fermat, Newton, Leibniz, Jean Bernoulli, vont développer le calcul infinitésimal, avant que Bolzano et Weierstrass ne perfectionnent la notion de limite deux siècles plus tard.
Mais les approches des savants du XVII[e] s'adaptent bien à l'utilisation des flottants sur machine. L'informatique nous incite donc à un retour aux sources historiques.

Remarque

Analyse non standard et hyperréels

Il est à noter que dans les années 1960, un logicien américain d'origine allemande (encore un !...), Abraham Robinson, mit au point l'*analyse non standard* en utilisant les *nombres hyperréels*. Il s'agit de nos réels auxquels on adjoint des nombres *infinitésimaux* qui sont strictement inférieurs en valeur absolue à tout nombre réel non nul, et les nombres *infiniment grands* dont les inverses sont infinitésimaux. Ceci est fait de manière très rigoureuse et permet de justifier une incroyable intuition que Fermat avait présenté trois siècles plus tôt.
C'est également assez proche de la définition des flottants selon la norme IEEE 754

Pour Jean Bernoulli (1691), **les infiniment petits sont des quantités qui peuvent être ajoutées à des quantités finies sans changer leurs valeurs. Les courbes sont des polygones à côtés infiniment courts.**
Voici qui s'adapte à la notion de sous-capacité et au tracé de courbes sur machine.
Voyons ce que cela donne pour les tangentes à la courbe d'équation $y = x^2$:



Si x augmente de Δx, alors y devient $y + \Delta y = (x + \Delta x)^2 = x^2 + 2x\Delta x + (\Delta x)^2$. Or $y = x^2$.
Pour des valeurs de Δx non nulles, cela donne donc :

$$\frac{\Delta y}{\Delta x} = 2x + \Delta x$$

C'est ici qu'on a besoin de la notion intuitive d'infiniment petits...qui commence pour nous avec la sous-capacité.

Remarque

C'est Joseph-Louis Lagrange(1736-1813) ou plutôt Giuseppe Lodovico Lagrangia, un mathématicien Piémontais d'origine française, qui introduisit les termes et les notations que nous utilisons actuellement :

> Nous appellerons la fonction f_x, *fonction primitive*, par rapport aux fonctions f'_x, f''_x, &c. qui en dérivent, et nous appellerons celles-ci, *fonctions dérivées*, par rapport à celle-là.

Il a aussi réfuté la notion d'infiniment petit et a tenté de fonder l'analyse sur les séries de Taylor : nous allons en reparler.

Règles de dérivation

Dérivée d'une combinaison linéaire

Soit $y(x) = a \cdot u(x) + b \cdot v(x)$ avec a et b des constantes réelles. En posant comme précédemment $y + \Delta(y) = y(x + \Delta x)$ puis $u + \Delta(u) = u(x + \Delta x)$ et $v + \Delta(v) = v(x + \Delta x)$, nous obtenons $\Delta y = a \cdot \Delta u + b \cdot \Delta v$. Alors :

Théorème 2 - 1

Dérivation d'une combinaison linéaire

$$y = au + bv \Longrightarrow \frac{\mathrm{d}y}{\mathrm{d}x} = a \cdot \frac{\mathrm{d}u}{\mathrm{d}x} + b \cdot \frac{\mathrm{d}v}{\mathrm{d}x} \text{ ou bien } y' = a \cdot u' + b \cdot v'$$

Dérivée d'un produit

Recherche

Essayez de prouver de même une règle de dérivation d'un produit $y = u \cdot v$.

Théorème 2 - 2

Dérivation d'un produit

$$y = u \cdot v \Longrightarrow$$

Dérivée d'un quotient

Nous allons d'abord établir un résultat préliminaire. Pour x « petit » on « sent » que $\frac{1}{1-x} \approx 1$. Par exemple $\frac{1}{1-10^{-5}} \approx 1,0000100001 = 1 + 10^{-5} + 10^{-10}$.
Essayons d'être plus précis sur l'approximation. Posons

$$\frac{1}{1-x} = 1 + \delta$$

et essayons de déterminer ce δ sachant qu'on peut le négliger devant x.
Multiplions chaque membre de l'équation par $1 - x$. On obtient :

$$1 = 1 - x + \delta(1 - x)$$

c'est-à-dire, après simplifications :

$$\delta = x + \delta x$$

Ainsi, comme δx est négligeable devant x, on peut considérer que $\delta \approx x$ donc :

$$\frac{1}{1-x} \approx 1 + x$$

Remarque

Série géométrique

En 1593, François Viète établit avec une méthode similaire que, pour tout réel x tel que $|x| < 1$, on a :

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + x^4 + x^5 + \cdots$$

Recherche

Vous pouvez maintenant essayer d'établir une formule donnant la dérivée d'un quotient $y = \frac{u}{v}$.

Théorème 2 - 3

Dérivation d'un quotient

$$y = \frac{u}{v} \Longrightarrow$$

Fonctions composées

Soit trois fonctions f, g et h telles que $h = f \circ g$. Ainsi, $h(x) = f\big(g(x)\big)$.
Posons $g(x+\Delta x) = z + \Delta z$ et $h(x+\Delta x) = y + \Delta y$.
Alors $y + \Delta y = f\big(g(x+\Delta x)\big) = f(z+\Delta z)$. Or $\frac{\Delta y}{\Delta x} = \frac{\Delta y}{\Delta z} \cdot \frac{\Delta z}{\Delta x}$. On obtient donc :

Théorème 2 - 4

Dérivation de fonctions composées

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\mathrm{d}y}{\mathrm{d}z} \cdot \frac{\mathrm{d}z}{\mathrm{d}x} \text{ ou } h'(x) = f'\big(g(x)\big) \cdot g'(x)$$

Par exemple, si $h(x) = (3x-5)^2$, alors $y = h(x) = f\big(g(x)\big)$ avec $z = g(x) = 3x - 5$ et $f(z) = z^2$.
On en déduit que $h'(x) = \frac{\mathrm{d}y}{\mathrm{d}z} \cdot \frac{\mathrm{d}z}{\mathrm{d}x} = 2z \cdot 3 = 6(3x-5)$.

2 Comment calculer un logarithme sur machine..au XVII^{e} siècle ?

Calcul d'une approximation d'une racine carrée par la méthode de Héron

Le mathématicien Héron d'Alexandrie n'avait pas attendu Newton et le calcul différentiel pour trouver une méthode permettant de déterminer une approximation de la racine carrée d'un nombre entier positif puisqu'il a vécu seize siècles avant Sir Isaac.
Si x_n est une approximation strictement positive par défaut de $\sqrt{a}$, alors a/x_n est une approximation par excès de $\sqrt{a}$ (pourquoi ?) et vice-versa.
La moyenne arithmétique de ces deux approximations est $\frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$ et constitue une meilleure approximation que les deux précédentes.
On peut montrer c'est une approximation par excès (en développant $\left(x_n - \sqrt{a}\right)^2$ par exemple).

```
1 def heron(a, x, n) :
2     return x if n == 0 else heron(a, (x + a/x) / 2, n - 1)
```

Le résultat est assez bluffant :

```
1 In [5]: heron(2,1,5)
2 Out[5]: 1.414213562373095
3
4 In [6]: 2**0.5
5 Out[6]: 1.4142135623730951
```

Il y a cependant des discussions au sujet de l'exactitude des calculs dans $\mathbb{V}$: nous en reparlerons dans un prochain chapitre quand nous saurons diviser deux flottants :-)

Calcul d'une approximation de log 2 par la méthode de Briggs

Michael Stifel s'est intéressé dès 1544 aux fonctions vérifiant $\ell(x \cdot y) = \ell(x) + \ell(y)$ afin de simplifier les calculs des astronomes et des navigateurs. Cependant, cela nécessite de disposer de tables *logarithmiques* ($\lambda o \gamma o \varsigma$: mot, relation, $\alpha \rho \iota \theta \mu o \varsigma$: nombre ; les logarithmes sont donc des relations entre les nombres avant d'être l'anagramme d'algorithmes...).
Nous allons nous occuper comme Henry Briggs du logarithme de base 10.
Sachant que $\log_{10}(10) = 1$ et que $\log_{10}(a^p) = p \cdot \log_{10}(a)$ pour tout rationnel p, on en déduit alors que $\log_{10}\left(\sqrt{10}\right) = \frac{1}{2}$, $\log_{10}\left(\sqrt{\sqrt{10}}\right) = \frac{1}{4}$, etc.
Nous pouvons utiliser l'algorithme de Horner pour calculer les approximations des racines successives de 10 et on obtient le tableau suivant :

Nombres	10,0000	3,1623	1,7783	1,3335	1,0000
Logarithmes	1	0,5	0,25	0,125	0

Le problème, c'est que connaître le logarithme de 3,1623 n'est pas très intéressant. On aimerait plutôt connaître ceux des entiers de 1 à 10.
C'est ici qu'intervient l'anglais Henry Briggs qui publia en 1617 ses premières tables de logarithmes décimaux contenant 1000 valeurs avec quatorze décimales.
Voyons comment il a procédé pour le calcul de $\log(2)$.
Il a commencé par obtenir des valeurs approchées des nombres suivant :

$$\begin{array}{ll} \sqrt{10} = 10^{\frac{1}{2}} & \sqrt{2} = 2^{\frac{1}{2}} \\ \sqrt{\sqrt{10}} = 10^{\frac{1}{2^2}} & \sqrt{\sqrt{2}} = 2^{\frac{1}{2^2}} \\ \sqrt{\sqrt{\sqrt{10}}} = 10^{\frac{1}{2^3}} & \sqrt{\sqrt{\sqrt{2}}} = 2^{\frac{1}{2^3}} \\ \vdots & \vdots \\ \sqrt{\sqrt{\cdots\sqrt{10}}} = 10^{\frac{1}{2^{54}}} & \sqrt{\sqrt{\cdots\sqrt{2}}} = 2^{\frac{1}{2^{54}}} \end{array}$$

Il a finalement abouti à :

$$10^{\frac{1}{2^{54}}} \approx \underbrace{1.000\,000\,000\,000\,000\,127\,819\,149\,320\,032\,35}_{1+a}$$

$$2^{\frac{1}{2^{54}}} \approx \underbrace{1.000\,000\,000\,000\,000\,038\,477\,397\,965\,583\,10}_{1+b}$$

Or on cherche à déterminer $x = \log(2)$ mais $x = \log(2) \Leftrightarrow 10^x = 2$.
D'après les calculs de Briggs :

$$1 + b = 2^{\frac{1}{2^{54}}} = (10^x)^{\frac{1}{2^{54}}} = (1 + a)^x$$

Un résultat classique que nous allons très bientôt étudier montre que $(1 + a)^x \approx 1 + ax$ pour a assez proche de zéro et $x > 0$. On en déduit que $1 + b \approx 1 + ax$ et donc que :

$$\log(2) = x \approx \frac{b}{a} = \frac{3\,847\,739\,796\,558\,310}{12\,781\,914\,932\,003\,235}$$

```
In [7]: 3847739796558310 / 12781914932003235
Out[7]: 0.30102999566398114
```

Ainsi, $\log(2) \approx 0.301\,029\,995\,663\,981\,14$.
Cela nous permet alors d'avoir également $\log(4) = 2 \times \log(2)$ et $\log(8) = 3 \times \log(2)$ mais aussi $\log(5) = \log(10) - \log(2)$.
Il nous faudrait donc obtenir de même $\log(3)$ et $\log(7)$ pour compléter notre collection car $\log(6) = \log(3) + \log(2)$ et $\log(9) = 2 \cdot \log(3)$.
On peut donc à titre d'exercice calculer ces deux logarithmes pour obtenir :

$$\log(3) \approx 0.477\,121\,254\,719\,662\,44 \qquad \log(7) \approx 0.845\,098\,040\,014\,256\,7$$

La route est longue jusqu'aux 1000 logarithmes de Briggs[Roegel, 2010] :

Num. absolu.	Logarithmi.	Num. absolu.	Logarithmi.	Num. absolu.	Logarithmi.
1	0,00000,00000,0000 30102,99956,6398	34	1,53147,89170,4226 1258,91273,0802	67	1,82607,48027,0083 643,41100,0541
2	0,30102,99956,6398 17609,12590,5568	35	1,54406,80443,5028 1223,44564,1701	68	1,83250,89127,0624 634,01780,3102
3	0,47712,12547,1966 12493,87366,0830	36	1,55630,25007,6729 1189,92232,9970	69	1,83884,90907,3726 624,89492,7700
4	0,60205,99913,2796 9691,00130,0806	37	1,56820,17240,6699 1158,18725,4982	70	1,84509,80400,1426 616,03087,0482
5	0,69897,00043,3602 7918,12460,4762	38	1,57978,35966,1681 1128,10104,0969	71	1,85125,83487,1908 607,41477,1219
6	0,77815,12503,8364 6694,67896,3062	39	1,59106,46070,2650 1099,53843,0146	72	1,85733,24964,3127 599,03636,8919
7	0,84509,80400,1426 5799,19469,7768	40	1,60205,99913,2796 1072,38653,9178	73	1,86332,28601,2046 590,88596,1052
8	0,90308,99869,9194 5115,25224,4738	41	1,61278,38567,1974 1046,54336,7816	74	1,86923,17197,3098 582,95436,6072
9	0,95424,25094,3932 4575,74905,6068	42	1,62324,92903,9790 1021,91651,8169	75	1,87506,12633,9170 575,23288,8909

On peut retrouver ces résultats sur Sage qui permet de travailler très simplement avec MPFR :

```
# Travail commode avec la bibliothèque MPFR de C  via sage
# On ouvre dans emacs un fichier d'extension sage qui met emacs en mode sage
# La syntaxe de programmation est celle de Python
#

# On travaille sur 128 bits (on donne la précision = taille de la mantisse)
RN = RealField(113)

# Algorithme de Héron pour calculer ?..
def racine(a,x,n):
    if n == 0:
        return RN(x)
    else:
        return racine(RN(a), ??? * RN(0.5), n - 1)

# quid ?
def racine_n(a,n):
    if n == 0:
        return RN(a)
    else:
        return racine_n(racine(RN(a),1.0,10), n - 1)

# Calcul de log 2
a = ???
b = ???

logDeux = RealField(53)(b / a) # résultat projeté sur 64 bits
diffLog = logDeux - (log(2.))/ log(10.) # On compare avec le log 2 de sage

# Calcul de log 3 et log 7 ? Généralisation ? log 0 ?  log (-10) ?
```

Alors :

```
sage: logDeux
0.301029995663981182
sage: diffLog
5.55111512312578e-17
```

Polynômes interpolateurs

Qu'est-ce qu'un polynôme ?

Les polynômes en informatique sont un outil très puissant et très utile, notamment en théorie de l'information (détection et correction d'erreur). Les entiers en précision infinie sont considérés comme des polynômes. Les exemples abondent mais il s'agit d'une vision discrète de la notion : un polynôme est une liste finie de nombres et on munit l'ensemble des polynômes d'opérations spécifiques.
Nous allons ici étudier plutôt les fonctions polynomiales, i.e. les fonctions de la forme :

$$p \ : \ x \mapsto p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0$$

où les a_i sont des constantes arbitraires qui seront pour nous des nombres à virgule flottante. Si a_n est non nul, on dira que la fonction polynôme est de degré n.
Cette fonction correspond au polynôme :

$$P = [a_n, a_{n-1}, \ldots, a_0]$$

Dans $\mathbb{F}_2$ nous verrons que la distinction entre polynôme et fonction polynomiale est très importante. Dans $\mathbb{R}$ ou $\mathbb{V}$, c'est moins grave mais bon...
L'intérêt ? Il s'agit de fonctions très rapidement évaluables sur machine car faisant intervenir uniquement des additions et des multiplications.

Problème d'interpolation

On connaît quelques valeurs du logarithme donnée par la méthode de Briggs. Cependant, cette méthode est très fastidieuse : il faudrait trouver une autre idée pour calculer les valeurs intermédiaires.

Par exemple, on connaît pour l'instant des approximations des logarithmes de 1, 2, 3 et 4. Comment faire pour déterminer des approximations de tous les nombres entre 1 et 4 avec un pas de 0,25 ?

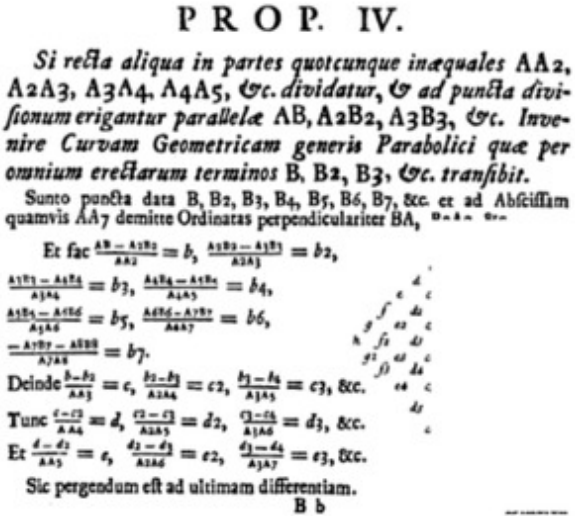
Nous allons chercher un polynôme de degré 3 qui passe par ces quatre points. On cherche donc a, b, c et d tels que :

$$\begin{cases} p(x) = a + bx + cx^2 + dx^3 \\ \texttt{map}\ p\ [1,2,3,4] = \texttt{map}\ \log\ [1,2,3,4] \end{cases}$$

C'est un simple système linéaire de 4 équations à quatre inconnues. Cependant, les équations ont une forme très spéciale. Voyons une première éthode proposée par Newton en 1676.

Interpolation de Newton

Notons $[y_1, y_2, y_3, y_4] = \texttt{map}\ \log\ [1,2,3,4]$.



PROP. IV.

Si recta aliqua in partes quotcunque inæquales AA2, A2A3, A3A4, A4A5, *&c. dividatur, & ad puncta divisionum erigantur parallelæ* AB, A2B2, A3B3, *&c. Invenire Curvam Geometricam generis Parabolici quæ per omnium erectarum terminos* B, B2, B3, *&c. transibit.*

Sunto puncta data B, B2, B3, B4, B5, B6, B7, &c. et ad Abscissam quamvis AA7 demitte Ordinatas perpendiculariter BA, ...

Et fac $\frac{AB - A1B1}{AA1} = b_1$, $\frac{A1B1 - A2B2}{A1A2} = b_2$,
$\frac{A2B2 - A3B3}{A2A3} = b_3$, $\frac{A3B3 - A4B4}{A3A4} = b_4$,
$\frac{A4B4 - A5B5}{A4A5} = b_5$, $\frac{A5B5 - A6B6}{A5A6} = b_6$,
$\frac{A6B6 - A7B7}{A6A7} = b_7$.

Deinde $\frac{b_1 - b_2}{AA3} = c_1$, $\frac{b_2 - b_3}{A1A4} = c_2$, $\frac{b_3 - b_4}{A2A5} = c_3$, &c.

Tunc $\frac{c_1 - c_2}{AA4} = d_1$, $\frac{c_2 - c_3}{A1A5} = d_2$, $\frac{c_3 - c_4}{A2A6} = d_3$, &c.

Et $\frac{d_1 - d_2}{AA5} = e_1$, $\frac{d_2 - d_3}{A1A6} = e_2$, $\frac{d_3 - d_4}{A2A7} = e_3$, &c.

Sic pergendum est ad ultimam differentiam.

B b

Problème d'interpolation dans *Methodus Differentialis* publié par Newton en 1676

Appliquons l'algorithme de Gauß à ce système même si le génie allemand ne naîtra qu'un siècle plus tard :

$x = 1$	$p(x) = a + b + c + d = y_1$
$x = 2$	$p(x) = a + 2b + 4c + 8d = y_2$
$x = 3$	$p(x) = a + 3b + 9c + 27d = y_3$
$x = 4$	$p(x) = a + 4b + 16c + 64d = y_4$

En soustrayant les lignes deux par deux, on fait disparaître a :

$$\begin{aligned} y_2 - y_1 &= b + 3c + 7d = \Delta y_1 \\ y_3 - y_2 &= b + 5c + 19d = \Delta y_2 \\ y_4 - y_3 &= b + 7c + 37d = \Delta y_3 \end{aligned}$$

En soustrayant les lignes deux par deux, on fait disparaître b :

$$\begin{aligned} \Delta y_2 - \Delta y_1 &= 2c + 12d = \Delta^2 y_1 \\ \Delta y_3 - \Delta y_2 &= 2c + 18d = \Delta^2 y_2 \end{aligned}$$

En soustrayant les lignes deux par deux, on fait disparaître c :
$\Delta^2 y_2 - \Delta^2 y_1 = 6d = \Delta^3 y_1$
ce qui donne :
$d = \frac{1}{6}\Delta^3 y_1$
$c = \frac{1}{2}\Delta^2 y_1 - \Delta^3 y_1$
$b = \Delta y_1 - \frac{3}{2}\Delta^2 y_1 + 3\Delta^3 y_1 - \frac{7}{6}\Delta^3 y_1 = \Delta y_1 - \frac{3}{2}\Delta^2 y_1 + \frac{11}{6}\Delta^3 y_1$
$a = y_1 - \Delta y_1 + \frac{3}{2}\Delta^2 y_1 - \frac{11}{6}\Delta^3 y_1 - \frac{1}{2}\Delta^2 y_1 + \Delta^3 y_1 - \frac{1}{6}\Delta^3 y_1 = y_1 - \Delta y_1 + \Delta^2 y_1 - \Delta^3 y_1$
Finalement :

$$P(x) = y_1 + (x-1)\Delta y_1 + \frac{1}{2}(x-1)(x-2)\Delta^2 y_1 + \frac{1}{6}(x-1)(x-2)(x-3)\Delta^3 y_1$$

NEWTON avait l'habitude de réunir les résultats dans un schéma de cette forme :

$$\begin{array}{lllll}
\boldsymbol{y_1} & & & & \\
 & \boldsymbol{\Delta y_1} & & & \\
y_2 & & \boldsymbol{\Delta^2 y_1} & & \\
 & \Delta y_2 & & \boldsymbol{\Delta^3 y_1} & \\
y_3 & & \Delta^2 y_2 & & \\
 & \Delta y_3 & & & \\
y_4 & & & &
\end{array}$$

avec $\Delta y_i = y_{i+1} - y_i$, $\Delta^2 y_i = \Delta y_{i+1} - \Delta y_i$ et $\Delta^3 y_i = \Delta^2 y_{i+1} - \Delta^2 y_i$.
On peut généraliser à des abscisses successives de 1 à n :

Théorème 2 - 5

Interpolation de Newton passant par les points $(1, y_1)$, $(2, y_2)$,..., (n, y_n)

$$p(x) = y_1 + (x-1)\Delta y_1 + \frac{1}{2!}(x-1)(x-2)\Delta^2 y_1 + \cdots + \frac{1}{(n-1)!}(x-1)\cdots(x-(n-1))\Delta^{n-1} y_1$$

voire à d'autres valeurs d'entiers successifs :

Théorème 2 - 6

Interpolation de Newton passant par les points (α, y_1), $(\alpha+1, y_2)$,..., $(\alpha+n-1, y_n)$

$$p(x) = y_1 + (x-\alpha)\Delta y_1 + \frac{1}{2!}(x-\alpha)(x-\alpha-1)\Delta^2 y_1 + \cdots + \frac{1}{(n-1)!}(x-\alpha)\cdots(x-\alpha-(n-2))\Delta^{n-1} y_1$$

On peut généraliser à des suites quelconques d'abscisses mais c'est un peu plus compliqué.

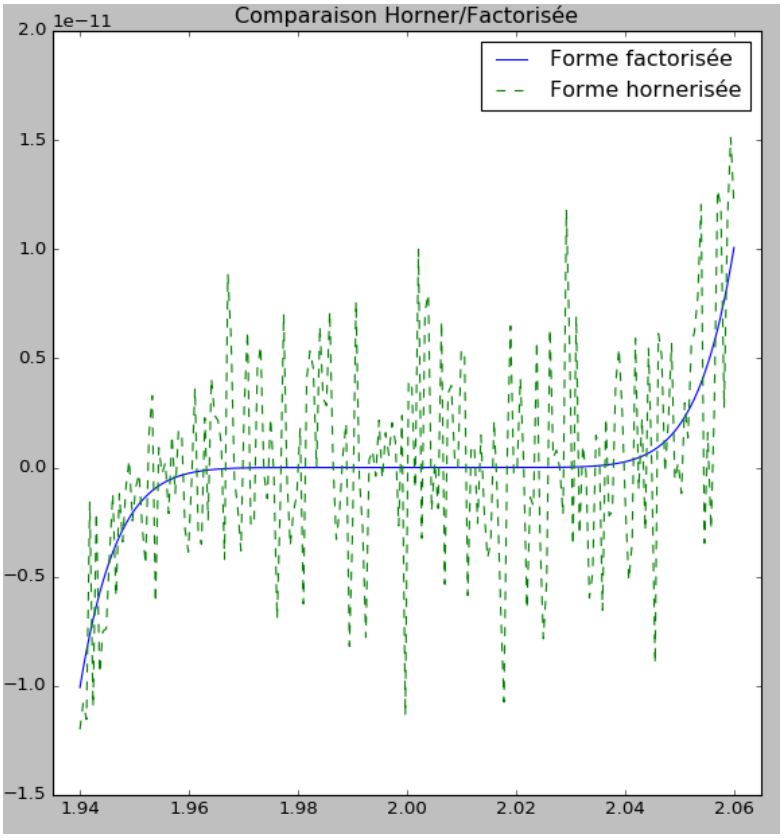
3 Polynômes et erreurs

Nous allons étudier à la Recherche 2 - 8 page 67 un moyen d'évaluer une fonction polynomiale en un nombre particulier.
Cependant, nous allons être à nouveau confrontés au problème des erreurs suite au calcul sur les nombres à virgule flottante.
Considérons par exemple la fonction $x \mapsto (x-2)^9$ au voisinage de 2.
Une petite illustration vaut mieux qu'un long discours...Avec le code suivant :

```
1  def compareFactHorn():
2      pasc = [a*b for (a,b) in zip(pascal(9), [(-2)**k for k in range(9,-1,-1)])]
3      X  = np.linspace(1.94, 2.06, 200)
4      facto = lambda x : (x - 2)**9
5      horno = lambda x : horn(pasc,x)
6      plt.plot(X, facto(X) , label = 'Forme factorisée')
7      plt.plot(X, horno(X) , label = 'Forme hornerisée',ls='--')
8      plt.axis([1.935, 2.065, -1.5e-11, 2e-11])
```

```
9     plt.legend()
10    plt.title('Comparaison Horner/Factorisée')
11    plt.show()
```

On obtient :



Bon, évidemment, on est allé chercher l'exemple qui fâche. Ailleurs, les choses marchent bien mieux.

Il existe un algorithme de Horner compensé comme il en existe un pour la somme mais c'est un peu plus compliqué. Les curieux pourront se référer à un article de Philippe Langlois et Nicolas Louvet [Langlois et Louvet, 2008].

À retenir

Comme nous ne sommes qu'en premier cycle, nous mettrons de côté à partir de maintenant ces problèmes d'erreur dans les évaluations.
Il faudra tout de même retenir que ces problèmes existent et sont omniprésents dans la machine : est-ce que mon joueur est sous le feu de l'ennemi ? Je dois savoir si, en 3D, il est à gauche ou à droite d'une zone ce qui revient sur machine à calculer un déterminant dépendant de 3 paramètres, bref évaluer un polynôme de trois variables. Si je me trompe dans mes calculs, quand je suis prêt de la zone frontière, mon joueur peut mourir dans d'atroces souffrances (sur mon écran seulement, bien sûr).

4 Série de Taylor

Dérivées d'ordre supérieur

Nous avons eu une approche intuitive de la dérivation jusqu'à maintenant. Elle nous suffira la plupart du temps. Cependant il ne faut pas oublier qu'un manque de rigueur dans l'étude de ces notions peut entraîner de graves erreurs.
Nous rappelons donc quelques définitions qui nous servirons à mieux comprendre certains problèmes.

Définition 2 - 1

Dérivabilité
Soit f une fonction numérique définie sur un intervalle I de $\mathbb{R}$.
Soit $a \in I$. On dit que f est dérivable en a si, et seulement si, le rapport :

$$\tau_a(h) = \frac{f(a+h) - f(a)}{h}$$

admet une limite finie lorsque a tend vers 0.
Dans ce cas on note $f'(a)$ cette limite.

Rappelons que $f'(a)$ est aussi le coefficient directeur de la tangente à la courbe représentative de f au point d'abscisse a.

Définition 2 - 2

Dérivée $k^{ème}$
Si f' est dérivable sur I on note f'' ou $f^{(2)}$ la dérivée de f' et on l'appelle la dérivée seconde de f. De même si f'' est dérivable sur I, f''' ou $f^{(3)}$, la dérivée de f'', est la dérivée troisième de f. Plus généralement on note $f^{(k)}$ la dérivée $k^{ième}$ de f avec la convention $f^{(0)} = f$, $f^{(k+1)} = \left(f^{(k)}\right)'$.

