UPMC

Master P&A/SDUEE

MNCS UE 4P056

Méthodes Numériques et Calcul Scientifique

Introduction

2016–2017

Jacques.Lefrere@upmc.fr

Table des matières

1 Rappels sur les tableaux

1.1 Trois types de tableaux

— **tableaux statiques : taille fixée** à la compilation ; réservation par le compilateur

⇒ manque de souplesse mais parfois utile (ex : les 12 mois par an)

— **tableaux automatiques : taille définie lors de l'exécution**

allocation et libération «automatiques» (par le compilateur) sur la **pile** (***stack***)

⇒ **portée limitée** à la procédure (au bloc en C99) et aux procédures appelées

impossible de les rendre accessibles à l'appelant

⇒ taille limitée par celle de la pile (`ulimit -s` pour changer)

— **tableaux dynamiques : taille variable** et emplacement définis à l'exécution

allocation et libération «manuelles» (par l'utilisateur), sur le **tas** (***heap***)

⇒ **portée globale** : peuvent être alloués dans une procédure et rendus accessibles dans l'appelant (standard fortran 2003 ou option de fortran 95)

1.2 Cycle élémentaire d'un tableau de taille variable (pile et tas)

	automatique (pile)	dynamique (tas)
1	choix de la taille du tableau	
2	**allocation** de la mémoire	
	implicite lors de la déclaration	**explicite** par une allocation dynamique
3	utilisation du tableau	
4	**libération** de la mémoire	
	implicite par sortie de la portée	**explicite** par libération manuelle

Risques de non-libération avec les tableaux sur le tas

Si on alloue via un pointeur de tableau une cible anonyme, ne pas désassocier ce pointeur avant de libérer la zone, sinon **fuite de mémoire** (*memory leak*) $\Rightarrow$ grave si dans une boucle

Ce cycle (1-2-3-4) peut faire partie d'une boucle...

1.3 Tableaux 2D : application aux matrices

Ordre de stockage en mémoire

Langage C	fortran
Indice le plus **rapide** (éléments contigus)	
le plus **à droite**	le plus **à gauche**
Rangement des matrices	
par **lignes**	par **colonnes**
tableaux de tableaux $\Rightarrow$ difficile d'accéder aux colonnes	sections de tableaux $\Rightarrow$ facile d'accéder aux lignes/colonnes

N.-B. : on peut toujours « **aplatir** » les tableaux 2D en tableaux 1D (comme dans certaines bibliothèques mathématiques, `lapack`) quitte à calculer les positions.

Allocation dynamique sur le tas d'un tableau 2D en C

Simulation des tableaux 2D au moyen de pointeurs de pointeurs :
structuration nécessaire de la mémoire avec pointeurs de début de ligne.

1.4 Bilan sur l'allocation de mémoire

Quand utiliser l'allocation automatique (sur la **pile**) **?**

Si le tableau est utilisé seulement :

— dans la procédure où il est déclaré,

— ou dans une procédure appelée par celle-ci

```
int tab[n][p]
```
ou
```
INTEGER, DIMENSION(n,p)::tab
```

Quand utiliser l'allocation dynamique (sur le **tas**) **?**

Si le tableau est utilisé aussi dans la procédure appelant celle où il est défini

```
int **tab
```
ou
```
INTEGER, DIMENSION(:,:), ALLOCATABLE::tab
```

Attention : les deux types de déclarations ne sont pas équivalentes en C

$\Rightarrow$ ne pas mélanger les syntaxes.

Pour les tableaux dynamiques en C : penser à utiliser la bibliothèque **`libmnitab`**.

2 Rappels sur les entrées-sorties formatées

2.1 Fichiers formatés : syntaxe d'ouverture/fermeture

<table>
<tr><th>Ouverture d'un flot en C</th><th>connexion d'un fichier à une unité logique en f90</th></tr>
<tr><td colspan="2">FILE *fopen(const char *path,
const char *mode)</td><td>OPEN(UNIT=unit, FILE=filename &
[, STATUS=mode] [, IOSTAT=i]...)
FORM="formatted" si texte (défaut)</td></tr>
<tr><td>mode position
"r" read début
"w" write début
"a" append fin</td><td>ajouter
+ si mise à jour,
b si binaire</td><td>mode = "old"
"new"
"replace"</td></tr>
<tr><td colspan="2">rend NULL si erreur</td><td>IOSTAT$\neq$ 0 si erreur</td></tr>
<tr><td colspan="3">lectures ou écritures</td></tr>
<tr><td colspan="2">fermeture (et vidage du tampon)</td><td></td></tr>
<tr><td colspan="2">int fclose(FILE *stream)</td><td>CLOSE(unit[,IOSTAT=ivar,...])</td></tr>
</table>

2.2 Entrées-sorties standard

Affichage sur l'écran (par défaut) :

```
printf("fmt", liste_d'expressions);
fprintf(stdout,"fmt",liste_d'express.);
```

```
PRINT FMT, liste_d'express.
WRITE(*,FMT) liste_d'express.
```

Affichage sur la sortie d'erreur :

```
fprintf(stderr,"fmt",liste_d'express.);
```

```
WRITE(0,FMT) liste_d'express.
```

Lecture au clavier (par défaut) :

```
scanf("fmt", liste_de_pointeurs);
fscanf(stdin, "fmt", liste_de_point.);
```

```
READ FMT, liste_de_variables
READ(*, FMT) liste_de_var.
```

Sous UNIX, utiliser les redirections pour accéder à des fichiers :

< en entrée > en sortie

2.3 Règles de base pour changer de ligne

Langage C

Forcer les retour ligne par **\n**

Sauter une ligne

en sortie `printf("\n");`

en entrée `scanf("%*[^\n]\n")`

NB : `\n` ignoré si conversion numérique

Fortran (sauf si **`advance="no"`**)

Chaque instruction d'entrée/sortie
⇔ un changement de ligne

Sauter une ligne : avec une liste vide

en sortie `WRITE(*, *)`

en entrée `READ(*, *)`

2.4 Application aux tableaux 1D

Tableau 1D : (pas de distinction ligne/colonne)

```
int t[]={1,2,3};
```

```
INTEGER, DIMENSION(3):: t=[1,2,3]
```

Affichage en ligne

```
for (i=0; i<3; i++){
  printf("%d ",t[i]);
}
printf("\n");
```

⇒

```
1 2 3
```

⇐

```
WRITE(*,*) t(:)
```

Affichage en colonne

```
for (i=0; i<3; i++){
  printf("%d\n", t[i]);
}
```

⇒

```
1
2
3
```

⇐

```
DO i=1, 3
  WRITE(*,*) t(i)
END DO
```

2.5 Affichage de tableaux 2D

Tableau 2D : `mat` de profil `(nl, nc)` :

nl=2 **nc=3**

11	12	13
21	22	23

affichage **nl** lignes par **nc** colonnes

```
for (i=0; i<nl; i++){
  for (j=0; j<nc; j++){
    printf("%d ", t[i][j]);
  }
  printf("\n");
}
```

double boucle

```
DO i=1, nl
  WRITE(*,*) mat(i,:)
END DO
```

boucle sur les lignes

section de tableau = ligne

3 Compilation séparée des procédures

Découper le code en plusieurs fichiers sources
(une ou quelques procédures par fichier)
$\Longrightarrow$ séparer
(1) phase des **compilations** et
(2) phase de l'**édition de liens**

Rappel

unité de compilation	
langage C	Fortran
le fichier	le module

3.1 Intérêt de la compilation séparée des procédures

— modularisation du code
— mise au point plus rapide (ne recompiler que partiellement)
— réutilisation des procédures
— création de bibliothèques d'objets (collections de fichiers objets)
— automatisation de la compilation avec l'utilitaire **`make`**

3.2 Mise en œuvre robuste de la compilation séparée

Donner les moyens au **compilateur** de vérifier si **le nombre, le type** (correspondance de type en fortran, conversion en C) **et la position des arguments des procédures** lors d'un appel sont conformes à l'interface ou prototype de la procédure.

langage C	Fortran
passage par **copie** donc **conversion** des arguments sauf pour les pointeurs	passage par **référence** donc **respect exact** du type, mais généricité

