UPMC

Master P&A/SDUEE

MNCS UE MU4PY209

Méthodes Numériques et Calcul Scientifique

Introduction

2020–2021

Albert.Hertzog@lmd.ipsl.fr

Table des matières

1 UE MNCS

1.1 Introduction

— **Objet :** étude/programmation de **méthodes numériques**

⇒ Comprendre certaines « boîtes noires » disponibles dans des bibliothèques numériques

⇒ Se sensibiliser aux erreurs numériques (troncature, arrondi)

— **Outil :** Langage compilé **C++**

⇒ Structurer/Construire un code informatique

⇒ Compilation séparée (réutilisation de partie de codes)

⇒ Notions de performances de code

N.B. : **Python** sera principalement utilisé pour produire des graphiques

1.2 Calendrier prévisionnel

	Semaine	Cours (1h30)	Travaux encadrés ou Contrôle continu
1	14/01 ou 15/01	Rappels, environnement de travail Erreurs d'arrondi et de troncature	
2	21/01 ou 22/01		Tabulation de fonction
3	28/01 ou 29/01		Dérivation numérique
4	04/02 ou 05/02	Équations différentielles ordinaires	Dérivation numérique
5	11/02 ou 12/02		**Contrôle continu 1 (sur consoles)** Équations différentielles scalaires
6	18/02 ou 19/02		Équations différentielles scalaires
	25/02 ou 26/02	*Semaine de travail personnel*	
7	4/03 ou 5/03		Résolution d'EDO vectorielles
8	11/03 ou 12/03	Équations aux dérivées partielles	**Contrôle continu 2 (compte-rendu)**
9	18/03 ou 19/03		Équations aux dérivées partielles statiques
10	25/03 ou 26/03		Équations aux dérivées partielles statiques
11	1/04 ou 2/04		Équations aux dérivées partielles dynamiques
12	8/04 au 9/04		**Contrôle continu 3 (sur consoles)**

1.3 Modalités d'évaluation

— **1re session :** CC1 20 %, CC2 30 %, CC3 50 %

— **2e session :** 100 % console

1.4 Page web de l'UE

La page **Moodle** de l'UE sera maintenue à jour pendant le semestre pour fournir :

— les transparents de cours

— les énoncés de TE

— tout autre document (texte, liens internet) utile à l'UE

1.5 Modalités d'enseignement

- **Présentiel :**
  - TE & cours : salles info de l'UFR de physique, **couloir 22-23, 109–111–112**
- **Distanciel :**
  - Cours : **Zoom** (lien sur Moodle)
  - TE : **Discord** (lien sur Moodle)
  - Environnement de travail : **JupyterLab** (cf. fin exposé) ou solution personnelle

Merci de répondre au sondage sur la page **Moodle** de l'UE !

2 Rappels sur les tableaux

2.1 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** au bloc en C++ et aux fonctions appelées dans ce bloc

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 le programmeur), sur le **tas** (***heap***)

⇒ **portée globale** : peuvent être alloués dans une fonction et rendus accessibles dans la fonction appelante

2.2 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...

2.3 Solution adoptée pour cette UE

Utilisation de la libraire Eigen (https://eigen.tuxfamily.org/)
qui prend en charge les allocations sur le tas ou la pile selon la taille des tableaux.
Eigen définit en fait des **objets C++** (nouveaux types), et des **méthodes** associées, que nous utiliserons.

Objets Eigen

— **VectorXf** : Vecteur de taille variable (**X**) de float (**f**)

— **MatrixXf** : Matrice de taille variable (**X**) de float (**f**)

(En fait, Eigen définit beaucoup d'autres types : Vector2i, Matrix4d, ArrayXXf etc.)

La taille du vecteur (de la matrice) pourra être définie :

— à la **déclaration** de la variable (qui sera souvent tardive),

— lors de son **affectation** (utilisation à gauche du signe **=**) : **resizing** implicite

2.4 Tableaux 2D natifs C++ et Matrix Eigen

Tableaux natifs C++	Objets **Matrix** Eigen
Accès à l'élément (i, j)	
`tab[i][j]`	`mat(i, j)`
Rangement des matrices	
par **lignes** (*row-major*)	par **colonnes** (*column-major*) par défaut
tableaux de tableaux $\Rightarrow$ difficile d'accéder aux colonnes	sections de tableaux $\Rightarrow$ facile d'accéder aux lignes/colonnes `mat.row(i),mat.col(j)`
Indice le plus **rapide** (éléments contigus)	
le plus **à droite**	le plus **à gauche**

Attention : L'instruction `mat[i, j]` appliquée à une matrice Eigen ne produira pas le résultat escompté !

3 Rappels sur les entrées-sorties formatées

3.1 Instructions préalables

`#include <iostream>` : instruction au pré-processeur

`using namespace std;` : évite de préciser l'espace de nom (`std::cout`)

3.2 Entrées-sorties standard

Affichage sur l'écran (par défaut) :

```
cout << "Distance : " << dist << " km" << endl;
```

Affichage sur la sortie d'erreur :

```
cerr << "Impossible d'ouvrir le fichier : " << fichname << endl;
```

Lecture au clavier (par défaut) :

```
cin >> var_1 >> var_2;
```

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

`<` en entrée `>` en sortie

3.3 Formatage des entrées-sorties

Les entrées-sorties standard utilisent un formatage par défaut (nombre de chiffres affichés, notation « point fixe »), que l'on peut souhaiter modifier. On fait appel pour cela à des « manipulateurs », qui nécessitent l'inclusion d'un fichier d'entête :

```
#include <iomanip>
```

Nous utiliserons 2 manipulateurs :

`setiosflags(ios::scientific)` : notation mantisse+exposant

`setprecision(7)` : affiche 7 chiffres après la virgule (pour les réels)

Les modifications du formatage sont permanentes (jusqu'au prochain appel des manipulateurs), et peuvent être enchaînées :

```
cout << setiosflags(ios::scientific) << setprecision(7);
```

3.4 Application aux tableaux 1D

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

```
Eigen::VectorXi v(3); // déclaration
v << 1, 2, 3; // initialisation (surcharge de <<)
```

Affichage en ligne

```
cout << v;
```
$\Rightarrow$
```
1 2 3
```

Affichage en colonne

```
for (int i=0; i<3; i++){
  cout << v(i) << endl;
}
```
$\Rightarrow$
```
1
2
3
```

3.5 Affichage de tableaux 2D

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

nl=2

nc=3

11	12	13
21	22	23

affichage **nl** lignes $\times$ **nc** colonnes

```
cout << "Entrer les dimensions (nl, nc)" << endl;
cin >> nl >> nc;
Eigen::MatrixXf mat(nl, nc);//Déclaration tardive
mat << 11, 12, 13, 21, 22, 23;
cout << mat << endl;
```

Eigen prend tout en charge ! !

