**TE 7**

# Procédures en fortran et C

## Éléments de correction

# A Notions fondamentales

### Exercice 1 : Introduction aux pointeurs A

**Notions de base sur les pointeurs**

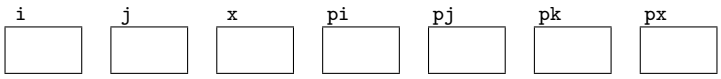
exo1-pointeurs.c

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

#define ULI (unsigned long int) // conversion

int main(void) {
 int i=1, j=2;
 int *pi=NULL, *pj=NULL, *pk;
 float x, *px=NULL;

 pi = &i;
 pj = &j;
 printf("i=%d, j=%d, *pi=%d, *pj=%d\n",
        i, j, *pi, *pj);
 printf("pi=%lu , pj=%lu\n", ULI pi, ULI pj);
 exit(EXIT_SUCCESS); // fin etape 1

 i *= 10;
 printf("i=%d, *pi=%d, pi=%lu\n", i, *pi, ULI pi);
 *pi *= 10;
 printf("i=%d, *pi=%d, pi=%lu\n", i, *pi, ULI pi);
 exit(EXIT_SUCCESS); // fin etape 2

 printf("pk=%lu\n", ULI pk);
 pk = pj; // memes types
 printf("*pk=%d, pk=%lu\n", *pk, ULI pk);
 if (pk == &j) {
    printf("pk pointe vers j\n");
 }
 exit(EXIT_SUCCESS); // fin etape 3

 *pk = *pi;
 printf("i=%d, *pk=%d\n", i, *pk);
 exit(EXIT_SUCCESS); // fin etape 4

 x = 3.5f;
 px = &x;
 printf("x=%g , *px=%g, px=%lu\n", x, *px, ULI px);
 *px = *pj;
 printf("j=%d , *px=%g, px=%lu\n", j, *px, ULI px);
 exit(EXIT_SUCCESS); // fin etape 5
}
```

exo1-pointeurs.f90

```f90
program pointeur

implicit none

integer, target  :: i=1, j=2
integer, pointer :: pi=>null(), pj=>null(), pk
real, target     :: x
real, pointer    :: px=>null()

pi => i
pj => j
write (*,*) "i =", i, ", j =", j
write (*,*) "pi =", pi, ", pj =", pj
write (*, *) "pi => i :", associated(pi, i)
stop ! fin etape 1

i = i*10
write (*,*) "i =",i,", pi =",pi
pi = pi*10
write (*,*) "i =",i,", pi =",pi
stop ! fin etape 2


pk => pj
if (associated(pk, j)) then
  write (*,*) "pk pointe vers j"
  write (*,*) "pk =", pk, "j =", j
endif
stop ! fin etape 3

pk = pi ! memes types
write (*,*) "i =", i, " pk =", pk
stop ! fin etape 4

x = 3.5
px => x
write (*,*) "x =", x, "px =", px
px = pj
write (*,*) "j =", j, " x =", x, "px =", px
stop ! fin etape 5

end program pointeur
```

Le code **fourni** dans les fichiers `exo1-pointeurs.c` et `exo1-pointeurs.f90` permet d'expérimenter les opérations sur les pointeurs en dehors de tout passage de paramètre dans des procédures. Exécuter ce code étape par étape en passant en commentaire au fur et à mesure les instructions d'arrêt (`exit` en C et `stop` en fortran). À chaque étape du programme, représenter schématiquement les zones mémoire des différentes variables et noter leur valeur ; préciser vers quelle zone pointent les variables pointeurs ainsi que les valeurs des variables pointées.




**Précautions à prendre avec les pointeurs**

Reprendre la même démarche avec les fichiers `exo2-pointeurs.c` et `exo2-pointeurs.f90`. Expliquer avec précision quelles instructions provoquent des erreurs ou des avertissements lors de la compilation. Certains comportements peuvent apparaître aléatoires à l'exécution. Modifier les programmes pour les éviter (quitte à supprimer certaines instructions).

exo2-pointeurs.c

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

#define ULI (unsigned long int)

int main(void) {
 int i=1;
 int *pi=NULL, *pj;
 float x, *px=NULL;

 printf("pi=%lu\n", ULI pi);
 *pi = 4; // essayer d'executer
 exit(EXIT_SUCCESS); // fin etape 1

 pi = &i;
 printf("pi=%lu\n", ULI pi);
 *pi *= 10;
 printf("i=%d, *pi=%d, pi=%lu\n", i, *pi, ULI pi);
 exit(EXIT_SUCCESS); // fin etape 2

 printf("pj=%lu\n", ULI pj);
 // resultat aleatoire
 *pj = 5; // essayer d'executer
 pj = NULL;
 printf("pj=%lu\n", ULI pj);
 // resultat certain
 exit(EXIT_SUCCESS); // fin etape 3

 x = 3.5f;
 px = &x;
 printf("x=%g , *px=%g, px=%lu\n", x, *px, ULI px);
 *px = *pi;
 printf("i=%d , *px=%g, px=%lu\n", i, *px, ULI px);
 exit(EXIT_SUCCESS); // fin etape 4

 px = pi; // essayer pour etape 5
 printf("i=%d , *px=%g, px=%lu\n", i, *px, ULI px);
 exit(EXIT_SUCCESS); // fin etape 5
}
```