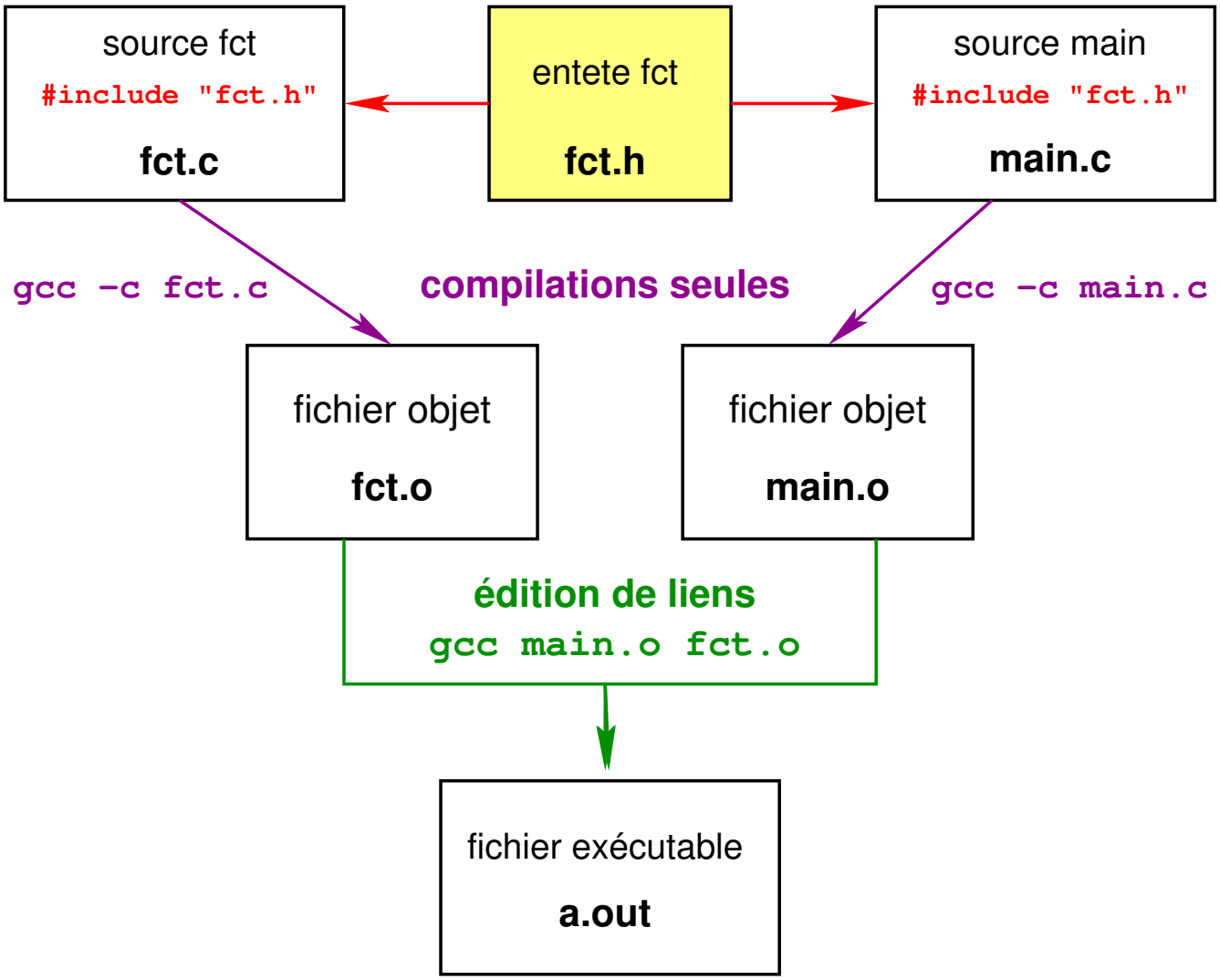
Première solution : dupliquer l'information sur l'interface en la déclarant au niveau des procédures qui l'utilisent.

langage C	Fortran
déclaration de **prototype**	déclaration d'**interface**

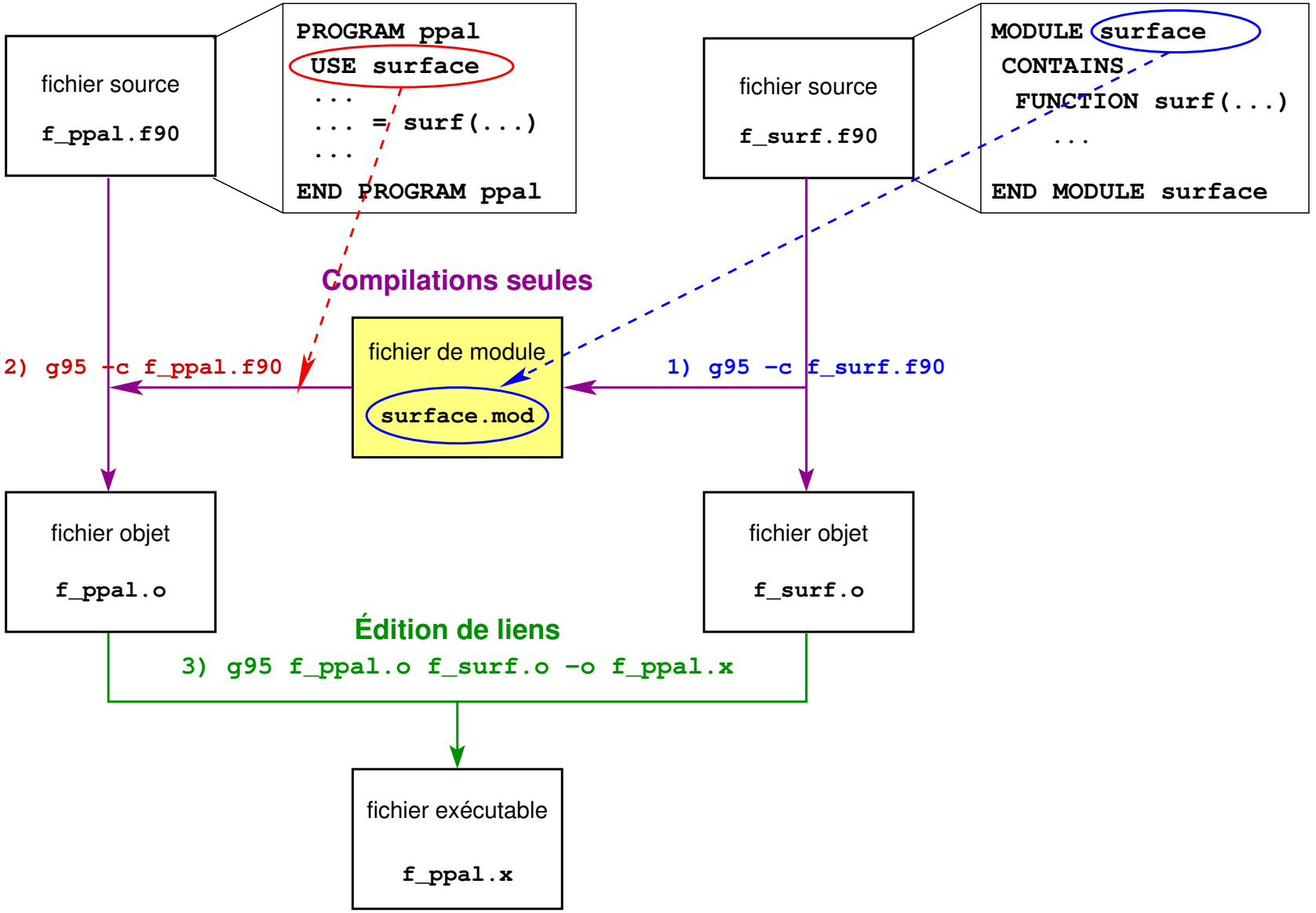
Mais risque d'incohérence entre **déclaration et définition**, en particulier dans la phase de développement $\Rightarrow$ méthode déconseillée.

$\Longrightarrow$ **Solution plus robuste**

langage C	Fortran
Déclarer les prototypes dans les fichiers d'**entête `*.h`** et les inclure à la fois : — dans la définition — et dans les fonctions appelantes.	Encapsuler les procédures dans des **modules** : le compilateur crée alors des fichiers **`.mod`** décrivant l'interface. L'instruction **`USE`** permet de relire ces interfaces quand on compile les appelants.
$\Rightarrow$ Éviter les déclarations multiples si plusieurs inclusions **`#ifndef MY_FCT_H`** **`#define MY_FCT_H`** **insérer ici le prototype** de `my_fct` **`#endif /*MY_FCT_H*/`**	$\Rightarrow$ Compiler les modules utilisés avant de compiler les appelants



Compilation séparée en C avec **fichier d'entête partagé** par appelé et appelant



Compilation séparée en fortran :

1) compilation du module 2) compilation de l'appelant 3) édition de liens

4 Procédures à argument fonction

4.1 Objectif et méthode

Mettre en œuvre des **méthodes fonctionnelles** (intégration, dérivation numérique, tabulation sur fichier, ...) via des procédures agissant sur des fonctions dites test.
Ces procédures fonctionnelles peuvent être assemblées dans une bibliothèque.

distinguer :
- **la fonction formelle** `fct` appelée quand on définit la procédure fonctionnelle : méthode `integ` pour intégration par exemple
- et **les fonctions effectives** `f1`, `f2`, ... arguments lors de l'appel de la procédure `y1=integ(f1, a, b), y2=integ(f2, a, b)`

— ne pas avoir à recompiler la méthode quand on change de fonction test.
⇒ la placer dans un **module** (fortran) ou un **fichier séparé** (C).

— les procédures doivent pouvoir s'appliquer à toute une catégorie de fonctions.
⇒ déclarer l'interface de la **fonction formelle** dans la procédure fonctionnelle.

— le choix de la fonction test **effective** se fait dans l'appelant.