NB : Noter la différence entre rangement des éléments en mémoire (column-major) et initialisation/affichage (row-major).

3.6 Fichiers formatés : syntaxe d'ouverture/fermeture

Connexion d'un flot à un fichier en C++

```
void std::fstream::open(const string &*filename,
                        ios::openmode mode = ios::in|ios::out)
```

mode		position	valeur par défaut
ios::in	read	début	ifstream
ios::out	write	début	ofstream
ios::app	append	fin	

ajouter `| ios::bin` si binaire

empêche la connexion si erreur

lectures ou écritures

fermeture (et vidage du tampon)

```
void std::fstream::close()
```

3.7 Exemple d'écriture sur fichier

ecrit.cpp

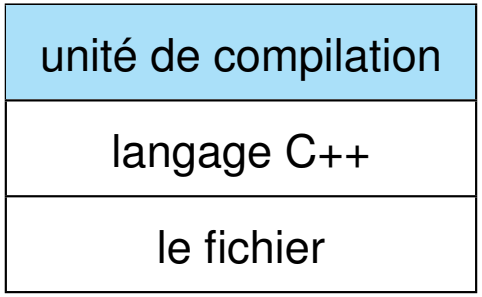
```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <Eigen/Dense>
using namespace std;
void ecrit(string fichname, const Eigen::VectorXf &x,
                            const Eigen::VectorXf &y) {
 ofstream my_file; //declaration d'un objet output file stream

 my_file.open(fichname); //connexion au fichier
 if (! my_file) { //test de la reussite
   cerr << "Impossible d'ouvrir " << fichname << endl;
   exit(EXIT_FAILURE);
 }
 //definition du format de sortie
 my_file << setiosflags(ios::scientific) << setprecision(7);
 for (int i = 0; i < x.size(); i++) {
   my_file << x(i) << " " << y(i) << endl;
 }
 my_file.close(); //fermeture de la connexion
}
```

4 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



4.1 Intérêt de la compilation séparée des fonctions

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

4.2 Mise en œuvre robuste de la compilation séparée

Donner les moyens au **compilateur** de vérifier si **le nombre, le type et la position des arguments des fonctions** lors d'un appel sont conformes au prototype de la fonction.

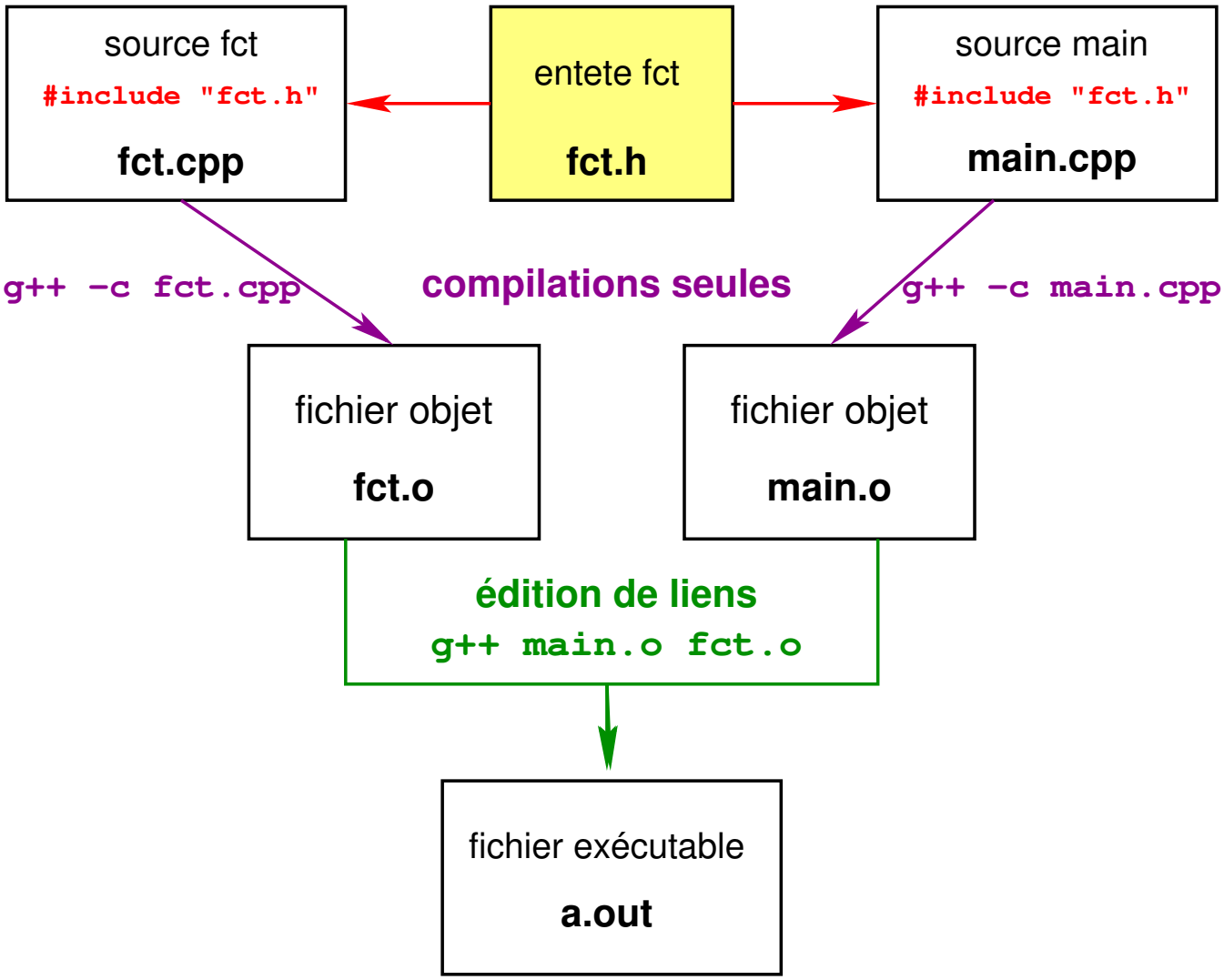
langage C++	
passage par **copie** donc **conversion** des arguments sauf pour les pointeurs	passage par **référence** donc **respect exact** du type

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

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

$\Longrightarrow$ **Solution plus robuste**

langage C++
Déclarer les prototypes dans les fichiers d'**entête `*.h`** et les inclure à la fois : — dans la définition — et dans les fonctions appelantes à l'aide de la directive **`#include "fich.h"`**
$\Rightarrow$ Éviter les déclarations multiples si plusieurs inclusions **`#ifndef MY_FCT_H`** **`#define MY_FCT_H`** **insérer ici les prototype** des fonctions de `my_fct` **`#endif`**



