La boîte de Pandore des mathématiques expérimentales

Guillaume Connan(*)

Les projets de programmes de mathématiques de Terminale des séries générales laissent une large part à l'observation, l'approche intuitive, l'expérimentation le plus souvent assistée par ordinateur, la connaissance de résultats admis. D'autre part, la formation continue et initiale des professeurs est de plus en plus réduite, voire tout simplement supprimée dans certaines académies.
Dans ce court article, nous essaierons de mettre en avant les dangers d'une telle approche de l'enseignement des mathématiques à travers une expérimentation qui trouverait sa place en classe mais conduit à des résultats catastrophiques si elle est menée par des professeurs non formés aux subtilités informatiques et si elle s'adresse à des élèves au bagage mathématique trop superficiel.

1. Avertissement

La scène qui va vous être présentée peut heurter certaines âmes sensibles et nous la déconseillons aux professeurs de plus de quarante ans qui ont passé leur Bac C, qui ont eu droit à une formation initiale en informatique sans logiciel « tout-à-cliquer »(1) et ont même profité d'une formation continue(2) abondante dispensée par des IREM en plein essor. Certains vont sûrement réagir comme devant ces films-catastrophe hollywoodiens qui durent deux heures alors que dans la réalité tout aurait dû s'arrêter au bout de cinq minutes faute de combattants. Et bien non ! Nous irons jusqu'au bout du drame, les monstres irrationnels qui envahiront notre pseudo salle de classe n'arrêteront pas notre héros qui ira de catastrophe en catastrophe, fonçant tête baissée vers le drame inéluctable.
Film gore de série Z ? Scénario de science-fiction complètement irréaliste ? Drame d'anticipation présentant une réalité presque palpable ? À vous de juger…
La longueur et la technicité de l'article peuvent en rendre difficile la lecture pour des collègues peu familiers des questions abordées, entre mathématiques et informatique. Après un parcours global (pour avoir une idée des questions abordées), le lecteur pourra le reprendre en détail par petits bouts, en analysant chacune des séquences et en testant les programmes proposés (avec Python ou un autre langage). C'est à ce prix que l'article donne sa pleine mesure. De même, les travaux suggérés peuvent

(*) IREM de Nantes. guillaume.connan@univ-nantes.fr
(1) L'essor des systèmes libres a remis en question l'idée d'un ordinateur-boîte noire échappant au contrôle de son utilisateur : celui-ci peut, s'il le désire, reprendre la main et ne pas succomber aux sirènes du clic magique qui semble faire ce que l'on veut sans qu'on puisse le vérifier. Celles et ceux qui ont découvert l'informatique jusqu'à la moitié des années 80 étaient de fait dans cette situation.
(2) Pour les plus jeunes, consultez des archives en papier sur ce sujet préhistorique disponibles dans les rares bibliothèques IREM encore accessibles.

être proposés à une classe sur une longue période (un jeu de piste au long cours) : proposer aux élèves des énigmes sur lesquelles ils butent, pour les lever une à une peut devenir le projet d'une année.

2. L'approche « expérimentale »

Dans l'esprit du programme, il peut paraître intéressant d'essayer d'obtenir des approximations de π par des méthodes numériques assistées par ordinateur. Il ne s'agira pas d'utiliser un logiciel de calcul formel mais de faire construire par les élèves des algorithmes de calcul et de les expérimenter à l'aide d'un langage de programmation. Nous utiliserons dans cet article Python 3. Nous avancerons à vue, dans la peau d'un professeur ayant quelques minutes(3) d'avance sur l'expérience qu'il propose.
Ces expériences laissent une large part à l'intuition. De ce point de vue, il paraît évident que la méthode des trapèzes sera plus efficace que la méthode des rectangles et que plus le pas de la subdivision sera petit, meilleure sera l'approximation. Et pourtant…

3. Calcul de π et calcul intégral

Comment faire le lien en Terminale S entre π et le calcul intégral ? La trigonométrie ayant pratiquement disparu de l'enseignement secondaire, on ne peut que se tourner vers l'aire du disque vue dès l'école primaire.
Un premier travail consiste donc à essayer d'orienter les élèves vers le résultat :

$$\pi = 4\int_0^1 \sqrt{1-t^2}\,\mathrm{d}\,t.$$

On introduit ensuite cette fonction, que l'on nommera par la suite f, sur Python. On n'utilisera qu'une version aussi épurée que possible du logiciel. On aura cependant besoin de sqrt qui calcule la racine carrée d'un nombre et de pi qui donne une valeur approchée de π afin de pouvoir la confronter à nos résultats expérimentaux. On importe donc ces deux commandes dans notre session Python :

_Python3 _

```
>>> from math import sqrt,pi
```

Ensuite, on définit la fonction f :

_Python3 _

```
def f(t):
    return sqrt(1–t*t)
```

Le problème peut alors être posé en ces termes : « comment calculer une approximation de l'intégrale de f sur [0,1] ? ».

