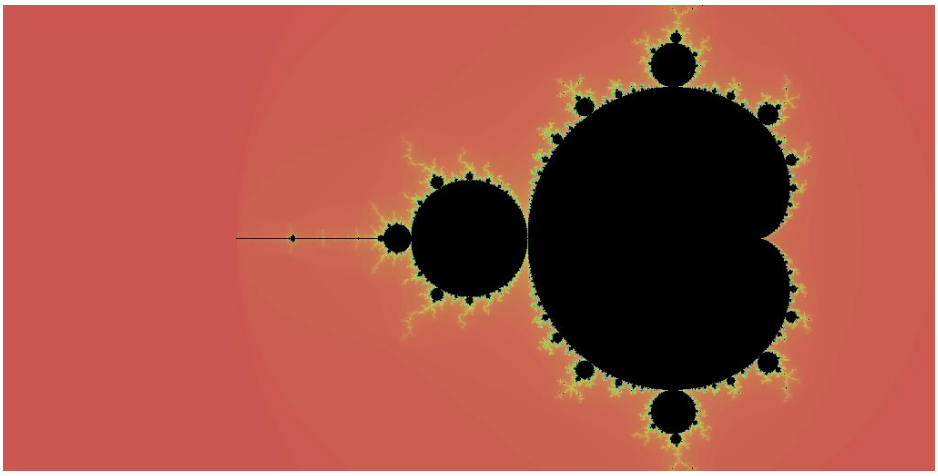
Sorbonne Université

MU4PY109
Mention Physique et Applications
Mention Science de l'Océan, de l'Atmosphère et du Climat
Notes de cours
Année universitaire 2023-2024

Méthodes informatiques

Auteur :
Pascal Viot



Remerciements :
Je tiens à remercier Jacques Lefrere qui a été pionnier pour le développement de la physique numérique en Master, et bien entendu tous les membres de l'équipe : Sylvain Baumont, Johan Biscaras, Jérémie Klinger, Jean-Baptiste Madeleine, Philippe Sindzingre and Sofian Teber.

1er septembre 2023

CHAPITRE 1

L'interface personne-machine

1.1 Objectifs

La compréhension de phénomènes physiques, chimiques, biologiques passe très souvent par une étape de modélisation. Celle-ci peut provenir d'observations préalables ou simplement de l'imagination de ses concepteurs. Une fois le problème mathématiquement bien défini (ce qui n'est pas une opération simple), il reste à le résoudre. Hormis dans un nombre limité de situations, cette tâche n'est pas possible à réaliser de manière exacte. Pendant des siècles, il convenait soit de simplifier le modèle initial pour avoir une solution, au risque de perdre les ingrédients essentiels de cette modélisation, soit de faire des approximations parfois pas très bien contrôlées. L'avènement des ordinateurs a permis de réaliser des opérations logiques et numériques avec une fréquence dépassant très rapidement les capacités humaines habituelles. Avec la croissance quasi-interrompue de la puissance des ordinateurs, on a vu exploser les possibilités d'obtenir des solutions à des problèmes très compliqués. Ces résultats reposent eux-mêmes sur deux pieds : le développement de méthodes numériques et celui de méthodes informatiques. Réaliser l'apprentissage de cette combinaison est nécessairement une gageure que nous allons tenter de défier.

Compte tenu du temps limité, des choix importants doivent être faits dès maintenant pour éviter une dispersion néfaste à l'apprentissage. Dans un document séparé, nous allons présenter quelques méthodes numériques permettant de réaliser des calculs qui interviennent très souvent en sciences : calculs d'intégrales, intégration d'équations différentielles, opérations matricielles, transformées de Fourier, méthode Monte Carlo. Cette liste, loin d'être exhaustive, permet de comprendre, indépendamment de la méthode informatique choisie dans un second temps, ce qui est le plus adapté pour obtenir un résultat numérique au problème que l'on souhaite résoudre.

Une fois que l'on a sélectionné la méthode la plus adaptée, il faut que l'ordinateur fasse le travail correspondant à votre choix (que l'on espère pertinent). Jusqu'à présent, le "OK Google" ou le chat GPT ne sont pas encore suffisamment évolués pour que cela vous permette de dire "résoudre mon problème avec la méthode que j'ai choisie". C'est donc l'objet de la seconde partie de cette unité d'enseignement, maîtriser les outils qui permettent à l'ordinateur d'exécuter les tâches que vous souhaitez qu'il fasse pour vous.

C'est peu de dire que la manière dont les calculs faits par l'ordinateur sont éloignés de notre langage habituel. Il y a donc eu des interfaces successives depuis 80 ans pour essayer de rendre les choses de plus en plus simples à l'utilisateur. Malgré tous ces efforts, les choses restent compliquées pour deux raisons principales : il est nécessaire d'avoir un langage intermédiaire entre la personne et la machine car le langage de la machine n'est pas facilement utilisable par une personne. La traduction de ce langage intermédiaire est réalisée de deux manières différentes : soit à partir d'un interpréteur qui traduit et exécute chaque ligne de code au fil de l'eau ; cela présente l'avantage d'être rapide quand on écrit un programme progressivement, mais cela rend l'exécution assez lente. Avec un langage de bas niveau un compilateur traduit l'ensemble du code en créant au passage des fichiers intermédiaires (si le code est constitué de plusieurs parties) et réalise un assemblage de la totalité de ceux-ci avec les bibliothèques nécessaires à leur exécution. Au final, l'exécution est nettement plus efficace (très souvent en gagnant un à deux ordres de grandeur en efficacité). Pour choisir le langage le plus adapté, cela nécessite de savoir le temps que l'utilisateur met à écrire le code et à l'additionner avec celui de l'exécution. Or, généralement, le temps consacré au développement pour un code compilé est nettement supérieur à celui pour un code interprété. Cette situation paradoxale nous conduit à utiliser un représentant de chacun de ces langages.

Le choix retenu pour cette enseignement repose sur deux langages dont le développement actuel est très intense depuis ces deux dernières décennies. Il s'agit du Python pour le langage interprété et du C++ pour le langage compilé.

Ces notes de cours cherchent à placer en parallèle les concepts développés dans chacun des langages et soulignent que les différences initiales sont en train de s'éclipser. Ces deux langages reposent sur la notion de programmation objet avec la possibilité d'une programmation procédurale. Les langages compilés reposent historiquement sur le typage fort des objets tandis que les langages interprétés sur un typage faible. Les évolutions du Python et du C++ semblent converger vers une direction commune. Une fois développé un code en Python, il existe une manière simple de compiler ce code et qui permet de gagner une efficacité en exécution et de se rapprocher de celle d'un code écrit en C++. On peut imaginer que dans dix ans, l'unification pourrait avoir lieu ce qui simplifierait à la fois l'usage et le développement.

1.2 Les environnements de développement intégré

Comme nous l'avons vu précédemment, chaque langage nécessite de réaliser plusieurs étapes avant d'arriver au but recherché qui est celui d'avoir une exécution correspondant à la mise en équation du modèle considéré. Cela nécessite un code correct, lisible et doit être ensuite traduit par l'interprétateur ou le compilateur. Il existe des produits tout-en-un qui réalisent toutes ces opérations en minimisant les étapes intermédiaires. On appelle cela des environnements de développement intégré (en anglais l'acronyme est IDE pour integrated development environment). Le choix retenu est Codeblocks pour le C++ et Spyder pour Python. Ils offrent l'avantage de pouvoir les utiliser après un très rapide apprentissage et ils sont adaptés pour développer des codes pas très volumineux. Ils présentent aussi l'avantage d'être disponibles sur les principales plateformes informatique : Codeblocks existe sur Windows, Linux. La version MacOs n'est malheureusement pas maintenue et il faut alors utiliser soit le produit propriétaire xcode ou visualstudio. CodeBlocks a l'avantage d'une prise en main, mais il n'évolue plus depuis plusieurs années et le passage à vscode deviendra sans doute nécessaire. Pour Python, l'environnement Spyder est réellement multiplateforme et mise à jour sur tous les systèmes et de plus en évolution constante.

1.2.1 Codeblocks

Pour installer Codeblocks sur un ordinateur personnel, vous pouvez l'installer quel que soit votre système (voir l'appendice A pour les détails).

Pour lancer l'application, vous pouvez utiliser, soit le menu des applications, soit le rechercher en tapant le mot Codeblocks. Si vous utilisez une fenêtre terminal sous Linux, vous tapez codeblocks.

Il apparaît une fenêtre, similaire à celle qui se trouve sur la figure 1.1.

Sur le bandeau supérieur se trouvent des menus pour des fonctionnalités génériques (par exemple le menu **File** permet d'ouvrir un fichier existant ou d'en créer un nouveau), le menu **Edit** permet de réaliser les opérations standards que l'on rencontre avec un éditeur) et donc des fonctionnalités spécifiques (**build**, **debug**).

Les avantages de Codeblocks par rapport à la combinaison d'un éditeur basique et une fenêtre terminal pour réaliser la compilation et l'exécution sont les suivants :

- L'éditeur colorise le programme (ce qui le rend plus lisible et donc vérifiable).
- Quand on construit le code, l'éditeur suggère des nomes existants de variables ou de méthodes, ce qui diminue considérablement les fautes de syntaxe.
- On peut générer une indentation qui est aussi très favorable à la lisibilité.
- Pour compiler un programme, un seul raccourci clavier permet de sauvegarder le fichier en cours, de le compiler, et l'exécuter (si la compilation a réussi)

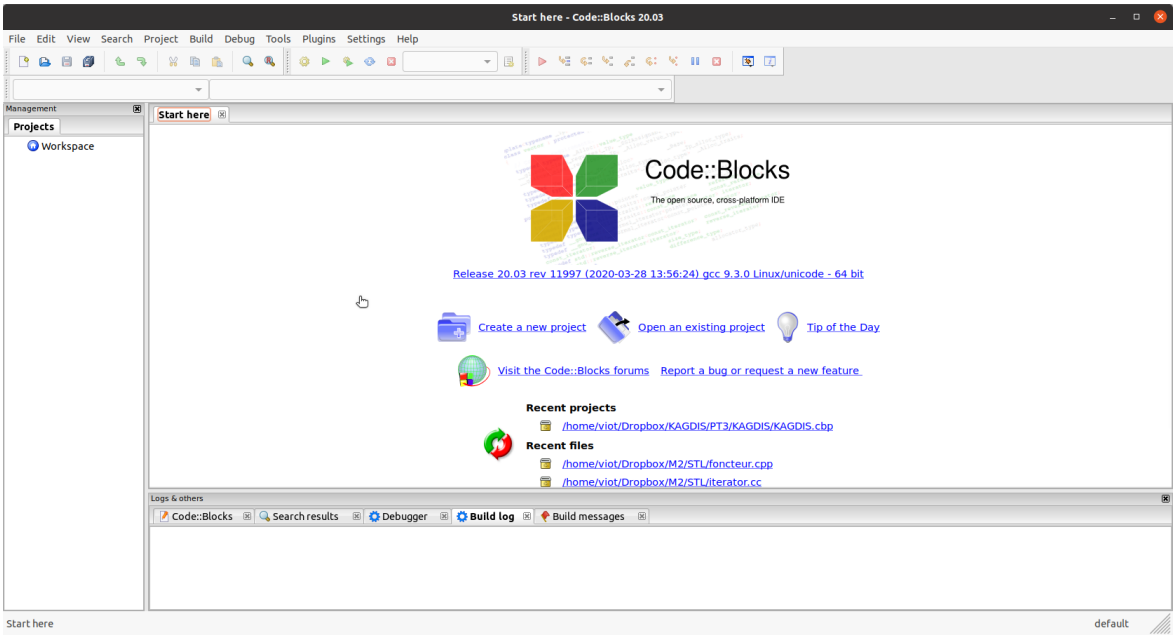


FIGURE 1.1 – EDI Codeblocks

- L'interaction entre compilateur et éditeur conduit à l'affichage des erreurs de code directement dans l'éditeur.

Un premier programme

Pour créer un programme, on a la séquence **File/New** et une fenêtre s'ouvre. On vous demande de préciser le nom du fichier que vous souhaitez créer (dans notre cas, il s'agit d'un programme C++ qui doit avoir un suffixe **.cpp**), ainsi que le répertoire dans lequel il sera enregistré.

Après avoir écrit un code très simple, on peut le compiler et l'exécuter très simplement, en utilisant la séquence **Build/Buid and Run** (qui a le raccourci **F9**).

Le compilateur signale une erreur qui est expliquée dans la fenêtre du bas de CodeBlocks, ainsi qu'une marque dans l'éditeur à la ligne concernée (voir Fig.1.2).

L'erreur de frappe est donc facile à identifier et on ajoute le caractère " **:**" manquant. On presse à nouveau la touche **F9**, on voit que Codeblocks ne signale plus d'erreur et une fenêtre terminal apparaît dans laquelle le code est exécuté. (voir Fig.1.3)

Pour finir cette première prise en main de Codeblocks, le code écrit n'est pas bien formaté. En utilisant la commande **Plugins/Source Code Formatter**, on obtient un code propre. Si la police de caractères vous paraît un peu petite, on peut agrandir celle-ci simplement. Il suffit de la combinaison de touches **ctrl+** pour l'agrandir et **ctrl-** pour la diminuer. (On peut le faire à la souris en appuyant sur **ctrl** et en faisant rouler la roulette de la souris soit vers le haut soir vers le bas) La figure 1.4 donne le résultat suivant.

Les possibilités de CodeBlocks sont très supérieures à celle décrite dans ce tutoriel rapide. Nous verrons plus loin dans ce cours la gestion de projets et l'utilisation d'un débogueur qui permet d'identifier les erreurs d'un programme.

Raccourcis clavier

Pour éviter l'utilisation trop fréquente de la souris, voici un tableau des commandes à travers des touches de raccourcis. Il contient la signification des icônes qui existent dans le bandeau situé sous

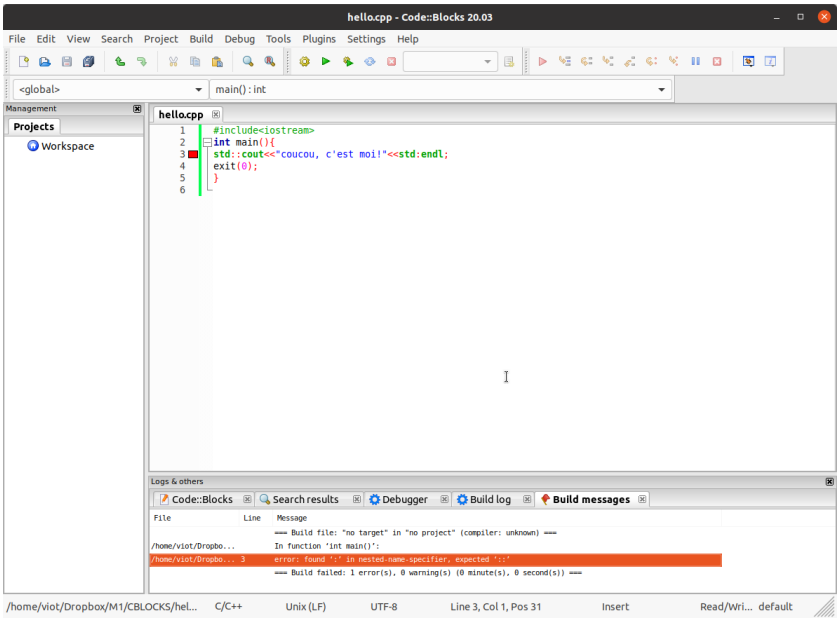


FIGURE 1.2 – EDI Codeblocks : erreur identifiée dans l'éditeur

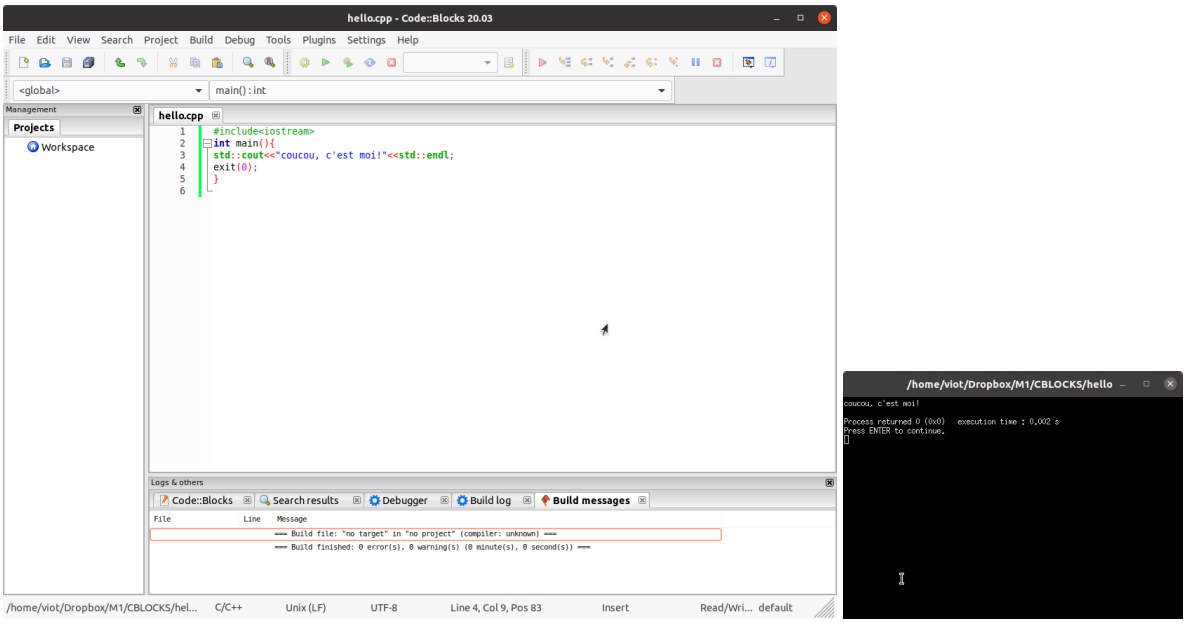


FIGURE 1.3 – EDI Codeblocks

les menus et qui sont aussi des raccourcis de commandes accessibles plus rapidement à la souris que les commandes de menus. Pour les fonctions usuelles (manipulation de fichiers), il s'agit des touches usuelles que l'on retrouve dans la majorité des logiciels et cela évite la mémorisation spécifique associée à un logiciel.

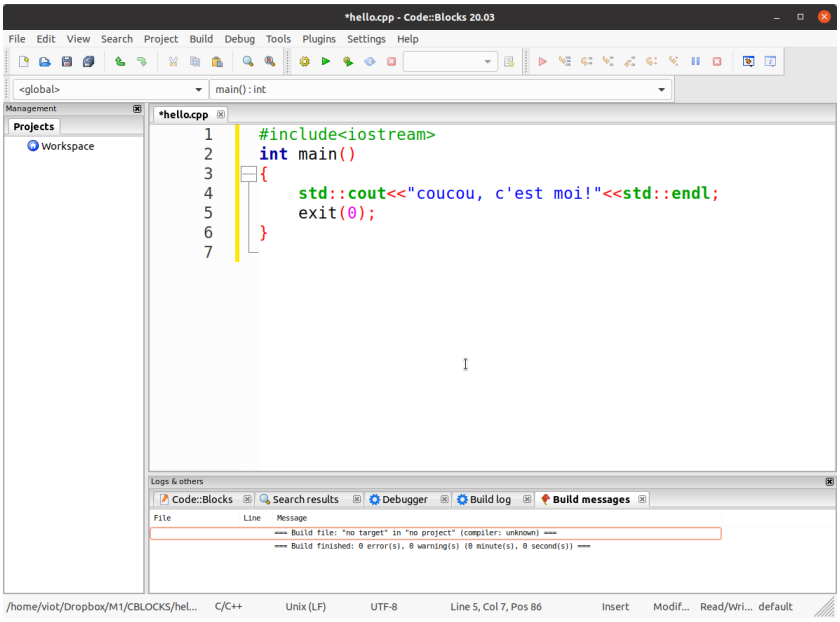


FIGURE 1.4 – EDI Codeblocks : avec le formatage

Commande	Menu	Raccourci clavier	Icone du bandeau
Nouveau Fichier	File/New	Ctrl+n	
Ouvrir fichier	File/Open	Ctrl+o	
Sauvegarde Fichier	File/Save	Ctrl+S	
Copier	Edit/Copy	Ctrl+c	
Coller	Edit/Paste	Ctrl+v	
Couper	Edit/Cut	Ctrl+x	
Annuler Commande	Edit/Undo	Ctrl+z	
Refaire Commande	Edit/Redo	Maj+Ctrl+z	
Panneau "Management" à gauche	View/Manager	Maj+F2	
Panneau "Log & others" en bas	View/Logs	F2	
Compiler et exécuter	Build/Build and Run	F9	
Compiler	Build/Build	Ctrl+F9	

Ces commandes permettent de développer, compiler et exécuter un programme constitué d'un seul fichier. Nous reviendrons plus tard dans ce cours sur la gestion de projets avec CodeBlocks.

Comme vous pouvez le voir, CodeBlocks est un outil logiciel utilisable pour des langages autres que le C++. Pour ceux qui ont déjà la connaissance d'un autre language comme C ou Fortran, vous pouvez bien entendu vous familiariser avec ce logiciel avec votre bibliothèque personnelle de programmes réalisés les années précédentes. La version actuelle est la 20.03 et ne change plus depuis quelques années. Il est probable qu'une autre interface puisse être choisi dans le futur. Toutefois, celle-ci contient les ingrédients de base que l'on retrouve dans les autres interfaces.