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

5 Arguments des fonctions

5.1 Passage par copie

Par défaut, une copie de l'argument effectif est créée lors de l'entrée dans la fonction (et détruite lors la sortie) :

$\Rightarrow$ **Impossible de modifier** l'argument effectif dans le programme appelant (**protection contre la modification**)

$\Rightarrow$ Pour des arguments effectifs de grande taille (vecteur, matrice) : **temps de recopie** significatif

arg_copie.cpp

```
#include <iostream>
using namespace std;

void incr(float x) {
   x++;
   cout << x << endl; // 3.1
}

int main() {
  float f = 2.1;

  cout << f << endl; // 2.1
  incr(f);
  cout << f << endl; // 2.1
}
```

Privilégier ce mode de passage pour des arguments de petite taille qu'on ne souhaite pas modifier dans le programme appelant

5.2 Passage par référence

Si l'**argument formel** est déclaré avec un **&**, le passage de l'argument effectif est fait par référence :

⇒ **Possibilité de modifier** l'argument effectif dans le programme appelant

⇒ On ne passe que l'adresse mémoire de l'argument effectif : **très rapide**

arg_ref.cpp

```
#include <iostream>
using namespace std;

void incr(float &x) {
   x++;
   cout << x << endl; // 3.1
}

int main() {
  float f = 2.1;

  cout << f << endl; // 2.1
  incr(f);
  cout << f << endl; // 3.1
}
```

Utiliser obligatoirement ce mode de passage pour des arguments que l'on souhaite modifier dans le programme appelant

L'ajout du qualificateur **const** devant l'argument formel garantit que l'argument effectif ne peut pas être modifié (erreur de compilation) :

$\Rightarrow$ Permet de transmettre un argument de grande taille à une fonction sans le temps de recopie associé

arg_ref_const.cpp

```
#include <iostream>
using namespace std;

void incr(const float &x) {
// x++; erreur de compilation
   cout << x << endl; // 2.1
}

int main() {
  float f = 2.1;

  cout << f << endl; // 2.1
  incr(f);
  cout << f << endl; // 2.1
}
```

Privilégier ce mode de passage pour des arguments constants, possiblement de grande taille (vecteurs, matrices)

5.3 Cas des arguments fonctions

5.3.1 Objectif et méthode

Mettre en œuvre des **fonctions-méthodes** (intégration, dérivation numérique, tabulation sur fichier, ...) via des fonctions-méthodes agissant sur des fonctions dites test.

distinguer : • **la fonction formelle** `f` utilisée quand on définit la fonction-méthode :
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 **fichier séparé** (C++).

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

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

5.4 Structure minimale du code

Programme principal

Visibilité de l'interface des fonctions tests effectives

```
#include "fonctions.h"
```

Visibilité de l'interface de la fonction-méthode

```
#include "methode.h"
```

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

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

Définition des fonctions effectives

Fichiers des fonctions

déclarations f1f2.h

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

définitions f1f2.cpp

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

Fonctions-méthodes s'appliquant aux fonctions

```
/* fichier methode.cpp */
#include <functional> //plutôt dans methode.h
#include "methode.h"
using namespace std;
void trait( function<float(float)> f,   ...){
```

Déclaration de l'interface de la fonction formelle

comme argument de la fonction-méthode

via une **instanciation de la classe générique** (*template class*) `function`

```
...
```

appel de la fonction formelle

```
... = f(u)
}
/* fin du fichier methode.cpp */
```

5.5 Exemple de passage de fonction en argument

Programme principal

main.cpp

```
#include <iostream>
#include <Eigen/Dense>
// entêtes personnelles
#include "affiche.h" // la méthode : tabulation
#include "f1f2.h"    // les fonctions test
using namespace std;

int main(){
  float x0, x1, pas;
  int n;
  /* saisie des paramètres */
  cout << "entrer min, max et le nb de pts" << endl;
  cin >> x0 >> x1 >> n;
```

```
  // declaration tardive
  Eigen::VectorXf x(n); // Vecteur d'abscisses
  pas = (x1 - x0) / (float) (n-1);
  for(int i=0; i<n; i++){ // création grille des abscisses
     x(i) = x0 + pas * i;
  }
  cout << "tabulat. de f1 entre " << x(0) << " et "
       << x(n-1) << endl;
  affiche(f1, x);  /* appel de la méthode pour f1 */
  cout << "tabulat. de f2 entre " << x(0) << " et "
       << x(n-1) << endl;
  affiche(f2, x);  /* appel de la méthode pour f2 */
  exit(EXIT_SUCCESS);
}
```

Les fonctions test

f1f2.h (déclarations)

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

f1f2.cpp (définitions)

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

La méthode

affiche.h (déclaration)

```
#ifndef AFFICHE_H
#define AFFICHE_H
#include <Eigen/Dense>
#include <functional>
// prototype de la méthode
void affiche(std::function<float(float)> f,
             const Eigen::VectorXf &x);
// premier argument = pointeur vers une fonction test
#endif
```

affiche.cpp (définition)

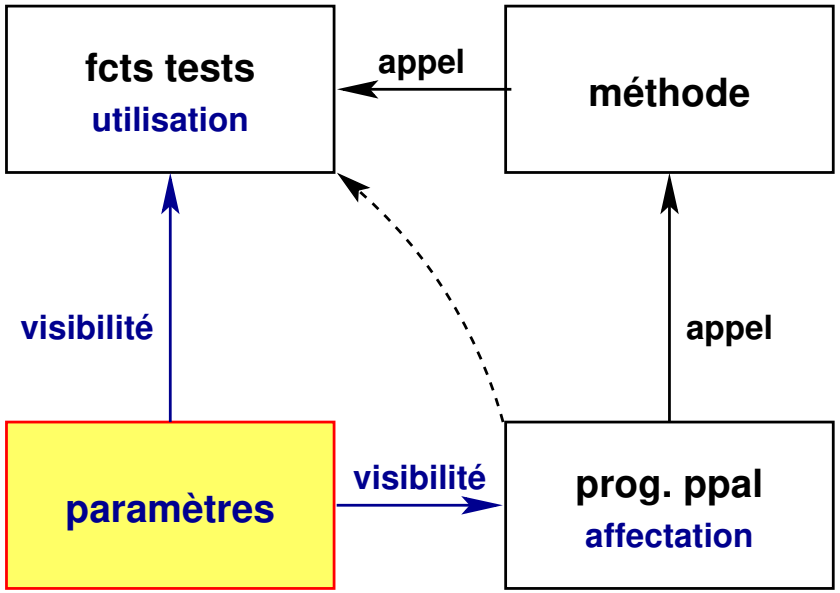
```
#include <iostream>
// entêtes personnelles
#include "affiche.h" // déclaration de la fonction affiche
using namespace std;

void affiche(function<float(float)> f, const Eigen::VectorXf &x) {
  // affichage des valeurs de la fonction f aux points x
  for(int i = 0; i < x.size(); i++){ // affichage
     cout << x(i) << f(x(i)) << endl;
  }
}
```

5.6 Variables globales

5.6.1 Nécessité des variables globales

Quand une **fonction-méthode** (intégration, dérivation, ...) est appliquée à une fonction test, on intègre ou dérive par rapport à **une variable.**
Or cette fonction test peut **dépendre d'autres paramètres** que la méthode n'a pas à connaître, mais que la fonction utilise quand on l'évalue.
Ces paramètres seront (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.



