Le calcul pratique de π

L'exemple des algorithmes compte-gouttes

Découvrons les principes généraux utilisés pour calculer, à la main ou avec une machine, des nombres avec une longue suite de décimales. Nous donnons l'exemple d'un programme à la fois court (158 caractères) et astucieux qui calcule π avec 2 400 décimales, et nous en expliquons le fonctionnement. En le transposant, on obtient une méthode de calcul de π à la main qui permet d'obtenir plus de décimales de π que Ludolff von Ceulen (35) ou que Johann Dahse (200) en quelques heures ou... quelques jours. Le chapitre se termine par de brèves remarques sur les techniques d'accélération de la convergence.

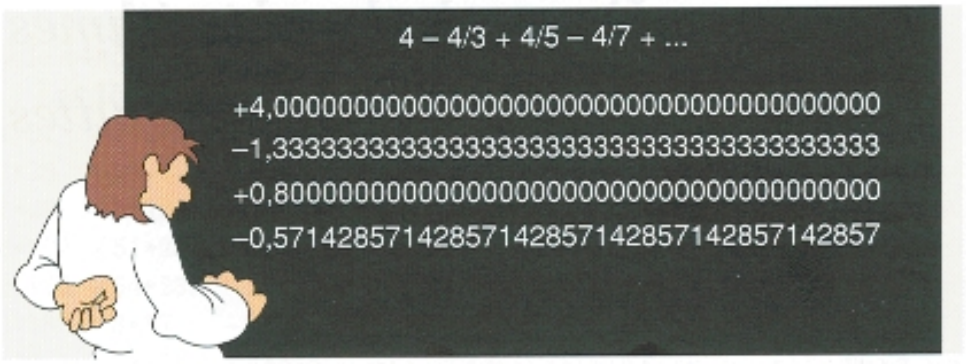
Principes généraux de la pratique du calcul de π

Pour calculer π avec un grand nombre de décimales en utilisant une formule de série, les principes de base sont les mêmes, que l'on procède à la main ou avec une machine. Il faut d'emblée calculer avec beaucoup de chiffres. Si vous voulez 1 000 décimales, et si votre série comporte comme premier terme 1/3, vous commencerez sans doute par écrire sur une feuille de papier ou dans un tableau de mémoires de l'ordinateur : 0,3333333...33 (avec 1 000 fois le «3»).

Pour effectuer les opérations élémentaires d'addition, de multiplication et de division (ou d'extraction de racine carrée, voir l'annexe), on procède comme on l'a appris à l'école, en prenant une feuille de papier et en fractionnant éventuellement le travail, si l'on calcule à la main, ou en programmant le procédé de l'école si l'on utilise un ordinateur.

Les opérations de l'ordinateur sont prévues pour être menées avec des nombres de quelques chiffres (10 par exemple). On ne peut pas les utiliser telles quelles avec les grands nombres : il faut donc programmer les opérations élémentaires en les ramenant à des opérations entre nombres courts. Pour exploiter au mieux les opérations arithmétiques disponibles, il est souvent intéressant de travailler dans une base supérieure à 10, car cela réduit le nombre de «chiffres» à traiter ; une bonne idée est de travailler en base 100 (ou 1 000, etc.) car, à la fin du calcul, aucune conversion n'est nécessaire : par exemple, si un nombre en base

Pour calculer un grand nombre de décimales de π, par exemple avec la série de Madhava-Gregory-Leibniz, il faut calculer dès le départ avec un grand nombre de décimales.



1 000 s'écrit comme la suite des trois «chiffres» 324, 746 et 783, son écriture en base 10 est la suite des neuf chiffres 3, 2, 4, 7, 4, 6, 7, 8, 3.

Le choix de la base de calcul b est aussi fait de manière à tirer le meilleur parti de l'arithmétique offerte par le langage de programmation, où les calculs sont par exemple limités aux entiers inférieurs à $65\,536 = 2^{16}$. Pour multiplier des entiers en base b, il faut connaître la table de multiplication entre entiers $\leq b$ (c'est pour cela que nous apprenons la table de multiplication des chiffres décimaux). Comme les résultats d'une telle table varient entre 0 et b^2, il faut choisir b de telle façon que tous les résultats de la table soient correctement calculés, donc inférieurs à 65 536 (dans notre exemple). Cette contrainte, ajoutée à celle du paragraphe précédent, conduit fréquemment à choisir pour base b la plus grande puissance de 10 dont le carré est inférieur à l'entier maximum du langage. Dans notre exemple, celui-ci est égal à 65 536, d'où $b = 100$.

Au chapitre suivant, nous verrons que la programmation de la multiplication par la méthode usuelle apprise à l'école n'est pas la plus efficace. Pour arriver au million de décimales, les programmeurs se sont contentés d'utiliser la méthode usuelle, mais pour aller plus loin, ils ont dû utiliser les méthodes plus perfectionnées.

Toutes sortes d'astuces pour écourter les calculs, éviter de faire deux fois la même opération et contrôler les résultats intermédiaires et finaux, sont possibles et sont effectivement mises en œuvre par les calculateurs de π.

Programmes mystérieux

Pour programmer une méthode utilisant des arcs tangentes, ce qui conduit à multiplier des grands nombres, une redéfinition de l'arithmétique est nécessaire. Ces nouvelles instructions allongent les programmes, mais certaines astuces permettent d'en limiter la taille totale. Le record en la matière est sans doute le programme suivant en langage C, qui calcule 2 400 décimales de π. Son auteur, inconnu, a réalisé un véritable exploit en n'utilisant que 158 caractères.