4. Méthode des rectangles

C'est la plus simple à introduire. Il fut un temps où on l'étudiait en première dans le cadre de la découverte des suites.

(3) Oui, bon, c'est peut-être un peu exagéré…

On donne ici des versions générales pouvant servir dans d'autres expériences. Nos fonctions Python prennent comme argument une fonction *f*, les bornes *a* et *b* de l'intervalle d'intégration et le nombre N de subdivisions.On peut utiliser une boucle for ou while. C'est la deuxième option qui a été choisie ici car elle est plus « parlante ». On aurait pu utiliser une subtilité de Python que l'on retrouve dans de nombreux langages : t+=dt signifie en fait t reçoit t + dt. On utilisera cette notation par la suite. Ici, le gain n'est qu'en écriture, mais quand on travaille sur des listes par exemple, cela fait gagner beaucoup de mémoire … mais ceci est une autre histoire.

_Python3 _

```
def int_rec_droite(f,a,b,N):
    S=0
    t=a
    dt=(b–a)/N
    while t<b:
        S=S+f(t)*dt
        t=t+dt
    return S

def int_rec_gauche(f,a,b,N):
    S=0
    t=a
    dt=(b–a)/N
    while t+dt<b:
        S=S+f(t+dt)*dt
        t=t+dt
    return S
```

On peut alors effectuer nos premiers calculs :

_Python3 _

```
>>> 4*int_rec_gauche(f,0,1,100)
3.1204170317790454
>>> 4*int_rec_gauche(f,0,1,10000)
3.1413914777848557
```

Sans travail sur la majoration de l'erreur, on ne peut pas dire grand chose. Alors on compare avec la valeur approchée pi de π que propose Python dans le module math :

_Python3 _

```
>>> pi–4*int_rec_gauche(f,0,1,10000)
0.0002011758049373924
```

Si on veut faire travailler les élèves sur les boucles, on peut les amener à écrire ceci :

_Python3 _

```
>>> for k in range(8): print(pi–4*int_rec_gauche(f,0,1,10**k))
...
3.141592653589793
0.23707432138101003
0.021175621810747725
0.002037186678767622
```

```
0.0002011758049373924
2.003710526476965e–05
2.001187481948108e–06
1.9943306694969465e–07
```

On observe qu'il semble falloir dix millions d'itérations pour obtenir six bonnes décimales de π.
Pas très efficace… C'est le moment de se souvenir (?) d'un résultat du collège : comment calculer l'aire d'un trapèze ?

5. Méthode des trapèzes

Après divers petits dessins et le rappel de $\dfrac{(\text{petite base} + \text{grande base}) \times \text{hauteur}}{2}$, on arrive à :

_Python3 _

```
def int_trap(f,a,b,N):
    S=0
    t=a
    dt=(b–a)/N
    while t+dt<=b:
        S+=(f(t)+f(t+dt))*dt/2
        t+=dt
    return S
```

Normalement, à vue d'œil, ça devrait être plus précis :

_Python3 _

```
>>> for k in range(8): print(pi–4*int_trap(f,0,1,10**k))
...
1.1415926535897931
0.03707432436124236
0.003996969006682782
0.00012660703439637544
1.1758915650084134e–06
3.7144431530578004e–08
4.015919596866979e–09
–5.628750798791771e–10
```

Bon. Un élève observateur(4) s'étonne tout de même du dernier résultat, les trapèzes ayant l'air de rester malgré tout sous la courbe représentative de f…
«Bonne remarque » répond le professeur, une première goutte de sueur perlant sur son front. « Il s'agit sûrement d'erreurs d'arrondi ».

Pas de répit ! Un autre élève lève la main :
«Monsieur, j'ai trouvé d'autres résultats avec la boucle for ».
Argh ! Y en a toujours un pour ne pas écouter les consignes et faire ses calculs dans son coin. Sur l'écran de l'élève il y a en effet :

(4) C'est quand même une des qualités principales mises en avant par les programmes !...

_Python3 _

```
>>> def int_trap(f,a,b,N):
    S=0
    dt=(b–a)/N
    for k in range(N):
        S+=(f(a+k*dt)+f(a+(k+1)*dt))*dt/2
    return S
… … … … …
>>> for k in range(8): print(pi4*int_trap(f,0,1,10**k))
...
1.1415926535897931
0.037074327341474866
0.001175621810749039
3.7186678768730275e–05
1.1759784679377105e–06
3.718782926043218e–08
1.1759344609174605e–09
3.643840784661734e–11
```