Définition 2 - 3

Fonctions de classe C^n, de classe C^∞
On dit que f est de classe C^n, $n \in \mathbb{N}$, sur I ssi f est n fois dérivable et ses n dérivées sont continues sur I. On remarquera que si $f^{(n+1)}$ existe sur I alors f est de classe C^n sur I qui s'écrit $f \in C^n(I)$. On dit que f est de classe C^∞ sur I pour exprimer que f est indéfiniment dérivable et, a fortiori, que toutes ses dérivées sont continues.

Relations de domination

> *Brooks's Law [prov.]*
> *« Adding manpower to a late software project makes it later » – a result of the fact that the expected advantage from splitting work among N programmers is $O(N)$, but the complexity and communications cost associated with coordinating and then merging their work is $O(N^2)$*
>
> in « The New Hacker's Dictionnary »
> http://outpost9.com/reference/jargon/jargon_17.html#SEC24

Les notations de Landau(1877-1938) ont en fait été crées par Paul Bachmann(1837-1920) en 1894, mais bon, ce sont tous deux des mathématiciens allemands.
Par exemple, si l'on considère l'expression :

$$f(n) = n + 1 + \frac{1}{n} + \frac{75}{n^2} - \frac{765}{n^3} + \frac{\cos(12)}{n^{37}} - \frac{\sqrt{765\,481}}{n^{412}}$$

Quand n est « grand », disons 10 000, alors on obtient :

$$f(10\,000) = 10\,000 + 1 + 0,0001 + 0,00000000075 - 0,0000000000000765 + \text{ peanuts}$$

Tous les termes après n comptent pour du beurre quand n est « grand » : on a expérimenté ceci sur machine avec la notion de *sous-capacité*.

Donnons une définition pour plus de clarté :

Définition 2 - 4

« Grand O »
Soit f et g deux fonctions de $\mathbb{N}$ dans $\mathbb{R}$. On dit que f est un « grand O » de g et on note $f = O(g)$ ou $f(n) = O(g(n))$ si, et seulement si, il existe une constante strictement positive C telle que

$$|f(n)| \leqslant C|g(n)|$$

pour tout $n \in \mathbb{N}$.

Dans l'exemple précédent, $\frac{1}{n} \leqslant \frac{1}{1} \times 1$ pour tout entier n supérieur à 1 donc $\frac{1}{n} = O(1)$.
De même, $\frac{75}{n^2} \leqslant \frac{75}{1^2} \times 1$ donc $\frac{75}{n^2} = O(1)$ mais on peut dire mieux : $\frac{75}{n^2} \leqslant \frac{75}{1} \times \frac{1}{n}$ et ainsi on prouve que $\frac{75}{n^2} = O\left(\frac{1}{n}\right)$.
En fait, un grand O de g est une fonction qui est au maximum majorée par un multiple de g.
Point de vue algorithmique, cela nous donne un renseignement sur la complexité au pire.
Pour le tri fusion, on obtient pour $n > 1$:

$$K(n) \leqslant 4nc\left(\lfloor \log_2(n) \rfloor + 1\right) \leqslant 4nc \log_2(n)\left(1 + \frac{1}{\log_2(n)}\right) \leqslant 8cn \log_2(n)$$

On en déduit que $K(n) = O(n \log_2(n))$: la complexité est au pire en $n \log_2(n)$, c'est ce qui nous importe. Le rôle des constantes est accessoire car chercher trop de précisions serait illusoire : l'« oubli » de ces constantes correspond en fait au différences entre langages, processeurs, etc.
On peut cependant faire mieux avec le tri fusion car on a aussi une minoration.
C'est le moment d'introduire une nouvelle définition :

Définition 2 - 5

« Grand Oméga »
Soit f et g deux fonctions de $\mathbb{R}$ dans lui-même. On dit que f est un « grand Oméga » de g et on note $f = \Omega(g)$ ou $f(n) = \Omega(g(n))$ si, et seulement si, il existe une constante strictement positive C telle que

$$|f(n)| \geqslant C|g(n)|$$

pour tout $n \in \mathbb{N}^*$.

Remarque

Comme Ω est une lettre grecque, on peut, par esprit d'unification, parler de « grand omicron » au lieu de « grand O »...

Remarque

$f = \Omega(g) \iff g = O(f)$...

On montre donc facilement pour le tri fusion que $K(n) = \Omega(n \log_2(n))$ grâce à l'inégalité $2cn\lfloor \log_2(n) \rfloor \leqslant K(n)$.
Ainsi on a dans ce cas en même temps $f = O(n \log_2(n))$ et $f = \Omega(n \log_2(n))$: c'est encore plus précis et nous incite à introduire une nouvelle définition :

Définition 2 - 6

« Grand Théta »

$$f = \Theta(g) \iff \begin{cases} f = O(g) \\ f = \Omega(g) \end{cases}$$

Le coût de l'algorithme se trouve donc coincé entre deux valeurs de même ordre. On peut ainsi dire que la complexité du tri fusion est en $n \log_2(n)$ ce qui est *souvent* mieux que le tri par insertion dont la complexité *au pire* est en n^2. Mais attention ! *Au pire* ne signifie pas *toujours* : si la liste est déjà triée, le tri par insertion est plus économique.
Voici maintenant une petite table pour illustrer les différentes classes de complexité rencontrées habituellement :

coût \ n	100	1000	10^6	10^9
$\log_2(n)$	≈ 7	≈ 10	≈ 20	≈ 30
$n\log_2(n)$	≈ 665	$\approx 10\,000$	$\approx 2\cdot 10^7$	$\approx 3\cdot 10^{10}$
n^2	10^4	10^6	10^{12}	10^{18}
n^3	10^6	10^9	10^{18}	10^{27}
2^n	$\approx 10^{30}$	$> 10^{300}$	$> 10^{10^5}$	$> 10^{10^8}$

Gardez en tête que l'âge de l'Univers est environ de 10^{18} secondes...
Il existe également deux autres « comparateurs » que l'on utilise peu en algorithmique mais qui peuvent s'avérer utiles dans d'autres domaines.

Définition 2 - 7

« Petit o »
$f = o(g)$ si, et seulement si, pour toute constante positive ε, il existe un entier n_0 tel que, pour tout $n \geqslant n_0$,

$$|f(n)| \leqslant \varepsilon\,|g(n)|$$

On dit alors souvent que f est *négligeable* devant g.
Contrairement au grand O, la majoration doit se faire quelque soit la constante ε et non pas seulement pour une constante arbitraire.

Définition 2 - 8

Fonctions équivalentes
$f(n) \sim g(n) \iff f(n) = g(n) + o(g(n))$

Voici quelques propriétés fort utiles que nous démontrerons à l'occasion :

Propriétés 2 - 1

Manipulation des O

- $O(f) + O(g) = O(f+g)$;
- $f = O(f)$;
- $k \cdot O(f) = O(f)$ si k est une constante ;
- $O(O(f)) = O(f)$;
- $O(f)\cdot O(g) = O(f\cdot g)$;
- $O(f\cdot g) = f\cdot O(g)$.

Par exemple, $2n^3 + 3n^2 + 5n = O(n^3) + O(n^3) + O(n^3) = O(n^3)$.

Polynôme de Taylor

Brook Taylor
(1685-1731)

Brook Taylor fut un savant anglais qui reprit les travaux de ses aînés Newton, Leibniz, Jacques Bernoulli et Gregory.
Musicien, philosophe, peintre, il est surtout célèbre pour les formules qui portent son nom. Il les obtint en étudiant le problème de Kepler qui concerne le calcul de certaines anomalies dans la révolution des planètes.
Son but est encore une fois d'obtenir une interpolation d'une fonction compliquée à l'aide d'un polynôme simple.
Il obtient le résultat suivant :

Théorème 2 - 7

Polynôme de Taylor
Soit I un intervalle, soit f une fonction de Ivers $\mathbb{R}$ de classe C^n avec $n \in \mathbb{N}$ et soit a un élément de I. Il existe alors un *unique* polynôme $T_{n,a}$ de degré au plus n dont les dérivées jusqu'à l'ordre n coïncident en a avec celles de f. De plus :

$$T_{n,a}(x) = f(a) + (x-a)f'(a) + \frac{(x-a)^2}{2!}f''(a) + \frac{(x-a)^3}{3!}f^{(3)}(a) + \cdots + \frac{(x-a)^n}{n!}f^{(n)}(a)$$

Nous admettrons l'unicité mais *vous vérifierez* tout de même que pour tout entier $0 \leqslant k \leqslant n$,

$$T_{n,a}^{(k)}(a) = f^{(k)}(a)$$

Par exemple, la fonction exponentielle admet comme polynôme de TAYLOR à l'ordre 2 en 0 :

$$\begin{aligned} T_{n,0}(x) &= \exp(0) + (x-0)\exp'(0) + \frac{(x-0)^2}{2!}\exp''(0) \\ &= 1 + x + \frac{x^2}{2} \end{aligned}$$

Le problème principal de sa formule est d'estimer l'erreur commise en considérant $T_{n,a}$ à la place de f :

$$f(x) = T_{n,a}(x) + R_{n,a}(x)$$

Voilà pourquoi on retrouve son nom associé à trois formules qui donnent une meilleure idée de ce reste (*quelle erreur commet-on ?*) et qui ont été établies par trois de ses collègues.
Nous ne ferons pas les preuves qui sont assez techniques.
Nous admettrons également que le polynôme de TAYLOR approche *mieux*[a] f au voisinage de a que tous les autres polynômes de degré au plus n.

Formule de Taylor-Lagrange

Théorème 2 - 8

Si f est de classe C^n sur $[a,x]$ et si $f^{(n+1)}$ existe sur $]a,x[$ alors il existe au moins un réel $c \in]a,x[$ tel que :

$$f(x) = T_{n,a}(x) + \frac{(x-a)^{n+1}}{(n+1)!} f^{(n+1)}(c)$$

On dit que l'on a écrit la formule de Taylor Lagrange à l'ordre n.

Il est souvent pratique de réécrire cette formule en posant $x = a + h$. Le nombre c appartient donc à l'intervalle $]a, a+h[$. Il existe donc un réel $\theta \in]0,1[$ tel que :

$$f(a+h) = \sum_{k=0}^{n} \frac{h^k}{k!} f^{(k)}(a) + \frac{h^{n+1}}{(n+1)!} f^{(n+1)}(a+\theta h)$$

Formule de Taylor-Mac Laurin

Dans la formule de Taylor-Lagrange on remplace a par 0. On obtient :

Théorème 2 - 9

$$f(x) = \sum_{k=0}^{n} \frac{x^k}{k!} f^{(k)}(0) + \frac{x^{n+1}}{(n+1)!} f^{(n+1)}(\theta x)$$

Formule de Taylor avec reste intégral

Si f est de classe C^{n+1} sur I contenant a :

Théorème 2 - 10

$$\forall x \in I, f(x) = T_{n,a}(x) + \int_a^x \frac{(x-t)^n}{n!} f^{(n+1)}(t)dt$$

Formule de Taylor-Young.

Si f est de classe C^n sur un intervalle I contenant a alors :

Théorème 2 - 11

$$\forall x \in I, f(x) = T_{n,a}(x) + o\left((x-a)^n\right)$$

a. Ce qui signifie que $|f(x) - T_{n,a}(x)| \leqslant |f(x) - P(x)|$ pour tout polynôme P de degré au plus n.

5 Développements limités

Voisinage

$a \in \mathbb{R}$, nous appellerons voisinage de a toute partie de $\mathbb{R}$ contenant un intervalle ouvert contenant a. L'intervalle $]a-\alpha, a+\alpha[$, avec $\alpha \in \mathbb{R}^{+*}$, est un voisinage de a, nous le noterons V_a et $V_a - \{a\}$ sera noté V_a^*. Un voisinage à droite de a est un intervalle du type $]a, a+\alpha[$ et $]a-\alpha, a[$ est un voisinage à gauche de a. Dans ce qui suit nous ne distinguerons pas ces types de voisinages, en conséquence V_a^* pourra tout aussi bien aussi désigner un voisinage à droite ou à gauche, lors des applications, le contexte permettra de les distinguer. Un voisinage de $+\infty$ est un voisinage du type $[A, +\infty[$ avec A réel positif aussi grand que l'on veut que nous pourrons noter $V_{+\infty}^*$. Notez que $]-\infty; -A]$ est un voisinage de $-\infty$.

Définition

Définition 2 - 9

On dit que f, définie sur V_a, admet un développement limité d'ordre $n \in \mathbb{N}$ au voisinage de a (on écrit en abrégé : « f admet un $DL_n V(a)$ ») s'il existe un polynôme $P_n \in \mathbb{R}_n[X]$ (ensemble des polynômes de degré inférieur ou égal à n) et une fonction ε vérifiant :

$$\begin{cases} \forall x \in V_a, f(x) = P_n(x-a) + (x-a)^n \varepsilon(x) \\ \varepsilon(a) = 0 \\ \lim\limits_{x \to a} \varepsilon(x) = 0 \end{cases}$$

c'est-à-dire $f(x) = P_n(x-a) + o\Big((x-a)^n\Big)$.

Il faut bien remarquer que :

$$q > p \to (x-a)^q \underset{a}{=} o\Big((x-a)^p\Big)$$

Plus la puissance p est grande, plus $(x-a)^p$ est petit.
Nous n'irons pas très loin dans l'étude des développements limités mais elle nous permettra d'avoir des approximations polynomiales de nombreuses fonctions ce qui nous permettra de comprendre de nombreuses méthodes employées en informatique.

6 Exemples de résolutions numériques d'équas diff

Ce que l'on dirait si l'on faisait des maths à l'IUT

Soit U un ouvert de $\mathbb{R} \times \mathbb{R}^n$ et $f : U \longrightarrow \mathbb{R}^n$ une application continue sur l'ouvert U.
Alors pour $(t_0, x_0) \in U$, on appelle solution du problème de Cauchy :

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

tout couple (I, x) où I est un intervalle contenant t_0 et $x : I \longrightarrow \mathbb{R}^n$ une solution sur I de l'équation différentielle $x'(t) = f(t, x(t))$ telle que $x(t_0) = x_0$.
L'équation différentielle qui apparaît dans (C) sera appelée équation différentielle scalaire si $n = 1$, et système différentiel sinon.
Si l'application $f : U \longrightarrow \mathbb{R}^n$ est de classe C^1 sur l'ouvert U, alors, d'après le théorème de Cauchy-Lipschitz, pour $(t_0, x_0) \in U$, il existe une unique solution maximale au problème de Cauchy (C).
Dans la plupart des cas, on ne sait pas résoudre explicitement le problème de Cauchy (C) ; d'où la nécessité de mettre au point des méthodes numériques de résolution approchée d'un tel problème.

Méthode d'Euler

La méthode d'Euler est la méthode la plus simple pour résoudre numériquement une équation différentielle. Son intérêt pratique est limité par sa faible précision, c'est pourquoi elle est largement utilisée :-)
L'idée d'Euler consiste à utiliser l'approximation :

$$x'(t) \approx \frac{x(t+h)-x(t)}{h} \quad \text{pour } h \text{ petit}$$

Autrement dit,

$$x(t+h) \approx x(t) + h \cdot x'(t) = x(t) + h \cdot f(t, x(t))$$

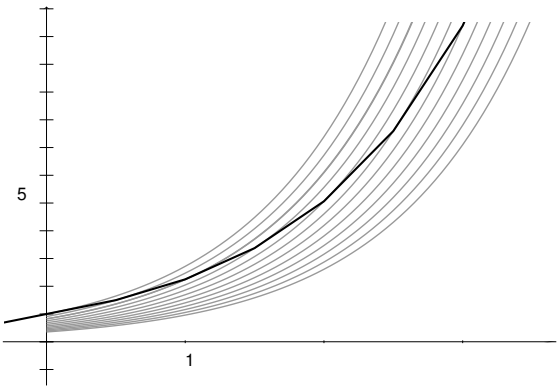
Partant du point (t_0, x_0), on suit alors la droite de pente $f(t_0, x_0)$ sur l'intervalle de temps $[t_0, t_0 + h]$.
On pose alors :

$$\begin{cases} t_1 = t_0 + h \\ x_1 = x_0 + h \cdot f(t_0, x_0) \end{cases}$$

De nouveau, partant du point (t_1, x_1), on suit alors la droite de pente $f(t_1, x_1)$ sur l'intervalle de temps $[t_1, t_1 + h]$, et ainsi de suite.
On construit ainsi une suite de points de la manière suivante :

$$\forall k \in [\![0, N]\!], \quad \begin{cases} t_{k+1} = t_k + h \\ x_{k+1} = x_k + h \cdot f(t_k, x_k) \end{cases}$$

La ligne brisée joignant les points $\{(t_k, x_k) | k \in [\![0, N]\!]\}$ donnera une solution approchée de notre équation différentielle.
Il faut bien comprendre qu'à chaque étape, on repart dans la direction d'une solution exacte de l'équation différentielle, mais qui n'est pas celle qui est solution du problème de Cauchy initial.
Sur le graphique ci-contre, on trace une solution approchée pour la condition initiale $x(0) = 1$ et les courbes intégrales de notre équation qui passent par les points (t_k, x_k).



Pour juger de la qualité d'une méthode (ou schéma) numérique de résolution d'équations différentielles, il faut prendre en compte plusieurs critères.

— L'erreur de consistance donne l'ordre de grandeur de l'erreur effectué à chaque pas. Par exemple, dans le cas de la méthode d'Euler, l'erreur de consistance mesure l'erreur qu'entraîne le fait d'approcher le nombre dérivé par un taux d'accroissement. La sommation des erreurs de consistance à chaque pas donnera l'ordre de grandeur de l'erreur globale (sous des hypothèses de régularité pour la fonction f). À l'aide de la formule de Taylor-Lagrange, on peut montrer que dans le cas de la méthode d'Euler, l'erreur de consistance est dominée par h^2 ; on dit alors que cette méthode est d'ordre 1. Ainsi, d'un point de vue théorique, plus le pas est petit, meilleure sera l'approximation.

Un schéma numérique est dit consistant si la somme des erreurs de consistance tend vers 0 quand le pas h tend vers 0 ; ce qui est le cas de la méthode d'Euler (en gros, car $Nh^2 \sim h \to 0$ où N désigne le nombre d'itérations).

— La stabilité contrôle la différence entre deux solutions approchées correspondant à deux conditions initiales voisines : un schéma est dit stable si un petit écart entre les conditions initiales $x(t_0) = x_0$ et $\tilde{x}(t_0) = x_0 + \varepsilon$ et de petites erreurs d'arrondi dans le calcul récurrent des $\tilde{x}_k$ provoquent une erreur finale $|x_k - \tilde{x}_k|$ contrôlable. La méthode d'Euler est une méthode stable.

— Un schéma est dit convergent lorsque l'erreur globale (qui est le maximum des écarts entre la solution exacte et la solution approchée) tend vers 0 lorsque le pas tend vers 0. En fait, pour que le schéma soit convergent, il suffit que la méthode soit consistante et stable.

— Enfin pour la mise en application du schéma, il faut aussi prendre en compte l'influence des erreurs d'arrondi. En effet, afin de minimiser l'erreur globale théorique, on pourrait être tenté d'appliquer la méthode d'Euler avec un pas très petit, par exemple de l'ordre de 10^{-16}, mais ce faisant, outre que le temps de calcul deviendrait irréaliste, les erreurs d'arrondi feraient diverger la solution approchée très rapidement, puisque pour des flottants Python de l'ordre de 10^{-16}, les calculs ne sont plus du tout exacts !

  En pratique, il faut prendre h assez petit (pour que la méthode converge assez rapidement du point de vue théorique), mais pas trop petit non plus (pour que les erreurs d'arrondis ne donnent pas lieu à des résultats incohérents, et que les calculs puissent être effectués en un temps fini). La question de trouver de manière théorique le pas optimal peut s'avérer un problème épineux. Dans un premier temps, on peut se contenter de faire des tests pratiques comme dans l'exemple suivant.

Dans le script suivant, on applique la méthode d'Euler sur l'intervalle $[0, 1]$ avec la condition initiale $x(0) = 1$.

```
1  from math import exp
2
3
4  print("{:^10} | {:^12} | {:^12} | {:^13}".format('h','x(t)','exp(t)','erreur'))
5  print('-'*57)
6  for i in range(1, 8):
7      h, t, x  = 10**(-i), 0, 1
8      while t < 1:
9          t, x = t + h, x * (h + 1)
10     print("{0:>10g} | {1:>.10f} | {2:>.10f} | {3:> .10f}"
11             .format(h, x, exp(t),  exp(t) - x))
```

```
1     h      |     x(t)     |    exp(t)    |    erreur
2  ---------------------------------------------------------
3        0.1 | 2.8531167061 | 3.0041660239 |  0.1510493178
4       0.01 | 2.7048138294 | 2.7182818285 |  0.0134679990
5      0.001 | 2.7169239322 | 2.7182818285 |  0.0013578962
6     0.0001 | 2.7184177414 | 2.7185536702 |  0.0001359288
7      1e-05 | 2.7182954199 | 2.7183090114 |  0.0000135915
8      1e-06 | 2.7182804691 | 2.7182818285 |  0.0000013594
9      1e-07 | 2.7182819660 | 2.7182820996 |  0.0000001336
10     1e-08 | 2.7182817983 | 2.7182818347 |  0.0000000363
11     1e-09 | 2.7182820738 | 2.7182818299 | -0.0000002438
```

Pour un pas $h \lesssim 10^{-9}$, le temps de calcul devient trop important, de plus, on voit poindre l'effet des erreurs d'arrondis, puisque la solution approchée est devenu supérieure à l'exponentielle, ce qui est mathématiquement faux, en vertu de l'inégalité :

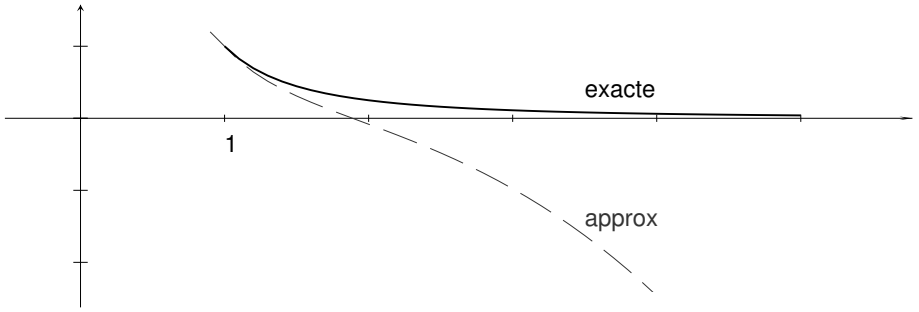
$$\forall n \in \mathbb{N}, \quad \left(1 + \frac{1}{n}\right)^n \leq e$$

En général, il ne suffit pas qu'un schéma numérique soit convergent pour qu'il donne de bons résultats sur n'importe quelle équation différentielle. Encore faut-il que le problème soit mathématiquement bien posé (en particulier, les hypothèses du théorème de Cauchy-Lipschitz doivent être vérifiées), qu'il soit numériquement bien posé (continuité suffisamment bonne par rapport aux conditions initiales) et qu'il soit bien conditionné (temps de calcul raisonnable).

Considérons par exemple le problème de CAUCHY :

$$\begin{cases} x(1) = 1 \\ x'(t) = 3\frac{x(t)}{t} - \frac{5}{t^3} \end{cases}$$

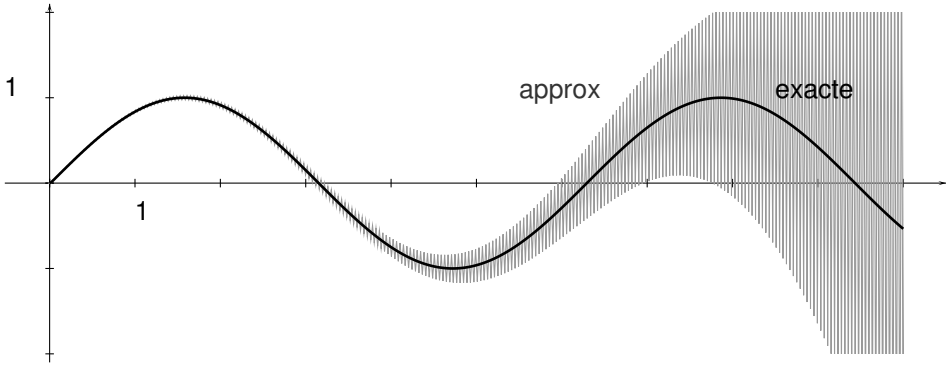
dont la solution est la fonction ? ? ?



On constate qu'ici la solution approchée s'écarte assez rapidement de la solution exacte φ. En effet, la forme générale des solutions de l'équation différentielle précédente est donnée par $t \mapsto \lambda x^3 + 1/x^2$. La condition $x(1) = 1$ impose $\lambda = 0$, mais dès qu'on s'écarte de la solution φ on suit « tangentiellement » des courbes intégrales qui comportent un terme en λx^3 et donc qui diffère notablement de la solution. Donc le problème est mal posé. Ici le schéma numérique n'est pas en cause.
Soit à résoudre à présent le problème de CAUCHY :

$$\begin{cases} x(1) = 1 \\ x'(t) = 100\,(\sin(t) - x) \end{cases}$$

dont la solution est la fonction $\varphi : t \mapsto \frac{1}{10001}(-100\cos t + 10000\sin t + 100\exp(-100\cdot t)) \approx \sin t$. Que se passe-t-il si on résout ce problème avec un pas de l'ordre de 0.02 ?



Avec un tel pas, la solution approchée oscille en s'éloignant de plus en plus de la solution exacte. En effet, le schéma numérique est donné par :

$$x_{k+1} = x_k + 100\,h\,(\sin t_k - x_k) = (1 - 100h)x_k + 100h\sin t_k$$

Donc une erreur de ε_k sur x_k aura les répercussions suivantes sur le calcul des termes ultérieurs :

$$\varepsilon_{k+1} = (1 - 100h)\varepsilon_k \implies \varepsilon_{k+n} = (1 - 100h)^n \varepsilon_k$$

Ainsi donc, tant que $1 - 100h \geq -1$, c'est-à-dire tant que $h \lesssim 0.002$, une petite erreur sur l'un des termes aura des répercussions allant en s'amplifiant.
Bien que le problème soit numériquement bien posé, il est en fait mal conditionné.

Méthode de Runge-Kutta d'ordre 4

Si, comme nous l'avons déjà dit, la méthode d'EULER présente un intérêt théorique, on préfère en pratique des méthodes d'ordre plus élevé. Parmi la multitude des schémas numériques (méthodes

à un pas comme celle de TAYLOR, méthodes à pas multiples comme celles d'ADAMS-BASHFORTH, d'ADAMS-MOULTON, de prédiction-correction) celle qui présente le meilleur rapport précision/-complexité est certainement celle de RUNGE-KUTTA d'ordre 4.

En voici le schéma pour $\forall k \in [\![0, N]\!]$,

$$\begin{cases} t_{k+1} &= t_k + h \\ x_{k+1} &= x_k + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{cases} \quad \text{où} \quad \begin{cases} k_1 &= h \cdot f(t_n, x_n) \\ k_2 &= h \cdot f\left(t_n + \frac{h}{2}, x_n + \frac{k_1}{2}\right) \\ k_3 &= h \cdot f\left(t_n + \frac{h}{2}, x_n + \frac{k_2}{2}\right) \\ k_4 &= h \cdot f(t_n + h, x_n + k_3) \end{cases}$$

Les coefficients qui apparaissent dans ces formules mystérieuses sont judicieusement ajustés pour obtenir une méthode d'ordre 4, sans pour autant avoir à calculer les dérivées successives de f (comme c'est le cas dans les méthodes de TAYLOR), ou à recourir à une formule de récurrence d'ordre au moins 2 pour définir x_k (comme c'est le cas dans les méthodes à pas multiples).

Définissez une fonction adaptée à ce schéma. Vous devriez observer pour l'exponentielle

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

```
    h      |   erreur euler    |    erreur rk4
-----------------------------------------------------
       0.1 |  1.5104931784e-01 |  2.5338874514e-06
      0.01 |  1.3467999038e-02 |  2.2464341498e-10
     0.001 |  1.3578962232e-03 |  2.2204460493e-14
    0.0001 |  1.3592881559e-04 | -2.6645352591e-13
     1e-05 |  1.3591551171e-05 | -5.2180482157e-12
     1e-06 |  1.3591611072e-06 |  2.1464163780e-11
```

On constate que pour $h = 0.001$, la méthode d'EULER ne donne que trois décimales significatives du nombre e, alors que la méthode de RUNGE-KUTTA en donne quatorze ! On remarque également qu'il est inutile d'appliquer la méthode de RUNGE-KUTTA avec un pas $h \gtrsim 0.001$, puisque dans ce cas les erreurs d'arrondi prennent le dessus par rapport au gain théorique de précision.