Obtenir 2 400 décimales de π
avec un programme de 158 caractères en langage C :

```
int a=10000,b,c=8400,d,e,f[8401],g;main(){for(;b-c;)
f[b++]=a/5;for(;d=0,g=c*2;c-=14,printf("%.4d",e+d/a),
e=d%a)for(b=c;d+=f[b]*a,f[b]=d%--g,d/=g--,--b;d*=b);}
```

(pour certains compilateurs C, il faut changer int a=10000 par longint a=10000).

Le même programme traduit en Basic :

```
DEFLNG a-g : DIM f(8401) AS LONG
  a = 10000 : c = 8400
  WHILE (b <> c)
        f(b) = a \ 5 : b = b + 1
  WEND
  WHILE (c > 0)
        g = 2 * c : d = 0 : b = c
        WHILE (b > 0)
              d = d + f(b) * a : g = g - 1 : f(b) = d MOD g
              d = d \ g : g = g - 1 : b = b - 1
              IF (b <> 0) THEN d = d * b
        WEND
        c = c - 14 : x$ = STR$ (e + d \ a) : L = LEN (x$)
        PRINT LEFT$ ("0000", 5 - L) ; RIGHT$ (x$, L - 1) ;
        e = d MOD a
  WEND
```

J'avais proposé ces programmes en mai 1994 dans un article de la revue *Pour la Science* (le premier m'avait été communiqué par Éric Wegrzynowski qui l'avait trouvé sur réseau *Internet*, le second est une traduction effectuée par Philippe Mathieu). Je n'expliquais pas leur fonctionnement, car je ne le connaissais pas à l'époque. Un certain nombre de lecteurs m'ont alors écrit pour me proposer des solutions. Les meilleures étaient celles de Francis Dalaudier, Emmanuel Dimarellis, François Balsalobre, Alain Desprès, Robert Domain, Claude Chaunier, Gilles Esposito-Farèse, Jean-Paul Michel, René Manzoni et Daniel Saada, auteur d'un article sur cette méthode de calcul de π *(voir la bibliographie)*. Je les remercie, car c'est grâce à leur travail que je peux donner des éclaircissements sur ce qui apparaissait auparavant comme un miracle. Rien n'illustre mieux l'ingéniosité mise en œuvre dans certains programmes de calcul de π.

Signalons au préalable que certains langages possèdent des instructions spéciales donnant un accès immédiat à un grand nombre de décimales de π. Par exemple, en langage *Maple* (langage de calcul formel

très utilisé à la fois par les ingénieurs et dans les divers cycles d'enseignement scientifique), le programme d'une ligne :

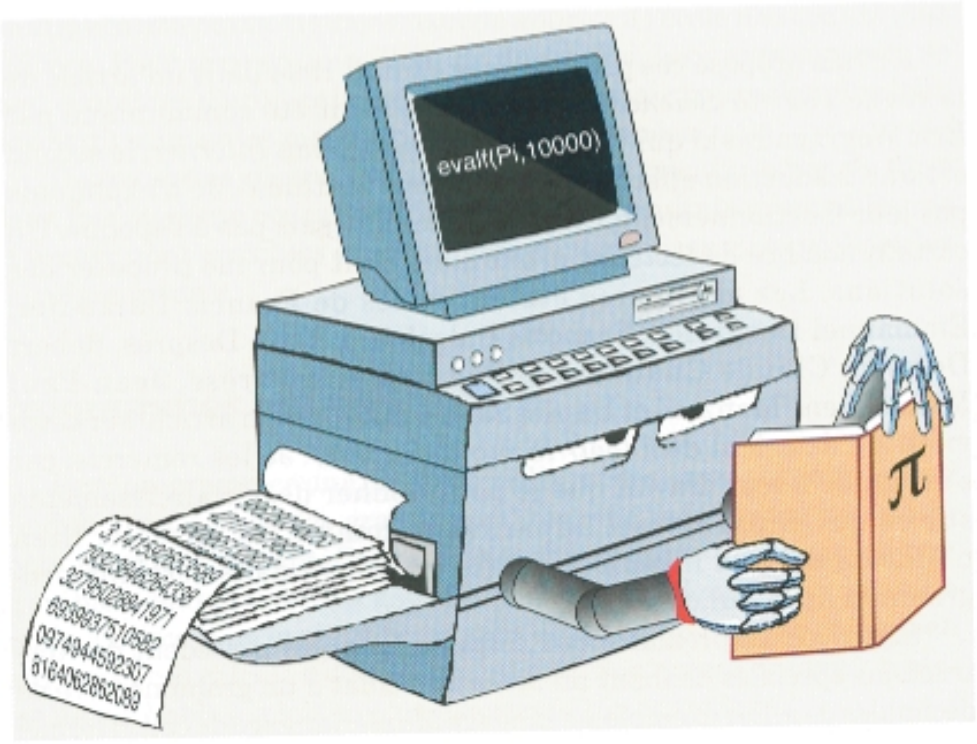
evalf(Pi,10000) ;

provoque instantanément l'affichage de 10 000 décimales de π. Chose étrange, le programme :

evalf(Pi,10001) ;

donne 10 001 décimales de π, mais au bout d'un temps incomparablement plus long. Cela s'explique parfaitement. Tout d'abord, *Maple* contient dans ses propres bibliothèques des sous-programmes de calcul de π, qu'il est donc inutile de reprogrammer (ces sous-programmes utilisent des méthodes expliquées au prochain chapitre). Ensuite, sachant que de nombreux utilisateurs vont demander à avoir 100, 1 000 ou même 10 000 décimales de π, les concepteurs de *Maple* ont écrit explicitement dans un sous-programme les 10 000 premières décimales de π, que le logiciel se contente de recopier quand on le lui demande. En jargon informatique, on dit que les 10 000 décimales de π ont été mises «en dur» dans *Maple*. Cela permet bien évidemment de répondre instantanément aux amateurs pas trop exigeants, et cela explique la discontinuité entre 10 000 et 10 001 (il se peut que, pour me faire mentir, les programmeurs écrivent *en dur*, dans la prochaine version de *Maple*, 10 001 décimales de π, déplaçant ainsi la discontinuité).

