TE 6

Structures de contrôle en fortran et C

Éléments de correction

Références
- Polycopié FORTRAN : chap. **4** ; chap. 2, section 2 ; annexe C
- Polycopié C : chap. 6
- Transparents : cours 3

A Structures de contrôle

A.1 Structures conditionnelles

Exercice 1 : Affectation et test A

Compiler les programmes `affectation-test.c` [1] et `affectation-test.f90` suivants, Dans le cas où un exécutable est produit, tester le programme avec les valeurs $(2, 2)$, $(0, 0)$, $(2, 0)$ puis $(0, 2)$; expliquer. Corriger les programmes pour obtenir le comportement attendu.

affectation-test.c
```
#include <stdio.h>
#include <stdlib.h>
int main(void){
int i, j;
printf("saisir deux entiers\n");
scanf("%d %d", &i, &j);
printf("i=%d  j=%d\n", i, j);
if (i=j) {
  printf("i et j sont egaux : i=%d  j=%d\n", i, j);
}
exit(EXIT_SUCCESS);
}
```

affectation-test.f90
```
program affectation
implicit none

integer :: i, j
write(*,*) "saisir deux entiers"
read(*,*) i, j
write(*,*) "i=", i, " j=", j
if (i=j) then
  write(*,*) "i et j sont egaux, i=", i, " j=", j
end if

end program affectation
```

Solution

Il faut utiliser l'opérateur de comparaison `==` et non celui d'affectation. Le programme se compile correctement en C, et affecte à `i` la valeur de `j` et teste si le résultat est différent de zéro. Il est donc impératif de comprendre avec précision le message d'avertissement du compilateur : `parenthèses suggérées autour de l'affectation utilisée comme valeur de vérité` qui signale une affectation (`=`) utilisée comme booléen.
En fortran, le compilateur, même avec les options par défaut, détecte une erreur de syntaxe car l'expression après `if` doit être de type booléen. On doit donc corriger `=` en `==` pour produire un exécutable.

Exercice 2 : A

Écrire un programme qui saisit deux nombres **entiers** `x` et `y`, les affiche, puis les compare et affiche (selon les cas) un des trois messages suivants : `x` est plus grand que `y`, `x` est plus petit que `y` ou `x` est égal à `y`.

1. À l'aide de `if` sans `else`.

1. En C, compiler d'abord avec `gcc` seul, puis avec les options de l'alias `gcc-mni-c99` et noter l'intérêt des options.

2. À l'aide de structures de type `if... else ...` imbriquées.
3. B Écrire une deuxième version de ce programme avec une seule structure `if ...` comportant un `else if`.
4. B Dans quel langage peut-on aussi utiliser une structure d'énumération de cas (`case`) ?
5. Transformer le programme pour utiliser des réels (`float` ou `real`). Tester alors avec les valeurs `9.8765430` et `9.8765435`

Solution :

comparaison0.c

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

int main(void){
  int x, y;
  printf("entrer deux entiers\n");
  scanf("%d %d", &x, &y);
  printf(" x = %d , y = %d\n", x, y);
  if (x < y){
    printf(" x < y \n");
  }
  if (x > y){
    printf(" x > y \n");
  }
  if (x == y){
    printf(" x = y \n");
  }
  exit(EXIT_SUCCESS);
}
```

comparaison0.f90

```
PROGRAM comparaison
! deux if...endif imbriqués
IMPLICIT NONE
INTEGER :: x,y

WRITE(*,*) "Donner deux entiers x et y"
READ(*,*) x, y
WRITE(*,*) 'x=', x, 'et y=', y
IF (x > y) THEN
    WRITE(*,*) 'x > y'
ENDIF
IF (x < y) THEN
    WRITE(*,*) 'x < y'
ENDIF
IF (x == y) THEN
    WRITE(*,*) 'x=y'
ENDIF

END PROGRAM comparaison
```

Deux tests seulement sont nécessaires.

comparaison.c

```
#include <stdio.h>
#include <stdlib.h>
int main(void){
  int x, y;
  printf("entrer deux entiers\n");
  scanf("%d %d", &x, &y);
  printf(" x = %d , y = %d\n", x, y);
  if (x < y){
    printf(" x < y \n");
  }
  else{
    if (x > y){
      printf(" x > y \n");
    }
    else{
      printf(" x = y \n");
    }
  }
  exit(EXIT_SUCCESS);
}
```

comparaison0.f90

```
PROGRAM comparaison
! deux if...endif imbriqués
IMPLICIT NONE
INTEGER :: x,y

WRITE(*,*) "Donner deux entiers x et y"
READ(*,*) x, y
WRITE(*,*) 'x=', x, 'et y=', y

IF (x > y) THEN
    WRITE(*,*) 'x > y'
ELSE
    IF (x < y) THEN
        WRITE(*,*) 'x < y'
    ELSE
        WRITE(*,*) 'x=y'
    ENDIF
ENDIF

END PROGRAM comparaison
```

comparaison2.c

```c
#include <stdio.h>
#include <stdlib.h>
int main(void){
  int x, y;
  printf("entrer deux entiers\n");
  scanf("%d %d", &x, &y);
  printf(" x = %d , y = %d\n", x, y);
  if (x < y){
      printf(" x < y \n");
  }
  else if (x > y){ /* ne change rien */
  /* else porte tjs sur le dernier if*/
      printf(" x > y \n");
  }
  else{
      printf(" x = y \n");
  }
  exit(EXIT_SUCCESS);
}
```

comparaison2.f90

```
PROGRAM comparaison2
! introduction de elseif
! => un seul end if
IMPLICIT NONE
INTEGER :: x,y

WRITE(*,*) "Donner deux entiers x et y"
READ(*,*) x, y
WRITE(*,*) 'x=', x, 'et y=', y

IF (x > y) THEN
    WRITE(*,*) 'x > y'
ELSE IF (x < y) THEN
    WRITE(*,*) 'x < y'
ELSE
    WRITE(*,*) 'x=y'
END IF ! un seul après elseif

END PROGRAM comparaison2
```

En fortran, on peut utiliser des intervalles ouverts dans les `CASE` : ainsi en testant `x-y` que l'on peut comparer à une des constantes, on peut exprimer les différentes possibilités avec `SELECT CASE`
En C, il faudrait énumérer explicitement les valeurs possibles pour la différence...

comparaison3.f90

```
PROGRAM comparaison3
IMPLICIT NONE
INTEGER :: x,y

WRITE(*,*) "Donner deux entiers x et y"
READ(*,*) x, y
WRITE(*,*) 'x=', x, 'et y=', y

