CHAPITRE IV

Numération positionnelle et conversion de base

Objectifs

- Réviser la numération en base $b \geq 2$ et l'algorithme de changement de base.
- Implémenter une application surprenante : le calcul de e et de π à une précision arbitraire.
- Préparer le terrain pour le calcul arrondi (chap. XV) et le calcul arrondi fiable (chap. XVI)

Ce chapitre explique d'abord comment convertir la représentation d'un nombre de la base 10 à la base 2, puis comment convertir entre deux bases quelconques. L'idée est simple, mais les applications sont étonnantes : avec très peu d'analyse, cette méthode permet de calculer quelques milliers de décimales de $e = 2,71828\ldots$ Le projet traitera ensuite le calcul de $\pi = 3,14159\ldots$

Sommaire

1. Numération positionnelle

1.1. Un premier exemple : la conversion de la base 10 à la base 2. Rappelons d'abord la division euclidienne des entiers : pour tout $a, b \in \mathbb{Z}$ avec $b \neq 0$ il existe une unique paire $(q, r) \in \mathbb{Z} \times \mathbb{Z}$ telle que $a = bq + r$ et $0 \leq r < |b|$. Dans la suite on utilisera la notation $a \operatorname{div} b := q$ pour le quotient, et $a \bmod b := r$ pour le reste d'une telle division euclidienne.

Exemple 1.1. Comment trouver la représentation binaire du nombre $11,6875_{\text{dec}}$? C'est facile pour la partie entière $11_{\text{dec}} = \mathbf{1} \cdot 2^3 + \mathbf{0} \cdot 2^2 + \mathbf{1} \cdot 2^1 + \mathbf{1} \cdot 2^0 = \texttt{1011}_{\text{bin}}$. Ce développement s'obtient par une division euclidienne itérée selon le schéma suivant :

$$\begin{aligned} 11 \bmod 2 &= \mathbf{1} \quad \text{et} \quad 11 \operatorname{div} 2 = 5 \\ 5 \bmod 2 &= \mathbf{1} \quad \text{et} \quad 5 \operatorname{div} 2 = 2 \\ 2 \bmod 2 &= \mathbf{0} \quad \text{et} \quad 2 \operatorname{div} 2 = 1 \\ 1 \bmod 2 &= \mathbf{1} \quad \text{et} \quad 1 \operatorname{div} 2 = 0 \end{aligned}$$

Pareil pour la partie fractionnaire $0,6875_{\text{dec}} = \mathbf{1} \cdot 2^{-1} + \mathbf{0} \cdot 2^{-2} + \mathbf{1} \cdot 2^{-3} + \mathbf{1} \cdot 2^{-4} = \texttt{0.1011}_{\text{bin}}$. Ici on multiplie par 2 au lieu de diviser, on extrait la partie entière et continue avec la partie fractionnaire :

$$\begin{aligned} 0,6875 \cdot 2 &= \mathbf{1},3750 \\ 0,3750 \cdot 2 &= \mathbf{0},7500 \\ 0,7500 \cdot 2 &= \mathbf{1},5000 \\ 0,5000 \cdot 2 &= \mathbf{1},0000 \end{aligned}$$

Exercice/M 1.2. Vérifier la conversion de $31,9_{\text{dec}}$ en binaire donnée en chapitre I, page 12. Cette fois-ci on obtient une représentation binaire qui est périodique. (Pourquoi ?)

1.2. Représentation en base b. Les bases 10 et 2 n'ont rien de particulier, à part leur usage fréquent, et nous regarderons dans la suite une base b quelconque, avec $b \in \mathbb{N}$, $b \geq 2$. Explicitons d'abord les deux algorithmes sous-jacents aux exemples précédents.

Algorithme IV.1 Représentation d'un nombre naturel $a \geq 0$ en base b

Entrée: Un nombre naturel $x \in \mathbb{N}$ et un entier $b \geq 2$

Sortie: L'unique suite $(x_{-n}, \dots, x_0)$ dans $[\![0, b[\![$ telle que $x = \sum_{k=-n}^{0} x_k b^{-k}$ et $x_{-n} \neq 0$

Initialiser $n \leftarrow -1$
tant que $x > 0$ **faire** $n \leftarrow n+1$, $x_{-n} \leftarrow x \bmod b$, $x \leftarrow x \operatorname{div} b$
retourner $(x_{-n}, \dots, x_0)$

Exercice/M 1.3. Vérifier que l'algorithme IV.1 est correct. Plus explicitement : étant donnés $x \in \mathbb{N}$ et $b \geq 2$, pourquoi produit-il une suite $(x_{-n}, \dots, x_0)$ dans $[\![0, b[\![$ vérifiant $x = \sum_{k=-n}^{0} x_k b^{-k}$? Pourquoi cette suite est-elle unique ? Ceci justifie d'appeler x_k le $k$ième chiffre de x dans le développement en base b.

Algorithme IV.2 Représentation d'un nombre réel $r \geq 0$ en base b

Entrée: Un nombre réel $x \geq 0$ et deux entiers $b \geq 2$, $p \geq 0$

Sortie: L'unique suite $(x_{-n}, \dots, x_0, \dots, x_p)$ dans $[\![0, b[\![$ avec $x_{-n} \neq 0$ vérifiant $x = \sum_{k=-n}^{p} x_k b^{-k} + \varepsilon_p$ avec un reste $0 \leq \varepsilon_p < b^{-p}$

Trouver d'abord la représentation $(x_{-n}, \dots, x_0)$ de la partie entière $\lfloor x \rfloor$.
pour k **de** 1 **à** p **faire** $x \leftarrow x - \lfloor x \rfloor$, $x \leftarrow bx$, $x_k \leftarrow \lfloor x \rfloor$
retourner $(x_{-n}, \dots, x_0, \dots, x_p)$

Exercice/M 1.4. Vérifier que l'algorithme IV.2 est correct : étant donnés $x \geq 0$ et $b \geq 2$ et $p \geq 0$, pourquoi produit-il une suite $(x_{-n}, \dots, x_0, \dots, x_p)$ dans $[\![0, b[\![$? Pourquoi a-t-on $x = \sum_{k=-n}^{p} x_k b^{-k} + \varepsilon_p$ avec un reste $0 \leq \varepsilon_p < b^{-p}$? Pourquoi cette suite est-elle unique ?

Exercice/M 1.5. Montrer que l'algorithme IV.2 est *stable* dans le sens suivant : augmenter la précision de p à $p+1$ *ajoute* un chiffre sans changer les chiffres précédents. On peut ainsi, pour $p \to \infty$, obtenir un *développement infini* en base b. Définir ce que c'est et expliquer pourquoi il n'y a plus unicité. Quels nombres admettent plus d'un développement infini ? L'algorithme ci-dessus n'en produit qu'un, lequel ?