Bien entendu, ce qui nous intéresse ici, c'est le calcul de π *ex nihilo* et «sans tricher», ce que font effectivement les deux programmes de la page 95. Mais comment ?



Le programme le plus rapide pour afficher 10 000 décimales de π est le programme qui lit une base de données dans laquelle se trouvent les décimales de π. C'est ce qu'ont bien compris les ingénieurs de chez Maple.

Une série d'Euler judicieusement utilisée

Ces programmes utilisent une série due à Euler, que nous avons déjà rencontrée au chapitre 4 :

$$\pi = 2\left(1 + \frac{1}{3} + \frac{1\times 2}{3\times 5} + \frac{1\times 2\times 3}{3\times 5\times 7} + \frac{1\times 2\times 3\times 4}{3\times 5\times 7\times 9} + \cdots\right) = 2\sum_{n=0}^{\infty} \frac{1\times 2\times \ldots n}{1\times 3\times \ldots (2n+1)}$$

Puisque les termes de la série sont strictement positifs, la suite converge vers π par valeurs inférieures. Le rapport de deux termes consécutifs est inférieur à 1/2, si bien que l'erreur commise en s'arrêtant au terme n est inférieure au dernier terme utilisé (en effet, si le rapport de deux termes consécutifs était égal à 1/2, l'erreur correspondrait à la somme infinie $a_n/2 + a_n/2^2 + a_n/2^3 + \ldots$, qui est égale à a_n). Par conséquent, pour connaître π avec une précision de n décimales, il suffit de sommer $\log_2(10^n) \approx 3{,}32\,n$ termes.

La série d'Euler ne converge pas très rapidement ; elle converge même plus lentement que la méthode d'Archimède, où les calculs sont toutefois plus complexes (puisqu'il faut extraire des racines carrées). Cette série convient bien pour un programme court donnant quelques centaines de décimales, mais si l'on en voulait davantage, elle serait inadéquate.

Comme les calculs sont effectués en base 10 000, à la fin du calcul, les chiffres sont affichés par groupe de 4. Cela explique le *printf("%.4d"* de la version en langage C. En langage *Basic*, une difficulté est créée lors de l'impression à cause de la suppression des zéros précédant un entier (par exemple, 21 n'est pas affiché 0021 comme on le souhaiterait). C'est ce qui justifie le *PRINT LEFT$("0000", 5 - L) ; RIGHT$(x$, L - 1)*.

Le programme calcule 600 «chiffres» en base 10 000, ce qui équivaut à 2 400 chiffres décimaux. En utilisant la relation précédente sur la convergence de la série, on vérifie que le nombre de termes utilisés est $600\times 4\times 3{,}32 = 8\,400$ (arrondi par excès), nombre qui apparaît dans la première ligne du programme. La valeur $14 = 4\times 3{,}32$ apparaît pour la même raison dans la partie du programme commandant le décalage entre chiffres successifs. Les chiffres sont stockés dans un tableau noté *f( )*. Le programme comporte une boucle d'initialisation, suivie d'une double boucle qui effectue le calcul et l'impression.

Le calcul effectué dans la double boucle correspond à l'exploitation de la série terme par terme lorsqu'on l'écrit sous une forme particulière (appelée forme de Horner et fréquemment utilisée pour limiter le nombre de multiplications lors de l'évaluation des polynômes) :

$$1\,000\,\pi \approx \frac{10\,000}{5}\left(1 + \frac{1}{3}\left(1 + \frac{2}{5}\left(1 + \frac{3}{7}\left(\cdots + \frac{8\,399}{16\,799}\right)\right)\right)\right)$$

Le programme obtient progressivement les bons chiffres du début de π, et il les affiche aussitôt. En même temps, il utilise la partie du tableau non affichée pour stocker une sorte de reste dont il a besoin pour obtenir les décimales suivantes : comme nous l'avons vu au début du chapitre, il faut calculer dès le début avec un nombre de chiffres de l'ordre de celui voulu à la fin.

Les algorithmes compte-gouttes : π au tableau

Pour comprendre plus finement encore le calcul effectué par ces programmes, examinons-en maintenant une version «manuelle», qui constitue également (indépendamment de tout programme informatique) une méthode explicite de calcul de π. On peut l'apprendre au même titre que les méthodes de multiplication ou de division à la main, où l'on doit disposer des rangées de chiffres, ou que celle du calcul de racines carrées rappelée page 105.

Les idées exploitées dans cette méthode de calcul de π sont dues à A. Sale, qui les avait d'abord appliquées au calcul du nombre *e* en 1968. L'adaptation au calcul de π a été proposée en 1988 par Daniel Saada et, indépendamment, par Stanley Rabinowitz en 1991. Il est probable que l'auteur du programme C de 158 caractères a eu connaissance de l'article de 1991 de Rabinowitz. Certaines subtilités *(voir plus loin)* interviennent dans l'évaluation de l'erreur pour assurer que les chiffres fournis sont corrects. Pour un calcul plus poussé de π, elles devraient être prises en compte *a priori*, et non pas *a posteriori*, en constatant, après comparaison avec un autre calcul, que l'algorithme ne se trompe pas ! Cette analyse détaillée de l'algorithme a été faite en 1995 par Stanley Rabinowitz et Stan Wagon *(voir page 100)*.