exo2-pointeurs.f90

```f90
program pointeur

implicit none

integer, target  :: i=1
integer, pointer :: pi=>null(), pj
real, target     :: x
real, pointer    :: px=>null()

write(*,*) " pi assoc :", associated(pi)
pi = 4 ! essayer d'executer
stop ! fin etape 1

pi => i
write(*,*) " pi assoc :", associated(pi)
pi = 4
write (*,*) "i =",i,", pi =",pi
stop ! fin etape 2

write(*,*) " pj assoc :", associated(pj)
! resultat aleatoire
pj = 5 ! essayer d'executer
nullify(pj)
write(*,*) " pj assoc :", associated(pj)
! resultat certain
stop ! fin etape 3

x = 3.5
px => x
write (*,*) "x =", x, "px =", px
px = pi
write (*,*) "i =", i, " x =", x, "px =", px
stop ! fin etape 4

px => pi  ! essayer de compiler
write (*,*) "i =", i, "px =", px
stop ! fin etape 5

end program pointeur
```

exo2-pointeurs+.c

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

#define ULI (unsigned long int)

int main(void) {
 int i=1;
 int *pj, *pi=NULL;
 float x, *px=NULL;

 printf("pi=%lu\n", ULI pi);
 //*pi = 4; // essayer d'executer
 //exit(EXIT_SUCCESS); // fin etape 1

 pi = &i;
 printf("pi=%lu\n", ULI pi);
 *pi *= 10;
 printf("i=%d, *pi=%d, pi=%lu\n", i, *pi, ULI pi);
 //exit(EXIT_SUCCESS); // fin etape 2

 printf("pj=%lu\n", ULI pj);
 // resultat aleatoire
 //*pj = 5; // essayer d'executer, peut echouer
 pj = NULL;
 printf("pj=%lu\n", ULI pj);
 // resultat certain
 //exit(EXIT_SUCCESS); // fin etape 3

 x = 3.5f;
 px = &x;
 printf("x=%g , *px=%g, px=%lu\n", x, *px, ULI px);
 *px = *pi; // garder pour voir l'effet ! (warning)
 printf("i=%d , *px=%g, px=%lu\n", i, *px, ULI px);
 //exit(EXIT_SUCCESS); // fin etape 4

 // des entiers en octal correspondant
 // en float a des "valeurs speciales"
 i = 017740000000; // +Inf en float
 i = 037740000000; // -Inf en float
 i = 017740000001; // +NaN (quiet)
 i = 037740000001; // -NaN (quiet)
 i = 037750000001; // -NaN (signaling)
 px = pi; // essayer pour etape 5
 printf("i=%d , *px=%g, px=%lu\n", i, *px, ULI px);
 exit(EXIT_SUCCESS); // fin etape 5
}
```

En fortran, l'association entre pointeurs de types différents provoque une erreur dès la compilation.

```
px => pi  ! essayer de compiler
1
Error: Different types in pointer assignment at (1)
```

Cela prévient des erreurs graves qui pourraient en résulter. Il faut donc supprimer cette instruction pour réussir à compiler y compris pour mettre en œuvre les étapes précédentes.

En langage C, au contraire, elle ne produit qu'un warning à la compilation :

```
warning: assignment from incompatible pointer type
```

En C, on peut donc par exemple pointer une variable de type entier via un pointeur de `float` : le motif de 32 bits représentant un entier signé ne peut pas toujours être interprété en tant que `float` valide. Il correspond soit à des valeurs très différentes de la valeur de l'entier, soit à des codes de nombres réels dénormalisés (de valeur absolue inférieure à `FLT_MIN`, cf. annexe du polycopié fortran sur la norme IEEE), voire des codes spéciaux pour les `NaN`, `+Inf` ou `-Inf`. Le programme `exo2-pointeurs+.c` permet de stocker les valeurs octales d'un entier signé qui codent ces valeurs non numériques en `float`.
En fortran, on peut affecter des motifs de bits à des variables réelles via des lectures en format `B` (norme fortran 2008, avec `gfortran` ou `ifort`). Mais il n'est pas possible de faire pointer un pointeur de réel vers un entier : une solution pour lire le motif binaire comme un entier est d'utiliser la fonction intrinsèque `transfer`.

## Exercice 2 : Fonctions avec retour sans effet de bord A

Éditer les fichiers suivants, `produit.c` en C, puis `produit.f90` en fortran 90. À chaque question, prendre soin de faire une copie en incrémentant le nom de façon à conserver toutes les versions.

produit.c

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

/* declaration : prototype */
int prod(int x, int y);

/* definition : les instructions */
int prod(int x, int y) {
  int z;    /* locale */
  z = x * y;
  return z; /* valeur de retour */
}

int main(void) {
  int a1, b1, a2, b2;
  int p;
  printf("Saisir 4 entiers a1,b1,a2,b2\n");
  scanf("%d %d %d %d",
        &a1, &b1, &a2, &b2);
  /* deux appels dans une expression */
  p = prod(a1, b1) + prod(a2, b2);
  printf("a1*b1 + a2*b2 = %d\n", p);
  exit(EXIT_SUCCESS);
}
```

produit.f90

```
module m_prod ! => nom du fichier .mod
  implicit none
  contains ! procédures de module
    function prod(x, y)
      integer             :: prod
      integer, intent(in) :: x, y
      integer             :: z ! locale
      z = x * y
      prod = z ! valeur de retour
      return     ! sans valeur
    end function prod
end module m_prod

program produit
  use m_prod ! relit m_prod.mod
  ! ce qui rend visible l'interface
  integer :: a1, b1, a2, b2
  integer :: p
  write(*,*) "Saisir 4 entiers a1,b1,a2,b2"
  read(*,*) a1, b1, a2, b2
  ! deux appels dans une expression
  p = prod(a1, b1) + prod(a2, b2)
  write(*,*) "a1*b1 + a2*b2 = ", p
end program produit
```

**1)** Ces programmes font appel à une fonction `prod` qui ne modifie pas ses arguments et rend une valeur à l'appelant. Compiler dans les deux langages, observer les fichiers créés. Tester ces programmes. Modifier la fonction `prod` pour qu'elle n'utilise plus de variable locale `z` et tester.

B Essayer d'introduire une modification de la valeur de `x` dans `prod` en fortran. Expliquer.