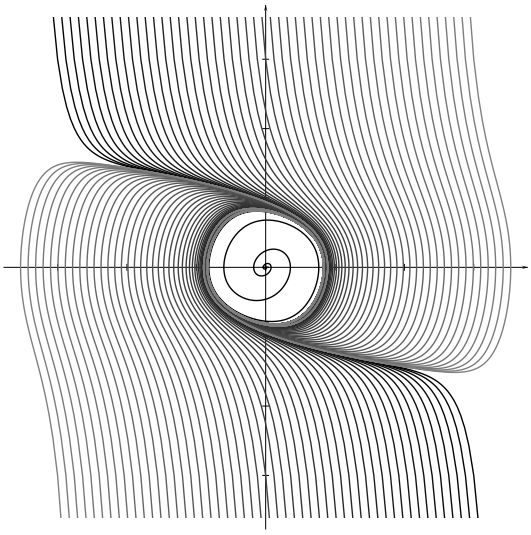
Système

Soit à représenter les courbes intégrales du système autonome suivant, appelé oscillateur de van der Pol :

$$\begin{cases} \frac{\mathrm{d}x}{\mathrm{d}t} = y \\ \frac{\mathrm{d}y}{\mathrm{d}t} = \frac{y}{2} - x - y^3 \end{cases}$$

Les variables x, $k1$,... $k4$ représentent à présent des listes. Or une instruction comme $x + k1/2$ renverra un message d'erreur. En effet, la division d'une liste par un entier n'est pas définie ; de plus, l'addition de deux listes effectue la concaténation des listes, et non l'addition des listes composante par composante.

Il faut alors travailler non plus avec des listes, mais avec des objets supportant la vectorisation, comme c'est le cas des tableaux de la librairie de tierce partie **NumPy** ; en outre, cette option a l'avantage de réduire de manière significative les temps de calcul.



Un exemple d'équation différentielle linéaire d'ordre 2

On rappelle qu'une équation différentielle linéaire d'ordre n peut toujours s'écrire comme un système différentiel d'ordre 1. Illustrons ceci dans le cas $n = 2$: en notant :

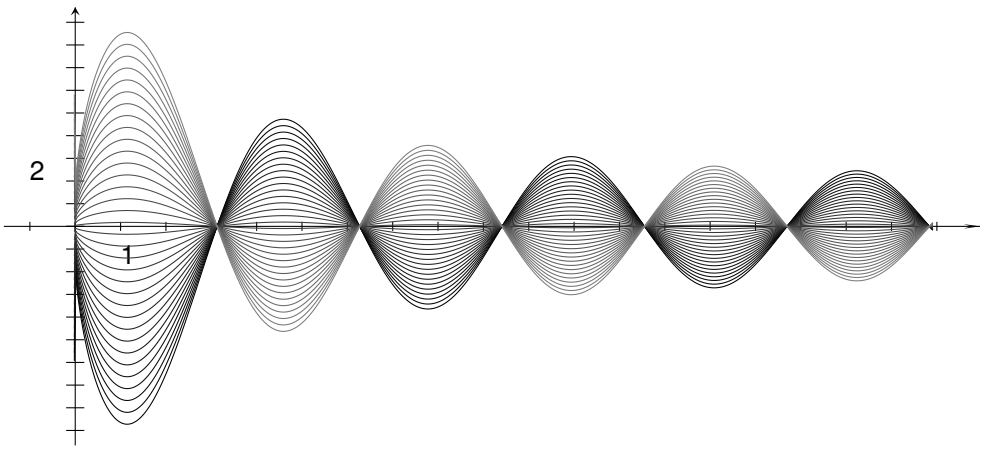
$$A(t) = \begin{pmatrix} 0 & 1 \\ -b(t) & -a(t) \end{pmatrix}, \quad X(t) = \begin{pmatrix} x(t) \\ x'(t) \end{pmatrix}, \quad C(t) = \begin{pmatrix} 0 \\ c(t) \end{pmatrix}$$

on a les équivalences

$$\begin{aligned} \forall t \in I \quad x'' + a(t)x' + b(t)x = c(t) &\iff \forall t \in I \quad \begin{cases} x' = x' \\ x'' = -bx - ax' + c \end{cases} \\ &\iff \forall t \in I \quad \frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} x \\ x' \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -b & -a \end{pmatrix} \cdot \begin{pmatrix} x \\ x' \end{pmatrix} + \begin{pmatrix} 0 \\ c \end{pmatrix} \\ &\iff \forall t \in I \quad X' = A(t) \cdot X + C(t) \end{aligned}$$

Considérons, par exemple, l'équation différentielle du 2e ordre dite de Bessel :

$$t^2x''(t) + tx'(t) + \left(t^2 - \alpha^2\right)x(t) = 0 \iff X'(t) = A(t) \cdot X(t) \quad \text{avec} \quad A(t) = \begin{pmatrix} 0 & 1 \\ \frac{\alpha^2 - t^2}{t^2} & -\frac{1}{t} \end{pmatrix}$$



7 Approximation de Pi et calcul approché d'intégrale au petit bonheur

On s'intéresse à l'intégrale sur $[0,1]$ de la fonction $f : t \mapsto \sqrt{1-t^2}$: quel est son rapport avec π ?
Nous utiliserons Python pour rester basique et mieux voir les problèmes. Nous pourrons cependant aller du côté de Sage pour avoir un beau graphique ou faire un calcul en précision infinie.

```
sage: P = plot(sqrt(1-x^2),xmin=0,xmax=1)
sage: P.show(aspect_ratio=1)
```

Méthode des rectangles

C'est a priori la plus grossière mais la plus simple à écrire. Petit rappel :

$$\int_a^b f(x)\,\mathrm{d}x = \sum_{k=0}^{n} \frac{b-a}{n} f\left(a + k\frac{b-a}{n}\right) + E_n$$

avec E_n l'erreur commise.
On pourrait avoir cette idée pour calculer cette approximation sur Python :

```
def int_rec_droite(f,a,b,N):
    S = 0
    dt = (b-a)/N
    for k in range(N):
        S += f(a + k*dt)*dt
    return S
```

Écrivez la fonction jumelle **`int_rec_gauche(f,a,b,N)`**.
Il *semble* falloir dix millions d'itérations pour obtenir six bonnes décimales de π...

```
In [1]: from math import sqrt,pi

In [4]: for k in range(8): print(pi-4*int_rec_gauche(f,0.,1.,10**k))
3.141592653589793
0.2370743273414746
0.021175621810747725
0.0020371866787654014
0.00020117597846525115
2.0037187828503278e-05
2.001175991139803e-06
2.0003720369032862e-07
```

Méthode des trapèzes

C'est déjà un peu plus précis, à vue d'œil. Créer une fonction **`int_trap(f,a,b,N)`**. et faites les mêmes tests.

```
In [6]: for k in range(8): print(pi-4*int_trap(f,0.,1.,10**k))
-0.32250896154796127
-0.01081877967185152
-0.00034420431021464637
-1.0891323113604301e-05
-3.444348353198734e-07
-1.0892040602783482e-08
-3.4457015019029313e-10
-1.162758778150419e-11
```

On multiplie notre fonction par 4 et donc son erreur avec. Soyons plus malins...

```
In [7]: for k in range(8): print(0.25*pi-int_trap(f,0.,1.,10**k))
-0.08062724038699032
-0.00270469491796288
-8.605107755366159e-05
-2.7228307784010752e-06
-8.610870882996835e-08
-2.7230101506958704e-09
-8.614253754757328e-11
-2.9068969453760474e-12
```

Pour plus de précision, on pourrait être tenté de prendre un plus petit arc de cercle, disons entre $\frac{\pi}{3}$ et $\frac{\pi}{2}$, en utilisant encore f. En effet, la fonction n'étant pas dérivable en 1, on a un peu peur que cela crée des perturbations. Et intuitivement, on se dit qu'en limitant l'intervalle d'intégration, on devrait limiter l'ampleur de l'approximation.
Avec la méthode `for` :

```
In [9]: for k in range(8): print(pi/12-(int_trap(f,0,1/2,10**k)-sqrt(3)/8))
-0.005817179530668071
-6.0117319471197916e-05
-6.014041918356305e-07
-6.014064968251631e-09
-6.013917142055902e-11
-6.002975894148221e-13
-8.43769498715119e-15
-4.418687638008123e-14
```

Cela semble mieux fonctionner. Bizarre tout de même ce dernier résultat....
On a utilisé des segments de droite horizontaux, des segments de droite obliques...Et si l'on utilisait des segments de parabole : intuitivement, cela colle plus à la courbe, cela devrait être plus précis.

Méthode de Simpson

On interpole la courbe par un arc de parabole. On veut que cet arc passe par les points extrêmes de la courbe et le point d'abscisse le milieu de l'intervalle. Pour cela, on va déterminer les c_i tels que :

$$\int_a^b f(x)\mathrm{d}x = c_0 f(a) + c_1 f\left(\frac{a+b}{2}\right) + c_2 f(b)$$

soit exacte pour $f(x)$ successivement égale à 1, x et x^2.
Posons $h = b - a$ et ramenons-nous au cas $a = 0$. On obtient le système suivant :

$$\begin{cases} c_0 + c_1 + c_2 = h \\ c_1 + 2c_2 = h \\ c_1 + 4c_2 = \frac{4}{3}h \end{cases}$$

Résolvez-le et expliquez le résultat :

$$\int_a^b f(x)\mathrm{d}x = \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right)$$

Créez **`int_simps(f,a,b,N)`**.
On subdivise l'intervalle d'intégration et on utilise la méthode de Simpson sur chaque subdivision avec notre petit arc de cercle :

```
In [17]: for k in range(8): print(pi/12-(int_simps(f,0,1/2,10**k)-sqrt(3)/8))
5.500957927112582e-05
6.664729046423901e-09
6.684097719755755e-13
3.885780586188048e-16
```

```
-1.7208456881689926e-15
-4.907185768843192e-14
4.030109579389318e-14
-6.844913524872709e-12
```

Étrange cette précision fluctuante....
On s'aperçoit qu'on est un peu comme un spécialiste de la tectonique des plaques prenant des photos d'une plage tous les mois à la même heure. Il peut tout à fait se produire que le montage des photos mette en évidence que le niveau de la mer descend et il pourra dire à la radio que les gaz à effet de serre n'ont aucune influence sur la fonte des glaces...

Finalement, il va falloir faire un peu plus de mathématiques et d'informatique et ne pas se contenter d'observer des résultats au petit bonheur.
Avant d'attaquer les mathématiques, il faut avoir à l'esprit que le processeur compte en base 2 et nous en base 10 et que son « zéro » vaut environ $2,2\times10^{-16}$: cela vient du fait que la mantisse a 53 bits ce qui correspond en gros à 16 chiffres décimaux. Il faut bien noter que ce n'est pas zéro mais une valeur nommée communément (pas seulement sur Python) **epsilon**.
Comme c'est l'ordinateur qui va compter, il faudrait plutôt chercher à le ménager et en tenir compte.

Il nous faut donc quand même regarder sous le capot pour comprendre la panne.
Le problème, c'est que nous ne travaillons pas en précision infinie. En chargeant le module **sys**, on a accès à la commande **float_info.epsilon** qui donne la différence entre **1.0** et le nombre en notation scientifique suivant le plus proche :

```
In [18]: from sys import*
In [19]: float_info.epsilon
Out[19]: 2.220446049250313e-16
```

Eh oui, la largeur des pipe-lines du microprocesseur n'est pas infinie. Ses « réels » admettent des successeurs.
Mais attention, entre 0 et 1, les choses sont différentes :

```
In [20]: e = float_info.epsilon
In [21]: e
Out[21]: 2.220446049250313e-16
In [22]: e/2
Out[22]: 1.1102230246251565e-16
In [23]: 1 + e/2
Out[23]: 1.0
In [24]: 1 + e/2 == 1
Out[24]: True
In [25]: 0 + e/2
Out[25]: 1.1102230246251565e-16
In [26]: e * 1e16
Out[26]: 2.220446049250313
In [27]: -1 - e/2
Out[27]: -1.0
In [28]: 1 - e/2
Out[28]: 0.9999999999999999
```

Cet **epsilon** n'est pas zéro, rappelons-le, mais détermine à partir de quand deux flottants seront considérés comme égaux par le système.
Utilisons à nouveau le test `pseudo_egal_float` introduit à la section précédente.

```
def pseudo_egal_float(a, b):
    return abs(a - b) <= (float_info.epsilon * min(abs(a), abs(b)))
```

Si on n'y prête pas attention, on peut arriver à des résultats surprenants :

```
In [30]: pseudo_egal_float(0.1 + 0.1 + 0.1 , 0.3)
Out[30]: True
In [31]: 0.1 + 0.1 + 0.1 == 0.3
Out[31]: False
In [32]: 3 * 0.1 == 0.3
Out[32]: False
In [33]: 4 * 0.1 == 0.4
Out[33]: True
In [34]: 10 + float_info.epsilon - 10
Out[34]: 0.0
In [35]: 1 + float_info.epsilon - 1
Out[35]: 2.220446049250313e-16
In [36]: 10 + 10*float_info.epsilon - 10
Out[36]: 1.7763568394002505e-15
In [37]: x = 0.1
In [38]: 3*x - 0.3
Out[38]: 5.551115123125783e-17
In [39]: 4*x - 0.4
Out[39]: 0.0
```

Rappelons également que le processeur est plus à l'aise avec les puissances de 2 car une multiplication par 2 ou une de ses puissances revient à un décalage dans son écriture binaire.
Ici, $0,25 = \frac{1}{4}$ en base 10 donc $\frac{1}{100} = 0,01$ en base 2.
Pour $0,1 = \frac{1}{10}$ en base 10, on obtient $\frac{1}{1010}$ en base 2 ; posez la division !
On retrouve au bout d'un moment un reste précédent donc $\frac{1}{10}$ en base 10 admet en base 2 un développement périodique : **`0,0 0011 0011 0011...`**.
On peut alors savoir comment ces nombres sont codés sur un ordinateur disposant de 53 bits pour la mantisse, faire de même pour $0,3$ et $3 \times 0,1$ et découvrir pourquoi **`0.3-3*0.1`** est non nul.
Pour un exposé brillant et complet sur le sujet, étudier les nombreux documents que William Kahan, fondateur de la norme IEEE 754, a mis en ligne sur sa page web : .
Reprenons l'affinement successif de notre subdivision mais avec des nombres de subdivisions égaux à des puissances de 2 :

```
In [60]: for k in range(13): print(pi/12 - (int_simps(f,0,1/2,2**k) - sqrt(3)/8))
5.500957927112582e-05
3.935118889741851e-06
2.5687273891294993e-07
1.624742479444663e-08
1.0185862153733183e-09
6.371086991308061e-11
3.9825920339353615e-12
2.490230244234226e-13
1.5376588891058418e-14
6.661338147750939e-16
-5.551115123125783e-17
-1.1657341758564144e-15
-7.771561172376096e-16
```

Tout a l'air de bien fonctionner jusqu'à 2^{10} mais ensuite, on arrive aux alentours de **`epsilon`** et cela commence à se détraquer informatiquement.
En affinant grossièrement à coups de puissances de 10, nous étions passés à côté du problème.
Comme quoi, agir intuitivement en mathématiques ou en informatique (ici : « plus on subdivise petit, meilleure sera la précision ») peut entraîner de graves erreurs...
La fonction est aussi à prendre en considération, puisqu'elle demande de soustraire à 1 un tout petit nombre :

```
In [62]: f(1e-9)
Out[62]: 1.0
```

De plus, cela explique aussi les différences entre la « méthode **while** » et la « méthode **for** ». Le test **while t+dt<=b** peut s'arrêter pour de mauvaises raisons. Par exemple :

```
In [63]: 1 + 0.1 + 0.1 + 0.1 - 0.3 <= 1
Out[63]: False
```

La boucle peut ainsi s'arrêter inopinément, alors qu'on est loin de la précision demandée.
Précédemment, avec la méthode « gros sabots », nous n'avions pas vu de différence entre la méthode des trapèzes et la méthode de SIMPSON ce qui contredisait notre intuition. Observons à pas plus feutrés ce qui se passe et incluons les rectangles :

```
In [66]:
print('-'*75)
print('{:22s} | {:22s} | {:22s}'.format('rectangle gauche', 'trapeze', 'simpson'))
print('-'*75)

for k in range(10):
    print("{:1.16e} | {:1.16e} | {:1.16e}".format(\
        pi/12 - (int_rec_gauche(f, 0, 1/2, 2**k)-sqrt(3)/8), \
        pi/12 - (int_trap(f, 0, 1/2, 2**k)-sqrt(3)/8), \
        pi/12 - (int_simps(f, 0, 1/2, 2**k)-sqrt(3)/8)))

---------------------------------------------------------------------------
rectangle gauche       | trapeze                | simpson
---------------------------------------------------------------------------
4.5293036853039759e-02 | -5.8171795306680707e-03 | 5.5009579271125819e-05
1.9737928661185844e-02 | -1.4896493887858187e-03 | 3.9351188897418510e-06
9.1241396362000127e-03 | -3.7497837725530836e-04 | 2.5687273891294993e-07
4.3745806294723244e-03 | -9.3912877730750743e-05 | 1.6247424794446630e-08
2.1403338758707591e-03 | -2.3488877122057605e-05 | 1.0185862153733183e-09
1.0584224993743230e-03 | -5.8728876539682062e-06 | 6.3710869913080614e-11
5.2627480586020514e-04 | -1.4682637225482686e-06 | 3.9825920339353615e-12
2.6240327106874517e-04 | -3.6706854433798952e-07 | 2.4902302442342261e-13
1.3101810126225910e-04 | -9.1767299481571030e-08 | 1.5376588891058418e-14
6.5463166981361010e-05 | -2.2941835209344674e-08 | 6.6613381477509392e-16
```

On distingue mieux cette fois la hiérarchie des méthodes selon leur efficacité.
Nous n'avons pas le choix : il est temps à présent d'invoquer les mathématiques pour préciser un peu ces observations, même si cela peut paraître violent de se lancer dans un raisonnement mathématique avec des calculs, des théorèmes, sans machine.
En fait, ces trois méthodes sont des cas particuliers d'une même méthode : on utilise une interpolation polynomiale de la fonction f de degré 0 pour les rectangles, de degré 1 pour les trapèzes et de degré 2 pour SIMPSON.

8 Méthodes de Newton-Cotes et programmation objet

Voici une présentation plus professionnelle, aussi bien au niveau mathématique qu'informatique mais il aurait été abrupt de commencer par là...
Pour donner une valeur approchée de l'intégrale I, une première idée est de remplacer la fonction f par un polynôme P qui interpole f en plusieurs points. Cependant, dès qu'on augmente le nombre de pivots, on risque fort de se retrouver confronté au phénomène de RUNGE.
On modifie alors l'idée de départ ainsi : on commence par subdiviser régulièrement l'intervalle d'intégration en n sous-intervalles $[x_j, x_{j+1}]$ où

$$x_0 = a, \quad x_n = b, \quad \text{et} \quad \forall j \in [\![0, n]\!], \quad x_j = a + jh, \quad \text{avec } h = \frac{b-a}{n}$$

Ensuite, on remplace f par un polynôme P_j qui interpole f sur chacun des « petits » segments $[x_j, x_{j+1}]$.

— Si P_j interpole f au point x_j, alors le graphe de P_j est une droite horizontale : c'est la méthode des rectangles.

— Si P_j interpole f aux points x_j et x_{j+1}, alors le graphe de P_j est une droite affine : c'est la méthode des trapèzes.

— Si P_j interpole f aux points x_j, $\frac{x_j+x_{j+1}}{2}$ et x_{j+1}, alors le graphe de P_j est une parabole : c'est la méthode de SIMPSON 1/3.

Enfin, on somme les intégrales de chaque polynôme P_j sur $[x_j, x_{j+1}]$, et on obtient une valeur approchée de I :

$$I = \sum_{j=0}^{n-1} \int_{x_j}^{x_{j+1}} f(x)\mathrm{d}x = \sum_{j=0}^{n-1} \left(\int_{x_j}^{x_{j+1}} P_j(x)\mathrm{d}x + E_j \right) \approx \sum_{j=0}^{n-1} \int_{x_j}^{x_{j+1}} P_j(x)\mathrm{d}x$$

Comme ces méthodes ont déjà été introduites à la section ??, on ne s'attarde que sur la méthode de SIMPSON 1/3 composée.
On se place sur l'intervalle $[x_j, x_{j+1}]$ et on note $\xi_j = \frac{x_j+x_{j+1}}{2}$; alors d'après les formules des différences divisées, le polynôme d'interpolation de f aux points (x_j, ξ_j, x_{j+1}) est donné par

$$P_j(x) = f[x_j] + f[x_j, \xi_j](x - x_j) + f[x_j, \xi_j, x_{j+1}](x - x_j)(x - \xi_j)$$

On en déduit, après calculs,

$$\int_{x_j}^{x_{j+1}} P_j(x)\mathrm{d}x = \frac{h}{6}\left(f(x_j) + 4f(\xi_j) + f(x_{j+1})\right)$$

Par sommation on obtient

$$\begin{aligned} I &\approx \frac{b-a}{6n}\left(f(x_0) + 4f\left(\tfrac{x_0+x_1}{2}\right) + 2f(x_1) + 4f\left(\tfrac{x_1+x_2}{2}\right) + \cdots + 2f(x_{n-1}) + 4f\left(\tfrac{x_{n-1}+x_n}{2}\right) + f(x_n)\right) \\ &\approx \frac{b-a}{6n}\left(f(x_0) + 2\sum_{j=1}^{n-1} f(x_j) + 4\sum_{j=0}^{n-1} f\left(\tfrac{x_j+x_{j+1}}{2}\right) + f(x_n)\right) \\ &\approx \frac{b-a}{6n}\sum_{k=0}^{2n} w_k f(a_k) \quad \text{avec} \quad a_k = a + k\cdot\frac{b-a}{2n} \text{ et } w_k = \begin{cases} 1 \text{ si } k = 0 \text{ ou } 2n \\ 4 \text{ si } 0 < k < 2n \text{ et } k \text{ impair} \\ 2 \text{ si } 0 < k < 2n \text{ et } k \text{ pair} \end{cases} \end{aligned}$$

On peut montrer que l'erreur commise est majorée par

$$|E| \leqslant \frac{1}{2880}\cdot h^4 \cdot \left\|f^{(4)}\right\| \cdot (b-a) \qquad (**)$$

On dit alors que la méthode est d'ordre 4 ; noter que la méthode est exacte pour tout polynôme de degré au plus 3.
Pour conclure cette section, revenons un instant sur l'inégalité $(**)$: cette majoration de l'erreur possède un intérêt théorique indéniable ; néanmoins, elle ne permet pas de déterminer directement l'ordre du pas h à choisir pour obtenir une approximation de I avec une précision donnée. En effet, comment déterminer une borne de la dérivée quatrième de f ? Numériquement, ce problème est trop complexe. En pratique, on applique la méthode avec deux pas différents (h et $2h$ pour minimiser les évaluations de f) et on utilise la différence des deux approximations numériques comme estimation de l'erreur du moins bon résultat.

9 Le nombre Pi et les arctangentes

Le nombre Pi et Machin

Python possède un module de calcul en multiprécision avec les quatre opérations arithmétiques de base et la racine carrée : xcom]getcontext().prec

```
In [72]: from decimal import*
# on regle la precision a 30 chiffres
In [73]: getcontext().prec = 30
# on rentre les nombres entre apostrophes, comme des chaines
```

```
In [74]: deux=Decimal('2')
In [75]: deux.sqrt()
Out[75]: Decimal('1.41421356237309504880168872421')
In [76]: getcontext().prec = 50
In [77]: deux.sqrt()
Out[77]: Decimal('1.4142135623730950488016887242096980785696718753769')
```

Mais ça ne va pas nous avancer à grand chose puisque pour avoir 15 bonnes décimales avec la méthode de SIMPSON, il nous a fallu dix millions d'itérations.
Faisons un petit détour par l'histoire...
Tout commence à peu près en 1671, quand l'écossais James GREGORY découvre la formule suivante :

$$\arctan(x) = \sum_{k=0}^{+\infty} \frac{(-1)^k x^{2k+1}}{2k+1}$$

L'inévitable LEIBNIZ en publie une démonstration en 1682 dans son *Acta Eruditorum* où il est beaucoup question de ces notions désuètes que sont la géométrie et la trigonométrie.
Dans la même œuvre, il démontre ce que nous appelons aujourd'hui le critère spécial des séries alternées :

Soit (u_n) *une suite réelle alternée. Si* $(|u_n|)$ *est décroissante et converge vers 0 alors :*

— $\sum u_n$ *converge ;*

— $|R_n| \leqslant |u_{n+1}|$ *où* (R_n) *est la suite des restes associés à* $\sum u_n$.

On peut même envisager d'évoquer une démonstration de ce théorème en terminale puisqu'on y parle essentiellement de suites adjacentes...
On en déduit donc que :

$$\left|\arctan(x) - \sum_{k=0}^{n} \frac{(-1)^k x^{2k+1}}{2k+1}\right| < \frac{x^{2n+3}}{2n+3}$$

En 1706, l'anglais John MACHIN en démontra la fameuse formule :

$$4\arctan\frac{1}{5} - \arctan\frac{1}{239} = \frac{\pi}{4}$$

ce qui lui permit d'obtenir 100 bonnes décimales de π sans l'aide d'aucune machine mais avec la formule proposée par son voisin écossais.
On appelle polynômes de GREGORY les polynômes $G_n(X) = \sum_{k=0}^{n} \frac{(-1)^k X^{2k+1}}{2k+1}$.
Écrivez une fonction **greg(a,N)** qui donne une valeur de $G_N(a)$.
Pour cela, on aura besoin de travailler en multiprécision.
On crée une fonction **pi_machin** : que fait-elle ?

```
def pi_machin(d):
    getcontext().prec = d + 2
    rap = 2*log(5)/log(10)
    N = int(d/rap) + 1
    return 4*(4*greg(5,N) - greg(239,N))
```

On récupère sur Sage une approximation de π.

```
sage: pi.n(digits=1000)
```

On la copie dans Python et on compare :

```
PI =
Decimal('3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706798214808651328230664709384460955058223172535940812848111745028410270193852110555964462294895493038196442881097566593344612847564823378678316527120190914564856692346034861045432664821339360726024914127372458700660631558817488152092096282925409171536436789259036001133053054882046652138414695194151160943305727036575959195309218611738193261179310511854807446237996274956735188575272489122793818301194912983367336244065664308602139494639522473719070217986094370277053921717629317675238467481846766940513200056812714526356082778577134275778960917363717872146844090122495343014654958537105079227968925892354201995611212902196086403441815981362977477130996051870721134999999837297804995105973173281609631859502445945534690830264252230825334468503526193118817101000313783875288658753320838142061717766914730359825349042875546873115956286388235378759375195778185778053217122680661300192787661119590921642019 9')
```

En une seconde et demie, on a mille bonnes décimales de π...

```
In [91]: PI-pi_machin(1000)
Out[91]: Decimal('-1.5E-1000')
```

EXERCICES

Recherche 2 - 1 Formule de dérivation

En utilisant les résultats de la section 2.1.2 page 41, démontrez la formule de dérivation de $x \mapsto x^n$ et $x \mapsto 1/x^n$ pour tout entier positif n.

Recherche 2 - 2 Méthode de Briggs

Inspirez-vous de la méthode présentée à la section 2.2.2 page 42 pour calculer une approximation de $\log(3)$ et $\log(7)$.

Recherche 2 - 3 Interpolation de Newton(1) : le gag

Déterminez le polynôme de Newton passant par les points (1,1), (2,8), (3,27) et (4,64).

Recherche 2 - 4 Interpolation de Newton(2)

Déterminez le polynôme de Newton passant par les points (1,5), (2,2), (3,5), (4,2) et (5,2).