5.6.2 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

5.6.3 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 la référence à cette variable).

Les paramètres des fonctions tests doivent être déclarés comme variables globales partagées entre les fonctions à tester qui les déclarent et le programme principal qui les redéclare et les définit.

Fichiers des fonctions

f1f2.cpp

```
...
#include "f1f2.h"
//paramètres
float k, tau;

float f1(float t){
...
}
float f2(float t){
...
}
```

Programme principal

main.cpp

```
...
#include "f1f2.h"
using namespace std;
//paramètres
extern float k, tau;

int main(){
...
tau = 2.5;
k = 1.;
...
}
```

6 Utilitaire `make` et bibliothèques

6.1 Génération d'applications avec `make`

6.1.1 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.

6.1.2 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-c++**

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**

6.1.3 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)
```

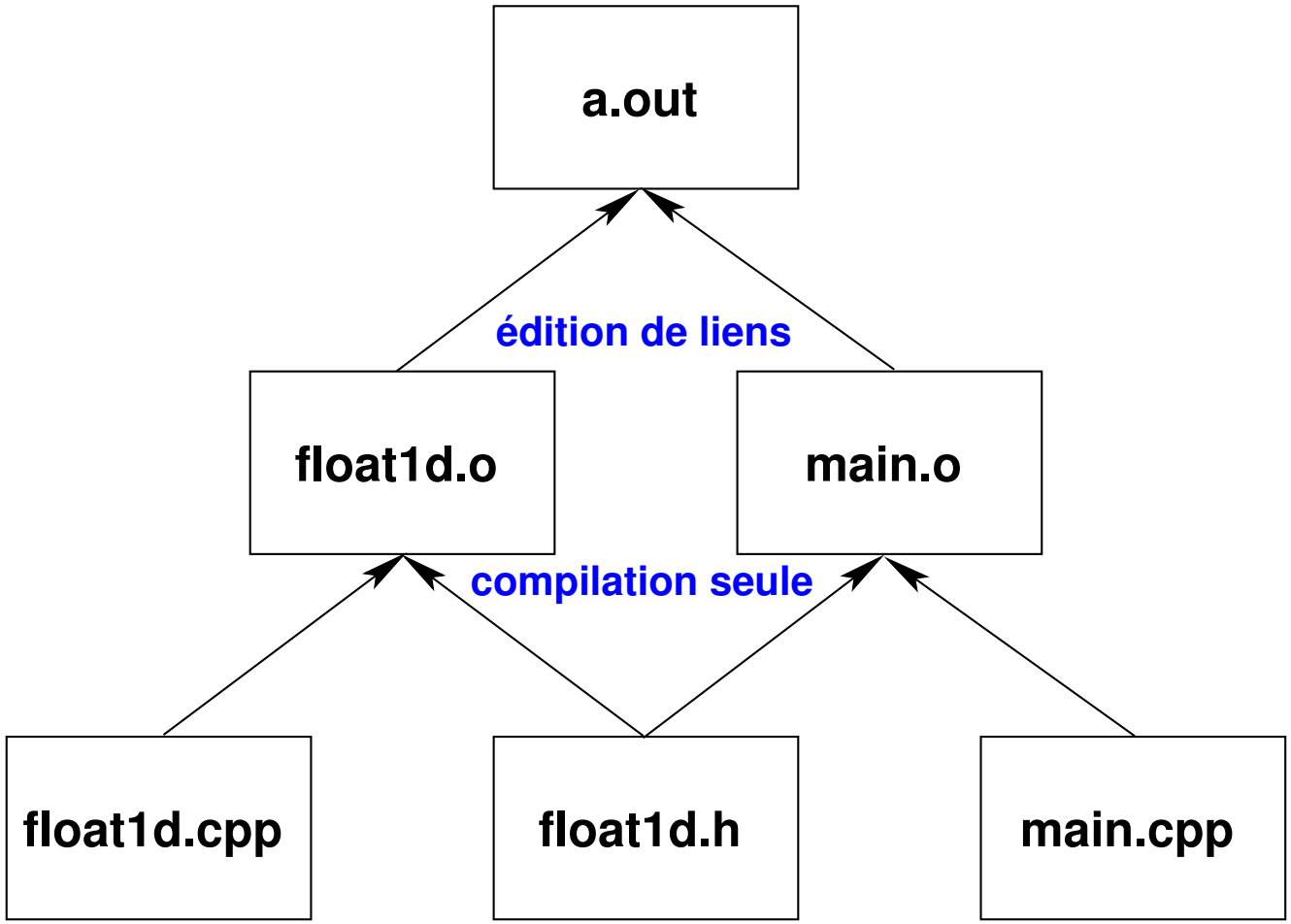
Analyse récursive des dépendances via les règles : pour construire une cible, **`make`** analyse ses dépendances :

— 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.

6.1.4 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
__TAB__g++ float1d.o main.o

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

# pour éviter un problème si un fichier "clean" existe
.PHONY: clean
# ménage : suppression des fichiers reconstructibles
clean:
__TAB__/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.cpp`**, puis **`make`**
   `a.out` dépend de `main.o`,
   qui dépend de `main.cpp`, qui est plus récent
   ⇒ recompilation de `main.cpp` ⇒ `main.o` à jour
   ⇒ `main.o` plus récent que `a.out` ⇒ édition de liens.
   NB : pas de recompilation de `float1d.cpp`

Aides à la construction du makefile

g++ -MM fichier.cpp affiche les dépendances de `fichier.o`

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

6.1.5 Variables interprétées par make, makefile paramétré

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

Paramétrage des outils (compilateur par exemple)

CPPC = g++ choix du compilateur C++

CFLAGS = -W -Wall -pedantic -std=c++17 options du compilateur C++

```
float1d.o : float1d.cpp float1d.h
___TAB___$(CPPC) $(CFLAGS) -c  float1d.cpp
```

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

Listes de fichiers

```
SRCS = main.cpp float1d.cpp
```

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

```
SRCS = \
           main.cpp \
           float1d.cpp
```

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 macros prédéfinies

Compilation