Solution

En fortran, création du fichier `m_prod.mod` qui contient l'interface des procédures définies dans le module `m_prod`. L'attribut `intent(in)` de `x` interdit toute modification, par exemple :

```
In file produit.f90:7
      x = x + 1
      1
Error: Can't assign to INTENT(IN) variable 'x' at (1)
```

**2)** Déclarer la variable `p` en tant que réel `float` en C (ajuster aussi le format d'affichage) et `real` en fortran dans le programme principal. Expliquer les comportements à la compilation et à l'exécution. Quelle conversion est effectuée ?

Solution

Le produit, calculé en entier, est converti en réel, ce qui modifie l'affichage en fortran. Noter que le problème de l'éventuel dépassement de capacité se pose dans le type entier dans la fonction : le produit est limité en valeur absolue à `INT_MAX` en C et `HUGE(1)` en fortran, c'est à dire $2^{31} - 1$ sur une machine 32 bits.

**3)** En conservant `p` réel dans chaque langage, déclarer les variables `a1`, `b1`, `a2` et `b2` en tant que réels dans le programme principal (et adapter le format de saisie en C). Expliquer les comportements à la compilation et à l'exécution. En langage C, reprendre la compilation en ajoutant l'option `-Wconversion` et conclure.

Solution

La fonction reste définie avec des paramètres formels entiers, mais elle est appelée avec des paramètres effectifs réels. Le fortran refuse alors de convertir et si l'interface de `prod` est visible de l'appelant (ce

qui est le cas avec le `USE m_prod`), l'erreur est détectée dès la compilation :

```
  p = prod(a1, b1) + prod(a2, b2)
                           1
Error: Type mismatch in parameter 'x' at (1). Passing REAL(4) to INTEGER(4)
```

et de même pour `a1`.

Le C fait naturellement les conversions (à cause du passage par copie de valeur). Le compilateur n'avertit pas sauf avec l'option `-Wconversion`.

```
gcc-mni produit-fl+int.c
produit-fl+int.c: In function 'main':
produit-fl+int.c:19: attention : conversion to 'int' from 'float' may alter its value
produit-fl+int.c:19: attention : conversion to 'int' from 'float' may alter its value
produit-fl+int.c:19: attention : conversion to 'int' from 'float' may alter its value
produit-fl+int.c:19: attention : conversion to 'int' from 'float' may alter its value
produit-fl+int.c:19: attention : conversion to 'float' from 'int' may alter its value
```

**Un détail en C :** Jusqu'à gcc 4.2, l'option `-Wconversion` n'était pas recommandée en usage normal où le prototype est déclaré, car elle émettait un avertissement même dans le cas où les arguments formels de `prod` sont déclarés `float` ainsi que les arguments lors de l'appel. Il n'y a alors plus de conversion, mais `-Wconversion` signalait un comportement différent de celui où on ne déclare pas de prototype pour `prod` (où, pour des raisons historiques les `float` sont systématiquement promus en `double`).

```
produit-float.c:19: warning: passing argument 1 of 'prod' as integer rather
than floating due to prototype
```

Depuis gcc 4.3, ce comportement est celui de `-Wtraditional-conversion` alors que `-Wconversion` avertit des vrais problèmes potentiels de conversion sans référence à un type implicite... Donc, l'option `-Wtraditional-conversion` est de peu d'intérêt.

Attention, en nov 2010, `sappli` est encore sous `gcc-3.4.6` (et les mandriva 2007.1 sous `gcc-4.1.2`!) Donc on évitera `-Wconversion` sur le serveur. Heureusement, les mandriva 2009.1 utilisent `gcc-4.3.2`, donc `-Wconversion` est conseillé. C'est un cas où le serveur et les machines virtuelles n'ont pas le même comportement.

Extrait du `man gcc` 4.3 ou supérieur :

```
-Wtraditional-conversion

Warn if a prototype causes a type conversion that is different from what would happen to the
same argument in the absence of a prototype. This includes conversions of fixed point to
floating and vice versa, and conversions changing the width or signedness of a fixed point
argument except when the same as the default promotion.

-Wconversion

between real and integer, like abs (x) when x is double; conversions between signed and
unsigned, like unsigned ui = -1; and conversions to smaller types, like sqrtf (M_PI). Do not
warn for explicit casts like abs ((int) x) and ui = (unsigned) -1, or if the value is not
changed by the conversion like in abs (2.0). Warnings about conversions between signed and
unsigned integers can be disabled by using -Wno-sign-conversion.
```

**4)** Modifier la fonction `prod` de manière à ce que le programme se compile sans avertissement lorsque `a1`, `b1`, `a2` et `b2` sont déclarés réels dans le programme principal. Tester de nouveau.

Solution

Il faut déclarer réels les paramètres d'entrée et aussi de préférence le résultat (sinon double conversion).

## Exercice 3 : Procédures sans valeur de retour A

Ce sont les sous-programmes (`subroutine`) en fortran et les fonctions sans retour (résultat de type `void`) en C. Elles sont censées produire des effets de bord : entrées/sorties ou modification des paramètres passés.

