Variations sur le schéma de Horner
(Programmation avec Maple)

Préparation à la nouvelle épreuve d'informatique
de l'École Polytechnique.

Jean-Michel Ferrard

Avant-propos

Ce document vise à la préparation à la nouvelle épreuve d'informatique de l'École Polytechnique.

Les caractéristiques de cette épreuve peuvent être consultées à l'adresse :

http ://www.enseignement.polytechnique.fr/informatique/concours/

Parmi les langages de programmation possibles, on a choisi Maple, dont on n'a utilisé que les fonctionnalités de base, conformément aux demandes des concepteurs de l'épreuve.

Le présent document est consacré à l'algorithme de Horner et à ses variations dans des domaines parfois inattendus. Voici un bref résumé des différentes thèmes abordés :

- Partie I : *Évaluation d'un polynôme*

  C'est l'aspect le plus classique de l'algorithme de Horner. On voit deux méthodes, dont l'une est récursive. On s'intéresse aussi à l'évaluation en un point de $\mathbb{C}$ d'un polynôme à coefficients réels, toujours dans le souci de diminuer le nombre d'opérations à effectuer.

- Partie II : *Division synthétique*

  L'algorithme de Horner (évaluation d'un poynôme P en un point α) cache en fait une méthode de division de P par $x - \alpha$. Dans cette partie, on voit également comment former (à moindre frais) la division euclidienne de P par $X^2 + \alpha X + \beta$.

- Partie III : *Translatés d'un polynôme, dérivées successives*

  On voit ici comment calculer les coefficients du polynôme $P(X + \alpha)$ à partir de ceux de $P(X)$. On constate qu'on y arrive par deux algorithmes de Horner imbriqués. Développer $P(X + \alpha)$ c'est aussi exprimer le polynôme P dans la base des $(X - \alpha)^k$. On en déduit en particulier une méthode pour calculer simultanément les dérivées successives de P en α.

- Partie IV : *Règle des signes de Descartes*

  Elle donne une indication sur les racines positives ou négatives d'un polynôme. L'utilisation de translations permet alors de localiser des racines de P sur n'importe quel intervalle.

- Partie V : *Méthode de Newton de résolution de $P(x) = 0$*

  Elle permet d'approcher une racine réelle du polynôme P. On se sert d'algorithmes de Horner en parallèle pour calculer simultanément les valeurs de $P(x)$ et de $P'(x)$ (voire de $P''(x)$.)

- Partie VI : *Forme de Newton du polynôme interpolateur*

  Il s'agit d'écrire ici le polynôme interpolateur d'une famille de points sous une forme qui permet (entre autres) facilement l'ajout d'un point supplémentaire. L'évaluation de ce polynôme s'effectue par une forme particulière de l'algorithme de Horner.

- Partie VII : *Autour du théorème chinois*

  Dans cette partie, on voit comment trouver la solution d'un système de congruences. On voit que des calculs "à la Horner" permettent d'obtenir cette solution en minimisant le nombre d'opérations, et surtout en limitant la taille des calculs intermédiaires.

- Partie VIII : *Algorithmes "compte-gouttes"*

  Dans cette partie, on étudie et on met en œuvre des méthodes qui permettent d'obtenir rapidement un grand nombre de décimales des nombres e et π. Là encore, ce sont des calculs "à la Horner".

Table des matières

Énoncé

Un polynôme P est représenté par un tableau unidimensionnel T de réels, indicé à partir de 1.

Le type d'un tel tableau est `array(numeric)`.

Un tel tableau, lu dans l'ordre des indices croissants, représente un unique polynôme représenté lui-même dans le sens des puissances décroissantes.

Par exemple, le tableau $T = [3, 0, -5, 2, 1]$ représente le polynôme $P = 3X^4 - 5X^2 + 2X + 1$.

On utilisera la fonction `size` suivante pour calculer la taille d'un tableau unidimensionnel T : c'est un majorant strict du degré du polynôme P associé à T.

```
> size:=proc(T::array(numeric))
>   RETURN(op(2,op(2,eval(T))));
> end:
```

Par exemple, les tableaux T et U sont de tailles respectives 5 et 7. Evidemment, ils représentent deux polynômes qui sont de degrés respectifs 4 et 2.

```
> T:=array([3,0,-5,2,1]):
> U:=array([0,0,0,0,2,4,1]):
> size(T), size(U);
```

$$5\ ,\ 7$$

Par commodité, on nommera de la même manière un polynôme P et le tableau qui le représente.

I. Évaluation d'un polynôme

Dans cette première partie, on voit comment évaluer un polynôme P à coefficient réels avec l'algorithme de Horner, c'est-à-dire en se basant sur la deuxième expression ci-dessous de $P(t)$.

$$P(t) = \sum_{k=0}^{n} a_k t^k = ((\cdots(((a_n t + a_{n-1})\, t + a_{n-2})\, t + a_{n-3} + \cdots)\, t + a_1)\, t + a_0$$