En réorganisant soigneusement les calculs effectués par le programme, on élabore une méthode de calcul de π à la main qui porte le nom «d'algorithme compte-gouttes». Si l'on est bon calculateur, on obtient en quelques heures, par cette méthode, autant de décimales de π que Ludolff von Ceulen, qui s'acharna pendant des années sur ce calcul. Les algorithmes compte-gouttes pour *e* et π ont été popularisés en septembre 1995 par Ian Stewart dans la revue *Pour la Science*.

Pour calculer π avec cette technique, on est amené à faire une série de petits calculs, comme lorsqu'on pose une division. Les décimales arrivent une à une, goutte à goutte. Le résultat est obtenu plus lentement que lors d'une division, mais le procédé reste tout à fait fascinant. Il est amusant que cette méthode n'ait été découverte que si récemment.

Règles de remplissage du tableau

• Initialisation : *(a)* on remplit la ligne *A* avec la suite des nombres entiers et la ligne *B* avec la suite des nombres impairs ; *(b)* on remplit la ligne «début» avec des «2», puis on place des «0» sur toutes les lignes «retenue» de la dernière colonne («**0**» en gras) ; *(c)* On remplit la première ligne «×10» en multipliant la ligne précédente par 10.

• En partant du bord droit du tableau et en allant vers la gauche, on remplit progressivement les trois lignes «retenue», «somme» et «reste» du premier sous-tableau : *(a)* on fait la somme des nombres des lignes «×10» et «retenue» ; *(b)* on place le résultat dans la ligne «somme» ; *(c)* on évalue alors le quotient de cette somme par le nombre de la ligne *B* situé dans la même colonne ; cette opération donne d'une part un reste que l'on place sur la ligne «reste», et d'autre part un quotient que l'on multiplie par le nombre de la ligne *A* (toujours dans la même colonne), ce qui donne un nombre *n* que l'on place dans la ligne «retenue» de la colonne précédente ; *(d)* on continue de la même façon en progressant vers la gauche. Au début, 20 + 0 donne 20 qui, divisé par 25, donne un reste de 20 et un quotient de 0. Le quotient est ainsi «0», trois fois de suite. Puis, lorsqu'on arrive à la colonne 9, on trouve

A	π	*r*	1	2	3	4	5	6	7	8	9	10	11	12
B			3	5	7	9	11	13	15	17	19	21	23	25
début		2	2	2	2	2	2	2	2	2	2	2	2	2
× 10		20	20	20	20	20	20	20	20	20	20	20	20	20
retenue		10	12	12	12	10	12	7	8	9	0	0	0	**0**
somme	3	30	32	32	32	30	32	27	28	29	20	20	20	20
reste		0	2	2	4	3	10	1	13	12	1	20	20	20
× 10		0	20	20	40	30	100	10	130	120	10	200	200	200
retenue		13	20	33	40	65	48	98	88	72	150	132	96	**0**
somme	1	13	40	53	80	95	148	108	218	192	160	332	296	200
reste		3	1	3	3	5	5	4	8	5	8	17	20	0
× 10		30	10	30	30	50	50	40	80	50	80	170	200	0
retenue		11	24	30	40	40	42	63	64	90	120	88	0	**0**
somme	4	41	34	60	70	90	92	103	144	140	200	258	200	0
reste		1	1	0	0	0	4	12	9	4	10	6	16	0
× 10		10	10	0	0	0	40	120	90	40	100	60	160	0
retenue		4	2	9	24	55	84	63	48	72	60	66	0	**0**
somme	1	14	12	9	24	55	124	183	138	112	160	126	160	0
reste		4	0	4	3	1	3	1	3	10	8	0	22	0

Pour obtenir *N* chiffres de π par l'algorithme compte-gouttes, il faut partir d'un tableau de largeur 3,4 × *N* et calculer *N* sous-tableaux. Les nombres en rouge illustrent une étape du remplissage du premier sous-tableau : dans la colonne 9, on porte sur la ligne somme 20 + 0 = 20. Ce nombre est alors divisé par le nombre *B* de la colonne, 19. On écrit le reste de la division, 1, sous la ligne somme. Quant au quotient, aussi égal à 1, il est multiplié par le numéro de la colonne, ce qui donne 9 ; ce produit est reporté sur la ligne retenue de la colonne 8, et on recommence les mêmes opérations dans cette nouvelle colonne.

que 20 divisé par 19 donne un reste de 1, et que le quotient 1 multiplié par 9 donne 9. Ensuite, 9 + 20 donne 29 qui, divisé par 17, donne un reste de 12 et un quotient de 1 ; celui-ci, multiplié par 8, donne 8, d'où la somme suivante, 28, etc.

- Une fois le premier sous-tableau rempli, en enlevant un chiffre au nombre de la colonne *r*, on trouve le premier chiffre de π : 3.
- Pour le deuxième chiffre de π, on remplit d'abord la colonne «×10» du deuxième sous-tableau, en multipliant la ligne «reste» du premier sous-tableau par 10. En repartant de la droite, on remplit comme précédemment toutes les cases du deuxième sous-tableau. Une fois ce dernier rempli, on trouve le deuxième chiffre de π en enlevant un chiffre au nombre de la colonne *r*.
- Pour obtenir *n* chiffres de π, il faut initialiser un tableau à $3{,}32 \times n$ colonnes (arrondi à la valeur supérieure) avec des «2». Dans notre exemple, nous devons nous arrêter dès que nous avons trois décimales de π. Un chiffre obtenu n'est garanti exact que s'il est suivi d'un chiffre différent de 9. En outre, on trouve parfois le «chiffre» 10 (lorsqu'un nombre plus grand que 100 se trouve dans la colonne *r*). Il faut alors modifier le précédent chiffre de π en l'augmentant d'une unité. Heureusement, cela se produit rarement : en partant d'un tableau de 116 «2» (bon pour 35 décimales), on trouve le chiffre «4» au 32e sous-tableau, puis le chiffre «10» au 33e sous-tableau, ce qui amène à corriger le «4» précédent en «5».