**1)** Copier, puis éditer, tester et commenter les programmes `produit_et_somme` suivants, en C puis en fortran 90 :

produit_et_somme.c

```c
#include <stdio.h>
#include <stdlib.h>
/* declaration : prototype */
void prod_som(int x, int y);
/* definition */
void prod_som(int x, int y){
  /* x et y :  parametres d'entree */
  /* variables locales */
  int prod, som;
  prod = x * y ;
  som = x + y ;
  printf("produit = %d\n", prod);
  printf("somme = %d\n", som);
  return;
}

int main(void){
  int a, b;
  printf("Saisir a, b:\n");
  scanf("%d %d", &a, &b);
  prod_som(a, b);
  exit(EXIT_SUCCESS);
}
```

produit_et_somme.f90

```
module m_prod_som
implicit none
contains
  subroutine prod_som(x, y)
    ! parametres d'entree
    integer, intent(in) :: x, y
    ! variables locales
    integer             :: prod, som
    prod = x*y
    som = x+y
    write(*,*) 'produit =', prod
    write(*,*) 'somme = ', som
    return
  end subroutine prod_som
end module m_prod_som
program produit_et_somme
  use m_prod_som
  integer :: a, b
  write(*,*) 'Saisir a, b:'
  read(*,*) a, b
  call prod_som(a, b)
end program produit_et_somme
```

**2)** On souhaite modifier le programme `produit_et_somme` de manière à ce que les valeurs du produit et de la somme soient renvoyées au programme principal qui les imprimera : on déclarera donc les variables `prod` et `som` dans le programme principal et la procédure `prod_som` comportera alors quatre paramètres.

En C, les arguments sont **passés par copie de valeur** et les modifications effectuées sur les paramètres dans la fonction appelée n'affectent que sa copie locale des paramètres. Il est donc nécessaire de communiquer à la fonction appelée l'adresse du paramètre pour le partager avec l'appelant : le type du paramètre formel recevant copie de cette adresse est donc un pointeur sur le type de la variable à modifier, elle-même déclarée côté appelant.

En fortran, les arguments sont par défaut **passés par référence** ; les zones mémoires des paramètres sont alors communes à l'appelé et à l'appelant. Les modifications effectuées sur les paramètres dans la procédure sont donc visibles par l'appelant.

À titre d'exemple, expliquer la différence de syntaxe entre les fonctions `printf` et `scanf`. Modifier le code `produit_et_somme.c` de manière à passer en argument l'adresse des variables à modifier.

Effectuer la modification en précisant l'attribut `intent` pour tous les paramètres du sous-programme.

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

void prod_som(int x, int y, int *p_prod, int *p_som);

void prod_som(int x, int y, int *p_prod, int *p_som){
                      /* pointeurs sur les entiers */
  *p_prod = x * y;
  *p_som = x + y;

  printf("fonction: produit = %d\n", *p_prod);
  printf("fonction: somme = %d\n", *p_som);
  return;
}

int main(void){
  int a, b;
  int prod, som;

  printf("Saisir 2 entiers a, b:\n");
  scanf("%d %d", &a, &b);
  printf("a = %d b = %d\n", a, b);
  prod_som(a, b, &prod, &som);

  printf("main: produit = %d\n", prod);
  printf("main: somme = %d\n", som);

  exit(EXIT_SUCCESS);
}
```

```fortran
module m_prod_som
implicit none
contains
  subroutine prod_som(x, y, prod, som)
    ! parametres d'entree
    integer, intent(in)  :: x, y
    integer, intent(out) :: prod, som
    prod = x * y
    som = x + y
    write(*,*) 'ss-pgm : produit =', prod
    write(*,*) 'ss-pgm : somme = ', som
    return
  end subroutine prod_som
end module m_prod_som

program produit_et_somme
  use m_prod_som
  integer :: a, b
  integer :: s, p
  write(*,*) 'Saisir 2 entiers a, b:'
  read(*,*) a, b
  write(*,*) "a= ", a, " b= ", b
  call prod_som(a, b, p, s)
  write(*,*) 'pgm ppal: produit =', p
  write(*,*) 'pgm ppal: somme = ', s
end program produit_et_somme
```

**3)** En Fortran 90 et en C, écrire une procédure `echange` qui admet deux arguments entiers `x` et `y`, et qui les intervertit. Les variables `x` et `y` sont à la fois des arguments d'entrée et de sortie. Appeler cette procédure depuis le programme principal et imprimer les valeurs avant et après l'échange (on pourra vérifier ici la nécessité d'utiliser des pointeurs en C).

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

void vrai_echange(int *p_x, int *p_y); /* declaration */

void vrai_echange(int *p_x, int *p_y){ /* definition */
  int c;
  c = *p_y;
  *p_y = *p_x;
  *p_x = c;
  printf("vrai_echange: p_x et p_y = %p %p \n", p_x, p_y);
  return;
}

void faux_echange(int x, int y); /* declaration */

void faux_echange(int x, int y){ /* definition */
  int c;
  c = y;
  y = x;
  x = c;
  printf("faux_echange: x et y = %d %d \n", x, y);
  return;
}

int main(void){
  int a, b;
  printf("Saisir 2 entiers a et b:\n");
  scanf("%d %d", &a, &b);
  printf("main: a et b avant vrai_echange = %d %d\n", a, b);
  vrai_echange(&a, &b);
  printf("main: a et b apres vrai_echange = %d %d\n", a, b);
  faux_echange(a, b); /* echange des copies */
  printf("main: a et b apres faux_echange = %d %d\n", a, b);
  exit(EXIT_SUCCESS);
}
```

```fortran
module m_echange
  implicit none
  contains
    subroutine echange(x, y)
      integer, intent(inout) :: x, y
      integer :: z ! locale
      z = x
      x = y
      y = z
      write(*,*) "dans échange : a = ", x, " b = ", y
    end subroutine echange
end module m_echange

program ppal
use m_echange
  integer :: a, b
  write (*,*) "Saisir 2 entiers a et b"
  read(*,*) a, b
  write(*,*) "ppal avant échange : a = ", a, " b = ", b
  call echange(a, b)
  write(*,*) "ppal après échange : a = ", a, " b = ", b
end program ppal
```

# B Applications

## Exercice 4 : Fonction de Planck A

Soit la fonction de Planck

$$\text{Planck}(x) = \frac{x^3}{e^x - 1} \quad \text{où} \quad x = \frac{h\nu}{kT} \quad \text{est un argument sans dimension}$$

**1)** Écrire un programme `planck` qui, en vue d'un tracé, échantillonne sur un intervalle fini la fonction de Planck et affiche sur la sortie standard les valeurs de $x$ et de Planck($x$) (à raison d'un point par ligne). Dans un premier temps, on décrira l'intervalle $[0,5; 10]$ par pas de 0,5

**2)** Déplacer le calcul de la fonction de Planck dans une fonction `f_planck` à un argument $x$ renvoyant la valeur de Planck($x$). La fonction `f_planck` sera appelée à chaque passage dans la boucle décrivant l'intervalle.