SELECT CASE (x-y)
CASE (1:)  ! 1, 2, 3, etc
    WRITE(*,*) 'x > y'
CASE (:-1) ! -1, -2, -3, etc
    WRITE(*,*) 'x < y'
CASE(0)
    WRITE(*,*) 'x=y'
END SELECT

END PROGRAM comparaison3
```

Enfin, si on transforme le code pour utiliser des types réels, on doit se souvenir que leur représentation en machine est approximative : sur 32 bits (`float` ou `REAL`), la précision relative de représentation est de $\varepsilon = 2^{-23} \approx 1,2\ 10^{-7}$, donc on ne peut pas distinguer les valeurs `9.8765430` et `9.8765435`.

Exercice 3 : Années bissextiles A

On rappelle qu'une année est bissextile :

– si elle est multiple de 4 mais n'est pas multiple de 100, ou
– si elle est multiple de 400.

1. Écrire un programme `bissextile.c` (`bissextile.f90`) qui détermine si une année est bissextile, en utilisant des tests imbriqués. Le programme prendra en charge la saisie de l'année.

   Solution

bissextile0.c

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
 int annee;
 printf ("Saisissez une année : ");
 scanf ("%d", &annee);
 if (annee%100 == 0) {
   if (annee%400 == 0){
      printf ("%d est bissextile\n", annee);
   }
   else{
      printf ("%d n'est pas bissextile\n", annee);
   }
 }
 else{
   if (annee%4 == 0){
      printf ("%d est bissextile\n", annee);
   }
   else{
      printf ("%d n'est pas bissextile\n", annee);
   }
 }
 exit (EXIT_SUCCESS);
}
```

bissextile0.f90

```
program bissextile
implicit none
 integer :: annee
 write(*,*) "Saisissez une année : "
 read(*, *) annee
 if (mod(annee,100)==0) then
   ! cas d'un siècle: il faut qu'il soit multiple de 4
   if (mod(annee,400)==0) then
      write(*,*) annee, " est bissextile"
   else
      write(*,*) annee, " n'est pas bissextile"
   endif
 else
   if(mod(annee,4)==0) then
   ! autre cas: il faut que l'annee soit multiple de 4
      write(*,*) annee, " est bissextile"
   else
      write(*,*) annee, " n'est pas bissextile"
   endif
 endif
end program bissextile
```

bissextile.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h> // C99 seulement

int main(void)
{
 int annee;
 bool bi; // en C89, utiliser un entier
 printf ("Saisissez une année : ");
 scanf ("%d", &annee);
 if (annee%100 == 0) {
    bi = (annee%400 == 0);
 }
 else{
    bi = (annee%4 == 0);
 }
 if(bi) {
   printf ("L'année %d est bissextile\n", annee);
 } else {
   printf ("L'année %d n'est pas bissextile\n", annee);
 }
 exit (EXIT_SUCCESS);
}
```

bissextile.f90

```
program bissextile
implicit none
 integer :: annee
 logical :: bissext

 write(*,*) "Saisissez une année : "
 read(*, *) annee
 if (mod(annee,100)==0) then
   ! cas d'un siècle: il faut qu'il soit multiple de 4
   bissext = (mod(annee,400)==0)
 else
   ! autre cas: il faut que l'anne soit multiple de 4
   bissext = (mod(annee,4)==0)
 endif
 if (bissext) then
   write(*,*) "L'année", annee, " est bissextile"
 else
   write(*,*) "L'année", annee, " n'est pas bissextile"
 end if
end program bissextile
```

2. En faire une version `bissextile2` où la condition est évaluée sous forme de booléen avec des opérateurs logiques.

Solution

bissextile2.c

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
 int an;

 printf ("Saisissez une année : ");
 scanf ("%d", &an);
 if (((an%4 == 0) && (an%100 != 0)) || (an%400 == 0)) {
    printf ("L'année %d est bissextile\n", an);
 } else {
    printf ("L'année %d n'est pas bissextile\n", an);
 }
 exit (EXIT_SUCCESS);
}
```

bissextile2.f90

```
program bissextile
implicit none
integer :: annee

write(*,*) "Saisissez une année : "
read(*, *) annee
if (((mod(annee,4) == 0) .and. (mod(annee,100) /= 0)) &
      .or. (mod(annee,400) == 0)) then
    write(*,*) "L'année", annee, " est bissextile"
else
    write(*,*) "L'année", annee, " n'est pas bissextile"
end if
end program bissextile
```

3. Copier l'un des programmes précédents pour construire un programme `liste_bissextile` affichant la liste des années bissextiles entre les années 1880 et 2020 en utilisant une structure itérative avec compteur.

   Solution

liste_bissextile.c

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
 int an;

 for (an=1880; an<=2020; an++) {
   if (((an%4 == 0) && (an%100 != 0)) || (an%400 == 0)) {
         printf ("L'année %d est bissextile\n", an);
   }
 }
 exit (EXIT_SUCCESS);
}
```

liste_bissextile.f90

```f90
program l_bissextile
implicit none
integer :: an
do an=1880, 2020, 1
 if (((mod(an,4) == 0) .and. (mod(an,100) /= 0)) &
      .or. (mod(an,400) == 0)) then
    write(*,*) "L'année", an, " est bissextile"
 end if
end do
end program l_bissextile
```

A.2 Structures itératives

Exercice 4 : Sommes A

1. À l'aide d'une structure itérative avec compteur, écrire un programme qui imprime la suite des entiers de 1 à 10. Le compléter progressivement pour qu'il affiche côte à côte la somme des entiers de 1 à 10, leurs carrés et la somme de ces carrés. Compléter le programme en affichant de plus les sommes calculées grâce aux expressions analytiques suivantes :

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \quad \text{et} \quad \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \tag{6.1}$$

   Comparer les résultats obtenus.

2. AB À l'aide d'une structure `while` en C ou `do while` en fortran, écrire un programme qui imprime la suite des entiers positifs, la somme de ces entiers, leurs carrés et la somme de ces carrés, en arrêtant le calcul dès que la somme des carrés dépasse 500.

Solution

Dans les expressions des sommes (6.1), on peut utiliser une division entière car le numérateur est divisible par le dénominateur. Mais il faut impérativement effectuer les multiplications avant la division, ce qui est assuré par l'évaluation de gauche à droite : `i*(i+2)/2` n'est pas égal à `i*((i+1)/2)`. Il est cependant préférable d'imposer cet ordre explicitement par des parenthèses (lignes 14 et suivante en C, lignes 12 et suivante en fortran).

1.