1. Écrire une fonction `evalh` évaluant P en le réel t, avec la syntaxe `evalh(`P, t`)`.
   Cette valeur sera calculée par un algorithme de Horner itératif.
   Indiquer le nombre d'additions et de multiplications nécessitées par cet algorithme. [S]
2. Écrire une version récursive de la fonction `evalh`, avec la même syntaxe d'appel. [S]
3. Dans cette question, on cherche à évaluer $P(z)$, où $z = x + iy$ est un nombre complexe.
   On suppose que les seules opérations possibles sont l'addition et le produit de nombres réels.
   Écrire une fonction `evalhc`, déduite de la version itérative de `evalh` pour que `evalhc(`P, x, y`)` renvoie le tableau $[x', y']$ donnant la partie réelle et la partie imaginaire de $P(x + iy)$.
   Combien cette méthode nécessite-t-elle d'opérations arithmétiques sur les réels ? [S]
4. Écrire une autre version de `evalc`, plus écononome en opérations sur les réels.
   On pensera à la division euclidienne de P par $X^2 - 2\mathrm{Re}\,(z)X + |z|^2$. [S]

II. Division synthétique

Soit P un polynôme et t un réel. On vient de voir comment calculer $y = P(t)$.
Dans les questions 1 et 2, on voit comment effectuer la division de P par $X - t$.
On sait que $P(t)$ est le reste dans cette division : il faut donc calculer le quotient Q.

1. Écrire une fonction `quo1` donnant le quotient Q de P par le monôme $X - t$.
   La syntaxe d'appel sera `quo1(`P, t`)` et le résultat sera le tableau associé à Q. [S]

2. Écrire une *procédure* `Quo1` calculant à la fois le quotient Q et le reste $R = P(t)$.
   Cette procédure modifie le tableau P en y plaçant les coefficients de Q puis $R = P(t)$
   La syntaxe d'appel sera `Quo1(`P, n, t`)`, où n est la taille du tableau P.
   L'argument n signifie en fait qu'on utilise les n premiers coefficients du tableau P.
   On verra dans la question suivante quelle utilité il y a à utiliser cette syntaxe. [S]

3. Dans cette question, on divise P par $X^2 + \alpha X + \beta$.
   On note $P = (X^2 + \alpha X + \beta) \sum_{k=0}^{n-2} b_{k+2} X^k + b_1 X + b_0$ cette division.
   Écrire une *procédure* `Quo2` calculant à la fois le quotient Q et le reste $R = b_1 X + b_0$.
   On utilisera la syntaxe `Quo2(`P, α, β`)`.
   Cette procédure transforme le tableau $[a_n, a_{n-1}, \ldots, a_0]$ en $[\overbrace{b_n, b_{n-1}, \ldots, b_2}^{Q}, \overbrace{b_1, b_0}^{R}]$ [S]

III. Translatés d'un polynôme, dérivées successives

On considère un polynôme $P = \sum_{k=0}^{n} a_k X^k$, et t un réel.
On sait qu'il est possible d'écrire P sur la base des polynômes $(X - t)^k$, $k \in \mathbb{N}$.
Plus précisément : $P(X) = \sum_{k=0}^{n} b_k (X - t)^k \Leftrightarrow P(X + t) = \sum_{k=0}^{n} b_k X^k$.
On se propose ici de voir comment passer du tableau $[a_n, \ldots, a_1, a_0]$ au tableau $[b_n, \ldots, b_1, b_0]$.
On résout ainsi deux problèmes équivalents :
- Trouver les coefficients de $Q(X) = P(X + t)$ (*translaté* de P) sur la base des X^k.
- Trouver les coefficients du polynôme P sur la base des $(X - t)^k$.

1. Avec les notations ci-dessus, écrire une procédure `Translat` modifiant $P = [a_n, a \ldots, a_1, a_0]$ pour y placer les coefficients $b_n, \ldots, b_1, b_0$.
   La syntaxe est `Translat(`P, t`)`, et on fera appel à la procédure `Quo1`. [S]

2. Écrire une *fonction* `translat`, renvoyant le tableau $Q = [b_n, \ldots, b_1, b_0]$.
   On utilisera la syntaxe `Translat(`P, t`)`, et on ne fera pas appel à la procédure `Quo1`. [S]

3. Vérifier que `translat` nécessite $\frac{1}{2}(n^2 - n)$ additions et autant de multiplications.
   Écrire une autre version de cette fonction nécessitant moins de $3n$ multiplications.
   Indication : en utilisant des homothéties de rapport t ou $1/t$, se ramener à $t = 1$.
   Cette économie se fait au prix d'un tableau supplémentaire contenant $[t^n, t^{n-1}, \ldots, t, 1]$. [S]

4. Écrire une fonction `derivs` prenant en argument un polynôme P (représenté par le tableau $[a_n, \ldots, a_1, a_0]$) et un réel t, et renvoyant le tableau $[P^{(n)}(t), \ldots, P'(t), P(t)]$ des dérivées successives de P au poit t (par ordre décroissant de l'ordre de dérivation.) [S]

IV. Règle des signes de Descartes

Considérons le polynôme $P = \sum_{k=0}^{n} a_k X^k$, représenté par le tableau $P = [a_n, a_{n-1}, \ldots, a_1, a_0]$.

On appelle *changement de signe* dans P tout couple (i, j), avec $i < j$ et :

– Les coefficients a_i et a_j sont non nuls et de signe contraire.

– Pour tout k tel que $i < k < j$ on a $a_k = 0$.

On observe donc un changement de signe quand deux coefficients non nuls consécutifs de P (ordonné suivant les puissances croissantes ou décroissantes) sont de signes contraires.

Par exemple, le polynôme $P = X^8 - 3X^5 + 2X^4 + X^2 - X - 1$ présente 3 changements de signe.

Notons s le nombre de changements de signe de P.

La **règle de Descartes** affirme que le nombre r de racines réelles strictement positives de P est inférieur ou égal à s, et plus précisément que la différence $s - r$ est un entier pair.