Une nouvelle goutte de sueur perle sur le front du professeur. Il lui revient en tête des pages d'un livre sur l'algorithmique qu'il a lu dans un souci d'autoformation sur le sujet : *test d'arrêt*, *invariant de boucle*, tout se mélange dans sa tête(5).
« Vous avez vu monsieur, il n'y a plus de résultat négatif ».
Pour détourner le propos, le professeur propose aux élèves 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, le professeur a un peu peur que cela crée des perturbations. Et intuitivement, il se dit qu'en limitant l'intervalle d'intégration, on devrait limiter l'ampleur de l'approximation.
Le problème, c'est qu'il faut faire un peu de trigonométrie : trouver le cosinus et le sinus de $\frac{\pi}{3}$. Allez, soyons fous...
Après un petit dessin au tableau, le professeur amène donc ses élèves à montrer que :

$$\pi = 12\left(\int_0^{\frac{1}{2}} f(t)\,\mathrm{d}t - \frac{\sqrt{3}}{8}\right).$$

Mais là, ça devient très bizarre :

_Python3 _

```
>>> for k in range(7): print(pi–12*( int_trap(f,0,1/2,10**k)–sqrt( 3)/8))
…
0.14159265358979312
0.0014430555975519788
0.05206198505358994
0.005197162208523842
1.4433143569192453e–09
1.3836487511298401e–11
5.1961539169198545e–06
```

(5) Ce résultat sera expliqué plus loin.

Il n'y a plus de corrélation entre la finesse de la subdivision et la précision du calcul ! La précision a l'air d'aller et venir comme la marée... Il y a quelque chose qui cloche là-dedans, mais quoi ? Un ordinateur, ça ne se trompe pas dans les calculs !
Ah, et revoilà l'élève avec sa boucle for :
« Moi ça marche monsieur » :

_Python3 _

```
>>> for k in range(7): print(pi–12*(int_trap(f,0,1/2,10**k)–sqrt(3)/8))
…
0.14159265358979223
0.0014430555975519788
1.4433724657703095e–05
1.443375703402694e–07
1.4433609862862795e–09
1.439337538045038e–11
1.1457501614131615e–13
```

Il faut lui clouer le bec ! Allons vers un problème mathématique plus compliqué.
On a utilisé des segments de droite horizontaux, des segments de droite obliques…
Et si l'on utilisait des segments de parabole : intuitivement, ça colle plus à la courbe, ça devrait être plus précis.

6. Méthode de Simpson

Ce n'est pas au programme, même pas au programme de Maths Sup, mais on nous demande d'observer, alors on peut sortir une formule de son chapeau et expliquer à peu près d'où elle vient ou on peut en faire un petit sujet de recherche à la maison…
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 :

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

Ouh la la, c'est un système 3×3. On peut faire appel à un logiciel de calcul formel comme préconisé dans le programme quand il y a de gros calculs… Ou résoudre ce système à la main !

On trouve alors $c_0 = c_2 = \dfrac{h}{6}$ et $c_1 = \dfrac{4}{6}h$, puis, toujours pour $f(\boldsymbol{x})$ valant 1, x ou x^2 :

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

On peut se contenter de donner directement cette formule.
Finalement, le tout est de l'utiliser avec Python :

_Python3 _

```
def int_simps(f,a,b,N):
    S=0
    t=a
    dt=(b–a)/N
    while t+dt<=b:
        S+=(f(t)+4*f(t+dt/2)+f(t+dt))*dt/6
        t+=dt
return S
```

Ce qui est étonnant, c'est qu'on trouve à peu près la même précision qu'avec la méthode des trapèzes :

_Python3 _

```
>>> for k in range(8): print(pi–4*int_simps(f,0,1,10**k))
…
0.16552491016462367
0.00514558833917933
0.000766184709476761
0.0001192583254159274
1.6234044686314064e–07
5.117280821309578e–09
3.783345636776403e–09
–5.92078830408127e–10
```

Pourquoi pas ? Et si nous prenions notre petit arc de cercle :

_Python3 _

```
>>> for k in range(8): print(pi–12*( int_simps(f,0,1/2,10**k)–sqrt(3)/8))
…
0.0006601149512537319
7.997674877913141e–08
0.0520477428322228
0.005197018063311631
–2.042810365310288e–14
–5.88862292261183e–13
5.1961537730349505e–06
–8.213874025386758e–11
```

Alors là, on ne comprend carrément plus rien ! Ça échappe totalement à notre intuition !
L'élève à la boucle for ne lève plus la main ! Le professeur va quand même observer son écran

_Python3 _

```
>>> for k in range(8): print(pi–12*( int_simps(f,0,1/2,10**k)–sqrt(3)/8))
…
0.0006601149512537319
7.997674877913141e–08
8.021139308311831e–12
5.329070518200751e–15
2.1316282072803006e–14
–3.907985046680551e–14
–3.375077994860476e–14
–5.284661597215745e–13
```