```c
#include <stdio.h>
#include <stdlib.h>

int main(void){
  int i, s, c, s2, sth, s2th;
  s = 0;
  s2 = 0;
  printf("entier, carré, somme "
   "des entiers, somme des carrés\n");
  for( i = 1; i<=10; i++){
    c = i * i;
    s += i;
    s2 += c;
    sth = (i*(i+1)) / 2; /**/
    s2th = (i*(i+1)*(2*i+1)) / 6;
    printf("%d %d %d %d %d %d\n",
           i, c, s, sth, s2, s2th);
  }
  exit(EXIT_SUCCESS);
}
```

```f90
PROGRAM sommes
IMPLICIT NONE
INTEGER:: i, s, c, s2, sth, s2th
s = 0
s2 = 0
WRITE(*,*) "entier, carré, &
   &somme des entiers, somme des carrés"
DO i = 1, 10
   c = i * i
   s = s + i
   s2 = s2 + c
   sth = (i*(i+1)) / 2 !
   s2th = (i*(i+1)*(2*i+1)) / 6
   WRITE(*,*) i, c, s, sth, s2, s2th
END DO
END PROGRAM sommes
```

2.

```c
#include <stdio.h>
#include <stdlib.h>

int main(void){
  int i, s, c, s2, sth, s2th;
  i = 1;
  s = 0;
  s2 = 0;
  printf("entier, carré, somme "
   "des entiers, somme des carrés\n");
   while( s2 < 500){
    c = i * i;
    s += i;
    s2 += c;
    sth = (i*(i+1)) / 2;
    s2th = (i*(i+1)*(2*i+1)) / 6;
    printf("%d %d %d %d %d %d\n",
           i, c, s, sth, s2, s2th);
    i++;
  }
  exit(EXIT_SUCCESS);
}
```

```
PROGRAM sommes2
IMPLICIT NONE
INTEGER:: i, s, c, s2, sth, s2th
i = 1
s = 0
s2 = 0
WRITE(*,*) "entier, carré, &
   &somme des entiers, somme des carrés"
DO  WHILE (s2 < 500)
   c = i * i
   s = s + i
   s2 = s2 + c
   sth = (i*(i+1)) / 2
   s2th = (i*(i+1)*(2*i+1)) / 6
   WRITE(*,*) i, c, s, sth, s2, s2th
   i = i + 1
END DO
END PROGRAM sommes2
```

Exercice 5 : Somme de la série alternée harmonique A

On se propose de calculer la somme de la série suivante :

$$1 - \frac{1}{2} + \frac{1}{3} + \cdots + \frac{(-1)^{n+1}}{n} + \cdots = \ln 2$$

Notations :

$$\text{somme partielle} \quad S_n = \sum_{i=1}^{n} u_i \qquad S = \lim_{n\to\infty} S_n = \sum_{i=1}^{\infty} u_i$$

$$\text{reste} \quad R_n = S - S_{n-1} = \sum_{i=n}^{\infty} u_i \qquad e_n = \frac{-R_n}{S} = \frac{S_{n-1} - S}{S}$$

On rappelle que, pour une série alternée, le reste R_n, caractérisant l'erreur de troncature de la série, peut être majoré en valeur absolue par le premier terme omis dans la somme finie : $|R_n| \leqslant |u_n|$.

Écrire et tester à chaque étape un programme `altern.f90` et un programme `altern.c` qui :

1. pour n fixé, calculent et affichent la somme finie S_n ainsi que l'écart relatif e_n avec la limite théorique ;
2. déterminent le nombre `nmax` de termes suffisant pour assurer une erreur relative de **troncature** majorée par `reltol`, valeur que l'on saisira au clavier dans le domaine $[10^{-4},\ 10^{-2}]$;
3. AB effectuent la sommation dans l'*ordre* qui limite les erreurs d'arrondi ; on testera l'effet du changement de l'ordre de sommation pour des précisions relatives jusqu'à 10^{-7}.

On demande de travailler en simple précision (`REAL` en fortran et en `float` en C).

Cet exercice pose le problème classique des deux types d'erreur en calcul numérique :

– **L'erreur de troncature déterministe** ou reste R_n : elle est due au fait que l'on ne somme qu'un **nombre fini de termes** pour évaluer la somme de la série. Elle intervient même si la précision de représentation des flottants est infinie. Dans le cas d'une série alternée, le premier terme non pris en compte donne un majorant de cette erreur. Cela constitue donc un critère d'arrêt de la boucle de calcul qui peut être évalué sans connaître la limite théorique (c'est en général cette méconnaissance qui nécessite le calcul de la série) : $|R_n| \leqslant |u_n| \leqslant \eta$.
Dans le cas de la série harmonique, ce critère permet de prévoir le nombre de termes avant de les calculer, ce qui autorise une boucle avec compteur. En effet $|u_n| = 1/n$ donc l'erreur de troncature est en $1/n$ (voir pente -1 en log-log sur la figure 6.1).

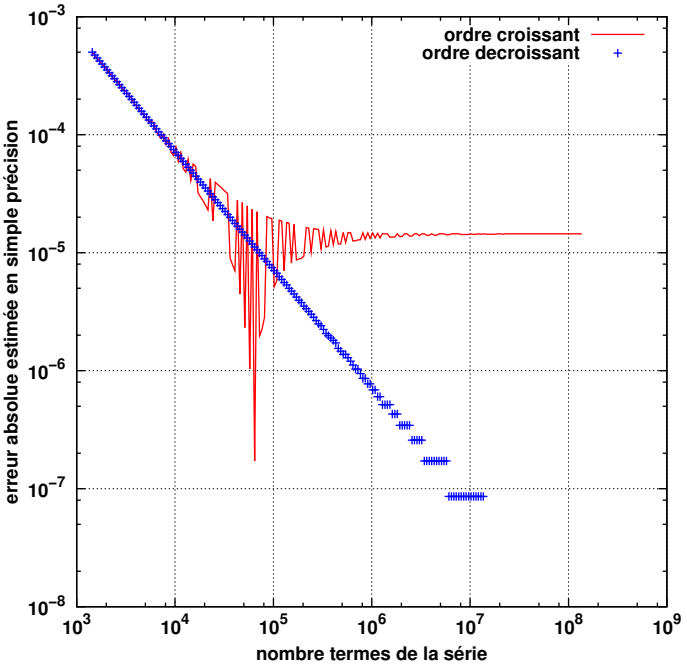

FIGURE 6.1 – La figure 6.1 montre que l'erreur reste supérieure à 10^{-5} si on effectue la sommation dans l'ordre croissant des indices. Dans ce cas, l'erreur totale est dominée par l'erreur d'arrondi dès que l'on atteint environ 100 000 termes ; il est alors inutile de sommer plus de termes. Remarquer l'aspect erratique de la courbe quand on approche de l'asymptote horizontale (le caractère bruité de l'erreur d'arrondi peut faire que pour un nombre particulier de points, ici, environ 70 000, l'erreur soit ponctuellement très faible, mais elle peut recroître si on augmente le nombre de termes). Si au contraire, on commence la sommation par les termes les plus petits, on prolonge, sur environ deux décades, la droite de pente -1 en log-log attendue quand l'erreur est dominée par la troncature (la valeur absolue du reste varie en $1/n$). Noter qu'il faudrait travailler avec une précision plus grande pour évaluer correctement l'erreur au delà de 10^{-7} ; d'ailleurs, on observe ici la quantification de l'erreur à `EPSILON(1.)*LOG(2.)` et ses multiples entiers.

– **L'erreur d'arrondi, de nature aléatoire**, qui dépend de la précision de représentation des réels en virgule flottante sur un **nombre fini de bits**. La précision relative des additions est donnée par ε. Dans le cas d'une série, si on ajoute des termes u_n très faibles devant la somme partielle, plus précisément si $|u_n|/|S_n| \leqslant \varepsilon$, la somme ne progresse plus. Mais, bien avant ce «blocage», la précision finie des additions provoque une accumulation d'erreurs d'arrondi qui limite la précision sur la somme de façon dramatique si effectue la somme dans le sens des indices croissants.
  De ce point de vue, il vaut mieux programmer la sommation en commençant par les termes les plus faibles, ce qui ne pose ici aucun problème pratique ici car le terme général est donné par une expression analytique (et non déterminé par récurrence). Dans le cas de la série harmonique, on gagne au moins deux décades en effectuant la somme selon les indices décroissants.

Solution

Les deux ordres sont mentionnés dans les programmes (l'ordre déconseillé est commenté).