4.2 Structure minimale du code

Programme principal

Visibilité de l'interface des fonctions tests effectives

```
#include "fonctions.h"                    USE m_fonctions
```

Visibilité de l'interface de la méthode fonctionnelle

```
#include "methode.h"                      USE m_methode
```

Appels de la méthode `trait` appliquée aux fonctions tests `f1` et `f2`

```
trait(&f1, ...) ;                 CALL trait(f1, ...)
trait(&f2, ...) ;                 CALL trait(f2, ...)
```

Définition des fonctions tests

Fichiers des fonctions

déclarations f1f2.h

```
float f1(float t);
float f2(float t);
```

définitions f1f2.c

```
...
#include "f1f2.h"
float f1(float t){
...
}
float f2(float t){
...
}
```

Module des fonctions

fcts.f90

```
MODULE m_fonctions
CONTAINS
! définition de f1 et f2
  REAL FUNCTION f1(t)
     ...
  END FUNCTION f1

  REAL FUNCTION f2(t)
     ...
  END FUNCTION f2
END MODULE m_fonctions
```

Méthode s'appliquant aux fonctions

```
/* fichier methode.c */
#include "methode.h"
void trait(float (*pfct)(float),
           ...){
```

Déclaration de l'interface de la fonction formelle comme argument dans le prototype de la méthode via un **pointeur de fonction**

```
...
```

appel de la fonction formelle

```
... = ... (*pfct)(u)...
}
/* fin du fichier methode.c */
```

```
MODULE m_methode
CONTAINS
  SUBROUTINE trait(fct, ...)
```

Déclaration de l'interface de la fonction formelle

```
    INTERFACE
      REAL FUNCTION fct(x)
        ...
      END FUNCTION fct
    END INTERFACE
    ...
```

appel de la fonction formelle

```
    ... = ... fct(u) ...
  END SUBROUTINE trait
END MODULE m_methode
```

4.3 Exemple de passage de fonction en argument

4.3.1 Fortran

ppal.f90

```
PROGRAM ppal     ! le programme principal
USE m_affiche    ! module traitement d'une fct réelle qcq d'arg. réel
USE m_fonctions ! module des fonctions tests effectives
IMPLICIT NONE
REAL :: x0, x1, pas
REAL, DIMENSION(:), ALLOCATABLE :: x
INTEGER:: n, i
WRITE(*,*) 'entrer le min, le max et le nb de pas'
READ(*,*) x0, x1, n
ALLOCATE(x(n))
pas = (x1 - x0) / REAL(n-1)
DO i=1, n
  x(i) = x0 + pas * (i-1)
ENDDO
WRITE(*,*) 'tabulation de f1 entre', x0, ' et ', x1
CALL affiche(f1, x) ! procédure appliquée à f1
WRITE(*,*) 'tabulation de f2 entre', x0, ' et ', x1
CALL affiche(f2, x) ! procédure appliquée à f2
DEALLOCATE(x)
END PROGRAM ppal
```

fonctions.f90

```
MODULE m_fonctions ! définition des fonctions effectives
IMPLICIT NONE
CONTAINS
  FUNCTION f1(t)      ! f1 = double
    REAL :: f1
    REAL, INTENT(in) :: t
    f1 = 2.* t
    RETURN
  END FUNCTION f1
  FUNCTION f2(t)      ! f2 = triple
    REAL :: f2
    REAL, INTENT(in) :: t
    f2 = 3.* t
    RETURN
  END FUNCTION f2
END MODULE m_fonctions
```

affiche.f90

```
MODULE m_affiche ! module des procédures fonctionnelles
IMPLICIT NONE
CONTAINS ! les procédures applicables à des fcts qcq
SUBROUTINE affiche(fct, xx) ! par exemple proc. affiche
  REAL, DIMENSION(:), INTENT(in) :: xx
  INTERFACE  ! déclaration de la fonction passée en arg
    FUNCTION fct(t)
      REAL                :: fct ! type du résultat
      REAL, INTENT(in) :: t    ! type de l'argument
    END FUNCTION fct
  END INTERFACE
  INTEGER :: i
  DO i=1, SIZE(xx) ! aff. les val. de fct aux pts de xx
    WRITE(*,*)  xx(i), fct(xx(i))
  ENDDO
END SUBROUTINE affiche
END MODULE m_affiche
```

Variante : interface abstraite et déclaration PROCEDURE en fortran

Déclaration dans un module d'une interface commune à plusieurs procédures

utiliser **ABSTRACT INTERFACE** pour nommer l'interface (**fctrr** ici)

abstract-interf.f90

```
MODULE m_abstract_interf ! décl. d'interface abstraite
! après un USE de ce module
! PROCEDURE(fctrr) :: f1, f2
! permet de déclarer pls fcts respectant cette interface
 IMPLICIT NONE
 ABSTRACT INTERFACE  ! interface abstraite des fonctions
   FUNCTION fctrr(t) ! nommer fctrr ce type de fonction
     REAL              :: fctrr ! type du résultat
     REAL, INTENT(in)  :: t     ! type de l'argument
   END FUNCTION fctrr
 END INTERFACE
END MODULE m_abstract_interf
```

L'interface abstraite est référencée par **PROCEDURE(fctrr)::**

pour déclarer des procédures de ce type

affiche.f90

```
MODULE m_affiche ! module des procédures fonctionnelles
IMPLICIT NONE
CONTAINS ! les procédures applicables à des fcts qcq
SUBROUTINE affiche(fct, xx) ! par exemple proc. affiche
  USE m_abstract_interf ! qui définit l'interface fctrr
  REAL, DIMENSION(:), INTENT(in) :: xx
  PROCEDURE(fctrr) :: fct
  INTEGER :: i
  DO i=1, SIZE(xx) ! aff. les val. de fct aux pts de xx
    WRITE(*,*)  xx(i), fct(xx(i))
  ENDDO
END SUBROUTINE affiche
END MODULE m_affiche
```

4.3.2 Langage C

main.c

```c
#include <stdio.h>
#include <stdlib.h>
/* entêtes personnelles nécessaires dans le main */
#include "float1d.h" /* création de vecteurs sur le tas */
#include "affiche.h" /* la méthode : tabulation de fonction */
#include "f1f2.h"    /* les fonctions test */

int main(void){
  float x0, x1, pas;
  float * x = NULL; /* pointeur sur tableau des abscisses */
  int n, i;
  /* saisie des paramètres */
  fprintf(stderr, "entrer min, max et le nb de pts\n");
  scanf("%f %f %d", &x0, &x1, &n);
```

```
  x = float1d(n); /* allocation tableau 1D des abcisses */
  pas = (x1 - x0) / (float) (n-1);
  for(i=0; i<n; i++){ /* création grille des abscisses */
     x[i] = x0 + pas * i;
  }
  printf("tabulat. de f1 entre %f et %f\n", x[0], x[n-1]);
  affiche(&f1, x, n);  /* appel de la procédure pour f1 */
  printf("tabulat. de f2 entre %f et %f\n", x[0], x[n-1]);
  affiche(&f2, x, n);  /* appel de la procédure pour f2 */
  free(x); /* libération du tableau x */
  x = NULL;
  exit(EXIT_SUCCESS);
}
```

float1d.h

```
#ifndef FLOAT1D_H
float * float1d(int n);
#define FLOAT1D_H
#endif /* FLOAT1D_H */
```

float1d.c

```
#include <stdlib.h>
#include "float1d.h"

float * float1d(int n){
  float * ptf=NULL;
  if (n >0) {
     ptf = (float *) calloc(n, sizeof(float)) ;
  }
  if (ptf==NULL) exit(EXIT_FAILURE);
  return ptf;
}
```

Les fonctions test

f1f2.h (déclarations)

```
#ifndef F1F2_H
float f1(float t);
float f2(float t);
#define F1F2_H
#endif /* F1F2_H */
```

f1f2.c (définitions)

```
#include "f1f2.h"
float f1(float t){
  return 2.f*t;
}
float f2(float t){
  return 3.f*t;
}
```

La méthode

affiche.h (déclaration)

```
#ifndef AFFICHE_H
/* prototype de la méthode */
void affiche( float (* pfct )(float), float xx[], int n ) ;
/* premier argument = pointeur vers une fonction test */
#define AFFICHE_H
#endif /*  AFFICHE_H */
```

affiche.c (définition)

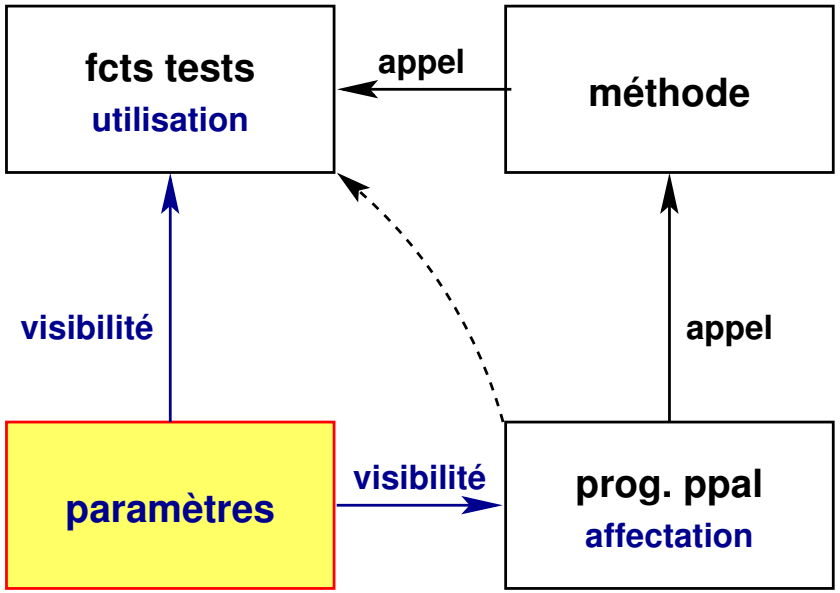
```
#include <stdio.h>
/* entêtes personnelles nécessaires dans affiche.c */
#include "affiche.h" /* déclaration de la fonction affiche */


void affiche( float (* pfct )(float), float xx[], int n ) {
          /* argument fonction => pointeur vers une fonction */
  int i ;
  /* impression des valeurs de la fonction fct aux points de xx */
  for(i=0; i<n; i++){ /* affichage */
     printf("%f %f\n", xx[i], (*pfct)(xx[i])) ;
  }
  return;
}
```

4.4 Variables globales

4.4.1 Nécessité des variables globales

Quand une **méthode fonctionnelle** (intégration, dérivation, ...) est appliquée à une fonction test, on intègre ou dérive par rapport à **une variable.**
Or cette fonction peut **dépendre d'autres paramètres** que la méthode n'a pas à connaître, mais que la fonction utilise quand on l'évalue.
Ces paramètres peuvent être par exemple saisis dans le programme principal et transmis à la fonction test qui n'est appelée que via la méthode.
La seule solution pour les transmettre est alors d'en faire des **variables globales** et de ne pas leur accorder de visibilité dans la méthode.