Remarque 1.6. Dans l'algorithme IV.2 nous ignorons comment est donné le nombre réel x, c'est-à-dire comment il est représenté concrètement sur machine. La seule chose qu'il faut savoir est d'effectuer deux opérations élémentaires : multiplication par b, puis séparation des parties entière et fractionnaire. Bien qu'élémentaires, ces opérations ne sont pas toujours faciles : pour vous en convaincre, essayez d'appliquer l'algorithme IV.2 à $\sqrt{7}$ ou à $\ln 2$ ou à $\int_0^1 e^{-x^2/2} dx$ ou tout autre nombre réel qui vous vient à l'esprit.

Par conséquent, dans toutes les applications suivantes, nous supposons que x est lui-même donné dans une numération positionnelle, c'est-à-dire par une somme $x = \sum x_k v_k$ où les x_k sont les « chiffres », pondérés par des valeurs positionnelles v_k, choisies convenablement selon le contexte. Ce cadre est suffisamment général pour être intéressant, et en même temps il se prête bien au calcul.

1.3. Conversion de base. Ce paragraphe explique comment convertir un développement en base b en une autre base c. La partie entière se traite comme ci-dessus et ne pose aucun problème. Nous nous concentrerons donc sur la partie fractionnaire uniquement. Autrement dit, nous allons implémenter des calculs avec des *nombres à virgule fixe*, c'est-à-dire avec des développements de la forme

$$x_0, x_1 x_2 x_3 \dots x_m \quad \text{en base } b.$$

Une telle représentation n'est rien d'autre que la donnée d'une suite de chiffres $x_0, x_1, \dots, x_m$. Afin d'avoir une notation commode nous introduisons l'application $\langle \cdot \rangle_b : \mathbb{Z}^{m+1} \to \mathbb{Q}$ qui associe à chaque suite finie $x = (x_0, x_1, \dots, x_m)$ le nombre rationnel $\langle x \rangle_b := \sum_{k=0}^{m} x_k b^{-k}$ ainsi représenté. C'est une application $\mathbb{Z}$-linéaire, dont l'image consiste de tous les nombres rationnels de la forme z/b^{-m} avec $z \in \mathbb{Z}$.

Définition 1.7. On dit que $q \in \mathbb{Q}$ est *représenté* par $x \in \mathbb{Z}^{m+1}$ en base b si $\langle x \rangle_b = q$. Une telle représentation est appelée *normale* (par rapport à b) si $0 \leq x_k < b$ pour tout $k = 1, 2, \ldots, m$.

Pour simplifier nous n'exigeons pas que $0 \leq x_0 < b$; l'entier x_0 joue le rôle de la partie entière et peut prendre n'importe quelle valeur dans $\mathbb{Z}$. Ce sont les chiffres après la virgule qui nous intéressent ici.

☞ *Dans tout ce chapitre nous allons utiliser ces développements dits* à virgule fixe. ✍

Notons d'abord que la normalisation est toujours possible : si par exemple x_m ne vérifie pas $0 \leq x_m < b$, on effectue une division euclidienne $x_m = qb + x'_m$ avec quotient $q = x_m \operatorname{div} b$ et un reste $x'_m = x_m \bmod b$ vérifiant $0 \leq x'_m < b$. On remplace alors x_m par x'_m et ajoute la retenue q à x_{m-1}. En itérant ce procédé pour $k = m-1, \ldots, 1$ on obtient le résultat suivant :

Proposition 1.8 (normalisation)**.** *Toute représentation $x \in \mathbb{Z}^{m+1}$ peut être normalisée par rapport à la base b, c'est-à-dire qu'il existe une représentation normale $\bar{x} \in \mathbb{Z}^{m+1}$ telle que $\langle \bar{x} \rangle_b = \langle x \rangle_b$.*

L'algorithme IV.3 ci-dessous construit une telle représentation normale à partir de x.

Algorithme IV.3 Normalisation par rapport à une base b

Entrée: Un vecteur $x = (x_0, x_1, \ldots, x_m) \in \mathbb{Z}^{m+1}$ représentant $\bar{x} = \langle x \rangle_b$ en base $b \geq 2$

Sortie: Le vecteur normal $x = (x_0, x_1, \ldots, x_m) \in \mathbb{Z}^{m+1}$ représentant $\bar{x}$ en base $b \geq 2$

pour k **de** m **à** 1 **faire** // l'indice k parcourt de droite à gauche
  $x_{k-1} \leftarrow x_{k-1} + (x_k \operatorname{div} b)$ // ajouter la retenue à x_{k-1}
  $x_k \leftarrow x_k \bmod b$ // réduire x_k modulo b
fin pour
retourner $(x_0, x_1, \ldots, x_m)$

Nous allons prouver la proposition 1.8 en montrant que l'algorithme est correct :

Lemme 1.9 (correction)**.** *L'algorithme IV.3 est correct. Plus explicitement, la valeur représentée $\langle x \rangle_b$ ne change pas durant l'exécution de l'algorithme, et le vecteur renvoyé x est normalisé.*

DÉMONSTRATION. Évidemment l'algorithme se termine (toujours après m itérations). Vérifions d'abord l'invariance. Par définition on a $\langle x \rangle_b = \sum_{k=0}^{m} x_k b^{-k}$. La division euclidienne donne $x_k = b \cdot q + x'_k$ tel que $0 \leq x'_k < b$ et $q \geq 0$. Si l'on pose $x'_{k-1} = x_{k-1} + q$ on voit aisément que $\langle x \rangle_b = \langle x' \rangle_b$. (Le détailler.)

Montrons par récurrence que le résultat est normalisé. Supposons qu'avant l'itération k les chiffres $x_{k+1}, \ldots, x_m$ sont normalisé, c'est-à-dire que $0 \leq x_j < b$ pour tout $j = k+1, \ldots, m$. Ces chiffres-là ne changent pas lors de l'itération k, et après le chiffre x_k vérifie $0 \leq x_k < b$ par construction. □

C'est toujours une excellente idée de majorer les calculs intermédiaires : explosent-ils ou restent-ils bornés ? Plus concrètement, on peut se demander si les petits entiers (de type `int`) suffiront pour implémenter l'algorithme. Le lemme suivant en donne la réponse :

Lemme 1.10 (majoration)**.** *Dans l'algorithme IV.3, si tous les x_k satisfont initialement $0 \leq x_k < cb$, avec une constante $c \in \mathbb{N}$, alors la retenue ne dépasse jamais $2c$.*

DÉMONSTRATION. Pour $k > m$ la retenue est nulle, il n'y donc rien à montrer. Supposons que la retenue ajoutée à x_k a été inférieure à $2c$. Alors $0 \leq x_k < c(b+2)$. La division euclidienne $x_k = bq + x'_k$ donne donc $0 \leq x'_k < b$ et $0 \leq q < 2c$ car $b \geq 2$. On conclut par récurrence □

Le lemme suivant confirme que tronquer une représentation en base b donne la partie entière.