```
1  program altern
2  ! sommation de la série alternée
3  ! ---       n
4  ! \     (-1)
5  ! /     ----    =   Ln(2)
```

```
! ---     n
implicit none
real            :: reltol
real            :: somme
real            :: somme_finie = 0.
integer         :: nmax
integer         :: i
!
somme = log(2.)
write(*,*) 'entrer la précision relative'
read(*,*) reltol
nmax = INT(1. / (somme * reltol)) - 1
write(*,*) ' nombre de termes suffisant ', nmax
!-- ordre à éviter pour les erreurs d'arrondi ---
!  do i = 1, nmax, 1                           |
!-- commencer par les plus petits ---------------
do i =  nmax, 1, -1
  somme_finie = somme_finie + (-1)**(i+1) / real(i)
end do
write(*,*) ' somme approchée ', somme_finie
write(*,*) ' somme  ', somme
write(*,*) ' erreur relative', (somme_finie - somme )/somme
end program altern
```

```
/*
 sommation de la série alternée
 ---       n
 \     (-1)
 /     ----    =   Ln(2)
 ---     n
*/
#include <stdio.h>
#include <stdlib.h>
#include <tgmath.h> // c99
int main(void){
float reltol;
float somme;
float somme_finie = 0.;
int nmax;
int i;

somme = log(2.f);
printf("entrer la précision relative\n");
scanf("%g", &reltol);
nmax = (int)(1.f / (somme * reltol)) - 1;
printf("nombre de termes suffisant %d\n", nmax);
/*
-- ordre à éviter pour les erreurs d'arrondi ---
 for(i=1; i<=nmax; i++)  {
-- commencer par les plus petits ---------------
*/
for(i=nmax; i>=1; i--){
  /* ne pas appeler pow pour alterner les signes ! */
  somme_finie = somme_finie + (-1+2*(i%2)) / (float)i ;
}
printf("somme approchée %f\n", somme_finie);
```

```
printf("somme  %f\n", somme);
printf(" erreur relative %g\n", (somme_finie - somme )/somme);
exit(EXIT_SUCCESS);
}
```

En commençant par les plus petits termes et en alternant les signes des termes de façon plus efficace.

```
program altern
! sommation de la série harmonique alternée
implicit none
real            :: reltol
integer         :: signe
real            :: ui
real            :: somme, somme_finie
integer         :: i, nmax
!
somme = log(2.)
write(*,*) 'entrer la précision relative'
read(*,*) reltol
nmax = INT(1. / (somme * reltol)) - 1
write(*,*) ' nombre de termes suffisant ', nmax
signe = (-1)**(nmax+1)
ui = real(signe) / real(nmax)
somme_finie = ui
! commencer par les plus petits
do i =  nmax-1, 1, -1
  signe = -signe
  ui = signe / real(i)
  somme_finie = somme_finie + ui
end do
write(*,*) 'somme approchée ', somme_finie
write(*,*) 'somme           ', somme
write(*,*) 'erreur relative', (somme_finie - somme )/somme
end program altern
```

```
/* sommation de la série harmonique alternée */
#include <stdio.h>
#include <stdlib.h>
#include <tgmath.h> // c99

int main(void){
float reltol;
float ui;
int signe;
float somme, somme_finie;
int i, nmax;

somme = log(2.f);
printf("entrer la précision relative\n");
scanf("%g", &reltol);
nmax = (int)(1.f / (somme * reltol)) - 1;
printf("nombre de termes suffisant %d\n", nmax);
// commencer par les plus petits
```

```
signe = -1+2*(nmax%2); // -1 si pair ou +1 si impair
ui = (float)signe /(float) nmax;
somme_finie = ui;
for(i=nmax-1; i>=1; i--){
  /* ne pas appeler pow pour alterner les signes ! */
  signe = -signe;
  ui = (float)signe /(float) i;
  somme_finie = somme_finie + ui;
}
printf("somme approchée %10.8f\n", somme_finie);
printf("somme           %10.8f\n", somme);
printf(" erreur relative %g\n", (somme_finie - somme )/somme);
exit(EXIT_SUCCESS);
}
```

Calcul avec différents nombres de termes pour les courbes