```
float1d.o : float1d.cpp float1d.h
__TAB__g++ -c $< -o $@
```

`$<` signifie première dépendance seulement, donc en développant

```
__TAB__g++ -c float1d.cpp -o float1d.o
```

Édition de liens

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

`$^` signifie toutes les dépendances, donc en développant

```
__TAB__g++ float1d.o main.o -o a.out
```

6.1.6 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__g++ -c $< -o $@
```

```
%.o: %.c %.h # syntaxe + riche (dépendances multiples)
__TAB__g++ -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`.

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

Objectif : estimer numériquement **la dérivée première** $f'(t)$ d'une fonction f en t à partir des échantillons de la fonction f aux instants $t + ih$, où h est le pas d'échantillonnage de f.

Plusieurs approximations de $f'(t)$ ou schémas aux différences finies envisageables. Les plus simples sont les schémas à deux termes :

$$f'_{dg}(t) = \frac{f(t) - f(t-h)}{h} \quad \text{décentré gauche} \quad \textit{backward}$$

$$f'_{dd}(t) = \frac{f(t+h) - f(t)}{h} \quad \text{décentré droite} \quad \textit{forward}$$

$$f'_{c}(t) = \frac{f(t+h) - f(t-h)}{2h} \quad \text{centré} \quad \textit{centered}$$

Démonstration en TE qu'il faut préférer le schéma **centré**

7.1 Erreur d'estimation associée

L'erreur d'estimation, $f'_c(t) - f'(t)$ pour le schéma centré, comporte deux contributions qui s'ajoutent en valeur absolue :

— l'**erreur systématique de troncature** déterministe de valeur absolue e_t liée au **nombre fini de termes** dans l'estimateur (2 termes dans ce schéma).
Dérivation théorique $\Longleftrightarrow$ multiplication par $\boldsymbol{ik}$ dans l'espace de Fourier