Chez lui aussi, c'est n'importe quoi…

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 le réchauffement de la planète et la fonte des glaces, c'est n'importe quoi.
Ou bien un autre scientifique (ou le même…) peut jeter un caillou dans l'eau la nuit et prendre des photos au flash toutes les T secondes avec T la période de l'onde créée par le caillou et en conclure que jeter un caillou dans l'eau ne produit aucun effet sur la surface de l'eau si le hasard fait bien les choses.
Ici aussi, on observe sans garde-fou théorique une situation et on peut faire et en conclure absolument n'importe quoi pour peu que le hasard du choix de nos mesures ne nous confronte pas aux problèmes, jusqu'à ce qu'enfin, une catastrophe survienne…

7. Que faire ?...

Finalement, on ne va pas donner cette activité expérimentale aux élèves car elle semble troubler plus qu'éclairer les esprits.
Ou bien, soyons fous, nous allons faire un peu de mathématiques et d'informatique et ne pas se contenter d'observer des résultats au petit bonheur pour chercher à comprendre ce qui se passe.
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 $2{,}2 \times 10^{-16}$. En fait, 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 flottant suivant le plus proche :

_Python3 _

```
>>> from sys import*
>>> float_info.epsilon
2.220446049250313e–16
```

Eh oui, la mémoire de l'ordinateur n'est pas infinie. Ses « réels » admettent des successeurs.
Mais attention, entre 0 et 1, les choses sont différentes :

_Python3 _

```
>>> from sys import*
>>> e=float_info.epsilon
>>> e
2.220446049250313e–16
>>> e/2
1.1102230246251565e–16
>>> 1+e/2
1.0
>>> 1+e/2==1
True
>>> 0+e/2==0 False
>>> e*1e16
2.220446049250313
>>> –1–e/ 2
–1.0
>>> 1–e/2
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.
On peut définir une fonction qui illustrera ce phénomène :

_Python3 _

```
def 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 :

_Python3 _

```
>>> egal_float(0.1 + 0.1 + 0.1 , 0.3)
True
>>> 0.1 + 0.1 + 0.1 == 0.3
False
>>> 3 * 0.1 == 0.3
False
>>> 4 * 0.1 == 0.4
True
>>> 10+float_info.epsilon–10
0.0
>>> 1+float_info.epsilon–1
```

```
2.220446049250313e–16
>>> 10+10*float_info.epsilon–10
1.7763568394002505e15
>>> x=0.1
>>> 3*x–0.3
5.551115123125783e17
>>> 4*x–0.4
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.
Pour le faire comprendre aux élèves, on peut faire écrire 0,25 en base 2 : il suffit de poser la division comme à l'école primaire, mais avec les dividende et diviseur écrits en base 2.

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 ; posons la division :

1	1010
10	0,00011
100	
1000	
10000	
– 1010	
1100	
– 1010	
10	

On retrouve alors 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 24 bits pour la mantisse, faire demême pour 0,3 et $3 \times 0,1$ et découvrir pourquoi 0.3 – 3 * 0.1 est non nul.
Reprenons l'affinement successif de notre subdivision mais avec des nombres de subdivisions égaux à des puissances de 2 :

_Python3 _

```
>>> for k in range(13): print(pi–12*( int_simps(f,0,1/2,2**k)–sqrt(3)/8))
…
0.0006601149512537319
4.72214266773463e–05
3.0824728671774437e–06
1.9496909775540416e–07
```

```
1.222303502856903e–08
7.645306610015723e–10
4.779110440722434e–11
2.9887203822909214e–12
1.8474111129762605e–13
7.993605777301127e–15
–4.440892098500626e–16
–1.3766765505351941e–14
–8.881784197001252e–15
```

Tout a l'air de bien fonctionner jusqu'à 2^{10}, mais ensuite on arrive aux alentours de epsilon et ça commence à se détraquer informatiquement comme nous l'évoquerons dans l'appendice.
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 :

_Python3 _

```
>>> f(1e–8)
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 :

_Python3 _

```
>>> 1+0.1+0.1+0.1–0.3<= 1
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 :

_Python3 _

```
>>> for k in range(11):
    print(pi–12*( int_rec_gauche(f,0,1/2,2**k)–sqrt(3)/8),"\t",\
        pi–12*( int_trap(f,0,1/2,2**k)–sqrt(3)/8),"\t",\
        pi–12*( int_simps(f,0,1/2,2**k)–sqrt(3)/8))
… … … …
0.5435164422364771      0.14159265358979312     0.0006601149512537319
0.23685514393423013     0.035893249610888134    4.72214266773463e–05
0.10948967563440082     0.009008728472729821    3.0824728671774437e–06
0.052494967553668115    0.0022544939728321722   1.9496909775540416e–07
0.025684006510449997    0.0005637697200326919   1.222303502856903e–08
0.012701069992491654    0.00014095159728411133  7.645306610015723e–10
0.006315297670323794    3.5238472716692115e–05  4.779110440722434e–11
```

0.00314883925282583	8.8096540218352e–06	2.9887203822909214e–12
0.0015722172151479974	2.2024157457778415e–06	1.8474111129762605e–13
0.0007855580037752219	5.506040738900708e–07	7.993605777301127e–15
0.00039264135087524465	1.3765102657714579e–07	–4.440892098500626e–16

On distingue mieux cette fois la hiérarchie des méthodes selon leur efficacité.
«Monsieur,monsieur, regardez, on peut améliorer l'aspect ! Avec ça : »

_Python3 _