Par exemple, cette règle permet d'affirmer que le polynôme $P = X^8 - 3X^5 + 2X^4 + X^2 - X - 1$ possède ou bien trois ou bien une seule racine(s) réelle(s) strictement positive(s).

Appliquée à $P(-X)$, cette règle donne une indication sur les racines réelles strictement négatives.

Avec notre exemple, $P(-X) = X^8 + 3X^5 + 2X^4 + X^2 + X - 1$ présente un seul changement de signe. Le polynôme P possède donc exactement une racine réelle strictement négative.

1. Écrire une fonction `varsign` donnant le nombre de changements de polynôme P. [S]

2. Écrire une fonction `rootsup` donnant un majorant du nombre de racines de P strictement supérieures à un réel donné a. On utilisera la syntaxe `rootsup(`P, a`)`. [S]

3. Écrire une fonction `rootinf` donnant un majorant du nombre de racines de P strictement inférieures à un réel donné a. On utilisera la syntaxe `rootinf(`P, a`)`. [S]

V. Méthode de Newton de résolution de $P(x) = 0$

Soit $f : I \subset \mathbb{R} \to \mathbb{R}$, de classe $\mathcal{C}^k$ $(k \geq 1)$. La méthode de Newton consiste à chercher une solution x de $f(x) = 0$ en formant une suite définie par $x_0 \in I$ et par $\forall\, n \in \mathbb{N}, x_{n+1} = x_n - \dfrac{f(x_n)}{f'(x_n)}$.
Dans la suite de cette question f est un polynôme P à coefficients réels.
La convergence n'est pas assurée (sauf vers la plus grande racine réelle α de P si $x_0 > \alpha$.)
La vitesse de convergence est quadratique vers une racine simple, et linéaire vers une racine multiple.

1. Écrire une fonction `newton` donnant x_{n+1} connaissant P et x_n.
   On utilisera évidemment une schéma de Horner pour évaluer $P(x_n)$ et $P'(x_n)$.
   On fera en sorte que ces deux schémas soient menés en parallèle. [S]
2. On peut songer à appliquer la méthode de Newton à la fraction rationnelle $f = \dfrac{P}{P'}$.
   Ses zéros sont ceux de P et ils sont tous simples.
   Cela assure une vitesse de convergence au moins quadratique pour toutes les racines de P.
   Dans ce cas on doit donc utiliser la relation : $x_{n+1} = x_n - \dfrac{P(x_n)P'(x_n)}{P'^2(x_n) - P(x_n)P''(x_n)}$.
   Écrire alors une fonction `newton2` donnant x_{n+1} connaissant P et x_n. On fera en sorte que les calculs de $P(x_n), P'(x_n), P''(x_n)$ soient menés en parallèle. [S]

VI. Forme de Newton du polynôme interpolateur

Soit $\mathcal{F}$ une famille de n points $A_k(x_k, y_k)$ du plan, avec $1 \leq k \leq n$, d'abscisses x_k distinctes.
On sait qu'il existe un polynôme unique P, de degré $\leq n-1$, tel que $P(x_k) = y_k$ pour tout k.
Il y a plusieurs façons d'écrire ce polynôme interpolateur, dont la *forme de Newton*.
Elle consiste à écrire P dans la base $H_1, H_2, \ldots, H_n$ de $\mathbb{R}_{n-1}[X]$ définie par :

$$H_1 = 1, \quad H_2 = X - x_1, \quad H_3 = (X - x_1)(X - x_2), \quad \ldots, \quad H_n = (X - x_1)(X - x_2)\cdots(X - x_{n-1})$$

Le problème est double :
- Calculer les coordonnées de P sur la base $H_1, H_2, \ldots, H_n$.
- Connaissant ces coordonnées, évaluer P en un point quelconque.
  C'est dans ce deuxième problème qu'intervient un schéma de Horner.

1. Pour $1 \leq i \leq j \leq n$, on définit récursivement les coefficients $d_{i,j}$ de la manière suivante :
   Pour tout i de $\{1, \ldots, n\}$, $d_{i,i} = y_i$. Si $i < j$, on pose $d_{i,j} = \dfrac{d_{i+1,j} - d_{i,j-1}}{x_j - x_i}$.
   Écrire une fonction `dij` calculant le coefficient d'indice i, j.
   La syntaxe sera `dij`(X, Y, i, j) où X, Y sont les tableaux des abscisses et ordonnées.
   La fonction `dij` utilisera bien sûr la définition récursive des coefficients $d_{i,j}$. [S]

2. Avec les notations ci-dessus, on montre que $P = d_{1,1}H_1 + d_{1,2}H_2 + \cdots + d_{1,n}H_n$.
   Écrire une fonction `pnewton` formant le tableau $[d_{1,1}, d_{1,2}, \ldots, d_{1,n}]$ à partir des tableaux X, Y (tous les deux indicés de 1 à n) des abscisses et ordonnées.
   La fonction `pnewton` s'appuiera essentiellement sur la fonction `dij` qui calcule les coefficients $d_{i,j}$ (et en particulier les $d_{1,k}$) de façon récursive. [S]

3. La méthode précédente a un défaut : des calculs intermédiaires sont effectués plusieurs fois.
   D'autre part, on voit bien qu'il suffit ici de calculer les seuls coefficients $d_{1,k}$.
   Écrire une *procédure* `PNewton` calculant le tableau $[d_{1,1}, d_{1,2}, \ldots, d_{1,n}]$.
   On n'utilisera pas la fonction récursive `dij`. Au contraire les coefficients $d_{1,k}$ seront calculés par une méthode itérative. Voici une indication sur les premières étapes du calcul :
   - On part du tableau $Y = [d_{1,1}, d_{2,2}, \ldots, d_{n,n}]$.
   - La première étape transforme ce tableau en $[d_{1,1}, d_{1,2}, \ldots, d_{j,j+1}, \ldots, d_{n-1,n}]$.
   - L'étape suivante conduit à $[d_{1,1}, d_{1,2}, d_{1,3}, \ldots, d_{j,j+2}, \ldots, d_{n-2,n}]$, etc.