```
! version avec appels multiples pour tester la précision
program altern
! sommation de la série alternée
! ---       n
! \     (-1)
! /     ----    =   Ln(2)
! ---     n
implicit none
real            :: reltol
real            :: somme
real            :: somme_finie = 0.
integer         :: nmax
integer         :: i
integer         :: k,kmax=20
!
somme = log(2.)
!write(*,'(a)') '# ordre optimal '
write(*,'(a)') '# ordre non optimal '
write(*,'(a)') '#  nmax       somme   somme appr.  &
              &  | err. rel. demandée erreur rel. réelle'
do k=1, kmax
   somme_finie = 0.
   reltol = 10**(-(k-1)/4. -3.)
   nmax = int(1. / (somme * reltol)) - 1
   !-- ordre à éviter pour les erreurs d'arrondi ---
   do i = 1, nmax, 1                            !
   !-- commencer par les plus petits --------------
   ! do i = nmax, 1, -1
     somme_finie = somme_finie + (-1)**(i+1) / real(i)
   end do
   write(*,'(i11,2x,f11.9,2x,f11.9,2x,es15.8,2x,es15.8)') &
         nmax, somme, somme_finie, reltol, (somme_finie - somme )/somme
end do
end program altern
```

```
/*
 version avec appels multiples pour tester la précision
 sommation de la série alternée
```

```
 ---       n
 \     (-1)
 /     ----    =   Ln(2)
 ---     n
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(void){
float reltol;
float somme;
float somme_finie = 0.;
int nmax;
int i, k;
/* int kmax=20; */ /* peu de points mais rapide */
int kmax=200;  /* bcp de points => bruit mais lent */

somme = log(2.) ;
/* printf("# log(2) = %s\n", M_LN21); */ /* dans math.h */
/*
printf("# ordre non optimal\n");
 */
printf("# ordre optimal \n");
printf("#    nmax       somme    somme appr. "
              "| err. rel. demandée erreur rel. réelle\n");
for( k=1; k<=kmax; k++){
   somme_finie = 0.;
   reltol = pow(10., (double)((-(k-1))/40. -3.)) ;
   nmax = (int) (1. / (somme * reltol)) - 1 ;
   /*  ordre à éviter pour les erreurs d'arrondi ---*/
   /*
   for( i = 1; i <=nmax; i++){
   */
   /* -- commencer par les plus petits -------------*/
   for( i = nmax; i>=1; i--){
       somme_finie = somme_finie + (-1+2*(i%2)) / (float)(i);
   }
   printf("%11d  %11.9f  %11.9f  %15.8e  %15.8e\n",
    nmax, somme, somme_finie, reltol, (somme_finie - somme )/somme);
}
   exit(EXIT_SUCCESS) ;
}
```

Exercice 6 : Recherche des nombres premiers AB

1. Écrire un programme qui détermine et affiche si un nombre entier fourni par l'utilisateur est premier ; le tester.
   Puis en faire une deuxième version qui optimise la recherche des diviseurs entiers (quel pas choisir et où s'arrêter ?).
2. Transformer ce programme pour qu'il affiche et compte tous les nombres premiers inférieurs à un entier fourni par l'utilisateur (on pourra comparer la durée d'exécution suivant l'optimisation avec la commande `time` à condition de rediriger les entrées/sorties).

```fortran
program premiers

implicit none

integer :: n, n_prem, i, div
logical :: premier

write (*,*) "Entrer un entier positif"
read (*,*) n

n_prem = 1
write(*,*) 1
n_prem = n_prem + 1
write(*,*) 2
do i=3, n, 2
   premier = .true.
   do div=2, int(sqrt(real(i)))
      if (mod(i, div) == 0) then
         premier = .false.
         exit
      endif
   enddo
   if (premier) then
      write (*,*) i
      n_prem = n_prem + 1
   endif
enddo

write (*,*) "Il y a ",n_prem," nombres premiers <= ", n

end program premiers
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <tgmath.h>
#include <stdbool.h>

int main(void) {

int n, n_prem;
bool premier;
printf("Entrer n\n");
scanf("%d", &n);
printf("nb premiers <= %d\n", n) ;

n_prem = 1;
printf("1\n") ;
n_prem++ ;
printf("2 \n") ;
for (int i=3; i <= n; i++) {
   premier = true;
   for (int diviseur =2; diviseur  <= (int)sqrt((float)i); diviseur++) {
      if (i % diviseur == 0) {
         premier = false;
         break;
```

```
24        }
25     }
26     if (premier == true) {
27        printf("%d\n", i);
28        n_prem++;
29     }
30  }
31  printf("Il y a %d nombres premiers <= %d\n", n_prem, n);
32
33  exit(EXIT_SUCCESS);
34  }
35
```

Exercice 7 : Précision des flottants : recherche de ε AB

Les réels représentés en virgule flottante sont codés sur un nombre fixe de bits répartis entre :
– signe,
– mantisse (qui détermine la précision),
– exposant (qui détermine le domaine).
Les réels sont donc représentés approximativement et dans un intervalle fini par un ensemble discret et fini de valeurs. Dans chaque octave $[2^n, 2^{n+1}]$, l'exposant est fixe le nombre de flottants est donné par le nombre de combinaisons possibles entre les bits de la mantisse. Pour des flottants sur 32 bits, 23 bits sont consacrés à la mantisse et les 2^{23} réels de l'octave $[2^n, 2^{n+1}]$ sont en progression arithmétique de raison $2^n\varepsilon$ où $\varepsilon = 2^{-23}$. En particulier, pour $n = 0$, la valeur `1.` est représentée exactement, et son successeur est $1 + \varepsilon$. Mais son prédécesseur, dans l'octave $[1/2, 1]$, est $1 - \varepsilon/2$. L'écart relatif entre deux réels consécutifs est donc compris entre $\varepsilon/2$ et ε. La précision relative de représentation des réels est donc majorée par ε.

epsilon.c

```
#include <stdio.h>
#include <stdlib.h>
// definition des caracteristiques des flottants
#include <float.h>
int main(void){
  float eps;
  eps = FLT_EPSILON;
  printf("FLT_EPSILON  %g\n", eps);
  exit(EXIT_SUCCESS);
}
```

epsilon.f90