4.4.2 Risque de masquage

Veiller à **ne pas redéclarer les variables globales** sous peine de les **masquer** par des homonymes locaux $\Rightarrow$ utiliser l'option **`-Wshadow`** du compilateur

4.4.3 Variables globales en fortran

Paramètres déclarés dans le module des fonctions, mais en dehors des fonctions du module qui y ont cependant accès.
`use fcts` $\Rightarrow$ visibles **sans redéclaration** dans le programme principal qui leur affecte des valeurs.

Le module de méthodes n'aura pas de visibilité sur le module des fonctions effectives, donc il ne verra pas ces paramètres.

4.4.4 Variables globales en C

Rappel : ne pas confondre

— Déclaration simple : sans initialisation
— Définition : toute déclaration avec initialisation est une définition

En compilation séparée, la portée d'une variable globale est limitée au fichier.
Mais une déclaration avec le qualificatif **`extern`** indique la **redéclaration** d'une variable définie dans un autre fichier (en fait sa référence).

Les paramètres doivent être déclarés comme variables globales partagées entre le programme principal qui peut les définir et les fonctions à tester qui les redéclarent.

Une solution possible :
Utiliser le préprocesseur pour inclure deux fichiers presque identiques :

— un fichier contenant les définitions de ces paramètres dans le fichier de la fonction `main` qui leur affectera des valeurs ;
— un fichier contenant les redéclarations de ces paramètres (avec **`extern`**) dans celui définissant les fonctions.

5 Utilitaire `make` et bibliothèques

5.1 Génération d'applications avec `make`

5.1.1 Principe

La commande **`make`** permet d'**automatiser** la génération et la mise à jour d'applications ou **cibles** (target) qui **dépendent** d'autres fichiers (prérequis) : **`make`** applique des **commandes unix** constituant des **recettes** (recipes) de construction. **`make`** minimise les opérations de mise à jour en s'appuyant sur les **dates** de modification des fichiers et les **règles** (rules) de dépendance :

— **cible** (target) : en général un fichier à produire (par ex. un exécutable)

— **dépendances** : ensemble des fichiers nécessaires à la production d'une cible

— **recette** (recipe) : liste des commandes unix à exécuter pour construire une cible (compilation pour les fichiers objets, édition de liens pour l'exécutable)

— **règle** (rule) : ensemble cible + dépendances + recette

L'utilisateur doit décrire les règles de son projet dans un fichier **`makefile`**.

Application la plus classique : reconstituer un programme exécutable à partir des fichiers sources en ne recompilant que ceux qui ont été modifiés.

5.1.2 Utilisation élémentaire de **make**

par défaut un projet et un fichier **makefile** par répertoire.

make *cible*

lance la production de la ***cible*** en exploitant le fichier **makefile** du répertoire courant.

make -n *cible*

affiche les commandes que devrait lancer **make** pour produire la ***cible***

make

lance la production de la **première** cible du fichier **makefile**

L'option **-f *fichier*** de **make** permet de spécifier le nom du fichier décrivant les règles : **make -f Makefile-c99**

make permet de gérer au mieux l'espace disque : prévoir une cible de nettoyage pour supprimer les fichiers qui peuvent être reconstruits.
NB : cette cible n'est pas un fichier $\Rightarrow$ la déclarer **.PHONY**

5.1.3 Construction du fichier `makefile`

Première étape : décrire, les relations entre les fichiers via un fichier **`makefile`** qui liste les règles de reconstruction en répondant aux questions :

Fabriquer quoi ? (**cible**) quand ? (**prérequis** plus récents) et comment ? (**règle**) .

Le fichier **`makefile`** est construit à partir de l'**arbre des dépendances**.

Syntaxe des règles du fichier **`makefile`** :

```
cible: liste des dépendances
(tabulation) commandes de construction   (en shell)
```

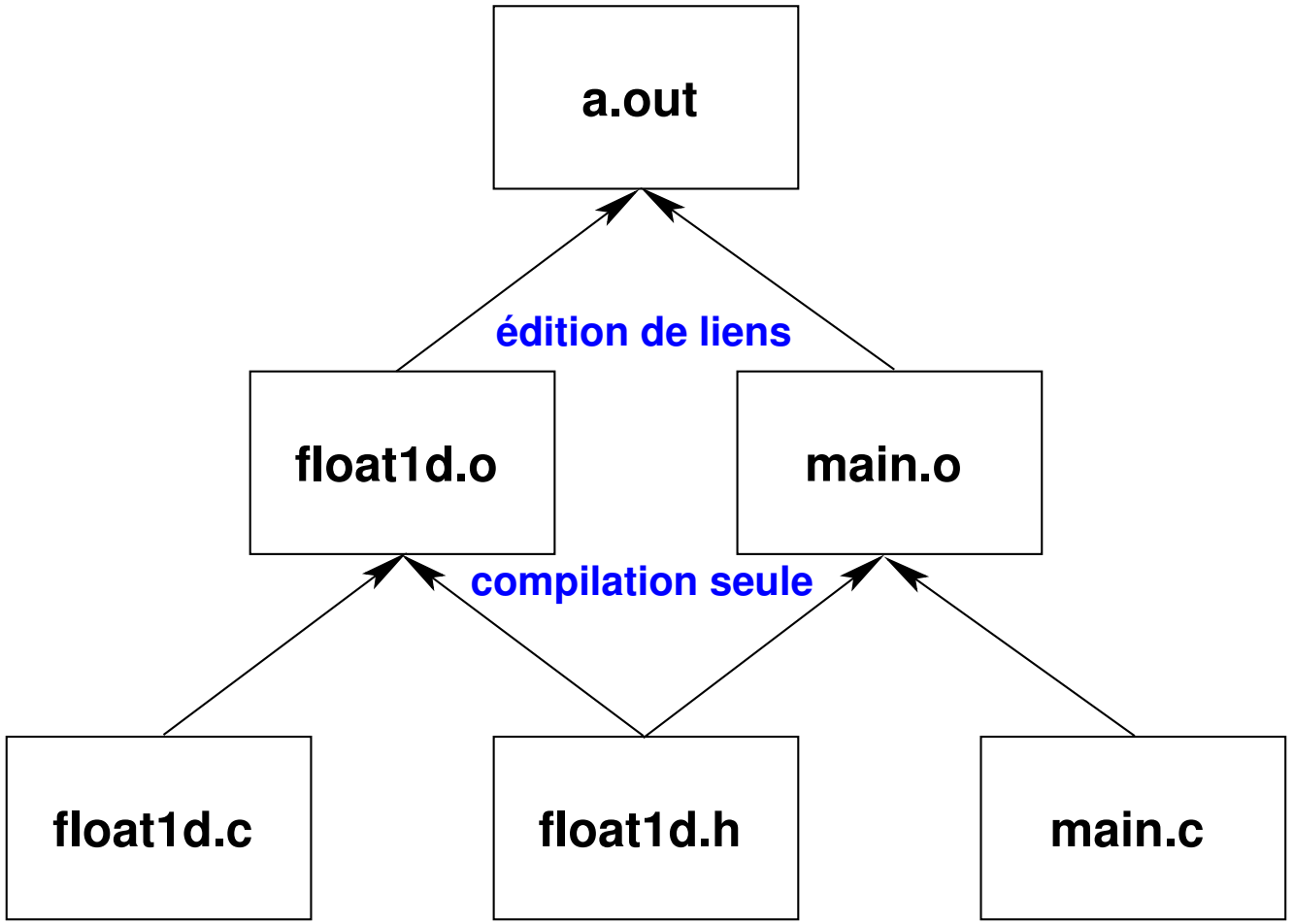
Analyse récursive des dépendances via les règles : pour construire une cible, **`make`** analyse ses dépendances :

— si une dépendance est elle-même la cible d'une autre règle,
cette autre règle est évaluée avec ses dépendances,... ainsi récursivement.

— si une dépendance est **plus récente que la cible** ou si la cible n'existe pas,
la reconstruction est lancée.