Dérivation numérique centrée à 2 termes $\Longrightarrow \times \boldsymbol{i \sin(kh)/h}$
qui n'est proche de $\boldsymbol{ik}$ que pour $h \to 0$.

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

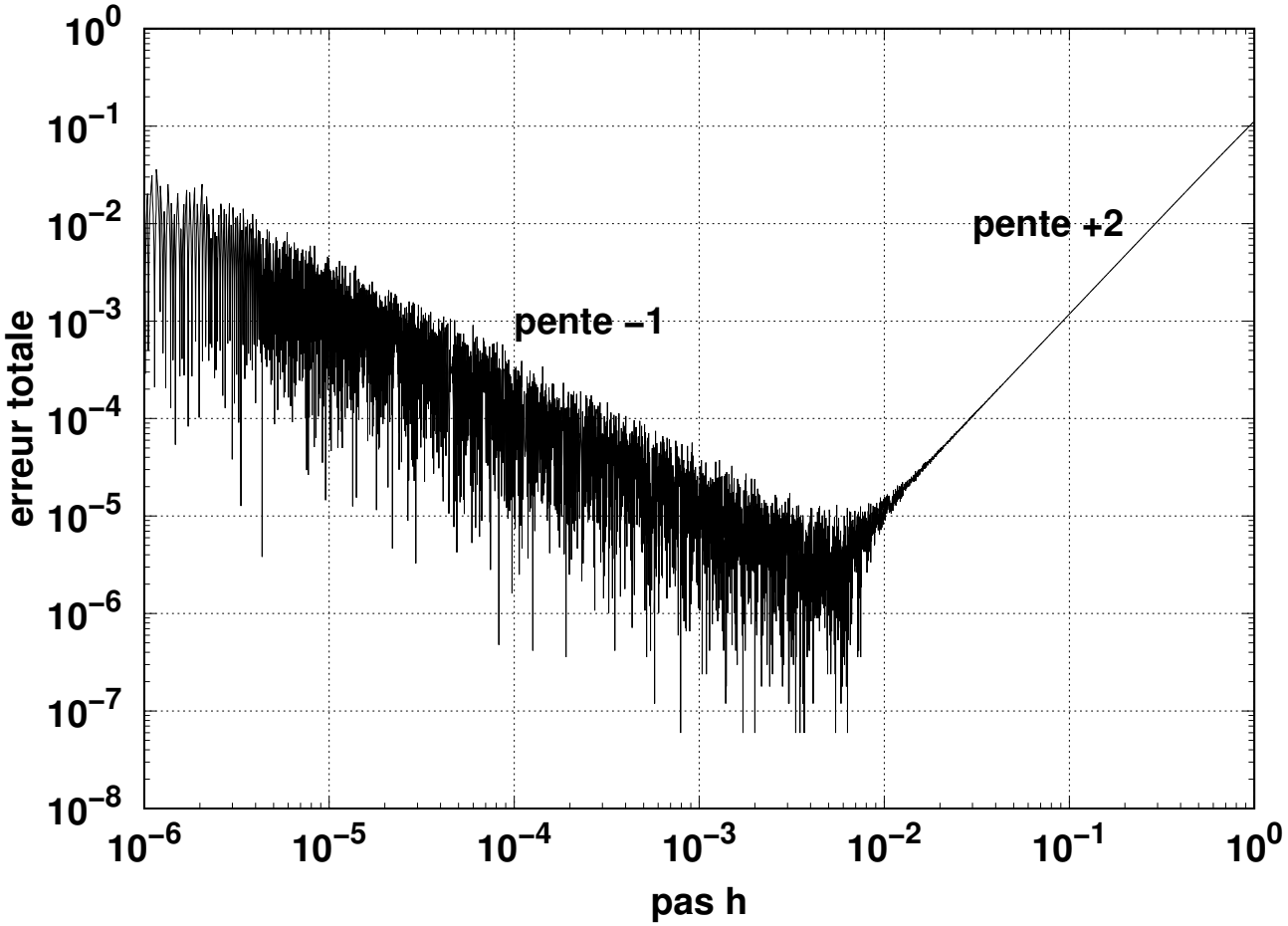


FIGURE 1 – **Erreur totale** e de l'estimateur centré à 2 termes de la dérivée première de la fonction sinus en $t = \pi/4$ en fonction du pas h **en échelle log-log** .

7.1.1 Estimation de l'erreur de troncature

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

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

$$f(t-h) = f(t) - h\frac{\mathrm{d}f}{\mathrm{d}t}(t) + \frac{h^2}{2}\frac{\mathrm{d}^2 f}{\mathrm{d}t^2}(t) - \frac{h^3}{6}\frac{\mathrm{d}^3 f}{\mathrm{d}t^3}(t-\theta' h)$$

où θ et θ' sont dans l'intervalle $[0,\ 1]$.

Estimateur centré à 2 termes de la dérivée :

$$\frac{f(t+h) - f(t-h)}{2h} = f'(t) + \frac{h^2}{12}\left[f'''(t+\theta h) + f'''(t-\theta' h)\right]$$

Erreur absolue de troncature liée à la dérivée troisième :

$$e_t \approx \frac{\boldsymbol{h^2}}{6}|f'''(t)|$$

7.1.2 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

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

Erreur absolue d'arrondi sur $f'_c(t)$ majorée par e_a :

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

7.2 Comparaison des erreurs pour schéma centré à 2 termes

Erreur d'arrondi

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

décroit si h croît

Pente en log-log -1

Dominante pour h faible

Erreur de troncature

$$e_t \propto h^2$$

croit avec h

Pente en log-log $+2$

Dominante pour h grand

Majorant de l'erreur absolue totale e

$$e \leqslant \frac{\varepsilon|f(t)|}{h} + \frac{h^2}{6}|f'''(t)| = e_m$$

Les deux erreurs varient en sens inverse selon le pas h

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