Recherche 2 - 5

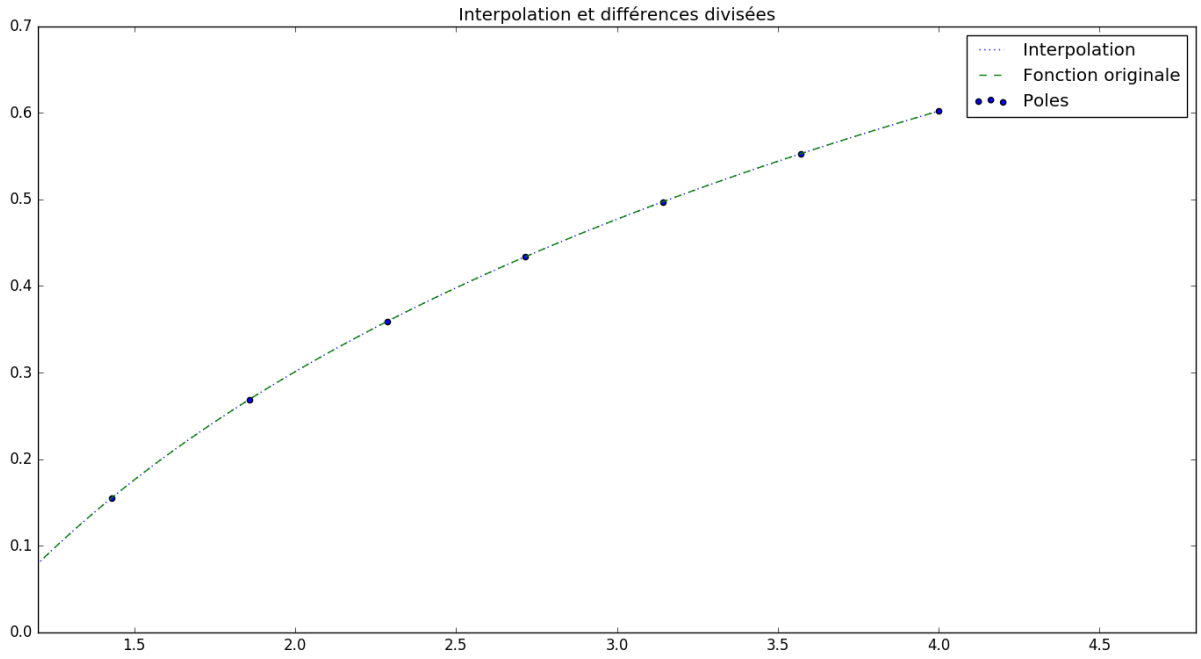
Écrivez la formule du théorème 2 - 6 page 46 sans ... mais avec $\sum$ et $\prod$

Recherche 2 - 6 Calculs des coefficients des polynômes de Newton

Reprenez les calculs effectués dans la section 2.2.3.3 page 45 mais avec n points d'abscisses 0 à $n-1$.

Recherche 2 - 7 Polynôme de Newton : le programme

Nous voudrions programmer l'interpolation du logarithme décimal sur [1,4] pour obtenir un tracé tel que celui-ci :



obtenu en utilisant la bibliothèque `MatplotLib` (`import matplotlib.pyplot as plt`) :

```
In [25]: compareTrace(np.log10, 16, 1, 4, 0, 0.7)
```

On pourra créer des fonctions intermédiaires :

1. Déterminez une fonction `ddNewton :: [a] -> [a]` qui renvoie la liste des $\Delta^i y_1$ du « triangle » de NEWTON (ceux en gras...) :

```
In [60]: ddNewton([5,2,5,2,2])
Out[60]: [5, -3, 6, -12, 21]
```

   On ourra par exemple partir d'un tableau carré où chaque ligne `i` est remplie de `Y[i]` si on note `Y` le paramètre de la fonction.

2. La formule du théorème 2 - 5 page 46 laisse entrevoir un `zipWith` (ou braguette avec). Nous disposons déjà de la liste des Δ^i. Il nous reste à construire la *base* des polynômes de NEWTON puis à zipper et additionner (bref faire un produit scalaire : une liste est un vecteur...) pour créer une fonction `interpol` :

```
In [61]: interpol(1.5,[1,8,27,64],1)
Out[61]: 3.375
```

3. Pour le tracé, expliquez :

```
def compareTrace(f, xmin, xmax, ymin, ymax):
    X  = np.linspace(xmin, xmax, 500)
    Y  = f(X)
    nb_subdivs = xmax - xmin + 1
    subdiv = np.linspace(xmin, xmax, nb_subdivs)
    fY = f(subdiv)
    coeffs = ddNewton(fY)
    YY = ( lambda x: interpol( x, subdiv, fY, coeffs ) )(X)
    plt.scatter(subdiv, fY, label = 'Poles')
    plt.plot(X, YY , label = 'Interpolation', ls=':')
    plt.plot(X, Y , label = 'Fonction originale',ls='--')
    plt.axis([xmin*1.2, xmax*1.2, ymin, ymax])
    plt.legend()
    plt.title('Interpolation et différences divisées')
    plt.show()
```

Recherche 2 - 8 Algorithme de Horner

La méthode que nous allons voir porte le nom du britannique William George HORNER (1786 - 1837) mais en fait elle fut publiée presque 10 ans auparavant par un horloger londonien, Theophilius HOLDRED et simultanément par l'italien Paolo RUFFINI (1765 - 1822) mais fut déjà utilisée par NEWTON 150 ans auparavant et par le chinois ZHU SHIJE cinq siècles plus tôt (vers 1300) et avant lui par le Persan SHARAF AL-DIN AL-MUZAFFAR IBN MUHAMMAD IBN AL-MUZAFFAR AL-TUSI vers (1100) et avant lui par le Chinois LIU HUI (vers 200) révisant un des résultats présent dans *Les Neuf Chapitres sur l'art mathématique* publié avant la naissance de JC...
Soit P un polynôme. Il s'agit de calculer l'évaluation de P en une valeur particulière.
Prenons l'exemple de $P(x) = 3x^5 - 2x^4 + 7x^3 + 2x^2 + 5x - 3$. Le calcul classique nécessite 5 additions et 15 multiplications (vérifiez-le !).
On peut faire pas mal d'économies de calcul en suivant le schéma suivant :

$$\begin{aligned} P(x) &= \underbrace{a_n x^n + \cdots + a_2 x^2 + a_1 x}_{\text{on met } x \text{ en facteur}} + a_0 \\ &= \left(\underbrace{a_n x^{n-1} + \cdots + a_2 x}_{\text{on met } x \text{ en facteur}} + a_1 \right) x + a_0 \\ &= \cdots \\ &= \left(\cdots \left(\left((a_n x + a_{n-1}) x + a_{n-2} \right) x + a_{n-3} \right) x + \cdots \right) x + a_0 \end{aligned}$$

Ici cela donne $P(x) = (((((3x) - 2)x + 7)x + 2)x + 5)x - 3$ c'est-à-dire 5 multiplications et 5 additions.
En fait il y a au maximum 2×degré de P opérations (voire moins avec les zéros).
Notez de plus qu'une machine disposant de la FMA effectue ce calcul encore plus exactement.

```
In [64]: horn([3,-2,7,2,5,-3], 1)
Out[64]: 12

In [65]: horn([3,-2,7,2,5,-3], 2)
Out[65]: 27
```

Reproduisez le graphe de la section 2.3 page 46.

Recherche 2 - 9 Méthode d'Euler

On ne dispose que de la dérivée d'une fonction : peut-on tracer malgré cela l'allure de sa courbe représentative ?
On créera une fonction Python permettant par exemple de tracer la courbe représentative de la fonction dont la dérivée est la fonction $x \mapsto x$ et qui s'annule pour $x = 1$.
Généralisez votre fonction Python pour faire de même avec une fonction qui vérifie une équation différentielle

$$y' = f(x, y)$$

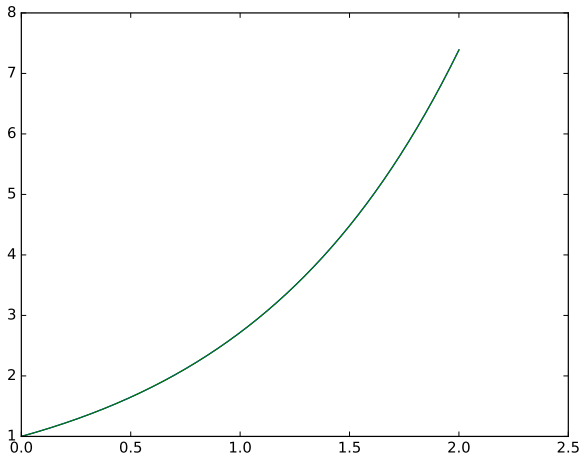
On commencera par créer la liste des coordonnées des points de la courbe approchée.
On l'utilisera ensuite dans la fonction :

```
def affiche_euler(f, x0, y0, xf, h) :
    xs, ys = euler(f, x0, y0, xf, h)
    plt.plot(xs, ys)
    plt.show()
```

On obtient par exemple avec :

```
In [151]: affiche_euler(lambda x,y :y, 0,1,2,1e-4)
```

la courbe suivante :



Essayez d'étendre au cas des équations différentielles d'ordre 2 : $y'' = f(x, y, y')$.

Recherche 2 - 10 Construction de l'exponentielle

Une équation différentielle

L'équation différentielle $f' = k\,f$ se retrouve dans de nombreux problèmes : désintégration des noyaux des atomes d'un corps radioactif, datation au carbone 14, évolution d'une population où la croissance est proportionnelle au nombre d'habitants, etc. Le problème est de trouver une fonction la satisfaisant.
Par exemple, certains phénomènes en mécanique conduisent à étudier l'équation différentielle $f'' = -f$. Nous connaissons au moins deux fonctions la satisfaisant : cosinus et sinus.

Construction approchée du graphe d'une solution par la méthode d'Euler

Soit f une fonction dérivable sur $\mathbb{R}$ vérifiant $f(0) = 1$ et, pour tout x, $f'(x) = f(x)$. Utilisez la méthode d'Euler pour déterminer une représentation graphique de f sur $[0,2]$.

Analyse : étude des propriétés mathématiques d'une solution

Soit f une fonction dérivable sur $\mathbb{R}$ vérifiant $f(0) = 1$ et, pour tout x, $f'(x) = k\,f(x)$, avec $k \neq 0$.

1. Montrez que $f'(0) = k$.
2. Soit y un réel fixé et g_y la fonction définie par $g_y(x) = f(x+y)f(-x)$
   i. Montrez que g_y est dérivable sur $\mathbb{R}$ et calculez $g_y'(x)$.
   ii. Calculez $g_y(0)$ et déduisez-en que pour tous x et y réels, $f(x+y)\,f(-x) = f(y)$ (1)
3. Montrez alors successivement que :
   i. pour tout réel x, $f(x)f(-x) = 1$
   ii. f ne s'annule pas sur $\mathbb{R}$
   iii. pour tous réels x et y, $f(x+y) = f(x)\,f(y)$.

Unicité de la fonction solution

Peut-on trouver une autre fonction, φ, distincte de f, et vérifiant les mêmes propriétés que f, à savoir : φ est une fonction dérivable sur $\mathbb{R}$, vérifiant $\varphi(0) = 1$ et, pour tout x, $\varphi'(x) = k\,\varphi(x)$, avec $k \neq 0$?

Comme f ne s'annule pas sur $\mathbb{R}$, on peut définir la fonction $\psi = \varphi/f$.
Vérifiez que ψ est dérivable sur $\mathbb{R}$, calculez sa dérivée. Que peut-on en déduire pour ψ ? Montrez alors que $f = \varphi$.

Synthèse

Nous avons cherché des solutions au problème :

f une fonction dérivable sur $\mathbb{R}$ vérifiant $f(0) = 1$ et, pour tout x, $f'(x) = k\,f(x)$, avec $k \neq 0$.

Nous avons montré que, *si une telle fonction existe* (ce que nous prouverons dans un prochain chapitre), alors elle est unique et elle vérifie nécessairement la relation $f(x+y) = f(x)\,f(y)$ (2).
Il reste à vérifier que, réciproquement, une fonction dérivable, non nulle, vérifiant la relation (2) est nécessairement telle que $f(0) = 1$ et vérifie pour tout réel x $f'(x) = kf(x)$, avec k un réel non nul.

Cette vérification n'est pas anodine et conclut notre raisonnement d'*analyse-synthèse*.

1. Montrez que f ne s'annule pas et que f est à valeurs strictement positives.
2. Montrez que, comme f n'est pas la fonction nulle, alors $f(0) = 1$ en utilisant la relation (2).
3. Soit a un réel fixé. On définit la fonction $\varphi : x \mapsto f(x+a)$ et la fonction $\psi : x \mapsto f(x) \times f(a)$.
   Montrez que $f'(x+a) = f(a) \times f'(x)$, puis que, pour tout réel a, $f'(a) = k\,f(a)$, où k est un réel que vous déterminerez.

Recherche 2 - 11 Lynx et lapins

Alfred Lokta
US (1880-1949)

Vito Volterra
It. (1860-1940)

Vous connaissez ce problème par cœur...

$$\begin{cases} \dfrac{\mathrm{d}x(t)}{\mathrm{d}t} &= x(t)\left(\alpha - \beta y(t)\right) \\ \dfrac{\mathrm{d}y(t)}{\mathrm{d}t} &= -y(t)\left(\gamma - \delta x(t)\right) \end{cases}$$

— t est le temps ;

— $x(t)$ est l'effectif des proies en fonction du temps ;

— $y(t)$ est l'effectif des prédateurs en fonction du temps ;

— les dérivées $\mathrm{d}x(t)/\mathrm{d}t$ et $\mathrm{d}y(t)/\mathrm{d}t$ représentent la variation des populations au cours du temps.

Les paramètres suivants caractérisent les interactions entre les deux espèces :

— α, taux de reproduction des proies (constant, indépendant du nombre de prédateurs) ;

— β, taux de mortalité des proies dû aux prédateurs rencontrés ;

— γ, taux de mortalité des prédateurs (constant, indépendant du nombre de proies) ;

— δ, taux de reproduction des prédateurs en fonction des proies rencontrées et mangées ;

Vous connaissez la méthode d'Euler par cœur...
Tracez x et y en fonction de t puis y en fonction de x.

Recherche 2 - 12 MINES-PONTS 2015

Introduction

Les imprimantes sont des systèmes mécatroniques (combinaison synergique et systémique de la mécanique, de l'électronique et de l'informatique en temps réel) fabriqués en grande série dans des usines robotisées. Pour améliorer la qualité des produits vendus, il a été mis en place différents tests de fin de chaîne pour valider l'assemblage des produits. Pour un de ces tests, un opérateur connecte l'outil de test sur la commande du moteur de déplacement de la tête d'impression et sur la commande du moteur d'avance papier. Une autre connexion permet de récupérer les signaux issus des capteurs de position. Différentes commandes et mesures sont alors exécutées. Ces mesures sont transmises sous la forme d'une suite de caractères vers un ordinateur. Cet ordinateur va effectuer différentes mesures pour valider le fonctionnement de l'électromécanique de l'imprimante. L'ensemble des mesures et des analyses est sauvegardé dans un fichier texte. Afin de minimiser l'espace occupé, les fichiers sont compressés. Une base de données stocke les informations concernant les mesures et les imprimantes sur lesquelles elles portent, et permet à l'entreprise d'améliorer la qualité de la production après diverses études statistiques.

Le problème comporte trois grandes parties indépendantes, sauf par le thème traité.

— Dans la partie 2.11, on s'intéresse au processus de réception par l'ordinateur des données transmises par le capteur.

— Dans la partie 2.12, on procède au traitement des données numériques récupérées à fin de validation de l'imprimante assemblée.

— Dans la partie 2.13, on étudie quelques aspects de la base de données constituée.

Réception des données issues de la carte d'acquisition

Le capteur

Le capteur utilisé est analogique, et comporte un convertisseur (appelé convertisseur analogique numérique) qui traduit les mesures effectuées sous forme numérique. Chaque donnée est ainsi codée comme un entier.

Q1. Rappeler un principe de représentation en machine des entiers signés sur 10 bits. Préciser la plage des valeurs entières représentables selon ce principe.

Q2. On considère que les valeurs analogiques s'étendent en pleine échelle de -5 V à 5 V et sont converties en entiers signés représentés sur 10 bits. Donner une valeur approchée de la résolution de la mesure en volts.

Dans la suite du problème, la question du codage des entiers en binaire n'intervient plus.

Liaison avec l'ordinateur

Une liaison série asynchrone permet la communication entre la carte de commande/acquisition et le PC. Les informations correspondant à une mesure sont envoyées par la carte électronique sous la forme d'une suite de caractères, appelée trame. Plusieurs trames consécutives correspondant à différents jeux de mesures peuvent être transmises sans interruption, le format même des trames devant permettre de les délimiter. Un exemple de trame est :

'U'	'0'	'0'	'3'	'+'	'0'	'1'	'2'	'+'	'0'	'0'	'4'	'-'	'0'	'2'	'3'	'9'	'9'	'9'	'3'

Une trame est donc constituée de la manière suivante :

— un caractère d'en-tête, qui est une des trois lettres `'U'`, `'I'`, `'P'`, permettant d'identifier la nature de la mesure effectuée (`'U'` : tension moteur, `'I'` : courant moteur, `'P'` : position absolue) ;

— trois chiffres constituant une valeur entière précisant le nombre N de mesures qui constituent la suite de la trame ;

— un ensemble de N blocs consécutifs de quatre caractères. Chacun des blocs est constitué d'un caractère de signe `'+'` ou `'-'`, suivi de trois chiffres donnant une valeur absolue. L'entier signé ainsi codé sur quatre caractères donne une valeur issue de la conversion analogique/numérique d'une mesure ;

— un dernier bloc de quatre chiffres constituant ce qu'on appelle une somme de contrôle (*checksum*). Celle-ci est calculée en formant la somme des valeurs des données précédentes de la trame, et en prenant le reste de la division euclidienne par 10000 de cette somme.

La somme de contrôle (*checksum*) permet de vérifier si la trame a été correctement transmise. On peut, à réception d'une trame, calculer la somme de contrôle associée aux données reçues, et la comparer à la somme de contrôle contenue dans la trame. Si ces deux valeurs ne coïncident pas, les données ont été altérées lors de la transmission, et il convient de ne pas les exploiter.

La fonction `car_read(nbre_carac)` permet d'obtenir des caractères consécutifs issus de la liaison asynchrone. Elle prend en argument un entier `nbre_carac`, et renvoie une chaîne de caractères constituée de `nbre_carac` caractères. Ainsi, dans l'exemple ci-dessus, deux appels consécutifs à `car_read(5)` puis `car_read(3)`, renverront respectivement les chaînes `'U003+'` et `'012'`.

Q3. Toujours à partir de l'exemple donné plus haut, on suppose que les deux appels

`car_read(5)` puis `car_read(3)`

ont été effectués. Quelle valeur est alors renvoyée par l'appel `car_read(4)` ?

Le programme de lecture des données transmises à l'ordinateur lit des caractères jusqu'à obtention d'un caractère `x` d'en-tête égal à `'U'`, `'I'` ou `'P'`. Il fait alors appel à une fonction de lecture de trame `lecture_trame(x)` lisant les données numériques contenues dans la trame qui commence par `x`.

Q4. Écrire la fonction `lecture_trame(x)` prenant en argument le caractère d'en-tête `x` qui vient d'être lu, et renvoyant une liste `[x,L,S]`, où `L` est une liste d'entiers représentant les mesures transmises dans la trame commençant par `x`, et `S` est la valeur de la somme de contrôle transmise dans la trame commençant par `x`.

Dans la question suivante, on utilise une valeur issue d'un appel à la fonction

`lecture_trame(x)`.

Une telle valeur sera stockée dans une variable `trame` *via* une affectation

`trame = lecture_trame(x)`.

Q5. Écrire une fonction `checkSum(trame)` prenant en argument une liste `trame`, et renvoyant un booléen indiquant si le troisième élément de la liste `trame` est égal à la somme de contrôle calculée à partir des valeurs stockées dans le deuxième élément de cette liste.

Q6. Écrire une fonction `lecture_intensite()`, sans argument, et procédant au traitement suivant :

① obtenir un caractère jusqu'à ce que ce soit le caractère `'I'` ;

② lire la trame dont on vient de lire le caractère d'en-tête et stocker les informations dans une variable `trame` ;

③ si la variable `trame` ne présente pas une somme de contrôle valide, recommencer à ① ;

④ renvoyer la liste des mesures numériques alors contenues dans la variable `trame`.

On fait l'hypothèse que le canal de transmission est de qualité suffisante pour assurer qu'on finira par obtenir une trame dont la somme de contrôle est correcte.

Analyse des mesures

Les fonctions demandées pourront être écrites dans le langage `Python` (éventuellement à l'aide de sa bibliothèque `numpy`) ou dans le langage `Scilab`.
On exclut dans cette partie tout appel à une fonction de calcul approché d'intégrale ou de résolution approchée d'équations différentielles qui serait préprogrammée dans le langage choisi ou dans l'une de ses bibliothèques.

Traitement numérique

Q7. Rappeler le principe de la méthode des trapèzes permettant de calculer une valeur approchée de l'intégrale d'une fonction f sur un intervalle $[a,b]$. On ne demande pas ici de coder de fonction informatique permettant ce calcul.

Pour valider le fonctionnement de l'imprimante, on considère une suite de valeurs de mesures de l'intensité du courant (en ampères) du moteur de la tête d'impression, stockée dans un tableau `mesures`. On s'intéresse alors à la valeur moyenne et à l'écart-type de cette distribution de valeurs.
Pour une fonction f définie (et continue) sur un segment $[0,t_{\text{final}}]$, la valeur moyenne de la fonction sur ce segment est définie par :

$$f_{\text{moy}} = \frac{1}{t_{\text{final}}}\int_0^{t_{\text{final}}} f(t)\mathrm{d}t.$$

On ne dispose ici des valeurs de l'intensité qu'en des instants prédéfinis : les mesures ont été effectuées toutes les 2 millisecondes.

Q8. Écrire une fonction `trapezes(L)` prenant en argument un tableau `L` de valeurs numériques, et renvoyant une valeur approchée, calculée par la méthode des trapèzes, de la valeur moyenne d'une grandeur variant durant un certain intervalle de temps $[0,t_{\text{final}}]$, en supposant que le tableau `L` contient les valeurs de cette grandeur mesurées toutes les 2 millisecondes.

On définit l'écart-type d'une fonction sur un segment $[0,t_{\text{final}}]$ par :

$$f_{\text{ec}} = \sqrt{\frac{1}{t_{\text{final}}}\int_0^{t_{\text{final}}} (f(t)-f_{\text{moy}})^2\mathrm{d}t}.$$

Q9. Écrire une fonction `indicateurs(mesures)` prenant en argument un tableau `mesures` de flottants codant les valeurs de l'intensité mesurées toutes les 2 millisecondes sur un intervalle de temps $[0,t_{final}]$, et renvoyant la valeur de la moyenne et de l'écart-type des valeurs de ce tableau. On privilégiera l'utilisation de la fonction `trapezes`.

Validation des mesures

On sait que chaque moteur de l'imprimante, soumis à un signal d'entrée e, doit fournir un signal de sortie s lié au signal d'entrée par l'équation différentielle linéaire du premier ordre :

$$\frac{\mathrm{d}s}{\mathrm{d}t} = -\frac{k}{10}(s-e), \tag{2.1}$$

où k est un nombre strictement positif. Le bon fonctionnement d'un tel moteur est testé en vérifiant l'adéquation des valeurs mesurées avec des valeurs issues d'une résolution numérique de cette équation.

Q10. On considère le signal d'entrée $e(t) = \sin(\omega t)$ sur l'intervalle de temps $[0,1]$ (en secondes), où ω est une constante numérique stockée dans une variable `omega`.
Écrire une fonction `simulation(...)` renvoyant un tableau de valeurs approchées, calculées à l'aide de la méthode d'Euler, d'une solution de l'équation (2.1) aux instants $[0, 0.002, 0.004, \ldots, 1]$ (en secondes). Le candidat sera amené à choisir un ou des arguments pour la fonction programmée, et commentera le choix effectué.
On rappelle que tout appel à une fonction de résolution approchée d'équations différentielles préprogrammée est exclu.

On suppose désormais qu'à l'instant $t = 0$, la valeur $s(0)$ du signal de sortie doit être nul. Le signal de sortie réel, acquis par le capteur, est stocké dans une variable `mesures` de type tableau, contenant la valeur du signal aux instants $[0, 0.002, 0.004, \ldots, 1]$ (en secondes). L'imprimante doit avoir un comportement conforme à la solution de l'équation (2.1) avec $s(0) = 0$. En raison de différences de fabrication, k peut prendre les valeurs 0.5, 1.1 ou 2, et seulement celles-ci. Le comportement du moteur est considéré valide si l'écart maximal entre valeurs observées et valeurs simulées est inférieur à un certain flottant positif `eps`, pour au moins une valeur de k parmi 0.5, 1.1 ou 2.

Q11. Écrire une fonction `validation(mesures,eps)` prenant en arguments un tableau `mesures` de valeurs observées du signal de sortie et un flottant positif `eps`, et renvoyant un booléen indiquant si le comportement du moteur peut être validé au seuil `eps`.

Bases de données

On suppose que les résultats des analyses précédentes ont été stockés dans une base de données, constituée de deux tables, dont on donne une représentation simplifiée dans l'annexe de la page 73.

Après son assemblage et avant les différents tests de validation, un numéro de série unique est attribué à chaque imprimante. À la fin des tests de chaque imprimante, les résultats d'analyse ainsi que le nom du fichier contenant l'ensemble des mesures réalisées sur l'imprimante sont stockés dans la table `testfin`. Lorsqu'une imprimante satisfait les critères de validation, elle est enregistrée dans la table `production` avec son numéro de série, la date et l'heure de sortie de production, ainsi que le nom du modèle.

Q12. Écrire une requête SQL permettant d'obtenir les numéros de série des imprimantes ayant une valeur de `Imoy` comprise strictement entre 0.4 et 0.5.

Q13. Écrire une requête SQL permettant d'obtenir les numéros de série, les valeurs de `Imoy` et `Iec` ainsi que le modèle des imprimantes ayant été validées et dont la valeur de `Imoy` est comprise strictement entre 0.4 et 0.5.

Q14. Écrire une requête SQL permettant d'obtenir les modèles des imprimantes validées et pour chacun de ces modèles, le nombre d'imprimantes validées.

Q15. Écrire une requête SQL renvoyant le numéro de série et le nom du fichier de mesures des imprimantes qui n'ont pas été validées en sortie de production. Commenter très brièvement l'intérêt de cette information pour cette ligne de production.

Annexe

Base de données

testfin				
nSerie	dateTest	Imoy	Iec	fichierMes
230-588ZX2547	2012-04-22 14-25-45	0.45	0.11	mesure31025.csv
230-588ZX2548	2012-04-22 14-26-57	0.43	0.12	mesure41026.csv
⋮	⋮	⋮	⋮	⋮

production			
Num	nSerie	dateProd	modele
20	230-588ZX2547	2012-04-22 15-52-12	JETDESK-1050
21	230-588ZX2549	2012-04-22 15-53-24	JETDESK-3050
⋮	⋮	⋮	⋮

Recherche 2 - 13

La loi de Newton sur le refroidissement dit que la vitesse de refroidissement d'un objet est proportionnelle à l'écart de température entre l'objet et le milieu ambiant.
L'inspecteur Clouseau arrive sur les lieux d'un meurtre à 9h00. Il commence par prendre la température de la victime : 30 ° C. Une heure plus tard, la température du corps est tombée à 29 ° C. Sachant que la température normale du corps d'une personne vivante en bonne santé est de 37 ° C, que la victime était syldave, qu'elle aimait les films de gladiateurs et se trouvait dans une pièce maintenue à 0 ° C, estimez l'heure du décès de la victime et la couleur de ses yeux.

On pourra avoir une idée graphique avec nos petites fonctions Python s'inspirant de la méthode d'Euler.

Recherche 2 - 14

Dans la forêt syldave, des détritus naturels (feuilles, branches, animaux morts, cadavres d'espions, etc.) tombent sur le sol et s'y décomposent. La quantité $Q(t)$ exprimée en $g \cdot m^{-2}$ de détritus jonchant le sol varie avec le temps t. On suppose que de nouveaux détritus tombent au sol à un taux constant de $200\,g \cdot m^{-2}$ par année et que les détritus accumulés au sol se décomposent au taux de 50% de la quantité de détritus jonchant le sol.

1. On note $f(t) = Q(t) - 200$. Déterminez une équation différentielle vérifiée par f.
2. En déduire l'expression générale de $Q(t)$.
3. Exprimer $Q(t)$ sachant qu'au temps $t = 0$, on comptait $50\,g \cdot\ m^{-2}$

Recherche 2 - 15

Si vous allez vous promener dans la ville de Saint-Louis dans le Missouri aux États-Unis, vous pourrez y admirer la célèbre *Gateway Arch to the West* conçue en 1947 par l'architecte finlandais Ero SAARINEN et l'ingénieur Hannskarl BANDEL et dont la construction s'acheva en 1965. Cette arche a la forme d'une *chaînette* pondérée d'équation

$$y = 212 - 21\,\mathrm{ch}(0,033x)$$

où x et y sont mesurés en mètres et rend hommage aux pionniers partis à la conquête de l'Ouest.
Pouvez-vous déterminer la hauteur de l'arche et la distance entre les deux pieds ? On rappelle que $\mathrm{ch}\,x = \frac{e^x+e^{-x}}{2}$.

Recherche 2 - 16

L'équation de la hauteur h par rapport au sol d'un fil électrique suspendu entre deux poteaux s'obtient en résolvant l'*équation différentielle* :

$$h''(x) = k\sqrt{1 + (h'(x))^2}$$

où k est un paramètre qui dépend de la densité et de la tension du fil et x est mesuré en mètres horizontalement à partir d'une origine située sur le sol en-dessous du point où la hauteur du fil est la plus faible.