1.2.2 Spyder

Spyder est un environnement de développement intégré pour le langage Python. Contrairement à Codeblocks, Spyder est en évolution constante et rapide. La version disponible actuellement est la version 5.4.2 associée une version de Python 3.11. Nous privilégiez cette interface en raison des différentes

fonctionnalités disponibles, en particulier d'un débogueur ainsi que de l'explorateur de variables, permettant de suivre en détail l'exécution de votre code. A nouveau, les avantages de Jupyter permettent une présentation d'une feuille de calcul bien formaté. Il sera sans doute disponible dans les prochains mois et permettra de cumuler tous les avantages. A ce stade, nous utiliserons Spyder5. [1]

SI vous utilisez une installation à partir d'Anaconda ou celle donnée dans une distribution Linux, la version actuelle est la version 3.11 . Il est important de connaître la version de python mais aussi les versions des packages installées car les évolutions sont importantes entre les versions et peuvent provoquer quelques problèmes qu'il faut identifier rapidement.

A titre d'exemple, **scipy**, un module fort utile pour le calcul scientifique évolue très rapidement. par exemple la résolution d'équations différentielles n'est simple qu'à partir de la version 1.4 qui remonte à moins de deux ans. La version actuelle disponible sur la distribution anaconda est la version 1.5.

Pour installer Spyder sur un ordinateur personnel, vous pouvez aller le télécharger sur le site . Si vous utilisez une distribution Linux, votre installateur vous proposera de l'installer à partir d'un paquet de la distribution.

Pour toute plateforme, une distribution Anaconda est disponible et la version actuelle correspond à Python 3.11 et Lles salles informatiques de l'UFR disposent de cette installation et nous utiliserons cette version plutôt que celle de la distribution Ubuntu.

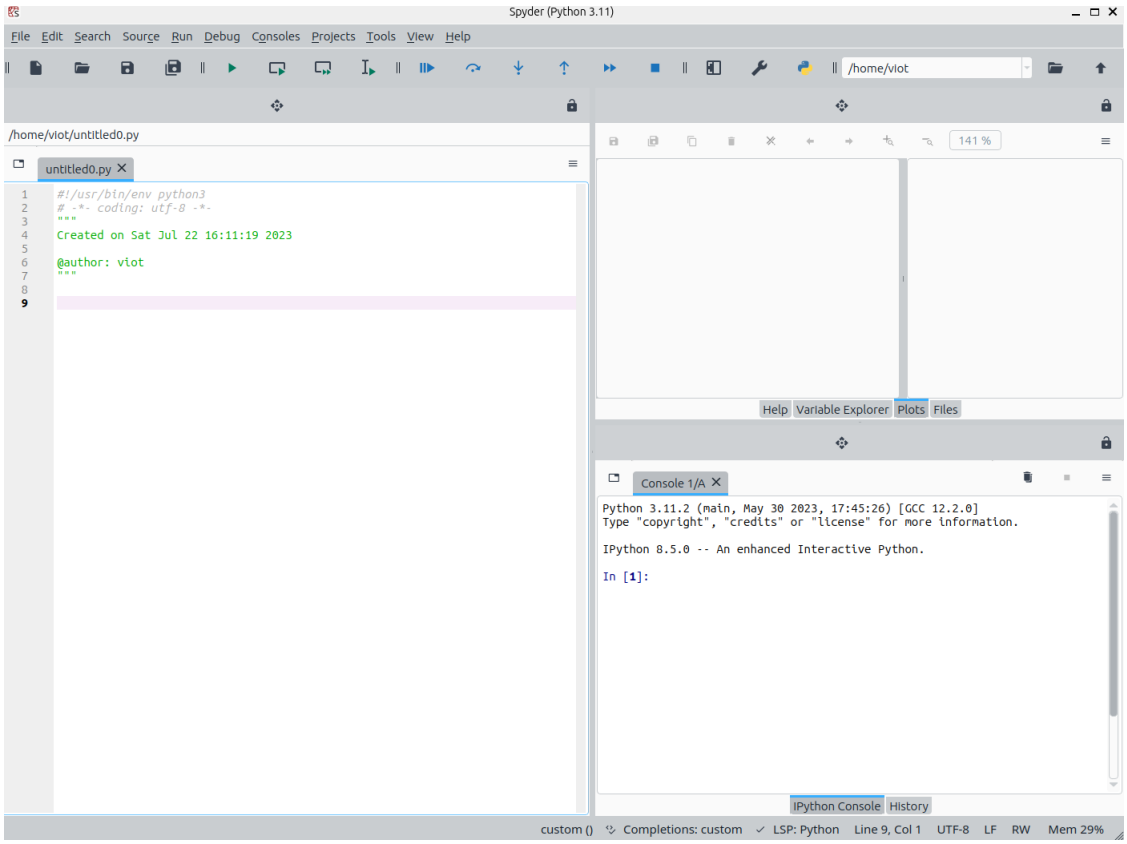


FIGURE 1.5 – EDI Spyder5

Pour lancer l'application, vous pouvez utiliser, soit le menu des applications, soit la rechercher en tapant le mot Spyder. Si vous utilisez une fenêtre terminal sous Linux, vous tapez spyder.

Il apparaît une fenêtre similaire à celle qui se trouve sur la figure 1.5.

1. Les versions 2 de Python sont devenues obsolètes depuis Janvier 2020 car elles ne sont plus maintenues et donc nous allons nous intéresser exclusivement à la version 3.

Sur le bandeau supérieur se trouvent des menus pour des fonctionnalités génériques (par exemple le menu **File** permet d'ouvrir un fichier existant ou d'en créer un nouveau), le menu **Edit** permet de réaliser les opérations standards que l'on rencontre avec un éditeur) et donc des fonctionnalités spécifiques (**Build**, **Debug**).

Les avantages de Spyder par rapport à la combinaison d'un éditeur basique et une fenêtre terminal pour réaliser la compilation et l'exécution sont pratiquement les mêmes que pour CodeBlocks pour le C++

- L'éditeur colorise le programme (ce qui le rend plus lisible) .
- L'indentation indispensable en Python est contrôlée en temps réel (vous voyez les messages d'erreur sur chaque ligne mal écrite
- Pour exécuter un programme, un seul raccourci clavier permet de sauvegarder le fichier en cours et de l'interpréter, voire de l'exécuter.
- L'interaction entre l'interprétateur et éditeur conduit à l'affichage des erreurs de code directement dans l'éditeur.

La fenêtre principale est constituée de trois sous-fenêtres : à gauche l'éditeur, à droite en haut une fenêtre qui peut-être échangée avec six autres fenêtres et permet de contrôler l'état des variables (et donc permet un débogage du programme), une seconde fenêtre pour afficher de l'aide sur une instruction, une autre sur l'aide en ligne, une qui est un explorateur de fichiers, une qui est un explorateur de variables et une autre où les graphes vont s'afficher.

Dans la fenêtre du bas à droite, il y a une première fenêtre dans laquelle le code Python sera exécuté à travers iPython et dans la dernière fenêtre on peut suivre l'historique des différentes commandes.

Un premier programme

Quand Spyder est lancée la première fois l'éditeur est ouvert avec un nom générique de Fichier. Pour le sauvegarder proprement, vous devez utiliser le Menu **File/Save as**. Une fenêtre de type explorateur de fichiers s'ouvre et vous devez sélectionner le répertoire et changer le nom du fichier. Il est utile de choisir un suffixe .py pour le nom du fichier.

Après avoir écrit une seule ligne (parler de code serait à ce stade présomptueux) simple, on peut l'interpréter et l'exécuter très simplement, en utilisant la séquence **Build/Build and Run** (qui a le raccourci F5).

Le résultat de l'exécution apparaît dans la fenêtre du bas à droite (voir Fig.1.6).

Si la police de caractères vous parait un peu petite, on peut agrandir celle-ci simplement. Il suffit de la combinaison de touches **ctrl+** pour l'agrandir et **ctrl-** pour la diminuer. (On peut le faire à la souris en appuyant sur **ctrl** et en faisant rouler la roulette de la souris soit vers le haut soit vers le bas.)

Les possibilités de Spyder sont aussi très supérieures à celle décrite dans ce tutoriel rapide. Nous verrons aussi plus loin la gestion de projets et l'utilisation d'un débogueur qui permet d'identifier les erreurs d'un programme.

Raccourcis clavier

Pour éviter l'utilisation trop fréquente de la souris, voici un tableau des commandes à travers des touches de raccourcis. Il contient la signification des icônes qui existent dans le bandeau situé sous les menus et qui sont aussi des raccourcis de commande accessibles plus rapidement à la souris que les commandes de menus. Pour les fonctions usuelles (manipulation de fichiers), il s'agit des touches usuelles que l'on retrouve dans la majorité des logiciels et cela évite la mémorisation spécifique associé à un logiciel.

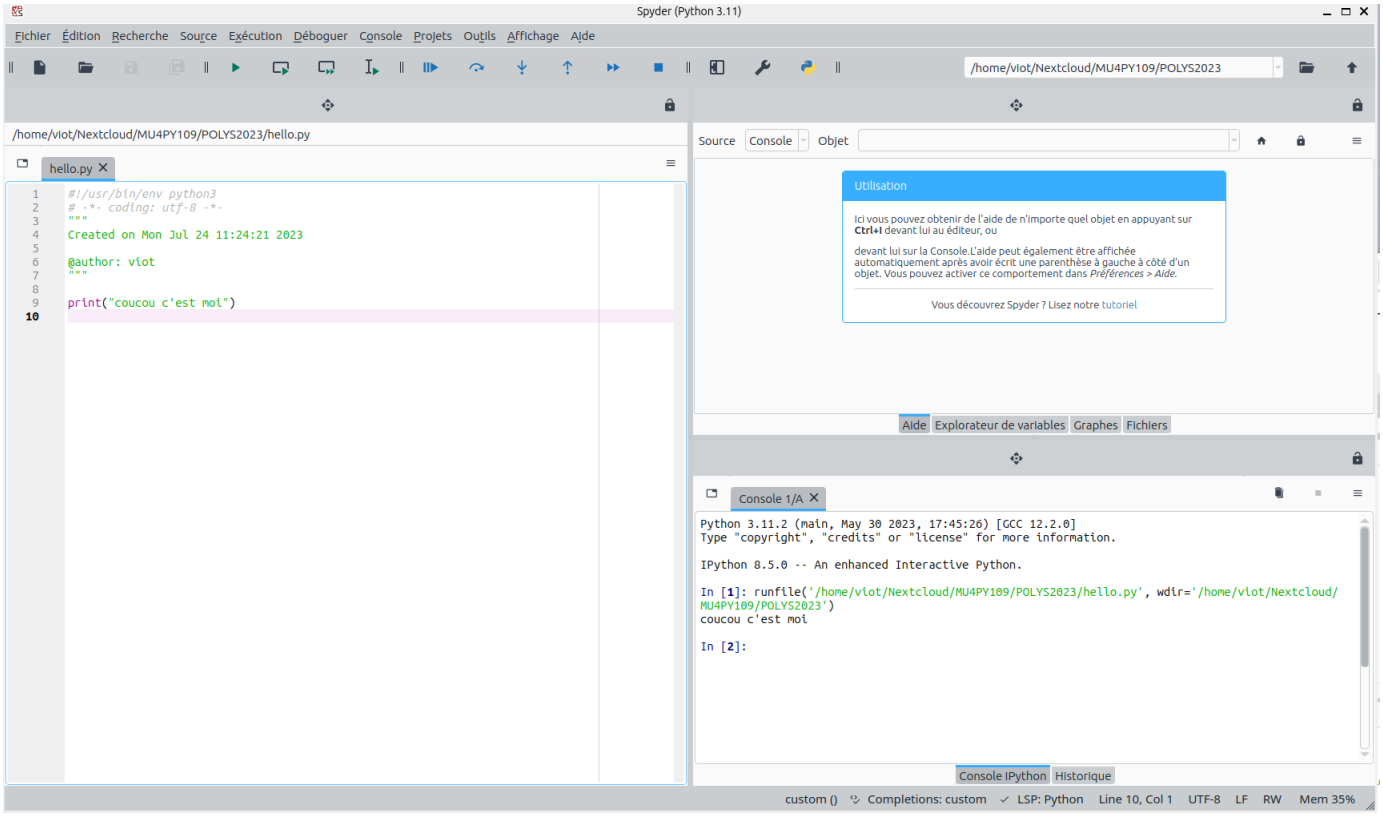


FIGURE 1.6 – EDI Spyder : une ligne de Code !

Commande	Menu	Raccourci clavier	Icone
Nouveau Fichier	File/New	Ctrl+n	
Ouvrir fichier	File/Open	Ctrl+o	
Sauvegarde Fichier	File/Save	Ctrl+Maj+S	
Copier	Edit/Copy	Ctrl+c	
Coller	Edit/Paste	Ctrl+v	
Couper	Edit/Cut	Ctrl+x	
Annuler Commande	Edit/Undo	Ctrl+z	
Refaire Commande	Edit/Redo	Maj+Ctrl+z	
Compiler et exécuter	Build/Build and Run	F5	
Compiler	Build/Build	Ctrl+F9	
Aide sur une commande		Ctrl I sur un mot dejà surligné à la souris	

La figure 1.6 montre l'affichage de l'aide sur la commande **print**.

Comme on peut le constater, un code avec Python peut se réduire à une seule instruction, contrairement au langage compilé. Cela indique une tendance que nous allons vérifier quand les programmes seront constitués de nombreuses tâches et non pas limités à un affichage trivial. A nouveau sur des codes très courts, la comparaison d'exécution entre le C++ et le Python n'apparaît pas comme significative. De manière générale, pour des programmes courts, tous les langages restent simples. Les qualités ou défauts apparaissent de manière importante quand les programmes sont plus longs à la fois pour l'écriture et l'exécution.

Nous reviendrons plus tard dans ce cours sur la gestion de projets avec Spyder4.

1.3 Compilateur, interpréteur et débogueur

Une fois un code écrit dans un langage, il est bien entendu nécessaire de transmettre celui-ci à l'outil logiciel qui va traduite le code en langage machine. Une seconde étape (qui peut être liée à la première) est de lancer l'exécution du programme. Si vous avez écrit un code en Python, ce travail est réalisé par un interprétateur dont la fonction est de traduire, au fur et à mesure, le contenu du code et d'exécuter ses instructions. Dans le cas d'un code écrit en C++, il faut nécessaire d'avoir écrit la totalité du code pour envisager l'exécution du code.

L'intérêt du langage interprété par rapport au langage compilé apparaît maintenant : plutôt que d'attendre d'avoir tout écrit (et surtout bien écrit, c'est à dire sans erreur), le langage interprété permet de tester les différentes étapes de l'évolution du code. La balance qui semble pencher en faveur du Python peut se retrouver dans une situation bien différente assez rapidement. En effet, les performances d'un langage interprété sont généralement entre un ou deux ordres de grandeur plus lentes qu'avec un langage compilé. Au final, le seul temps qui compte est la somme du temps humain passé à développer le code, et le temps passé par la machine à fournir le résultat. Pour le temps humain, cela comprend trois parties : formuler le problème avec les outils fournis par le langage (possibilités du langage, bibliothèques disponibles pour réaliser certaines taches, mise au point et validation du code écrit). Pour le temps machine, le compilateur réalise plusieurs opérations à la suite : la première est une analyse syntaxique du code, ce qui signifie qu'il vérifie que le code est écrit dans les spécifications du langage. En cas d'erreur, le processus est interrompu et la liste des erreurs est donnée par le compilateur. L'interface **Codeblocks** fournit un bonus par rapport à cette étape, car la localisation des erreurs apparaît dans l'éditeur et facilite les corrections à faire. Une situation similaire apparaît sur Spyder pour un code Python. Quand vous écrivez un code, l'analyseur syntaxique est déjà actif et ces erreurs sont signalées sur l'éditeur quand vous êtes en train d'écrire le code.

1.3.1 Options du compilateur

Le compilateur offre la possibilité de générer un exécutable qui est différent selon le compilateur et selon les options que l'on choisit. Quand on utilise Codeblocks. Le compilateur par défaut qui est utilisé est g++.

Si vous utilisez **Codeblocks**, vous voyez les commandes exécutées par le compilateur dans la fenêtre du bas où il apparaît

g++ -c /home/viot/Nextcloud/MU4PY109/POLYS/C++/hello.cpp
-o /home/viot/Nextcloud/MU4PY109/POLYS/C++/hello.o
g++ -o /home/viot/Nextcloud/MU4PY109/POLYS/C++/hello
/home/viot/Nextcloud/MU4PY109/POLYS/C++/hello.o

Pour la première ligne, il y a deux options

- **-c** signifie que le compilateur va transformer le code (correspondant à l'argument suivant) en un nouveau fichier dit fichier objet **hello.o**
- **-o** est l'option qui signifie que le fichier généré aura le nom qui suit cette option. Ainsi, à l'issue de cette commande, il apparaît le fichier hello.o qui est appelé le fichier objet.

Pour la seconde ligne, il utilise le fichier **hello.o** pour le transformer en un fichier exécutable appelé **hello**. Durant cette opération, il utilise le fichier **hello.o** et fait une édition de liens, ce qui signifie qu'il va ajouter au fichier les liens aux bibliothèques dont il a besoin pour l'exécution. Si aucune erreur ne survient, le fichier **hello** généré contient des éléments supplémentaires pour pouvoir être considéré comme un fichier exécutable par le système.

En utilisant la commande système **ldd** (dans une fenêtre terminal), vous pouvez identifier les bibliothèques utilisées par ce programme. En ouvrant une fenêtre terminal (sous Linux) et se plaçant dans le répertoire où se trouve le fichier **hello**, vous tapez ldd hello et la réponse est

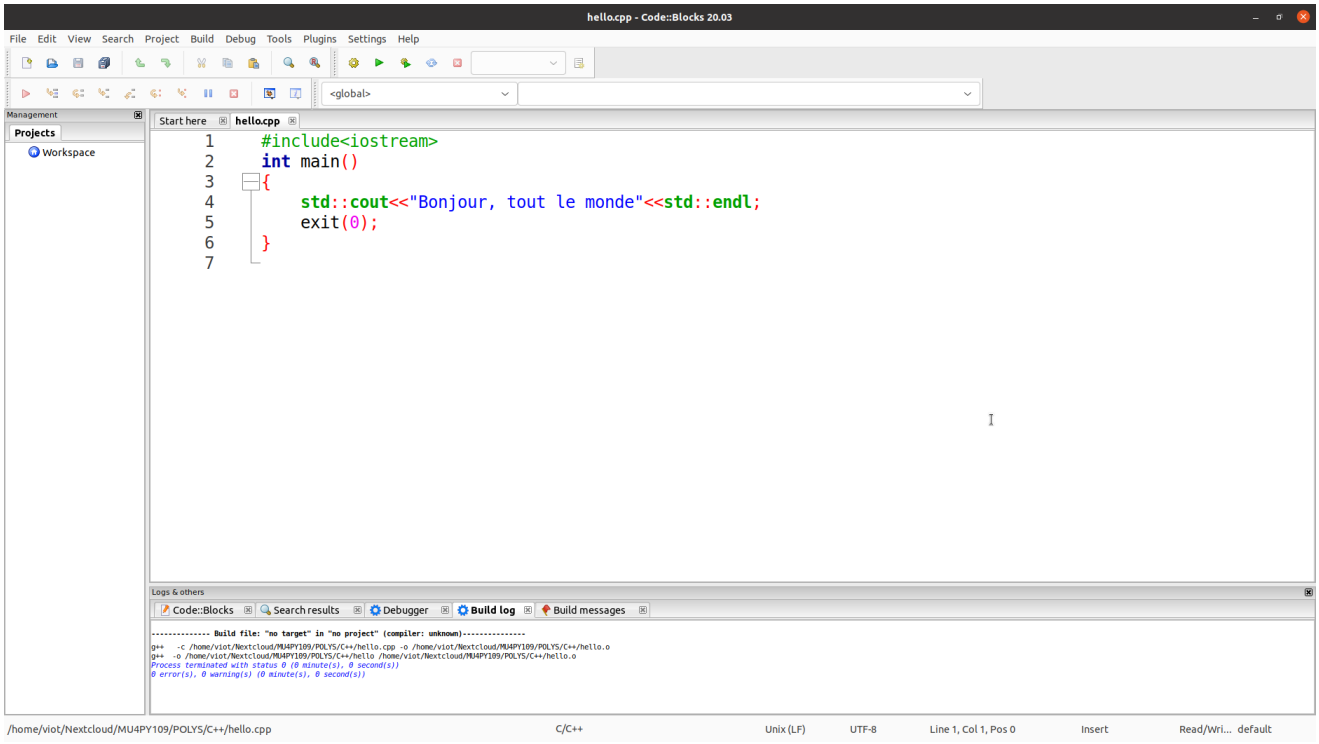


FIGURE 1.7 – EDI Codeblocks : avec le formatage

```
linux-vdso.so.1 (0x00007ffcb85bf000)
libstdc++.so.6 => /lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007f2a9997d000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f2a99791000)
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f2a99643000)
/lib64/ld-linux-x86-64.so.2 (0x00007f2a99bc4000)
libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f2a99628000)
```

Les 5 lignes signifient que le programme fera appel à 5 bibliothèques et qu'il a bien identifié tous les fichiers correspondants sur le système. Si un exécutable ne peut pas démarrer, il apparaît pour l'une des lignes l'expression **not found** qui signifie que la bibliothèque correspondante est manquante.

Il est possible d'avoir pour la compilation des options supplémentaires, comme ajouter une option d'optimisation. Le compilateur analysera le code de manière plus approfondie et va générer un exécutable qui aura de meilleures performances à l'exécution en général.