5.1.4 Exemple élémentaire de `makefile` pour application en C

Arbre des dépendances (exploré récursivement par `make`)



Makefile

```
# première cible = celle par défaut : l'exécutable
# règle pour l'exécutable : recette = édition de liens
a.out : float1d.o main.o
	gcc float1d.o main.o

# règles pour les objets : recette = compilation seule
float1d.o : float1d.c float1d.h
	gcc -c float1d.c
main.o : main.c float1d.h
	gcc -c main.c

# pour éviter un problème si un fichier "clean" existe
.PHONY: clean
# ménage : suppression des fichiers reconstructibles
clean:
	/bin/rm -f a.out *.o
```

Exemple d'utilisation avec les commandes `make` suivantes (dans l'ordre) :

1. **`make clean`** supprime les fichiers objets et l'exécutable
2. **`make`** ou **`make a.out`** lance la reconstruction de l'exécutable
   qui dépend des objets... supprimés
   ⇒ les reconstruire ⇒ évaluer les dépendances des objets,
   qui dépendent des sources et des entêtes.
   si ces fichiers sont présents, on lance la compilation
   si un fichier source ou entête manque, **`make`** signale une erreur
   Retour à la cible initiale
   ⇒ Édition de liens pour produire l'exécutable `a.out`
3. Modification du source **`main.c`**, puis **`make`**
   `a.out` dépend de `main.o`,
   qui dépend de `main.c`, qui est plus récent
   ⇒ recompilation de `main.c` ⇒ `main.o` à jour
   ⇒ `main.o` plus récent que `a.out` ⇒ édition de liens.
   NB : pas de recompilation de `float1d.c`

Aides à la construction du makefile

gcc -MM fichier.c affiche les dépendances de fichier.o

(s'appuie sur les #include *.h)

gfortran -cpp -M fichier.f90 ($\geqslant$ v4.6) (nécessite les .mod)

5.1.5 Variables interprétées par make, makefile paramétré

Possibilité de **définir des variables ou macros**, référence avec **$(**MACRO**)**

Paramétrage des outils (compilateur par exemple)

CC = gcc choix du compilateur C

CFLAGS = -W -Wall options du compil. C (alias non honorés dans make)

```
float1d.o : float1d.c float1d.h
__TAB__$(CC) $(CFLAGS) -c  float1d.c
```

Directive **include fichier.mk** pour inclure un fichier dans un makefile

Listes de fichiers

```
SRCS = main.c float1d.c
```

ou, en protégeant les changements de lignes (rien après le \)

```
SRCS = \
       main.c \
       float1d.c
```

Substitution de suffixe dans une liste (transformations textuelles)

```
OBJS = $(SRCS:.c=.o)   ⇒   main.o float1d.o
```

```
RM = /bin/rm -f
clean :
__TAB__$(RM) a.out $(OBJS)
```

Macros prédéfinies (ou variables automatiques) utilisables dans les règles :

⇒ **$@** le nom de la **cible** courante

⇒ **$<** le nom de **la première dépendance** permettant la génération de la cible dans le cas d'une règle implicite

— **$?** la liste des dépendances plus récentes que la cible

— **$^** la liste de **toutes les dépendances**

— **$*** le préfixe commun du nom de fichier de la cible courante et de la dépendance dans les règles par suffixe

Documentation sur `make`

`http://www.gnu.org/software/make/manual/make.html`

Exemples avec variables automatiques

Compilation

```
float1d.o : float1d.c float1d.h
__TAB__gcc -c $< -o $@
```

$< signifie première dépendance seulement, donc en développant

```
__TAB__gcc -c float1d.c -o float1d.o
```

Édition de liens

```
a.out : float1d.o main.o
__TAB__gcc $^ -o $@
```

$^ signifie toutes les dépendances, donc en développant

```
__TAB__gcc float1d.o main.o -o a.out
```

5.1.6 Règles associées à des suffixes

Des **règles implicites** génériques s'appuyant sur les suffixes des fichiers permettent d'automatiser la création des cibles les plus classiques.

```
.c.o: # une seule dépendance (attention à l'ordre !)
__TAB__gcc -c $< -o $@

%.o: %.c %.h # syntaxe plus riche (dépendances multiples)
__TAB__gcc -c $< -o $@
```

Liste des suffixes et règles génériques peuvent être complétées[a] dans le **makefile**.

```
.SUFFIXES: .tex .pdf
%.pdf: %.tex
__TAB__pdflatex $<
```

a. Attention: le suffixe `.mod` est associé par défaut non aux fichiers de module du fortran, mais aux fichiers source en langage modula-2 pour lesquels il existe des règles implicites, notamment de compilation de `.mod` vers `.o`. Il est possible d'ignorer les règles et suffixes implicites avec `make -r`.

5.2 Bibliothèques statiques de fichiers objets

— **Intérêt :** regrouper dans **un seul fichier** toute une collection de **fichiers objets** de procédures compilées pour simplifier les futures commandes d'édition de liens qui utilisent ces procédures.

— **Interfaces :** regrouper les interfaces de toutes les procédures de la bibliothèque dans un seul fichier (**`.mod`** en fortran, **`.h`** en C).

1. compiler les procédures à insérer dans la bibliothèque
   `gcc -c float1d.c float1d_libere.c`
2. insérer les objets dans la bibliothèque (statique)
   `ar` `rv` **`libtab.a`** `float1d.o float1d_libere.o`
3. puiser dans la bibliothèque lors de l'édition de liens
   `gcc main.c -ltab` (si `libtab.a` est dans le chemin de recherche)
   `gcc main.c libtab.a` (si on donne explicitement le nom du fichier)

5.2.1 Commande de gestion des bibliothèques statiques : ar

Actions principales de **ar** (syntaxe proche de **tar** sauf option f) :

→ **r** ajout ou **r**emplacement d'une liste de membres

```
ar rv libtab.a double1d.o double1d_libere.o
```

→ **t** lis**t**e les membres de la bibliothèque

```
ar tv libtab.a
rw-r--r-- 904/800    860 Jan 29 11:17 2008 double1d.o
rw-r--r-- 904/800    808 Jan 29 11:17 2008 double1d_libere.o
```

→ **x** e**x**traction d'une liste de membres

```
ar xv libtab.a double1d.o
```

→ **d** **d**estruction d'une liste de membres

```
ar dv libtab.a double1d_libere.o
```

→ **v** option (**v**erbose) avec messages d'information

→ **u** option (**u**pdate) mise à jour seulement ⇒ **ar ruv**

→ **s** génère un index des **s**ymboles définis par les membres de l'archive

5.2.2 Utilisation d'une bibliothèque statique

1) Pour la **compilation**, dans le code source utilisant la bibliothèque, les prototypes des fonctions sont déclarés via **`#include "tab.h"`** en C
   et les interfaces des procédures sont déclarées via **`USE tab`** en fortran
2) Lors de l'**édition de liens**, le code objet des procédures est extrait de l'archive **`libtab.a`** $\Rightarrow$ option **`-ltab`** placée **après** le fichier appelant

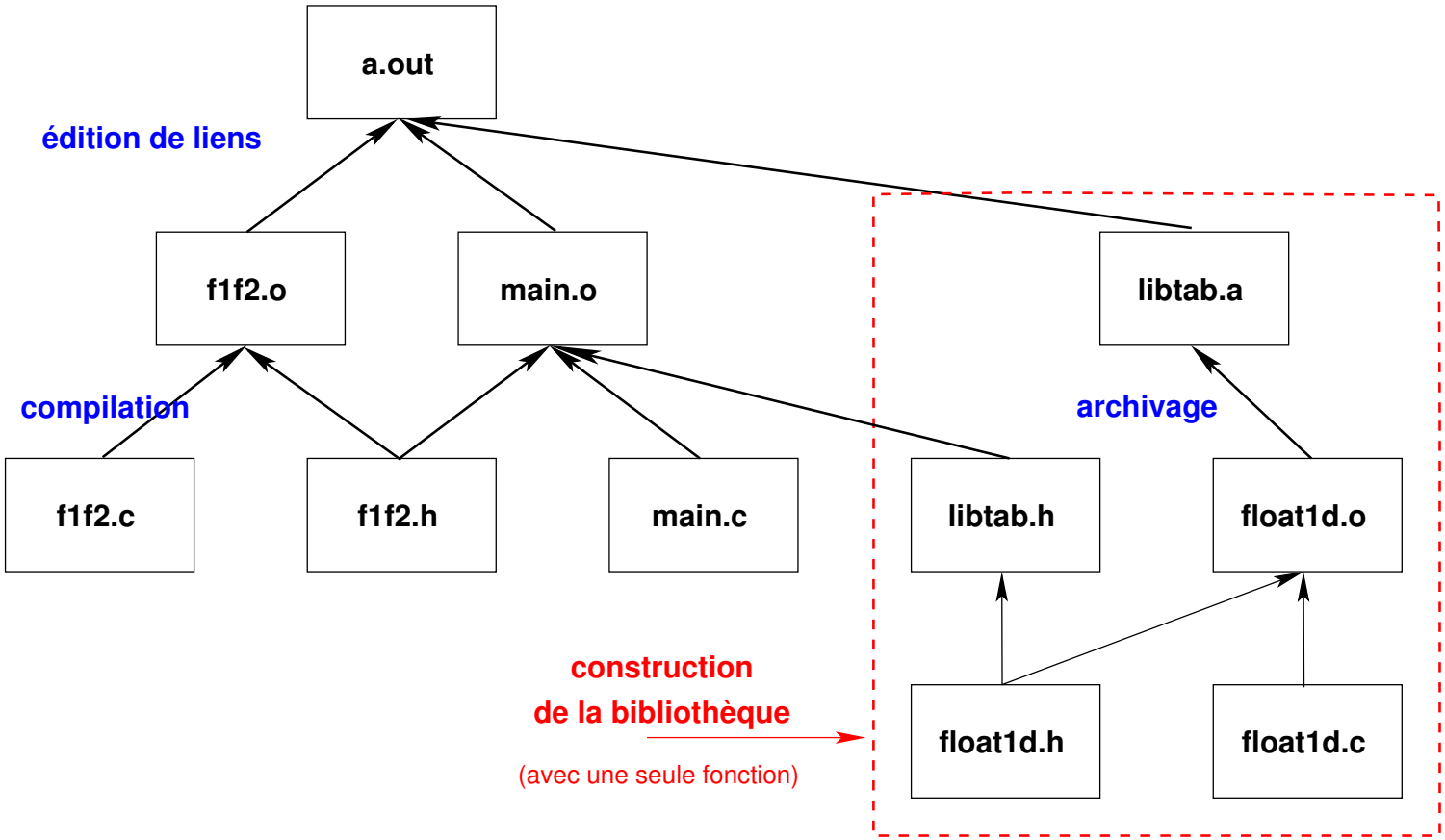
Si la bibliothèque est installée par l'utilisateur `user` dans des répertoires non-standard, **`/home/user/include`** et **`/home/user/lib`**, il faut compléter la liste des chemins d'accès avec les options **`-I`** et **`-L`** :

1) **`-I`** pour le répertoire où se trouve l'entête **`tab.h`** en C
   ou le fichier de module **`tab.mod`** en fortran
2) **`-L`** pour le répertoire où se trouve l'archive **`libtab.a`**

```
gcc -I/home/user/include -L/home/user/lib main.c -ltab
gfortran -I/home/user/include -L/home/user/lib \
         main.f90 -ltab
```

5.2.3 Exemple de `makefile` en C avec bibliothèque

Arbre des dépendances comportant une bibliothèque (représentation partielle).