1. Vérifiez que $h\ :\ x \mapsto \frac{1}{k}\mathrm{ch}(kx)$ satisfait cette équation différentielle.
2. Quelle est la hauteur minimale du fil si le paramètre k vaut $0,05$?
3. Quelle est la hauteur des poteaux (de même hauteur) s'ils sont distants de 30 m et que le paramètre k vaut 0,05 ?

Recherche 2 - 17 DL

Écrire la formule de Taylor-Mac Laurin et la formule de Taylor-Young à l'ordre n pour les fonctions suivantes et dans un voisinage de 0 :

1. $f(x) = e^x$
2. $f(x) = \dfrac{1}{1+x}$
3. $f(x) = \dfrac{1}{1-x}$
4. $f(x) = \ln(1+x)$
5. $f(x) = \sin(x)$
6. $f(x) = \sqrt{1+x}$, $n = 4$

Recherche 2 - 18 DL

Donner un $DL_3(0)$ pour les fonctions suivantes :

1. $f_1(x) = \ln(1+x)$
2. $g(x) = f_1(x^2)$
3. $f_2(x) = e^x$
4. $h(x) = f_2(-2x)$
5. $f_3(x) = f_1(x)f_2(x)$
6. $f_4(x) = f_1(x) + f_2(x)$

Recherche 2 - 19 Fonctions homographiques

Une fonction homographique est une fonction de la forme $x \mapsto \dfrac{ax+b}{cx+d}$ avec a, b, c et d des réels.

Notons g la fonction homographique correspondant au cas $a=0$, $b=d=1$ et $c=-1$.
Quel est l'ensemble de définition de g ?
Calculez $g^{(1)}(x)$, $g^{(2)}(x)$, $g^{(3)}(x$ et plus généralement $g^{(n)}(x)$.
Écrivez le plus simplement possible $T_{5,0}(g)(x)$.
Sur un même graphique, donnez l'allure des courbes représentatives de g et de $T_{5,0}(g)$ avec Python.

Soit $h(x)=\dfrac{5-3x}{1-x}$. Montrez qu'il existe deux réels α et β tels que $h(x)=\alpha+\dfrac{\beta}{1-x}$.
Déduisez-en $T_{5,0}(h)(x)$ pour x appartenant à un bon intervalle.

Recherche 2 - 20 Question ouverte...

Les polynômes de Newton et de Taylor se ressemblent non ? Dans quelle mesure ?

Recherche 2 - 21 DS

Déterminez la dérivée de la fonction $x\mapsto\sqrt{\left(\sqrt{\left(\sqrt{(\sqrt{x})}\right)}\right)}$. Vous donnerez l'expression finale sous la forme la plus simple possible.

Recherche 2 - 22 DS

Déterminez, en détaillant vos calculs, les polynômes de Taylor à l'ordre 4 au voisinage de 0 de :

1. $f_1(x)=\ln(1-x)$
2. $f_2(x)=\exp(-x)$
3. $f_3(x)=\cos(2x)$

Recherche 2 - 23 DS

Déterminez la dérivée de la fonction $t\mapsto\cos\Big(\cos\big(\cos(t)\big)\Big)$.

Recherche 2 - 24 DS

Déterminez le polynôme de Newton passant par les points (1,-5), (2,-5), (3,1) et (4,19).

Recherche 2 - 25 DS

On rappelle le résultat suivant :
Soit I un intervalle, soit f une fonction de I vers $\mathbb{R}$ de classe C^n avec $n\in\mathbb{N}$ et soit a un élément de I. Il existe alors un unique polynôme $T_{n,a}$ de degré au plus n dont les dérivées jusqu'à l'ordre n coïncident en a avec celles de f. De plus : $T_{n,a}(x)=f(a)+(x-a)f'(a)+\frac{(x-a)^2}{2!}f''(a)+\frac{(x-a)^3}{3!}f^{(3)}(a)+\cdots+\frac{(x-a)^n}{n!}f^{(n)}(a)$

1. Déterminez le polynôme de Taylor à l'ordre n de la fonction ch à partir du théorème rappelé ci-dessus.
2. Donnez directement le polynôme de Taylor à l'ordre n de la fonction $t\mapsto\frac{1}{1-t}$.
3. Déterminez le développement limité à l'ordre 5 au voisinage de 0 de la fonction $t\mapsto\frac{\mathrm{ch}(x^2)}{1-3x}$.

Recherche 2 - 26 Courbes de Bézier

Dans les années 60, les ingénieurs Pierre Bézier et Paul De Casteljau travaillant respectivement chez Renault et Citroën, réfléchissent au moyen de définir de manière la plus concise possible la forme d'une carrosserie.
Le principe a été énoncé par Bézier mais l'algorithme de construction par son collègue de la marque aux chevrons qui n'a d'ailleurs été dévoilé que bien plus tard, la loi du secret industriel ayant primé sur le développement scientifique...
Pour la petite histoire Pierre Bézier (diplômé de l'Ensam et de Supélec) fut à l'origine des premières machines à commandes numériques et de la Cao ce qui n'empêcha pas sa direction de le mettre à l'écart : il se consacra alors presque exclusivement aux mathématiques et à la modélisation des surfaces et obtint même un doctorat en 1977.
Paul De Casteljau était lui un mathématicien d'origine, ancien élève de la Rue d'Ulm, qui a un temps été employé par l'industrie automobile.
Aujourd'hui, les courbes de Bézier sont très utilisées en informatique.

Une Courbe de Bézier est une courbe paramétrique aux extrémités imposées avec des points de contrôle qui définissent les tangentes à cette courbe à des instants donnés.

Algorithme de Casteljau

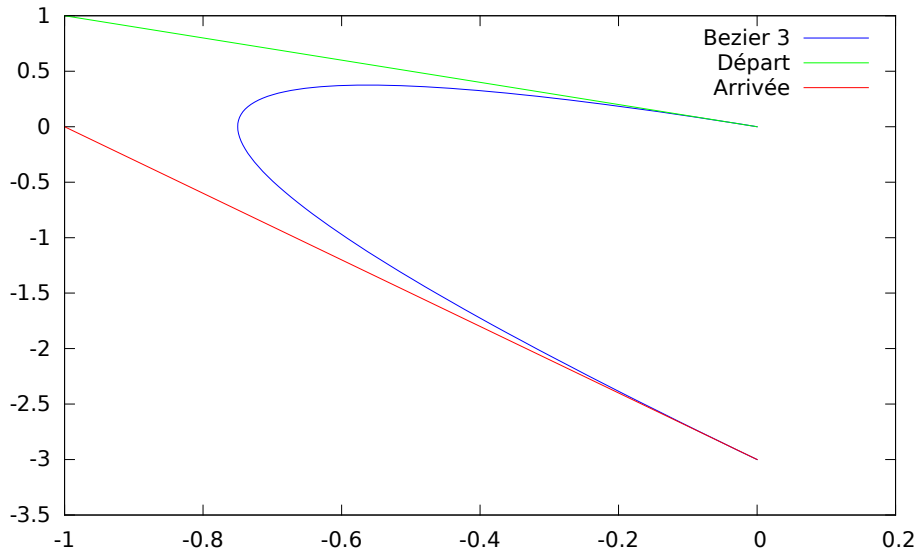
Soit t un paramètre de l'intervalle $[0,1]$ et P_1, P_2 et P_3 les trois points de contrôle.
On construit le point M_1 barycentre du système $\{(P_1,1-t),(P_2,t)\}$ et M_2 celui du système $\{(P_2,1-t),(P_3,t)\}$.
On construit ensuite le point M, barycentre du système $\{(M_1,1-t),(M_2,t)\}$.
Exprimez M comme barycentre des trois points P_1, P_2 et P_3.
Faites la construction à la main avec $t = 1/3$ par exemple puis utilisez Python.
Nous allons assimiler les points M_1, M_2 et M à des courbes paramétrées.
Ainsi $M_1(t) = (1-t)P_1 + tP_2$, $M_2(t) = (1-t)P_2 + tP_3$ puis

$$M(t) = (1-t)M_1(t) + tM_2(t) = (1-t)^2P_1 + 2t(1-t)P_2 + t^2P_3$$

Vérifiez que $M'(t) = 2(M_2(t) - M_1(t))$. Comment l'interpréter ?

Avec 4 points de contrôle

Faites une étude similaire (« à la main ») avec 4 points de contrôle.
On pourra utiliser une représentation similaire aux arbres de probabilité.
Créez une fonction **bezier3** qui trace la courbe de Bezier de degré 3 passant par une liste de 4 points de contrôle.
Avec les points (0,0),(-1,1),(-1,0),(0,-3)

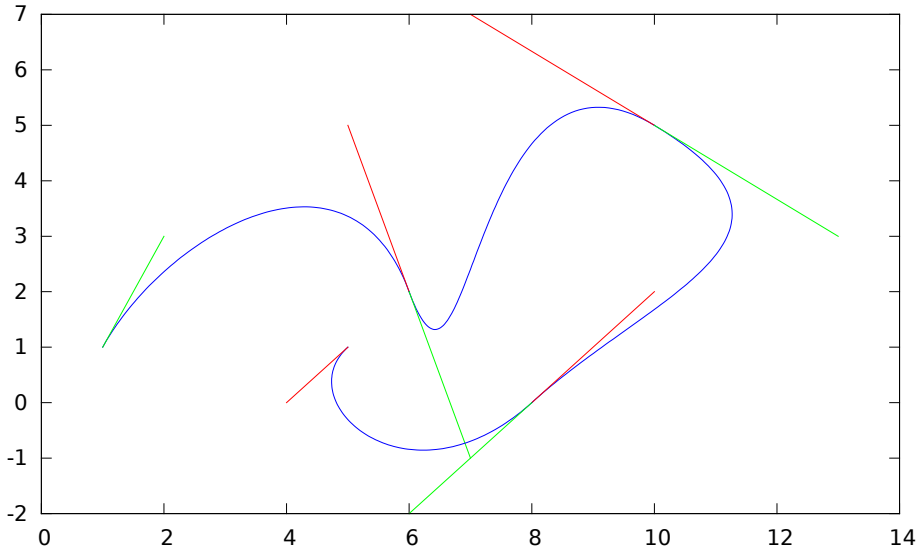


Quel est le rôle des différents points de contrôle ?

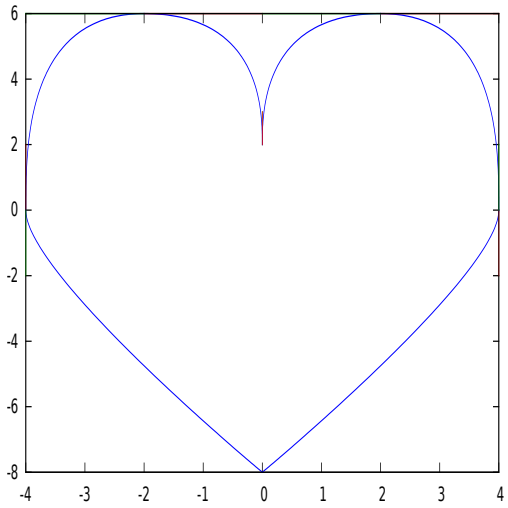
Courbe de Bézier du 3e degré avec un nombre quelconque de points de contrôle

Il est pratique de travailler avec des polynômes de degré trois pour avoir droit à des points d'inflexion.
Augmenter le nombre de points de contrôle implique a priori une augmentation du degré de la fonction polynomiale.
Pour remédier à ce problème, on découpe une liste quelconque en liste de listes de 4 points.
Cependant, cela est insuffisant pour obtenir un raccordement de classe C^1 (pourquoi ? pourquoi est-ce important d'avoir un raccordement de classe C^1 ?)
Pour assurer la continuité tout court, il faut que le premier point d'un paquet soit le dernier du paquet précédent.
Le dernier « vecteur vitesse » de la liste $[P_1,P_2,P_3,P_4]$ est $\overrightarrow{P_3P_4}$. Il faut donc que ce soit le premier vecteur vitesse du paquet suivant pour assurer la continuité de la dérivée.
Appelons provisoirement le paquet suivant $[P'_1,P'_2,P'_3,P'_4]$. On a d'une part $P'_1 = P_4$ et d'autre part $\overrightarrow{P_3P_4} = \overrightarrow{P'_1P'_2}$, i.e. $P'_2 = P_4 + \overrightarrow{P_3P_4}$.
On a donc $P'_3 = P_5$ et $P'_4 = P_6$.
Connaissant **bezier3**, construire une fonction qui trace une courbe de Bézier cubique avec un nombre quelconque de points de contrôle (on prendra un nombre pair de points pour se simplifier la vie).
Testez avec :

```
1 [(1, 1), (2, 3), (5, 5), (6, 2), (7, 7), (10, 5), (10, 2), (8, 0), (4, 0), (5, 1)]
```



C'était il y a peu la Saint-Valentin alors dessinez un cœur....



B-splines uniformes

Tout ceci est très beau mais il y a un hic : en changeant un point de contrôle, on modifie grandement la figure.
On considère m nœuds $t_0, t_1 n, \cdots, t_m$ de l'intervalle $[0,1]$.
Introduisons une nouvelle fonction :

$$\boldsymbol{S}(t) = \sum_{i=0}^{m-n-1} \boldsymbol{P}_i b_{i,n}(t) \ , \ t \in [0,1]$$

les P_i étant les points de contrôle et les fonctions $b_{j,n}$ étant définies récursivement par

$$b_{j,0}(t) = \begin{cases} 1 & \text{si} \quad t_j \leqslant t < t_{j+1} \\ 0 & \text{sinon} \end{cases}$$

et pour $n \geqslant 1$

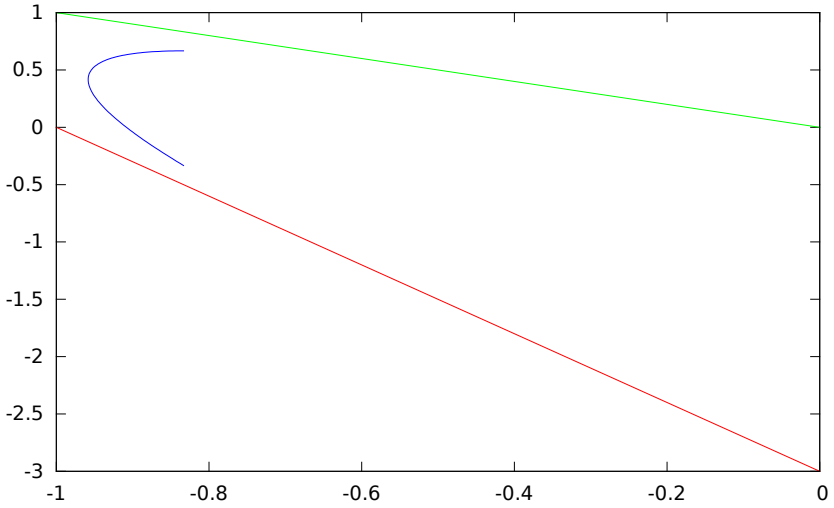
$$b_{j,n}(t) = \frac{t - t_j}{t_{j+n} - t_j} b_{j,n-1}(t) + \frac{t_{j+n+1} - t}{t_{j+n+1} - t_{j+1}} b_{j+1,n-1}(t).$$

On ne considérera par la suite que des nœuds équidistants : ainsi on aura $t_k = \frac{k}{m}$.
On parle de B-splines uniformes et on peut simplifier la formule précédente en remarquant également des invariances par translation.
À l'aide des formules précédentes, on peut prouver que dans le cas de 4 points de contrôle on obtient :

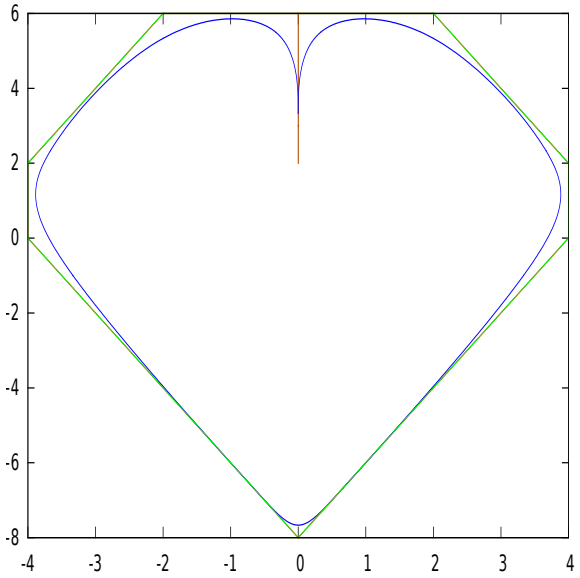
$$S(t) = \frac{1}{6}\Big((1-t)^3 P_0 + (3t^3 - 6t^2 + 4)P_1 + (-3t^3 + 3t^2 + 3t + 1)P_2 + t^3 P_3\Big)$$

Calculez $S(0)$, $S(1)$ puis $S'(0)$ et $S'(1)$: que peut-on en conclure ?
Reprenez l'étude faite avec les courbes de Bézier

```
1 bspline3([(0,0),(-1,1),(-1,0),(0,-3)])
```



Obtiendrez-vous un plus joli cœur ?



CHAPITRE 3

Méthodes itératives

Comme leur nom l'indique, les méthodes itératives (i.e. les suites définies par une relation $x_{n+1} = f(x_n)$ sont intimement liées à la notion de boucle. Nous verrons ensuite comment elles permettent d'effectuer certains calculs sur machine.

Mais ce n'est qu'un point de départ...Henri Poincaré s'est intéressé à la fin du XIX^{e} siècle à l'évolution des systèmes dynamiques en partant d'un problème de mécanique céleste : le soleil est beaucoup plus massif que les planètes, c'est pourquoi les planètes décrivent des trajectoires elliptiques autour de lui. Cependant, chaque planète est également influencée par ses congénères mais dans une mesure bien moindre : les trajectoires elliptiques sont certes déformées, mais de manière extrêmement lente. Que va-t-il se passer dans le long terme : ces micro-changements vont-ils avoir une grande influence ou bien rester négligeables ? L'étude des systèmes dynamiques était née : on étudie un système très simple a priori, décrit par un petit nombre de paramètres dont la loi dévolution est déterminée, et on étudie son évolution à long terme : la simplicité des systèmes cache parfois une incroyable complexité de l'évolution à long terme.

Cette théorie a été popularisée par la médiatisation de certains résultats de la théorie du chaos, de l'étude des fractales. En informatique, elle a influencé par exemple certains modes de compression d'images.

Elle pose aussi le problème de la prédictibilité : le futur peut-il être déduit du présent ? C'est un domaine passionnant, riche mathématiquement et en lien avec la physique, la biologie, les sciences de l'ingénieur, l'informatique... C'est enfin un domaine mathématique très vivant : l'un des gagnants de la médaille Fields 2014 est en effet le franco-brésilien **Artur AVILA** dont le domaine de recherche est justement lié aux systèmes dynamiques.

1 Une boucle sous toutes ses formes

Généralité

Regardons une simple boucle :

```
1  Algorithme Une boucle
2  r ← r0
3  Pour k de 1 à n Faire
4  |  r ← f(r)
5  FinPour
6  Retourner r
```

On construit ainsi une *suite* de valeurs prises par la variable r qui varie avec l'incrément k. Si on note r_k la valeur de la variable r lorsque l'incrément vaut k :

$$r_{k+1} = f(r_k)$$

L'algorithme boucle peut d'ailleurs être écrit récursivement :

```
1  Fonction boucle(r : type 1 n : entier naturel) : type 2
2  Si n == 0 Alors
3  |  Retourner r
4  Sinon
5  |  Retourner boucle(f(r),n-1)
6  FinSi
```

Ce qui peut s'écrire de manière plus concise à l'aide d'un *pliage* :

```
1  def boucle(f, r0, n) :
2      functools.reduce(lambda r, k: f(r), r0, range(1, n + 1))
```

et pour avoir la liste de toutes les valeurs intermédiaires on remplace `foldl` par `scanl`.
Le programmeur voudrait savoir si sa boucle va effectivement renvoyer ce qui est prévu dans sa spécification et surtout en combien de temps : s'il s'agit de calculer la vitesse d'une voiture en fonction de la distance qui la sépare de la voiture qui est en face, il faut faire vite.

Complexité de la somme

Nous allons étudier le plus classique des problèmes : il s'agit de calculer la somme des entiers de 1 à un entier naturel n non nul donné.
La *spécification* est immédiate : on crée une fonction ayant pour argument l'entier naturet n et renvoyant l'entier naturel correspondant à la somme des entiers de 1 à n.
Il reste à s'occuper de la *réalisation*.

Procédure et processus

Appelons $S(n)$ la somme des entiers de 1 à n avec $n \in \mathbb{N} \setminus \{0\}$.
On a $S(1) = 1$ et $S(n) = n + S(n-1)$.

```
1  def s1(n) :
2      assert n > 0, "l'argument doit être un entier naturel non nul"
3      if n == 1 :
4          return 1
5      return n + s1(n - 1)
```

Que se passe-t-il quand on calcule `s1(5)` ? On utilise le modèle de *substitution* :

```
s1(5)
5 + s1(4)
5 + (4 + s1(3))
5 + (4 + (3 + s1(2)))
5 + (4 + (3 + (2 + s1(1))))
5 + (4 + (3 + (2 + 1)))
5 + (4 + (3 + 3))
5 + (4 + 6)
5 + 10
15
```

On commence par quatre expansions suivies de cinq réductions.
La *procédure* est récursive car syntaxiquement `s1` apparaît dans sa propre définition (`return s1(n - 1) + n`).
Le *processus* est également récursif : les évaluations sont retardées. Intuitivement, cela est le cas car on progresse du cas compliqué vers le cas simple : on « descend » ici de n vers 1. L'interpréteur doit *empiler* les résultats des expansions tant qu'il ne peut pas les réduire. La taille de la pile nécessaire étant proportionnelle à l'argument n, on dit qu'il s'agit d'un processus linéairement récursif.

On peut avoir une autre idée : partir du cas simple et « monter » vers le cas compliqué (on parle dans ce cas habituellement d'*induction*).
Ici, la somme vaut 0 puis on ajoute 1, puis on ajoute au résultat 2, puis on ajoute au résultat 3, etc jusqu'à atteindre n.
Techniquement, on entretient un *accumulateur* de termes qui évolue et un compteur qui va évoluer de 1 jusque n.
L'accumulateur et le compteur évoluent selon la règle :

```
accumulateur ← accumulateur + compteur
compteur ← compteur + 1
```

la somme cherchée étant la valeur de l'accumulateur lorsque le compteur a dépassé n.

```python
def s2(n) :
    def s2rec(acc, compt) :
            if compt > n :
                return acc
            return s2rec(acc + compt, compt + 1)
    return s2rec(0, 1)
```

La procédure `s_iter` est syntaxiquement récursive car elle est définie à partir d'elle-même mais le processus n'est plus récursif ! les évaluations ne sont plus retardées :

```
s2(5)
s_iter(0 , 1)
s_iter(1 , 2)
s_iter(3 , 3)
s_iter(6 , 4)
s_iter(10, 5)
s_iter(15, 6)
15
```

Cette fois-ci, il n'y a plus de phase d'expansion suivie d'une phase de réduction. À chaque étape, il suffit de garder une trace des états de `acc` et `compt`. On parle alors de processus *itératif*.
En général, un processus itératif est un processus pouvant être décrit par un nombre fixe de *variables d'état* et par une règle fixe qui décrit l'évolution de chaque variable à chaque étape du processus ainsi qu'un éventuel test d'arrêt qui indique sous quelles conditions le processus doit s'arrêter.
Ici, le nombre d'étapes nécessaire est proportionnel à la taille de l'argument n. On parle de processus itératif linéaire.

En fait, une autre manière de distinguer un processus itératif d'un processu récursif est de remarquer que si le processus itératif est interrompu, on peut le relancer en donnant comme arguments les valeurs des variables au moment de l'interruption. Ce n'est pas possible pour un processus récursif car l'interpréteur garde des informations « cachées » dans une pile non accessibles par les variables du programme.
Attention ! Ici, les deux procédures sont syntaxiquement récursives mais les processus sont de nature différentes. Cependant, dans des langages comme Pascal, Ada, Python, un processus itératif décrit par une procédure récursive consommera malgré tout autant de mémoire qu'un processus récursif. C'est pourquoi ces langages utilisent des *boucles* (`for`, `while`).
On peut utiliser un pliage :

```
def s3(n) :
    return functools.reduce(lambda x, y: x + y, range(1, n + 1), 0)
```

une boucle inconditionnelle :

```
def s4(n) :
    s = 0
    for k in range(1,n+1) :
        s += k
    return s
```

ou conditionnelle :

```
def s5(n) :
    s, k = 0, 1
    while k <= n :
        s += k
        k += 1
    return s
```

...ou directement la fonction `sum` :

```
sum(range(100001))
```

Faisons une petite expérience :

```
In [12]: %timeit s1(n)
100 loops, best of 3: 3.94 ms per loop

In [13]: %timeit s2(n)
100 loops, best of 3: 5.14 ms per loop

In [14]: %timeit s3(n)
1000 loops, best of 3: 1.18 ms per loop

In [15]: %timeit s4(n)
1000 loops, best of 3: 564 Âs per loop

In [16]: %timeit sum(range(n+1))
10000 loops, best of 3: 185 Âs per loop

In [17]:
In [18]: %timeit s5(n)
1000 loops, best of 3: 1.04 ms per loop
```

Python ne gère pas les processus itératifs écrits récursivement.

Itération non linéaire : Fibonacci

Tout le monde connaît l'exemple classique de la suite des nombres de FIBONACCI dans laquelle chaque nombre est la somme des deux précédents :

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

On pense alors naturellement à :

```
1  def fib1(n) :
2      assert n >= 0, "l'indice doit être un entier naturel !"
3      if n <= 1 :
4          return 1
5      return fib1 (n - 1) + fib1 (n - 2)
```

Mais...

```
1  In [31]: %timeit fib1(10)
2  10000 loops, best of 3: 25.4 Âs per loop
3
4  In [32]: %timeit fib1(20)
5  100 loops, best of 3: 3.12 ms per loop
6
7  In [33]: %timeit fib1(25)
8  10 loops, best of 3: 35.1 ms per loop
9
10 In [34]: %timeit fib1(30)
11 1 loop, best of 3: 408 ms per loop
12
13 In [35]: %timeit fib1(35)
14 1 loop, best of 3: 4.38 s per loop
```

Qu'en pensez-vous ?

On peut avoir une autre idée en utilisant deux variables :

```
suivant ← suivant + courant
courant ← suivant
```

Construisez un processus itératif donnant plus efficacement le n-eme nombre de FIBONACCI (avec et sans boucle).

```
1  In [17]: %timeit fib1(35)
2  1 loop, best of 3: 4.32 s per loop
3
4  In [18]: %timeit fib2(35)
5  100000 loops, best of 3: 6.14 Âs per loop
6
7  In [19]: %timeit fib3(35)
8  100000 loops, best of 3: 1.88 Âs per loop
9
10 In [20]: %timeit fib2(70)
11 100000 loops, best of 3: 12.4 Âs per loop
12
13 In [21]: %timeit fib3(70)
14 100000 loops, best of 3: 3.71 Âs per loop
```

Commentaires ?

2 L'algorithme de Babylone

Tablette YBC 7289
XVII[e] avant JC

Il y a presque 4000 ans, les petits Babyloniens calculaient la racine carrée de 2 avec 6 bonnes décimales en utilisant un algorithme vu précédemment (cf section 2.2.1 page 42).
Il s'agit de reprendre ce qui a été vu précédemment avec par exemple :

$$r_0 = 1.0 \quad \text{et} \quad f \ : \ x \mapsto (x + 2/x) * 0.5$$

ou si vous préférez :

```
1 Algorithme Babylone
2 r ← r0
3 Pour k de 1 à n Faire
4 |  r ←(r+a/r)*0.5
5 FinPour
6 Retourner r
```

Premier problème : est-ce que la suite des valeurs calculées par la boucle converge vers $\sqrt{2}$?
Nous aurons besoin de deux théorèmes que nous admettrons et dont nous ne donnerons que des versions simplifiées.

Théorème 3 - 1

Théorème de la limite monotone
Toute suite croissante majorée (ou décroissante minorée) converge

Un théorème fondamental est celui du point fixe. Il faudrait plutôt parler des théorèmes du point fixe car il en existe de très nombreux avatars qui portent les noms de mathématiciens renommés : Banach, Borel, Brouwer, Kakutani, Kleene,... Celui qui nous intéresserait le plus en informatique est celui de Knaster-Tarski. On peut en trouver une présentation claire dans Dowek [2010].
Ils donnent tous des conditions d'existence de points fixes (des solutions de $f(x) = x$ pour une certaine fonction).
Nous nous contenterons pour l'instant de la version *light* vue au lycée.

Théorème 3 - 2

Théorème du point fixe : version (très) édulcorée du lycée
Soit I un intervalle fermé de $\mathbb{R}$, soit f une fonction de I vers I et soit (r_n) une suite d'éléments de I telle que $r_{n+1} = f(r_n)$ pour tout entier naturel n.
SI (r_n) est convergente ALORS sa limite est UN *point fixe* de f appartenant à I.