Lemme 1.11 (unicité)**.** *Si $x \in \mathbb{Z}^{m+1}$ est normale par rapport à la base b, alors $x_0 \leq \langle x \rangle_b < x_0 + 1$. Si $x, y \in \mathbb{Z}^{m+1}$ sont normales par rapport à la base b, alors $\langle x \rangle_b = \langle y \rangle_b$ entraîne $x = y$.*

DÉMONSTRATION. Évidemment $\langle x \rangle_b = \sum_{k=0}^{m} x_k b^{-k} \geq x_0$ car tous les termes de la somme $\sum_{k=1}^{m} x_k b^{-k}$ sont positifs ou nuls. D'autre part $\sum_{k=1}^{m} x_k b^{-k} \leq \sum_{k=1}^{m} (b-1) b^{-k} = 1 - b^{-m} < 1$.

Supposons maintenant que $x, y \in \mathbb{Z}^{m+1}$ sont normales et que $\langle x \rangle_b = \langle y \rangle_b$. Tout d'abord on constate que $\lfloor \langle x \rangle_b \rfloor = x_0$ et $\lfloor \langle y \rangle_b \rfloor = y_0$, ce qui implique $x_0 = y_0$. Ensuite $\lfloor b \langle x \rangle_b \rfloor = b x_0 + x_1$ et $\lfloor b \langle y \rangle_b \rfloor = b y_0 + y_1$, ce qui implique $x_1 = y_1$. On conclut ainsi par récurrence. □

L'argument de la preuve précédente n'est rien d'autre que le développement en base b vu dans l'algorithme IV.2. Après cette préparation, l'algorithme IV.4 formalise la méthode de l'exemple 1.1 :

Algorithme IV.4 Changement de la base b à la base c à une précision donnée n

Entrée: Un vecteur $x = (x_0, x_1, \ldots, x_m) \in \mathbb{Z}^{m+1}$ et trois entiers $b \geq 2$, $c \geq 2$, $n \geq 0$
Sortie: Un vecteur $y = (y_0, y_1, \ldots, y_n) \in \mathbb{Z}^{n+1}$, normal par rapport à la base c
Garanties: La valeur $\langle y \rangle_c$ est une approximation optimale dans le sens que $\langle y \rangle_c \leq \langle x \rangle_b < \langle y \rangle_c + c^{-n}$

pour j **de** 0 **à** n **faire**
  normaliser x par rapport à la base b // ceci fait appel à l'algorithme précédent
  $y_j \leftarrow x_0, \quad x_0 \leftarrow 0$ // transférer la partie entière x_0 dans y_j
  $x \leftarrow c \cdot x$ // multiplier chaque chiffre de x par c
fin pour
retourner $(y_0, y_1, \ldots, y_n)$

Proposition 1.12 (correction)**.** *L'algorithme IV.4 est correct, c'est-à-dire qu'il convertit la représentation $x \in \mathbb{Z}^{m+1}$ en base b en une représentation normale $y \in \mathbb{Z}^{n+1}$ en base c telle que $\langle y \rangle_c \leq \langle x \rangle_b < \langle y \rangle_c + c^{-n}$.*

DÉMONSTRATION. Évidemment l'algorithme se termine (toujours après m itérations).

Vérifions d'abord que les chiffres $y_1, \ldots, y_n$ ainsi produits sont normalisés dans le sens que $0 \leq y_j < c$. Ceci n'est pas forcément vrai pour y_0 qui reçoit la partie entière initiale. Mais pour $j = 1, \ldots, n$ nous savons que $0 \leq \langle x \rangle_b < c$. La normalisation de x assure que $x_0 \leq \langle x \rangle_b < x_0 + 1$ d'après le lemme 1.11.

Notons $x^{(j)}$ et $y^{(j)}$ les représentations au début de la $j$ème itération. On vérifie aisément par récurrence que l'algorithme préserve l'égalité $\langle x \rangle_b = \langle y^{(j)} \rangle_c + c^{-j} \langle x^{(j)} \rangle_b$. (C'est le point clé. Le détailler.) On en déduit l'encadrement $\langle y^{(j)} \rangle_c \leq \langle x \rangle_b < \langle y^{(j)} \rangle_c + c^{-j}$, ce qui prouve la correction de l'algorithme. □

Remarque 1.13 (approximation)**.** En général on ne peut pas espérer tomber exactement sur $\langle y \rangle_c = \langle x \rangle_b$, même avec une précision n arbitrairement grande. Pour le voir convertir par exemple $\frac{1}{3} = \langle 0,1 \rangle_3$ vers $\langle 0,3,3,3,\ldots \rangle_{10}$ en base 10, ou bien $\langle 0,1 \rangle_{10}$ en base 2. Mais en choisissant la précision n suffisamment grande, l'encadrement $\langle y \rangle_c \leq \langle x \rangle_b < \langle y \rangle_c + c^{-n}$ garantit une approximation aussi fine que souhaitée.

Remarque 1.14 (complexité)**.** L'algorithme IV.4 est de complexité *quadratique* : il nécessite mn itérations pour convertir la représentation initiale x de longueur m en la représentation finale y de longueur n.

Exercice/P 1.15. Implémenter les algorithmes IV.3 et IV.4 en deux fonctions

```
void normaliser( vector<int>& chiffres, int base );
void convertir( vector<int> chiffres1, int base1,
                vector<int>& chiffres2, int base2, int precision );
```

Justifier le mode de passage des paramètres. L'utilisation du type `int` risque-t-elle de provoquer des erreurs de dépassement de capacité ? Tester votre fonction avec un programme qui lit au clavier une représentation en base b de longueur m et la convertit en base c avec précision n. (Pour l'entrée-sortie des vecteurs vous pouvez vous servir du fichier `vectorio.cc`.)

☞ Les bases fréquemment utilisées sont 2, 8, 10, 16, et on aura rarement besoin d'une base $b > 16$. Il semble donc une bonne idée d'utiliser les chiffres `0...9` puis `A...F` pour représenter les entiers $0, \ldots, 15$. Adapter votre programme pour qu'il utilise cette notation.

2. Représentation factorielle

La représentation en base b est pratique et omniprésente, mais il y a d'autres représentations intéressantes qui sont parfois mieux adaptées. Nous regarderons dans la suite le *développement factoriel*. En imitant l'approche du §1.3, nous introduisons l'application $\langle \cdot \rangle_! : \mathbb{Z}^{m+1} \to \mathbb{Q}$ qui associe à chaque suite finie $x = (x_0, x_1, \ldots, x_m)$ le nombre rationnel $\langle x \rangle_! := \sum_{k=0}^{m} \frac{x_k}{(k+1)!}$. Par exemple $\langle 2,1,1,1 \rangle_! = 2 + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!}$ est le début de la série $\sum_{k=0}^{\infty} \frac{1}{k!}$ qui converge vers $e = 2,71828\ldots$.