   A la $n-1$-ième étape, on aboutit donc au tableau $[d_{1,1}, d_{1,2}, d_{1,3}, \ldots, , d_{1,n}]$.
   On utilisera la syntaxe `PNewton(`X, Y`)`. Les calculs seront faits dans le tableau Y. [S]

4. Avec la forme du Newton, il est très facile d'ajouter un nouveau point $A_0(x_0, y_0)$.
   Écrire une fonction `newpoint` passant du polynôme interpolateur P de $(x_1, y_1), \ldots, (x_n, y_n)$ (écrit sous sa forme de Newton) à celui des $n+1$ points $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$.
   Avec la syntaxe `newpoint(`X, P, x_0, y_0`)`, où X est la liste des n abscisses initiales, le résultat sera le tableau de la forme de Newton du polynôme interpolant $A_0, \ldots, A_n$. [S]

5. Écrire une fonction `evalnewton` calculant la valeur en un point x quelconque du polynôme d'interpolation P des n points $A_k(x_k, y_k)$. La syntaxe sera `evalnewton(`X, P, x`)`, où X est la liste des n absicsses et où P est le tableau de taille n représentant la forme de Newton du polynôme interpolateur. [S]

VII. Autour du théorème chinois

Dans cette partie (réservée aux élèves de MP*), on s'intéresse au "théorème chinois".

- Soient m, n dans $\mathbb{N}^*$, avec $m \wedge n = 1$. On sait qu'il existe une infinité de couples (u, v) de $\mathbb{Z}^2$ tel que $um + vn = 1$. On dit que u et v sont des *coefficients de Bezout* de m et n.
  Il existe en particulier un couple (u, v) unique tel que $|u| \leq \frac{n}{2}$ et $|v| \leq \frac{m}{2}$.
  Il est obtenu par l'algorithme d'Euclide (divisions successives) appliqué au couple (m, n).

- On se donne r entiers strictement positifs $n_1, n_2, \ldots, n_r$ premiers entre eux deux à deux.
  Soit $n = \prod_{k=1}^{r} n_k$. On considère l'application φ de $\mathbb{Z}/n\mathbb{Z}$ dans $\prod_{k=1}^{r} \mathbb{Z}/n_k\mathbb{Z}$ définie par :
  $$\forall x \in \mathbb{Z}/n\mathbb{Z}, \quad \varphi(x) = (x \bmod n_1,\ x \bmod n_2,\ \ldots,\ x \bmod n_r)$$

Cette application est un morphisme d'anneaux, injectif car son noyau est réduit à 0.
Il est donc bijectif pour des raisons de cardinal.
En particulier, pour tous entiers $a_1, \dots, a_r$: $\exists\,!\, x \in [0, \dots, n-1]$, $\begin{cases} x \bmod n_1 = a_1 \\ x \bmod n_2 = a_2 \\ \dots \\ x \bmod n_r = a_r \end{cases} \quad (S)$
Ce résultat est communément appelé *théorème chinois*.

1. *Calcul des coefficients de Bezout.*
   (a) Écrire une fonction récursive `bezout` recevant les entiers positifs m, n et renvoyant dans un tableau $[u, v, u \wedge v]$ les coefficients u, v tels que $um + vn = u \wedge v$. Pour cela, et si $m = nq + r$ est la division de m par n, on notera comment passer d'un couple de coefficients de Bezout de (n, r) à un couple de coefficients de Bezout de (m, n). [S]

   (b) Écrire une version itérative de la fonction `bezout`.
   Indication : considérer les équations (E_δ) : $am + bn = \delta$, d'inconnue (a, b) dans $\mathbb{Z}^2$.
   Le triplet $(1, 0)$ est solution de (E_m), et $(0, 1)$ est solution de (E_n).
   Soit $\alpha = q\beta + r$ la division euclidienne d'un entier α par un entier β.
   On suppose que (a_1, b_1) est solution de (E_α) et que (a_2, b_2) est solution de (E_β).
   On remarque alors que $(a_1 - qa_2, b_1 - qb_2)$ est solution de (E_r). [S]

2. *Solution d'un système de congruences*
   (a) Pour tous i, j de $\{1, \dots, r\}$ (avec $i \neq j$) on note $u_{i,j}$ et $u_{j,i}$ tels que $u_{i,j}\, n_i + u_{j,i}\, n_j = 1$.
   Pour tout j de $\{1, \dots, r\}$, on note $M_j = \prod_{i \neq j} u_{i,j}\, n_i$, et on pose $x = \left(\sum_{j=1}^{r} a_j M_j\right) \bmod n$.
   Montrer que l'entier x est l'unique solution du système (S).
   Écrire une fonction `chinese` calculant x (syntaxe `chinese(`$[a_1, \dots, a_r], [n_1, \dots, n_r]$`)`). [S]

   (b) Dans cette question, on calcule la solution x, au moyen de deux schémas de Horner.
   On pose $b_1 = a_1$, puis $b_2 = (a_2 - b_1)u_{1,2} \mod n_2$, etc, et finalement
   $$b_r = [([(a_r - b_1)u_{1,r} - b_2]u_{2,r} - b_3) \cdots - b_{r-1}]\, u_{r-1,r} \mod n_r$$
   Montrer que la solution x du système (S) s'écrit $x = \sum_{k=1}^{r} \left(b_k \prod_{i=1}^{k-1} n_i\right)$.
   Vérifier que x peut être calculé en utilisant un schéma de Horner.
   En déduire une nouvelle version de la fonction `chinese`.