$\Rightarrow$ pas optimal $\tilde{h}$

7.3 Recherche du minimum de l'erreur totale

Méthode analytique

$\tilde{h}$ qui minimise l'erreur totale $= \sqrt[3]{3\varepsilon \left| \dfrac{f(t)}{f'''(t)} \right|} \quad \Rightarrow \quad e(\tilde{h}) = |f(t)| \sqrt[3]{\dfrac{9\varepsilon^2}{8} \left| \dfrac{f'''(t)}{f(t)} \right|}$

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

Recherche graphique approximative

$\check{h}$ à l'intersection des droites en log-log $\quad \Rightarrow \quad \check{h} = \sqrt[3]{6\varepsilon \left| \dfrac{f(t)}{f'''(t)} \right|} = \tilde{h}\sqrt[3]{2} \approx 1.26\tilde{h}$

$$e(\check{h}) = |f(t)| \sqrt[3]{\frac{4\varepsilon^2}{3} \left| \frac{f'''(t)}{f(t)} \right|}$$

Dépendance en ε

$$h_{\text{min}} \propto \varepsilon^{1/3} \quad \text{et} \quad e_{\text{min}} \propto \varepsilon^{2/3}$$

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

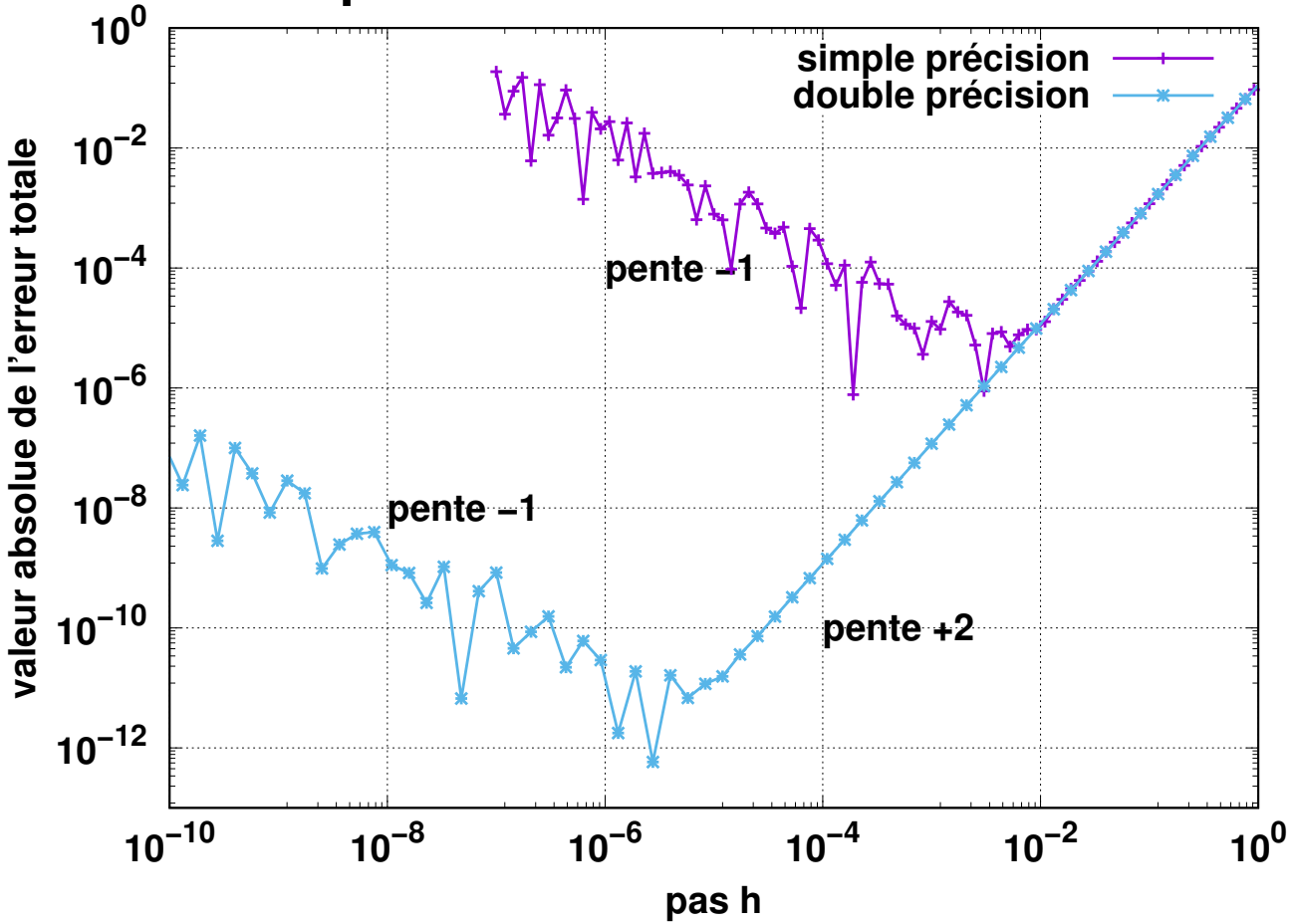


FIGURE 2 – Erreur totale (valeur absolue) $|e|$ en simple et double précision de l'estimateur à 2 termes de la dérivée première en fonction du pas h **en échelle log-log**.

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}.
ce qui multiplie e_{min} par $\left(2 \times 10^{-9}\right)^{2/3}$ soit environ 1.5×10^{-6}.

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}

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

Objectif : éliminer les termes en h^3 dans le développement de Taylor de f
en compensant $h^3 f^{(3)}(t)$ issu des points à $\pm h$
par $(2h)^3 f^{(3)}(t) = 8h^3 f^{(3)}(t)$ 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 :

$$f'(t) \approx f'_{c2}(t) = \frac{-f(t+2h) + 8f(t+h) - 8f(t-h) + f(t-2h)}{12h}$$

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 :