Le calcul en bases à pas variable

Une façon d'interpréter les calculs réalisés par l'algorithme compte-gouttes est de considérer la notion de *base de numération à pas variable*.

Le nombre π en base 10, c'est-à-dire dans la base à pas constant $\boldsymbol{d}$ = [1/10, 1/10, 1/10, ...], s'écrit π_d = [3; 1, 4, 1, 5,...], car :

$$\pi = \left(3 + \frac{1}{10}\left(1 + \frac{1}{10}\left(4 + \frac{1}{10}\left(1 + \frac{1}{10}\left(5 + \frac{1}{10}\cdots\right)\right)\right)\right)\right)$$

Or on a vu précédemment que la série d'Euler pouvait s'écrire :

$$\pi = \left(2 + \frac{1}{3}\left(2 + \frac{2}{5}\left(2 + \frac{3}{7}\left(2 + \frac{4}{9}\left(2 + \frac{5}{11}\cdots\right)\right)\right)\right)\right)$$

En généralisant la notation précédente, on dit que π, dans la base à pas variable $\boldsymbol{b}$ = [1/3, 2/5, 3/7, 4/9, ...], s'écrit : π_b = [2; 2, 2, 2, ...].

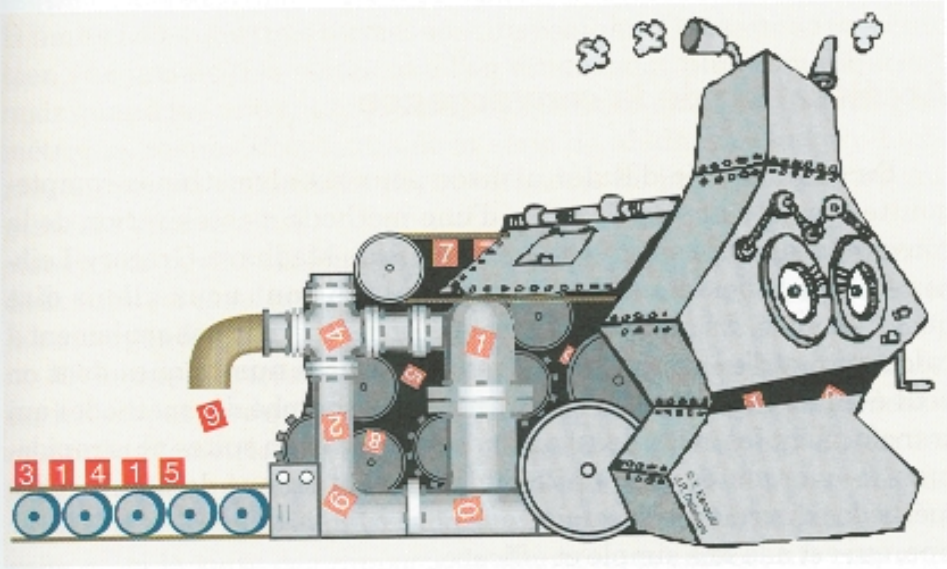
Dans ce contexte, l'algorithme compte-gouttes n'est que l'algorithme de conversion qui fait passer de π écrit en base $\boldsymbol{b}$ à π écrit en base $\boldsymbol{d}$. Voici l'explication de cette conversion :

Soit π' la somme partielle des 13 premiers termes de la série d'Euler de π. On voit que la première ligne du tableau, (2, 2, 2, ..), correspond à π' écrit dans la base $\boldsymbol{b}$.

On obtient la première ligne du premier sous-tableau en multipliant par 10 chaque chiffre de π' en base ***b*** ; elle contient donc une expression de 10π' en base ***b***. Toutefois cette expression n'est pas «normalisée», car certains de ses «chiffres» sont supérieurs aux dénominateurs des fractions de la base à pas variable ***b***. C'est exactement comme si, en base 10, on multipliait par 5 sans précaution chaque chiffre du nombre 453, ce qui donnerait [20 25 15]. Pour arriver à l'écriture «normalisée» 2265 (qui n'utilise que les chiffres de 0 à 9), il faut convertir [20 25 15] en partant du dernier «chiffre» et en faisant passer les retenues de droite à gauche : du 15, on retient le 5 et on passe au «chiffre» suivant avec la retenue 1 qui, ajoutée au 25, donne un 6 et une retenue de 2, etc. Le travail effectué lors du remplissage du premier sous-tableau correspond exactement à cette réécriture sous forme normalisée dans le cadre de la base particulière [1/3, 2/5, 3/7, 4/9, ...].

L'entier 30 se trouve en tête de cette forme normalisée de 10π', ce qui signifie que π' commence par 3. La dernière ligne du premier sous-tableau contient donc le nombre 10π' – 30 = 1,415... en base ***b***. Dans le sous-tableau suivant, on effectue la mise sous forme normalisée de 10(10π' – 30) = 100π' – 300 en base ***b***. Ce nombre commence par 13, ce qui signifie que π' a un «1» pour première décimale après la virgule, et ainsi de suite. Au total, le calcul des sous-tableaux est simplement la conversion en base 10 de π', qu'on avait écrit au départ en base ***b***.