```
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)))
```

« J'ai un beau tableau »

_Python3 _

rectangle gauche	trapeze	simpson
5.4351644223647710e–01	1.4159265358979312e–01	6.6011495125373187e–04
2.3685514393423013e–01	3.5893249610888134e–02	4.7221426677346301e–05
1.0948967563440082e–01	9.0087284727298211e–03	3.0824728671774437e–06
5.2494967553668115e–02	2.2544939728321722–e03	1.9496909775540416e–07
2.5684006510449997e–02	5.6376972003269188e–04	1.2223035028569029e–08
1.2701069992491654e–02	1.4095159728411133e–04	7.6453066100157230e–10
6.3152976703237940e–03	3.5238472716692115e–05	4.7791104407224339e–11
3.1488392528258302e–03	8.8096540218352004e–06	2.9887203822909214e–12
1.5722172151479974e–03	2.2024157457778415e–06	1.8474111129762605e–13
7.8555800377522189e–04	5.5060407389007082e–07	7.9936057773011271e–15

Pfff... Il y a toujours un élève pour s'occuper plus de la forme que du fond

Nous n'avons pas le choix : il est temps à présent d'invoquer les mathématiques pour préciser un peu ces observations, même si ça 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.
On peut également voir la méthode de SIMPSON comme le premier pas de l'accélération de convergence de ROMBERG de la méthode des trapèzes… Mais bon, on laissera ça pour plus tard.
Nous allons nous contenter dans un premier temps d'étudier l'erreur commise par la méthode des trapèzes. Nous avons juste besoin du théorème de ROLLE qui peut être étudié en Terminale à titre d'exercice.
On considère une fonction f de classe C^2 sur un intervalle $[a,b]$. On considère une subdivision régulière de $[a,b]$ et on pose :

$$x_j = a + j\frac{b-a}{\mathrm{N}}$$

avec N le nombre de subdivisions. On cherche une fonction polynomiale P de degré au plus 1 telle que $\mathrm{P}(a) = f(a)$ et $\mathrm{P}(b) = f(b)$.
L'intégrale de P sur $[a,b]$ vaut

$$\mathrm{J} = (b-a)\frac{f(a)+f(b)}{2}.$$

Soit $x \in\ ]a,b[$. On introduit la fonction g_x définie sur $[a,b]$ par :

$$g_x(t) = f(t) - \mathrm{P}(t) + k_x(t-a)(b-t)$$

avec k_x choisi tel que $g_x(x) = 0$.
La fonction g_x est de classe C^2 et s'annule en a, x et b donc on peut appliquer le théorème de Rolle sur $]a,x[$ et $]x,b[$ et g_x' s'annule donc sur chacun de ces intervalles.

On applique alors une nouvelle fois le théorème de Rolle à g_x' sur $]a,b[$ et g_x'' s'annule donc au moins une fois en un réel c_x de $]a,b[$.
On obtient alors que

$$k_x = \frac{f''(c_x)}{2}.$$

On obtient finalement que pour tout $x \in\ ]a,b[$, il existe $c_x \in\ ]a,b[$ tel que :

$$f(x) - \mathrm{P}(x) = -(x-a)(b-x)\frac{f''(c_x)}{2}.$$

On peut évidemment inclure les bornes de l'intervalle.

Soit M_2 un majorant de $|f''|$ sur $[a,b]$. On obtient donc, par intégration de l'égalité précédente :

$$\left|\int_a^b f(x)\,\mathrm{d}x - \mathrm{J}\right| \leq \frac{\mathrm{M}_2}{12}(b-a)^3,$$

puis, par application de cette inégalité sur chaque intervalle $\left[x_j, x_{j+1}\right]$:

$$\left|\int_a^b f(x)\,\mathrm{d}x - \mathrm{J}_\mathrm{N}\right| \leq \frac{\mathrm{M}_2}{12}\frac{(b-a)^3}{\mathrm{N}^2},$$

avec

$$\mathrm{J}_\mathrm{N} = \frac{b-a}{\mathrm{N}}\left(\frac{f(a)}{2} + \sum_{j=1}^{\mathrm{N}-1} f(x_j) + \frac{f(b)}{2}\right).$$

On obtient donc que quand N est multiplié par 2, le majorant de l'erreur est divisé par 4.
Cela correspond en effet à ce que nous observons avec Python :

_Python3 _

```
>>> for k in range(1,15):
    print(p–12*( int_trap(f,0,1/2,2**k)–sqrt(3)/8),"\t",\
```

