TE 1-A Introduction

Objectif Il s'agit de préparer les TE portant sur des thèmes numériques en introduisant la mise en œuvre conjointe des deux outils utilisés par la suite :

- un programme de calcul en fortran ou C, en fichiers source séparés, avec compilation gérée par l'outil `make` : ce code écrit ses résultats dans un fichier ;
- une application graphique (sous `gnuplot`, `scilab` ou `python`) qui relit ce fichier afin de tracer les résultats.

À titre d'introduction, on se propose de tabuler quelques fonctions avec un pas fixe en fortran ou C, pour les représenter ensuite graphiquement.
Dans un premier temps, il est conseillé de traiter l'ensemble (calcul et visualisation) dans une version simplifiée ne comportant pas les éléments signalés par le signe **(+)** dans l'énoncé qui seront implémentés ultérieurement.

1 Partie calcul en fortran ou C

1.1 Organisation des codes

La structure générale, présentée en cours, doit permettre la compilation séparée des différents fichiers. Elle comporte quatre fichiers :

1. Un programme principal (fichier `ppal.f90` ou `ppal.c`) auquel est confié le dialogue avec l'utilisateur pour choisir **(+)** la fonction à étudier, l'abscisse de début, le pas d'échantillonnage et l'abscisse de fin. Ce programme crée alors le tableau des abscisses et le transmet à la procédure `tabule` qui calcule les ordonnées. Il fournit enfin les tableaux de résultats à la procédure d'écriture sur fichier.
2. Un module (fichier `fcts.f90` ou `fcts.c`) définissant les différentes fonctions à tester (voir section 2).
3. Un module (fichier `methode.f90` ou `methode.c`) hébergeant le sous-programme ou la fonction `tabule` qui évalue la fonction passée en argument aux points de la grille fournie par le programme appelant.
   Cette procédure sera conçue de façon à pouvoir traiter une fonction quelconque. On doit pouvoir changer de fonction sans recompiler la procédure.
4. Un module (fichier `util.f90` ou `util.c`) hébergeant une procédure `ecrit` chargée d'écrire le fichier formaté des résultats.

Module des fonctions à tester
Fichier `fcts.f90` ou `fcts.c`

Définition des différentes fonctions à tester (voir section 2).
Un seul argument pour chacune des fonctions ; les paramètres éventuels sont des variables globales du module **(+)**.

Module de la méthode à appliquer
Fichier `methode.f90` ou `methode.c`

Arguments de la méthode `tabule` :

- la fonction à tester
- en C seulement, la taille commune des tableaux
- le tableau des abscisses rempli par le programme principal
- le tableau des ordonnées à remplir

Programme principal
Fichier `ppal.f90` ou `ppal.c`

- Saisie des paramètres : abscisse de début, abscisse de fin, pas
- Construction du vecteur `x` comportant la grille des abscisses
- Choix de la fonction à tabuler **(+)**
- Appel de la méthode `tabule` pour remplir le vecteur des ordonnées
- Appel de la procédure d'écriture des résultats sur fichier

Module d'écriture sur fichier
Fichier `util.f90` ou `util.c`

Rôle de la procédure `ecrit` :

- Ouverture du fichier
- Écriture de l'entête avec les valeurs extrêmes en abscisse et ordonnée **(+)**
- Boucle d'écriture des couples $x, f(x)$ à raison d'un couple par ligne
- Fermeture du fichier

1.2 Structure du fichier de résultats

1.2.1 Entête du fichier de résultats

(+) Le fichier de résultats comportera trois lignes d'**en-tête** indiquant respectivement :
- les bornes des abscisses ;
- les bornes des ordonnées (valeurs minimale et maximale [1]) ;
- le nombre de points et le pas d'échantillonnage.

Chacune des lignes d'entête débutera par le caractère `#` qui permet à `gnuplot` et à certains outils de `python` de considérer que ces lignes ne représentent pas les données à tracer.
Cet entête ne sera implémenté qu'après un premier test et une visualisation.

1.2.2 Corps du fichier de résultats

Le **corps du fichier** sera constitué des couples abscisse, ordonnée des points de la grille, à raison d'un point par ligne. On spécifiera un format compatible avec la dynamique des valeurs et assurant une précision relative de l'ordre de 10^{-7}.