Cela va nous aider à étudier notre suite définie par $r_{n+1} = f(r_n) = \frac{r_n + \frac{2}{r_n}}{2}$ et $r_0 = 1$.

Recherche

Démontrez que pour tout entier naturel non nul n, on a $r_n \geqslant \sqrt{2}$ puis que la suite est décroissante.
Démontrez que $\sqrt{2}$ est l'unique point fixe positif de f et conclure.

Deuxième problème : quelle est la vitesse de convergence ? Ici, cela se traduit par « combien de décimales obtient-t-on en plus à chaque itération ? ».

Recherche

Calculez $(r_{n+1} - \sqrt{2})$ en fonction de $(r_n - \sqrt{2})$ et essayez de répondre à la question précédente à l'aide de ce résultat.

On peut donc être sûr a priori que le résultat de notre boucle donnera le résultat spécifié (avec la convergence). Le nombre d'itérations pour une précision donnée peut lui aussi être calculé à l'avance en étudiant la vitesse.

Essayons d'être un peu plus rigoureux et général en introduisant la notion d'ordre d'une suite :

Définition 3 - 1

Ordre d'une suite - Constante asymptotique d'erreur

Soit (r_n) une suite convergeant vers ℓ. S'il existe un entier $k > 0$ tel que :

$$\lim_{n\to+\infty} \frac{|r_{n+1} - \ell|}{|r_n - \ell|^k} = C$$

avec $C \neq 0$ et $C \neq +\infty$ alors on dit que (r_n) est d'ordre k et que C est la constante asymptotique d'erreur.

Recherche

Déterminez par exemple l'ordre et la constante asymptotique d'erreur de la suite de Babylone donnant une approximation de $\sqrt{2}$.

Si l'on connaît l'ordre d'une suite, on va pouvoir prévoir l'évolution de la précision atteinte au fur et à mesure de l'avance de la boucle.
Ainsi, pour une suite d'ordre k avec comme constante asymptotique d'erreur C, on a pour n suffisamment grand :

$$|r_{n+1} - \ell| \approx C|r_n - \ell|^k$$

Recherche

Quel est le rapport entre $\log_{10} x$ et le nombre de décimales d'un nombre plus petit que 1 en base 10 ?

En posant $d_n = -\log_{10}|r_n - \ell|$ et $D = -\log_{10} C$ on obtient :

$$d_{n+1} = \ldots$$

Recherche

Que peut-on en conclure ?

Une application : on peut ainsi calculer rapidement sur un petit système embarqué (voire un *smart phone*) la vitesse de la voiture en fonction de la distance la séparant de la voiture qui la précède.

Application *safety sight* sur *smart phone*

3 Méthodes itératives pour résoudre une équation

Résoudre une équation du premier ou du second degré, vous savez faire...Mais dans les autres cas ?
Dans cette section, nous nous intéresserons à la résolution d'une équation du type $\varphi(x) = \ell$ où φ est une fonction de $\mathbb{R}$ dans lui-même.
On se ramènera même à une équation de la forme $f(x) = 0$ en posant $f(x) = \varphi(x) - \ell$.

Dichotomie (ou méthode de bi-section)

On veut résoudre une équation du type $f(x) = 0$ sur un intervalle I avec f une fonction continue qui change de signe sur un intervalle $[a, b]$ inclus dans I.
On sait qu'il existe au moins une solution α de notre équation sur $[a, b]$: pourquoi ?
Soit m le centre de l'intervalle. Trois cas se rencontrent :

— si $f(m) = 0$ alors la vie est belle ;

— si $f(m)$ a le même signe que $f(a)$ alors $\alpha \in [m, b]$ (pourquoi ?) ;

— si $f(m)$ a le même signe que $f(b)$ alors $\alpha \in [a, m]$ (pourquoi ?) ;

Recherche

On itère ce mécanisme jusqu'à obtenir un intervalle d'amplitude correspondant à la précision demandée sur α : pourquoi cela suffit-il ? Quel est l'avantage de cette méthode informatiquement parlant ? Quel est son principal désavantage ?

Il est à noter que les divisions par successives peuvent engendrer des problèmes selon la norme IEEE 754 (cf Goulard [2014]).
On parle de *dichotomie* du grec $\delta\iota\chi\sigma\tau\sigma\mu\iota\alpha$ qui signifie *division en deux parties égales.*

Ordre d'une suite et formules de Taylor

Considérons une suite convergente générée par un procédé itératif $x_{n+1} = \varphi(x_n)$. Supposons que φ est de classe C^1 au voisinage de sa limite ℓ. Si l'on remarque que $x_{n+1} = \varphi(x_n) = \varphi(\ell + (x_n - \ell))$ alors :

$$x_{n+1} = \varphi(\ell) + (x_n - \ell)\varphi'(\ell) + O(|x_n - \ell|^2)$$

On en déduit que :

$$x_{n+1} - \ell = (x_n - \ell)\varphi'(\ell) + O(|x_n - \ell|^2)$$

Recherche

On peut alors en conclure des choses sur l'ordre de la suite selon que $\varphi'(\ell)$ est nul ou pas : lesquelles ?

Méthode de Newton-Raphson-Halley-etc.

La méthode de résolution des équations numériques que nous allons voir à présent a été initiée par Isaac Newton vers 1669 sur des exmples numériques mais la formulation était fastidieuse. Dix ans plus tard, Joseph Raphson met en évidence une formule de récurrence. En 1694, la formule est encore améliorée par Edmund Halley, l'homme de la comète. Un siècle plus tard, Mouraille et Lagrange étudient la convergence des approximations successives en fonction des conditions initiales par une approche géométrique. Cinquante ans plus tard, Fourier et Cauchy s'occupe de la rapidité de la convergence.

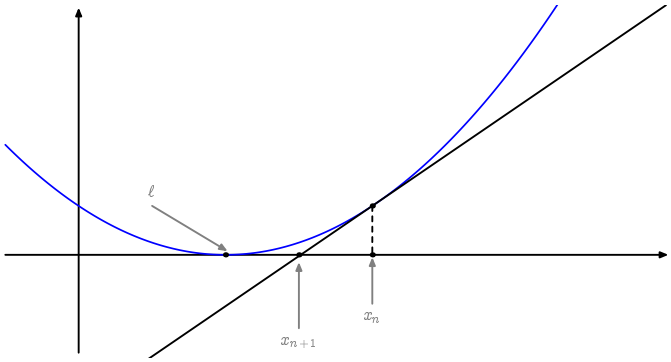
Approche intuitive graphique

— On part d'un nombre quelconque x_0 ;

— à partir de x_0, on calcule un nouveau nombre x_1 de la manière suivante (voir figure) : on trace la tangente au graphe de f au point d'abscisse x_0, et on détermine le point d'intersection de cette tangente avec l'axe des abscisses. On appelle x_1 l'abscisse de ce point d'intersection ;

— et on recommence : on calcule un nouveau nombre x_2 en appliquant le procédé décrit au point 2 où l'on remplace x_0 par x_1 ;

— etc.



À partir de cette description graphique de la méthode de Newton, trouver la formule, notée (1), donnant x_1 en fonction de x_0, puis x_{n+1} en fonction de x_n.
Quelles hypothèses doit-on faire sur f et les x_n pour que la formule ait un sens ?

Approche intuitive « taylorienne »

Nous voulons résoudre l'équation $f(x) = 0$. Utilisons à nouveau la ruse :

$$f(x_{n+1}) = f(x_n + (x_{n+1} - x_n)) = f(x_n) + (x_{n+1} - x_n)f'(x_n) + O((x_{n+1} - x_n)^2)$$

Intuitivement, pour que $f(x_{n+1})$ soit de plus en plus petit, il serait pratique que la partie linéaire de cette expression, $f(x_n) + (x_{n+1} - x_n)f'(x_n)$, soit la plus petite possible, c'est-à-dire nulle et on obtient à nouveau :

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

On remarque alors que si la suite converge vers une limite ℓ et si $f'(\ell)$ est non nul, alors on obtient bien $f(\ell) = 0$.
Maintenant, posons $\varphi(x) = x - \frac{f(x)}{f'(x)}$ et supposons que f soit de classe C^2 sur un intervalle ouvert V tel que l'équation $f(x) = 0$ admette une unique solution d'ordre 1 (c'est-à-dire que $f(\ell) = 0$ mais $f'(\ell) \neq 0$).
La méthode de Newton revient à construire la suite définie par $x_{n+1} = \varphi(x_n)$ avec $x_0 \in V$.

Recherche

Vérifiez que φ est dérivable sur V et que

$$\varphi'(x) = \frac{f(x)f''(x)}{(f'(x))^2}$$

et concluez en utilisant les résultats de la section 3.3.2 page ci-contre

À retenir

Il faut bien comprendre que nos approches sont un peu ole ole compte-tenu de nos outils mathématiques limités...
On retiendra que x_0 a besoin d'être suffisamment proche de la solution pour que l'équation $f(x) = 0$ admette une unique solution et que $f'(\ell)$ soit non nul. En fait, on pourrait montrer que la méthode marche encore si ℓ est une solution d'ordre 2.
Il faut aussi que la fonction f soit suffisamment régulière. Bref, il ne faut pas trop bricoler au hasard...

Étude de la suite associée à l'équation x^3-2x-5=0.

On veut résoudre l'équation $x^3 - 2x - 5 = 0$ par la méthode de Newton-Raphson. On note f la fonction $x \mapsto x^3 - 2x - 5$. C'est en fait sur cet exemple que Newton a raisonné.

Recherche

1. Montrez rapidement que l'équation $f(x) = 0$ admet une unique solution α sur $\mathbb{R}$. Montrez que $2 < \alpha < 3$.
2. Déterminez la fonction φ telle que $x_{n+1} = \varphi(x_n)$, la suite (x_n) étant celle décrite au paragraphe précédent en prenant $x_0 = 3$.
3. Étudiez le sens de variation de la fonction φ puis celui de φ' et déduisez-en que $[\alpha, 3]$ est stable par φ et que φ est strictement croissante sur $[\alpha, 3]$.
4. Que pouvez-vous en déduire sur la convergence de la suite (x_n) ?

Test d'arrêt

Afin de construire un algorithme donnant une approximation d'une solution d'une équation numérique par la méthode de Newton-Raphson, il faudrait déterminer un test d'arrêt c'est-à-dire savoir à partir de quel rang n $|x_n - \alpha|$ restera inférieur à une valeur donnée.
Il suffit de remarquer que $f(x_n) = f(x_n) - f(\alpha)$. Nous supposerons f de classe C^2 sur un « bon » voisinage I de α (ce critère nous échappe encore à notre niveau). Alors f est en particulier dérivable en α donc

$$f(x_n) = f(x_n) - f(\alpha) \sim f'(\alpha) \times (x_n - \alpha)$$

C'est-à-dire, puisque $f'(\alpha)$ est supposé non nul :

$$x_n - \alpha \sim \frac{f(x_n)}{f'(\alpha)}$$

Or f étant de classe C^2, on a f' continue et non nulle en α donc $f'(\alpha) \sim f'(x_n)$.
Finalement

$$x_n - \alpha \sim \frac{f(x_n)}{f'(x_n)} = x_n - x_{n+1}$$

Nous choisirons donc comme test d'arrêt $\left|\frac{f(x_n)}{f'(x_n)}\right| < p$ avec p la précision choisie.
Il ne reste plus qu'à écrire l'algorithme.

4 Sommes

« What is one and one and one and one and one and one and one and one ? »
« I don't know » said Alice. « I lost count ».
« She can't do Addition. »

Alice in wonderland - Lewis Carroll

Notation

Si on écrit $1 + 2 + 3 + 4 + \cdots + (n-1) + n$, on comprend que les $\cdots$ signifient qu'ils doivent être remplacés selon le motif induit par les termes les entourant.
On peut être amené à étudier des sommes plus générales : $a_0 + a_1 + \cdots + a_n$, chaque **terme** a_k de la somme étant défini d'une manière quelconque.
Si l'on écrit la somme avec des $\cdots$, il faut que le motif de construction soit présenté de manière suffisamment claire pour être induit :

$$1 + 4 + 9 + 22 + 53 + 128 + 309 + \ldots$$

n'est pas évident à comprendre mais :

$$1 + 4 + (2 \times 4 + 1) + (2 \times 9 + 4) + \cdots + (2a_{n-1} + a_{n-2})$$

est plus explicite.

Joseph FOURIER
(1768-1830)

En 1820, le mathématicien Joseph FOURIER [a] introduit le fameux sigma majuscule. Citons-le :

> *Le signe* $\sum_{i=1}^{+\infty}$ *indique que l'on doit donner au nombre entier* i *toutes ses valeurs 1, 2, 3,..., et prendre la somme des termes.*

Notez bien, en tant qu'informaticien(ne), que le i joue ici le rôle d'une variable locale et aurait très bien pu porter un autre nom sans changer le résultat.
Depuis, on a généralisé l'emploi des bornes afin de définir les indices à partir de certaines propriétés. On peut par exemple écrire les sommes suivantes avec ou sans bornes explicitement spécifiées :

$$\sum_{\substack{1\leqslant k<100\\ k \text{ impair}}} k^2 = \sum_{k=0}^{49} (2k+1)^2$$

Cette formulation est particulièrement adaptée aux changements d'indices :

$$\sum_{1\leqslant k\leqslant n} a_k = \sum_{1\leqslant k+1\leqslant n} a_{k+1} = \sum_{k=1}^{n} a_k = \sum_{i=0}^{n-1} a_{i+1}$$

La dernière forme demande un peu de réflexion alors que la deuxième beaucoup moins.
L'informaticien canadien Kenneth IVERSON, prix TURING en 1979 et inventeur des langages APL et J, introduit une notation fort pratique mais peu utilisée en France, nommée **crochets d'Iverson**. Soit P une certaine propriété :

$$[P] = \begin{cases} 1 & \text{si P est vraie} \\ 0 & \text{sinon} \end{cases}$$

Par exemple :

$$\sum_{\substack{1\leqslant k<100\\ k \text{ impair}}} k^2 = \sum_{k} k^2 \,[1 \leqslant k < 100] \times [k \text{ impair }]$$

Somme, boucle et récurrence

Une somme suggère naturellement l'emploi d'une boucle « pour ». En effet $\sum_{k=1}^{n} a_k$ se lit « somme des a_k **pour** k variant **de** 1 **à** n ».
On peut également penser à une somme en terme de suite. Si on note $S_n = \sum_{k=0}^{n} k^2$, alors :

$$S_0 = 0; \quad \forall n > 0,\ S_n = S_{n-1} + n^2$$

Cela nous rappelle d'ailleurs qu'une **suite** est une **fonction** de $\mathbb{N}$ dans tout autre ensemble (ici $\mathbb{N}$ lui-même).

Manipulation de sommes

La principale difficulté du calcul avec des sommes est d'obtenir par diverses manipulation une expression plus simple. Tout dérive le plus souvent des trois lois suivantes :

Distributivité $\sum_{k\in K} \lambda a_k = \lambda \sum_{k\in K} a_k$ (λ ne dépendant pas de k) ;

Associativité $\sum_{k\in K} (a_k + b_k) = \sum_{k\in K} a_k + \sum_{k\in K} b_k$;

Commutativité $\sum_{k\in K} a_k = \sum_{k\in K} a_{\pi(k)}$ ($\pi \in S_{\operatorname{Card} K}$).

Par exemple, nous allons prouver un résultat fort utile concernant les suites géométriques.
Nous voudrions calculer $S_n = \sum_{k=0}^{n} x^k$ pour tout réel $x \neq 1$.
Or

$$S_n + x^{n+1} = x^0 + \sum_{1\leqslant k\leqslant n+1} x^k = x^0 + \sum_{0\leqslant k-1\leqslant n} x^k = x^0 + \sum_{0\leqslant k-1\leqslant n} x^{(k-1)+1}$$

En posant $i = k - 1$ on obtient :

a. Nous aurions pu ne pas étudier les sommes, ne pas compresser les images et le son car FOURIER échappa de peu à la guillotine durant la Terreur : http://www.youtube.com/watch?v=MwGD13BnH-o

$$S_n + x^{n+1} = x^0 + \sum_{0 \leqslant i \leqslant n} x^{i+1} = x^0 + x \sum_{0 \leqslant i \leqslant n} x^i = 1 + xS_n$$

d'où :

$$S_n(1-x) = 1 - x^{n+1}$$

Finalement, comme $x \neq 1$, on obtient :

$$\sum_{k=0}^{n} x^k = \frac{1-x^{n+1}}{1-x}$$

Sommes multiples

Nous voulons exprimer $S = a_1b_1 + a_1b_2 + a_1b_3 + a_2b_1 + a_2b_2 + a_2b_3 + a_3b_1 + a_3b_2 + a_3b_3$ avec le symbole $\sum$:

$$\begin{aligned}
S &= \sum_{i,j} a_i b_j \, [1 \leqslant i, j \leqslant 3] \\
&= \sum_{i,j} a_i b_j \, [1 \leqslant i \leqslant 3] \, [1 \leqslant j \leqslant 3] \\
&= \sum_{i} \left(\sum_{j} a_i b_j \, [1 \leqslant i \leqslant 3] \, [1 \leqslant j \leqslant 3] \right) \\
&= \sum_{i} a_i \, [1 \leqslant i \leqslant 3] \left(\sum_{j} b_j \, [1 \leqslant j \leqslant 3] \right) \\
&= \left(\sum_{i} a_i \, [1 \leqslant i \leqslant 3] \right) \left(\sum_{j} b_j \, [1 \leqslant j \leqslant 3] \right) \\
&= \left(\sum_{i=1}^{3} a_i \right) \left(\sum_{j=i}^{3} b_j \right)
\end{aligned}$$

Nous voulons cette fois exprimer $S' = a_1b_1 + a_1b_2 + a_1b_3 + a_2b_2 + a_2b_3 + a_3b_3$ avec le symbole $\sum$:

$$\begin{aligned}
S' &= \sum_{i,j} a_i b_j \, [1 \leqslant i \leqslant j \leqslant 3] \\
&= \sum_{i,j} a_i b_j \, [1 \leqslant i \leqslant 3] \, [i \leqslant j \leqslant 3] \\
&= \sum_{i} \left(\sum_{j} a_i b_j \, [1 \leqslant i \leqslant 3] \, [i \leqslant j \leqslant 3] \right) \\
&= \sum_{i} a_i \, [1 \leqslant i \leqslant 3] \left(\sum_{j} b_j \, [i \leqslant j \leqslant 3] \right) \\
&= \sum_{i=1}^{3} a_i \left(\sum_{j=i}^{3} b_j \right)
\end{aligned}$$

Nous verrons d'autres exemples en exercice.

5 Sommes infinies (séries)

Les dangers des ... mal maîtrisés

À quoi peut bien être égal $S = \frac{1}{2^0} + \frac{1}{2^1} + \frac{1}{2^2} + \cdots$?
Multiplions les deux membres par 2, nous obtenons :
$2S = 2 + \frac{1}{2^0} + \frac{1}{2^1} + \frac{1}{2^2} + \cdots = 2 + S$
donc $S = 2$. Waouh, magique !
On recommence avec :
$T = 2^0 + 2^1 + 2^2 + \cdots$

alors :
$2T = 2^1 + 2^2 + 2^3 + \cdots = T - 2^0 = T - 1$ donc $T = -1$.
Waouh !...Euh...Quoique, même réveillé brutalement en plein somme il peut sembler bizarre que la somme de nombres strictement positifs donne un nombre strictement négatif...
Il va falloir prendre des précautions.

Séries

Définition 3 - 2

Soit u une suite réelle définie sur $I = \{n \in \mathbb{N} \,|\, n \geq n_0\}$. On appelle série de terme général u_n la suite S définie par $S_n = \sum_{k=n_0}^{n} u_k$. Si la suite S converge, on dit que la série de terme général u_n converge et on note $\sum_{k=n_0}^{+\infty} u_k = \lim_{n\to+\infty} S_n$. Une série qui ne converge pas est dite divergente, dans ce cas l'écriture $\sum_{k=n_0}^{+\infty} u_k$ **n'a pas de sens**.
La série de terme général u_n est notée $\left(\sum u_n\right)$.

Si on nous demande d'étudier la série $\left(\sum \frac{1}{n^2}\right)$, on nous demande d'étudier la suite S définie par $S_n = \sum_{k=1}^{n} \frac{1}{k^2}$ avec évidemment $n \in \mathbb{N}^*$.

Remarque

Si la suite u est définie à partir de n_0, rien ne nous empêche de changer les notations pour travailler avec la même suite définie à partir du rang 0 ou du rang 1 ou ... En effet, condidérons la suite v définie par $v_n = u_{n+n_0}$, il est cair que la suite v est definie sur $\mathbb{N}$ et, qu'étudier la suite v, c'est étudier la suite u. La conséquence de ceci est que, dans ce qui suit, les suites qui interviendront seront, suivant les besoins, définie à partir de $n = 0$ ou $n = 1$ ...

Remarque

S_n est appelé une somme partielle.

Théorème 3 - 3

Si la série $\left(\sum u_n\right)$ converge alors on a $\lim_{n\to+\infty} u_n = 0$

La démonstration est très simple.
Considérons $S_n = \sum_{j=0}^{n} u_j$. Par hypothèse $\lim_{n\to+\infty} S_n = \lim_{k\to+\infty} S_k = \ell$. Si nous remplaçons k par $n-1$, nous obtenons

$$\lim_{n\to+\infty} S_n = \lim_{n-1\to+\infty} S_{n-1} = \ell$$

mais comme $S_n - S_{n-1} = u_n$ on en déduit

$$\lim_{n\to+\infty} u_n = \lim_{n\to+\infty} (S_n - S_{n-1}) = \ell - \ell = 0$$

Remarque

La réciproque est totalement fausse, ce n'est pas parce que le terme général d'une série a pour limite zéro que cette série converge.

Série harmonique

Nous allons démontrer que la série $\left(\sum \frac{1}{n}\right)$, qui porte le nom de série harmonique, diverge. Nous avons donc $u_n = \dfrac{1}{n}$ avec $n \in \mathbb{N}^*$ évidemment et

$$S_n = \sum_{j=1}^{n} u_j = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

La suite S est une suite de termes strictement positifs et comme $S_{n+1} - S_n = \dfrac{1}{n+1} > 0$ nous sommes assurés que la suite S est strictement croissante : ainsi, soit elle diverge vers $+\infty$, soit elle converge vers une limite réelle ℓ.

Calculons $S_{2n} - S_n$.

$$\begin{aligned} S_n &= \sum_{j=1}^{n} u_j = 1 + \frac{1}{2} + \cdots + \frac{1}{n} \\ S_{n=2n} &= \sum_{j=1}^{2n} u_j = 1 + \frac{1}{2} + \cdots + \frac{1}{n} + \frac{1}{n+1} + \frac{1}{n+2} + \cdots + \frac{1}{n+n} \\ S_{2n} - S_n &= \frac{1}{n+1} + \frac{1}{n+2} + \cdots + \frac{1}{n+n} \end{aligned}$$

La dernière somme comporte exactement n termes tous $\geqslant \dfrac{1}{2n}$. On obtient alors

$$S_{2n} - S_n \geqslant \frac{1}{2n} + \frac{1}{2n} + \cdots + \frac{1}{2n} = \frac{n}{2n} = \frac{1}{2}$$

Supposons que (S_n) converge vers ℓ, alors $\lim\limits_{n\to+\infty} S_n = \lim\limits_{n\to+\infty} S_{2n} = \ell$.
En reportant dans l'inégalité obtenue précédemment : $\ell - \ell \geqslant \frac{1}{2}$ ce qui est absurde donc (S_n) diverge vers $+\infty$, doucement, mais sûrement.

Séries télescopiques

Intéressons-nous à la convergence de la série de terme général $u_n = \dfrac{1}{n^2-1}$; il est évident que u_n n'est définie que pour $n \geq 2$. Nous allons calculer $\lim\limits_{n\to+\infty} S_n$ avec $S_n = \sum\limits_{k=2}^{n} u_k$ en utilisant la remarque indispensable suivante :

$$\frac{1}{x^2-1} = \frac{1}{2}\left(\frac{1}{x-1} - \frac{1}{x+1}\right)$$

Cela donne :

$$\begin{aligned} S_n &= \sum_{k=2}^{n} u_k = \sum_{k=2}^{n} \frac{1}{k^2-1} = \frac{1}{2}\sum_{k=2}^{n}\left(\frac{1}{k-1} - \frac{1}{k+1}\right) \\ S_n &= \frac{1}{2}\left(\sum_{k=2}^{n}\frac{1}{k-1} - \sum_{k=2}^{n}\frac{1}{k+1}\right) \end{aligned}$$

Dans la première sommation remplaçons k par $j+1$ et dans la deuxième k par $j-1$:

$$S_n = \frac{1}{2}\left(\sum_{j=1}^{n-1}\frac{1}{j} - \sum_{j=3}^{n+1}\frac{1}{j}\right) = \frac{1}{2}\left(1 + \frac{1}{2} - \frac{1}{n} - \frac{1}{n+1}\right)$$

On obtient alors

$$\lim_{k\to+\infty} S_n = \frac{3}{4}$$

Il est alors permis d'écrire que $\sum\limits_{n=2}^{+\infty} u_n = \sum\limits_{n=2}^{+\infty} \dfrac{1}{n^2-1}$ a un sens ou existe et vaut $\dfrac{3}{4}$. On dit aussi que la somme de la série est $\dfrac{3}{4}$.

Remarque

Il ne faut pas croire que l'on est capable de trouver la somme de toute série convergente, c'est plutôt rare, par contre, on est très souvent capable de dire si une série converge ou non mais nous ne présenterons pas les outils évolués qui le permettent.

Série géométrique

On appelle série géométrique toute série dont le terme général est de la forme $u_n = q^n$.
D'après le théorème précédent, il est manifeste (**pourquoi ?** !) qu'une série géométrique converge si, et seulement si,

$$|q| < 1$$

Calculons dans ce cas la somme de la série :

$$\begin{aligned} S_n &= \sum_{j=0}^{n} q^j = 1 + q + q^2 + q^3 + \cdots + q^n = \frac{1-q^{n+1}}{1-q} \text{ et} \\ \lim_{n\to+\infty} S_n &= \frac{1}{1-q} = \sum_{j=0}^{+\infty} q^j \end{aligned}$$

Série exponentielle

C'est la série dont le terme général est de la forme $u_n = \frac{x^n}{n!}$ où x est un réel quelconque. Nous admettons (la démonstration vous sera donnée peut-être donnée l'année prochaine) qu'une telle série converge et que

$$\sum_{k=0}^{+\infty} \frac{x^k}{k!} = \mathrm{e}^x, \text{ pour tout } x \text{ réel.}$$

6 Logistique dynamique

Présentation du problème

Le mathématicien Belge Pierre-François VERHULST (28 octobre 1804 - 15 février 1849) proposa en 1838 un modèle d'évolution des populations animales qui porte son nom et qui rompt avec l'habituelle croissance exponentielle. On suppose qu'une population vaut p_n à un certain instant et qu'il existe une valeur d'équilibre e telle que la population tend à y revenir avec autant de force qu'elle s'en écarte. Il existe donc un coefficient positif k tel que :

$$\frac{p_{n+1} - p_n}{p_n} = -k(p_n - e)$$

Recherche

Déduisez-en qu'il existe une constante $R \geqslant 1$ telle que :

$$p_{n+1} = Rp_n(1 - \frac{k}{R}p_n)$$

puis qu'il existe une valeur maximum de la population $p_{\max}$ et que :

$$u_{n+1} = Ru_n(1 - u_n)$$

en notant u_n le rapport $\frac{p_n}{p_{\max}}$.

Depuis VERHULST on désigne par *suite logistique* ce type de suite. Il faut bien sûr considérer la plus ancienne des définition du mot :

> **LOGISTIQUE** n.f. **1.** (1611) Anc. nom de la partie de l'algèbre qui traite des quatre règles.

car nous n'utiliserons que les quatre opérations arithmétiques de base mais n'étudierons pas les problème de ravitaillement des armées.

Recherche

Dans quel intervalle varie R ?

Étudier la suite (u_n), c'est étudier le *système dynamique* défini par la fonction f. L'ensemble des x, $f(x)$, $f(f(x))$,...,$f^n(x)$,... est appelé l'*orbite* de x.

Exploration au petit bonheur

Peut-on se contenter d'observer le comportement d'une suite ? L'outil informatique permet-il de se passer d'une exploration théorique ?
Commençons donc par explorer au petit bonheur le comportement de ces suites à l'aide de Python.