```
program real_eps
implicit none
real :: eps
eps = epsilon(1.)
write(*,*) "epsilon(1.) = ", eps
end program real_eps
```

1. Exécuter les programmes `epsilon.c` (`epsilon.f90`) afin d'afficher la valeur de ε pour le type `float` en C et pour le type `REAL` en fortran respectivement. Que se passe-t-il si on imprime cette valeur en format `%f` en C ou en format `f8.6` en fortran ?
2. Justifier les valeurs affichées sachant que pour les flottants sur 32 bits, 23 bits sont consacrés à la mantisse, 8 à l'exposant et 1 au signe.
3. B On se propose dans la suite de rechercher un encadrement de ε par une méthode de dichotomie. On considère des intervalles $[f_1, f_2]$ de borne inférieure fixe $f_1 = 1.$, et de largeur $\delta = f_2 - f_1$ variable. La borne inférieure, f_1, est représentée exactement mais la borne supérieure, f_2, sera arrondie au flottant le plus proche. On part de $\delta = 1.$ et on divise δ par 2 à chaque étape de l'itération jusqu'à ce que la représentation approximative en flottant de la borne supérieure f_2 soient confondue avec f_1.

   Compléter le code fourni dans `rech-epsilon` pour mettre en œuvre l'itération [2] et afficher la largeur de l'intervalle. Comparer avec ε affiché précédemment. Expliquer à l'aide d'un schéma figurant $f_1 = 1.$ et son successeur, ainsi que f_2 et son approximation en flottant pour les dernières étapes de la dichotomie. On pourra essayer des divisions par des valeurs inférieures à 2 pour affiner l'encadrement de ε ; la recherche sera alors plus longue.

2. En C, on veillera à n'utiliser que des constantes de type `float` pour éviter des opérations en `double`.

rech-epsilon.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <float.h> // necessaire pour FLT_EPSILON
int main(void){
  float eps;
  float f1, f2, delta;
  eps = FLT_EPSILON;
  printf("FLT_EPSILON  %g\n", eps);
  // recherche de epsilon float par dichotomie
  // initialisation
  delta = 1.f;
  f1 = 1.f;
  f2 = f1 + delta;

  // a completer par la boucle de dichotomie

  exit(EXIT_SUCCESS);
}
```

rech-epsilon.f90

```
program real_eps
! recherche de epsilon real
implicit none
real :: eps
real :: f1, f2, delta
eps = epsilon(1.)
write(*,*) "epsilon(1.) = ", eps
! initialisation
delta = 1.
f1 = 1.
f2 = f1 + delta

! a completer par la boucle de dichotomie

end program real_eps
```

Solution

En flottant, le successeur (voisin de droite) de 1. est $1+\varepsilon$. Toute valeur comprise entre 1 et $1+\varepsilon/2$ est donc arrondie à 1 dans la représentation en flottant. A force de rapprocher `f2` de `f1=1`, on finit par passer à gauche de $1+\varepsilon/2$ et `f2` est arrondi à 1., ce qui le rend égal à `f1`. Si on divise par 2 à chaque itération, on détermine exactement $\varepsilon/2$ et f_2 reste représenté exactement sauf sa dernière valeur. Mais si le coefficient est différent, f_2 est arrondi et l'arrêt de la boucle ne donne pas exactement $\varepsilon/2$. Consulter l'annexe du polycopié fortran sur la représentation des flottants pour plus de détails. D'autre part, si on partait de `f2` inférieur à 1. le résultat serait différent : en effet le prédécesseur de `1.` est $1-\varepsilon/2$, car on change d'exposant à gauche de 1...

rech-epsilon.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <float.h> // necessaire pour FLT_EPSILON
int main(void){
  float eps;
  float f1, f2, delta;
  eps = FLT_EPSILON;
  printf("FLT_EPSILON  %g\n", eps);
  // recherche de epsilon float par dichotomie
  delta = 1.f;
  f1 = 1.f;
  f2 = f1 + delta;
  while (f1 != f2) {
    delta = delta/2.f;
    f2 = f1 + delta;
  }
  printf(" delta/FLT_EPSILON = %g\n", delta/eps);
  exit(EXIT_SUCCESS);
}
```

rech-epsilon.f90

```
program real_eps
! recherche de epsilon real
implicit none
real :: eps
real :: f1, f2, delta
eps = epsilon(1.)
write(*,*) "epsilon(1.) = ", eps
! initialisation
delta = 1.
f1 = 1.
f2 = f1 + delta
do while (f1 /= f2)
  delta = delta/2.
  f2 = f1 + delta
end do
write(*,*) "delta/epsilon = ", delta/epsilon(1.)
! testé avec g95 / gfortran 4.1 / nagfor
! mais  -fltconsistency nécessaire avec ifort
end program real_eps
```

B Compléments (facultatifs)

Exercice 8 : Nombre d'or AB

L'objectif est d'écrire un programme calculant le Nombre d'Or. Celui-ci peut être obtenu à partir de la suite de Fibonnacci u_n définie par la récurrence suivante :

$$u_{n+1} = u_n + u_{n-1} \quad \text{avec} \quad u_0 = 1 \quad \text{et} \quad u_1 = 1$$

La suite des rapports $v_n = \left(u_{n+1}/u_n\right)$ converge vers le Nombre d'Or de valeur théorique $\frac{1+\sqrt{5}}{2}$.

1. Quelle structure de contrôle permet de piloter le calcul des termes de la suite jusqu'à ce que le rapport v_n de deux termes consécutifs converge à la précision relative η près ?
2. Écrire le programme de calcul des suites u_n et v_n.
3. Tester le programme avec $\eta = 10^{-4}$ en `real` en fortran et en `float` en C. Combien d'itérations sont nécessaires ? Comparez la valeur trouvée à $\frac{1+\sqrt{5}}{2}$ calculée dans le même type.

4. Le reprendre avec $\eta = 10^{-6}$. Combien d'itérations sont nécessaires ?

5. B Que doit-on modifier pour atteindre la précision de 10^{-10} ? Quel est alors le nombre d'itérations nécessaires ? Reprendre la comparaison théorique avec attention.

Solution

La structure de contrôle est de type `while`, mais il est prudent d'insérer un compteur pour surveiller le nombre d'itérations nécessaires. Dans le cas où la convergence est mal connue, on prévoit de plus un arrêt exceptionnel du programme (via `STOP` en fortran ou `exit` en C), ou au moins une sortie de boucle (via `EXIT` en fortran ou `break` en C) quand le compteur atteint un nombre trop important d'itérations.

Éviter de nommer `EPSILON` le paramètre η car `EPSILON` est le nom d'une fonction intrinsèque du fortran (rend le plus grand flottant positif du type de son argument qui, ajouté à 1, redonne 1). En C, lui correspondent 2 constantes `FLT_EPSILON` et `DBL_EPSILON` définies au niveau préprocesseur dans `float.h` Tant que la précision demandée n'est pas limitée par la précision du type représentant les nombres manipulés, il suffit d'itérer pour converger (pas de problème d'algorithme ici). Mais il serait illusoire de rechercher une précision meilleure que celle donnée par `EPSILON(u_cour)` en fortran ou `FLT_EPSILON` en C en `float` ou `DBL_EPSILON` en C en `double`.

En C, utiliser la fonction `fabs` et non `abs` pour le calcul de l'écart relatif entre les deux itérations, car `abs` est une fonction à argument et résultats entiers ! Elle arrêterait la boucle dès que la valeur absolue de l'écart relatif est inférieure à 1/2.

En C89 les fonctions réelles sont par défaut en `double`, ce qui impose des conversions quand on travaille en `float`. En C99, il existe des versions `fabsf` de type `float` et `fabsl` de type `long double`, mais il est plus simple de travailler alors avec `#include <tgmath.h>` qui assure la généricité des fonctions mathématiques entre différents types de réels. On écrit alors toujours `fabs` et le compilateur appelle la fonction correspondant au type de l'argument. Attention, ne pas croire que l'on pourrait utiliser `abs` : la généricité a ici un sens plus faible qu'en fortran.

Solutions en Fortran