$$e_t \propto h^4$$

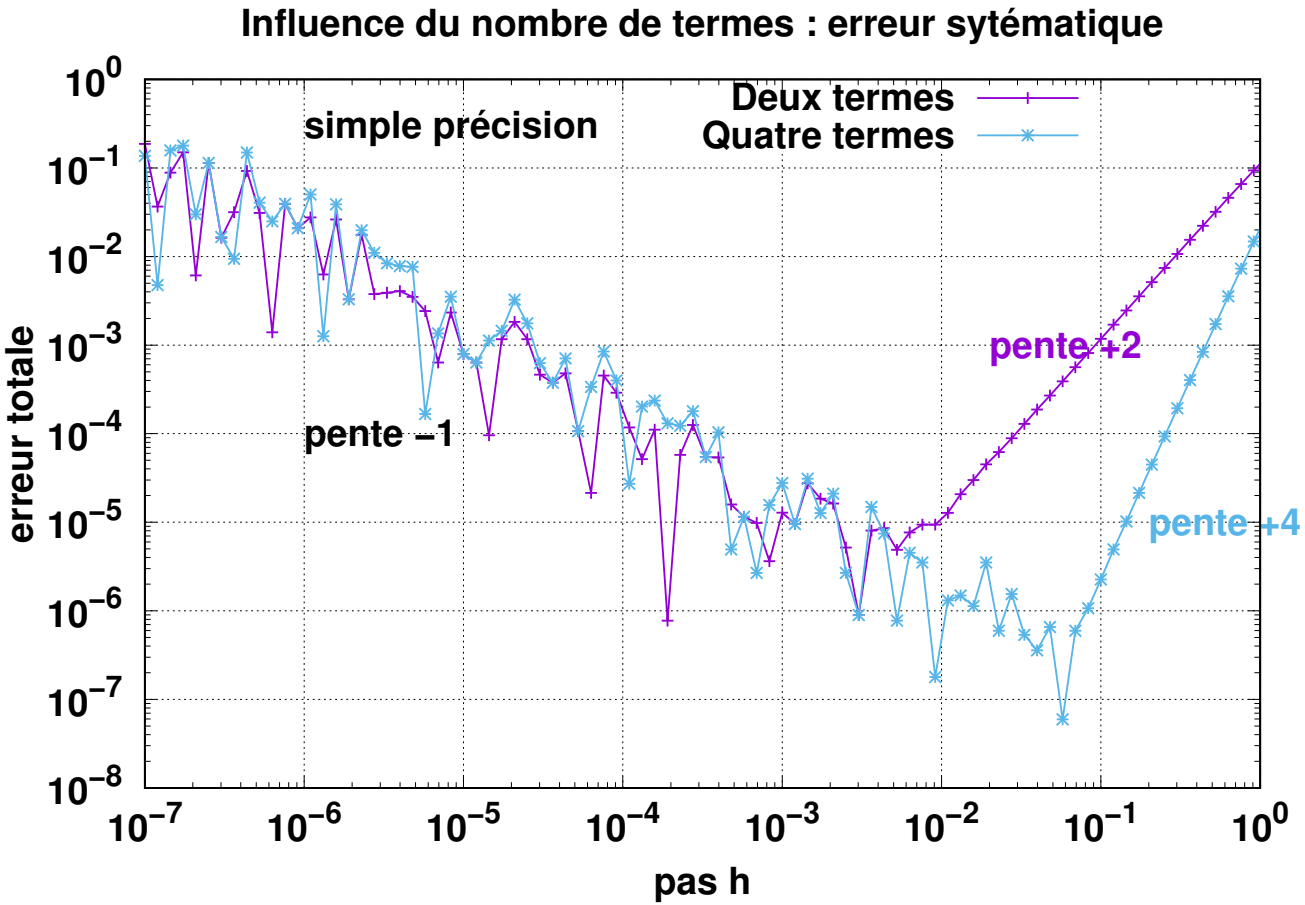


FIGURE 3 – Erreur totale e des estimateurs à deux termes (rouge) et à quatre termes (bleu) de la dérivée première en fonction du pas h **en échelle log-log**.

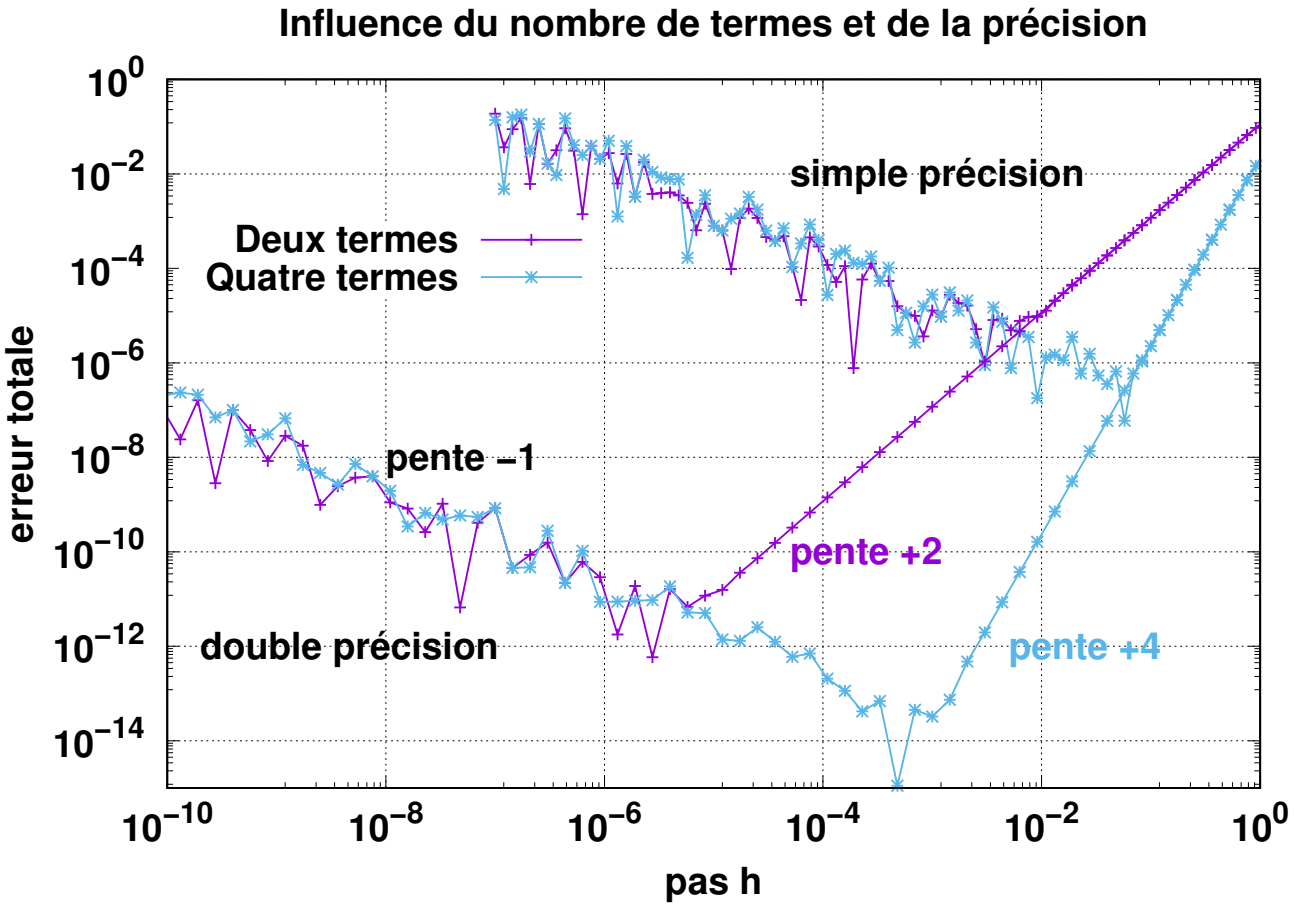


FIGURE 4 – Erreur totale e des estimateurs à deux termes (f'_c rouge) et à quatre termes (f'_{c2} bleu) de la dérivée première pour deux précisions en fonction du pas h

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.

7.6 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.

8 Présentation de JupyterLab

JupyterLab est un environnement de travail en ligne :

— solution **possible** pour le travail à distance

— accès via un navigateur $\Rightarrow$ **indépendant** du système de votre PC

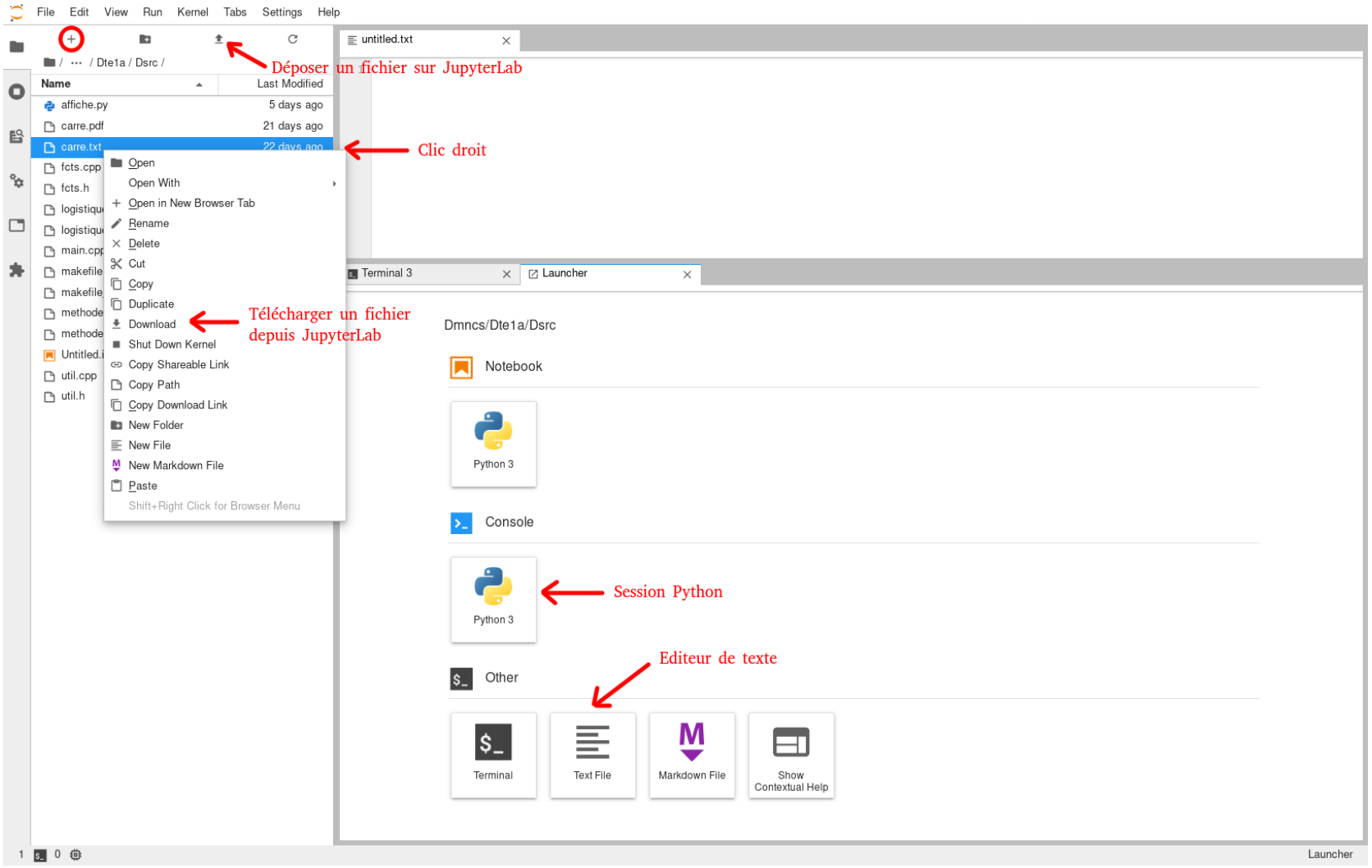
https://www.lmd.polytechnique.fr/jupyterhub

Accès via **login/mot de passe** :

— login : 1ère lettre du prénom + nom (ex : pdupond)

— mot de passe : vous sera envoyé par mail

Suivre procédure d'initialisation de l'environnement sur la section TE1A de la page Moodle MNCS


File Edit View Run Kernel Tabs Settings Help
/ Dte1a / Dsrc /
Name
Last Modified
affiche.py 5 days ago
carre.pdf 21 days ago
carre.txt 22 days ago
fcts.cpp
fcts.h
Open
Open With
Open in New Browser Tab
Rename
Delete
Cut
Copy
Duplicate
Download
Shut Down Kernel
Copy Shareable Link
Copy Path
Copy Download Link
New Folder
New File
New Markdown File
Paste
Shift+Right Click for Browser Menu
Déposer un fichier sur JupyterLab
Clic droit
Télécharger un fichier depuis JupyterLab
untitled.txt
Terminal 3
Launcher
Dmncs/Dte1a/Dsrc
Notebook
Python 3
Console
Python 3
Session Python
Other
Editeur de texte
Terminal
Text File
Markdown File
Show Contextual Help
Launcher