Remarque 2.1 (base mixte)**.** La représentation factorielle utilise ce que l'on appelle une *base mixte*. De tels systèmes sont très courants, par exemple pour parler d'une durée de temps : ainsi la durée de 2 semaines, 3 jours, 10 heures, 55 minutes et 17 secondes équivaut à $\mathbf{2} + \frac{\mathbf{3}}{7} + \frac{\mathbf{10}}{7 \cdot 24} + \frac{\mathbf{55}}{7 \cdot 24 \cdot 60} + \frac{\mathbf{17}}{7 \cdot 24 \cdot 60 \cdot 60}$ semaines.

Exercice/M 2.2. L'application $\langle\cdot\rangle_! : \mathbb{Z}^{m+1} \to \mathbb{Q}$ est $\mathbb{Z}$-linéaire et son image consiste de tous les nombres rationnels $\frac{z}{(m+1)!}$ avec $z \in \mathbb{Z}$. En particulier, tout nombre rationnel $q \in \mathbb{Q}$ admet une telle représentation avec m convenable. Expliquer pourquoi le développement en base b mène aux représentations périodiques pour certains nombres rationnels, alors que dans le système factoriel tout tel développement se terminera. Écrire deux algorithmes analogues aux algorithmes IV.1 et IV.2 qui développent un nombre réel $r \geq 0$ en base factorielle.

Définition 2.3. On dit que $x \in \mathbb{Z}^{m+1}$ est *normale* par rapport à la base factorielle si $0 \leq x_k \leq i$ pour tout $k = 1, \ldots, m$. (Comme avant nous ne posons aucune restriction sur la valeur x_0.)

Heureusement, comme en base b, la normalisation en base factorielle est toujours possible :

Lemme 2.4 (normalisation)**.** *Toute représentation $x \in \mathbb{Z}^{m+1}$ peut être normalisée par rapport à la base factorielle, c'est-à-dire qu'il existe une représentation normale $\overline{x} \in \mathbb{Z}^{m+1}$ telle que $\langle\overline{x}\rangle_! = \langle x\rangle_!$. L'algorithme IV.5 ci-dessous calcule une telle représentation normale à partir de x.*

Algorithme IV.5 Normalisation par rapport à la base factorielle

Entrée: Un vecteur $x = (x_0, x_1, \ldots, x_m) \in \mathbb{Z}^{m+1}$ représentant $\overline{x} = \langle x\rangle_!$ en base factorielle
Sortie: Le vecteur normal $x = (x_0, x_1, \ldots, x_m) \in \mathbb{Z}^{m+1}$ représentant $\overline{x}$ en base factorielle

pour k **de** m **à** 1 **faire** // l'indice k parcourt de droite à gauche
  $x_{k-1} \leftarrow x_{k-1} + x_k \operatorname{div} (k+1)$ // ajouter la retenue à x_{k-1}
  $x_k \leftarrow x_k \bmod (k+1)$ // réduire x_k modulo $k+1$
fin pour
retourner $(x_0, x_1, \ldots, x_m)$

Exercice/M 2.5. Prouver l'algorithme IV.5 : pourquoi la valeur $\langle x\rangle_!$ ne change-t-elle pas ?

Le lemme suivant affirme que tronquer une représentation factorielle donne la partie entière, et on en déduit que la représentation normale en base factorielle est unique :

Lemme 2.6 (unicité)**.** *Si $x \in \mathbb{Z}^{m+1}$ est normale par rapport à la base factorielle, alors $x_0 \leq \langle x\rangle_! < x_0 + 1$. Si $x, y \in \mathbb{Z}^{m+1}$ sont normales par rapport à la base factorielle, alors $\langle x\rangle_! = \langle y\rangle_!$ entraîne $x = y$.*

Exercice/M 2.7. Montrer le lemme précédent en prouvant que $\sum_{k=1}^{m} \frac{k}{(k+1)!} = 1 - \frac{1}{(m+1)!}$. Ensuite, pour l'unicité, expliciter comment la valeur $\langle x\rangle_!$ détermine x_0, puis les chiffres $x_1, x_2, x_3, \ldots$.

Reste à établir la conversion entre développement factoriel et développement en base b :

Proposition 2.8 (conversion)**.** *L'algorithme IV.6 convertit la représentation factorielle $x \in \mathbb{Z}^{m+1}$ en une représentation $y \in \mathbb{Z}^{n+1}$ en base b telle que $\langle y\rangle_b \leq \langle x\rangle_! < \langle y\rangle_b + b^{-n}$.*

Algorithme IV.6 Changement de base factorielle en base b avec précision n

Entrée: Un vecteur $x = (x_0, x_1, \ldots, x_m) \in \mathbb{Z}^{m+1}$ et deux entiers $b \geq 2$, $n \geq 0$
Sortie: Un vecteur $y = (y_0, y_1, \ldots, y_n) \in \mathbb{Z}^{n+1}$, normal par rapport à la base b
Garanties: La valeur $\langle y\rangle_b$ est une approximation optimale dans le sens que $\langle y\rangle_b \leq \langle x\rangle_! < \langle y\rangle_b + b^{-n}$

pour j **de** 0 **à** n **faire**
  normaliser x par rapport à la base factorielle // ceci fait appel à l'algorithme précédent
  $y_j \leftarrow x_0$, $x_0 \leftarrow 0$ // transférer la partie entière x_0 dans y_j
  $x \leftarrow b \cdot x$ // multiplier chaque chiffre de x par b
fin pour
retourner $(y_0, y_1, \ldots, y_n)$

Exercice/M 2.9. Prouver l'algorithme IV.6 : expliquer pourquoi les chiffres $y_1, \ldots, y_n$ sont normalisés dans le sens que $0 \leq y_k < b$, puis prouver l'inégalité $\langle y\rangle_b \leq \langle x\rangle_! < \langle y\rangle_b + b^{-n}$. (On pourra suivre la preuve de la proposition 1.12) Pourquoi ne peut-on en général pas espérer tomber sur l'égalité $\langle y\rangle_b = \langle x\rangle_!$ même avec une précision n arbitrairement grande ?

2.1. Calcul de e à 10000 décimales. Comme application on se propose de calculer $e = \sum_{i=0}^{\infty} \frac{1}{i!}$ en base b à une précision p donnée près. Plus explicitement, on cherche une suite de chiffres $(t_0, t_1, \ldots, t_p)$ telle que le nombre rationnel $t = \sum_{k=0}^{p} t_k b^{-k}$ ainsi représenté vérifie $t < e < t + b^{-p}$.

Exercice/M 2.10. Expliquer pourquoi cette spécification définit le vecteur $(t_0, t_1, \ldots, t_p)$ de manière univoque. Comment définir les « chiffres de e » ? Y a-t-il unicité ? Quel est le rapport entre les chiffres t_k et les chiffres de e ?