Pour changer les options, aller dans le menu **Settings/Compiler**, il apparaît une fenêtre avec un ensemble d'options. En cochant la case correspondant à **-03**, puis en validant et en cliquant sur le bouton **OK**, la fenêtre se referme. Si vous compilez à nouveau le programme, vous voyez que la deuxième ligne de compilation dans la fenêtre du bas contient maintenant l'option **-03**.

1.3.2 Le débogueur de C++

Une fois que la compilation a réussi à se faire, l'exécution va produire un résultat qui est parfois éloigné de ce que vous aviez prévu. Soit il s'agit d'un scoop, soit la plupart du temps, le code écrit contient une (ou des erreurs) qu'il faut identifier à l'intérieur du code. Il existe un outil qui va permettre de faire une exécution partielle du code. Pour cela, avant le démarrage, il convient de placer des points d'arrêt qui définissent les différentes étapes où le programme sera à chaque fois stoppé. Il est possible de connaître l'état des variables composant votre code. L'analyse de ces résultats intermédiaires va per-

mettre généralement de trouver l'erreur qui était présente. En la corrigeant et en exécutant à nouveau le programme, on verra les changements produits. Cette tache est réalisée par le débogueur qui s'appelle **gdb** et qui est utilisé par l'interface **Codeblocks** de manière simple.

Pour utiliser le débogueur sous **Codeblocks**, il y a une petite difficulté à surmonter. On ne peut pas utiliser celui-ci sur un fichier, mais il est nécessaire de créer un projet. La procédure est la suivante : à partir du menu **File**, on crée un nouveau projet (**Project**), puis on choisit une **console application** avec le langage C++. On a alors créé un petit fichier **main.cpp** (qui se trouve ci-dessous).

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

int main()
{
    double a[20],b;
    b=1;
    cout<<"b␣"<<b<<endl;
    for (int i=1; i<100; i++)
        a[i]=2*i;
    cout<<"b␣"<<b<<endl;
    return 0;
}
```

Il suffit alors de remplacer ce code par défaut par celui-ci du fichier à déboguer Si on veut passer cette étape, on peut créer, pour chaque nouveau programme, un projet et travailler directement sur la modification du fichier **main.cpp**.

Une fois cette opération réalisée, les principales fonctions du menu **Debug** sont maintenant actives. En utilisant le débogueur, on va pouvoir exécuter le programme, soit pas à pas, soit d'un point d'arrêt à un autre. Il est bien sûr préférable d'utiliser la seconde méthode : on a généralement une idée (même si elle est floue initialement) de l'endroit où l'exécution ne se produit pas telle qu'on l'espérait Pour placer des points d'arrêts, la procédure est la suivante : on met le curseur de la souris sur la ligne où l'on souhaite que le programme s'arrête (temporairement), en utilisant l'instruction **Debug/Toggle BreakPoint**. Il apparaît alors un point rouge sur la ligne concernée, analogue à ce que l'on peut voir sur la figure 1.8

Notez que cette instruction est involutive, puisque, si on sollicite à nouveau la même commande sur la même ligne, on fait disparaître ce point d'arrêt. Il est généralement utile de placer plusieurs points d'arrêt à des endroits stratégiques, c'est-à-dire là où l'on attend à des changements non prévus dans l'exécution du programme. Une fois cette étape réalisée, on peut commencer à faire fonctionner le débogueur. Avec l'instruction **Start/continue** du menu **Debug**, le programme exécute toutes les instructions jusqu'au premier point d'arrêt. Comme cela a déjà été dit, il est possible de faire exécuter pas à pas le code, mais a priori on suppose qu'avant le premier point d'arrêt, le programme se déroule comme prévu. Notez que quand le programme est en pause, il est possible d'obtenir des informations importantes concernant l'état des variables. On peut surveiller celles-ci en utilisant dans le menu **Debug** l'instruction **Debugging Windows/Watches** telle que celle-ci apparaît dans la figure 1.9. En regardant l'état des variables à ce stade, cela permet de voir si les résultats correspondent à ce que l'on espère. Si aucune anomalie n'apparaît, on peut poursuivre l'exécution avec le débogueur, soit en allant jusqu'au prochain point d'arrêt, soit en exécutant pas à pas. Si on a identifié l'erreur (ou la première erreur), on peut alors interrompre l'usage du débogueur avec l'instruction **Stop Debugger**. On peut alors faire la correction souhaitée dans l'éditeur, puis recompiler et exécuter le programme. Si malheureusement, l'exécution ne se passe toujours pas correctement, il faut alors recommencer à utiliser le débogueur. A noter que l'on peut continuer à ajouter ou retirer les points d'arrêt avant de redémarrer une nouvelle exécution avec le débogueur. En général, ce processus converge assez rapidement, car, en suivant l'exécution d'un code de cette manière, on obtient des informations essentielles qui permettent d'identifier l'origine des erreurs. Bien entendu, il s'agit d'une aide au développement, mais comme on peut le voir, il n'y a pas de

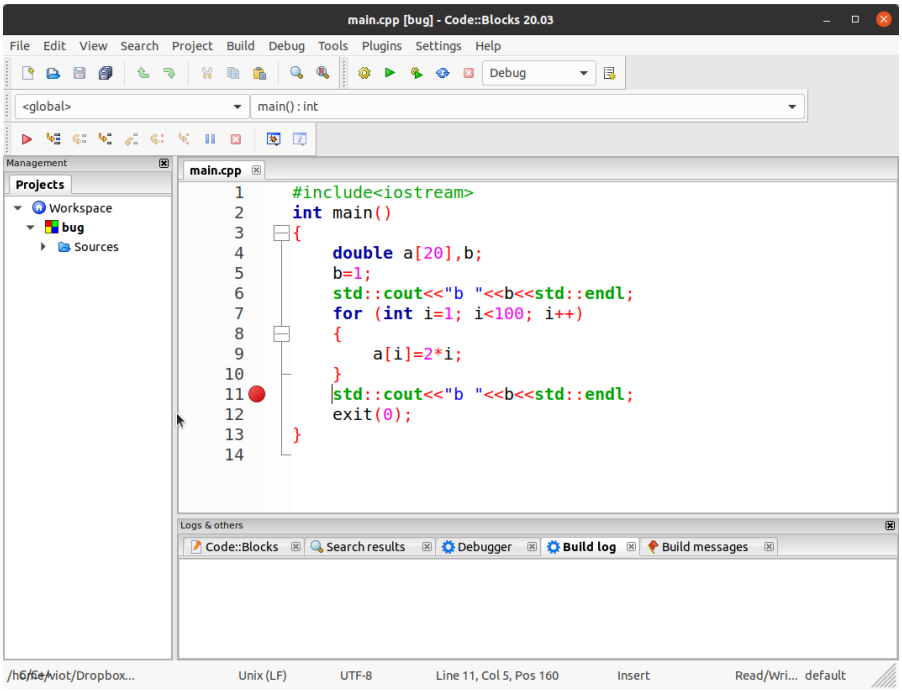


FIGURE 1.8 – Codeblocks : le débogueur

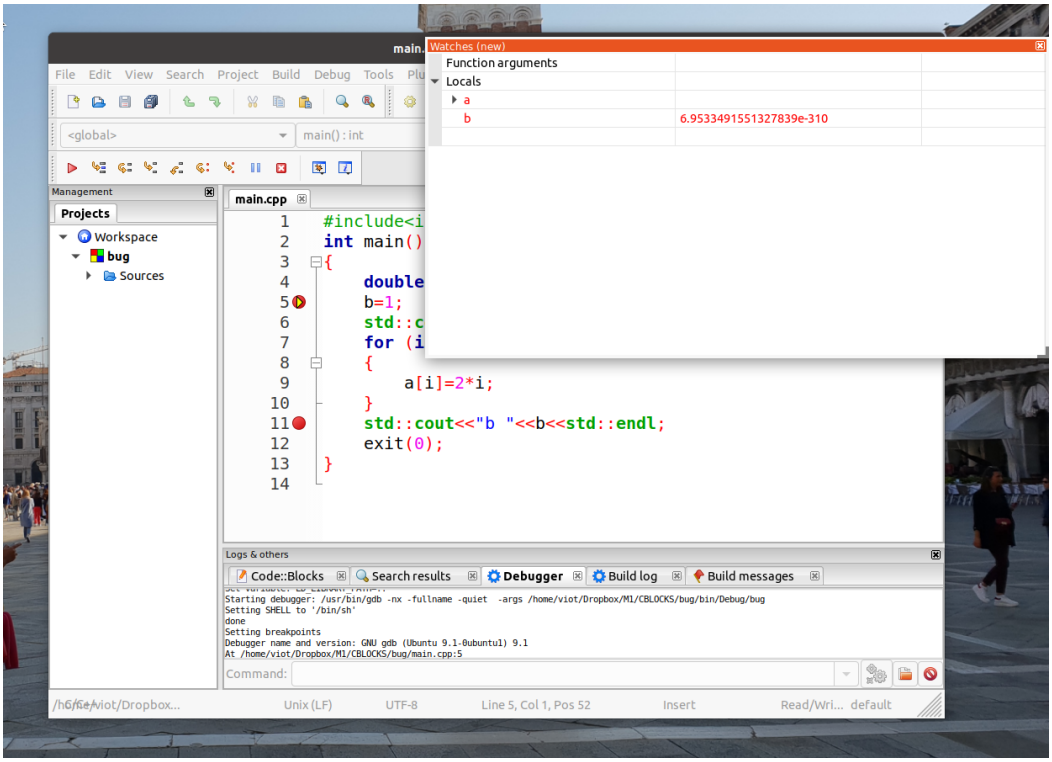


FIGURE 1.9 – Codeblocks : le débogueur

processus miracle où le débogueur écrit un code correct à notre place.

1.3.3 Débogueur de Python sous Spyder

De manière similaire à ce que l'on a vu avec **Codeblocks**, le débogueur est un outil indispensable pour la mise au point d'un code Python. Avec Spyder, son utilisation devient très simple, car il est disponible à partir d'un menu qui s'appelle tout simplement **Déboguer**. Ce menu dispose d'un ensemble de fonctions qui permettent l'exploration du code sous débogueur lors de son exécution.

Déboguer un programme sous Python consiste à utiliser **pdb** , ou **ipdb** sous iPython. Pour passer sous ce mode, il suffit d'activer la commande **Déboguer/Déboguer**. On voit apparaître dans la fenêtre IPython le passage en mode débogueur avec le prompt **ipdb**. Si vous avez détecté un dysfonctionnement de votre code (que bien sûr vous imaginiez parfait au moment de sa conception !), le débogueur va permettre une exécution progressive de votre code et l'affichage de l'état de variables à travers l'explorateur de variables. L'utilisation de ces deux outils va permettre dans la quasi-totalité des cas, de trouver l'origine du problème qui se trouve dans l'écriture. Cela remplace très avantageusement la manière très "vintage" consistant à ajouter des ordres d'impression de variables. Pour illustrer l'usage du débogueur, on considère le programme suivant.

En générant les éléments de la suite de Fibonacci, on cherche à obtenir que le résultat suivant : le rapport de deux nombres consécutifs de la suite de Fibonacci tend vers le nombre d'or.

On rappelle que la suite est définie par la relation de récurrence suivante :

$$u_{n+1} = u_n + u_{n_1} \tag{1.1}$$

avec les valeurs initiales qui sont $u_0 = 0$ et $u_1 = 1$. On cherche à vérifier que

$$\lim_{n\to+\infty} \frac{u_n}{u_{n-1}} = \frac{1+\sqrt{5}}{2}$$

Le code Python à analyser est

```
# -*- coding: utf-8 -*-
"""
Editeur de Spyder

"""
import numpy as np
import matplotlib.pyplot as plt
tab=x=y=np.zeros(40)
tab[1]=1
golden=0.5*(np.sqrt(5)+1)
for i in np.arange(2,40,1):
    tab[i]=tab[i-1]+tab[i-2]
    y[i-2]=np.log(abs(tab[i]/tab[i-1]-golden))
    x[i-2]=np.log(i)
plt.plot(x,y)
plt.show()
```

La figure 1.10 montre l'interface quand on a démarré le débogueur en cliquant sur **Deboguer/Deboguer**. La console IPython fait alors apparaître l'activation du débogueur ipdb. De plus, cette console souligne aussi que le débogueur est au début du programme. Il attend les instructions à lui communiquer.

Pour pouvoir analyser l'évolution du code lors de l'exécution, il est préférable de faire afficher l'explorateur de variables à la place de la page d'aide (fenêtre supérieure gauche de l'interface). Il suffit pour cela de cliquer sur l'onglet correspondant.

On va maintenant placer des points d'arrêt en se plaçant sur la (ou les) ligne(s) où le programme sera stoppé lors de l'exécution puis utiliser, pour chaque ligne, l'instruction du menu **Deboguer/Ajouter/-Supprimer un point d'arrêt**. Si aucun point d'arrêt n'existe, un point rouge apparaît alors dans l'éditeur

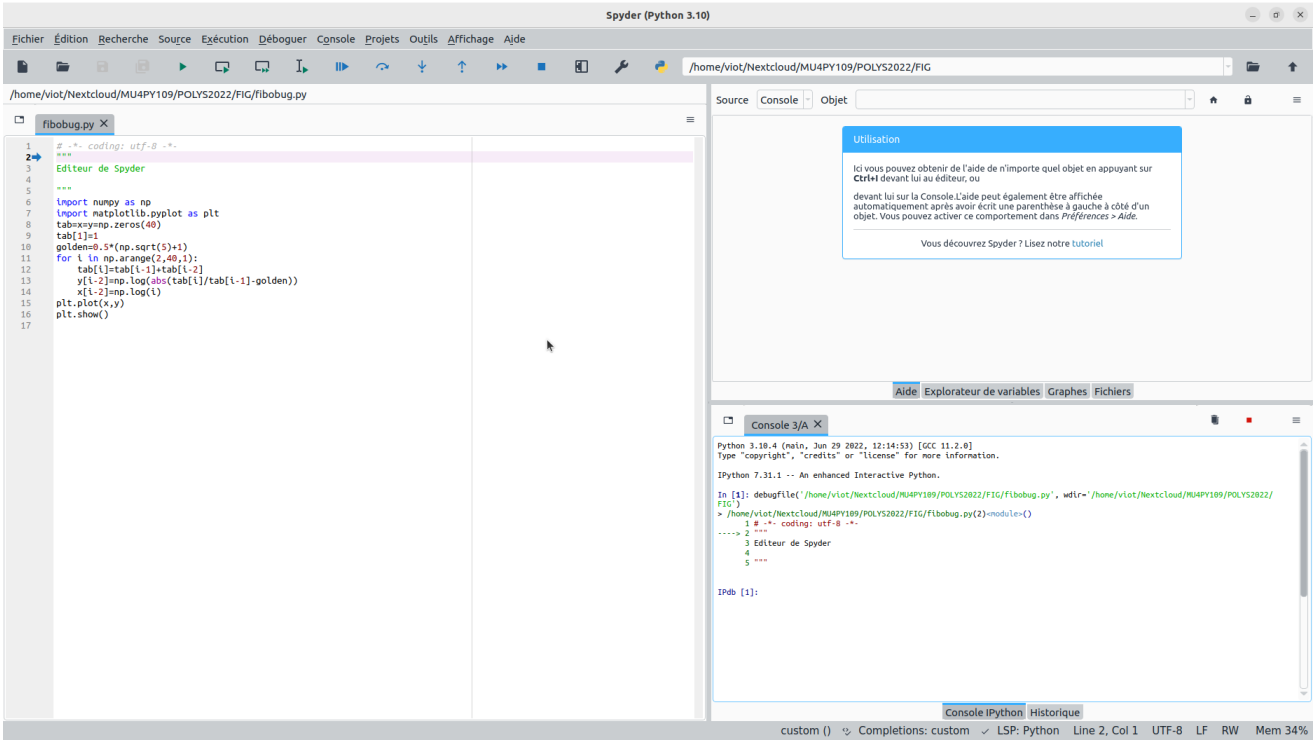


FIGURE 1.10 – EDI Spyder : le débogueur

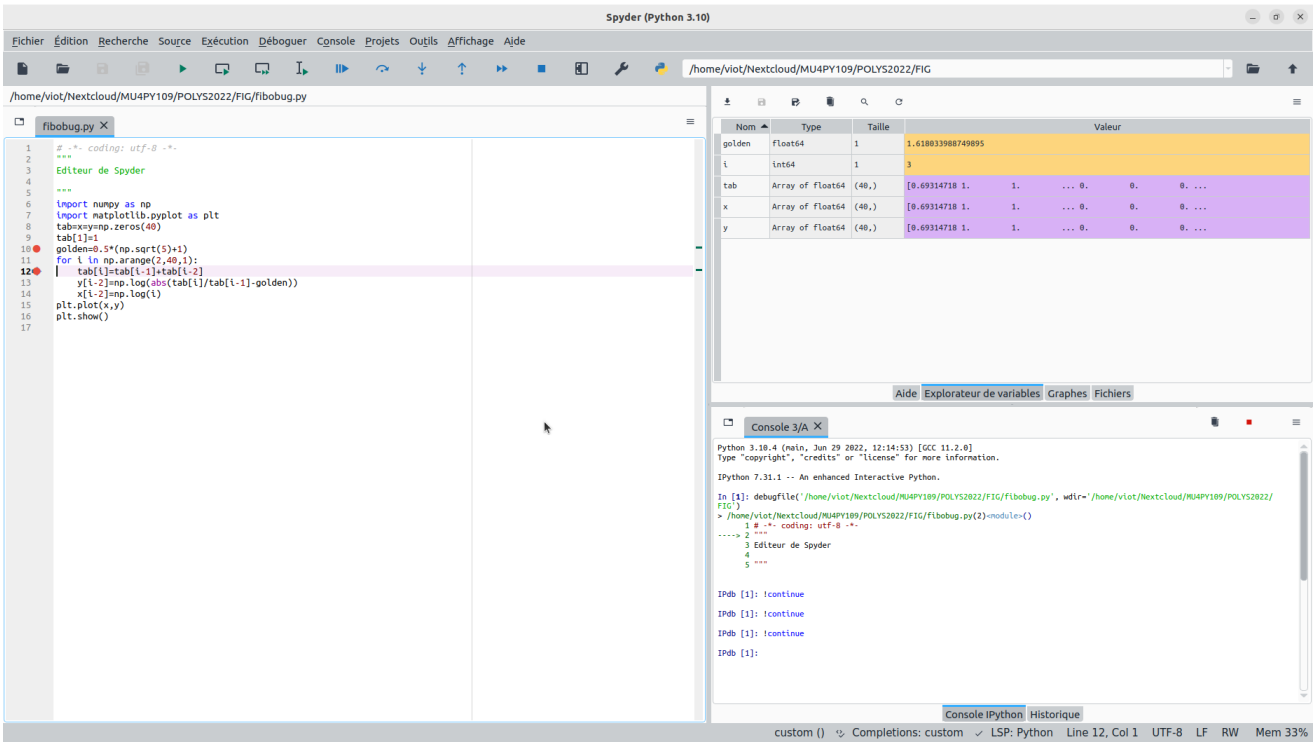


FIGURE 1.11 – EDI Spyder : installation des points d'arrêt

sur la ligne correspondante. Inversement, si un point d'arrêt est déjà présent, celui-ci est alors supprimé.

Sur la figue 1.11, on a placé un point d'arrêt avant la boucle et un second à l'intérieur de la boucle.

Pour commencer l'exécution sous le contrôle du débogueur, on a alors deux possibilités :

- Exécuter le programme de l'endroit où il se trouve (dans notre cas, il est au début du code) jusqu'au premier point d'arrêt Pour cela, il suffit d'utiliser l'instruction **Déboguer/Continuer**.
- Exécuter le programme de l'endroit où il se trouve pour une instruction supplémentaire. Pour cela, il suffit d'utiliser l'instruction **Déboguer/Pas vers l'extérieur**. Il y a même la possibilité de faire un pas en arrière en utilisant l'instruction **Déboguer/Pas vers l'intérieur**.

La figure 1.11 montre l'interface Spyder après avoir exécutée trois fois l'instruction **Deboguer/Continuer**. En analysant l'explorateur de variables, le bug apparaît maintenant clairement. Bien sur, les lecteurs avisés ont vu le problème en regardant attentivement le listing dès le début et n'ont pas eu besoin du débogueur pour trouver l'erreur. Tôt ou tard, on est tous confronté à la correction d'un code (particulièrement quand la longueur du code augmente) et le débogueur est une aide essentielle. Il est inversement possible que l'origine de l'erreur échappe encore à d'autres lecteurs aussi avisés. En continuant d'exécuter le programme sous le débogueur, on finit presque toujours par identifier le problème.

Finalement, une fois que l'on souhaite repasser dans le mode normal, il suffit alors d'utiliser **Deboguer/Arreter**, on voit alors la fenêtre **IPython** quittant le mode **ipdb** pour revenir au mode normal. On peut alors corriger le code et l'exécuter à nouveau. Si la correction faite ne donne pas le résultat souhaité, il faut à nouveau utiliser le débogueur. Ce processus itératif converge alors assez rapidement avec un peu d'usage.

CHAPITRE 2

Le langage Python : programmation procédurale

2.1 Avant-propos

Le langage Python, un langage interprété, est particulièrement bien adapté quand le temps de calcul reste faible, car le temps de développement est plus court en Python que dans un langage compilé. Cette caractéristique repose sur deux éléments : les ordinateurs ont une puissance de calcul qui n'a cessé de croître avec les années et Python est un langage qui s'est beaucoup développé au cours de ces dernières années, à la fois sur le plan conceptuel et par la création de bibliothèques qu'il est facile d'intégrer dans le programme que l'on crée.

Il est fréquent d'avoir commencé l'apprentissage du Python depuis plusieurs années. Le but de ces notes est de rappeler les bases d'utilisation du Python dans une version où l'on va privilégier le développement de code procédural. Toutefois, la base du Python reposant sur la programmation objet, on verra que l'on utilise très souvent cette structure sans connaître les fondements de cette programmation. Cela permettra donc de faire une introduction en douceur de ces concepts, tout en étant en mesure de développer un code où l'essentiel sera la création de fonctions. Dans l'appendice se trouvent plusieurs références sur Python, comprenant des éléments détaillés et d'autres sur les nombreuses fonctionnalités des modules que l'on utilise particulièrement dans le calcul scientifique. Au delà des notes de rappel dans ce chapitre, nous conseillons vivement la lecture de ces références.

Pour l'UE, l'environnement utilisé sera Spyder dont les fonctions essentielles ont été données au chapitre précédent. La première partie de ce chapitre est constituée de rappels élémentaires, destinés à raffraichir la mémoire. Il est nécessaire pour tous ceux qui ont une connaissance trop précaire d'acquérir au plus vite ces notions afin que l'on puisse se consacrer rapidement à des concepts plus élaborés.

2.2 Opérations arithmétiques

Les addition, soustractions, multiplications et divisions ont des règles analogues à celles que l'on rencontre sur des calculettes. Si vous voulez tester simplement, vous tapez dans la fenêtre iPython (en bas à droite) suivi de la touche entrée et vous obtenez le résultat

```
In [1]: 2+3
Out[1]: 5
```

de manière similaire

```
In [2]: 2-3
Out[2]: -1
In [3]:2*3
Out[3]: 6
In [4]: 2/3
Out[4]: 0.6666666666666666
```

Si vous obtenez pour l'opération **In [4]**, la valeur 0 à la place de 0.66.., c'est le signe que vous travaillez sur une version 2 de Python. Passez immédiatement à la version 3, la version 2 est maintenant obsolète !

Si vous souhaitez, vous pouvez utiliser l'éditeur et exécuter le fichier, le programme n'affiche rien et il faut écrire les lignes de la manière suivante

```
print(2+3)
print(2-3)
print(2*3)
print(2/3)
```

On a donc, en Python, la possibilité d'exécuter une seule ligne de code de manière très simple. Bien évidemment, remplacer votre calculette par Python ne serait pas un objectif intéressant, et nous allons voir que l'on peut rapidement aller plus loin.

Les quatre opérations suivantes sont l'exponentiation, la division entière (d'un réel ou d'un entier), le modulo et la valeur absolue.