   Remarque : Cette deuxième méthode a l'avantage de minimiser le nombre de "modulos" à calculer, et de ne produire aucun calcul intermédiaire qui sortirait de $[0, \dots, n-1]$. [S]

VIII. Algorithmes "compte-gouttes"

Dans cette section, on étudie des algorithmes permettant de déterminer une à une les décimales de e et de π. Connus sous le nom d'algorithmes "compte-gouttes", leur particularité est de n'utiliser que l'arithmétique des "petits" entiers, et de donner une à une les décimales successives du nombre considéré sans réutiliser les décimales déjà calculées. Ces algorithmes opèrent une conversion d'un

système de numération "à base variable" vers la numération décimale (ou en base 10^p si on veut p chiffres à la fois.) Cette conversion s'effectue en utilisant des calculs "à la Horner".

1. *Une numération à base variable, adaptée au nombre* e

   On note $\mathcal{B}$ l'ensemble des suites d'entiers $(b_n)_{n\geq 1}$ telles que
   – Pour tout $n \geq 2$, $b_n \in \{0, \ldots, n-1\}$, mais b_1 est quelconque dans $\mathbb{Z}$.
   – Pour tout $n \geq 1$, il existe $m \geq n$ tel que $b_m < n-1$.

   On va voir que tout x s'écrit d'une manière unique $x = \sum_{n=1}^{\infty} \frac{b_n}{n!}$, où la suite (b_n) est dans $\mathcal{B}$.
   On pourra alors noter $x = (b_1, b_2, \ldots, b_n, \ldots)_b$.

   On peut comparer ce développement avec la représentation décimale.

   Notons en effet $\mathcal{D}$ l'ensemble des suites d'entiers $(d_n)_{n\geq 1}$ telles que
   – Pour tout $n \geq 2$, $d_n \in \{0, \ldots, 9\}$, mais d_1 est quelconque dans $\mathbb{Z}$.
   – Pour tout $n \geq 1$, il existe $m \geq n$ tel que $d_m < 9$.

   On sait qu'on a une unique écriture $x = \sum_{n=1}^{\infty} \frac{d_n}{10^{n-1}}$, où la suite $(d_n)_{n\geq 1}$ est dans $\mathcal{D}$.
   Plus précisément, on a $d_1 = [x]$ et, pour tout $n \geq 2$, $d_n = [10^{n-1}x] \bmod 10$.

   Ce qui distingue les deux types de numération, c'est que l'écriture décimale utilise la base fixe $d = 10$ (et on décompose sur des puissances successives de $1/10^n$) alors que l'écriture $x = (b_1, b_2, \ldots, b_n, \ldots)_b$ utilise une "base" variable $1, 2, 3, \ldots$, et une décomposition sur les $1/n!$

   Evidemment pour $\mathrm{e} = \exp(1)$, on a : $\mathrm{e} = 2 + \sum_{k=2}^{\infty} \frac{1}{k!}$ c'est-à-dire $\mathrm{e} = (2, 1, 1, \ldots, 1, \ldots)_b$.

   Un algorithme de conversion de la base variable b à la base 10 (ou mieux à une base 10^p), permettra donc de récupérer les chiffres décimaux du nombre e.

   (a) Soit $(b_n)_{n\geq 1}$ une suite de $\mathcal{B}$. Montrer que $\sum_{k\geq 1} \frac{b_k}{k!}$ converge. On note x sa somme.
   Pour tout $n \geq 0$, on pose $x_n = \sum_{k=1}^{n} \frac{b_k}{k!}$ (par convention, $x_0 = 0$) et $r_n = \sum_{k=n+1}^{\infty} \frac{b_k}{k!}$.
   Montrer que pour tout n de $\mathbb{N}^*$, on a : $0 \leq r_n < \frac{1}{n!}$
   En déduire que $b_1 = [x]$ et, pour tout $n \geq 2$, $b_n = [n!x] - n!x_{n-1} = [n!x] \bmod n$. [S]

   (b) Soit x un nombre réel.
   Montrer qu'il existe une suite unique $(b_n)_{n\geq 1}$ de $\mathcal{B}$ telle que $x = \sum_{k=1}^{\infty} \frac{b_k}{k!}$.
   $\forall n \geq 1,\ x_n = \sum_{k=1}^{n} \frac{b_k}{k!}$ est alors une valeur approchée de x par défaut à $\frac{1}{n!}$ près. [S]

   (c) Écrire une fonction `todec` calculant $x_n = \sum_{k=1}^{n} \frac{b_k}{k!}$ à partir du tableau $B = [b_1, \ldots, b_n]$.
   On utilisera la syntaxe `todec`(B, n), sans vérifier si le tableau B est correct. [S]

   (d) Écrire une fonction `tovar` renvoyant $B = [b_1, \ldots, b_n]$ à partir de $x_n = \sum_{k=1}^{n} \frac{b_k}{k!}$.
   On utilisera la syntaxe `tovar`(x_n, n). [S]