1.3 Aspects spécifiques aux langages

1.3.1 Langage C99

Pour simplifier, on choisira d'utiliser des tableaux automatiques déclarés tardivement (selon le standard `c99`) dans le programme principal, après choix du nombre de points. La fonction à tabuler sera vue par `tabule` sous la forme d'un pointeur de fonction à valeur de type `float` et à un argument de type `float`. On respectera donc les prototypes suivants :

methode.h
```
void tabule( float (* pfct )(float), int n, const float xx[n], float yy[n] ) ;
```

util.h
```
void ecrit( char * filename, int n, const float xx[n], const float yy[n] );
```

1.3.2 Langage fortran

En fortran, les tableaux seront alloués sur le tas dans le programme principal, qui devra les libérer explicitement. Il est conseillé de créer une interface abstraite de fonction réelle d'une variable réelle avec `ABSTRACT INTERFACE` dans un module dédié.

```
module m_interf ! définissant l'interface abstraite des fonctions
  abstract interface  ! de la fonction abstraite passée en argument
    function ft(t)    ! nom à utiliser comme type de procédure
    implicit none
    real :: ft              ! type du résultat
    real, intent(in) :: t   ! type de l'argument
    end function ft
  end interface
end module m_interf
```

Le sous-programme `tabule` utilisera cette interface pour déclarer la fonction formelle qu'il traite.

```
subroutine tabule(fct, xx, yy) ! echantillonage d'une fct
 use m_interf ! rend localement visible le type de procédure ft
 procedure(ft) :: fct
 real, dimension(:), intent(in)  :: xx
 real, dimension(:), intent(out) :: yy
 ...
end subroutine tabule
```

1. En langage C, on pourra écrire des fonctions `maxfloat1d` et `minfloat1d` pour calculer maximum et minimum d'un tableau 1D de float.

Les fonctions tests seront définies dans le module indépendant `fcts`, mais elles devront respecter l'interface abstraite.
Enfin, la procédure `ecrit` respectera l'interface suivante :

```
subroutine ecrit(filename, xx, yy)
 character(len=*), intent(in) :: filename ! nom du fichier de resultats
 real, dimension(:), intent(in) :: xx, yy ! tableaux des x et y
 ...
end subroutine ecrit
```

1.4 Mise au point du `makefile`

On associera à ces codes un fichier `makefile` pour prendre en charge la compilation séparée et l'édition de liens. Le fichier sera construit progressivement et testé avant d'être paramétré.
Une fois la compilation effectuée à la main, on pourra utiliser les options `-MM` et `-M` des compilateurs pour extraire les dépendances des fichiers objets, fournissant un embryon de fichier `makefile` :
`gcc -MM fichier.c` affiche les dépendances de `fichier.o` (nécessite les `.h`)

`gfortran -cpp -M fichier.f90` affiche les dépendances de `fichier.o` (nécessite les `.mod`)

Les règles implicites suffiront alors à générer les fichiers objets au moins dans le cas du C. Placer la cible de l'exécutable en premier pour que la simple commande `make` permette de lancer sa construction. Introduire ensuite les options de compilation (variables `CFLAGS` en C et `FFLAGS` en fortran sous `make`) fournies habituellement via des alias (qui ne sont pas reconnus par `make`).
On adjoindra enfin une cible `clean` qui supprime les fichiers objets, les fichiers de modules (suffixe `.mod`) pour le fortran et l'exécutable. Attention, `make clean` ne doit pas être systématiquement lancé pendant la mise au point des codes !

2 Fonctions proposées

On pourra tout d'abord tester le code avec une fonction élémentaire facile à vérifier comme $y(t) = t^2$. On étudiera ensuite les fonctions suivantes entre $x = 0$ [2] et $x = 20$.

2.1 Équation logistique

L'équation différentielle (1) régit l'évolution d'une population (en fait, la variable $y = N/N_0$ représente le rapport du nombre d'individus N à une population de référence N_0) avec un taux de croissance $a > 0$ en présence d'un mécanisme la limitant à une valeur maximale $k > 0$.

$$\frac{\mathrm{d}y}{\mathrm{d}t} = ay\left(1 - \frac{y}{k}\right) \tag{1}$$

On suppose qu'à l'instant initial, $0 < y(0) < k$. Montrer que la solution analytique se met sous la forme (2).
On prendra $t_0 = 0.$, $y_0 = 0.5$, $a = 1.$ et $k = 1$.

$$y(t) = \frac{k}{1 + \dfrac{k - y_0}{y_0}\exp\left(-a(t - t_0)\right)} \tag{2}$$

2.2 Gaussienne

On considère l'équation différentielle (3). On suppose que $y_0 > 0$ et $a > t_0$.

$$\frac{\mathrm{d}y}{\mathrm{d}t} = -y\left(t - a\right)/\tau^2 \tag{3}$$

2. Attention : en C, `y0`, `y1` et `yn` sont des fonctions de Bessel, ainsi que `j0`, `j1` et `jn`. On évitera donc de nommer ainsi des variables.