```
print(2**3)
print(20//3)
print(20.//3)
print(20%3)
```

et on obtient les résultats suivants

```
8
6
6.0
2
```

En fait, la division "entière" s'applique aussi à des nombres réels. A titre d'exemple, vous pouvez essayer la ligne suivante

```
print(3.5*(20.5//3.5)+20.5%3.5)
```

2.3 Variables

Pour résoudre un problème un tout petit peu plus complexe, on a besoin de stocker des éléments dans des variables. En tapant l'instruction

```
a=1
```

on assigne la valeur 1 au caractère *a*, c'est-à-dire que *a* a pour valeur 1. En fait, on fait un peu plus que cela en Python et nous allons voir cela maintenant.

Choisir le nom de la variable est important quand on développe un code qui dépasse les 30 lignes. Même si cela ne concerne pas le début de ce cours, il vaut mieux prendre des bonnes habitudes (pour des raisons de place sur une page, je resterai un mauvais élève pour le poly et j'utiliserai des noms courts de variables). De plus, même si la liberté de choix est grande, il est bon de rappeler ce que l'on ne peut pas faire. Une variable étant une chaîne de caractères

- Elle ne peut pas commencer par un chiffre, mais des chiffres peuvent être présents dans le reste de la chaîne.
- Elle ne peut pas contenir de caractère accentué
- Elle ne peut pas être choisie parmi les mots réservés du Python

and	del	from	None	True	as	elif	is
global	nonlocal	try	assert	else	if	return	finally
not	while	break	except	import	or	with	continue
class	False	in	pass	raise	def	for	lambda
int	complex	float	bool	str			

- Elle ne doit pas contenir de point. On verra que cela a un rôle très important par la suite.

Cela laisse beaucoup de possibilités tout de même ! Tous les mots indiqués sont rejetés par l'interpréteur ; les mots en rouge ne sont pas refusés par l'interpréteur, mais cela introduit un dysfonctionnement majeur dans la suite du code. A proscrire absolument !

En Python, la casse est importante. Les deux lignes illustrent la création de deux variables distinctes en changeant la casse.

```
a=1
A=2
print(a,A)
```

Cet exemple montre l'usage de la deuxième règle

```
a=1
a2=2
print(a,a2)
```

2.4 Typage des variables

Quand on crée une variable, l'interprétateur Python décide du type de la variable. C'est évidemment un avantage par rapport aux langages compilés usuels avec lesquels il faut décider du type de la variable avant ou au moment de l'assignation. (On verra toutefois que les évolutions récentes de C++ permettent aussi de ne pas fixer le type dans le code). La deuxième simplification du Python 3 est de limiter le nombre de types prédéfinis avec un seul type d'entier dont la représentation n'est pas bornée. (La quantité de mémoire de l'ordinateur étant limitée, un entier maximum ou minimum existe, mais avec une valeur gigantesque. Pour vous convaincre de cette représentation, vous pouvez calculer le factoriel de très grands nombres et vous avez la réponse). Le tableau ci-dessous donne les 5 types du Python. (Pour les anciens qui ont commencé avec le Python 2, le type **long** a disparu et le type *int* a été redéfini afin d'avoir une variable de taille quelconque).

mot clé	type	taille mémoire	valeur min	valeur max
int	entier	variable	illimité	illimité
bool	booléen	1 octet	0=**False**	1=**True**
float	réel	8 octets	10^{-308}	10^{308}
complex	complexe	2×8 octets	10^{-308}	10^{308}
str	chaîne de caractères	variable		

Il existe d'autres types en Python, mais nous nous limitons à ceux-ci dans cette première présentation du langage. Pour connaître le type de la variable, il existe la fonction **type** qui renseigne sur la nature de la variable

```
a=2
type(a)
b=1.0
type(b)
c=1+2j
type(c)
```

Notons qu'un nombre complexe est écrit avec le caractère j pour la partie imaginaire. Bien entendu, pour exécuter ces ordres simples, vous pouvez utiliser la console **Ipython** de Spyder qui donnera la réponse pour la fonction type. Si vous placez ces instructions dans l'éditeur, il faut remplacer la ligne **type(x)** où *x* est une variable quelconque par **print(type(x))**.

Une fonctionnalité de **Spyder** permet d'obtenir la réponse sans la modification du code : dans la fenêtre droite supérieure, cliquez sur l'onglet explorateur de variables et vous avez un tableau qui décrit le type de chaque variable. C'est un outil très important qui permet de suivre l'état d'évolution du code très simplement et donc très utile pour la mise au point d'un programme.

2.5 Listes et tableaux

Après avoir assigné des variables, il convient d'être plus efficace en assignant des groupes de variables. Pour cela, les langages proposent généralement de créer des tableaux à une ou plusieurs dimensions. Le principe repose sur le principe suivant : on utilise un nom identique et, avec un ou plusieurs entiers, on a des indices qui repèrent les différents éléments du tableau. Historiquement, les langages traditionnels le proposent pour des éléments qui ont le même type. C'est une vertu essentielle de **Python** de pouvoir mettre au sein d'une même liste des variables de type complètement différents. Le prix à payer est alors une certaine lenteur à l'exécution et nous verrons que le module **numpy** permet, dans le cas où tous les les éléments sont des nombres, entiers ou réels d'exécuter nettement plus rapidement.

2.5.1 Listes

Une liste est une collection d'éléments avec des types différents ou identique. Pour définir une liste, on utilise les crochets pour l'ensemble des objets chacun étant séparé par une virgule.

```
mliste=[14,3.2,'couleur',2+3j]
```

On peut accéder à chacun de ces éléments à partir d'un indice : le premier élément commence à 0, comme dans tous les langages récents.

```
print(mliste[2])
  couleur
```

On peut aussi accéder à l'élément souhaité en partant du dernier avec un indice négatif.

```
print(mliste[-1])
  (2+3j)
print(mliste[-3])
  3.2
```

Quand on crée une liste qui est un objet en Python, on a accès à un ensemble de méthodes donnant accès soit à des informations, soit pour modifier cette liste. Avec **Spyder**, quand vous tapez le nom de la liste suivi d'un point, il apparaît une fenêtre avec un menu contenant la liste des méthodes disponibles avec la liste. Pour ceux qui suivent l'évolution de Spyder dans le détail, la version 3 ne précisait la nature des commandes accessibles. Dans cette nouvelle version, Spyder précise s'il s'agit d'une fonction ou d'une variable. (Voir Fig. 2.1).

Si on veut obtenir cette liste sous une forme texte, on peut aller dans la fenêtre **Ipython** située en bas à droite de Spyder et de taper **dir(mliste)**.

Parmi l'ensemble des méthodes, nous allons voir les plus utilisées :

- **append()**. Ajoute un objet à la fin de la liste
- **count()**. Compte le nombre d'éléments donnés dans l'argument de la méthode
- **insert()**. Insère un nouvel objet dans la liste, à la position donnée par le premier argument et l'objet donné comme second argument
- **reverse()**. Renverse l'ordre des objets.
- **extend()**. Insère une liste à la fin de la liste
- **pop()**. Retire le dernier objet de la liste

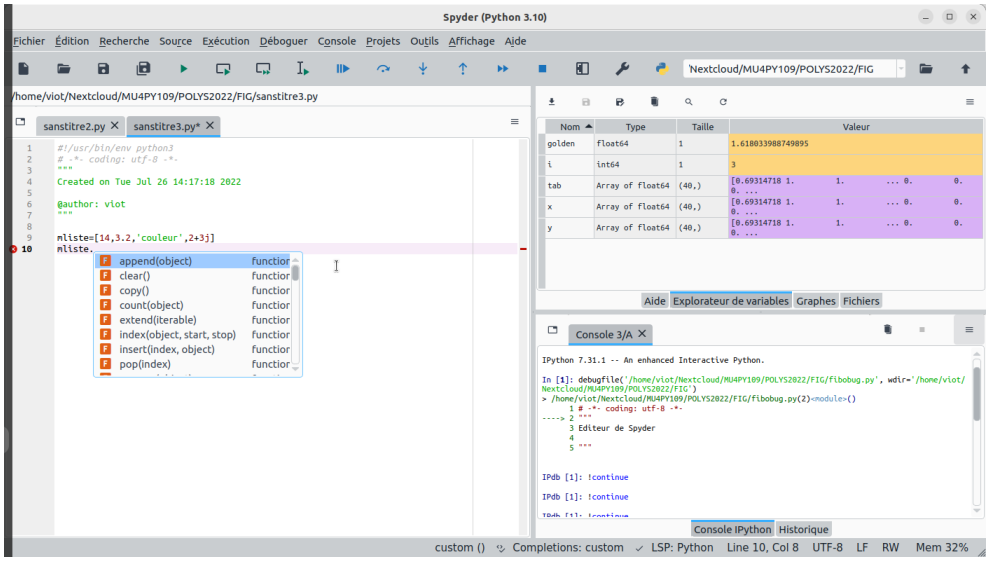


FIGURE 2.1 – EDI Spyder

- **sort()**. Trie les objets s'il ne sont constitués que de type **int** et **float**.
- **copy()**. Crée une liste identique à celle existante
- **remove()**. Enlève le premier objet de la liste donné dans l'argument
- **clear()**. Vide la liste qui est alors vide, mais qui existe toujours.

Le code ci-dessous illustre ces différentes méthodes, hormis **copy** et **sort**.

```
mliste=[14,3.2,'couleur',2+3j]
print(mliste.append(14))
print(mliste.count(14),mliste.count(2),mliste.count(5))
mliste.insert(2,"a")
print(mliste)
mliste.extend([7,10])
mliste.remove(3.2)
print(mliste)
mliste.clear()
print(mliste)
```

La méthode copy permet de créer une deuxième liste identique à la première, contrairement à l'assignation (signe =) qui crée un alias de la précédente. Le code suivant illustre cette différence.

```
mliste=[14,3.2,8,-1.1]
mlistealias=mliste
mlistecopy=mliste.copy()
mliste[-2]=17
print(mliste,mlistealias,mlistecopy)
mliste.sort()
print(mliste)
```

Le résultat est que **mliste** et **mlistealias** ont été modifiées car ce sont deux noms qui désignent la même liste, tandis mlistecopy n'a pas été modifié. Les deux dernière lignes illustrent la méthode de tri.

Nous avons utilisé les méthodes pour modifier les listes, mais on peut le faire par des fonctions, c'est-à-dire de manière procédurale. Le code suivant illustre la duplication de liste et l'initialisation d'une liste avec 10 zéros.

```
mliste=[14,3.2,8,-1.1]
mliste=3*mliste
print(mliste)
mliste2=10*[0]
print(mliste2)
```

Une fois ce code exécuté, il faut aller voir l'explorateur de variables qui renseigne sur l'état de chacune des variables.

On peut généraliser le concept de liste en créant des listes de listes. On peut accéder à chacun des objets en utilisant plusieurs indices correspondant à la "profondeur" de l'élément dans la liste.

```
mliste=[[14,3.2,4],[8,-1.1,7]]
print(mliste[0][2],mliste[1][1])

mliste2=[[14,3.2,[3,5]],[8,-1.1,7]]
print(mliste2[0][2][1],mliste2[1][1])
```

Pour le premier exemple, on entrevoit la possibilité de créer une matrice. Pour le deuxième exemple, l'objet est plus complexe, car il nécessite deux ou trois indices pour accéder à un élément.

Il est possible d'utiliser en Python un moyen pour récupérer des parties d'une liste de listes que l'on appelle le slicing en anglais et que l'on peut traduire comme tranchage ou saucissonnage, que je préfère.

Pour avoir sur une liste, tous les éléments correspondant hormis le premier on utilise **:-1**, pour avoir tous les éléments correspondant au deuxième indice, sauf le premier, on utilise **1 :**, pour avoir les éléments allant de l'indice **n+1** à l'indice **m**, on écrit **n :m**

```
mliste=[14,3.2,4,8,-1.1,7]
print(mliste[:-1],mliste[1:],mliste[2:5])
print(len(mliste))
```

La fonction **len** permet d'avoir le nombre d'éléments d'une liste

On peut aussi appliquer une fonction sur une liste. L'exemple suivant illustre le fait de prendre la valeur absolue sur une liste en créant une deuxième liste. Il est nécessaire de combiner deux fonctions **map** et **list**

```
mliste=[14,3.2,4,8,-1.1,7]
print(mliste[:-1],mliste[1:],mliste[2:5])
print(len(mliste))
mliste2=list(map(abs,mliste))
```

2.5.2 Tableaux

Les tableaux sont des objets très utiles dans le calcul scientifique et le module numpy est la bibliothèque spécifique qui permet à la fois de créer et de manipuler efficacement des tableaux de nombres.

Pour utiliser une bibliothèque il est nécessaire en Python de demander à l'interpréteur d'aller chercher cette bibliothèque (si un message d'erreur apparaît, cela signifie que la bibliothèque n'est pas installé sur votre système. Il faut alors l'installer). Par défaut il a un nombre limité d'objets et de fonctions disponibles. Pour réaliser ce chargement, il y a plusieurs méthodes mais on en privilégie une :

```
import numpy as np
```

L'instruction **import** attend le nom de la bibliothèque à utiliser. Dans l'exemple il s'agit de **numpy**. La suite de la ligne est optionnelle et elle signifie que tous les éléments de la bibliothèque **numpy** seront appelés avec un alias court **np**. Cela simplifie l'écriture du programme. On verra par la suite que l'on peut éventuellement importer une partie d'une bibliothèque et non pas la totalité, ce qui évite de charger inutilement la mémoire de votre ordinateur.

Tableaux à une dimension

Il existe plusieurs façons de créer un tableau à une dimension

- Créer un tableau avec un type donné, mais qui n'est pas initialisé.
- Créer un tableau de réels initialisés à zéro
- Créer un tableau de réels initialisés à un
- Créer un tableau de réels, avec des nombres allant du premier argument au deuxième argument (exclu) avec un pas donné par le troisième argument
- Créer un tableau de réels, avec des nombres allant du premier argument au deuxième argument avec un nombre d'éléments donné par le troisième argument.

Le code suivant illustre les différentes méthodes.

```
# -*- coding: utf-8 -*-
"""
Editeur de Spyder

"""
import numpy as np

tab=np.ndarray(100,dtype=int)
print(tab)
tab1=np.zeros((3,3))
print(tab1)
tab2=np.ones(100)
print(tab2)
tab3=np.arange(0.0,10.0,0.1)
tab4=np.linspace(0,10,100)
print(tab3)
print(tab4)
```

Noter la différence entre les tableaux **tab3** et **tab4**. Modifier le code pour que les deux derniers tableaux soient identiques.

Tableaux à plusieurs dimensions

Les tableaux à plusieurs dimensions peuvent être créés de manière similaire aux méthodes précédentes. Le code suivant correspond à la création de tableaux à deux dimensions. Les dimensions supérieures peuvent être déduites de ce code.

```
# -*- coding: utf-8 -*-
"""
Editeur de Spyder

"""
import numpy as np

tab=np.ndarray(100,dtype=int)
print(tab)
tab1=np.zeros((3,3))
print(tab1)
tab2=np.ones(100)
print(tab2)
tab3=np.arange(0.0,10.0,0.1)
tab4=np.linspace(0,10,100)
```

```
print(tab3)
print(tab4)
```

Numpy offre des possibilités très intéressantes pour manipuler les tableaux. A partir d'un tableau à une dimension, on peut créer un tableau à deux dimensions avec la méthode **reshape**.

```
tab3=np.arange(1,17,1)
print(tab3)
tab4=tab3.reshape(4,4)
print(tab4)
print(tab4[1:-1,1:-1])
```

Le tableau **tab4** est un tableau à deux dimensions (et non pas une liste de listes). Notez en particulier la différence de syntaxe pour accéder un élément du tableau. Ainsi, pour avoir la valeur de l'élément de la ligne i et de la colonne j de cette matrice, il suffit d'écrite **tab[i,j]**. Mais on peut aussi utiliser le saucissonnage pour récupérer une sous-matrice en appliquant celui-ci sur chacun des indices.

Il y a plein d'autres possibilités avec **numpy** et nous y reviendrons plus tard.

2.6 Les instructions itératives

Maintenant que l'on dispose d'objets qui sont indexés, il est utile de réaliser des opérations répétitives sur l'ensemble des éléments qui constituent la liste ou le tableau. Comme c'est le cas dans tous les langages, nous allons voir la syntaxe des boucles qui le permettent. Toutefois, en Python, parcourir une boucle consiste à interpréter les instructions à l'intérieur de la boucle à chaque itération et donc cela ralentit très souvent l'exécution Dans le cas des tableaux, nous verrons des méthodes alternatives développées dans la bibliothèque **numpy** pour accélérer l'exécution.

Dans le cas où le nombre d'itérations est connu à l'avance (c'est-à-dire qu'il ne dépend pas d'une condition logique), on utilise la boucle **for**. Le code suivant initialise une liste de 10 éléments en remplissant une liste avec le carré de la valeur de l'indice.

Explication du code : **range(10)** crée une liste allant de 0 à 9. La valeur *i* parcourt avec la boucle **for** l'intégralité de cette liste. Le caractère **:** est indispensable pour signifier que les instructions qui suivent constitue le bloc de la boucle for. Elles doivent être toutes indentées soit d'une tabulation, soit de 4 espaces. La méthode **append** ajoute l'argument comme valeur de l'élément ajouté à la fin de la liste. L'instruction **print(tab)** affichera le résultat et doit être en dehors de la boucle ce qui explique son absence d'indentation.

```
tab=[]
for i in range(10):
    tab.append(i*i)
print(tab)
```

A nouveau, il est très intéressant de regarder l'explorateur de variables pour suivre les résultats de ce mini-code en détails.

La deuxième situation où l'on a besoin d'effectuer des instructions de manière répétitive est celle dont le nombre d'itérations n'est pas connu à l'avance car elle dépend d'une condition logique dont l'état initial est **True**, mais finira par devenir **False**. La boucle **while** permet le traitement de cette situation. Le code suivant correspond au calcul des premiers termes de la suite de Fibonacci définie de la manière suivante

$$u_{n+1} = u_n + u_{n_1}$$

avec les conditions initiales $u_0 = 0$ et $u_1 = 1$.

```
tab=[0,1]
i=1
```

```
while (tab[i]<100):
       tab.append(tab[i]+tab[i-1])
       i=i+1
print(tab)
```

Explication du code : on initialise un tableau avec deux éléments 0 et 1 la condition logique vérifie si l'élément courant (la première fois il s'agit de **tab[1]**) est inférieur à 100, le résultat est une variable booléenne qui est vrai ou fausse. Si la condition est vraie, ce qui est le cas la première fois, la boucle peut commencer. A nouveau le caractère **:** signifie que les instructions de la boucle commenceront la ligne suivante avec un décalage. La méthode **append** ajoute un élément à la liste en utilisant la relation de récurrence

2.7 Les instructions conditionnelles

L'étape suivante pour construire les outils de la programmation concerne les instructions conditionnelles qui sont présentes dans tous les langages. Nous allons donc essentiellement présenter la syntaxe plutôt que de présenter des concepts connus dès que l'on a déjà étudié un autre langage.