**3)** Pour $|x|$ tendant vers zéro, l'expression initiale conduit à une indétermination. Lever cette indétermination par un développement de $\exp(x)$ au premier ordre. Modifier la fonction `f_planck` pour prendre en compte cette expression approchée quand $|x|$ est petit. On comparera $|x|$ avec :
- `EPSILON(x)` en fortran (fonction intrinsèque)
- `FLT_EPSILON` ou `DBL_EPSILON` en C (macro-constantes définies dans `float.h`)

Reprendre l'échantillonnage à partir de 0 pour un pas quelconque.

**4)** Tracer la fonction de Planck en fonction de $x$. Pour cela, rediriger les données de la sortie standard par défaut vers un fichier `planck1.dat` et utiliser `gnuplot` pour tracer les courbes (commande `gnuplot` puis `plot "planck1.dat" using ($1):($2) with points` où `$1` et `$2` sont les champs correspondant aux données que vous voulez tracer).

**5)** B Comparer les deux types d'erreurs dominantes commises au voisinage de $x = 0$ selon la méthode de calcul :
- **erreur systématique de troncature** (liée au nombre limité de termes) en utilisant le développement limité de l'expression autour de 0 d'une part : elle diminue quand on s'approche de 0.
- **erreur aléatoire d'arrondi** (en considérant le facteur le plus imprécisément évalué) avec l'expression initiale de Planck($x$) d'autre part : elle augmente quand on s'approche de 0.

Déterminer pour quelle valeur de $|x|$ ces deux erreurs sont égales et on doit changer de méthode de calcul afin de minimiser l'erreur totale. Modifier la fonction `f_planck` en conséquence.

Solution

```c
#include <stdio.h>
#include <stdlib.h>
#include <tgmath.h> // fonctions generiques
#include <float.h>
/* standard c99 pour sqrtf expf et fabsf */

float planck(float x); /* déclaration */

float planck(float x){
  float  y;
  if(fabs(x) > sqrt(2.f*FLT_EPSILON)) {
  // appelle fabsf et sqrtf  grace a tgmath.h
     y = x*x*x / (exp(x) - 1.f);
  }
  else{
     y = x * x ;
  }
  return y;
}

int main(void){
  int n, i;
  float x, xmin, xmax, dx ;
  float y ;
  xmin = 0.f;
  xmax = 10.f;
  dx = .5f;
  n =  1 + (int) ((xmax - xmin) / dx) ;
  for (i =1; i<= n; i++){
     x = xmin + (i -1) * dx ;
     y = planck(x) ;
     printf("%g %g\n", x, y);
  }
  exit(EXIT_SUCCESS) ;
}
```

```fortran
module m_planck
  implicit none
  contains
  function planck(x)
    real :: planck
    real, intent(in) :: x
    ! en comparant :
    ! l'erreur d'arrondi sur l'expression complète
    ! et l'erreur de troncature sur le développement
    !   au 1er ordre, on doit changer de mode de calcul
    ! pour x/2 = \epsilon / x
    if(abs(x) > sqrt(2.*epsilon(x))) then
       planck = x**3 /(exp(x) - 1.)
    else
       planck = x**2
    endif
  end function planck
end module m_planck

program tabule_planck
  use m_planck
  real :: x, y, xmin, xmax, dx
  integer :: n, i

  xmin = 0.
  xmax = 10.
  dx = .5
  n =  1 + int((xmax - xmin) / dx)
  do i = 1, n
     x = xmin + (i -1) * dx
     y = planck(x)
     write(*,*) x, y
  end do
end program tabule_planck
```

Quand on doit évaluer $f(x) = \dfrac{x^3}{\exp(x)-1}$ au voisinage de zéro, il y a une indétermination que l'on peut lever par un développement limité du dénomminateur $d(x) \approx 1+x+x^2/2+\cdots-1 \approx x\,(1+x/2+\cdots)$. Ainsi $f(x) \approx \dfrac{x^2}{1+x/2+\cdots} \approx x^2\,(1-x/2+\cdots)$. On peut tronquer le développement limité de $f(x)$ pour ne garder qu'un terme avec $f_1(x) = x^2$ : l'erreur relative de troncature associée avec ce calcul approché est de $|x/2|$.

Ainsi, pour évaluer la fonction de Planck, on doit distinguer :

(a) les très faibles valeurs de $x$ pour lesquelles on calcule $f_1(x) = x^2$ : on commet alors essentiellement une erreur de troncature.

(b) les autres valeurs de $x$ pour lesquelles on utilise la formule initiale $f_2(x) = \dfrac{x^3}{\exp(x)-1}$ qui présente une erreur d'arrondi qui peut être importante quand $|x|$ est faible.

Si on note $\varepsilon$ le plus grand flottant positif tel que $1+\varepsilon$ ne soit pas différent de 1, l'erreur relative maximale de représentation d'un flottant est $\varepsilon$. Dans une addition ou une soustraction, chacun des termes est représenté approximativement : le résultat est donc entaché d'une erreur dite d'arrondi dont la valeur absolue est majorée par la somme des erreurs de représentation des deux termes. Dans le cas d'une soustraction de termes proches, l'erreur relative d'arrondi peut donc être importante. Dans le calcul du dénominateur pour $x$ faible, l'erreur absolue sur $\exp(x)$ (mais aussi sur $\exp(x)-1$ car 1 est représenté exactement) est majorée par $\varepsilon$. Mais comme le dénominateur est de l'ordre de $x$, l'erreur relative sur ce dénominateur est de l'ordre de $\varepsilon/x$. Si on néglige l'erreur d'arrondi dans le calcul de $x^3$ et dans le rapport, l'erreur relative sur $f_2(x)$ est aussi de l'ordre de $\varepsilon/x$.