Makefile avec utilisation de la bibliothèque

```
# version sans création de la bibliothèque
# libtab.h et libtab.a sont supposés exister
CC = gcc # compilateur C
CFLAGS = -std=c99 -W -Wall -Wextra -Wshadow # options compil. C
LDFLAGS = # options pour l'édition de liens en C (ex -L~/lib/)
# règle de compilation pour le C (avec dépendance des headers)
%.o: %.c %.h
__TAB__$(CC) -c $(CFLAGS) $< -o $@
# règle de compilation pour le C (sans dépendance des headers)
# utilisée seulement en l'absence de header
#    (sinon: règle implicite)
.c.o :
__TAB__$(CC) -c $(CFLAGS) $< -o $@
# liste des sources (hors bibliothèque)
SRCS =  main.c affiche.c  f1f2.c
```

suite

```
# liste des objets
# (noms déduits des sources en remplaçant .c par .o)
OBJS = $(SRCS:.c=.o)
# première cible = celle par défaut = l'exécutable a.out
a.out : $(OBJS) libtab.a
	$(CC) $(LDFLAGS) $(OBJS) -L. -ltab
$(OBJS) : libtab.h # pas nécessaire pour tous les objets !

main.o: main.c libtab.h affiche.h f1f2.h

# pour éviter un problème si un fichier "clean" existe
.PHONY: clean
# suppression des fichiers reconstructibles
clean:
	/bin/rm -f a.out $(OBJS)
```

6 Gestion des changements de précision

Nécessité de techniques rapides pour passer de simple en double précision en modifiant les codes au minimum.

— solutions très différentes selon le langage ;

⚠ — supposent de déclarer les dépendances dans le fichier **`makefile`**

6.1 Changement de précision en fortran

6.1.1 Méthode rapide mais déconseillée

Option de compilation pour promouvoir les **`REAL` par défaut** en **`DOUBLE PRECISION`** (sur 8 octets)

`gfortran` : **`-fdefault-real-8`** (variables et constantes)

`g95` : **`-r8`**

Mais dépend du compilateur

6.1.2 Méthode plus portable conseillée

Définir les variantes des réels utilisables en fonction de la précision dans un module

mncs_kinds.f90

```
MODULE mncs_kinds
IMPLICIT NONE
INTEGER, PARAMETER :: sp=SELECTED_REAL_KIND(p=6) ! 6 chiffres => 32 bits
! INTEGER, PARAMETER :: sp=KIND(1.e0) ! simple précision
INTEGER, PARAMETER :: dp=SELECTED_REAL_KIND(p=15) ! 15 chiff. => 64 bits
! INTEGER, PARAMETER :: dp=KIND(1.d0) ! double précision
END MODULE mncs_kinds
```

et choisir la précision de travail (*working precision*) dans le module **mncs_precision**

mncs_precision.f90

```
MODULE mncs_precision
USE mncs_kinds ! visibilité des variantes sp et dp
! choix de la précision de travail : variante wp
!INTEGER, PARAMETER :: wp = sp ! simple precision
INTEGER, PARAMETER :: wp = dp ! double precision
END MODULE mncs_precision
```

Puis inclure un **`USE mncs_precision`** dans tous les codes ($\Rightarrow$ dépendance pour `make`) et déclarer **tous les réels avec la variante de type `wp`**.

```
USE mncs_precision
...
REAL(kind=wp)  :: hmin, hmax
```

⚠ Ne pas oublier de préciser la variante de type des **constantes réelles** par **`_wp`**

```
USE mncs_precision
...
REAL(KIND=wp), PARAMETER :: k = .1_wp
```

Pas de problème pour les entrées-sorties en format libre.

Mais adapter les formats s'ils sont précisés.

⚠ **Interfaces :** utiliser **`import wp`** pour donner la visibilité de **`wp`** dans l'interface

6.2 Changement de précision en C

Pas de changement pour les fonctions mathématiques si **`tgmath.h`**

6.2.1 Méthode utilisant `#define`

mncs_precision.h

```
#ifndef MNCS_PRECISION_H
#define MNCS_PRECISION_H
#define REAL double   /* surtout sans ; */
#endif /* MNCS_PRECISION_H */
```

Puis déclarer les flottants de type **REAL**

```
#include "mncs_precision.h"
...
REAL h, hmin, hmax;
```

6.2.2 Méthode utilisant `typedef`

mncs_precision.h

```
#ifndef MNCS_PRECISION_H
#define MNCS_PRECISION_H
typedef double real; /* ou typedef float real; avec ; */
#endif /* MNCS_PRECISION_H */
```

⚠ Tous les objets dépendent de **`mncs_precision.h`** ⇒ à préciser dans **`makefile`** (sinon risque de non-recompilation au changement de précision)

Puis déclarer les flottants de type **`real`**

Préciser le type des constantes flottantes (`1.5` si double, `1.5f` si float)

```
#include "mncs_precision.h"
...
real h, hmin, hmax;
```

— **Formats d'entrée** pour **`scanf`** : (modifier ou mieux) dupliquer les variables : une variable **`float`** pour saisie et copier sa valeur dans la variable **`real`**

— **Formats de sortie :** ne pas modifier car conversion en double par **`printf`**

7 Erreurs de troncature et d'arrondi : exemple de la dérivation numérique

Schéma aux différences finies **centré à deux termes** pour estimer

la dérivée première d'une fonction f en x_0 :

$$f'(x_0) \approx f_1(x_0) = \frac{f(x_0+h) - f(x_0-h)}{2h}$$

Deux types d'erreurs indépendantes s'ajoutent en valeur absolue :

— l'**erreur systématique de troncature** de valeur absolue e_t liée au **nombre fini de termes** dans l'estimateur.
Dériver $\Longleftrightarrow$ multiplier par $\boldsymbol{ik}$ dans l'espace de Fourier,
Dérivation numérique centrée à 2 termes $\Longrightarrow \times \boldsymbol{i \sin(kh)/h}$

— l'**erreur aléatoire d'arrondi** de valeur absolue e_a liée à la **représentation approximative des flottants** en machine et essentiellement due au calcul de la différence.

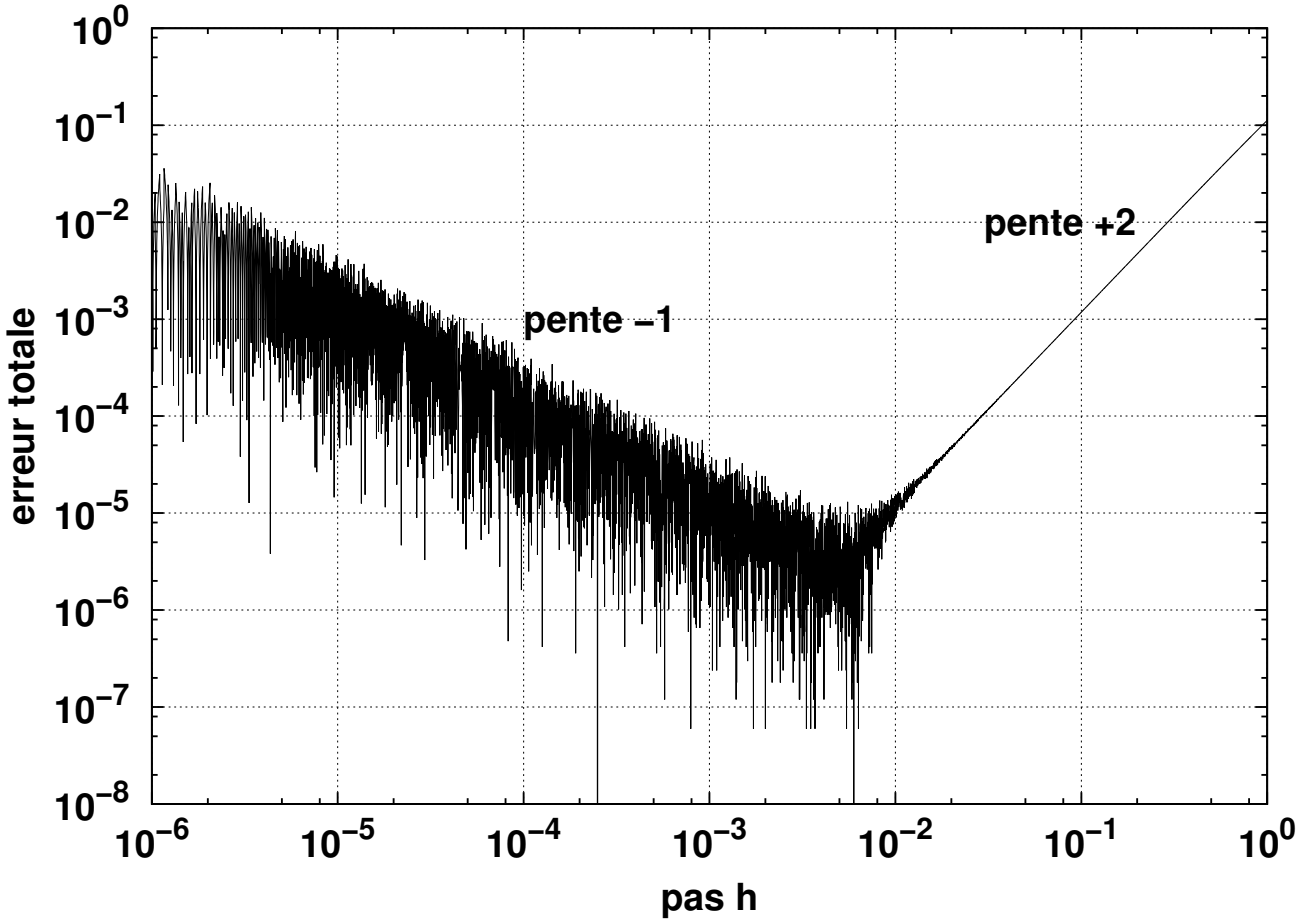


FIGURE 1 – **Erreur totale** e de l'estimateur à deux termes de la dérivée première de la fonction sinus au point $x = \pi/4$ en fonction du pas h **en échelle log-log**.