```
if condition:
       instructions1
else:
       inscructions2
```

La structure de base est la suivante : le mot clé **if** est suivi de la condition qui est vraie ou fausse et du caractère **:**. Dans le cas ou la condition est vraie, le groupe d'instructions1 (qui est donné avec une indentation d'au moins 4 caractères) est exécuté. Le mot clé **else** suivi du caractère **:** annonce que les instructions2 seront alors exécutées dans le cas où la condition logique est fausse. Dans le cas où **else** n'est pas présent, aucune instruction n'est exécutée quand la condition est fausse.

Pour des raisons historiques, il n'existe pas de groupe de structure conditionnelle (switch case) analogue à celui que l'on a en C++. Il est possible de le substituer en utilisant une cascade d'instructions précédentes de la manière suivante

```
if(x < y):
       st= "x␣is␣less␣than␣y"
elif (x == y):
       st= "x␣is␣same␣as␣y"
else:
       st="x␣is␣greater␣than␣y"
```

ce mini-code contient le mot clé **elif** suivi d'une seconde condition et d'un caractère **:**. Dans le cas où la deuxième condition est vraie (et donc où la première est fausse) la seconde instruction est exécutée. Si les conditions sont fausses le mot clé **else** suivi du caractère **:** permet de définir les instructions exécutées quand les deux première conditions sont fausses. En construisant une cascade de séquences **if elif ...elif else** on peut obtenir une sélection d'instructions par rapport à une séquence de conditions.

2.8 Opérateurs d'assignation et d'incrémention

Nous avons vu précédemment que l'opérateur = n'est pas seulement un opérateur d'assignation, qui signifie que le résultat du membre de droite est placé dans la mémoire de la variable donnée dans le membre de gauche mais c'est en quelque sorte une fonction d'initialisation d'un objet. Avec cet opérateur, on définit le type de la variable et on donne une valeur à l'objet ainsi défini.

```
a,b=1,5
```

permet donner la valeur 1 à **a** et la valeur 5 à **b**.

On peut surtout se permettre d'échanger des variables à partir de l'instruction suivante

```
a,b=b,a
```

Pour ceux qui ont travaillé avec des langages compilés, cela permet une écriture très élégante du code. On sait qu'il faut en fait une variable temporaire pour faire ce type d'échanges mais Python laisse ce travail à l'interpréteur au lieu d'ennuyer le programmeur.

Cet exemple est aisément généralisable à une séquence plus grande de variables. Dans l'exemple suivant, on réalise une permutation circulaire sur 3 variables.

```
a,b,c=b,c,a
```

Les opérateurs d'incrémentation étant empruntés au C, on garde la même syntaxe en Python. Plusieurs autres opérateurs permettant d'incrémenter une variable. (Les opérateurs unaires ++ et – ne sont pas présents)

+=	ajoute le membre de droite au membre de gauche
-=	soustrait le membre de droite au membre de gauche
*=	multiplie le membre de droite par le membre de gauche
/=	divise le membre de droite par le membre de gauche

2.9 Opérateurs de comparaison

Pour formuler une expression logique, il est souvent nécessaire d'avoir un opérateur de comparaison qui s'applique à deux expressions de même type. Le langage Python fournit plusieurs opérateurs que nous rassemblons ci-dessous

<	inférieur(strictement) à
>	supérieur (strictement) à
<=	inférieur ou égal à
>=	supérieur ou égal à
!=	différent de
==	égal à

Il existe aussi des opérateurs de comparaison logique en Python. Ils sont différents du C et C++

and	et logique
or	ou logique

2.10 Fonctions

L'étape suivante consiste à utiliser ou créer des fonctions. C'est aussi le cœur de la programmation procédurale. Pour créer une fonction, les règles sont les suivantes

- On utilise le mot clé **def** suivi du nom qui définit la fonction
- On écrit entre parenthèses la liste des arguments dont la fonction a besoin en tant que données.
- En utilisant une indentation d'au moins 4 caractères, on écrit la suite des instructions qui doivent être exécutées dans la fonction

- On utilise l'instruction **return** pour donner les différents arguments que doit retourner la fonction

Le mini-code suivante donne la définition d'une fonction pour une exponentielle amortie

$$f(t) = e^{-\alpha t}\cos(\omega t)$$

```
import numpy as np
def oscil_amorti(t,alpha,omega):
       f=np.exp(-alpha*t)*np.cos(omega*t)
       return(f)
```

Ce qui est très puissant en Python, c'est que la fonction que l'on a défini va pouvoir s'appliquer à différents objets. Si on exécute les instructions suivantes,

```
x=np.linspace(0,10,10)
print(x,oscil_amorti(x,0.1,2)) #on obtient un tableau de reels
print(oscil_amorti(1,0.1,2)) #on obtient un reel
print("bool",oscil_amorti(True,0.1,2)) #on obtient le meme reel (le boolean est converti en 1)
print(oscil_amorti(1-5j,0.1,2))#on obtient un complexe
```

On obtient le résultat

```
[ 1. -0.5425598 -0.21199513 0.664488 -0.55129652 0.06614227
0.36967147 -0.45410035 0.19674341 0.150125 ]
-0.3765452291051488
bool -0.3765452291051488
(-7983.544514583844+5963.8857439885705j)
```

Le résultat de la fonction est du type correspondant à celui utilisé par le calcul. C'est un très grand avantage par rapport aux autres langages qui nécessitent de définir une fonction par type. Il faut quand même faire attention à ne pas demander des calculs impossibles

```
print(oscil_amorti('a',0.1,2))
```

donne l'erreur suivante

```
'TypeError': can't␣multiply␣sequence␣by␣non-int␣of␣type␣'float'
```

Il est aussi possible de définir cette fonction en donnant par défaut des valeurs aux paramètres par défaut de la manière suivante

```
def oscil_amorti(t,alpha=0.2,omega=2):
    f=np.exp(-alpha*t)*np.cos(omega*t)
       return(f)
```

Cela permet de pouvoir utiliser la fonction avec un nombre d'arguments entre 1 et 3

```
x=np.linspace(0,10,10)
print(oscil_amorti(x)
print(oscil_amorti(x,0.2))
print(oscil_amorti(x,0.2,2))
```

Dans le premier appel, on utilise pour ω et α les valeurs par défaut et la fonction est calculée. Si vous essayez le même appel avec la définition précédente, **Python** donne une erreur, car il a besoin de trois paramètres.

La deuxième instruction change la valeur de α dans le calcul de la fonction mais garde la valeur par défaut pour le paramètre α. Pour la troisième instruction, les deux paramètres par défaut sont réinitialisés.

Une dernière possibilité s'offre avec la manière dont la fonction a été construite

```
print(oscil_amorti(x,omega=2,alpha=0.2))
```

On peut inverser l'ordre des paramètres en appelant la fonction en donnant l'assignation dans l'argument. Ce n'est pas a priori une manière recommandée de procéder, mais c'est parfois très utile quand on a un doute sur l'ordre des paramètres, car on s'assure que l'initialisation sera réalisée correctement, même si on se trompe sur l'ordre de saisie.

Un dernier point concernant les fonctions est l'introduction d'un commentaire. Il se place juste après la première ligne, commence et finit par la séquence """.

```
def oscil_amorti(t,alpha=0.1,omega=2):
      """ exponentielle decroissante oscillante"""
      f=np.exp(-alpha*t)*np.cos(omega*t)
      return(f)
```

Pour ce programme ultra-court, cela ne semble pas spectaculaire mais si vous tapez dans la fenêtre IPython l'instruction

```
oscil_amorti?
Signature: oscil_amorti(t, alpha=0.1, omega=2)
Docstring: exponentielle decroissante oscillante
File: ~/Dropbox/M1/NMNI/fonction2.py
Type: function
```

Vous obtenez à la fois la manière dont vous pouvez appeler la fonction, mais aussi le commentaire apparaît dans la réponse Doctring. Quand vous importez des modules personnels ou ceux d'une bibliothèque, vous avez ainsi un moyen très simple d'obtenir une information sur la fonction.

2.11 Entrée et Sortie d'un programme

2.11.1 Clavier et Écran

Il est bien nécessaire que le résultat produit par un programme soit communiqué à l'utilisateur. En Python, comme il s'agit d'un programme interprété (et comme les programmes que l'on considère sont plutôt courts), le plus simple est de définir les grandeurs d'entrée au début du code et de les actualiser à chaque nouvelle exécution. Il existe toutefois une instruction pour récupérer des éléments tapés au clavier que nous allons voir par souci d'être complet, mais dont je ne recommande pas l'usage. Inversement, il existe une instruction pour afficher des résultats en utilisant **print**. Rapidement, cela ne présente que peu d'intérêt, car il vaut mieux lire des données d'entrée sur un fichier que de faire une saisie fastidieuse au clavier (au delà de 3 nombres, c'est inutile et source d'erreur). De même, dès qu'une code produit des résultats non triviaux, on n'a pas quelques nombres, mais un ou plusieurs tableaux qu'il est plus intéressant d'afficher graphiquement ou de stocker sur un fichier pour un post-traitement que d'afficher à l'écran des tableaux de nombres.

Le mini-code suivant réalise cette action.

```
print('donner␣un␣nombre')
a=input()
print('voila␣le␣resultat␣de␣2+ce␣nombre',2+int(a))
```

En fait Python récupère une chaîne de caractères et on la convertit pour faire l'addition.

2.11.2 Fichiers et stockage

L'étape suivante est de pouvoir connaître des moyens de lire et de stocker des données sur un support physique local (en un mot le disque dur de votre ordinateur).

La démarche par défaut est de créer un descripteur de fichier, (l'option **w** permet de contrôler que le fichier défini ci-dessus est celui sur lequel on écrira). Si ce fichier existe déjà, le contenu précédent est effacé, sinon il sera créée avec le contenu présent.) puis on écrit à chaque itération une chaîne de caractères avec la méthode **write** et finalement on ferme le fichier Le code suivant réalise l'écriture d'un ensemble de deux colonnes (deux vecteurs) sur un fichier appelé fichier.txt.

```
import numpy as np
x=np.linspace(0.1,10,200)
y=np.log(x)
mon_fichi = open("fichier.txt", "w")
for i,j in zip(x,y):
       print(i,j)
       mon_fichi.write("{0}␣\t␣␣{1}\n".format(i,j))
mon_fichi.close()
```

Notez que, pour la boucle **for**, on utilise la fonction **zip** qui "accole" les deux vecteurs de manière à ce que la boucle puisse contenir deux variables qui sont incrémentés conjointement. L'instruction **write** attend une chaîne de caractères. Comme ceux-ci changent à chaque itération, on place entre accolades un numéro de la valeur de la variable à placer. Finalement, on utilise la méthode **format** d'une chaîne de caractères en donnant la séquence de variables que l'on souhaite écrire. Une fois la boucle exécutée, on utilise la méthode close pour fermer le fichier. On obtient un fichier texte dans lequel sont écrits les deux vecteurs **x** et **y** en colonnes.

Inversement, on peut lire ce même fichier avec une méthode similaire

```
x=[]
y=[]
mon_fichi=open("fichier.txt", "r")
for line in mon_fichi:
       a=line.rstrip("\n").split(sep='\t')
       x.append(float(a[0]))
       y.append(float(a[1]))
mon_fichi.close()
```

Le code ci-dessus crée deux listes vides, ouvre le fichier. Il fait une boucle sur l'ensemble des lignes. Pour chaque ligne lue qui correspond une chaîne de caractères, on fait les opérations suivantes : on enlève le dernier caractère qui correspond à un caractère de fin de ligne avec la méthode **rstrip**, puis on transforme la chaîne en une liste de chaîne de caractères avec la méthode **split** en déclarant le séparateur comme une tabulation, puis on insère chaque valeur dans les listes **x** et **y**.

La méthode précédente peut être grandement simplifiée quand on sait que le fichier à lire est constitué de nombres. Avec **loadtxt** de la bibliothèque **numpy**, on peut lire le fichier et remplir une matrice $m \times n$ où m est le nombre de lignes et n le nombre de colonnes.

Le mini-code suivant donne la méthode pour lire le fichier précédent avec **numpy**.

```
import numpy as np
a=np.loadtxt("fichier.txt")
```

De même pour l'écriture, on a une méthode plus rapide avec **savetxt**.

```
import numpy as np
x=np.linspace(0.1,10,200)
y=np.log(x)
A=np.transpose(np.array([x,y]))
np.savetxt("fichier2.txt",A,delimiter='\t',newline='\n')
```

Les trois premières lignes de ce code créent les deux vecteurs lignes de 200 éléments. La ligne suivante crée une matrice 200×2 que l'on transpose pour avoir une matrice 200×2 et l'instruction **savetxt** crée

un fichier s'il n'existe pas (ou sinon il écrase le précédent). Chaque ligne écrite comprend deux éléments séparés par une tabulation et chaque fin de ligne par un retour à la ligne. Cette méthode se généralise aisément pour une nombre de colonnes supérieur à 2.

On voit donc qu'il est préférable d'utiliser une bibliothèque adaptée pour réaliser des tâches spécifiques plutôt que de chercher à utiliser les méthodes de base du langage Python. C'est la raison pour laquelle nous allons approfondir trois bibliothèques standard pour le calcul scientifique que sont **matplotlib**, **numpy** et **scipy**. Nous verrons plus loin d'autres modules au fur et à mesure de ce cours quand cela est nécessaire. Ainsi Python offre une grande diversité dans la construction d'un code : ses nombreux modules permettent à la fois de faire un code plus court et souvent plus efficace.

CHAPITRE 3

Quelques modules de Python : matplotlib, numpy, scipy, glob et re

3.1 Introduction

Nous avons vu, au chapitre précédent, les éléments de base du langage Python qui présentent déjà plusieurs avantages par rapport à d'autres langages usuels. Pour être plus efficace dans l'écriture et l'exécution d'un programme, des bibliothèques ont été développées (et beaucoup sont toujours en développement) pour traiter des domaines spécifiques. Dans le domaine scientifique, trois modules sont particulièrement importants : **matplotlib** permet de transformer des tableaux en graphiques souvent à l'intérieur du même fichier qui a traité le problème. La séquence usuelle avec les langages compilés passe par l'écriture d'un fichier de résultats de calcul, suivie de sa relecture par un logiciel graphique, appelé grapheur. Avec le module **matplotlib**, cela permet de créer des graphiques qui utilisent des résultats disponibles dans les variables que le programme Python a créées, sans passer par une écriture de fichier de données puis par une relecture par un grapheur. L'aspect tout-en-un est bien évidemment un gain de temps pour le développement d'un code.

Nous avons vu une partie des fonctionnalités du module **numpy**, car il est bien évidemment très adapté pour la création, la manipulation et l'exécution de procédures sur les tableaux. L'échange de données entre la mémoire et les unités de stockage est aussi facilitée avec **numpy** qui possède ces propres fonctions pour la lecture et l'écriture de fichiers. Nous allons poursuivre l'approfondissement de la connaissance de ce module, mais aussi de celle du module **scipy** qui contient un grand nombre de méthodes numériques, qui sont illustrées dans le second polycopié.

3.2 Matplotlib

Le module dispose de nombreuses fonctionnalités pour tracer des graphes bidimensionnels. A nouveau une grande partie de cette section constitue un rappel pour la plupart des lecteurs. Pour ceux dont la connaissance de matplotlib est limitée, il convient d'acquérir rapidement les commandes les plus essentielles.

3.2.1 Graphe unique

Il faut commencer par le traçage d'un simple graphe. Le mini-code suivant montre qu'à partir d'une fonction et en chargeant une partie du module **matplotlib**, **matplotlib.pyplot** renommé **plt**, l'instruction **plot** permet d'obtenir un graphe simplement. Pour obtenir l'affichage de la figure il est nécessaire d'utiliser l'instruction **show()**.

Le résultat est illustré sur la figure 3.1

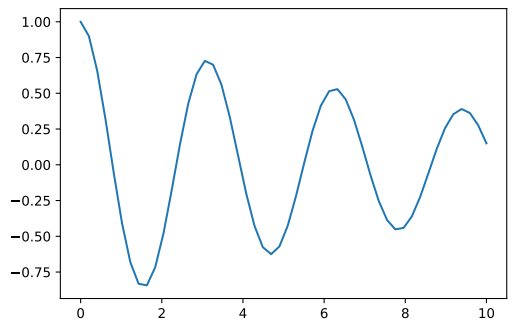


FIGURE 3.1 – Simple graphe

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Jun 16 09:31:44 2020

@author: viot
"""
import numpy as np
import matplotlib.pyplot as plt

def oscil_amorti(t,alpha=0.1,omega=2):
    """ exponentielle decroissante oscillante
        ↪ """
    f=np.exp(-alpha*t)*np.cos(omega*t)
    return(f)

x=np.linspace(0,10,50)
plt.plot(x,oscil_amorti(x))
plt.savefig("plot0.pdf")
plt.show()
```

Comme on peut le voir, le résultat est intéressant mais pas exceptionnel ! Il va falloir donc utiliser des méthodes supplémentaires pour obtenir un enrichissement de la figure précédente.

On peut ajouter des étiquettes sur les axes avec les instructions **xlabel** et **ylabel**. On peut aussi augmenter les polices de caractères ainsi que les étiquettes des axes. Voici la figure améliorée ainsi que le code associé.

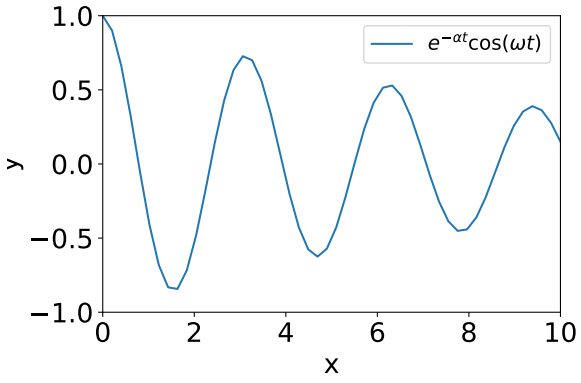


FIGURE 3.2 – Relooking de la première figure

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Jun 16 09:31:44 2020

@author: viot
"""
import numpy as np
import matplotlib.pyplot as plt
def oscil_amorti(t,alpha=0.1,omega=2):
    """ exponentielle decroissante oscillante
        ↪ """
    f=np.exp(-alpha*t)*np.cos(omega*t)
    return(f)
x=np.linspace(0,10,50)
plt.plot(x,oscil_amorti(x),label=r'$e^{-\alpha␣
    ↪ t}\cos(\omega␣t)$')
plt.xlabel('x',fontsize=20)
plt.ylabel('y',fontsize=20)
plt.legend(fontsize=15)
plt.xlim(0,10)
plt.ylim(-1,1)
plt.tight_layout()
plt.tick_params(labelsize=20)
plt.savefig("plot1.pdf")
```

Par rapport au code précédent, on a ajouté à la courbe une étiquette qui a besoin d'être placée dans l'ordre **plot**. La chaîne de caractères pour le label utilise le caractère **r** qui n'apparaît pas sur la figure

mais qui signifie que la chaîne de caractères sera considérée comme une chaîne Latex, ce qui explique les différents symboles $. Ceux-ci signifient que l'on va écrire une forme mathématique avec la syntaxe usuelle de Latex. (Je renvoie le lecteur à un tutoriel sur Latex s'il n'est pas très familier avec ce descripteur de langage). Les instructions **xlabel** et **ylabel** placent des étiquette sur les deux axes. **xlim** et **ylim** fixent les limites de la boite d'affichage. **tick_params** permet d'augmenter la police de caractères des chiffres affichés sur les axes. L'instruction **tight_layout** permet d'éviter le chevauchement de caractères. C'est une instruction qui permet d'obtenir un résultat graphique de meilleure qualité. On voit que, par défaut, Python calcule automatiquement certaines quantités et utilise des options. Il est toutefois nécessaire d'améliorer la présentation en changeant ces valeurs standards comme le fait ce script dans cette deuxième version.

Dans l'étape suivante, nous allons tracer deux courbes sur le même graphe.

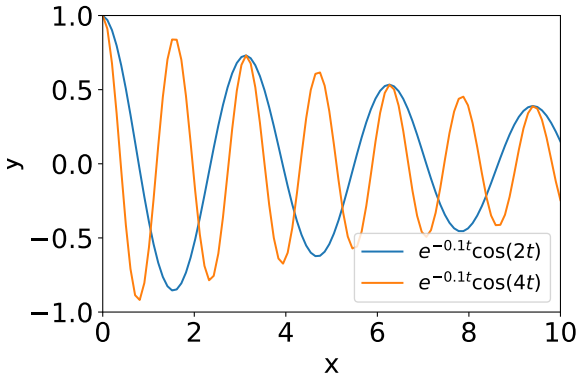


FIGURE 3.3 – Un graphe avec deux courbes

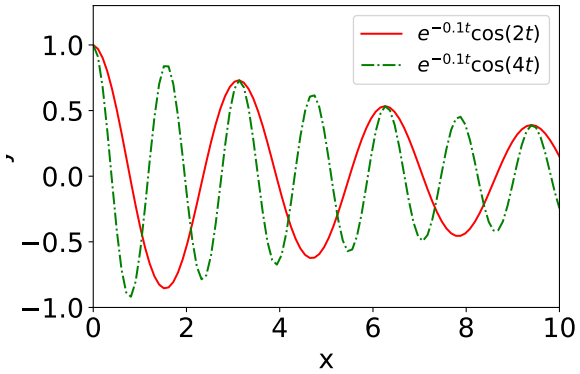
```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Jun 16 09:31:44 2020

@author: viot
"""
import numpy as np
import matplotlib.pyplot as plt
def oscil_amorti(t,alpha=0.1,omega=2):
    """ exponentielle decroissante oscillante
        ↪ """
    f=np.exp(-alpha*t)*np.cos(omega*t)
    return(f)
x=np.linspace(0,10,100)
plt.plot(x,oscil_amorti(x),label=r'$e^{-0.1␣t}\
    ↪ cos(2␣t)$')
plt.plot(x,oscil_amorti(x,0.1,4),label=r'$e
    ↪ ^{-0.1␣t}\cos(4␣t)$')
plt.xlabel('x',fontsize=20)
plt.ylabel('y',fontsize=20)
plt.legend(fontsize=15)
plt.xlim(0,10)
plt.ylim(-1,1)
plt.tight_layout()
plt.tick_params(labelsize=20)
plt.savefig("plot2.pdf")
plt.show()
```

La figure 3.3 réalise le tracé de deux courbes avec les options graphiques qui sont affichées sur le listing situé à droite de la figure.

A nouveau, il est nécessaire d'arranger un peu le code graphique pour avoir une figure plus présentable.

La figure 3.4 réalise le tracé de deux courbes, avec les changements des options graphiques qui sont affichées sur le listing situé à gauche de la figure. On note que l'on peut changer la couleur par défaut des courbes avec l'option **color** et que le type de tracé est contrôlé par la séquence **'-'** et **'-.'** qui correspondent respectivement à une courbe pleine et tiretée. Il existe bien sur d'autres options et je revoie le lecteur au manuel de **matplotlib** pour celles-ci. En changeant les limites de la figure, on évite aussi le chevauchement des courbes avec les légendes.




