TE 7

Procédures en fortran et C

A Notions fondamentales

Exercice 1 : Introduction aux pointeurs A

Notions de base sur les pointeurs

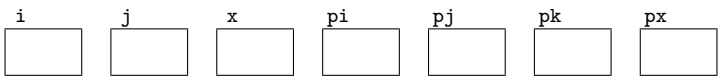
exo1-pointeurs.c

```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 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
```

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

```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.

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 ?

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.

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.

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.

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).

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.

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 ϕ.

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.

Exercice 6 : Approximation de π

On cherche à obtenir une estimation de π 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 ?

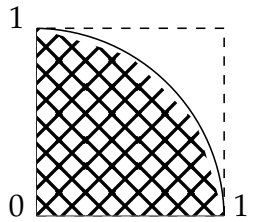
2) Écrire une fonction `mon_arctan` qui calcule la somme des n premiers termes de la série entière précédente. Les arguments de cette procédure seront x et n. Utiliser cette procédure pour obtenir une estimation de π. Contrôler la précision relative de l'approximation en comparant avec la valeur de π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

Deuxième méthode : Monte-Carlo **AB**

L'estimation de π 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.