7.1 Estimation de l'erreur de troncature

Développement en série de Taylor avec reste de $f(x \pm h)$ au deuxième ordre autour de x :

$$f(x+h) = f(x) + h\frac{\mathrm{d}f}{\mathrm{d}x}(x) + \frac{h^2}{2}\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}(x) + \frac{h^3}{6}\frac{\mathrm{d}^3 f}{\mathrm{d}x^3}(x+\theta h)$$

$$f(x-h) = f(x) - h\frac{\mathrm{d}f}{\mathrm{d}x}(x) + \frac{h^2}{2}\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}(x) - \frac{h^3}{6}\frac{\mathrm{d}^3 f}{\mathrm{d}x^3}(x-\theta' h)$$

Estimateur centré à 2 termes de la dérivée :

$$\frac{f(x+h) - f(x-h)}{2h} = f'(x) + \frac{h^2}{12}\left[f'''(x+\theta h) + f'''(x-\theta' h)\right]$$

L'erreur absolue de troncature :

$$e_t \approx \frac{\boldsymbol{h^2}}{6}|f'''(x)|$$

7.2 Estimation de l'erreur d'arrondi

Chacun des termes de la différence est représenté avec une précision relative ε imposée par le nombre de bits de la mantisse donc le type de flottant.

$\varepsilon =$ `EPSILON(1.)` $=$ `FLT_EPSILON` $= 2^{-23} \approx 1,2\,10^{-7}$ sur 32 bits

$\varepsilon =$ `EPSILON(1.D0)` $=$ `DBLE_EPSILON` $= 2^{-52} \approx 2,2\,10^{-16}$ sur 64 bits

$$|\delta_a f(x+h)| \approx |\delta_a f(x-h)| \leqslant \varepsilon |f(x)|$$

Erreur absolue d'arrondi sur $f_1(x)$ majorée par e_a :

$$\delta_a \left[\frac{f(x+h) - f(x-h)}{2h} \right] \leqslant e_a = \frac{2\varepsilon |f(x)|}{2\boldsymbol{h}}$$

7.3 Comparaison des erreurs pour schéma centré à 2 termes

Erreur d'arrondi

$$e_a \propto h^{-1}$$

Pente en log-log -1

Dominante pour h faible

Erreur de troncature

$$e_t \propto h^2$$

Pente en log-log $+2$

Dominante pour h grand

Majorant de l'erreur absolue totale e

$$e \leqslant \frac{\varepsilon|f(x)|}{h} + \frac{h^2}{6}|f'''(x)| = e_m$$

Les deux erreurs varient en sens inverse selon le pas h

$\Rightarrow$ **compromis** nécessaire pour minimiser la somme des erreurs

7.4 Recherche du minimum de l'erreur totale

7.4.1 Méthode analytique

$\tilde{h}$ qui minimise l'erreur totale $= \sqrt[3]{3\varepsilon \left| \dfrac{f(x)}{f'''(x)} \right|} \quad \Rightarrow \quad e(\tilde{h}) = |f(x)| \sqrt[3]{\dfrac{9\varepsilon^2}{8} \left| \dfrac{f'''(x)}{f(x)} \right|}$

Cas de $\sin(x)$ pour $x = \pi/4$,
$|f(x)| = |f^{(n)}(x)| = 1/\sqrt{2} \quad \Rightarrow \quad e(\tilde{h}) = \frac{\sqrt[3]{9\varepsilon^2}}{2\sqrt{2}}$

7.4.2 Recherche graphique approximative

$\check{h}$ à l'intersection des droites en log-log $\quad \Rightarrow \quad \check{h} = \sqrt[3]{6\varepsilon \left| \dfrac{f(x)}{f'''(x)} \right|} = \tilde{h}\sqrt[3]{2} \approx 1.26\tilde{h}$

$$e(\check{h}) = |f(x)| \sqrt[3]{\frac{4\varepsilon^2}{3} \left| \frac{f'''(x)}{f(x)} \right|}$$

7.5 Influence de la précision : réduction de l'erreur d'arrondi

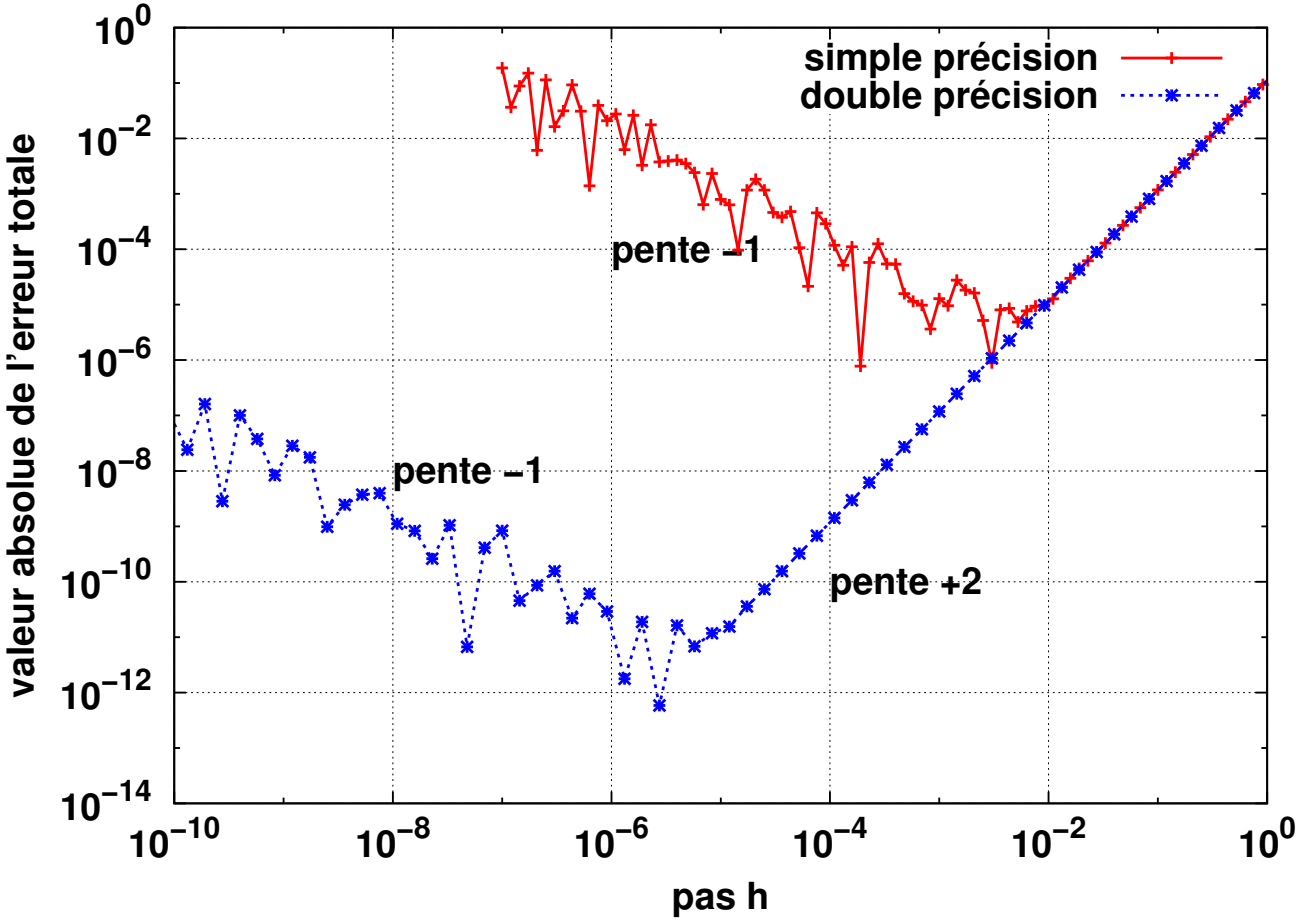


FIGURE 2 – Erreur totale (valeur absolue) $|e|$ en simple et double précision de l'estimateur à 2 termes de la dérivée première en fonction du pas h **en échelle log-log**.

Le passage en double précision translate l'erreur d'arrondi d'un facteur $2^{-52}/2^{-23} \approx 2 \times 10^{-9}$,
ce qui multiplie $\check{h}$ par $\sqrt[3]{2 \times 10^{-9}}$ soit environ 1.26×10^{-3}.

pas optimal	simple précision	double précision
$\tilde{h}$	7.1×10^{-3}	8.7×10^{-6}
$\check{h}$	8.7×10^{-3}	1.1×10^{-5}

erreur min	simple précision	double précision
$e(\check{h})$	1.9×10^{-5}	2.8×10^{-11}
$e(\tilde{h})$	2.5×10^{-5}	3.7×10^{-11}

7.6 Influence du nombre de termes sur l'erreur de troncature

Éliminer les termes en h^3 dans le développement de Taylor
en compensant $h^3 f^{(3)}(x)$ issu des points à $\pm h$
par $(2h)^3 f^{(3)}(x) = 8h^3 f^{(3)}(x)$ issu des points à $\pm 2h$
avec une pondération relative de $-1/8$.