Les algorithmes compte-gouttes (comme les programmes donnés précédemment, qui en sont la traduction) n'utilisent pas de nombres réels et n'ont besoin que d'entiers de taille raisonnable (si l'on n'est pas trop ambitieux). Par exemple, pour obtenir 1 000 décimales de π, des entiers de longueur 10 suffisent. Cette propriété n'a rien d'extraordinaire, car elle est commune à la grande majorité des algorithmes de



Goutte à goutte, la machine de l'algorithme compte-gouttes produit des décimales de π.

calcul de π ; ceux-ci, puisqu'ils reprogramment les algorithmes d'addition et de multiplication, n'utilisent en définitive que des entiers. Tout un mécanisme de report de retenue est présent dans les algorithmes compte-gouttes qui, à leur façon (très élégante), reprogramment aussi une arithmétique entre grands nombres (en fait, ils reprogramment la conversion de l'arithmétique en base b à l'arithmétique en base 10).

En conclusion, les algorithmes compte-gouttes sont une jolie trouvaille qui permet, d'une part, d'écrire des programmes courts pour calculer π et, d'autre part, d'effectuer à la main un calcul agréable de π. Toutefois, ils ne présentent aucun autre progrès par rapport à ce qui existait en 1974, car :

– il faut savoir jusqu'où on souhaite aller avant de commencer ; si l'on décide d'aller loin dans la suite des décimales de π, il faudra,dès le départ, faire de longs calculs (on ne peut pas décider en cours de route le nombre de gouttes que l'on veut !) ;

– pour calculer n décimales de π, les algorithmes compte-gouttes ont besoin d'un espace mémoire de taille proportionnelle à n, à la différence des algorithmes déduits de la formule de Bailey-Borwein-Plouffe, que nous décrirons au chapitre 8 ;

– pour calculer n décimales de π, la durée des calculs est proportionnelle à n^2 (ou même à $n^2 \times \ln(n)$). Avant 1974, tous les algorithmes utilisés pour calculer π étaient dans ce cas. Toutefois, en utilisant les méthodes de multiplication rapide, on peut faire une économie de temps considérable, comme nous le verrons au prochain chapitre.

Signalons pour finir la formule suivante, démontrée il y a quelques années par William Gosper, et qui pourrait donner lieu à un algorithme compte-gouttes plus efficace que le précédent :

$$\pi = 3 + \frac{1}{60}\left(8 + \frac{2\times 3}{7\times 8\times 3}\left(13 + \frac{3\times 5}{10\times 11\times 3}\left(18 + \frac{4\times 7}{13\times 14\times 3}\left(\ldots\right)\right)\right)\right)$$

Accélération de la convergence

En fait, la série d'Euler utilisée pour les algorithmes compte-gouttes résulte de l'application d'une méthode d'accélération de la convergence à la série affreusement lente de Madhava-Gregory-Leibniz. C'est un bel exemple d'une technique dont nous allons dire quelques mots. En analyse numérique, on ne cherche pas seulement à calculer π, et l'on rencontre souvent des suites numériques dont on veut évaluer la limite. C'est pourquoi l'on a cherché des méthodes qui transforment les suites lentement convergentes en suites plus rapidement convergentes. Les mathématiciens ont trouvé de nombreuses méthodes d'accélération, mais nous nous contenterons d'en présenter une, qui est à la fois simple et efficace.

Le procédé delta-2 d'Aitken, inventé en 1926 par le mathématicien et calculateur prodige Alexander Aitken, consiste à définir, à partir de la suite x_n que l'on veut accélérer, une nouvelle suite t_n de terme général (pour $n \geq 2$) :

$$t_n = \frac{x_n x_{n-2} - x_{n-1}^2}{x_n - 2x_{n-1} + x_{n-2}}$$

et dont la limite est la même que celle de x_n. Cette formule est utilisable dès que l'on dispose de trois termes consécutifs x_{n-2}, x_{n-1} et x_n de la suite à accélérer.

En appliquant le delta-2 d'Aitken à la série de Madhava-Gregory-Leibniz, dont la convergence est particulièrement lente ($x_{500} = 3{,}14358$, soit seulement deux décimales exactes !), on obtient une nouvelle suite t_n bien plus rapide :

$$\begin{aligned} t_2 &= 3{,}1666 \\ t_{10} &= 3{,}1418396 \\ t_{50} &= 3{,}14159465 \\ t_{500} &= 3{,}14159265559 \end{aligned}$$

Dans ce cas particulier, on montre qu'à un certain rang, l'erreur commise par t_n est dix fois inférieure à celle commise par x_n, puis 100 fois inférieure un peu plus loin, puis 1 000 fois inférieure encore plus loin, etc. Autrement dit, la quantité $(\pi - t_n)/(\pi - x_n)$ tend vers zéro quand n tend vers l'infini. C'est la définition précise de l'expression «la suite t_n accélère la convergence de la suite x_n».

La méthode d'Aitken est fondée sur une idée élémentaire que nous allons expliquer, car elle est caractéristique des méthodes d'accélération. Le procédé delta-2 d'Aitken est ajusté pour traiter parfaitement les suites géométriques, c'est-à-dire de la forme $L + a \times b^n$: si $|b|$ est inférieur à 1, une telle suite converge vers L, car b^n devient de plus en plus petit quand n augmente. Quand la suite à accélérer est réellement une suite géométrique, le procédé donne directement la limite L dès que trois termes sont disponibles. Ce cas ne se présente bien sûr jamais (les suites que l'on étudie sont plus compliquées), mais quand la suite traitée ressemble suffisamment à une suite géométrique, comme c'est le cas de la série de Madhava-Gregory-Leibniz, le procédé en accélère la convergence. Le principe heuristique général s'énonce ainsi :
«essayer de deviner la limite en partant d'une hypothèse de régularité de la suite ; si ça ne marche pas, ça permettra peut-être d'en accélérer la convergence.»