2. *Le calcul des décimales du nombre* e

   (a) Soit x un nombre réel. On sait que x s'écrit de manière unique $x = \sum_{k=1}^{+\infty} \frac{b_k}{k!}$ avec $(b_n) \in \mathcal{B}$.
   Pour tout $n \geq 2$, on note $x_n = \sum_{k=1}^{n} \frac{b_k}{k!} = (b_1, b_2, \ldots, b_n)_b$.
   Soit m dans $\mathbb{N}^*$. On se propose de trouver la partie entière de $10^m x_n = \sum_{k=1}^{n} \frac{10^m\, b_k}{k!}$.
   Il est bien sûr possible d'écrire $10^m x_n = (10^m\, b_1, 10^m\, b_2, \ldots, 10^m\, b_n)_b$.
   Mais dans cette écriture, les $10^m\, b_k$ (pour $k \geq 2$) ne sont en général pas dans $\{0, \ldots, k-1\}$.
   On doit donc convertir cette écriture vers la base $\mathcal{B}$.
   Notons $10^m x_n = (r_1, r_2, \ldots, r_n)_b$ l'écriture de $10^m x_n$ dans cette base.
   Imaginer un algorithme permettant de passer de $(b_1, b_2, \ldots, b_n)_b$ à $(r_1, r_2, \ldots, r_n)_b$.
   Indication : procéder par divisions successives avec report de retenue, de b_n à b_2. [S]

   (b) Écrire une *procédure* `goutte` :
   – Prenant en argument le tableau $X = [b_1, b_2, \ldots, b_n]$ et l'entier $m \geq 1$.
   – Plaçant le tableau $[r_1, r_2, \ldots, r_n]$ dans la variable X.
   Remarque : cette méthode donne en particulier la partie entière r_1 de $10^m x_n$. [S]

   (c) Montrer l'inégalité $\frac{1}{n!} < 10^{-(n+1)}$ pour $n \geq 27$. En déduire une fonction `chiffres_e` donnant les n premières décimales de e, avec la syntaxe `chiffres_e`(n, m), le paramètre m indiquant que les décimales sont obtenues par blocs de m chiffres successifs.
   Le résultat sera donné sous la forme d'une chaîne de caractères. [S]

3. *Le calcul des décimales de* π

   On admet l'égalité suivante : $\frac{\pi}{2} = \sum_{k=0}^{+\infty} \frac{(k!)^2\, 2^k}{(2k+1)!}$.
   Le terme général de cette série s'écrit $u_k = \frac{1 \cdot 2 \cdot 3 \cdots k}{3 \cdot 5 \cdot 7 \cdots (2k+1)}$.
   Ainsi $\frac{\pi}{2} = 1 + \frac{1}{3} + \frac{1 \cdot 2}{3 \cdot 5} + \frac{1 \cdot 2 \cdot 3}{3 \cdot 5 \cdot 7} + \cdots = 1 + \frac{1}{3}\left(1 + \frac{2}{5}\left(1 + \frac{3}{7}\left(1 + \frac{4}{9}\,(1 + \cdots)\right)\right)\right)$.
   On peut donc considérer le système de numération $\mathcal{P}$ à base variable $1, \frac{1}{3}, \frac{1 \cdot 2}{3 \cdot 5}, \frac{1 \cdot 2 \cdot 3}{3 \cdot 5 \cdot 7}, \ldots$
   Dans ce système de numération, on a visiblement : $\pi = (2, 2, 2, 2, \ldots)_{\mathcal{P}}$.
   Plus généralement, on considère des développements de la forme suivante :

$$x = \sum_{k=0}^{+\infty} p_k\, u_k = p_0 + \sum_{k=1}^{+\infty} p_k\, \frac{1 \cdot 2 \cdot 3 \cdots k}{3 \cdot 5 \cdot 7 \cdots (2k+1)} = p_0 + \frac{1}{3}\left(p_1 + \frac{2}{5}\left(p_2 + \frac{3}{7}\left(p_3 + \frac{4}{9}\,(p_4 + \cdots)\right)\right)\right)$$

   On notera $x = (p_0, p_1, p_2, \ldots)_{\mathcal{P}}$ le réel représenté par ce développement infini, avec $p_0 \in \mathbb{Z}$.
   On dit qu'un tel développement est *régulier* si $p_k \in \{0, \ldots, 2k\}$, pour tout $k \geq 1$, et si pour tout entier $n \geq 1$, il existe $m \geq n$ tel que $p_m < 2k$.
   On notera alors $x_n = (p_0, p_1, p_2, \ldots, p_{n-1}, 0, 0, \ldots)_{\mathcal{P}}$, pour tout $n \geq 1$.

(a) Montrer qu'un développement *régulier* $(0, p_1, p_2, p_3, \ldots)_{\mathcal{P}}$ converge vers un réel de $[0, 2[$.

La partie entière de $(p_0, p_1, p_2, p_3, \ldots)_{\mathcal{P}}$ est donc égale à p_0 ou à $p_0 + 1$.

Montrer qu'il n'y a pas unicité de la représentation d'un réel x dans le système $\mathcal{P}$. [S]

(b) Écrire une fonction `todec2` calculant $x_n = \sum_{k=0}^{n-1} p_k\, u_k$ à partir de $X = [p_0, \ldots, p_{n-1}]$.