Le basculement entre les deux calculs doit se faire lorsque les erreurs relatives sont égales, soit $\varepsilon/x = x/2$.

Ainsi,

(a) pour $|x| \leqslant \sqrt{2\varepsilon}$, on calcule $f(x) \approx x^2$ et l'erreur dominante est déterministe et due à la troncature

(b) pour $|x| \geqslant \sqrt{2\varepsilon}$, on calcule $f(x) = \dfrac{x^3}{\exp(x)-1}$ et l'erreur dominante est aléatoire et due à l'arrondi.

**N.-B. :** Dans la description d'un intervalle par découpage à pas constant, il est conseillé de calculer le nombre de pas à effectuer et de faire une boucle avec compteur. De plus, on évitera d'accumuler les erreurs d'arrondi par additions successives ( `x = x + dx` à chaque pas) en effectuant une multiplication par le numéro du pas ( `x = xmin + (i-1)*dx` où `i` est l'entier servant de compteur).

## Exercice 5 : Calcul du nombre d'or par dichotomie A

**1)** Montrer que la racine positive de l'équation $x^2 = x+1$ est le nombre d'or $\phi$.

**2)** *Résolution de l'équation $x^2 = x+1$ par dichotomie.*

Bref rappel de la méthode de dichotomie : on se donne un intervalle initial de recherche dont les bornes encadrent la racine ; puis on divise cet intervalle par le milieu et on le remplace par le sous-intervalle deux fois plus petit dans lequel se trouve la racine, et on itère le processus. La longueur de l'intervalle dont les bornes encadrent la racine diminue donc en $2^{-n}$ en fonction de l'ordre $n$ d'itération.

(a) Vérifier que cette équation admet une solution dans l'intervalle $[1\,;\,2]$.

(b) Écrire une fonction `dichotomie` qui accepte comme arguments les deux bornes de l'intervalle d'entrée (soit [a ;b]). Cette fonction remplace l'un des arguments par le milieu de l'intervalle de départ (par exemple $[\frac{a+b}{2}; b]$), de telle sorte que la solution de l'équation soit toujours contenue dans l'intervalle de sortie.

(c) Demander la précision souhaitée à l'utilisateur du programme et appeler la fonction `dichotomie` autant de fois que nécessaire pour trouver la solution de l'équation à la précision adéquate.

(d) B Indiquer ce qui limite la précision et comment modifier le programme pour calculer par exemple avec 10 chiffres siginificatifs.

Solution

La largeur de l'intervalle est divisée par 2 à chaque étape, donc il est possible de connaître a priori le nombre d'étapes nécessaires pour une précision donnée.

$$b_n - a_n = \frac{b_{n-1} - a_{n-1}}{2} = \frac{b_0 - a_0}{2^n}$$

On peut donc utiliser une boucle avec compteur au lieu de la boucle `while` présentée dans les codes ci-dessous.

Fortran (sous-type réel à choisir avec `prec`)

dichotomie-or.f90 précision ajustable

```
module rect ! Le module
  implicit none
  ! variable déclarée au niveau du module
  !integer, parameter :: prec=selected_real_kind(p=6) ! simple préc.
  !integer, parameter :: prec=selected_real_kind(p=15) !double prec.
  private f ! => pas de visibilté hors du module
  contains
    function f(x)
      real(kind=prec), intent(in) :: x
      real(kind=prec) :: f
      f = x**2-x-1._prec
    end function f

    subroutine dichotomie(x1,x2)
       real(kind=prec),intent(inout) :: x1, x2
       real(kind=prec)               :: x3
       x3 = (x1+x2)/2._prec
       if (f(x1)*f(x3) < 0) then
          x2 = x3
       else
          x1 = x3
       endif
    end subroutine dichotomie
end module rect

program nombredor
  use rect
  implicit none
  real(kind=prec) :: p, g, eps
  integer :: compteur
! Determination du nombre d'or par dichotomie dans [1,2]
  write(*,*) "Entrer la précision choisie eps:"
  write(*,*) "pas mieux que ", EPSILON(1._prec)
  read(*,*) eps
  p = 1.
  g = 2.
  compteur = 0
  do while (abs(g-p) > eps)
    call dichotomie(p,g)
    compteur = compteur + 1
  enddo
  write(*,*) "Le nombre d'iterations nécessaire pour &
   & une precision de ", eps, "est", compteur
  write(*,*) "Nombre d'or =", (g+p)/2._prec
  write(*,*) "Nombre d'or =", (1._prec+sqrt(5._prec))/2._prec
end program nombredor
```

Langage C

dichotomie-or.c version float

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <limits.h>
#include <float.h>

float f(float x);

float f(float x){
        return (x*x -x -1.f);
}

void dichotomie(float *p_x1, float *p_x2);

void dichotomie(float *p_x1, float *p_x2){
        float x3;
        x3 = (*p_x1 + *p_x2)/2.f;
        if (f(*p_x1)*f(x3) < 0){
                *p_x2 = x3;
        }
        else{
                *p_x1 = x3;
        }
        return;
}

int main(void){
        float p, g, eps;
        int compteur;
/* Determination du nombre d'or par dichotomie */
       printf("Entrer la precision choisie eps: \n");
       printf("pas mieux que %g\n", FLT_EPSILON);
       scanf("%g", &eps);
       p = 1.;
       g = 2.;
       compteur = 0;
       while(fabs(g-p) > eps){
           dichotomie(&p, &g);
           compteur++;
       }
       printf("Le nombre d'iterations necessaire "
       "pour une precision de %g est %d \n", eps, compteur);
       printf("Nombre d'or = %f \n", (g+p)/2.);
       printf("Nombre d'or = %f \n", (1.+sqrt(5.f))/2.f);
       exit(EXIT_SUCCESS);
}
```

dichotomie-or-dbl.c version double