C'est une idée très simple qui fonctionne d'une manière étonnante, car il n'est pas rare qu'une suite numérique ressemble suffisamment à une suite géométrique. Mais que signifie ce «suffisamment» ? L'un des résultats obtenus au sujet du delta-2 précise cette condition : le terme «écart» désignant la différence de deux termes consécutifs de la suite x_n, si la suite des rapports de deux écarts consécutifs converge

vers une limite non nulle b comprise entre -1 et $+1$, alors la suite donnée par le procédé delta-2 d'Aitken accélère la suite x_n (au sens indiqué précédemment). En bref : si x_n est convergente, de limite L, et que $(x_{n+2} - x_{n+1})/(x_{n+1} - x_n)$ converge vers un nombre b tel que $b \neq 0$ et $-1 \leq b < 1$ (la convergence de x_n est alors dite linéaire), alors $(L - t_n)/(L - x_n)$ tend vers zéro. La série de Madhava-Gregory-Leibniz possède effectivement une convergence linéaire, il était donc normal qu'elle soit accélérée par le delta-2 d'Aitken.

Bien d'autres propriétés de cette formule en apparence simple ont été démontrées ; en particulier, on a prouvé que le procédé delta-2 d'Aitken était optimal pour l'accélération des suites à convergence linéaire. Le résultat précis s'énonce ainsi :

- il n'existe pas de formule algébrique plus simple qui accélère la convergence de toutes les suites à convergence linéaire, et ;
- il n'existe pas de méthode transformant x_n en t_n et de nombre $\varepsilon > 0$ tels que $|L - t_n|/|L - x_n|^{1+\varepsilon}$ tende vers 0 pour toute suite à convergence linéaire (le taux d'accélération de 1 donné par le delta-2 ne peut donc pas être amélioré en $1 + \varepsilon$).

Les méthodes d'accélération de la convergence, aussi intéressantes soient-elles, sont toutefois incapables de résoudre tous les problèmes de convergence. Dans les cas très spécifiques, tel le calcul de π, il est souvent plus intéressant de trouver directement une méthode qui converge vite que de choisir une méthode qui converge lentement pour l'accélérer ensuite. Certains résultats négatifs expliquent d'ailleurs pourquoi il ne faut pas tout attendre des transformations accélératrices.

Voici deux de ces résultats. Le premier montre que les suites à convergence lente ne peuvent être accélérées que de manière exceptionnelle. Le second montre qu'il en est de même pour les suites à convergence très rapide.

- On dit qu'une suite de limite L est à convergence logarithmique si le rapport des erreurs de deux termes consécutifs, $(L - x_{n+1})/(L - x_n)$, s'approche de 1 quand n augmente. Pour une telle suite, plus on va loin, moins les progrès sont rapides. On connaît beaucoup de suites à convergence logarithmique qui sont accélérées par telle ou telle méthode, et l'on a même espéré que l'on pourrait trouver une méthode universelle pour les suites à convergence logarithmique : après tout, si elles sont lentes il doit bien être possible de les faire se presser un peu ! Ce rêve s'est envolé, car nous disposons de la preuve qu'aucune transformation de suites n'est efficace pour l'accélération de toutes les suites à convergence logarithmique. Autrement dit, vous pouvez espérer accélérer certaines suites à convergence logarithmique, mais vous ne pourrez jamais les accélérer toutes par une même méthode : les suites à convergence logarithmique sont si nombreuses, et, derrière

leur apparente régularité, elles se comportent de manière si désordonnée et si variée qu'aucune méthode n'arrivera jamais à les maîtriser globalement.

• L'autre résultat concerne les suites dont le rapport des erreurs de deux termes consécutifs tend vers zéro. Ce sont des suites qui convergent de plus en plus rapidement : le nombre moyen de chiffres gagnés à chaque itération va en augmentant. On pourrait considérer qu'il n'est pas très utile d'accélérer de telles suites... et c'est d'ailleurs impossible ! Comme dans le cas des suites à convergence logarithmique, il est aujourd'hui prouvé qu'aucune méthode d'accélération ne peut accélérer toutes les suites dont le rapport des erreurs de deux termes consécutifs tend vers zéro.