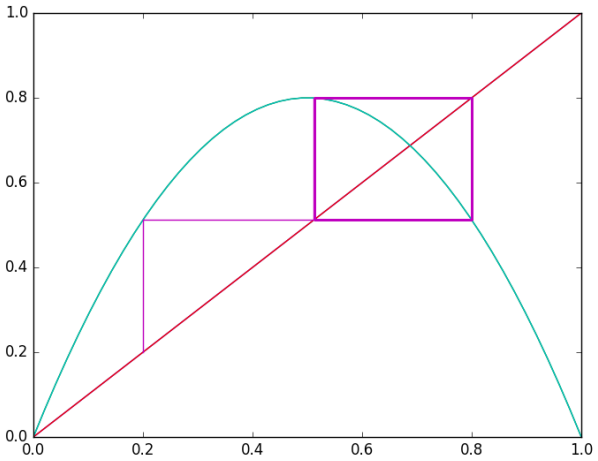
Recherche

Que fait **escargot** ? Comment interpréter les résultats suivants ?

```
1 import matplotlib.pyplot as plt
2 import numpy as np
```

```
3
4  def escargot(R, u0, n) :
5      f = lambda x : R*x*(1-x)
6      X = np.linspace(0, 1)
7      plt.plot(X, X)
8      plt.plot(X, f(X))
9      Ordo = []
10     u = u0
11     for k in range(n):
12         Ordo += [u,f(u)]
13         u = f(u)
14     plt.plot([u0] + Ordo, Ordo + [u])
```

```
1  In [9]: escargot(3.2,0.2,100)
2
3  In [10]: plt.show()
```



En version animée :

```
1  from matplotlib import animation
2
3  fig = plt.figure()
4  ax = plt.axes()
5  ligne, = plt.plot([],[])
6  axe, = plt.plot([],[])
7  courbe, = plt.plot([],[])
8  plt.xlim(0,1)
9  plt.ylim(0,1)
10 texte = ax.text(0.02, 0.95, '')
11
12 def init():
13     X = np.linspace(0, 1)
14     axe.set_data(X,X)
15     courbe.set_data([],[])
16     ligne.set_data([],[])
17     texte.set_text('')
18     return ligne,courbe,axe,texte
19
20 def anim_escargot(R, u0, n) :
21     f = lambda x : R*x*(1-x)
22     X = np.linspace(0, 1)
23     courbe.set_data(X, f(X))
24     texte.set_text('R = %.2f' % R)
```

```
    Ordo = []
    u = u0
    for k in range(n):
        Ordo += [u,f(u)]
        u = f(u)
    ligne.set_data([u0] + Ordo, Ordo + [u])
    return ligne,courbe,axe,texte

anim = animation.FuncAnimation(fig, lambda r :
       → anim_escargot(r*0.01,0.2,50),init_func=init,  blit=True, frames=400,
       → interval=60)
```

Convergence et points fixes

Vous connaissez parfaitement votre cours sur le rapport entre la convergence des suites définies par une relation $u_{n+1} = f(u_n)$ et les points fixes de f.
Malheureusement, on ne parle pas assez du caractère attractif ou répulsif des points fixes.
À l'aide du théorème des accroissements finis, on peut montrer que, ℓ étant un point fixe de f :

- si $|f'(\ell)| < 1$, le point fixe est attractif et (u_n) converge vers ℓ avec $|u_n - \ell| \leqslant k^n |u_0 - \ell|$ pour un certain $k \in]0;1[$;
- si $f'(\ell) = 0$, le point est dit *super attractif* et la suite converge très rapidement ;
- si $|f'(\ell)| > 1$, le point fixe est répulsif et (u_n) ne converge que si elle est constante et égale à ℓ à partir d'un certain rang ;
- si $|f'(\ell)| = 1$, le cas est litigieux.

Recherche

Mais au fait, ces points fixes, quels sont-ils ?
Discuter selon les valeurs de R et commencer à expliquer grossièrement certains comportements observés. Étudier en particulier « l'attractivité » des points fixes.

Étude de la convergence dans le cas 1<R<3

Cas 1<R<=2

Recherche

Traiter le cas $R = 2$.
Que peut-on dire du point fixe non nul dans les autres cas ? Dans quel intervalle se trouve-t-il ?
Discuter alors selon la position de u_0 par rapport à ce point fixe et à $\frac{1}{R}$

Cas 2<R<3

Les dessins obtenus avec Python peuvent donner des idées.

Recherche

Montrer que $I = \left[\frac{1}{2}; \frac{R}{4}\right]$ est stable par f.
En déduire que si $u_0 \in I$, alors (u_{2n}) puis (u_n) converge vers le point fixe non nul.
Étudier ensuite le cas $u_0 \in \left]0; \frac{1}{2}\right[$ puis le cas $u_0 \in \left]\frac{R}{4}; 1\right[$ et montrer qu'on peut se ramener aux cas précédents.

Théorème de Coppel et conséquences

On note $f^n = f \circ f \circ f \circ \cdots \circ f$.
Soit $f : I \to I$ une fonction continue. Soit $x \in I$. Si x est un point fixe de f^n mais n'est pas un point fixe de f^k pour tout $0 < k < n$, on dit que x est **n-périodique**.
L'ensemble $\left\{x, f(x), \cdots, f^{n-1}(x)\right\}$ est un **n-cycle** pour f.
Soit p_0, p_1,...,p_{n-1} un n-cycle On dit qu'il est **attractif** si $|(f^n)'(p_i)| < 1$ pour tout entier naturel i strictement inférieur à n. Dans ce cas, si u_0 est suffisamment proche de l'un des éléments du cycle, la suite admet les éléments du cycle comme valeurs d'adhérence.

Théorème 3 - 4

Théorème de Coppel
Soit $f : [a,b] \to [a,b]$ une fonction continue. Si f n'admet pas de 2-cycle, alors, pour tout $u_0 \in I$, la suite définie par $u_{n+1} = f(u_n)$ converge.

La démonstration est assez longue et nous ne nous en occuperons pas aujourd'hui.
Nous sommes pourtant en mesure de démontrer l'utile lemme suivant :

Lemme 3 - 1

Soit $f : [a,b] \to [a,b]$ une fonction de classe $\mathscr{C}^1$. Si f admet deux points fixes répulsifs consécutifs distincts α et β avec $\alpha < \beta$, alors il existe $p_2 \in]\alpha,\beta[$ tel que p_2 soit un point fixe de f^2.

Pour démontrer ce lemme, il suffit d'étudier la fonction définie sur $[a,b]$ par

$$g(x) = f^2(x) - x$$

en précisant en particulier le signe de $g'(\alpha)$ et $g'(\beta)$.

Recherche

Démontrer le lemme.

Cas R=3

C'est un cas ambigu car dans ce cas $|f'(p)| = |2-3| = 1$.
Cependant, on peut utiliser le théorème de Coppel pour montrer que la suite converge.

Recherche

Démontrer la convergence de la suite (u_n) dans le cas $R = 3$.
On pourra utiliser la fonction **`sympy.solve`**

Cas R>3

C'est maintenant que ça devient amusant...
Cycles d'ordre 2

Recherche

Les points fixes sont maintenant répulsifs : que peut-on en déduire ?
Utiliser éventuellement Python pour déterminer les éventuels points fixes de f^2.
Que se passe-t-il lorsque R tend vers 3 par valeurs supérieures ?

Attractivité des cycles d'ordre 2
Soit p_2 un point fixe de f^2 autre que 0 et p. Il faudrait maintenant savoir quand est-ce que le 2-cycle $(p_2, f(p_2))$ est attractif.

Recherche

Montrer que $(f^2)'(p_2) = -p_2^2+2p_2+4$. Pour cela, montrer que $(f^2)'(p_2) = (f)'(p_2)\cdot f'(f(p_2))$ et utiliser le fait que p_2 et $f(p_2)$ sont des racines d'un polynôme du second degré introduit au paragraphe précédent.
Commenter alors les deux graphiques définis par :

```
In [70]: escargot(0.99+6**0.5,0.5,200)

In [71]: plt.show()

In [72]: escargot(1.01+6**0.5,0.5,200)

In [73]: plt.show()
```

Cycles d'ordre 4
Le 2-cycle cesse d'être attractif pour $R > 1+\sqrt{6}$. D'après le théorème de Coppel, on en déduit que f^2 admet un 2-cycle, c'est-à-dire que f admet un 4-cycle : jusqu'à quelle valeur de R ?
C'est plus difficile à déterminer. On peut regarder graphiquement, par affinements successifs :

```
rmin = 3.5
rmax = 3.9
nbf = 10000
```

```
4
5  anim = animation.FuncAnimation(fig, lambda r : anim_escargot(rmin +
       → r/nbf,0.2,100),init_func=init,  blit=True, frames=int((rmax-rmin)*nbf),
       → interval=60)
6
7  plt.show()
```

Recherche

Quel renseignement nous donne ce graphique ?

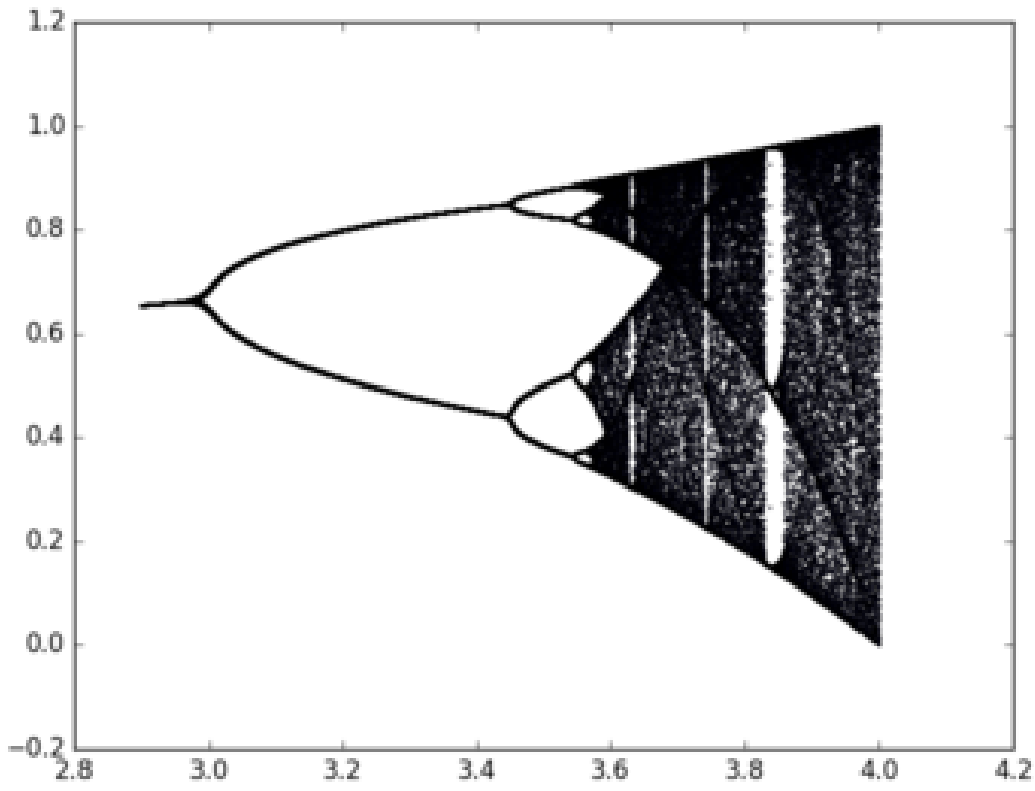
Théorème de Feigenbaum

Diagramme de Feigenbaum

```
1  def ucent(r,u0,nb) :
2      f = lambda x : r*x*(1-x)
3      tmp = 0.2
4      for k in range(100) :
5          tmp = f(tmp)
6      res = []
7      for k in range(nb) :
8          tmp = f(tmp)
9          res.append(tmp)
10     return ([r for _ in range(nb)], res)
11
12 def feig(r1, r2, pas, u0, nb) :
13     for r in (r1 + i*pas for i in range(int((r2-r1)/pas))) :
14         X,Y = ucent(r, u0, nb)
15         plt.scatter(X,Y,marker='.',s=1)
16     plt.show()
```

Recherche

Interpréter le dessin obtenu :



Le théorème

On doit au physicien Mitchell FEIGENBAUM des conjectures qui furent prouvées par la suite par des mathématiciens, des vrais, mais c'est le nom du physicien qui reste attaché aux bifurcations. Voici un condensé des résultats proposé par Daniel PERRIN

Théorème 3 - 5

Soit R_n la borne inférieure des R tel que f admette un cycle d'ordre 2^n.

- on a $R_0 = 0$, $R_1 = 3$, $R_2 = 1+\sqrt{6}$, $R_3 \approx 3,544090$, $R_4 \approx 3,564407$;
- la suite (R_n) est strictement croissante, majorée par 4. Elle converge vers un réel $R_\infty \approx 3,5699456$;
- Pour $R_n < R < R_\infty$, f_R admet un unique 2^n-cycle qui est attractif tant que R est strictement inférieur à R_{n+1} ;
- pour $R < R_\infty$, f_R n'a pas de cycle d'ordre p si p n'est pas une puissance de 2.

Voici qui confirme nos intuitions.

Exposant de Lyapounov

Pour mesurer la sensibilité d'un système dynamique aux conditions initiales, on mesure l'exposant de LYAPOUNOV introduit par le mathématicien russe Alexandre LYAPOUNOV à la fin du XIX^e siècle.
On considère une suite définie par la relation $u_{n+1} = f(u_n)$. Quelle est l'influence d'un écart e_0 sur u_0 pour la suite des itérés ?
Après une itération, l'écart absolu vérifie $|e_1| = |f(u_0+e_0) - f(u_0)|$ et l'écart relatif vaut $\frac{|e_1|}{|e_0|} = \frac{|f(u_0+e_0)-f(u_0)|}{|e_0|} \approx f'(u_0)$ pour $|e_0|$ suffisamment petit.
Après n itérations, l'écart relatif vaut

$$\frac{|e_n|}{|e_0|} = \frac{|e_1|}{|e_0|} \times \frac{|e_2|}{|e_1|} \times \cdots \times \frac{|e_n|}{|e_{n-1}|} = \prod_{k=1}^{n} f'(u_{k-1})$$

Si un des écarts devient nul, notre étude est sans intérêt. Comme effectuer un produit est une opération coûteuse, informatiquement parlant, nous allons pouvoir considérer le logarithme de ce produit.
Notre problème est de savoir si les écarts s'amplifient et donc le produit est supérieur à 1, ou bien si le système est stable et donc le produit est inférieur à 1.

$$\prod_{k=1}^{n} f'(u_{k-1}) < 1 \iff \sum_{k=1}^{n} \ln\left(f'(u_{k-1})\right) < 0$$

Pour relativiser le rôle du choix de n, nous allons normer cette somme en la divisant par n.
On définit alors l'exposant de LYAPOUNOV :

$$\lambda = \lim_{n\to+\infty} \frac{1}{n} \sum_{k=1}^{n} \ln\left(f'(u_{k-1})\right)$$

Recherche

Écrire une fonction **`lyapounov(u0,R,n)`** qui calcule une approximation numérique de l'exposant de Lyapounov qui dépend de la donnée de u_0, R et le nombre n d'itérations. Ensuite, l'utiliser pour représenter l'exposant en fonction de R.

Prolongements

Pour savoir ce qui se passe au-delà de R_∞, étudiez ce merveilleux document :
http://www.math.u-psud.fr/ perrin/Conferences/logistiqueDP.pdf écrit en 2008 par Daniel PERRIN de l'Université d'Orsay.

EXERCICES

Recherche 3 - 1

On considère que op est une opération à temps constant.
Quelle est la complexité des boucles suivantes :

```
Pour i de 1 à n Faire
    op
FinPour
```

```
Pour i de 1 à n Faire
    Pour j de 1 à 90n Faire
        op
    FinPour
    Pour k de 40n à 1 Faire
        op
    FinPour
FinPour
```

```
Pour i de 1 à n*n Faire
    Pour j de 1 à n*3 Faire
        Pour k de 1 à n Faire
            op
        FinPour
    FinPour
FinPour
```

```
Pour i de 1 à n Faire
    Pour j de 1 à i Faire
        op
    FinPour
FinPour
```

```
Pour i de 1 à n Faire
    Pour j de 1 à n Faire
        op
    FinPour
    Pour k de 1 à n Faire
        op
    FinPour
FinPour
```

```
Pour i de 1 à n Faire
    Pour j de 1 à i Faire
        Pour k de 1 à j Faire
            op
        FinPour
    FinPour
FinPour
```

```
Pour i de 1 à n Faire
    Pour j de 1 à n Faire
        op
    FinPour
FinPour
```

```
s ← 0
i ← n
TantQue i > 0 Faire
    Pour j de 0 à i − 1 Faire
        s ← s + 1
    FinPour
    i ← i // 2
FinTantQue
Retourner s
```

```
Pour i de 1 à n Faire
    Pour j de 1 à n Faire
        Pour k de 1 à n Faire
            op
        FinPour
    FinPour
FinPour
```

```
s ← 0
i ← 1
TantQue i < n Faire
    Pour j de 0 à i − 1 Faire
        s ← s + 1
    FinPour
    i ← i * 2
FinTantQue
Retourner s
```

```
s ← 0
i ← 1
TantQue i < n Faire
    Pour j de 0 à n − 1 Faire
        s ← s + 1
    FinPour
    i ← i * 2
FinTantQue
Retourner s
```

Recherche 3 - 2 CCP MP 2015

1. Donnez la décomposition en binaire (base 2) de l'entier 21.
   On considère la fonction `mystere` suivante :

```
def mystere(n, b) :
    """Données : n > 0 un entier et b > 0 un entier
       Résultat : ......."""
    t = [] # tableau vide
    while n > 0 :
        c = n % b
        t.append(c)
        n = n // b
    return t
```

   On rappelle que la méthode append rajoute un élément en fin de liste. Si l'on choisit par exemple `t = [4,5,6]`, alors, après avoir exécuté `t.append(12)`, la liste a pour valeur `[4,5,6,12]`.
   Pour $k \in \mathbb{N}^*$, on note c_k, t_k et n_k les valeurs prises par les variables `c`, `t` et `n` à la sortie de la k-ème itération de la boucle « while ».

2. Soit $n > 0$ un entier. On exécute `mystere(n,10)`. On pose $n_0 = n$.
   i. Justifier la terminaison de la boucle `while`.
   ii. On note p le nombre d'itérations lors de l'exécution de `mystere(n, 10)`. Justifiez que pour tout $k \in [\![0, p]\!]$, on a $n_k \leqslant \frac{n}{10^k}$. Déduisez-en une majoration de p en fonction de n.

3. En vous aidant du script de la fonction **mystere**, écrivez une fonction **somme_chiffres** qui prend en argument un entier naturel et renvoie la somme de ses chiffres. Par exemple, `somme_chiffres(256)` devra renvoyer 13.

4. Écrivez une version récursive de la fonction **somme_chiffres** que vous nommerez **somme_rec**. Est-elle plus efficace ?

5. Écrivez une version en Haskell de cette fonction : est-elle plus efficace ?

Recherche 3 - 3 Maths II CCP MP 2015 : sujet 0

1. Écrire une fonction factorielle qui prend en argument un entier naturel n et renvoie $n!$ (on n'accepetra pas bien sûr de réponse utilisant la propre fonction factorielle du module math de Python ou Scilab).

2. Écrire une fonction `seuil` qui prend en argument un entier M et renvoie le plus petit entier naturel n tel que $n! > M$.

3. Écrire une fonction booléenne nommée `est_divisible`, qui pend en argument un entier naturel n et renvoie `True` si $n!$ est divisible par $n + 1$ et `False` sinon.

4. On considère la fonction suivante nommée `mystere` :

```
def mystere(n):
    s = 0
    for k in range(1,n+1):
        s = s + factorielle(k)
    return s
```

i. Quelle valeur renvoie `mystere(4)` ?

ii. Déterminer le nombre de multiplications qu'effectue `mystere(n)`.

iii. Proposer une amélioration du script de la fonction `mystere` afin d'obtenir une complexité linéaire.

Recherche 3 - 4

Donnez l'ordre de complexité temporelle des fragments de code Java suivants :

```java
int sum = 0;
for (int n = N; n > 0; n /= 2)
   for (int i = 0; i < n; i++)
      sum++;
```

```java
int sum = 0;
for (int i = 1; i < N; i *= 2)
   for(int j = 0; j < i; j++)
      sum++;
```

```java
int sum = 0;
for (int i = 1; i < N; i *= 2)
   for (int j = 0; j < N; j++)
      sum++;
```

Recherche 3 - 5 Tri selectif

On parcourt la liste, on cherche le plus grand et on l'échange avec l'élément le plus à droite et on recommence avec la liste privée du plus grand élément.
On commence par chercher l'indice du maximum d'une liste. On part de 0 et on compare à chaque élément de la liste en faisant évoluer l'indice du maximum si nécessaire.

```python
def ind_maxi(xs):
    ind_tmp = 0
    n       = len(xs)
    for i in range(n):
        if xs[i] > xs[ind_tmp]:
            ind_tmp = i
    return ind_tmp
```

Ensuite, on copie la liste donnée en argument pour ne pas l'écraser. On parcourt la liste et on procède aux échanges éventuels entre le maximum et l'élément de droite.

```python
def tri_select(xs):
    cs = xs.copy()
    n  = len(cs)
    for i in range(n - 1,0,-1):
        i_m = ind_maxi(cs[:i + 1])
        if i_m != i:
            cs[i_m],cs[i] = cs[i],cs[i_m]
        # print(cs) : pour suivre l'évolution
    return cs
```

On va utiliser `permutation` de la bibliothèque `numpy.random` qui renvoie une permutation uniformément choisie par les permutations de S_n.

```
In [5]: ls = list(permutation(range(10)))

In [6]: ls
Out[6]: [8, 0, 4, 3, 5, 2, 7, 1, 9, 6]

```

```
In [7]: tri_select(ls)
[8, 0, 4, 3, 5, 2, 7, 1, 6, 9]
[6, 0, 4, 3, 5, 2, 7, 1, 8, 9]
[6, 0, 4, 3, 5, 2, 1, 7, 8, 9]
[1, 0, 4, 3, 5, 2, 6, 7, 8, 9]
[1, 0, 4, 3, 2, 5, 6, 7, 8, 9]
[1, 0, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 0, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 0, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Out[7]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

1. Il s'agit de démontrer la **correction** de ces deux fonctions.

   Démontrez que pour la première, l'invariant de boucle est : « `ind_temp` est l'indice du maximum des `i + 1` premiers termes de la liste en argument ».

   Quel est celui de la seconde ? Démontrez-le.

2. Nous mesurerons la complexité temporelle en nombre de comparaisons.

   Expérimentalement, nous pouvons en avoir une idée :

```
In [10]: ls = list(permutation(100))

In [11]: %timeit tri_select(ls)
1000 loops, best of 3: 561 µs per loop

In [12]: ls = list(permutation(200))

In [13]: %timeit tri_select(ls)
100 loops, best of 3: 2.03 ms per loop

In [14]: ls = list(permutation(400))

In [15]: %timeit tri_select(ls)
100 loops, best of 3: 7.89 ms per loop
```

   Démontrez-le.

Recherche 3 - 6 Questions sur le cours

Répondez aux questions de la page page 84

Recherche 3 - 7 Dichotomie

Déterminez la constante asymptotique de la méthode dichotomique.
Déterminez un algorithme (impératif et récursif) donnant l'approximation d'une solution avec comme arguments la fonction f, les bornes de l'intervalle de départ a et b et la précision p.
Vous éviterez les *tant que* et les conditionnelles basées sur une comparaison de flottants par sécurité...

```
*Main> dicho2 (\ x -> x**2 - 2) 1 2 1e-15
1.414213562373095
*Main> sqrt 2
```

Recherche 3 - 8 Babylone

Existe-t-il un lien entre l'algorithme de Babylone et la méthode de Newton-Raphson ?

Recherche 3 - 9 Newton 1

Répondez aux questions posées dans les encadré *recherche* de la section consacrée à la méthode de Newton-Raphson

Recherche 3 - 10 Newton 2

Programmez cette méthode...On pourra créer une fonction :

```
newtRaph :: (Fractional a, Ord a) => (a -> a) -> (a -> a) -> a -> a -> a
newtRaph                              f          f'          x    p = ...
```

Et vérifiez par exemple :

```
*Main> newtRaph (\ t -> t**2 - 2) (\ t -> 2*t) 1 1e-15
1.4142135623730951
```

Commentez ces résultats :

```
*Main> newtRaph (\ t -> t**3 - t) (\ t -> 3*t**2 - 1) 0.25 1e-15
0.0
*Main> newtRaph (\ t -> t**3 - t) (\ t -> 3*t**2 - 1) 0.5 1e-15
-1.0
*Main> newtRaph (\ t -> t**3 - t) (\ t -> 3*t**2 - 1) (-1/(sqrt 5)) 1e-15
  C-c C-cInterrupted.
```

Recherche 3 - 11 Newton 3 : calcul de l'inverse sur machine

Utilisez la méthode de Newton-Raphson pour déterminer une approximation de l'inverse d'un flottant sur machine.

```
*Main> inverse 1.5 1 1e-15
0.6666666666666666
```

Pour étudier les problèmes dus aux arrondis, on pourra lire les pages 157 à 167 de Muller et coll. [2010].

Recherche 3 - 12 Newton 4 : influence de la première approximation

Notons ε_n l'erreur relative après n itérations par la méthode de Newton-Raphson pour calculer $\sqrt{a}$.
Alors $\varepsilon_n = \frac{x_n - \sqrt{a}}{\sqrt{a}}$ et donc $x_n = \frac{1+\varepsilon_n}{\sqrt{a}}$. Or $x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$.
Montrez qu'alors

$$\varepsilon_{n+1} = \frac{\varepsilon_n^2}{2(1+\varepsilon_n)}$$

Si on est trop loin de la solution au départ, ε_n n'est pas négligeable devant 1 : comparez ε_{n+1} et ε_n.
Que se passe-t-il sinon ?

Recherche 3 - 13 Méthode de la sécante

Le calcul de la dérivée peut être embêtant sur machine. Que peut-il se passer si on remplace $f'(x_n)$ par $\frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$?

Recherche 3 - 14 Exponentiations

On veut calculer x^n avec n un entier naturel et x un élément d'un ensemble muni d'une loi multiplicative associative.
On pose $p_1 = x$ et $p_j = x^j$.
On impose de déterminer un mécanisme qui calcule p_i si on connait déjà $p_1, p_2, \ldots, p_{i-1}$ sous la forme :

$$p_i = p_j \cdot p_k \qquad \text{avec } 0 \leqslant j, k \leqslant i-1$$

1. Donnez un algorithme naïf. Quel est son coût ?
2. On écrit l'exposant en binaire. On remplace chaque 1 par CX et chaque 0 par C. On enlève le premier couple CX (le plus à gauche).
   On traduit C par « mettre au carré » et X par « multiplier par x ».
   Traduisez cet algorithme de manière plus constructive et étudiez sa complexité. Vous donnerez une version impérative et une version récursive.

Recherche 3 - 15 Recherches séquentielles et par saut

Un fichier séquentiel informatisé est schématiquement constitué ainsi, chaque fiche contient

- *L'adresse S de la fiche suivante si elle existe sinon* ***nil*** *pour indiquer la fin du fichier.*
- *L'adresse P de la fiche précédente si elle existe sinon* ***nil*** *pour indiquer qu'il n'y a rien en deçà.*
- *Une donnée D_i alphanumérique*
- *Une première adresse* ***début*** *qui est l'adresse de la première fiche.*

Nous supposerons que le fichier est trié suivant l'ordre croissant et pour simplifier l'étude nous poserons $D_i = i$ (la donnée est égale au numéro de la fiche). Le fichier contient N fiches. Nous appellerons temps d'accès à la fiche numéro i le nombre de fiches lues en partant de *début* pour arriver à la fiche i y compris la fiche i. Ainsi le temps d'accès à la fiche 1 est 1.

Recherche séquentielle.

Pour trouver la fiche i $(1 \leq i \leq N)$ on part de *début*, on lit les fiches dans l'ordre des numéros jusqu'à obtenir la fiche i. Si on cherche successivement les fiches 1,2,3,$\cdots$, N en repartant de *début* à chaque fois, quel est le temps moyen d'accès à une fiche ?

Recherche par saut.

- *Nous supposerons ici que N est le carré d'un entier, $N = a^2$ $(a \in N^*)$. Soit k un entier divisant N et supérieur à 1, $N = kj$.*
- *Nous créons un autre fichier dit de nœuds $N_1, N_2, \ldots, N_j$.*
- *Le départ, noté* **dép**, *contient l'adresse du premier nœud.*
- *Le premier nœud contient l'adresse de la fiche numéro k, la donnée de la fiche numéro k l'adresse du nœud suivant et l'adresse de la fiche $k-1$.*
- *Le deuxième nœud contient l'adresse de la fiche numéro $2k$, la donnée de la fiche numéro $2k$ l'adresse du nœud suivant et de la fiche numéro $2k-1$.*
- .....
- *Le dernier nœud (le $j^{ème}$) contient l'adresse de la fiche numéro N,* et celle de $N-1$, *la donnée de la fiche numéro N et* ***nil***.

Pour chercher une fiche i on part de **dép**, on passe au premier nœud et on regarde la donnée α portée par le nœud. Si $\alpha < i$ on passe au nœud suivant, sinon on passe à la fiche indiquée par le nœud et, si nécessaire, on revient en arrière dans le fichier jusqu'à atteindre la fiche i. Un temps d'accès à une fiche sera le nombre de nœuds parcourus + le nombre de fiches parcourues y compris celle cherchée.

Si on cherche successivement les fiches 1,2,3,$\cdots$, N en repartant de **dép** à chaque fois, on se propose de déterminer le temps moyen d'accès à une fiche et trouver la valeur de k qui minimise ce temps moyen.

1. Relier la recherche par saut avec la **??** page ??
2. Déterminer le temps d'accès à la fiche 1.
3. Déterminer le temps d'accès à la fiche 2 (si $k > 2$).
4. Déterminer le temps d'accès à la fiche k.
5. Déterminer le temps d'accès à la fiche $k-1$.
6. Déterminer le temps d'accès à la fiche $k+1$.
7. Déterminer le temps d'accès à la fiche ik.
8. Déterminer le temps d'accès à la fiche N.
9. Déterminer le temps d'accès à la fiche j.
10. Déterminer le temps total d'accès aux fiches du $i^{ème}$ paquet.
11. Déterminer le temps total d'accès à toutes les fiches.
12. Déterminer le temps moyen d'accès à une fiche en fonction de k et de N.
13. Déterminer la valeur de k qui minimise ce temps moyen.
14. Comparer le temps moyen d'accès à une fiche par une recherche séquentielle et une recherche par saut en prenant N = 10 000 et une unité de temps égale à $10^{-2}s$.

Recherche 3 - 16 Multiplication du paysan russe

Voici ce qu'apprenaient les petits soviétiques pour multiplier deux entiers. Une variante de cet algorithme a été retrouvée sur le papyrus de Rhind datant de 1650 avant JC, le scribe Ahmes affirmant que cet algorithme était à l'époque vieux de 350 ans. Il a survécu en Europe occidentale jusqu'aux travaux de Fibonacci.