```c
#include <stdio.h>
#include <stdlib.h>
#include <tgmath.h>
#include <limits.h>
#include <float.h>

double f(double x);

double f(double x){
        return (x*x -x -1.);
}

void dichotomie(double *p_x1, double *p_x2);

void dichotomie(double *p_x1, double *p_x2){
        double x3;
        x3 = (*p_x1 + *p_x2)/2.;
        if (f(*p_x1)*f(x3) < 0){
                *p_x2 = x3;
        }
        else{
                *p_x1 = x3;
        }
        return;
}

int main(void){
        double p, g, eps;
        int compteur;
/* Determination du nombre d'or par dichotomie */
       printf("Entrer la precision choisie eps: \n");
       printf("pas mieux que %lg\n", DBL_EPSILON);
       scanf("%lg", &eps);
       p = 1.;
       g = 2.;
       compteur = 0;
       while(fabs(g-p) > eps){
           dichotomie(&p, &g);
           compteur++;
       }
       printf("Le nombre d'iterations necessaire "
       "pour une precision de %g est %d \n", eps, compteur);
       printf("Nombre d'or = %.16lf \n", (g+p)/2.);
       printf("Nombre d'or = %.16lf \n", (1.+sqrt(5.))/2.);
       exit(EXIT_SUCCESS);
}
```

**N.-B. :** En C, ne pas confondre la fonction `abs` d'argument entier, avec `fabs` d'argument réel. Utiliser un test sur `abs` avec un argument de type `float` ou `double` arrêtera la boucle dès qu'il est inférieur à 1.

## Exercice 6 : Approximation de $\pi$

On cherche à obtenir une estimation de $\pi$ par différentes méthodes.

### Première méthode : série alternée A

On utilise le fait que $\arctan(1) = \dfrac{\pi}{4}$ et le développement en série entière de $\arctan(x)$ :

$$\arctan(x) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{2k+1} \quad \text{pour} \quad |x| \leqslant 1$$

**1)** Comment calculer de proche en proche $(-1)^k x^{2k+1}$ sans faire appel à des calculs de puissance ?

**2)** Écrire une fonction `mon_arctan` qui calcule la somme des $n$ premiers termes de la série entière précédente. Les arguments de cette procédure seront $x$ et $n$. Utiliser cette procédure pour obtenir une estimation de $\pi$. Contrôler la précision relative de l'approximation en comparant avec la valeur de $\pi$ calculée avec par exemple la fonction `atan`.

**3)** Sachant que, dans une série alternée, le reste peut être majoré par la valeur absolue du premier terme négligé dans la somme finie, intégrer dans le programme le calcul du nombre de termes suffisant pour limiter l'erreur de troncature à une valeur choisie par l'utilisateur (en ignorant les erreurs d'arrondi).

**4)** B Dans quel ordre doit on effectuer la somme pour minimiser l'erreur d'arrondi ?

Solution

**1)** Le calcul de la série entière nécessite une alternance de signe qui peut être programmée avec l'opérateur d'élévation à la puissance en fortran mais en C, on doit soit appeler la fonction `pow` soit utiliser l'opérateur `%` de modulo. On peut aussi explicitement programmer l'alternance de façon itérative en évitant l'élévation à la puissance grâce à une variable intermédiaire `u` :

```
u = x
s = u
do k=1, n
  u = -u * x * x
  s = s + u / (2*k + 1)
end do
```

```
u = x ;
s = u ;
for (k=1; k<=n; k++){
  u = -u * x * x ;
  s = s + u / (2*k + 1) ;
}
```

**2)** Les codes proposés pages 13 et 14 n'utilisent pas ces méthodes car ils commencent par sommer les plus petits termes (plus grands indices).

**3)** Si $|x| \leqslant 1$, la série est alternée et le reste est majoré par la valeur absolue du premier terme négligé. L'erreur absolue est donc majorée par $\dfrac{x^{2n+3}}{2n+3}$ c'est à dire $\dfrac{1}{2n+3}$ pour $x = 1$. Pour que l'erreur relative soit inférieure à $\varepsilon$, il faut sommer jusqu'à $k = n = \dfrac{2}{\pi\varepsilon} - 1,5$. L'erreur de troncature décroît linéairement avec le nombre de termes. Mais la précision relative finale est de toute façon limitée par l'erreur d'arrondi, soit `EPSILON(x)` en `REAL` et `FLT_EPSILON` en `float`, tous deux de l'ordre de $10^{-7}$.

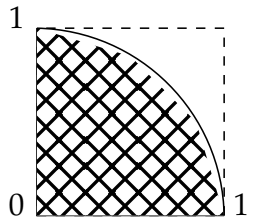
**4)** Encore faut-il limiter **l'accumulation** des erreurs d'arrondi. Pour cela, il est préférable de sommer en commençant par les plus petits termes. Dans le cas général $|x| \leqslant 1$, la mise en œuvre de la sommation dans cet ordre est facilitée par une factorisation qui s'inspire du schéma de Horner.

$$\sum_{k=0}^{n}(-1)^k \frac{x^{2k+1}}{2k+1} = x\left(1 - \frac{1}{3}x^2\left(1 - \frac{3}{5}x^2\left(1 - \frac{5}{7}x^2(\cdots(1 - \frac{2n-1}{2n+1}x^2))\right)\right)\right) \tag{7.1}$$

**Deuxième méthode : Monte-Carlo** AB

L'estimation de $\pi$ peut aussi être obtenue par un calcul d'aire. Dans la figure ci-contre, le rapport de l'aire du secteur angulaire de 90 degrés à l'aire du carré vaut :

$$\frac{\text{Aire en grisé}}{\text{Aire du carré}} = \frac{\pi}{4}$$




L'aire peut être estimée par une méthode de Monte-Carlo dont l'algorithme est le suivant. On tire aléatoirement $n$ points **répartis uniformément** dans le carré $[0\,;\,1] \times [0\,;\,1]$. Le rapport des aires est estimé par le rapport du nombre de points situés à l'intérieur du secteur angulaire au nombre total de points tirés.

On utilisera les générateurs de nombres pseudo-aléatoires uniformes fournis pour chaque langage :
- en fortran, le sous-programme intrinsèque standard `random_number` permet des tirer des valeurs équidistribuées sur l'intervalle $[0\,;\,1]$ via `CALL RANDOM_NUMBER(x)` où `x` est un **réel**.
- en C, la fonction `rand()` qui rend un **entier** entre 0 et `RAND_MAX` (voir `man 3 rand`).

**1)** Rajouter au module précédent une procédure effectuant l'intégration de Monte-Carlo.

**2)** B Comment évolue la précision avec le nombre de tirages ? Conclure sur la méthode qui vous semble la plus efficace.

Solution (en imposant le nombre de termes de la série et en commençant par les plus grands indices)
Pour le programme principal :