```
plt.plot(x,oscil_amorti(x),'-',color='red',
    ↪ label=r'$e^{-0.1␣t}\cos(2␣t)$')
plt.plot(x,oscil_amorti(x,0.1,4),'-.g',label=r'
    ↪ $e^{-0.1␣t}\cos(4␣t)$')
plt.xlim(0,10)
plt.ylim(-1,1.3)
```

FIGURE 3.4 – Relooking de la figure précédente

Pour finir cette introduction sur les graphes simples, il est bien entendu possible de tracer les courbes en utilisant une ou plusieurs échelles logarithmiques.

Le mini-code suivant illustre cette possibilité.

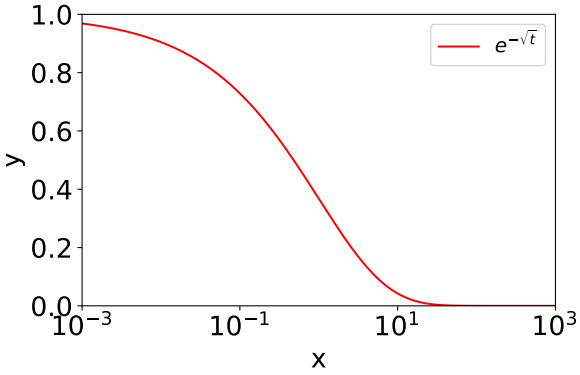


FIGURE 3.5 – Tracé log-linéaire d'une exponentielle étirée

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Jun 16 09:31:44 2020

@author: viot
"""
import numpy as np
import matplotlib.pyplot as plt
def exp_etire(t,alpha=0.5):
    """ exponentielle etire"""
    f=np.exp(-t**alpha)
    return(f)

x=np.geomspace(0.001,1000,10000)
plt.xlabel('x',fontsize=20)
plt.ylabel('y',fontsize=20)
plt.xscale("log")
plt.plot(x,exp_etire(x),'-',color='red',label=r
    ↪ '$e^{-␣\sqrt{t}}$')
plt.legend(fontsize=15)
plt.xlim(0.001,1000)
plt.ylim(0,1)
plt.tight_layout()
plt.tick_params(labelsize=20)
plt.savefig("plot3b.pdf")
plt.show()
```

Sur le code partiel situé à droite de la Figure 3.5, on peut noter que les points sur l'axe x ont été générés par **geomspace** au lieu de **linspace**, permettant ainsi une progression géométrique dans l'espacement des différents points. De plus **xscale** avec l'option **log** permet d'avoir une échelle logarithmique sur l'axe 0x. Une instruction similaire existe pour l'axe 0y.

3.2.2 Graphe multiple

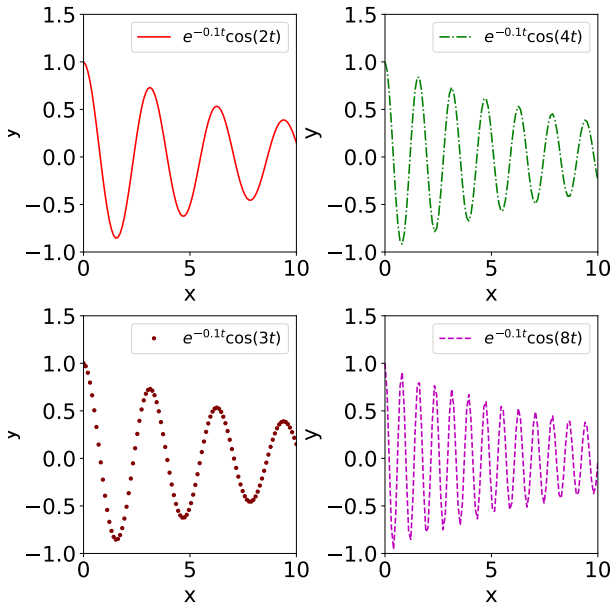


FIGURE 3.6 – Graphe multiple

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Jun 16 09:31:44 2020

@author: viot
"""
import numpy as np
import matplotlib.pyplot as plt
def oscil_amorti(t,alpha=0.1,omega=2):
    """ exponentielle decroissante
        ↪ oscillante"""
    f=np.exp(-alpha*t)*np.cos(omega*t)
    return(f)
x=np.linspace(0,10,100)
#plt.figure(figsize=(8,8))
fig,ax=plt.subplots(2,2,figsize=(8,8))
ax[0,0].plot(x,oscil_amorti(x),'-',color='
    ↪ red',label=r'$e^{-0.1␣t}\cos(2␣t)$')
ax[0,1].plot(x,oscil_amorti(x,0.1,4),'-.g',
    ↪ label=r'$e^{-0.1␣t}\cos(4␣t)$')
ax[1,0].plot(x,oscil_amorti(x),'.',color='
    ↪ maroon',label=r'$e^{-0.1␣t}\cos(3␣t)$
    ↪ ')
ax[1,1].plot(x,oscil_amorti(x,0.1,8),'--m',
    ↪ label=r'$e^{-0.1␣t}\cos(8␣t)$')
for i in range(2):
    for j in range(2):
        ax[i,j].set_xlabel('x',fontsize=15)
        ax[i,j].set_ylabel('y',fontsize=15)
        ax[i,j].set_xlim(0,10)
        ax[i,j].set_ylim(-1,1.2)
        ax[i,j].legend(fontsize=10)
        ax[i,j].tick_params(labelsize=15)


plt.tight_layout()
plt.savefig("plot4.pdf")
plt.show()
```

Dans la suite des possibilités offertes avec **matplotlib**, on peut réaliser une figure comprenant plusieurs graphes. La figure 3.6 comprend 4 sous-figures. Nous allons expliquer les différentes lignes ci-dessous, mais il faut souligner l'importance de l'instruction **tight_layout()** pour organiser correctement l'affichage.

Il existe deux instructions voisines pour créer des graphes multiples, mais on recommande d'utiliser **subplots** au lieu de **subplot**.

L'instruction subplots permet de créer des sous graphes dont la structure est donnée par les deux chiffres qui sont les premiers arguments de l'instruction, l'option **figsize** permet de changer les tailles verticale et horizontale. La valeur de retour ax est un tableau à une ou deux dimensions correspondant aux premiers paramètres de l'instruction **subplots**.

Pour la figure 3.6, en mettant la même valeur selon les deux directions, on obtient une figure carrée. Cette instruction peut bien évidemment s'appliquer dans le cas des graphes simples. Elle est plus cruciale dans le cas d'un graphe multiple pour éviter des déformations importantes non souhaitées.

Les méthodes des sous-figures sont un peu compliquées, car elles sont parfois identiques (**legend**, **tick_params**) à celle d'une figure simple, mais parfois différentes (**set_xlabel**, **set_ylabel**, **set_xlim**, **set_ylim**). Comme il existe un grand nombre d'options pour tracer une courbe, il est recommandé de vérifier le nom des nombreuses options disponibles avec une sous-figure avant de les utiliser.

3.2.3 Figures variées

Matplotlib n'est pas limité aux tracés de courbes simples. Il n'est pas question d'être exhaustif pour ce cours sur toutes les possibilités, mais d'élargir les exemples précédents en considérant quelques figures que l'on a l'habitude de tracer pour des calculs scientifiques.

Histogrammes

Il est fréquent d'avoir à tracer un histogramme de données. L'exemple suivant illustre la création d'un vecteur comprenant 1000 nombres aléatoires crées à partir d'une distribution gaussienne et de comparer graphiquement ce résultat avec la courbe d'une distribution gaussienne.

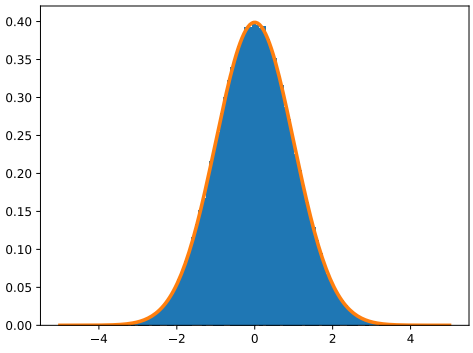


FIGURE 3.7 – Histogramme de variables aléatoires gaussiennes

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Jun 16 09:31:44 2020

@author: viot
"""
import numpy as np
import matplotlib.pyplot as plt
import time
def gauss(x,mean=0.,sigma=1.):
    return(np.exp(-(x-mean)*(x-mean)/(2*sigma*
        ↪ sigma))/sigma/np.sqrt(2*np.pi))

#np.random.seed(42)
x1=np.linspace(-5,5,200)
debut=time.time()
x=np.random.randn(500000)
plt.hist(x,bins=100,density=True)
plt.plot(x1,gauss(x1),lw=3)
fin=time.time()
print("temps ecoule en s",fin-debut)
plt.savefig("gauss.pdf")
plt.show()
```

Le code comprend tout d'abord une déclaration de la fonction gaussienne. On crée ensuite un vecteur de 200 éléments n entre −4 et 4. On crée un vecteur de 5000 valeurs aléatoires gaussiennes en utilisant une méthode de **numpy** dans la sous-bibliothèque **random** qui s'appelle **randn**. L'instruction **hist** est utilisée avec deux options : **bins** est fixé à 50, la valeur par défaut étant égale à 10, cela donne un histogramme un peu trop discrétisé, l'option **density=True** permet d'avoir un histogramme normalisé, ce qui est très utile pour une comparaison avec la valeur exacte pour un nombre infini de points.

Tracé de densité

Cela correspond à la situation où l'on a une fonction à deux dimensions dont on souhaite transformer les valeurs en un code couleur selon une échelle donnée. L'exemple typique pour illustrer ce concept est la figure obtenue pour la diffraction de la lumière. L'intensité de la lumière obtenue par diffraction à

travers un trou circulaire de rayon R est donnée par la formule

$$I = 4I_0\left(\frac{J_1(\eta)}{\eta}\right)^2 \tag{3.1}$$

avec

$$\eta = \frac{2\pi Rr}{\lambda d} \tag{3.2}$$

où r est la distance d'un point par rapport au centre du plan, λ la longueur d'onde, d la distance entre le trou et l'écran, J_1 la fonction de Bessel de première espèce et I_0 est l'intensité du faisceau incident.

La fonction J_1 fait partie de la bibliothèque de **scipy.special** ce qui explique la raison de l'importation de cette bibliothèque dans le listing de droite correspondant à la figure. Comme cela est illustré dans le second polycopié, il y a un grand nombre de fonctions spéciales avec scipy comme l'illustre les figures de ce chapitre dédié à ce sujet.

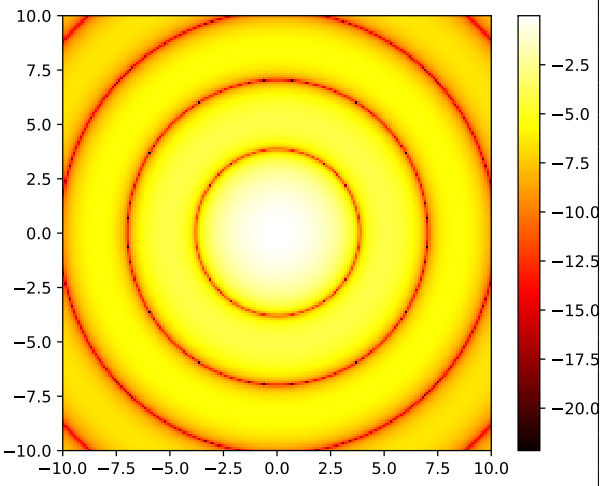


FIGURE 3.8 – Intensité lumineuse pour la diffraction lumineuse à travers un trou circulaire

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Jun 16 09:31:44 2020

@author: viot
"""
import numpy as np
from scipy.special import jv
import matplotlib.pyplot as plt
def diffrac(x,y):
    r2=x**2+y**2
    r=np.sqrt(r2)
    return(4*jv(1.0,r)**2/r2)

x=np.linspace(-10,10,200)
y=np.linspace(-10,10,200)
X, Y = np.meshgrid(x, y)
fig=plt.figure(figsize=(6,5))
Z=diffrac(X,Y)
plt.pcolor(X, Y, np.log(Z),cmap=plt.cm.hot)
plt.colorbar()
plt.savefig("diffrac.pdf")
plt.show()
```

Il existe de nombreuses possibilités supplémentaires pour générer des figures à deux dimensions et nous renvoyons le lecteur à plusieurs sites, donnés à la fin de ce chapitre, pour approfondir son usage. Nous allons maintenant considérer les deux autres modules spécifiques au calcul scientifique que sont **scipy** et **numpy** (dont les éléments de base ont été déjà introduits au chapitre précédent).

3.3 Numpy

Nous avons vu au chapitre précédent les premiers éléments de **numpy**. Nous revenons dans cette section sur des compléments utiles pour la manipulation des tableaux. Pour les tableaux à une dimension, nous avons vu l'instruction **linspace** pour créer un tableau à une dimension. Il existe aussi une deuxième instruction **arange** avec trois arguments : le premier se réfère à la valeur de départ, le second à la valeur finale (mais qui est exclue du vecteur) et la dernière est le pas entre les valeurs successives. Avec cette instruction, la taille du tableau est déterminée dès sa création et surtout la valeur finale ne fait pas partie du tableau, tandis qu'avec **linspace** la taille du tableau est donnée par le troisième argument

et le pas est calculé en fonction de la taille. Le choix de l'instruction est dicté par le problème qui est considéré. Le mini-code suivant donne le même résultat

```
x_ls=np.linspace(0,9.9,100)
x_ar=np.arange(0.,10,0.1)
print(x_ls-x_ar)
```

Pour un tableau **numpy** il est possible d'avoir des informations sur sa dimension, sa forme et sa taille. Rappelons que la liste des méthodes qui sont disponibles pour un objet est donnée par l'instruction **dir**. Dans le cas d'un tableau **numpy**, on obtient une très grande liste. Je rappelle que pour avoir la liste complète sous la forme texte, il suffit de taper dans la fenêtre iPython (en bas à droite de Spyder) l'instruction **dir(x_ls)**. On obtient alors plus d'une centaine de méthodes ! On va se contenter d'en expliquer quelques unes, mais bien évidemment vous pouvez aller satisfaire votre curiosité en allant explorer plusieurs méthodes en dehors de celles que nous allons expliquer. Les trois lignes suivantes vous permettent d'obtenir la dimension du tableau , la forme et le type des éléments du tableau.

```
print(x_ls.ndim)
print(x_ls.shape)
print(x_ls.dtype)
```

Pour les tableaux à 2 dimensions (et plus), on a vu plusieurs manières d'initialiser les tableaux. Comme souvent avec Python, il y plusieurs instructions pour obtenir le même résultat. Le mini-code suivant illustre que la transformation d'un tableau de 1 à 2 dimensions peut se faire avec la fonction **reshape** ou avec la méthode **reshape**. (voir la différence dans le listing suivant.)

```
A=np.arange(1,101,1)
B=np.reshape(A,(10,10))
C=A.reshape((10,10))
print(A.ndim)
print(B.ndim)
print(C.ndim)
print(A.shape)
print(B.shape)
print(C.shape)
print(C-B)
```

3.3.1 Les Matrices

Avec **numpy**, il y a une deuxième possibilité pour créer et gérer les matrices avec le type **matrix**. La création historique du concept reposait sur une similitude de comportement avec les matrices créées sous Matlab. Aujourd'hui, on peut utiliser généralement les deux types avec les instructions de **scipy**. Le concept de matrice est toutefois limité à des objets à deux indices contrairement au tableaux **ndarray** de numpy.

```
A=np.array([[1,2],[3,4]])
B=np.matrix([[1,2],[3,4]])
print(A-B)
print(B.shape)
print(A.shape)
```

L'utilisation des matrices sous **numpy** est plutôt découragée et deviendra probablement obsolète dans une version future de Python. C'est un peu un problème récurrent associé aux langages en forte évolution avec des changements assez fréquents de nombreux concepts.

Les opérations d'addition et de soustraction entre tableaux de même forme et même dimension fonctionnent normalement (comme les opérations sur les matrices). La multiplication d'un tableau par un

scalaire est aussi possible. Par contre le produit de deux matrices **A*B** ne donnent pas la multiplication de deux matrices mais le produit terme à terme entre les éléments de la matrice. Pour obtenir le produit de deux matrices, il faut utiliser **dot(A,B)**. Compte tenu de l'importance de l'algèbre linéaire et surtout des méthodes numériques, nous reviendrons sur ces questions à la fois avec le module **scipy** qui comprend la plupart des méthodes standards de calcul et le second polycopié sur les méthodes numériques illustrent l'utilisation de Python à de nombreuses reprises.

3.4 Scipy

Le module **Scipy** permet d'utiliser la plus grande partie des méthodes numériques dont on peut avoir besoin pour résoudre numériquement des équations mathématiques. Cela comprend, par exemple, le calcul d'intégrales numériques, la résolution d'équations linéaires et non linéaires, celle des équations différentielles ordinaires,.. Il est bien entendu hors de question d'être exhaustif sur l'ensemble des possibilités offertes par **scipy** pour la résolution numérique. Le but recherché est, à partir de quelques problèmes numériques, de réussir à mettre en place l'utilisation de méthodes développées dans ces bibliothèques. Il s'agit surtout de ne pas réinventer la roue quand on doit faire face à un problème numérique très standard. Python, de même que C++, dispose aujourd'hui d'un très large éventail de bibliothèques qu'il convient d'utiliser efficacement. Cette tâche nécessite au préalable de maîtriser les concepts de base qui sont développés à la fois dans le langage lui-même et avec les module **numpy**. Par la suite, les méthodes présentes dans **scipy** permettent de résoudre numériquement les problèmes, avec un contrôle de la précision des résultats. Nous verrons bien sûr qu'il y a des limites avec Python quand il s'agit de traiter des problèmes numériques lourds et dans ce cas, le recours à un langage compilé comme C++ sera nécessaire, mais d'une façon naturelle, car les bibliothèques et la majorité des concepts que l'on aura vus en Python se retrouve aujourd'hui en C++.

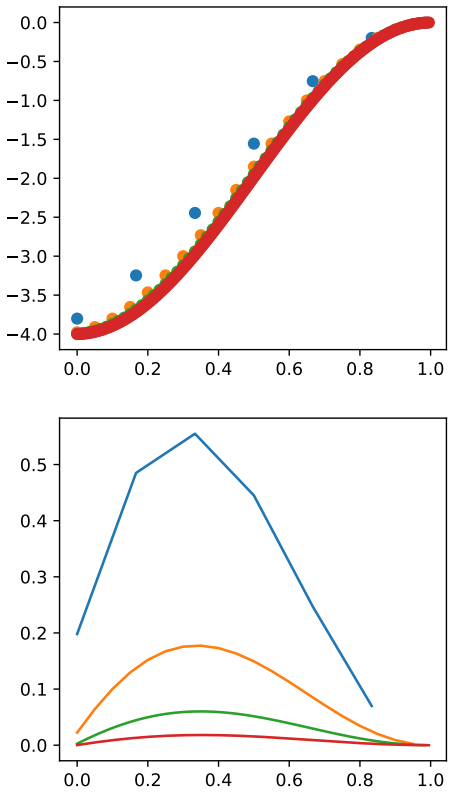
Algèbre linéaire

Les opérations que l'on a effectuées sur des matrices comprennent bien sur les opérations de base, addition, soustraction, multiplication, mais aussi transposition, inversion, diagonalisation avec la recherche de vecteurs propres,...

Nous allons donc faire une courte revue sur ces possibilités. Illustrons tout d'abord la construction de matrice : on considère la matrice 6×6 suivante

$$A = \begin{pmatrix} -2 & 1 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 1 & -2 \end{pmatrix}$$

Il est simple de généraliser a une matrice de taille $n \times n$ avec la même structure. Analytiquement, on peut montrer les valeurs propres de cette matrice à la limite des très grandes tailles sont données par la fonction $-2 - 2\cos(2\pi x)$ où x est un "indice" réel allant de 0 à 1. Avec le code Python suivant, on va construire ce type de matrice pour quatre valeurs de n