On utilisera la syntaxe `todec2(`X`)`, sans vérifier si le tableau X est correct. [S]

(c) Soit $x_n = (p_0, p_1, p_2, \ldots, p_{n-1}, 0, \ldots)_{\mathcal{P}}$ la somme d'un développement régulier fini.

Soit m un entier strictement positif. Imaginer un algorithme permettant d'obtenir un développement régulier de $10^m x_n$. [S]

(d) Ecrire une fonction `goutte2`, comme celle de la question (2b), réalisant la conversion étudiée à la question précédente. Vérifier qu'une application répétée de cette fonction donne par exemple les 30 premières décimales de $\pi_{21} = (2, 2, 2, \ldots, 2)_{\mathcal{P}} = 2 \sum_{k=0}^{20} u_k$. [S]

(e) On sait que $\pi = 2 \sum_{k=0}^{+\infty} u_k$. Pour tout $m \geq 1$, on pose $\pi_m = 2 \sum_{k=0}^{m-1} u_k$.

Vérifier que $0 < \pi - \pi_m < 4u_m$ (observer que $u_k < \frac{1}{2} u_{k-1}$ pour tout $k \geq 1$.)

Montrer en outre que u_m est strictement inférieur à $\frac{2}{3}\, 2^{-m}$, pour $m \geq 1$.

En déduire que si $m \geq \frac{10}{3}\, n$, alors $0 < \pi - \pi_m < 5 \cdot 10^{-n}$. [S]

(f) Déduire de ce qui précède une fonction `chiffres_pi` donnant les n premières décimales de π, avec la syntaxe `chiffres_pi`(n, m), le paramètre m indiquant que les décimales sont calculées par groupes de m chiffres consécutifs.

Le résultat sera donné sous la forme d'une chaîne de caractères. [S]

Corrigé

I. Évaluation d'un polynôme

Deux versions de l'algorithme de Horner :

1. Le polynôme $P = \sum_{k=0}^{n} a_k X^k$ est représenté par le tableau $P = [a_n, a_{n-1}, \ldots, a_1, a_0]$.

   Le calcul de $y = P(t)$ s'effectue en posant d'abord $y = 0$, puis successivement :
   $$y \leftarrow yt + a_n = a_n, \quad y \leftarrow yt + a_{n-1} = a_n t + a_{n-1},$$
   $$y \leftarrow yt + a_{n-2} = a_n t^2 + a_{n-1} t + a_{n-2}, \ \ldots, \quad y \leftarrow yt + a_0 = P(t)$$
   Voici donc la fonction itérative `evalh` qui calcule $y = P(t)$:

```
> evalh:=proc(P::array(numeric),t::numeric)
>   local y,k; y:=0;                        # initialisation
>   for k to size(P) do y:=y*t+P[k] od;     # boucle de calcul de y = P(t)
>   RETURN(y)                               # renvoie la valeur calculée
> end:
```

   On calcule ici la valeur du polynôme $P = 3X^4 - 5X^2 + 2X + 1$ au point $t = 100$:

```
> P:=array([3,0,-5,2,1]): evalh(P,100);
```

   $$299950201$$

   Il est clair que l'algorithme précédent nécessite n additions et n multiplications. [Q]

2. Soit $P = \sum_{k=0}^{n} a_k X^k$, et le quotient $Q = \sum_{k=0}^{n-1} a_{k+1} X^k$ dans la division de P par X.

   Pour tout réel t, on a donc $y = P(t) = tQ(t) + a_0$.

   Les polynômes P et Q sont représentés par les tableaux $\begin{cases} P = [a_n, a_{n-1}, \ldots, a_2, a_1, a_0] \\ Q = [a_n, a_{n-1}, \ldots, a_2, a_1] \end{cases}$

   On voit que pour calculer $y = P(t)$, il suffit de calculer $z = Q(t)$ et de poser $y = tz + a_0$.

   Pour évaluer $Q(t)$, on peut encore utiliser le tableau initial P, à condition de se limiter aux $n+1$ premiers coefficients, c'est-à-dire à $a_n, a_{n-1}, \ldots, a_2, a_1$.

   Il en découle la fonction `evalhr`, version récursive de l'algorithme de Horner.

   Tout le travail est effectué par la fonction locale `h`, qui utilise toujours le tableau P représentant le polynôme initial, mais qui reçoit en argument la longueur du sous-tableau de P représentant l'un polynômes quotients successifs. Pour calculer $y = P(t)$, il suffit donc d'un appel initial à cette fonction locale, en lui transmettant la taille du tableau P.

```
> evalhr:=proc(P::array(numeric),t::numeric)
>   local h;
>   h:=proc(m)
>     if m=1 then P[1] else h(m-1)*t+P[m] fi;
>   end:
>   RETURN(h(size(P)));     # appel initial, sur tout la longueur de P
> end:
```

On reprend l'exemple qui a servi à illustrer la version récursive de l'algorithme :

```
> P:=array([3,0,-5,2,1]): evalhr(P,100);
```

$$299950201$$

Remarque : si on s'en tient aux directives des concepteurs de la nouvelle épreuve d'informatique de l'X, on doit éviter d'emboîter les fonctions. Il faut donc "sortir" la fonction locale `h`.