```
PROGRAM or
IMPLICIT NONE
REAL, PARAMETER :: eps = 1.e-6
REAL :: err
REAL :: u_prec,u_cour
REAL :: v_prec,v_cour
REAL :: somme
REAL :: nombre_or
INTEGER :: i

nombre_or = (1.+SQRT(5.))/2.
u_prec = 1.
u_cour = 1.
err = 10.
i = 0
WRITE(*,*) "precision limitée à ", EPSILON(u_cour)
DO WHILE (err > eps)
    v_prec = u_cour / u_prec
    somme = u_cour + u_prec
    u_prec = u_cour
    u_cour = somme
    v_cour = u_cour / u_prec
    err = ABS((v_cour - v_prec) / v_prec)
    i = i + 1
END DO
WRITE(*,*) "nb d'itérations ", i
WRITE(*,*) "écart relatif ", err
WRITE(*,*)"Limite de la suite (vn): ",v_cour, &
      "Nombre d'or: ", nombre_or
END PROGRAM or
```

```
PROGRAM or
IMPLICIT NONE
INTEGER, PARAMETER :: dp = SELECTED_REAL_KIND(p=14)
! la variante de type dp demandée
! permet une précision de 10^{-14} au moins
REAL(kind=dp), PARAMETER :: eps = 1.e-10
REAL(kind=dp) :: err
REAL(kind=dp) :: u_prec,u_cour
REAL(kind=dp) :: v_prec,v_cour
REAL(kind=dp) :: somme
REAL(kind=dp) :: nombre_or
INTEGER       :: i
nombre_or = (1._dp+SQRT(5._dp))/2._dp
! constantes dans la variante dp du type REAL
u_prec = 1.
u_cour = 1.
err = 10.
i = 0
WRITE(*,*) "precision limitée à ", EPSILON(u_cour)
DO WHILE (err > eps)
    v_prec = u_cour / u_prec
    somme = u_cour + u_prec
    u_prec = u_cour
    u_cour = somme
    v_cour = u_cour / u_prec
    err = ABS((v_cour - v_prec) / v_prec)
    i = i + 1
END DO
WRITE(*,*) "nb d'itérations ", i
WRITE(*,*) "écart relatif ", err
WRITE(*,*) "Limite de la suite (vn): ", v_cour, &
       "Nombre d'or: ", nombre_or
END PROGRAM or
```

Solutions en C

```c
#include <stdio.h>
#include <stdlib.h>
#include <tgmath.h> // C99 => -lm au lien
#include <float.h> /* pour FLT_EPSILON */

/* version  en float */

int main(void){
  float eps = 1.e-6f; /* constante de type float */
  float err;
  float u_prec, u_cour;
  float v_prec, v_cour;
  float somme;
  float nombre_or;
  int i;
  nombre_or = (1.f+sqrt(5.f))/2.f; /* float */
  u_prec = 1.f;
  u_cour = 1.f;
  err = 10.f ;
  i = 0 ;
  printf(" précision limitée à %g\n", FLT_EPSILON);
  while(err > eps) {
      v_prec = u_cour/u_prec;
      somme  = u_cour + u_prec;
      u_prec = u_cour;
      u_cour = somme;
      v_cour = u_cour/u_prec;
      /* attention : abs attend et rend un entier ! */
      /* l'employer ici arrêterait la boucle trop tôt */
      err = fabs((v_cour - v_prec)/v_prec);
      i++;
  }
  printf("Limite de la suite (vn) : %g\n", v_cour);
  printf("nb itérations : %d\n", i);
  printf("écart relatif : %g\n", err);
  printf("Nombre d'or : %g\n", nombre_or);
  exit(EXIT_SUCCESS);
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <tgmath.h> // C99 => -lm au lien
#include <float.h> /* pour DBL_EPSILON */