Quant au calcul, voici une démarche possible, qui n'est rien d'autre qu'un changement de base :

- Tout d'abord, le vecteur $x = (2, 1, 1, \ldots, 1, 1) \in \mathbb{Z}^{m+1}$ représente une valeur approchée $s := \langle x \rangle_!$ vérifiant $s < e < s + \frac{2}{(m+2)!}$. (Le montrer.)
- En convertissant x de base factorielle en base b, comme ci-dessus, on obtient $y \in \mathbb{Z}^{n+1}$ qui représente une valeur approchée $t := \langle y \rangle_b$ vérifiant $t \leq s < t + b^{-n}$.

Ce procédé produit alors un développement $y \in \mathbb{Z}^{n+1}$ en base b qui représente une valeur approchée vérifiant $t < e < t + \varepsilon$ avec une erreur $\varepsilon = \frac{2}{(m+2)!} + b^{-n}$.

Exercice/M 2.11. Pour calculer $p = 10000$ chiffres valables de e nous considérons $n = 10010$, disons. Trouver m tel que $\frac{2}{(m+2)!} \leq b^{-n}$, afin de garantir $\varepsilon < 2b^{-n}$. On fixe ainsi les deux paramètres m et n utilisés dans le procédé ci-dessus. Expliquer pourquoi le développement y ainsi trouvé donne presque n chiffres exacts de e : au pire, le dernier chiffre devrait être augmenté par 1, avec éventuelle propagation des retenues sur les chiffres précédents. Malgré cette ambiguïté concernant les derniers chiffres, pourquoi peut-on espérer d'ainsi obtenir au moins 10000 chiffres valables de e ?

Avant de mettre en œuvre ce développement de e en C++, assurons-nous que le type `int` suffit pour tous les calculs intermédiaires effectués dans l'algorithme IV.6.

Proposition 2.12 (majoration)**.** *Soit $x \in \mathbb{Z}^{m+1}$ normal par rapport à la base factorielle. En effectuant la multiplication par b puis la normalisation, tous les calculs intermédiaires sont majorés par bm.*

DÉMONSTRATION. Commençons par un vecteur normal $x \in \mathbb{Z}^{m+1}$, c'est-à-dire $0 \leq x_k \leq k$. La multiplication par b produit un vecteur $x' = bx$ avec $x'_k = bx_k \leq bk$. Jusqu'ici le plus grand nombre qui puisse apparaître est bm. Montrons que la normalisation suivante ne produit pas de nombres plus grands. On a $bx_m \leq bm < b(m+1)$, donc la retenue $a_m = bx_m \operatorname{div} (m+1)$ est majorée par $b-1$. Pour la position d'avant on obtient ainsi $bx_{m-1} + a_m < bm$, donc la retenue a_{m-1} est également majorée par $b-1$. Par récurrence on conclut que chaque retenue a_k est majorée par $b-1$ et que $bx_{k-1} + a_k < bk$. Ainsi tous les calculs intermédiaires sont majorés par bm. □

Pour le calcul de e, le vecteur initial $x = (2, 1, 1, \ldots, 1) \in \mathbb{Z}^{m+1}$ est normal par rapport à la base factorielle. Sous la condition que bm n'excède pas la capacité de `int`, nous pouvons alors implémenter le calcul de e en utilisant le type `int`.

Exercice/P 2.13. Écrire un programme qui calcule les 10000 premiers chiffres du développement décimal de e. Que valent les chiffres aux positions 9991 à 10000 ?

3. Quelques conseils pour une bonne programmation

Tout travail fait, contemplons la citation « algorithms + data structures = programs » donnée au début du chapitre. Plus concrètement, explicitons quelques recommandations de bon sens pour le développement d'un travail informatique. Dans le cadre de ce cours ces règles ne sont que des conseils, mais elles deviennent primordiales pour tout projet important, en particulier si différentes tâches sont réparties sur une équipe. Dans ce but nous proposons une liste de six étapes essentielles :

1. ***Spécification:*** Cette étape préliminaire consiste à définir ce que l'on *veut* faire. Ainsi on doit détailler le domaine d'application et le résultat exigé, ce qui constitue la *spécification* de la fonction recherchée. Il faut en particulier s'assurer que la spécification correspond bien à l'objectif envisagé, qu'elle est univoque et ne contient pas de contradictions.

   ☞ Pour un projet important cette phase aboutit souvent à la rédaction d'un *cahier des charges* qui fixe l'objectif et détaille la spécification sous forme d'un contrat. Remarquons que la description supplémentaire de ce que la fonction *ne fait pas* peut également servir à clarifier l'objectif.

2. ***Structuration des données et choix des algorithmes:*** Le langage utilisé, le compilateur et les bibliothèques fournissent certaines *structures de données* avec leurs opérations spécifiques. Ceci est le "bas" du programme, c'est-à-dire le niveau le plus élémentaire.

   Pour résoudre le problème spécifié dans l'étape précédente, il faut maintenant décider de la *représentation informatique* des objets en question et décrire les opérations dont on a besoin. Autrement dit, il s'agit de formuler un *modèle informatique* du problème posé et de la solution envisagée. Ceci est le "haut" du programme, c'est-à-dire le niveau le plus complexe.

3. ***Programmation modulaire:*** Le principe consiste à partager un problème donné en plusieurs sous-problèmes que l'on traite successivement, pour arriver finalement aux opérations élémentaires, déjà implémentées. Idéalement c'est une traduction directe de la structuration élaborée dans l'étape précédente. L'implémentation du programme peut ainsi se faire de haut en bas (*top-down*) ou de bas en haut (*bottom-up*).

4. ***Preuve de correction:*** Le programme étant écrit, on le *prouve*, c'est-à-dire qu'on montre qu'il vérifie la spécification. Cette étape développe alors un raisonnement précis et complet, souvent issu de l'analyse faite dans les étapes précédentes. Avouons que la preuve de correction est plus ou moins facile pour les programmes simples, mais elle devient un défi inextricable pour des programmes complexes (ou mal organisés).

   ☞ L'idéal serait la preuve automatique des programmes, ce qui est possible avec certains langages de programmation récents. Évidemment une telle approche demande une programmation beaucoup plus rigoureuse : le programmeur doit fournir le code du programme avec le code de sa preuve. Le compilateur ne peut en général pas *inventer* une preuve de correction, mais il peut très bien *vérifier* une preuve, convenablement formalisée. En tout cas, le C++ en est loin.

5. ***Tests:*** Si tout ce qui précède a été fait correctement, cette étape est superflue. Néanmoins il est en général prudent de s'assurer sur des *exemples* que l'on a atteint le but. À noter que les tests ne peuvent en général porter que sur très peu d'exemples, la difficulté étant de n'oublier aucun cas particulier. En général, la phase de tests s'effectue de bas en haut : on teste d'abord les fonctions élémentaires pour finir avec la fonction principale.

6. ***Présentation:*** Il convient de bien présenter le code source et de le commenter sans retenue. Ce n'est pas seulement une préoccupation esthétique ! Ce souci d'explication du programme assure sa compréhension, puis sa diffusion et son évolution éventuelle. Dans ce but il sera également souhaitable d'établir une documentation, détaillée et complète, indépendante de l'implémentation.