```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Jun 16 09:31:44 2020

@author: viot
"""
import numpy as np
from scipy import linalg
import matplotlib.pyplot as plt
plt.clf()
plt.figure(figsize=(4,8))

for n in (6,20,60,200):
    x=np.arange(0,n)/n
    A=-2*np.eye(n)+np.eye(n,k=1)+np.eye(n,k
        ↪ =-1)
    print(A)
    Vp=linalg.eigvals(A)
    Vpr=np.real(Vp)
    Vpr.sort()
    plt.subplot(2,1,1)
    plt.plot(x,Vpr,'o')
    plt.subplot(2,1,2)
    plt.plot(x,(Vpr+2+2*np.cos(np.pi*x)),'-'
        ↪ )
plt.savefig('vp.pdf')
plt.show()
```

FIGURE 3.9 – Valeurs propres en fonction de i/n pour différentes valeurs n

La figure 3.9 montre que les différentes valeurs propres obtenues pour les quatre matrices pour n parcourant $6, 20, 60, 200$ ainsi que la valeur exacte pour la limite d'une matrice de taille infinie.

Le code Python qui suit s'explique de la manière suivante. Hormis les deux modules usuels de **numpy** et **matplotlib**, on charge le sous-module de **scipy** qui correspond à l'algèbre linéaire. On définit une taille de figure qui évitera une forme par défaut, peu compatible avec la présence de 2 sous-figures. On définit alors une boucle qui va utiliser les 4 entiers des matrices dont on souhaite calculer les valeurs propres On définit un indice qui va parcourir l'ensemble des valeurs propres de chaque matrice. Le facteur $1/n$ permet d'avoir un indice qui est toujours compris entre 0 et 1. On utilise l'instruction **eye** qui permet de créer une matrice de dimension $n \times n$ avec une diagonale principale remplie de 1 et avec les autres éléments mis à 0. Quand on ajoute l'option $k = 1$ les valeurs non nulles sont placées sur la diagonale décalée de 1 unité vers le haut et quand $k = -1$ le décalage est vers le bas. On a donc une matrice tridiagonale A. On calcule les valeurs propres avec l'instruction **eigenvals**. Les valeurs propres sont en général complexes, mais on sait qu'elles sont réelles pour une matrice symétrique. On peut vérifier que les parties imaginaires sont généralement nulles. La fonction **real** de **numpy** convertit les complexes en nombres réels. On trie ensuite le tableau des valeurs propres et on trace la courbe dans le graphe de gauche (Pour la matrice la plus grande, le nombre de valeurs propres est égal à 200, et la courbe apparaît quasiment continue). Les différentes courbes semblent converger vers une courbe limite. La figure de droite affiche la différence entre les valeurs propres de chaque matrice et la courbe limite.

Noter que la compacité de ce code, qui, en quelques lignes, permet à la fois de résoudre ce problème

non trivial et d'analyser ce résultat en détails. Cette étude peut être poursuivi en montrant que que si on multiplie les valeurs de la courbe de droite par n on obtient une courbe universelle, que je laisse le lecteur déterminer.

Dans l'exemple suivant, on définit les trois matrices de Pauli que l'on rencontre en mécanique quantique et l'on va vérifier les propriétés les plus standards pour utiliser les opérations de multiplication matricielle. Les trois matrices de Pauli sont

$$\sigma_1 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \sigma_2 = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \sigma_3 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix},$$

On a les propriétés suivantes :

- $\sigma_i^2 = I_2$ pour $i = 1, 2, 3$ et I_2 désigne la matrice identité à deux dimensions.
- $\sigma_1\sigma_2 = i\sigma_3$ ainsi que les deux autres relations obtenues par permutation circulaire des indices.
- $[\sigma_1, \sigma_2] = 2i\sigma_3$ où les crochets désignent le commutateur des deux opérateurs.

Le code suivant en Python réalise ces vérifications

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Jun 16 09:31:44 2020

@author: viot
"""
import numpy as np
sigma1=np.array([[0,1],[1,0]])
sigma2=np.array([[0,-1j],[1j,0]])
sigma3=np.array([[1,0],[0,-1]])
print(np.dot(sigma1,sigma1)-np.eye(2))
print(np.dot(sigma2,sigma2)-np.eye(2))
print(np.dot(sigma3,sigma3)-np.eye(2))
print('operateur␣@',sigma1@sigma1-np.eye(2))
print(sigma1@sigma2-1j*sigma3)
print(sigma1@sigma2-sigma2@sigma1-2j*sigma3)
```

Ce code montre que l'on peut remplacer l'instruction **dot** de **numpy** par l'opérateur @ plus élégant. Il faut juste être conscient que cette opérateur réalise la même opérations que **dot** pour des tableaux à deux dimensions. Dans le cas de tableaux à trois dimensions et au delà, ce n'est plus équivalent. Si on garde cette différence en tête, l'écriture est, en effet, plus simple.

Pour ceux qui sont intéressés par les matrices de Pauli, il faut savoir qu'elles sont prédéfinies dans le module **simpy** (qui a pour fonction d'implémenter le calcul symbolique) et aussi dans le logiciel **qutip**, écrit en Python, qui permet de faire de nombreux calculs en mécanique quantique

3.5 Quelques commandes sous IPython

Spyder propose une fenêtre **IPython** qui permet d'exécuter à la fois le script en cours de rédaction mais aussi des commandes système que l'on retrouve sur les différent systèmes d'exploitation. On peut s'interroger sur l'utilité d'avoir des commandes en doublon. La réponse vient du fait que les commandes systèmes ne sont pas les mêmes sous **Unix** ou sous **Windows**. Comme l'interpréteur installé dépend du système d'exploitation, on peut écrire des scripts sous **Python**, qui peuvent ensuite être utilisés sur un autre système. A nouveau, cela réduit le temps de développement d'un code de manière importante, car on élargit l'utilisation du code écrit pour différentes plateformes.

Ces commandes sont les mêmes sous **Python** que celles d'un système sous **Unix**. Cela signifie que l'on peut les utiliser avec **Spyder** sur les différentes plateformes avec la même syntaxe. Pour ceux qui

connaissent les commandes **Unix**, il n'y a pas de changement majeur. Pour ceux qui sont habitués au Winshell, la syntaxe est différente. Avec **MacOx** dont la majeure partie est un dérivé **Unix**, le changement sera très faible. Une partie des options de ces commandes ne sont disponibles que sous Unix. En particulier sous Windows, une bonne partie n'existe pas.

Voici quelques commandes de base

- **ls** donne la liste des noms du répertoire courant.
- **ls -l** donne la liste des noms avec des détails. Pour les utilisateurs de Windows, la commande est **dir**, mais elle est réservée par Python pour donner la liste des fonctions disponibles.
- **ls -lt** donne la liste des noms avec des détails, triée du plus récent au plus ancien.
- **ls -ltr** donne la liste des noms avec des détails, triée du plus ancien au plus récent.
- **pwd** donne le répertoire courant sur le lequel **Spyder3** (et la console **IPython**) travaillent.
- la commande **cd** suivi d'un argument donnant, soit le chemin relatif par rapport au répertoire courant, soit le chemin absolu permet de changer le répertoire courant. A nouveau, attention pour les utilisateurs de **Windows**, le séparateur de répertoires est le caractère **/** et non pas \.
- Quand vous chargez dans **Spyder** un fichier **Python** qui se trouve dans un répertoire qui n'est pas dans le précédent, la console IPython se met à jour automatiquement dans ce nouveau répertoire. On peut le vérifier avec la commande **pwd**.
- la commande **cp** permet de copier un fichier sur un second fichier. Elle permet aussi de copier plusieurs fichiers sur un répertoire qui correspond au dernier argument donné sur la liste qui suit la commande **cp**.
- la commande **rm** permet de détruire le ou les fichiers qui sont donnés comme argument de la commande.
- Pour les deux commandes précédentes, l'utilisation du caractère**?** permet de sélectionner l'ensemble des fichiers où ce caractère est quelconque. Par exemple **sin?.txt** correspond aux 9 fichiers **sin1.txt**, **sin2.txt**, **sin3.txt**,.., **sin9.txt**.
- L'utilisation du caractère * permet de sélectionner l'ensemble des fichiers où une chaîne de caractère quelconque peut remplacer ce caractère. Par exemple, **sin*.txt** sélectionner tous les fichiers commençant par **sin** et finissant par **.txt**. Cela inclut bien entendu les fichiers précédents, mais aussi des fichiers comme **sin10.txt** s'il existe.
- De nombreuses autres commandes sont disponibles sous **IPython** : **cat**, **more**, **less**,... Contrairement à leur utilisation dans une fenêtre terminal, les trois commandes affichent la totalité du fichier dans la fenêtre IPython.

Une commande très utile sous **Unix** est **grep** qui permet la recherche d'expressions régulières dans un fichier ou plusieurs fichiers. Cette commande n'est pas disponible à partir d'une fenêtre IPython mais il existe un module Python dont le but d'exécuter des recherches et des substitutions sur des chaînes de caractères.

3.6 Module Re

Les expressions régulières sont divisées en trois groupes : (i) les expressions régulières de base, (ii) les expressions régulières étendues et finalement (iii) les expressions régulières avancées. Une expression régulière est aussi appelé motif de recherche.

Afin de rechercher un ensemble de chaînes de caractères, une liste dite de métacaractères permet de réaliser des fonctions que nous allons détailler ci-dessous. Tout d'abord, la liste des métacaractères pour les expressions régulières est

```
. ^ $ * + ? { } [ ] \ | ( )
```

- Le point **.** désigne n'importe quel caractère hormis celui de fin de ligne. Ainsi l'expression régulière **p.r.** permet de considérer des mots comme **part**, **pore**, **paru**, **pire**,...
- La paire de crochets entourant une chaîne de caractères correspond à une expression régulière où il apparaît un au moins des caractères contenus. Par exemple [aeiou] représente une voyelle hormis le y. Si on ajoute un tiret entre deux caractères dans une paire de crochets cela correspond à un intervalle. Par exemple [0-9] désigne n'importe quel chiffre, [a-zA-Z] désigne tous les caractères minuscules ou majuscules de l'alphabet. Finalement, si l'on place l'accent ^ en première position de la liste, cela constitue une expression régulière où aucun des caractères de liste n'apparaît

Afin d'illustrer les concepts les plus simples, nous allons travailler sur un fichier texte qui est une fable de La Fontaine bien connue.

Dans le premier mini-code, on cherche toutes les fois le mot Corbeau apparaît

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Jul 15 17:48:04 2020

@author: viot
"""

import re
fi=open("lafontaine.txt")
pattern=re.compile("[cC]orbeau")

for line in fi:
    result=pattern.findall(line)
    if len(result)>0:
        print(result)
```

On définit le groupe de mots à rechercher. La présence des crochets permet de dire que l'on cherche le mot corbeau avec et sans majuscule. La boucle sur le fichier lit celui-ci ligne à ligne et on cherche dans chaque ligne toutes les occurrences qui apparaissent. On affiche le résultat si celui-ci est non vide. Tous les adeptes de La Fontaine savaient que le mot Corbeau apparaît quatre fois. Vous pouvez chercher le mot renard et déterminer de manière similaire. Le rapport des occurrences augure peut-être de la fin de la fable.

D'autres options plus détaillées seront vues lors de travaux pratiques.

3.7 Module glob

Dans une simulation numérique, il est fréquent de stocker les résultats dans les fichiers dont les noms sont associés aux grandeurs enregistrées avec une indexation donnée des valeurs numériques. Pour tracer des courbes collectant les données dans des fichiers différents, le module **glob** permet de choisir simplement les fichiers à utiliser. Le mini-code suivant crée 9 fichiers texte où il enregistre les fonctions $\sin(ix)$ pour i allant de 1 à 9.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Jul 13 18:34:56 2020

@author: viot
"""
```

```
import numpy as np
x=np.linspace(0,10,200)
for i in np.arange(1,10,1):
    y=i*np.sin(x)
    X=np.concatenate((x,y)).reshape(2,200).transpose()
    np.savetxt("sin"+str(i)+".txt",X)

#for i in range(1,10,1):
# np.savetxt("sin"str(i)+".txt"x,sin(x))
```

Pour sauver, dans chaque fichier, deux colonnes correspondant, pour la première, aux valeurs des abscisses, et, pour la seconde, aux valeurs de la fonction, on crée un vecteur résultant de la fusion des deux vecteurs que l'on transforme en une matrice de 2 lignes et 200 colonnes pour finalement prendre la transposée de cette matrice qui correspond aux données que l'on souhaite enregistrer.

Dans le mini-code suivant, on va lire tous les fichiers texte et les tracer sur le même graphe.

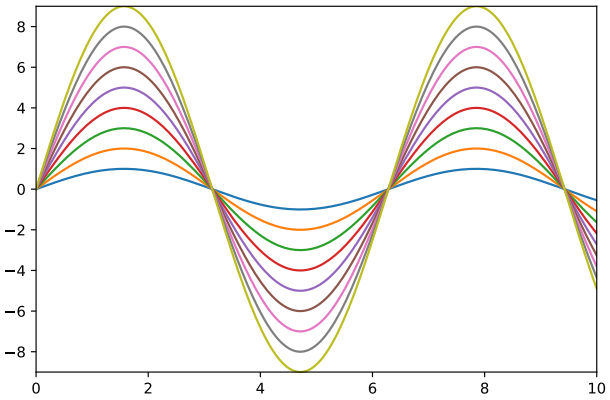


FIGURE 3.10 – Tracé de 9 courbes

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Jul 13 18:34:56 2020

@author: viot
"""

import numpy as np
import matplotlib.pyplot as plt
import glob
fi=sorted(glob.glob("sin*.txt"))
for ifi in fi:
    X=np.loadtxt(ifi)
    plt.plot(X[:,0],X[:,1])
plt.xlim(0,10)
plt.ylim(-9,9)
plt.tight_layout()
plt.savefig("sinx.pdf")


#for i in range(1,10,1):
# np.savetxt("sin"str(i)+".txt"x,sin(x))
```

La quatrième ligne du code utilise **glob.glob** pour récupérer la liste de tous les fichiers **sin*.txt** dans le répertoire courant. La fonction **sorted** permet de trier par ordre croissant les fichiers sélectionnés.

CHAPITRE 4

Le langage C++ : programmation procédurale

4.1 Introduction

Ces notes sont une introduction sur le langage C++ et toujours orientées vers le calcul scientifique. Historiquement développé pour être un langage qui soit un sur-ensemble du langage C, le C++ a acquis son autonomie et a introduit de nombreux concepts dont la notion d'objet est la plus connue. Son interfaçage avec le langage C permet d'utiliser la quasi totalité des bibliothèques disponibles en C, quand celles-ci n'ont pas été réécrites directement dans un C++ natif. Ce langage est en constante évolution depuis la dernière décennie. Les versions s'appellent d'ailleurs C++11, C++14, C++17, C++20 et C++23. On peut bien évidemment faire un parallèle avec l'évolution continue du Python. L'abstraction grandissante du langage procure l'avantage de pouvoir réaliser des programmes très sophistiqués et très sécurisés. Le développement de la Standard Template Library (STL) ainsi que la bibliothèque Boost conduit parfois à des problèmes d'incompatibilité avec les versions précédentes. Toutefois, ces deux bibliothèques ne sont aujourd'hui constituées que de fichiers d'en-tête ce qui rend la portabilité entre les différentes plateformes nettement plus simple.

La programmation procédurale repose sur le concept simple suivant : un programme est un outil qui consiste, à partir de données insérées au début du programme, à appliquer une série de transformations que l'on peut exprimer à partir de fonctions définies dans le code. Il se trouve que les programmes scientifiques peuvent généralement s'exprimer à travers ce concept initial. En fait, tous les premiers langages ont été développés sur ce modèle. On peut donc s'interroger pour savoir s'il est souvent judicieux, soit de faire évoluer le langage, soit plus radicalement d'en changer. La première raison est liée à l'accroissement considérable des possibilités offertes par les ordinateurs. Cela comprend la rapidité d'exécution, et leur architecture interne qui évolue très rapidement. (On peut aujourd'hui utiliser le processeur graphique en même temps que le processeur pour réaliser des calculs scientifiques.). La deuxième raison est associée au développement de code collaboratif dans lequel le C++ est en mesure de proposer des règles très adaptées à ce type de développement.

4.2 Les déclarations de variables

Historiquement, le langage C++ est un langage assez fortement typé, c'est-à-dire qu'il possède un assez grand nombre de types de variables prédéfinies et que les variables doivent être initialement associées à un type donné. L'évolution de ce langage montre la contrainte se relâche avec le temps.

4.2.1 Types prédéfinis

Une variable est associée à un ensemble d'octets de la mémoire. Selon le type de la variable, le nombre d'octets réservé est différent. Le tableau ci-dessous donne pour différents types le nombre d'octets réservé ainsi que l'intervalle borné inférieurement et supérieurement où l'on peut représenter ces nombres.

En C++ il n'y a pas de type **complex**, mais une classe **complex** sur laquelle nous reviendrons ultérieurement. Le tableau n'est pas exhaustif, mais généralement suffisant pour les problèmes que l'on a traiter. Deux lignes sont en italique, car elles correspondent à des types qu'il n'est pas nécessaire d'utiliser en général. Plus spécifiquement le type float était le type standard des processeurs 32 bits et permettait d'ef-

fectuer les calculs plus rapidement. Le standard étant depuis plus d'une décennie avec des processeurs 64 bits, cela est inutile de travailler avec des float qui engendre des problèmes d'imprécision récurrent. Nous reviendrons avec quelques exemples sur ce problème, plus loin dans ce cours.

Type	Taille	borne inférieure	borne supérieure
short int	*entier de 2 octets*	−32768	+32767
int	entier de 4 octets	−2147483648	2147483647
long	entier de 8 octets	−9223372036854775808	9223372036854775807
float	*réel de 4 octets*	$-3.4.10^{38}$	$3.4.10^{38}$
double	réel de 8 octets	$-1.7.10^{308}$	$1.7.10^{308}$
char	caractère de 1 octet	−128	127
bool	booléen	0(false)	1 (true)
unsigned int	entier de 4 octets	0	4294967295

4.3 Les instructions itératives

Pour les boucles, on dispose des mêmes fonctionnalités que celle du langage C. Il est donc très fortement recommandé de déclarer l'indice dans l'instruction de la boucle. Cela revient à limiter la portée de l'indice strictement à l'intérieur de la boucle Nous reviendrons plus amplement sur cette notion de portée de variable car de manière générale pour éviter des erreurs de programmation, maîtriser l'accès aux variables est un pilier de la programmation moderne. Pour un nombre d'itérations connu à l'avance, on peut écrire une itération avec la boucle **for**

```
for (unsigned int i=m;i<n;i+=step){
...
}
```

où **s**tep correspond au pas d'incrémentation. Si n est strictement inférieur à m, aucune instruction de la boucle ne sera exécutée. Pour une valeur de **s**tep égale à 1, on peut écrire plus simplement

```
for (unsigned int i=m;i<n;i++){
...
}
```

On a aussi besoin aussi de faire des instructions de manière répétée, mais dont le nombre n'est pas connu à l'avance, mais qui est fonction d'une condition qui est initialement vraie mais devra devenir fausse en un nombre fini d'itérations. On réalise cela avec **while** et **do-while** avec la syntaxe suivante :

```
while(condition) {
...
}
```

les instructions de la boucle ne sont exécutées que si la condition est vérifiée. Il faut donc s'assurer que la condition devienne fausse en un nombre fini d'opérations. Il est par exemple possible de réécrire une boucle **for** à partir d'une boucle **while**

```
unsigned int i=m;
while (i<n){
...
i++;
}
```

Il reste le cas où la condition n'existe que si les instructions sont exécutées au moins une fois. Il existe la boucle **do-while**

```
do{
...
}
while(condition);
```

Comme je sais que j'ai affaire à un public où il y a des gamers de folie, vous pouvez rencontrer parfois une boucle infinie. On peut sortir de cette boucle avec l'instruction **break**. Voici un code élémentaire qui utilise cette astuce.