```
        (pi–12*( int_trap(f,0,1/2,2**(k1))–sqrt(3)/8))/4)
… … …
0.035893249610888134    0.03539816339744828
.0.009008728472729821   0.008973312402722033
0.0022544939728321722   0.0022521821181824553
0.0005637697200326919   0.0005636234932080431
0.00014095159728411133  0.00014094243000817297
3.5238472716692115e–05  3.523789932102783e–05
8.8096540218352e–06     8.809618179173029e–06
2.2024157457778415e–06  2.2024135054588e–06
5.506040738900708e–07   5.506039364444604e–07
1.3765102657714579e–07  1.376510184725177e–07
3.4412758864732496e–08  3.4412756644286446e–08
8.603182166666556e–09   8.603189716183124e–09
2.1507857717040224e–09  2.150795541666639e–09
5.37708544356974e–10    5.376964429260056–10
```

Dans la colonne de gauche se trouvent les résultats déjà calculés.Dans la colonne de droite, le résultat de la ligne i est celui de la ligne $i-1$ de la colonne de gauche divisé par 4 : l'erreur sur le majorant et sur le calcul lui-même est bien corrélée.

On peut montrer de même qu'avec la méthode de SIMPSON, si f est de classe C^4 sur $[a,b]$, alors le majorant de l'erreur est

$$\frac{M_4}{2\,880}\frac{(b-a)^5}{N^4}$$

avec M_4 un majorant de $f^{(4)}$ sur $[a,b]$.
Cette fois, on obtient que quand N est multiplié par 2, le majorant de l'erreur est divisé par 16. On observe bien le même phénomène qu'avec la méthode des trapèzes :

_Python3 _

```
>>> for k in range(1,9):
        print(pi–12*( int_simps(f,0,1/2,2**k)–sqrt(3)/8),"\t",\
          (pi–12*( int_simps(f,0,1/2,2**(k1))–sqrt(3)/8))/16)
… … …
4.72214266773463e–05    4.125718445335824e–05
3.0824728671774437e–06  2.951339167334144e–06
1.9496909775540416e–07  1.9265455419859023e–07
1.222303502856903e–08   1.218556860971276e–08
7.645306610015723e–10   7.639396892855643e–10
4.779110440722434e–11   4.778316631259827e–11
2.9887203822909214e–12  2.986944025451521–e12
1.8474111129762605e–13  1.867950238931826e–13
```

Alors, sommes-nous condamnés à ne pas dépasser une précision de 16 décimales ? Pourtant on parle du calcul d'un milliard de décimales de π. Pourtant, les logiciels de calcul formel dépassent eux aussi cette limitation.
Pour cela, il faut repenser totalement la façon de calculer et réfléchir à des

algorithmes permettant de passer outre la limitation de représentation des nombres. Mais ceci est une autre histoire…
Moralité, comme aurait pu le dire un contemporain français de NEWTON, *démonstration mathématique et domination de la machine valent mieux qu'observation et utilisation au petit bonheur ou encore les mathématiques ne sont pas uniquement une science expérimentale assistée par ordinateur*(6) On veut faire une petite expérience toute simple de calcul et on ne se rend pas compte qu'on vient d'ouvrir la boîte de Pandore… Mais comment s'en rendre compte quand les professeurs sont submergés de nouvelles notions à enseigner et qu'ils ne disposent que de l'autoformation pour s'en sortir où à la rigueur d'une « formation » de Bassin d'une journée…
Pour plus de précisions sur l'intégration numérique, on peut se reporter à la thèse soutenue par Laurent FOUSSE [FOUSSE, 2006] en 2006 sur le sujet et au cours de L2 de Bernard PARISSE [PARISSE, 2010].

8. Appendice : quelques mots sur la manipulations des flottants

En 1985, une norme de représentations des nombres a été proposée afin, entre autres, de permettre de faire des programmes portables : il s'agit de la norme IEEE-754 (Standard for Binary Floating-Point 13 Arithmetic).
En simple précision, un nombre est représenté sur 32 bits (en fait 33…) :
– le premier donne le signe ;
– un 1 implicite puis 23 bits pour la partie fractionnaire ;
– 8 bits pour l'exposant.
Les 24 bits qui ne sont ni le signe, ni l'exposant représentent la mantisse qui appartient à [1,2[.
Notons s_x le bit de signe, e_x les bits d'exposant, $m_x = 1 + f_x$ la mantisse avec fx la partie fractionnaire. Il y a deux zéros, –0 et +0 car tout nombre est signé. Par exemple :
– $s_{-0} = 1$;
– $b_{-0} = 00000000$
– $f_{-0} = 00000000000000000000000$
Il y a aussi deux infinis. Par exemple $s_{-\infty} = 1$, $e_{-\infty} = 11111111$ et $f_{-\infty} = 0$…
Il y a enfin le *Not a Number*, noté NaN, qui est codé comme $+\infty$, mais avec une partie fractionnaire non nulle.

La majorité des résultats troublants observés vient des erreurs d'arrondis, les deux principales étant l'*élimination* et l'*absorption*.
La première intervient lors de la soustraction de deux nombres très proches. Par exemple :

$$\begin{array}{l} \quad 1.10010010000111111011011 \\ - \ 1.10010010000110000000000 \\ \hline = \ 0.00000000000111111011011 \end{array}$$

La mantisse est ensuite renormalisée pour devenir :

$$1.11111011011000000000000$$

(6). Toute allusion aux nouveaux programmes du lycée ne serait que pure coïncidence…

Les zéros ajoutés à droite sont faux.
L'absorption intervient lorsqu'on additionne deux nombres d'ordre de grandeur très différents : les informations concernant le plus petit sont perdues.

$$\begin{array}{r} 1.10010010000111110001011 \\ + \qquad\qquad\qquad 1.00000001111 \\ \hline = 1.10010010000111111010010110111 1 \end{array}$$

Mais après normalisation, on a :

1.10010010000111110100101

Voici un exemple classique : le calcul de $\sum_{k=1}^{2^{15}} \frac{1}{k^2}$ dans un sens puis dans l'autre.

_Python3 _