☞ Très souvent un projet de programmation n'est pas un processus linéaire mais *cyclique*. Si la preuve ou les tests (étapes 4 et 5) exhibent des erreurs, on est forcé de réviser l'étape 3 (pour une erreur de programmation) voire recommencer l'étape 2 (pour une erreur de conception plus fondamentale). Si les tests montrent que le programme s'exécute trop lentement ou plus généralement mobilise trop de ressources, on sera mené à reconsidérer l'étape 3 (pour optimiser l'implémentation) voire l'étape 2 (pour choisir des structures et/ou des algorithmes plus efficaces). Si finalement on découvre des ambiguïtés ou même des contradictions dans la spécification, on est forcé de reconsidérer l'étape 1.

Que j'aime à faire apprendre
un nombre utile aux sages !
Glorieux Archimède, artiste, ingénieur,
Toi de qui Syracuse aime encore la gloire,
Soit ton nom conservé par de savants grimoires.

PROJET IV

Calcul de π à 10000 décimales

Objectifs

- Calculer les chiffres de π à une précision donnée.
- Développer une preuve de correction pour cette implémentation.

Ce projet présente une méthode simple, publiée par S. Rabinowitz et S. Wagon en 1995, pour calculer π à une précision donnée près. Cette méthode est suffisamment efficace pour nos besoins, mais surtout elle a le mérite d'être très élémentaire : il ne s'agit que d'un changement de base ! Sur un ordinateur standard on arrive ainsi facilement à calculer les 10000 premiers chiffres de π.

Ce projet se décompose en deux parties : la première discute l'analyse mathématique, la deuxième concerne l'implémentation. Il va sans dire que les deux, une analyse détaillée et une implémentation soigneuse, sont essentielles pour que votre futur programme soit correct. Si vous travaillez soigneusement ces deux étapes, non seulement votre logiciel produira des décimales, mais vous saurez même *prouver* qu'il s'agit des décimales de π.

☞ *Remarque.* — Si vous préférez aller directement aux sources, vous pouvez consulter l'article original de Stanley Rabinowitz et Stan Wagon, *A spigot algorithm for the digits of* π, American Mathematical Monthly 102 (1995), no. 3, pages 195–203. Les auteurs ajoutèrent à la fin une implémentation hâtive en Pascal, dont vous trouverez une traduction en C++ dans le fichier `rabinowitz.cc`. Malheureusement ce programme n'est pas entièrement correct (voir les commentaires dans notre fichier source). Ceci ne diminue en rien le travail mathématique des auteurs, mais souligne l'importance d'une implémentation soigneuse.

Sommaire

1. Quel est le but de ce projet ?

Avant d'entrer dans le vif du sujet, précisons ce que l'on envisage à faire. Tout d'abord on choisit une base $b \geq 2$, disons $b = 10$ pour fixer les idées. Dans le développement en base b on peut écrire $\pi = \sum_{k=0}^{\infty} \pi_k^{(b)} b^{-k}$ avec des chiffres $\pi_k^{(b)} \in [\![0, b[\![$ pour tout $k \geq 1$. Dans la pratique, c'est-à-dire en temps limité, on ne peut calculer qu'un nombre fini de chiffres. On fixe alors un nombre p de chiffres à calculer, ce que nous appelons la précision souhaitée. Le but de ce projet est d'écrire une fonction

```
approx_pi( int base, int precision, vector<int>& chiffres )
```

qui calcule les chiffres $(t_0, t_1, \ldots, t_p)$ d'un nombre rationnel $t = \sum_{k=0}^{p} t_k b^{-k}$ de sorte que l'on puisse garantir $t < \pi < t + b^{-p}$. Afin de garantir sa correction on développera une preuve mathématique.

Exercice/M 1.1 (optionnel). Rappelez pourquoi tout nombre réel admet une représentation en base b, puis expliciter les réels qui en admettent plus d'une. Expliquez pourquoi la représentation est unique pour le nombre π. (Vous pouvez admettre que π est irrationnel.) On peut donc parler du $k$ième chiffre de π dans le développement en base b, ce que nous avons noté $\pi_k^{(b)}$. Expliquer pourquoi la spécification de $(t_0, t_1, \ldots, t_p)$ donnée ci-dessus définit ce vecteur de manière univoque. Quel est le rapport entre les chiffres t_k et les chiffres $\pi_k^{(b)}$?

2. Une représentation convenable de π

Notre point de départ est la présentation de π par la série suivante :

$$\frac{\pi}{2} = \sum_{k=0}^{\infty} \frac{1\cdot 2\cdots k}{3\cdot 5\cdots(2k+1)} = 1 + \frac{1}{3} + \frac{1\cdot 2}{3\cdot 5} + \frac{1\cdot 2\cdot 3}{3\cdot 5\cdot 7} + \dots \tag{1}$$

Votre futur programme convertira tout simplement cette série en un développement en base b.

Exercice/M 2.1 (optionnel). Dans tout ce projet vous pouvez considérer la formule précédente comme définition de π. Pour ceux qui se demandent s'il s'agit bien du nombre π usuel, voir par exemple P. Eymard & J.-P. Lafon, *Autour du nombre π*, §2.7 (Édition Hermann, Paris 1999). Voici une preuve élégante, contribution de Vincent Munnier (Licence de Mathématiques à l'Institut Fourier en 2004) :

On considère les intégrales de Wallis, $I_k = \int_0^{\pi/2} \sin^k t\, dt$. On trouve $I_0 = \frac{\pi}{2}$ et $I_1 = 1$, puis, par une intégration par parties, la relation de récurrence $I_{k+1} = \frac{k}{k+1} I_{k-1}$. On obtient ainsi $I_{2k+1} = \frac{k!^2 \cdot 2^{2k}}{(2k+1)!}$. Nous constatons que le terme général de notre série est justement $2^{-k} I_{2k+1}$. Nous avons alors :

$$\sum_{k=0}^{\infty} \frac{1\cdot 2\cdots k}{3\cdot 5\cdots(2k+1)} = \sum_{k=0}^{\infty} \int_0^{\pi/2} 2^{-k} \sin^{(2k+1)} t\, dt = \int_0^{\pi/2} \sum_{k=0}^{\infty} 2^{-k} \sin^{(2k+1)} t\, dt$$
$$= \int_0^{\pi/2} \frac{2\sin t}{2-\sin^2 t} = \int_0^{\pi/2} \frac{2\sin t}{1+\cos^2 t} = \big[-2\arctan(\cos t)\big]_0^{\pi/2} = \frac{\pi}{2}$$

Comme exercice, vérifier les détails de ce calcul. Pourquoi peut-on interchanger la somme et l'intégrale ?