```
1  Fonction MULRUSSE(x:entier ,y: entier, acc:entier):entier
2  Si x==0 Alors
3  |  Retourner acc
4  Sinon
5  |  Si x est pair Alors
6  |  |  Retourner MULRUSSE(x/2,y*2,acc)
7  |  Sinon
8  |  |  Retourner MULRUSSE((x-1)/2,y*2,acc+y)
9  |  FinSi
10 FinSi
```

Que vaut `acc` au départ ?

Écrivez une version récursive de cet algorithme en évitant l'alternative des lignes 5 à 9.

Écrivez une version impérative de cet algorithme.

Prouvez la correction de cet algorithme.

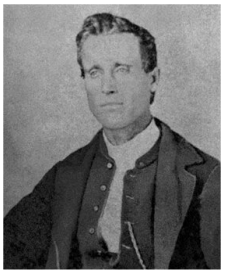
Étudiez sa complexité.

En python, `x >> 1` décale l'entier x d'un bit vers la doite et `x << 1` décale x d'un bit vers la gauche en complétant par un zéro à droite, `x& y` renvoie l'entier obtenu en faisant la conjonction logique bit à bit des représentations binaires de x et y et `~x` renvoie le complément à 1 de x.

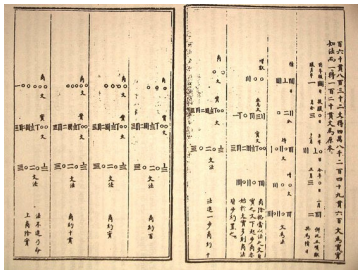
Ré-écrivez la multiplication russe en utilisant que ces opérations bit à bit (pas de division ni de multiplication).

Recherche 3 - 17 Schéma de Horner-Ruffini-Holdred-Newton-Al-Tusi-Liu-Hui...

Paolo Ruffini
(1765 - 1822)



William Horner
(1765 - 1822)



Neuf chapitres sur l'art mathématique

La méthode que nous allons voir porte le nom du britannique William George Horner (1786 - 1837) mais en fait elle fut publiée presque 10 ans auparavant par un horloger londonien, Theophilius Holdred et simultanément par l'italien Paolo Ruffini (1765 - 1822) mais fut déjà utilisée par Newton 150 ans auparavant et par le chinois Zhu Shije cinq siècles plus tôt (vers 1300) et avant lui par le Persan Sharaf al-Din al-Muzaffar ibn Muhammad ibn al-Muzaffar al-Tusi vers (1100) et avant lui par le Chinois Liu Hui (vers 200) révisant un des résultats présent dans *Les Neuf Chapitres sur l'art mathématique* publié avant la naissance de JC...

Il faut cependant noter que Ruffini l'avait employée en fait comme un moyen de calculer rapidement le quotient et le reste d'un polynôme par $(X-\alpha)$.

Dans toute la suite, un polynôme de dégré n sera représenté par le vecteur de ses coefficients. Par exemple, `[1 2 3]` correspond au polynôme $1+2x+3x^2$.

Prenons l'exemple de $P(x)=3x^5-2x^4+7x^3+2x^2+5x-3$. Le calcul classique nécessite 5 additions et 15 multiplications.

On peut faire pas mal d'économies de calcul en suivant le schéma suivant :

$$\begin{aligned} P(x) &= \underbrace{a_n x^n + \cdots + a_2 x^2 + a_1 x}_{\text{on met } x \text{ en facteur}} + a_0 \\ &= \left(\underbrace{a_n x^{n-1} + \cdots + a_2 x}_{\text{on met } x \text{ en facteur}} + a_1 \right) x + a_0 \\ &= \cdots \\ &= (\cdots(((a_n x + a_{n-1})x + a_{n-2})x + a_{n-3})x + \cdots)\, x + a_0 \end{aligned}$$

Ici cela donne $P(x) = (((((3x) - 2)x + 7)x + 2)x + 5)x - 3$ c'est-à-dire 5 multiplications et 5 additions.
Comparez les complexités au pire du calcul de $P(t)$ pour $t \in \mathbb{K}$. Vous prendrez comme « unité de complexité » les opérations arithmétiques de base : `+ - *`.
Déterminez une fonction `horner(P,t)` qui évalue le polynôme `P` en `t` selon le schéma de HORNER.

Recherche 3 - 18 Local min

Write a program that, given an array a[] of N distinct integers, finds a local minimum : an index i such that both `a[i] < a[i-1]` and `a[i] < a[i+1]` (assuming the neighboring entry is in bounds). Your program should use 2 lg N compares in the worst case.

Recherche 3 - 19 Bitonic search

An array is bitonic if it is comprised of an increasing sequence of integers followed immediately by a decreasing sequence of integers. Write a program that, given a bitonic array of N distinct `int` values, determines whether a given integer is in the array. Your program should use 3 lg N compares in the worst case.

Recherche 3 - 20 Floor and ceiling

Given a set of comparable elements, the ceiling of x is the smallest element in the set greater than or equal to x, and the floor is the largest element less than or equal to x. Suppose you have an array of N items in ascending order. Give an O(log N) algorithm to find the floor and ceiling of x.

Recherche 3 - 21 Common elements

Given two arrays of N 64-bit integers, design an algorithm to print out all elements that appear in both lists. The output should be in sorted order. Your algorithm should run in N log N. Hint : mergesort, mergesort, merge. Remark : not possible to do better than N log N in comparison based model.

Recherche 3 - 22 Mumixam

Given an array a[] of N real numbers, design a linear-time algorithm to find the maximum value of a[j] - a[i] where j $\geqslant$ i.

Recherche 3 - 23 DÉFI

Résolvez le problème suivant : http://introcs.cs.princeton.edu/java/assignments/collinear.html

Recherche 3 - 24

On travaille dans $\mathbb{R}$. Calculer les sommes :

1. $\sum_{i=3}^{6} ij$, $\sum_{i=0}^{4} i^2$, $\sum_{k=0}^{4} k^2$, $\sum_{j=0}^{4} i^2$, $\sum_{i=1}^{50} a_j$
2. $\sum_{i=1}^{n\geq 1} a$, $\sum_{i=h}^{n\geq h} a$, $\sum_{i=1}^{n\geq 1} 3$
3. $\sum_{1\leq i<j\leq 5} ij$, $\sum_{1\leq i<j\leq 5} (i+2j)$

Recherche 3 - 25

On travaille dans $\mathbb{R}$, calculer les sommes $\sum_{i=1}^{n} i$, $\sum_{i=50}^{100} j$, $\sum_{i=h}^{k} i$, $\sum_{i=h}^{k} (i+j)$, $\sum_{i=h}^{k} ij$, $\sum_{i=h}^{k} (\alpha i + j)$

Recherche 3 - 26

On considère un tableau de nombres ayant n lignes et p colonnes. Les lignes sont numérotées du haut en bas et les colonnes de la gauche vers la droite. Le nombre se trouvant sur la $i^{ème}$ ligne et $j^{ème}$ colonne est noté $a_{i,j}$ ou a_{ij}. Par exemple, si $n = 4$ et $p = 3$, le tableau se présente par

$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{2,1}$	$a_{2,2}$	$a_{2,3}$
$a_{3,1}$	$a_{3,2}$	$a_{3,3}$
$a_{4,1}$	$a_{4,2}$	$a_{4,3}$

On note L_i la somme des nombres de la ligne numéro i, C_j la somme des nombres de la colonne j et T la somme de tous les nombres du tableau.

1. Exprimer L_i et C_j à l'aide d'un symbole $\sum$.
2. Exprimer T de deux façons.

Recherche 3 - 27

On reprend les notations de l'exercice précédent avec $n = p = 5$. Que calculent les sommes suivantes ? On pourra donner la solution en grisant les cases du tableau qui correspondent aux nombres intervenant dans les sommes.

1. $\sum_{i=1}^{5}\sum_{j=1}^{5} a_{ij}$
2. $\sum_{j=1}^{5}\sum_{i=1}^{5} a_{ij}$
3. $\sum_{i=1}^{5}\sum_{j=1}^{i} a_{ij}$
4. $\sum_{j=1}^{5}\sum_{i=1}^{j} a_{ij}$
5. $\sum_{i=1}^{5}\sum_{j\geq i} a_{ij}$
6. $\sum_{i+j=6} a_{ij}$
7. $\sum_{1\leq i<j\leq 5} a_{ij}$

Recherche 3 - 28

Un fichier séquentiel informatisé est schématiquement constitué ainsi, chaque fiche contient

- *L'adresse S de la fiche suivante si elle existe sinon* ***nil*** *pour indiquer la fin du fichier.*
- *L'adresse P de la fiche précédente si elle existe sinon* ***nil*** *pour indiquer qu'il n'y a rien en deçà.*
- *Une donnée D_i alphanumérique*
- *Une première adresse* ***début*** *qui est l'adresse de la première fiche.*

Nous supposerons que le fichier est trié suivant l'ordre croissant et pour simplifier l'étude nous poserons $D_i = i$ (la donnée est égale au numéro de la fiche). Le fichier contient N fiches. Nous appellerons temps d'accès à la fiche numéro i le nombre de fiches lues en partant de *début* pour arriver à la fiche i y compris la fiche i. Ainsi le temps d'accès à la fiche 1 est 1.

Recherche séquentielle.

Pour trouver la fiche i ($1 \leq i \leq N$) on part de *début*, on lit les fiches dans l'ordre des numéros jusqu'à obtenir la fiche i.
Si on cherche successivement les fiches 1,2,3,$\cdots$, N en repartant de *début* à chaque fois, quel est le temps moyen d'accès à une fiche ?

Recherche par saut.

- *Nous supposerons ici que N est le carré d'un entier, $N = a^2$ ($a \in N^*$). Soit k un entier divisant N et supérieur à 1, $N = kj$.*
- *Nous créons un autre fichier dit de nœuds $N_1, N_2, \ldots, N_j$.*
- *Le départ, noté* **dép**, *contient l'adresse du premier nœud.*
- *Le premier nœud contient l'adresse de la fiche numéro k, la donnée de la fiche numéro k l'adresse du nœud suivant et l'adresse de la fiche $k-1$.*
- *Le deuxième nœud contient l'adresse de la fiche numéro $2k$, la donnée de la fiche numéro $2k$ l'adresse du nœud suivant et de la fiche numéro $2k-1$.*
- ……
- *Le dernier nœud (le $j^{ème}$) contient l'adresse de la fiche numéro N, et celle de $N-1$, la donnée de la fiche numéro N et* ***nil***.

Pour chercher une fiche i on part de **dép**, on passe au premier nœud et on regarde la donnée α portée par le nœud. Si $\alpha < i$ on passe au nœud suivant, sinon on passe à la fiche indiquée par le nœud et, si nécessaire, on revient en arrière dans le fichier jusqu'à atteindre la fiche i. Un temps d'accès à une fiche sera le nombre de nœuds parcourus + le nombre de fiches parcourues y compris celle cherchée.
Si on cherche successivement les fiches 1,2,3,$\cdots$, N en repartant de **dép** à chaque fois, on se propose de déterminer le temps moyen d'accès à une fiche et trouver la valeur de k qui minimise ce temps moyen.

1. Déterminer le temps d'accès à la fiche 1.
2. Déterminer le temps d'accès à la fiche 2 (si $k > 2$).
3. Déterminer le temps d'accès à la fiche k.
4. Déterminer le temps d'accès à la fiche $k-1$.
5. Déterminer le temps d'accès à la fiche $k+1$.
6. Déterminer le temps d'accès à la fiche ik.
7. Déterminer le temps d'accès à la fiche N.
8. Déterminer le temps d'accès à la fiche j.
9. Déterminer le temps total d'accès aux fiches du $i^{ème}$ paquet.
10. Déterminer le temps total d'accès à toutes les fiches.
11. Déterminer le temps moyen d'accès à une fiche en fonction de k et de N.
12. Déterminer la valeur de k qui minimise ce temps moyen.
13. Comparer le temps moyen d'accès à une fiche par une recherche séquentielle et une recherche par saut en prenant $N = 10000$ et une unité de temps égale à $10^{-2}s$.

Recherche 3 - 29 Achille et la tortue

Le paradoxe suivant a été imaginé par Zénon d'Élée (490-430 Avant JC). Achille fait une course avec la tortue. Il part 100 mètres derrière la tortue, mais il va dix fois plus vite qu'elle. Quand Achille arrive au point de départ de la tortue, la tortue a parcouru 10 mètres. Pendant qu'Achille parcourt ces 10 mètres, la tortue a avancé d'un mètre. Pendant qu'Achille parcourt ce mètre, la tortue a avancé de 10cm... Puisqu'on peut réitérer ce raisonnement à l'infini, Zénon conclut qu'Achille ne peut pas dépasser la tortue...

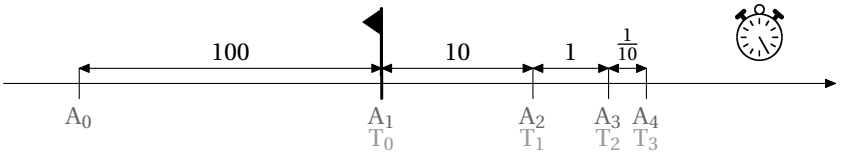
Pour étudier ce problème, nous aurons besoin d'un résultat intermédiaire.

Il est assez simple de démontrer par récurrence, par exemple, que pour tout entier $n > 1$ et tout réel $x > -1$, on a :

$$(1+x)^n > 1 + nx \quad \text{(Inégalité de Bernoulli)}$$

Qu'en déduisez-vous au sujet de la limite des suites de terme général q^n ?

Voici la situation :



Intéressons-nous d'abord à la distance Achille-Tortue. Notons d_n la distance :

$$d_n = T_n - A_n = T_n - T_{n-1} = \frac{1}{10}(T_{n-1} - A_{n-1}) = \frac{1}{10}d_{n-1}$$

La suite (d_n) est donc géométrique de raison $1/10$ et de premier terme 100. On en déduit que $d_n = 100 \times \left(\frac{1}{10}\right)^{n-1}$.

Or $|1/10| < 1$, donc $\lim\limits_{n\to+\infty} (1/10)^{n-1} = 0 = \lim\limits_{n\to+\infty} d_n$.

Ainsi Achille va rattrapper la tortue, mais au bout d'une infinité de trajets : pour le Grec Zénon, la notion d'infini étant « au-delà du réel » ; pour lui Achille ne rattrapera jamais la tortue, ce qui est absurde.

Intéressons-nous plutôt à la durée du trajet d'Achille pour atteindre la tortue, en supposant sa vitesse constante et égale à v.

Notons $t_1 = \frac{d_1}{v} = \frac{100}{v}$, $t_2 = \frac{d_2}{v} = \frac{1}{10}\frac{d_1}{v} = \frac{1}{10}t_1$, etc. La suite (t_n) est donc géométrique de raison $\frac{1}{10}$ de premier terme $t_1 = 100/v$, d'où $t_n = \frac{100}{v}\left(\frac{1}{10}\right)^{n-1}$. La durée du trajet est alors :

$$\begin{aligned}\tau_n &= t_1 + t_2 + \cdots + t_n \\ &= \frac{100}{v}\left(1 + \frac{1}{10} + \left(\frac{1}{10}\right)^2 + \cdots + \left(\frac{1}{10}\right)^{n-1}\right) \\ &= \frac{1000}{9v}\left(1 - \left(\frac{1}{10}\right)^n\right)\end{aligned}$$

Nous venons de voir qu'Achille atteindra la tortue quand n tend vers $+\infty$. Or nous savons calculer $\lim\limits_{n\to+\infty} \tau_n = \frac{1000}{9v}$ qui est un nombre fini. Achille va donc effectuer cette infinité de trajets en un temps fini égal à $1000/9v$.

Zénon pensait qu'une somme infinie de termes strictement positifs était nécessairement infinie, d'où le paradoxe à ses yeux. Il a fallu des siècles à l'esprit humain pour dépasser cette *limite*.

Vous pouvez donc prouver le petit théorème suivant :

Théorème 3 - 6

Série géométrique

La suite de terme général

$$S_n = \sum_{k=0}^{n} q^n$$

converge si, et seulement si, $|q| < 1$. Dans ce cas :

$$S = \lim_{n \to +\infty} S_n = \frac{1}{1-q}$$

Recherche 3 - 30 Tapis de Sierpinski

Monsieur Sierpinski avait ramené d'un voyage en Orient un tapis carré de 1 mètre de côté dont il était très content. Jusqu'au jour où les mites s'introduisirent chez lui.
En 24 heures, elles dévorèrent dans le tapis un carré de côté trois fois plus petit, situé exactement au centre du tapis. En constatant les dégats, Monsieur Sierpinski entra dans une colère noire ! Puis il se consola en se disant qu'il lui restait huit petits carrés de tapis, chacun de la taille du carré disparu. Malheureusement, dans les 12 heures qui suivirent, les mites avaient attaqué les huit petits carrés restants : dans chacun, elles avaient mangé un carré central encore trois fois plus petit. Et dans les 6 heures suivantes elles grignotèrent encore le carré central de chacun des tout petits carrés restants. Et l'histoire se répéta, encore et encore ; à chaque étape, qui se déroulait dans un intervalle de temps deux fois plus petit que l'étape précédente, les mites faisaient des trous de taille trois fois plus petite...

1. Faire des dessins pour bien comprendre la géométrie du tapis troué. Calculer le nombre total de trous dans le tapis de Monsieur Sierpinski après n étapes. Calculer la surface S_n de tapis qui n'a pas encore été mangée après n étapes. Trouver la limite de la suite $(S_n)_{n\geq 0}$. Que reste-t-il du tapis à la fin de l'histoire ?
2. Calculer la durée totale du festin « mitique »...

Recherche 3 - 31

$|x| < 1$ et on note $S_n(x) = \sum_{k=0}^{n} x^k$, $\sigma_n(x) = \sum_{k=0}^{n} kx^k = \sum_{k=1}^{n} kx^k$.

1. Calculer $S_n'(x)$.
2. Calculer $\sum_{k=0}^{+\infty} x^k$.
3. Calculer la limite de $S_n'(x)$ lorsque n tend vers $+\infty$.
4. Exprimer $\sigma_n(x)$ à l'aide de $S_n(x)$.
5. Calculer $\sum_{k=0}^{+\infty} kx^k$.

Recherche 3 - 32

On note $\Pr([X = k]) = \frac{\lambda^k}{k!} e^{-\lambda}$. Calculer

1. $E(X) = \sum_{k=0}^{+\infty} k \Pr([X = k])$.
2. $E(X^2) = \sum_{k=0}^{+\infty} k^2 \Pr([X = k])$.
3. $E(X^2) - E(X)^2$.

Recherche 3 - 33

On note $\Pr([X = k]) = p(1-p)^{k-1}$ avec $p \in]0,1[$. Calculer $\sum_{k=1}^{+\infty} \Pr([X = k])$ et $\sum_{k=1}^{+\infty} k \Pr([X = k])$.

Recherche 3 - 34

On considère le tableau $A = (a_{ij}) = \begin{bmatrix} 1 & 2 & -2 & 4 & 0 \\ 0 & 1 & 2 & -2 & 4 \\ 4 & 0 & 1 & 2 & -2 \\ -2 & 4 & 0 & 1 & 2 \\ 2 & -2 & 4 & 0 & 1 \end{bmatrix}$. Calculez :

1. $S_1 = \sum_{i=1}^{5} \sum_{j=1}^{5} a_{ij}$
2. $S_2 = \sum_{j=1}^{5} \sum_{i=1}^{5} a_{ij}$
3. $S_3 = \sum_{i=1}^{5} \sum_{j=1}^{i} a_{ij}$
4. $S_4 = \sum_{j=1}^{5} \sum_{i=1}^{j} a_{ij}$
5. $S_5 = \sum_{i=1}^{5} \sum_{j \geq i} a_{ij}$
6. $S_6 = \sum_{i+j=6} a_{ij}$
7. $S_7 = \sum_{1 \leq i < j \leq 5} a_{ij}$

Recherche 3 - 35

Donnez une expression la plus simple possible de $\sum_{i=p}^{i=s-1} \left(k \times i + \frac{h(h-1)}{2} \right)$ sans symbole Σ, ni $\cdots$.

Recherche 3 - 36

On travaille dans $\mathbb{R}$. Calculer les sommes le plus simplement possible sans utiliser ni $\sum$ ni ... :

$S_1 = \sum_{i=3}^{6} ij$ $S_{11} = \sum_{i=1}^{n} i$ $S_{101} = \sum_{i=h}^{k} (i+j)$

$S_{10} = \sum_{1 \leq i < j \leq 5} (i+2j)$ $S_{100} = \sum_{i=h}^{k} i$ $S_{110} = \sum_{i=h}^{k} ij$

Recherche 3 - 37 DS

On considère la fonction u définie par :

```
1 u(n) = pliage(lambda x, k : (x + 4 / x) * 0.5, range(1, n + 1), 2.5)
```

1. Traduisez cette fonction par une boucle *pour* en pseudo-langage.
2. Traduisez cette fonction sous forme d'une suite (u_n) définie par récurrence.
3. Démontrez que, pour tout entier naturel, $u_n \geqslant 2$.
4. Démontrez que la suite (u_n) est décroissante.
5. Déduisez-en le rôle de la fonction Python u.

Recherche 3 - 38 DS

Le code C suivant...

```
1  #include <stdio.h>
2  #include <float.h>
3  double truc(double x) {
4    return x >= 0 ? x : -x;
5  }
6  double mystere(double t) {
7    double d, x = 1;
8    if (!t) return 0;
9    if (t < 0) return 0./0.;
10
11   do {        d = (t / x - x) / 2;
12     x+= d;
13     printf("%1.20f, %1.20f\n",x,d);
14   } while (truc(d) >= truc(x) * (DBL_EPSILON * 10));
15
16   return x;
17 }
18 int main() {
19   double x = 2;
20   printf("mystere(%g) = %1.15f\neps = %1.20f\n", x, mystere(x),DBL_EPSILON);
21   return 0;
22 }
```

...affiche :

Shell

```
1.50000000000000000000, 0.50000000000000000000
1.41666666666666651864, -0.08333333333333337034
1.41421568627450966460, -0.00245098039215674302
1.41421356237468986983, -0.00000212389981979477
```

Shell

```
1.41421356237309492343, -0.00000000000159483537
1.41421356237309492343, 0.00000000000000011102
mystere(2) = 1.414213562373095
 eps = 0.00000000000000022204
```

En vous appuyant sur les résultats donnés dans le chapitre 3 du poly, expliquez ce que font les fonctions truc et mystere, quels sont les rôles des paramètres et variables t, d et x, le rôle de la ligne 9 et la signification du test de la ligne 14.

LECTURES RECOMMANDÉES POUR ALLER PLUS LOIN

C. Brezinski et M. Zaglia,
Méthodes numériques itératives : Algèbre linéaire et non linéaire,
vol. Mathématiques à l'Université, Ellipses Éditions Marketing, ISBN 9782729828875, 2006.

A. Casamayou, P. Chauvin et G. Connan,
Programmation en python pour les mathématiques,
vol. Sciences sup, Dunod 2011, Paris, ISBN 978-2-10-057422-3, adresse : http://opac.inria.fr/record=b1133732, 2012.

A. Casamayou, G. Connan, T. Dumont, L. Fousse, F. Maltey, M. Meulien, M. Mezzarobba, C. Pernet, N. M. Thiéry et P. Zimmermann,
Calcul mathématique avec Sage,
Amazon, ISBN 9781481191043, adresse : http://hal.inria.fr/inria-00540485, 2013,
electronic version available under Creative Commons license.

F. de Dinechin,
« Page personnelle », adresse : http://perso.citi-lab.fr/fdedinec/, 2014.

G. Dowek,
Les démonstrations et les algorithmes : Introduction à la logique et à la calculabilité,
Ecole Polytechnique, ISBN 9782730215695, 2010.

P. Dugac,
Histoire de l'analyse : autour de la notion de limite et de ses voisinages,
Vuibert, Paris, 2003.

L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier et P. Zimmermann,
« Mpfr : A multiple-precision binary floating-point library with correct rounding »,
ACM Trans. Math. Softw., vol. 33, n° 2, juin 2007, ISSN 0098-3500, doi : 10.1145/1236463.1236468, adresse : http://doi.acm.org/10.1145/1236463.1236468.

D. Goldberg,
« What every computer scientist should know about floating point arithmetic »,
ACM Computing Surveys, vol. 23, n° 1, p. 5–48, 1991.

F. Goualard,
« Page personnelle », adresse : http://goualard.frederic.free.fr/, 2014.

E. Hairer et G. Wanner,
L'analyse au fil de l'histoire,
vol. Scopos, Springer, Berlin, New York, Paris, ISBN 3-540-67463-2, adresse : http://opac.inria.fr/record=b1096819, 2001.

N. J. Higham,
Accuracy and Stability of Numerical Algorithms,
Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second édition, ISBN 0-89871-521-0, adresse : ftp://ftp.demec.ufpr.br/CFD/bibliografia/Higham_2002_Accuracy%20and%20Stability%20of%20Numerical%20Algorithms.pdf, 2002.

W. Kahan et J. D. Darcy,
« How Java's floating-point hurts everyone everywhere »,
Technical report, inst-BERKELEY-MATH-EECS, inst-BERKELEY-MATH-EECS :adr, adresse : http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf, juin 1998.

W. Kahan,
« Page personnelle », adresse : http://www.cs.berkeley.edu/~wkahan/, 2012.

P. Langlois et N. Louvet,
« Compensated Horner algorithm in K times the working precision »,
Dans RNC-8, Real Numbers and Computer Conference, Santiago de Compostela, Spain, J. Brugera et M. Daumas (coordinateurs), juil. 2008, adresse : http://hal.inria.fr/inria-00267077/fr/,
(HAL-CCSC inria-00267077).

J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé et S. Torres,
Handbook of Floating-Point Arithmetic,
Birkhäuser Boston, 2010,
ACM G.1.0 ; G.1.2 ; G.4 ; B.2.0 ; B.2.4 ; F.2.1., ISBN 978-0-8176-4704-9.

M. Pichat,
« Correction d'une somme en arithmétique à virgule flottante »,
Numer. Math., vol. 19, p. 400–406, 1972.

D. Roegel,
« A reconstruction of the tables of Briggs' Arithmetica logarithmica (1624) »,
Research report, adresse : http://hal.inria.fr/inria-00543939, 2010.

S. M. Rump, T. Ogita et S. Oishi,
« Accurate floating-point summation part i : Faithful rounding »,
SIAM J. Sci. Comput., vol. 31, n° 1, p. 189–224, oct. 2008, ISSN 1064-8275, doi : 10.1137/050645671, adresse : http://dx.doi.org/10.1137/050645671.

J. R. Shewchuk,
« Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates »,
Discrete & Computational Geometry, vol. 18, n° 3, p. 305–363, oct. 1997.

P. Sterbenz,
Floating-point computation,
vol. Prentice-Hall series in automatic computation, Prentice-Hall, ISBN 9780133224955, adresse : http://books.google.fr/books?id=MKpQAAAAMAAJ, 1973.