```
def som_croissante(N):
    S=0
    for k in range(1,N+1):
        S+=1/k**2
    return S

def som_decroissante(N):
    S=0
    for k in range(N,0,1):
        S+=1/k**2
    return S
```

Alors :

_Python3 _

```
>>> som_croissante(2**15)
1.6449035497357558
>>> som_decroissante(2**15)
1.6449035497357580
```

En fait, le résultat correct est :

_Python3 _

```
sage: k=var('k')
sage: s=sum((1/k**2),k,1,2**15
sage: s.n(digits=20)
1.6449035497357579868
```

Les conséquences de telles erreurs non prises en compte peuvent être plus graves qu'une séance de TP ratée.
Le 4 juin 1996 par exemple, la fusée Ariane 5 a explosé en vol, trente secondes après son décollage. Après enquête, il s'est avéré que la vitesse horizontale de la fusée par rapport au sol était calculée sur des flottants 64 bits, puis convertie en entier signé 16 bits. Cette méthode avait été appliquée sur Ariane 4 avec succès car sa vitesse était

plus faible donc tenait sur un entier 16 bits, mais ce n'était plus le cas pour Ariane 5...[7]
Un problème d'élimination dans l'horloge des missiles *Patriot* a causé également la mort de dizaines de personnes pendant la première guerre du Golfe[8].
Pour un exposé beaucoup plus détaillé sur l'arithmétique flottante, voir LEFÈVRE et ZIMMERMANN [2004].

9. Épilogue : bye bye π ?

Et bien non !
Python possède un module de calcul en multiprécision avec les quatre opérations arithmétiques de base et la racine carrée :

_Python3 _

```
>>> from decimal import*
# on règle la précision à 30 chiffres
>>> getcontext().prec = 30
# on rentre l es nombres entre apostrophes, comme des chaînes
>>> deux=Decimal('2')
>>> deux.sqrt()
Decimal('1.41421356237309504880168872421')
>>> getcontext().prec = 50
>>> deux.sqrt()
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* ;
– $|R_n| \leq |u_{n+1}|$ *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 [voir par exemple Rodot, 2010, p. 53].

(7) Voir le rapport de la commission présidée par J.L. LIONS disponible à cette adresse : http ://www.ima.umn.edu/ arnold/disasters/ariane5rep.html
(8) Voir http ://ta.twi.tudelft.nl/nw/users/vuik/wi211/disasters.html

On en déduit donc que :

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

On peut alors utiliser Python pour déterminer une fonction mini_greg(a,d) avec $0<a<1$ déterminant une valeur de n telle que $\dfrac{a^{2n+3}}{2n+3}\leq 10^{-d}$:

_Python3 _

```
def mini_greg(a,d):
    getcontext().prec = d+2
    n=1
    ad=Decimal('1')/Decimal(str(a))
    D=Decimal('1e-'+ str(d))
    nd=ad**3
    dd=Decimal('3')
    fd=nd/ad
    while fd > D :
        nd*=ad*ad
        dd+=2
        fd=nd/ad
        n+=1
    return n
```

Elle n'est pas très efficace. On peut toutefois remarquer que l'inégalité équivaut à

$$\left(2+\frac{3}{n}\right)\log\frac{1}{a}\geq\frac{d}{n}-\frac{\log(2n+3)}{n}.$$

Donc pour n grand, le rapport entre le nombre de décimales cherchées et le nombre de termes nécessaires est de l'ordre de $2\log\dfrac{1}{a}$..