Montrer que la solution analytique se met sous la forme (4) d'une gaussienne centrée en a et de largeur τ.
L'évaluation numérique de l'expression (4) est susceptible de provoquer des dépassements de capacité à cause des exponentielles alors que le résultat final reste dans le domaine des réels sur 32 bits. Reformuler l'expression (4) avant de la coder afin d'éviter les dépassements de capacité et aussi de minimiser les erreurs d'arrondi (éviter les calculs de différences).
On choisira $t_0 = 0.$, $y_0 = 0.5$, $a = 4.$ et $\tau = \sqrt{2}$.

$$y(t) = y_0 \exp\left[\frac{1}{2}\left(\frac{t_0 - a}{\tau}\right)^2\right] \exp\left[-\frac{1}{2}\left(\frac{t - a}{\tau}\right)^2\right] \quad (4)$$

3 Partage des paramètres des fonctions avec le programme principal

Les fonctions test traitées par la méthode sont des fonctions d'une seule variable. Mais elles peuvent comporter des paramètres que la méthode ne doit pas connaître.
Dans un premier temps, on peut considérer comme fixés les paramètres a, k, τ, y_0 et t_0 des fonctions à tester. Ils peuvent alors être locaux à ces fonctions.

(+) Dans un second temps, on souhaite que le programme principal puisse choisir les paramètres a, k, τ, y_0 et t_0 des fonctions. À cet effet, il faut leur accorder une visibilité dans le programme principal au lieu de les considérer comme locaux à chaque fonction : ce seront donc des **variables globales**.

en fortran Ces paramètres doivent donc être déclarés dans le module `fcts` mais en dehors des fonctions du module qui y ont cependant accès. L'instruction `use fcts` les rend alors visibles sans redéclaration dans le programme principal qui leur affecte des valeurs.

en C Ces paramètres doivent être déclarés comme variables globales partagées entre le programme principal et les fonctions à tester. À cet effet, on doit préciser le qualificatif `extern` lors de leur déclaration. Une solution consiste à utiliser le préprocesseur pour inclure un fichier contenant les déclarations de ces paramètres à la fois dans le fichier définissant les fonctions et dans celui de la fonction `main` qui leur affectera des valeurs.

4 Lecture des données et visualisation en `python`

Sur la machine virtuelle `OpenSuse`, la version 3 de `python` est disponible. On utilisera le package `numpy` pour les aspects numériques et les structures de tableaux multidimensionnels de type `array`. La visualisation fera appel au package `matplotlib`. Les packages seront chargés en indiquant les raccourcis `np` et `plt` pour les espaces de noms via :

```
import numpy as np
import matplotlib.pyplot as plt
```

4.1 Méthode de travail

On testera tout d'abord les instructions `python` de façon interactive, avec `ipython`[3]. Puis on les intégrera rapidement dans un fichier texte `affiche.py` qui sera édité pour apporter les améliorations nécessaires ; ce script sera exécuté par l'instruction `python affiche.py`.

4.2 Lecture du fichier

La lecture des données numériques depuis le fichier texte nommé `"fic"` peut s'effectuer avec la fonction `loadtxt` de `numpy` :
`mat = np.loadtxt("fic", dtype='float')`
qui rend une matrice dont les éléments sont par défaut des flottants. Elle possède un paramètre optionnel `skiprows` qui permet de spécifier le nombre de lignes d'entête à sauter avant de lire la matrice.

3. On pourra lancer `ipython --matplotlib`.

La lecture de l'entête peut se faire avec la primitive `readline` de façon très comparable au fortran ou au C, par exemple pour lire les deux premières lignes du fichier texte `"fic"`.

```
file = open("fic",'r')
texte1 = file.readline()
texte2 = file.readline()
file.close()
```

Chaque ligne est stockée dans une chaîne qui comporte le changement de ligne `\n` en fin. Si on les concatène avec l'opérateur `+`, on obtiendra une chaîne de deux lignes terminée par un changement de ligne. On pourra remplacer `"\n"` par un blanc avec
`ligne = (texte1+texte2).replace("\n", " ")`
Ces informations pourront être utilisées pour documenter le graphique.

4.3 Tracé graphique

La fonction principale d'affichage de courbes $y = f(x)$ est `plot`. Elle admet comme arguments le vecteur **colonne** des abscisses et celui des ordonnées pour tracer une courbe. Pour tracer plusieurs courbes avec la même grille d'abscisses, il faut lui fournir des matrices construites avec les vecteurs des différentes courbes.

```
# une courbe ou x et v1 sont des tableaux 1D de même taille
plt.plot(x, y1) // affiche y1(x)
```

D'autres fonctions permettent notamment de fournir une légende, d'ajouter un titre, des libellés d'axes, de choisir des échelles logarithmiques...

```
plt.grid(True) # ajout d'une grille
# libelles d'axes
plt.xlabel("x")
plt.ylabel("f2(x)")
# une ou des lignes de titre
plt.title(texte1+texte2, fontsize=8)
# legende (une chaîne par courbe)
plt.legend(["f2"])
```

La production de fichiers graphiques pour sauvegarder la figure affichée s'effectue avec `savefig`, par exemple au format pdf :
`plt.savefig("f2.pdf")`