Une première solution est de réécrire la fonction `evalhr` de la manière suivante, en modifiant la syntaxe d'appel et en confiant donc à l'utilisateur le soin de préciser la taille du tableau initial :

```
> evalhr2:=proc(P::array(numeric),n::integer,t::numeric)
>   if n<=1 then P[1] else evalhr2(P,n-1,t)*t+P[n] fi;
> end:
```

On reprend encore le même exemple (remarquer que les syntaxes d'appel sont différentes.)

```
> P:=array([3,0,-5,2,1]): evalhr2(P,5,100);
```

$$299950201$$

[Q]

3. Voici la fonction `evalhc`, directement calquée sur la version itérative de `evalh`.
   On utilise les variables locales u et v pour désigner la partie réelle et la partie imaginaire du nombre complexe qui finira par être égal à $P(x+iy)$.

```
> evalhc:=proc(P::array(numeric),x::numeric,y::numeric)
>   local u,v,k,t; u:=0; v:=0;
>   for k to size(P) do
>      t:=u*x-v*y+P[k]; v:=u*y+v*x; u:=t;
>   od;
>   RETURN(array([u,v]))
> end:
```

   On calcule ici $P(-11+8i)$, avec $P = 3X^4 - 5X^2 + 2X + 1$.

```
> P:=array([3,0,-5,2,1]): evalhc(P,-11,8);
```

$$[-83487, -59296]$$

   On vérifie tout de même que le résultat est correct :

```
> p:=x->evalc(3*x^4-5*x^2+2*x+1): p(-11+8*I);
```

$$-83487 - 59296I$$

   On voit que l'algorithme précédent nécessite $3n$ additions et $4n$ multiplications de réels. [Q]

4. Posons $r = 2\mathrm{Re}\,(z) = 2x$ et $m = |z|^2 = x^2 + y^2$.
   Considérons la division du polynôme $P = \sum_{k=0}^{n} a_k X^k$ par $B = (X-z)(X-\bar{z}) = X^2 - rX + m$.
   Cette division euclidienne s'écrit $P = (X^2 - rX + m)\sum_{k=0}^{n-2} b_{k+2}X^k + b_1 X + b_0$.
   Avec ces notations, on a bien sûr $P(z) = b_1 z + b_0 = (b_1 x + b_0) + ib_1 y$.

Par identification, on trouve tout d'abord : $b_n = a_n$, $b_{n-1} = a_{n-1} + rb_n$.

Ensuite, pour tout k de $\{1, \ldots, n-2\}$: $b_k = a_k + rb_{k+1} - mb_{k+2}$.

Par identification de termes constants, on trouve enfin $b_0 = a_0 - mb_2$.

Posons plutôt $b'_0 = a_0 + rb_1 - mb_2$ pour rester dans la continuité des formules donnant les b_k.

On a alors $P(z) = (b_1x + b_0) + ib_1y = (b_1x + b'_0 - rb_1) + ib_1y = (b'_0 - b_1x) + ib_1y$. Voici donc la deuxième version de la fonction `evalhc`.

```
> evalhc2:=proc(P::array(numeric),x::numeric,y::numeric)
>   local r,m,b,c,k,t;
>   r:=x+x; m:=x*x+y*y; b:=0; c:=0;
>   for k to size(P) do t:=P[k]+r*b-m*c; c:=b; b:=t; od;
>   RETURN(array([b-c*x,c*y]))
> end:
```

On reprend l'exemple précédent, pour vérifier que les résultat est correct :

```
> P:=array([3,0,-5,2,1]): evalhc2(P,-11,8);
```

$$[-83487, -59296]$$

On note que cet algorithme nécessite $2n+3$ additions et $2n+4$ multiplications de réels, au lieu des $3n$ additions et $4n$ multiplications de la première version. [Q]

II. Division synthétique

Considérons le polynôme $P = \sum_{k=0}^{n} a_k X^k$, représenté par le tableau $P = [a_n, a_{n-1}, \ldots, a_1, a_0]$.

Soit $Q = \sum_{k=0}^{n-1} b_k X^k$ le quotient de P par $X - t$, représenté par $Q = [b_{n-1}, b_{n-2}, \ldots, b_1, b_0]$.

On a $P = (X-t)Q + P(t) = (X-t)\sum_{k=0}^{n-1} b_k X^k + P(t) = b_{n-1}X^n + \sum_{k=1}^{n-1}(b_{k-1} - b_k t)X^k + P(t) - b_0 t$.

Par identification, on trouve $b_{n-1} = a_n$ et : $\forall\, k \in \{1, .., n-1\},\ b_{k-1} = b_k t + a_k$.

Ainsi le calcul de $Q = [b_{n-1}, \ldots, b_1, b_0]$ s'effectue en posant virtuellement $b_n = 0$, puis :

$$b_{n-1} \leftarrow b_n t + a_n = a_n, \quad b_{n-2} \leftarrow b_{n-1}t + a_{n-1} = a_n t + a_{n-1},$$
$$b_{n-3} \leftarrow b_{n-2}t + a_{n-2} = a_n t^2 + a_{n-1}t + a_{n-2}, \ \ldots, \quad b_0 \leftarrow b_1 t + a_1$$

Ce sont les mêmes calculs que dans la question 1.a (algorithme de Horner itératif), mais il faut conserver les résultats intermédiaires plutôt que de se contenter du résultat final.

1. Pour remplir le tableau Q, on écrira donc successivement :

   $$Q[1] \leftarrow P[1],\ Q[2] \leftarrow Q[1]\,t + P[2],\ Q[3] \leftarrow Q[2]\,t + P[3],\ Q[n-1] \leftarrow Q[n-2]\,t + P[n-1]$$

   On en déduit la forme itérative pour la fonction `quo1` :

```
> quo1:=proc(P::array(numeric),t::numeric)
>   local n,Q,k;  n:=size(P);            # taille du tableau P
```