On peut établir les formules d'addition des tangentes puis montrer que, a et b étant deux réels tels que $ab\neq 1$, on a

$$\arctan a+\arctan b=\arctan\left(\frac{a+b}{1-ab}\right)+\varepsilon\pi$$

avec $\varepsilon=0$ si $ab<1$, $\varepsilon=1$ si $ab>1$ et $a>0$ et $\varepsilon=-1$ si $ab>1$ et $a<0$.
Seul le premier cas va nous intéresser dans ce paragraphe.
En 1706, l'anglais John MACHIN en déduisit 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 polynôme de GREGORY le polynôme

$$G_n(X)=\sum_{k=0}^{n}\frac{(-1)^k\,X^{2k+1}}{2k+1}.$$

Écrivons une fonction greg(a,N) qui donne une valeur de $G_N(a)$:

_Python3 _

```
def greg(a,N):
    ad=Decimal('1')/Decimal(str(a))
    nd=ad
    dd=Decimal('1')
    s=nd/dd
    for k in range(1,N):
        nd*=Decimal('1')* ad*ad
        dd+=Decimal('2')
        s+=nd/dd
    return s
```

Il ne reste plus qu'à l'utiliser dans la formule de Machin :

_Python3 _

```
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))
```

Ce n'est pas très optimal car on lance deux fois greg.
On récupère une approximation de π quelque part pour vérification et on compare juste pour le plaisir (enfin pas seulement car on a un peu peur des erreurs d'arrondi !) :

_Python3 _

```
>>> PI–pi_ machin(1000)
Decimal('-1.5E1000')
```

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

_Python3 _

```
>>> pi_machin(1000)
Decimal('3.14159265358979323846264338327950288419716939937510582097494459230781640628\
62089986280348253421170679821480865132823066470938446095505822317253594081284811174 50\
28410270193852110555964462294895493038196442881097566593344612847564823378678316527 12\
01909145648566923460348610454326648213393607260249141273724587006606315588174881520 92\
09628292540917153643678925903600113305305488204665213841469519415116094330572703657 59\
59195309218611738193261179310511854807446237996274956735188575272489122793818301194 91\
29833673362440656643086021394946395224737190702179860943702770539217176293176752384 67\
48184676694051320005681271452635608277857713427577896091736371787214684409012249534 30\
14654958537105079227968925892354201995611212902196086403441815981362977477130996051 87\
07211349999998372978049951059731732816096318595024459455346908302642522308253344685 03\
52619311881710100031378387528865875332083814206171776691473035982534904287554687311 59\
56286388235378759375195778185778053217122680661300192787661119590921642019915')
```

Pour d'autres formules avec des arctangentes, on pourra se référer à un vieil article de la revue PLOT [Briand, 1991].
Moralité : l'arithmétique des ordinateurs, c'est un domaine très compliqué. Il vaut mieux éviter ce genre de désagrément aux élèves et les faire travailler avec

des logiciels de calcul formel comme XCAS ou permettant de ne pas s'inquiéter de la précision des calculs. Ou alors, on peut faire travailler les élèves sur quelques notions d'arrondis en base 2. Dans tous les cas, il faudra que le professeur soit bien conscient des dangers potentiels.

10. Post-scriptum

Les erreurs présentées ici ne sont pas dues à Python et on les retrouve sur d'autres langages :

– avec CAML :

```
#0.3–.3*.0.1;;
–: float = –5.5511151231257827e–17
```

– avec SCILAB :

```
–> 0.3–3* 0.1
 ans =
.–5.551D–17
```

– avec giac/XCAS :

```
1>> 0.3–3* 0.1
1.42108547152e–14
// Time 0
```

– avecMAXIMA :

```
(%i1) 0.3–3* 0.1;
(%o1)                –5.5511151231257827E–17
```

– avec GP/PARI :

```
? 0.3–3* 0.1
%1 = 1.4693679385278593850 E–39
```

11. Remerciements

Je tiens à remercier vivement Gérard KUNTZ, Bernard PARISSE, Laurent FOUSSE, François SAUVAGEOT et Jean-Philippe VANROYEN pour leur relecture attentive et leurs conseils précieux.

Références

G. BRIAND, « π et la galaxie des arctangentes », PLOT, vol. Nº 54, p. 23–32, 1991.

L. FOUSSE, *Intégration numérique avec erreur bornée en précision arbitraire*, PhD in Computer Science, Université Henri Poincaré – Nancy 1, adresse : http://komite.net/laurent/data/pro/publis/these20070228.pdf, 2006.

V. LEFÈVRE ET P. ZIMMERMANN, « Arithmétique flottante », Research Report RR-5105, INRIA, adresse : http://hal.inria.fr/inria00071477/ PDF/ RR5105. pdf, 2004.

B. PARISSE, «Mathématiques assistées par ordinateur », Cours, Institut Fourier (CNRS UMR 5582) - Université de Grenoble I, adresse : http:// wwwfourier. ujfgrenoble. fr/~parisse/mat249/mat249.pdf, 2010.

O. RODOT, *Analyse mathématique, une approche historique*, De Boeck, 2010.