3. Développement en base de Wallis et conversion en base b

Comme valeur approchée de π on considère la somme finie

$$s = \sum_{k=0}^{m} 2 \frac{1\cdot 2\cdots k}{3\cdot 5\cdots(2k+1)}. \tag{2}$$

Exercice/M 3.1. Montrer que $s < \pi < s + 2^{1-m}$.

En imitant l'approche du chapitre IV, nous introduisons l'application $\langle\cdot\rangle_\pi : \mathbb{Z}^{m+1} \to \mathbb{Q}$ qui associe à chaque suite finie $x = (x_0, x_1, \dots, x_m)$ le nombre rationnel en « base de Wallis »

$$\langle x\rangle_\pi := \sum_{k=0}^{m} x_k \frac{1\cdot 2\cdots k}{3\cdot 5\cdots(2k+1)} \tag{3}$$

Par exemple $\langle 2,2,2,2\rangle_\pi = 2 + 2\cdot\frac{1}{3} + 2\cdot\frac{1\cdot 2}{3\cdot 5} + 2\cdot\frac{1\cdot 2\cdot 3}{3\cdot 5\cdot 7}$ est le début de la série présentée ci-dessus.

Définition 3.2. On dit que $x \in \mathbb{Z}^{m+1}$ est *normale* par rapport à la base de Wallis si $0 \le x_k \le 2k$ pour tout $k = 1, \dots, m$. (Comme avant nous ne posons aucune restriction sur la valeur x_0.)

Exercice/M 3.3. Toute représentation $x \in \mathbb{Z}^{m+1}$ peut être normalisée par rapport à la base de Wallis, c'est-à-dire qu'il existe une représentation normale $\bar{x} \in \mathbb{Z}^{m+1}$ telle que $\langle\bar{x}\rangle_\pi = \langle x\rangle_\pi$. Montrer que l'algorithme IV.7 ci-dessous calcule une telle représentation normale à partir de x.

Algorithme IV.7 Normalisation par rapport à la base de Wallis

Entrée: Un vecteur $x = (x_0, x_1, \dots, x_m) \in \mathbb{Z}^{m+1}$ représentant $\bar{x} = \langle x\rangle_\pi$ en base de Wallis

Sortie: Le vecteur normal $x = (x_0, x_1, \dots, x_m) \in \mathbb{Z}^{m+1}$ représentant $\bar{x}$ en base de Wallis

pour k **de** m **à** 1 **faire**	// l'indice k parcourt de droite à gauche
$q_k \leftarrow x_k \operatorname{div} (2k+1)$	// calculer la retenue
$x_{k-1} \leftarrow x_{k-1} + k q_k$	// ajouter la retenue à x_{k-1}
$x_k \leftarrow x_k \bmod (2k+1)$	// réduire x_k modulo $2k+1$
fin pour	
retourner $(x_0, x_1, \dots, x_m)$	

Exercice/M 3.4. Si $x \in \mathbb{Z}^{m+1}$ est normale par rapport à la base de Wallis, alors $x_0 \le \langle x\rangle_\pi < x_0 + 4$. Montrer cette majoration, par exemple en comparant avec une série géométrique convenable.

Exercice/M 3.5 (optionnel). Contrairement aux représentations analysées en chapitre IV, on ne peut pas garantir la majoration $\langle x\rangle_\pi < x_0 + 1$. (Trouver un contre-exemple.) En particulier notre preuve que la représentation normale est unique n'est plus valable pour la base de Wallis. (Trouver également un contre-exemple.) Vous pouvez optimiser notre majoration en montrant que $\langle x\rangle_\pi < x_0 + 2$. Plus précisément, montrer que $\langle 0,2,4,6,\dots,2m\rangle_\pi = 2 - 2^{m+1}/\binom{2m+1}{m} \to 2$.

Reste à établir la conversion entre développement en base de Wallis et développement en base b :

Exercice/M 3.6. Montrer que l'algorithme IV.8 ci-dessous convertit la représentation $x \in \mathbb{Z}^{m+1}$ en base de Wallis en une représentation $y \in \mathbb{Z}^{n+1}$ en base b telle que $\langle y\rangle_b \le \langle x\rangle_\pi < \langle y\rangle_b + 4b^{-n}$. Les chiffres $y_1,\dots,y_n$ produits dans la boucle sont-ils forcément normalisés ? Quelle majoration peut-on garantir ? Motiver ainsi la normalisation de y à la fin.

Algorithme IV.8 Changement de base de Wallis en base b avec précision n

Entrée: Un vecteur $x = (x_0, x_1, \dots, x_m) \in \mathbb{Z}^{m+1}$ et deux entiers $b \ge 2$, $n \ge 0$
Sortie: Un vecteur $y = (y_0, y_1, \dots, y_n) \in \mathbb{Z}^{n+1}$, normal par rapport à la base b
Garanties: Les nombres représentés sont proches, càd $\langle y\rangle_b \le \langle x\rangle_\pi < \langle y\rangle_b + 4b^{-n}$

```
pour j de 0 à n faire
  normaliser x par rapport à la base de Wallis   // voir l'algorithme précédent
  y_j ← x_0                                      // y_j n'est pas forcément la partie entière de ⟨x⟩_π, mais …
  x_0 ← 0                                        // en soustrayant y_j on assure au moins que 0 ≤ ⟨x⟩_π < 4
  x ← b·x                                        // multiplier tous les chiffres par b
fin pour
normaliser y par rapport à la base b, retourner y
```

Exercice/M 3.7. Supposons que l'algorithme IV.8 est appliqué à un vecteur initial $x \in \mathbb{Z}^{m+1}$ qui est normal par rapport à la base de Wallis. Montrer que les valeurs intermédiaires qui peuvent intervenir pendant les calculs sont majorées par $4bm$. (Vous pouvez prendre le lemme 2.12 pour modèle.) En utilisant le type `unsigned int` à 32 bits et une base $b \in [\![2,16]\!]$, jusqu'à quelle longueur m et quelle précision p peut-on aller ? Même question pour le type `unsigned short` à 16 bits.

4. De l'analyse mathématique à l'implémentation

Nous pouvons maintenant calculer π à une précision donnée près :
- Tout d'abord, le vecteur $x = (2,2,2,\dots,2,2) \in \mathbb{Z}^{m+1}$ représente une valeur approchée $s := \langle x\rangle_\pi$ de π vérifiant $s < \pi < s + 2^{1-m}$ (voir exercice 3.1).
- En convertissant x de la base de Wallis à la base b, on obtient $y \in \mathbb{Z}^{n+1}$ qui représente une valeur approchée $t := \langle y\rangle_b$ de s telle que $t \le s < t + 4b^{-n}$ (voir exercice 3.6).

Ce procédé produit alors un développement $y \in \mathbb{Z}^{n+1}$ en base b qui représente une valeur approchée t de π telle que $t < \pi < t + \varepsilon$ avec un écart $\varepsilon = 2^{1-m} + 4b^{-n}$.