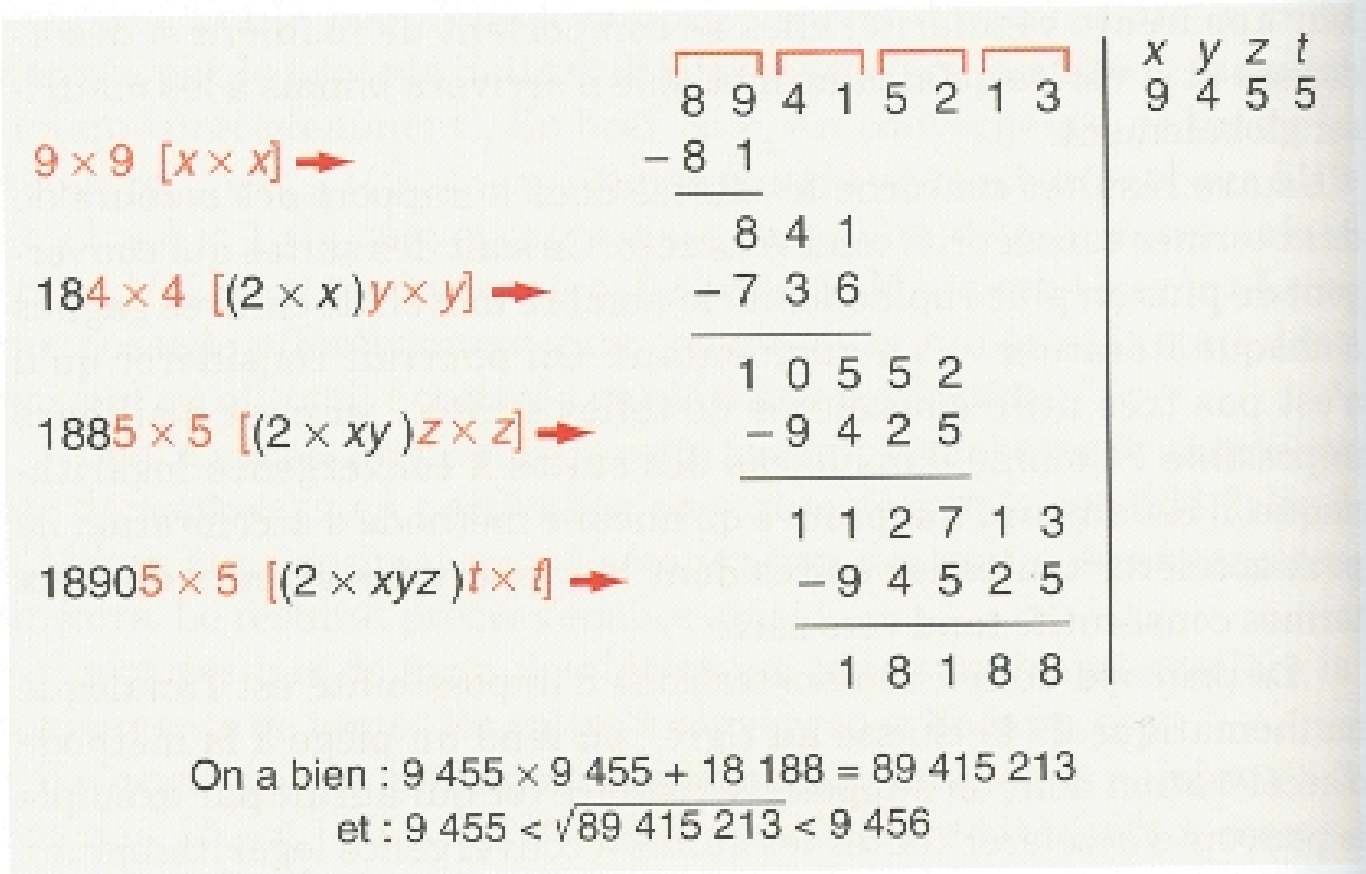
Le principe de ces démonstrations d'impossibilité est l'analogue mathématique de la chasse au tigre : on tend un piège à la méthode d'accélération dont on suppose l'existence (et qui aurait par exemple le pouvoir d'accélérer toutes les suites à convergence logarithmique). Pour cela, on lui donne certaines suites particulières et on note son comportement. Au bout d'un certain temps, on en sait assez sur «ses habitudes» et on lui concocte alors une suite qu'elle est dans l'impossibilité d'accélérer. Comme autre résultat négatif, on a établi que la réunion de deux familles accélérables n'est pas forcément accélérable. Cela prouve que les procédés de composition d'algorithmes d'accélération ne réussissent pas toujours à synthétiser les bonnes propriétés des méthodes d'accélération qu'ils composent.

Sur toutes ces questions, on consultera les livres de Claude Brezinski indiqués dans la bibliographie ou mon ouvrage sur les transformations de suites qui est plus particulièrement consacré aux résultats de limitation.

Annexe : l'extraction de racines carrées

Il existe une méthode d'extraction des racines carrées, posée sous la forme d'une sorte de division, qui permet de calculer des racines carrées avec une précision aussi grande qu'on le souhaite, aussi bien à la main qu'avec une machine. Au prochain chapitre, nous verrons une autre méthode de calcul des racines carrées, mais nous rappelons ici, pour le plaisir, cette ancienne et belle méthode.

Elle ramène la recherche de la racine carrée d'un entier de longueur n à $n/2$ multiplications environ d'un entier long par un entier court. Cela ne convient pas pour de très grands calculs (sauf si on regroupe les diverses multiplications intermédiaires, mais il est alors plus simple d'utiliser la méthode de Newton, expliquée au prochain chapitre, avec une multiplication rapide). En revanche, cette méthode



On a bien : 9 455 × 9 455 + 18 188 = 89 415 213

et : $9\,455 < \sqrt{89\,415\,213} < 9\,456$

peut être utile pour des calculs de taille moyenne (jusqu'à quelques millions de décimales), programmés à partir des formules d'Eugène Salamin et Richard Brent présentées au chapitre suivant.

Après avoir séparé le nombre dont on cherche la racine carrée en tranches de deux chiffres en partant de la droite, on considère la première tranche de gauche, et on cherche le plus grand entier x dont le carré soit inférieur à cette tranche. Dans l'exemple ci dessus, $x = 9$, car $9 \times 9 = 81$ est le plus grand carré inférieur à 89. Après soustraction, il reste 8. On descend deux chiffres de plus, ce qui donne 841. On cherche à présent le plus grand entier y tel que $[2x]y \times y$ (on accole le chiffre y au produit $2x$, puis on multiplie le résultat par y) soit inférieur à 841. Ici $y = 5$ ne convient pas, car $185 \times 5 = 925$ est trop grand. En revanche, $y = 4$ convient, car $184 \times 4 = 736$. On continue de même jusqu'à ce que toutes les tranches de deux chiffres soient abaissées.