```
1 program pi
2 use m_approx
3 implicit none
4 integer :: n
5 real :: x, mon_pi
6 write(*,*) 'Entrer un nombre reel x tel que |x| <=1'
7 read (*,*) x
```

```
write(*,*) 'Entrer le nombre de termes du developpement limite'
read (*,*) n
write(*,*) "Le developpement limite a l'ordre ", 2*n+1, " &
            &de 4 * atan(", x, ") est ", 4 * mon_arctan(x,n)
! mon_arctan est une fonction
write(*,*) "Avec la fonction intrinsèque "
write(*,*) " 4 * atan(", x,") = ", 4 * atan(x)
write(*,*) ! saut de ligne
write(*,*) 'Entrer le nombre de points dans la methode de Monte-Carlo'
read (*,*) n
call montecarlo(n, mon_pi)
! montecarlo est un sous-programme
write(*,*) 'Valeur approchee de pi = ', mon_pi
end program pi
```

Et pour le module :

```
module m_approx
implicit none
contains
 function mon_arctan(x, n) ! calcul de atan(x) à l'ordre n
   real, intent(in) :: x
   integer, intent(in) :: n
   real :: mon_arctan
   integer :: k
   mon_arctan = 0.
   do k=n, 0, -1 ! faire la somme en commençant par les plus petits
      mon_arctan = mon_arctan + (-1)**k * x**(2*k + 1) / (2*k + 1)
   enddo
 end function mon_arctan

 subroutine montecarlo(n, resul)
   integer, intent(in) :: n
   real, intent(out) :: resul
   ! version "vectorielle" sans boucle
   real, dimension(n) :: tirage_x, tirage_y, modul
   call random_number(tirage_x) ! on tire n nb aleatoires compris dans [0;1[
   call random_number(tirage_y) ! idem
   modul = tirage_x**2 + tirage_y**2 ! module carré de (x,y)
   ! compter le nombre de points dans le quart de disque
   resul = 4. * count(modul <= 1) / real(n)
 end subroutine montecarlo
end module m_approx
```

En C (version `float`)

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

float mon_arctan(float x, int n);
float mon_arctan(float x, int n){ /* calcul de atan(x) à l'ordre n */
   float somme;
   int k ;
   somme = 0.f;
   for(k=n; k>=0; k--) { /* faire la somme en commençant par les plus petits */
      somme += (1 - 2*(k%2)) * pow(x, (float) (2 * k + 1)) /
                               (float) (2 * k + 1) ;
   }
   return somme ;
```

```
}

void montecarlo(int n, float *result);
void montecarlo(int n, float *result){
  int compt = 0;
  int k;
  float x, y ;
  for (k = 1; k<=n ; k++){
      x =  (float) rand()/ ((float) RAND_MAX + 1.);
      y =  (float) rand()/ ((float) RAND_MAX + 1.);
      if( (x*x + y*y) <= 1 ) {
         compt++;
      }
  }
  *result = 4.f * (float) compt / (float) n ;
}

int main(void){
  int n;
  float x, mon_pi ;
  x = 1.f;
  printf("Entrer le nombre de termes du développement limité\n");
  scanf("%d", &n);
  printf("Le développement limité à l'ordre %d de 4 * atan(%f)"
       " est  %f\n", 2*n +1, x, 4.f * mon_arctan(x, n) );
  printf("Et 4 * atan(%f) = %f\n", x, 4.f * atan(x));
  printf("Erreur relative par la série %e\n",
          (mon_arctan(x,n) - atan(x))/ atan(x));
  printf("Entrer le nombre de points utilisés par la méthode de Monte-Carlo\n");
  scanf("%d", &n);
  montecarlo(n, &mon_pi) ;
  printf("Valeur approchée de pi = %f\n", mon_pi);
  printf("Erreur relative par MonteCarlo %e\n", (mon_pi/4.f - atan(x))/ atan(x));
  exit(EXIT_SUCCESS);
}
```

La vitesse de convergence de la méthode de Monte Carlo (comptage de points) est faible : l'écart relatif est proportionnel à $1/\sqrt{N}$, alors qu'il évolue en $1/N$ dans le calcul par série alternée.
**N.-B. :** En langage C, on prendra soin de ne pas calculer `RAND_MAX + 1` en type `int`, sous peine de dépassement de capacité qui donne une valeur négative.