☞ Si l'on veut arriver à p chiffres valables de π, il faut choisir une précision initiale $\tilde p$ légèrement plus grande que la précision souhaitée p. En voici l'argument détaillé :

Exercice/M 4.1. Étant donné $\tilde p \in \mathbb{N}$ expliquer pourquoi le choix $n = \tilde p + 3$ et $m \ge 2 + \tilde p \log_2 b$ permet de calculer $y \in \mathbb{Z}^{n+1}$ tel que $t = \langle y\rangle_b$ satisfasse $t < \pi < t + \varepsilon$ avec un écart $\varepsilon < b^{-\tilde p}$. Ensuite, tronquer y à $\tilde y = (y_0, y_1, \dots, y_{\tilde p})$ donne presque $\tilde p$ chiffres valables de π : au pire, le dernier chiffre devrait être augmenté de 1, avec éventuelle propagation des retenues sur les chiffres précédents. Malgré cette ambiguïté concernant les derniers chiffres, pourquoi peut-on espérer obtenir au moins $p = 10000$ chiffres valables de π en partant d'une longueur initiale de $\tilde p = 10010$?

☞ Approche pragmatique : on construit d'abord un vecteur $\tilde y$ de longueur $\tilde p$ avec précision $b^{-\tilde p}$, puis on supprime, un par un, les chiffres terminaux dont on ne peut être sûr. Disons, par exemple, les décimales de π aux positions 760 à 769 sont `3499999983`. Si l'on calcule jusqu'à la position 765, on n'est sûr que des chiffres jusqu'à la position 760. En général, il faut supprimer tous les 9 terminaux ainsi que le chiffre d'avant. (Le justifier.) Même s'il restera finalement moins que les p chiffres souhaités, au moins le programme est honnête. Si besoin en est, l'utilisateur pourra le relancer avec une précision p plus grande.

Exercice/P 4.2. Écrire une fonction `approx_pi` qui prend comme paramètres la base b (comprise entre 2 et 16) et la précision souhaitée p (comprise entre 1 et 10^6) et qui construit le vecteur $(t_0, t_1, \ldots, t_p)$ des p premières décimales de π. Veillez à supprimer, un par un, les chiffres terminaux dont on ne peut garantir la correction. Que valent les chiffres aux positions 9991 à 10000 du développement décimal de π ?

☞ Voici quelques indications :
- On n'a besoin que de deux vecteurs x et y. Ne pas oublier de normaliser y à la fin.
- Il sera utile que le programme affiche l'avancement du travail, par exemple la précision achevée (la taille de y déjà atteinte), éventuellement les chiffres y_k déjà obtenus.
- Pour l'affichage on utilisera les chiffres usuels `0123456789ABCDEF`, voir l'exercice 1.15. L'affichage regroupé en blocs de dix chiffres, avec dix blocs par ligne, semble assez lisible sur l'écran.
- Par curiosité et pour une comparaison/vérification rapide de vos résultats, vous pouvez faire une statistique des chiffres jusqu'à la position p donnée.
- Quand votre programme marche et vous vous êtes convaincu de sa correction, nettoyez le code source et rédigez la version finale des commentaires. Essayez de produire un code dont vous serez fier et qui vous plaira toujours d'ici un an.

5. Quelques points de réflexion

Structuration du projet. Reconsidérez le projet IV (ou III ou I) sous l'angle des remarques faites au chapitre IV, §3. Où la spécification s'est-elle faite dans ce projet ? La structuration des données ? L'implémentation fut-elle menée de haut en bas ou de bas en haut ? Arrive-t-on à prouver que le programme est correct ? En quoi les tests étaient-ils décisifs ? Le programme final arrive-t-il à un niveau « publiable » ? Vaut-il la peine de peaufiner la documentation et la présentation du code source ?

☞ Pour un exemple extrême, reconsidérez le programme I.21, à trois lignes seulement :

```
const int a=10000; int b=52514; vector<int> c(b); int d=0, e=0, f, g, h;
for( ; (f=b-=14); d=1, cout << setw(4) << setfill('0') << g+e/a << flush )
  for( g=e%=a; (h=--f*2); e/=h ) e=e*f+a*( d ? c[f] : a/5 ), c[f]=e%--h;
```

Avec beaucoup d'effort on pourrait exhiber de vagues ressemblance avec l'algorithme de ce projet. Mais sans aucun contexte il semble tombé du ciel. (Ou vient-il plutôt de l'enfer ?) Bien que syntaxiquement correct, ce programme est extrêmement mal présenté et l'absence de toute documentation le rend inutilisable.

Complexité quadratique. Notre approche est suffisamment efficace pour calculer 10000 décimales de π, encore un record mondial vers la fin des années 1950. De plus notre calcul ne nécessite que quelques minutes ! Le programme arrive-t-il aussi bien à calculer 10^5 décimales ? puis 10^6 décimales ? Comment expliquer le ralentissement observé ?

Notre algorithme est d'une complexité dite *quadratique* en fonction de la précision p : afin de calculer un par un les n chiffres de y, on itère une boucle sur $1, \ldots, n$. Dans chaque itération la multiplication puis la normalisation de x exécute elle-même une boucle sur $m, \ldots, 1$. Le nombre total des instructions est donc proportionnel au produit mn, qui vaut à peu près $\text{const} \cdot p^2$. Dix fois plus de chiffres nécessitent donc cent fois plus de temps !

Entre-temps des méthodes beaucoup plus efficaces on été développées, et on est ainsi arrivé en 1999 à calculer plus de $2 \cdot 10^{11}$ décimales de π. Pour en savoir plus vous pouvez consultez *Le fascinant nombre* π de Jean-Paul Delahaye (Pour la Science, Paris 1997) ou π *unleashed*, de Jörg Arndt et Christoph Haenel (Springer-Verlag, Berlin 2001).

Généralisation. Notre approche marche plus généralement pour tout nombre réel σ qui se présente sous forme d'une série $\sigma = \sum_{k=0}^{\infty} x_k v_k$ dans un « système positionnel généralisé ». Pour cela on suppose que les « chiffres » x_k sont des entiers, et que les « valeurs positionnelles » v_k vérifient un certain type de récursion : on suppose $v_0 = 1$ et $v_k = v_{k-1} \frac{p_k}{q_k}$ avec deux nombres naturels $p_k, q_k \geq 1$ vérifiant $\frac{p_k}{q_k} \leq \theta$ pour tout k et une constante $\theta < 1$. Ceci englobe nos trois exemples précédents : la base b (avec $p_k = 1$ et $q_k = b$), la base factorielle (avec $p_k = 1$ et $q_k = k+1$), et la base de Wallis (avec $p_k = k$ et $q_k = 2k+1$).

Si vous voulez, vous pouvez étudier cette nouvelle situation, légèrement généralisée, en y étendant le développement précédent. Si jamais vous tombez sur une constante mathématique qui se présente sous cette forme, vous aurez déjà un algorithme quadratique, ce qui permettra de calculer quelques milliers de décimales.