Schéma aux différences finies centré à **quatre termes**
pour estimer la dérivée première d'une fonction f :

$$f'(x) \approx f_{1_b}(x) = \frac{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)}{12h}$$

Les termes en puissances paires de h se compensent par symétrie
Erreur de troncature issue du terme en $h^5 f^{(5)}$ dans le développement de f donc :

$$e_t \propto h^4$$

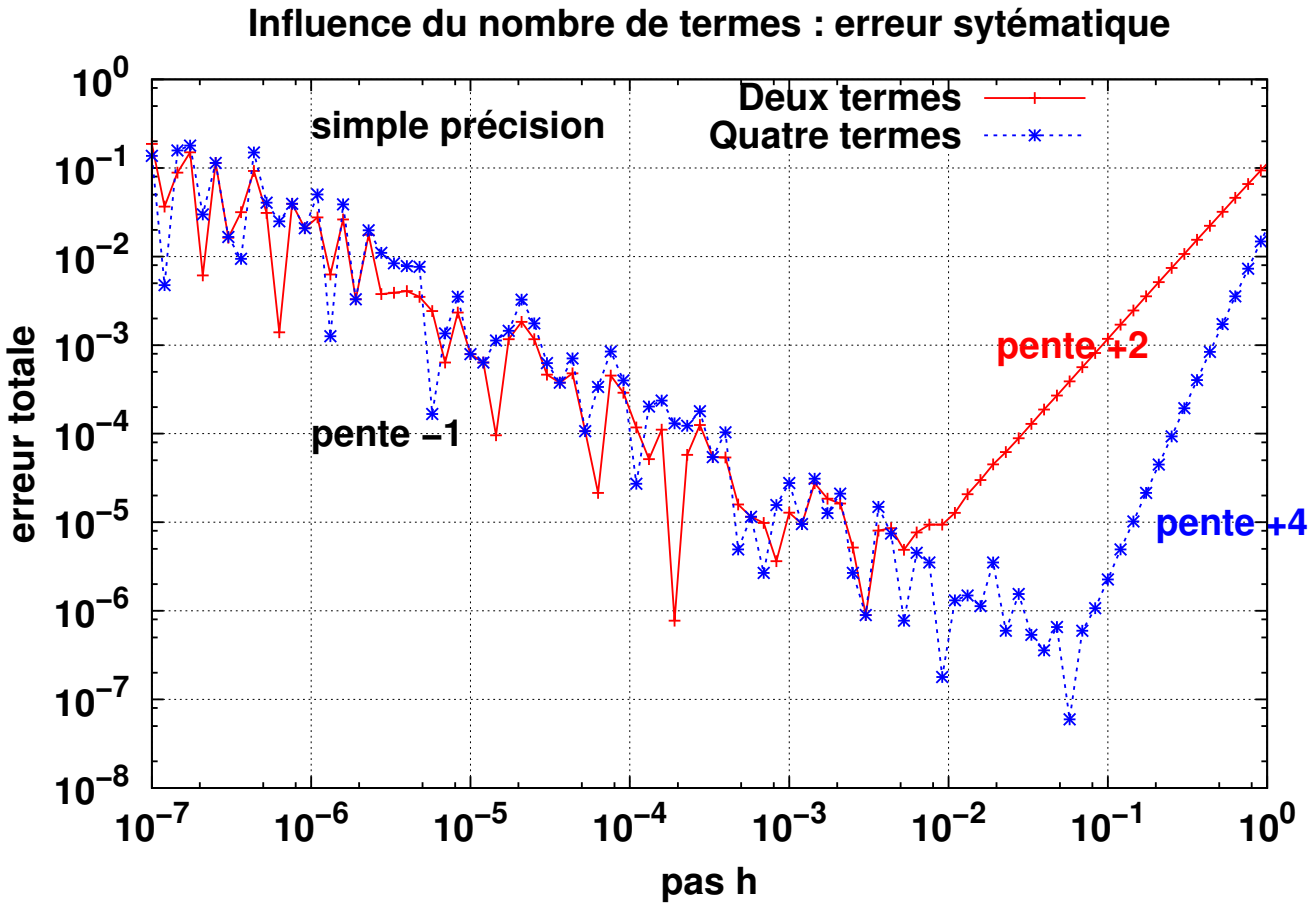


FIGURE 3 – Erreur totale e des estimateurs à deux termes (rouge) et à quatre termes (bleu) de la dérivée première en fonction du pas h **en échelle log-log**.

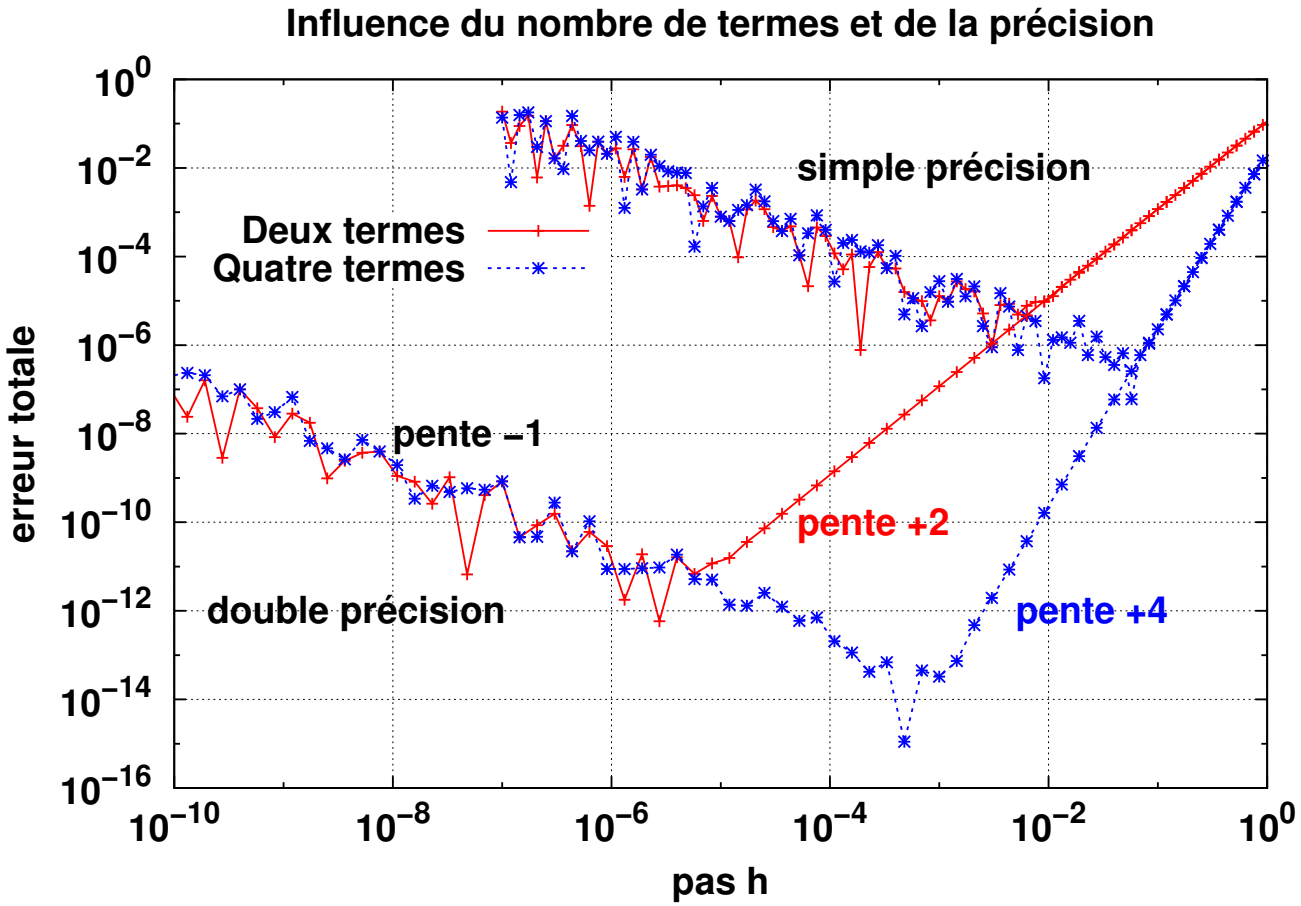


FIGURE 4 – Erreur totale e des estimateurs à deux termes (rouge) et à quatre termes (bleu) de la dérivée première pour deux précisions en fonction du pas h

Conclusion

Passer d'un schéma à 2 termes à un schéma à 4 termes **améliore nettement l'erreur de troncature**, qui varie en h^4 au lieu de h^2 .

L'erreur d'arrondi augmente mais très peu.

À précision des réels donnée, l'optimum est obtenu pour un pas plus grand et l'erreur totale est plus faible. Le schéma à 4 termes est donc préférable.

Dérivées d'ordre supérieur

Les schémas aux différences finies pour la dérivée d'ordre p

— présentent une **erreur d'arrondi** en h^{-p}

— mais leur **erreur de troncature** dépend du nombre de termes utilisés, par ex. en h^2 pour une dérivée seconde avec un schéma centré à 3 termes.