/* version  en double */

int main(void){
  double eps = 1.e-10;
  double err;
  double u_prec, u_cour;
  double v_prec, v_cour;
  double somme;
  double nombre_or;
  int i;
  nombre_or = (1.+sqrt(5.))/2.;         /* double */
  /* nombre_or = (1.f+sqrt(5.f))/2.f; */ /* float */
  u_prec = 1.;
  u_cour = 1.;
  err = 10. ;
  i = 0 ;
  printf(" précision limitée à %g\n", DBL_EPSILON);
  while(err > eps) {
      v_prec = u_cour/u_prec;
      somme  = u_cour + u_prec;
      u_prec = u_cour;
      u_cour = somme;
      v_cour = u_cour/u_prec;
      err = fabs((v_cour - v_prec)/v_prec);
      i++;
  }
  printf("Limite de la suite (vn) : %g\n", v_cour);
  printf("nb itérations : %d\n", i);
  printf("écart relatif : %g\n", err);
  printf("Nombre d'or : %f\n", nombre_or);
  exit(EXIT_SUCCESS);
}
```

Exercice 9 : Somme de la série entière de argsh(x) B

La fonction argument sinus hyperbolique admet le développement en série entière suivant lorsque $|x| \leqslant 1$:

$$\begin{aligned} \operatorname{argsh} x &= x - \frac{1}{2\times 3}\, x^3 + \frac{1\times 3}{2\times 4\times 5}\, x^5 - \frac{1\times 3\times 5}{2\times 4\times 6\times 7}\, x^7 + \cdots = \sum_{n=0}^{\infty} a_n(x) \\ \text{où} \quad a_n(x) &= \frac{(-1)^n}{2n+1}\, \frac{(2n)!}{2^{2n}(n!)^2}\, x^{2n+1} \end{aligned} \tag{6.2}$$

1. Indiquer pourquoi programmer le calcul de cette série en calculant le terme général $a_n(x)$ tel qu'il est écrit ci-dessus n'est ni numériquement sûr, ni numériquement efficace.
2. En calculant $\dfrac{a_{n+1}(x)}{a_n(x)}$, montrer qu'il existe une relation de récurrence liant $a_{n+1}(x)$ à $a_n(x)$.
3. Programmer le calcul de la série en utilisant cette récurrence. Laisser le choix de x à l'utilisateur et calculer par défaut les 100 premiers termes de la série. Comparer à la fin le résultat obtenu avec celui fourni par la fonction `asinh` en C, et en utilisant le fait que $\operatorname{argsh}(x) = \ln\left(x + \sqrt{x^2+1}\right)$ en fortran [3].
4. En fait, il ne sert à rien de rajouter des termes dans la somme dès que le rapport entre le terme à ajouter et la somme partielle devient inférieur en valeur absolue à ϵ, donné en fortran par la fonction intrinsèque `EPSILON(1.)` et en C par la constante `FLT_EPSILON` définie dans le fichier d'entête `<float.h>` Modifier le programme pour arrêter la sommation dès que ce rapport est atteint.

3. La fonction intrinsèque `asinh`, extension de type `gnu` sous `gfortran` norme 2003, est intégrée à la norme du fortran 2008.

On demande de travailler en simple précision (`REAL` en fortran et en `float` en C).

Solution

Évaluer des factorielles en type entier standard (sur 32 bits) expose à un dépassement de capacité dès 13!. De toute façon, il n'est pas possible d'effectuer le calcul du coefficient de a_n en division entière. Tenir compte des simplifications par les entiers impairs serait la moindre des précautions. En pratique, il est beaucoup plus rapide de calculer les $a_n(x)$ de proche en proche en tenant compte du rapport :

$$\frac{a_{n+1}}{a_n} = -\frac{(2n+1)^2}{(2n+2)(2n+3)}x^2$$

On peut remarquer que la série est alternée pour $|x| \leqslant 1$, puisque $\left|\frac{a_{n+1}(x)}{a_n(x)}\right| < 1$.

```
1  program argsh
2
3  implicit none
4
5  real :: x, s, u, x2, d, ri
6  integer :: i
7
8  write (*,*) "Entrer x"
9  read (*,*) x
10
11 x2 = x*x ! pour éviter de le calculer dans la boucle
12 u = x
13 s = 0.
14 do i=0, 99
15    ri = real(i) ! conversion explicite
16    s = s + u
17    u = -u * (2.*ri+1.)**2 / (2.*ri+2.) / (2.*ri+3.) * x2
18    !u = -u * (2.*i+1.)**2 / (2.*i+2.) / (2.*i+3.) * x2
19    if (abs(u/s) < EPSILON(1.)) then
20        write (*,*) "sortie de la boucle pour i = ", i
21        exit
22    endif
23 enddo
24
25 d = s - log(x + sqrt(x**2 + 1))
26 write (*,*) "serie:", s, " fonction: ", log(x + sqrt(x**2 + 1)), "ecart:", d
27
28 end program argsh
```

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <tgmath.h>
4  #include <float.h>
5
6  int main(void) {
7  float x, s, u, x2, d;
8
9  printf("Entrer x\n");
10 scanf("%g", &x);
11
12 // ordre naturel : on part de i=0
13 x2 = x*x; // pour éviter de le calculer dans la boucle
```

```
14 u=x;
15 s=0.;
16 for (int i=0; i<100; i++) {
17    s += u;
18    u *= -(2.f*i+1.f)*(2.f*i+1.f)/(2.f*i+2.f)/(2.f*i+3.f)*x2;
19    if (fabs(u/s) < FLT_EPSILON) {
20       printf("sortie de la boucle => %d\n", i);
21       break;
22    }
23 }
24 d = s - asinh(x);
25 printf("serie : %g fct : %g ecart %g\n", s, asinh(x), d);
26
27 exit(EXIT_SUCCESS);
28 }
```

Compléments

- Si x est petit, la convergence de la série est rapide à cause du facteur x^{2n+1} : le nombre de termes nécessaires est faible et l'ordre de sommation n'est pas critique.
- Au contraire, quand $|x| \to 1$, la convergence est lente et il est alors prudent de sommer en commençant par les termes les plus petits pour éviter l'accumulation d'erreurs d'arrondi.

Par ailleurs, comme le développement limité est un polynôme, il est plus précis et plus rapide de l'évaluer selon l'**algorithme de Horner**, rappelé en (6.3).

$$\begin{aligned} P_n(x) &= c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n \\ &= c_0 + x\left(c_1 + x\left(c_2 + \cdots + x(c_{n-2} + x(c_{n-1} + x c_n))\cdots\right)\right) \\ &= \left(\left(\left(\cdots(c_n x + c_{n-1})\, x + c_{n-2}\right) x + \cdots\right)\, x + c_1\right)\, x + c_0 \end{aligned} \tag{6.3}$$

Cette méthode impose de commencer le calcul par le monôme de degré le plus élevé. On peut procéder en deux étapes :

- on calcule tout d'abord les coefficients par récurrence montante mais on les stocke dans un tableau ;
- puis on applique l'algorithme de Horner, rappelé en (6.3), avec les coefficients déjà calculés (en redescendant les indices) pour évaluer le polynôme.

On peut enfin éviter l'emploi de tableaux grâce à une modification du schéma de Horner pour ne faire intervenir que les rapports de deux coefficients consécutifs.

$$\begin{aligned} P_n(x) &= c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n \\ &= c_0 + x c_1/c_0\left(1 + x c_2/c_1\left(1 + \cdots + x c_{n-2}/c_{n-3}(1 + x c_{n-1}/c_{n-2}(1 + x c_n/c_{n-1}))\cdots\right)\right) \end{aligned}$$

Cette formulation, qui ne nécessite qu'une boucle, permet de commencer la sommation par l'indice le plus élevé, donc par les termes les plus petits, ce qui limite l'erreur d'arrondi. Mais, comme on ne calcule pas explicitement chaque terme de la somme, on ne peut pas déterminer simplement le nombre de termes nécessaires à une précision donnée par majoration du reste (comme on le fait pour la série harmonique alternée).