```
int main()
{
	while(1)
	{
		char c;
		std::cin>>c;
		if (c=='q') break;
	}
	exit(0);
}
```

Executez ce code et comprenez le ! Pour le calcul scientifique, je vous déconseille cette procédure car il faut être sur que vous pouvez sortir de la boucle. La deuxième raison est que nous devons rester sérieux !

4.4 Les instructions conditionnelles : if, case, switch, break

La syntaxe des instructions conditionnelles est très standard et similaire à celle du langage C. Si la condition à examiner est simple ou double, l'instruction **if** est recommandée,

```
if (condition)
{
instructions1;}
else
{
instructions2}
```

Cette instruction signifie que si la condition est vraie, les instructions du groupe 1 sont exécutées, tandis que si la condition est fausse, les instructions du groupe 2 sont exécutées.

Les instructions **switch**, **case** et **break** ont la structure suivante

```
switch (isc)
{
case 1: instructions1;break;
case 5: instructions2;break;
case 6: instructions3;
default : instructionsN;break;
}
```

Dans ce dernier exemple, on a le comportement suivant : si la **isc** vaut 1 les instructions 1 sont exécutées, si la **isc** vaut 5 les instructions 2 sont exécutées, si la **isc** vaut 6, les instructions 3 et N sont exécutées, si la **isc** est différente des trois autres valeurs, les instructions N sont exécutées.

4.5 Opérateurs d'assignation et d'incrémentation

Le signe = est l'opérateur d'assignation d'une variable : il signifie que le résultat du membre de droite est placé dans la mémoire de la variable donnée dans le membre de gauche. A noter que ce n'est pas la

même signification que le signe = dans une équation mathématique.

Il existe en C++ plusieurs opérateurs permettant d'incrémenter une variable

++	incrémente d'une unité
−−	décrémente d'une unité
+ =	ajoute le membre de droite au membre de gauche
− =	soustrait le membre de droite au membre de gauche
* =	multiplie le membre de droite par le membre de gauche
/ =	divise le membre de droite par le membre de gauche

Les deux premiers opérateurs sont qualifiés d'unaires. Cela signifie qu'ils n'ont besoin que d'un seul opérande. L'instruction

```
x++;
```

est équivalente à

```
x=x+1;
```

Comme nous venons de le voir une grande partie de la syntaxe est identique en C et en C++. A noter que les opérateurs unaires sont parfois utilisés à l'intérieur d'une instruction d'assignation permettant avec une seule instruction, de réaliser deux opérations. Bien entendu, il faut connaître l'ordre d'exécution

```
x=a++;
```

Cette instruction signifie que, **x** est d'abord assigné à la valeur de **a** et ensuite on incrémente **a** de 1. Inversement l'instruction suivante

```
x=++a;
```

signifie que on incrémente d'abord **a** de 1, puis on donne à **x** la nouvelle valeur de **a**.

Cette syntaxe est à déconseiller à des débutants (et peut-être au delà). On peut très bien substituer aux deux instructions précédentes une syntaxe plus lisible

```
x=a;a++;
```

```
a++;x=a;
```

4.6 Opérateurs de comparaison

Pour formuler une expression logique, il est souvent nécessaire d'avoir un opérateur de comparaison qui s'applique à deux expressions typées. Le langage C++ fournit plusieurs opérateurs que nous rassemblons ci-dessous

<	inférieur(strictement) à
>	supérieur (strictement) à
<=	inférieur ou égal à
>=	supérieur ou égal à
! =	différent de
==	égal à

Par exemple,

```
double a=5, b=7;
bool c;
 c=a>b;
```

le résultat de la comparaison par l'opérateur > est une variable booléenne qui est placée dans la variable **c** (dans ce cas le résultat est **False**).

Il existe aussi des opérateurs de comparaison logique en C++ pour obtenir des résultats booléens à partir de deux expressions logiques

&&	et logique
\|\| | ou logique |

4.7 la bibliothèque standard : STL

Le langage C++, comme le langage C, nécessite d'inclure des fichiers d'en-tête pour que le compilateur (préprocesseur pour être précis) puisse connaître le type de classes à utiliser. Nous allons tout d'abord voir les fichiers qui sont associés aux éléments de base du C++. Le C++ dispose aussi de bibliothèques importantes qui permettent à la fois une écriture plus simple, mais aussi de disposer d'outils pour résoudre des problèmes numériques avec efficacité. Une première bibliothèque s'appelle la STL (pour Standard Template Library). Elle repose sur la construction de templates qui sont des modèles, pour permettre la réécriture de manière plus simple de concepts abstraits. Comme beaucoup de bibliothèques récentes en C++, il n'y a pas de bibliothèque binaire à lier à l'exécutable au moment de la compilation, mais la bibliothèque n'est constituée que de fichiers d'en-tête. L'intérêt de cette construction est qu'on laisse au compilateur le soin de construire les outils adaptés au code au moment de la compilation et donc cela évite d'avoir une bibliothèque qui dépend de la version du compilateur. Le prix à payer est la lenteur que l'on peut constater pour la compilation d'un programme en C++. En effet, le travail laissé au compilateur est beaucoup plus important que pour les autres langages.

La STL contient aussi la bibliothèque standard du C ce qui permet d'utiliser les instructions du C pour ceux qui souhaitent faire une migration progressive du C au C++. La note de bas de page liste l'ensemble des fichiers traduits pour le C++ [1]

Si vous débutez en C++, il n'est donc pas nécessaire de faire appel à des fichiers et de choisir une écriture C++ plus spécifique.

Revenons sur ces spécificités du C++. La STl contient cinq composantes

- Containers (Conteneurs) Contient la définition d'une liste d'objets abstraits que nous allons voir ci-dessous
- Algorithms (Algorithmes) Contient des algorithmes permettant de réaliser des opérations sur les différents conteneurs
- Functors (Foncteurs) Permet de créer des objets qui surchargent l'appel de fonctions. A ce stade, c'est très abstrait, mais nous allons nous concentrer sur les applications
- Iterators (Iterateurs) Permet de manipuler des séquences de containers précédemment définis. En termes simples, c'est une généralisation de la notion d'indice pour un tableau.

1. <cassert> (ANSI C++) == <assert.h> (ANSI C) <cctype> (ANSI C++) == <ctype.h> (ANSI C) <cerrno> (ANSI C++) == <cerrno.h> (ANSI C) <cfloat> (ANSI C++) == <float.h> (ANSI C) <ciso646> (ANSI C++) == <ciso646.h> (ANSI C) <climits> (ANSI C++) == <limits.h> (ANSI C) <clocale> (ANSI C++) == <locale.h> (ANSI C) <cmath> (ANSI C++) == <math.h> (ANSI C) <csetjmp> (ANSI C++) == <setjmp.h> (ANSI C) <csignal> (ANSI C++) == <signal.h> (ANSI C) <cstdarg> (ANSI C++) == <stdarg.h> (ANSI C) <cstddef> (ANSI C++) == <stddef.h> (ANSI C) <cstdio> (ANSI C++) == <stdio.h> (ANSI C) <cstdlib> (ANSI C++) == <stdlib.h> (ANSI C) <cstring> (ANSI C++) == <string.h> (ANSI C) <ctime> (ANSI C++) == <time.h> (ANSI C) <cwchar> (ANSI C++) == <wchar.h> (ANSI C) <cwtype> (ANSI C++) == <wtype.h> (ANSI C)

- Gestion des entrées/sorties : Saisir des données au clavier (une manière très vintage et peu recommandée), l'affichage de résultats à l'écran (rapidement peu gérable) ; mais aussi lecture et écrire de fichiers.

La STL permet de créer des objets à partir de conteneurs dont les formes sont diverses. On peut donc bien évidemment créer des tableaux comme dans tout langage, mais aussi des objets plus sophistiqués comme des listes, des paires,... Cette bibliothèque continue à évoluer comme le langage C++. Voici une liste partielle des objets que l'on peut créer

- <array> Permet de créer un tableau proche des spécificités du langage C.
- <bitset> Permet de créer un tableau de bits.
- <deque> Permet de créer une queue de données dans laquelle on peut enlever et ajouter des éléments à chaque extrémité de cette queue.
- <forward_list> Permet de créer une liste chainée (C++11).
- <fstream> Permet de créer un fichier , de lire son contenu et d'écrire des données dans un fichier
- <iostream> Permet de saisir des données au clavier et d'afficher des résultats à l'écran.
- <list> Permet de créer une liste doublement chaînée.
- <map> Permet de créer un tableau associatif.
- <queue> Permet de créer une queue de données.
- <set>Permet de créer un ensemble de données.
- <stack> Permet de créer une pile.
- <vector> Permet de créer un tableau dynamique.

Il y a encore de nombreux fichiers d'en-tête dans la bibliothèque standard que nous n'allons pas détailler : <bitset> <exception> <functional> <iomanip> <ios> <iosfwd> <istream> <iterator> <limits> <list> <locale> <map> <memory> <new> <numeric> <ostream> <queue> <set> <sstream> <stack> <stdexcept> <streambuf> <string> <typeinfo> <utility> <valarray>

Cette brève description peut faire redouter le pire dans l'apprentissage et l'usage du C++ car le nombre de concepts à connaître, ainsi que leur utilisation peut s'avérer d'une complexité redoutable. Ce n'est bien entendu pas le but et la démarche de ce cours et ce pour plusieurs raisons : Le temps imparti pour l'apprentissage de ce langage complexe dépasse de loin celui dédié à cet Unité d'Enseignement. Dans une première approche, il s'agit d'être opérationnel pour résoudre des problèmes concrets, dans un temps raisonnable. De plus, dans cette première approche, nous souhaitons passer de manière douce des concepts du Python à ceux du C++, en restant en partie sur une logique de programmation procédurale dans une première partie.

La démarche que nous allons suivre est la suivante : utiliser, quand cela est nécessaire les concepts développés dans une bibliothèque (la STL en premier, mais pas uniquement) et rester proche de la programmation procédurale, même si on utilise par moment ceux de la programmation objet, mais sans développer des nouveaux outils dans cette logique de programmation. Dans les deux chapitres consacrés à la programmation objet pour nos deux langages, nous irons un peu plus loin pour comprendre comment définir de nouveaux outils. Dans ce chapitre, nous resterons utilisateur d'outils complexes, sans aller dans le détail de leur création.

Le fil rouge de ce chapitre sera d'illustrer chaque concept introduit par des mini-codes qu'il est nécessaire de compiler, de tester et ensuite de modifier pour s'approprier rapidement les nouveaux éléments que l'on présente.

Pour faire appel aux méthodes d'une classe il est par défaut nécessaire de donner le nom de la classe suivi de **: :** puis du nom de la méthode. Cette 'lourdeur' peut-être évitée avec l'instruction **using namespace** suivie du nom de la classe. Cela permet d'utiliser le nom de la méthode dans le code sans mentionner le nom de la classe. Le défaut de l'utilisation de **namespace**) est que si l'on a une fonction

ou une classe avec le même nom que la fonction membre de la classe, on risque d'avoir une confusion. En pratique, pour les codes de taille modeste ou moyenne, le risque est très faible et nous utiliserons la plupart du temps ce raccourci car cela allège l'écriture.

4.8 Les entrées et les sorties

4.8.1 Clavier et écran

Pour lire des données formatées sur l'entrée standard (clavier), on utilise la fonction membre **c**in et pour afficher à l'écran (sortie standard) les résultats, on utilise la fonction membre **cout**. Pour un programme C++, il faut introduire le (ou les) fichier(s) en-tête nécessaire(s), définir le début du programme principal qui est de type **int**.

```
#include <iostream>
int main()
{
   double a;
   std::cout<<"donnez un nombre"<<std::endl;
   std::cin>>a;
   std::cout<<"ce nombre est egal a "<<a<<std
       ↪ ::endl;
   std::cout<<a;
   exit(0);
}
```

```
#include <iostream>
using namespace std;
int main()
{
   double a;
   cout<<"donnez un nombre"<<endl;
   cin>>a;
   cout<<"ce nombre est egal a "<<a<<endl;
   cout<<a;
   exit(0);
}
```

Les deux programmes correspondent à la même tache : dans celui de gauche, les fonctions membres sont appelées à partir de la classe **std**, tandis que dans le programme de droite l'usage de namespace a réduit l'écriture du code.

Dans le mini-code précédent, on déclare une variable réelle que l'on utilise par la suite. Pour afficher à l'écran le message d'invite, on utilise la fonction cout suivi de deux chevrons de type inférieur «. (Pour avoir un moyen mnémotechnique sur le sens, on peut dire que les arguments qui suivent sont envoyés vers la sortie standard.) L'instruction **endl** permet de déplacer le curseur qui se trouve à la fin de ligne au premier caractère de la ligne suivante. L'instruction **cin** signifie les caractères saisis au clavier sont "envoyés" dans la variable (en fait ils sont transformés en un nombre car le type de variable **a** est un double). On affiche le résultat obtenu sur la ligne suivante. L'instruction **exit(0)** signifie la fin du programme principal avec la valeur **0** qui signifie que la sortie du programme s'est bien passée.

La saisie au clavier ne présentant aucun intérêt (on ne fait pas des programmes pour remplacer une calculatrice), on n'utilise pas ce genre de méthodes en exercices. Cette section illustre simplement le concept d'entrée-sortie, mais pour un programme plus conséquent, on lit un fichier de données. Cela permet de garder la mémoire des paramètres utilisés et on écrit les résultats sur un fichier, afin de pouvoir traiter les résultats avec d'un grapheur.

4.8.2 Lecture et écriture de fichiers

Pour lire les données formatées sur un fichier, on fait appel à la classe des entrées sorties définies dans **fstream**.

On considère le fichier de données **fichier.dat** dont le contenu est

```
1 3.1
2 5.1
```

Pour lire ce fichier et l'afficher à l'écran, on écrit

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
   double a,b;
   ifstream fichier;
   fichier.open("fichier.dat");
   while (fichier>>a>>b)
   {
   cout<<"a = "<<a<<" et b = "<<b<<endl;}
      fichier.close();
   exit(0);
}
```

Pour écrire sur un fichier, il faut préalablement définir un descripteur de fichier, puis associer ce descripteur à un nom de fichier au moment de son ouverture. L'écriture est analogue à celle sur la sortie standard. Une fois tous les résultats écrits, on ferme le fichier avant de terminer le programme

```
#include <fstream>
using namespace std;
int main()
{
   double a=23;
   ofstream fichier ;
   fichier.open("fichier.txt");
   fichier<<"ce nombre est egal a "<<a<<endl;
   fichier.close();
   exit(0);
}
```

4.9 Variable : adresse et portée, pointeur et référence

Pour développer un code un peu conséquent (c'est-à-dire qui dépasse les 30 lignes), il est nécessaire d'organiser celui-ci en différentes parties qui, de préférence, seront largement indépendantes. Cela est vrai dans une logique de programmation procédurale, mais aussi voire plus dans le cas d'une programmation objet. Les langages eux-mêmes ont évolué avec le temps pour favoriser et souvent développer ces méthodes. La raison de cette évolution est simple : quand on écrit un code dont les fonctionnalités augmentent, le nombre de lignes augmente aussi. La probabilité de générer des erreurs de codage s'accroît alors de manière importante et la mise au point peut être chronophage. Une mauvaise écriture du code, peut aussi entraîner une instabilité lors de l'exécution, et rapidement, une impossibilité de recycler celui-ci dans un projet plus vaste. Pour comprendre un peu plus en profondeur cette logique, il est nécessaire d'introduire quelques notions fondamentales.

4.9.1 Adresse d'une variable

Nous avons vu la manière de définir une variable en C++ qui consiste à choisir le type, suivi d'un nom. Une fois créée, cette variable peut alors être assignée à une valeur, avec une instruction supplémentaire ou directement sur la ligne de déclaration. Quand le code est transformé, c'est-à-dire compilé, on génère un fichier exécutable qui pourra être lancé pour réaliser la tâche souhaitée.

Quand l'exécution démarre, une partie de la mémoire de l'ordinateur est réservée à l'exécution : pour chaque programme. Celle-ci est divisée en quatre parties : la première concerne les variables (DATA en anglais), une seconde pour les instructions du programme (pas celles écrites en C++, mais celles du

code binaire) une troisième partie que l'on appelle la pile (STACK en anglais) utilisée en autres lors de l'appel de fonctions et une dernière que l'on appelle le Tas (HEAP en anglais), la mémoire disponible sur l'ordinateur à un instant donné et qui peut devenir disponible pour le programme en cours d'exécution. Lors de la création de tableaux dynamiques, l'extension de la mémoire demandée à l'ordinateur est prise sur le tas.

Nous allons tout d'abord nous focaliser sur la première partie. La mémoire d'un ordinateur est utilisée par le processeur pour faire des échanges fréquents entre le processeur qui réalise les opérations logiques et arithmétiques en utilisant très souvent des variables et la mémoire en déposant les variables modifiées dont un emplacement existe en mémoire. Il y a donc un mécanisme qui permet au processeur d'aller chercher la variable dans un endroit précis de la mémoire. Chaque octet de la mémoire est repéré par une adresse, c'est un nombre entier qui commence à 0 et peut aller jusqu'à la taille maximale de la mémoire.

Il est possible de connaître cette information en C++ (et pour la plupart des langages) en utilisant l'opérateur d'indirection &.

```
#include <iostream>
using namespace std;
int main()
{
    double a=23;
    int b=2;
    int c;
    c=4;
    cout<<"a␣valeur␣et␣adresse"<<a<<"␣"<<&a<<endl;
    cout<<"b␣valeur␣et␣adresse"<<b<<"␣"<<&b<<endl;
    cout<<"c␣valeur␣et␣adresse"<<c<<"␣"<<&c<<endl;
    cout<<"difference␣adresse␣c-b␣"<<&c-&b<<endl;
    exit(0);
}
```

Cela donne le résultat suivant.

```
a valeur et adresse23 0x7ffd27d0f7c0
b valeur et adresse2 0x7ffd27d0f7b8
c valeur et adresse4 0x7ffd27d0f7bc
difference adresse c-b 1
```

Dans ce mini-code, on déclare 3 variables : une réelle et deux entières (de deux manières différentes, mais équivalentes). **&a** est l'adresse de la variable **a**,.. Le résultat est exprimé en hexadécimal pour chaque variable. La dernière ligne de code calcule la différence d'adresses entre les deux variables **b** et **c**. Cette opération est possible entre adresses de variables de même type, le résultat donne 1 qui correspond à la taille d'un entier et non pas d'un octet. Si on fait la différence entre les deux adresses de c et b données sur le résultat, on trouve 4, ce qui correspond à la taille d'un entier en C++. A noter que, si on recommence l'exécution du code, les adresses mémoires changent car le programme a une probabilité faible d'être à nouveau placé au même endroit que la fois précédente. Par contre, la différence d'adresses reste la même.

4.9.2 Portée d'une variable

Quand on construit un code, il est nécessaire d'identifier l'utilisation des différentes variables dont on a besoin pour faire des calculs. Il y a des données que l'on ne souhaite pas changer au cours du calcul, des variables dans lesquelles on écrit les résultats du calculs et des variables que l'on appellera locales, et qui servent pour des calculs intermédiaires, mais dont on ne souhaite pas conserver le contenu au delà du calcul. Il est très fortement recommandé de respecter cette règle pour construire un code qui évitera les erreurs de programmation. L'évolution des langages de programmation permet de réaliser cet objectif entre les différentes parties d'un code. Si vous partagez une variable avec d'autres parties d'un code, vous vous exposez à ce que ces variables soient modifiées sans que vous le souhaitiez. Si on peut prendre une image simple, si vous utilisez la même éponge pour nettoyer votre vaisselle et pour nettoyer vos toilettes, vous risquez un problème d'hygiène rapidement. A nouveau, cette règle semble contraignante sur un code court, mais elle se révèle indispensable sitôt que l'on construit un code un

tout petit peu complexe. Nous allons la retrouver très rapidement dans l'utilisation des fonctions.

Pour réaliser cela simplement, les accolades permettent de limiter la portée des variables à l'intérieur d'un ensemble d'instructions. Le mini-code suivant illustre cette règle :

```
#include <iostream>
using namespace std;
int main()
{
    double a=23;
    cout<<"a␣valeur␣"<<a<<endl;
    {
    int c;
    c=4;
    cout<<"c␣valeur␣"<<c<<endl;
    }
exit(0);
}
```

```
#include <iostream>
using namespace std;
int main()
{
    double a=23;
    cout<<"a␣valeur␣"<<a<<endl;
    {
    int c;
    c=4;
    cout<<"c␣valeur␣"<<c<<endl;
    }
cout<<"c␣valeur␣"<<c<<endl;
exit(0);
}
```

Compilez et exécutez le code de gauche, puis celui de droite, vous obtenez alors l'erreur de compilation suivante.

```
In function int main():
portee2.cpp:12:20: error: c was not declared in this scope
12 | cout<<"c␣valeur␣"<<c<<endl;
```

Ce qui signifie clairement que la variable **c** n'existe plus en dehors des accolades. Cette notion de portée est très souvent au cœur des langages modernes, car elle permet d'éviter toutes les erreurs de programmation que l'on rencontrait dans les langages anciens. Soyez rassuré il risque suffisamment de moyens de générer des erreurs de programmation que l'on crée soi-même. C'est donc une chance d'avoir des langages qui évitent une grande partie des écueils classiques.

4.9.3 Pointeur d'une variable

Nous venons de voir qu'il est possible d'obtenir une information sur la localisation d'une variable dans la mémoire de l'ordinateur au cours de l'exécution. L'étape suivante consiste à créer une variable qui contiendra l'adresse d'une variable. On appelle cette variable un pointeur. Pour déclarer cette variable, on doit déclarer le type de pointeur, suivi de l'opérateur * et du nom du pointeur.

```
double *c;
```

A ce stade, on a créé une variable à qui l'on peut assigner l'adresse d'une variable de type double, mais on ne précise pas encore cette adresse. Il faut donc déjà avoir une variable qui existe. On assignera une première variable par l'instruction **double a=2.0;** Pour que le pointeur **c** soit assigné avec l'adresse de **a**, on écrit

```
c=&a;
```

qui signifie que la valeur du pointeur est assignée à l'adresse de la variable **a**.

Une dernière fonctionnalité associée aux pointeurs est de pouvoir récupérer la valeur de la case mémoire associée à l'adresse du pointeur avec l'opérateur de déréférencement qui est l'étoile (à ne pas confondre avec la multiplication car il s'agit ici d'un opérateur unaire.)

```
double b=*c;
```

Cette instruction signifie que la variable réelle **b** est assignée à la valeur correspondant à l'adresse qui se trouve stockée dans la variable **c**, c'est à dire **a** puisque **c** pointe sur **a**.

Le mini-code suivant illustre le fait que l'on peut changer la valeur d'une variable en utilisant un pointeur de cette variable

```
#include <iostream>
using namespace std;
int main()
{
   double a=23;
   double *b;
   b=&a;
   cout<<"a␣valeur␣et␣adresse␣"<<a<<"␣"<<&a<<endl;
   cout<<"*b␣valeur␣pointee␣par␣b␣"<<*b<<endl;
   cout<<"␣valeur␣et␣adresse␣de␣b␣"<<*b<<"␣"<<b<<"␣"<<&b<<endl;
   *b=5.0;
   cout<<"b␣et␣a␣valeurs␣"<<*b<<"␣"<<a<<endl;
   exit(0);
}
```

Le résultat est le suivant

```
a valeur et adresse 23 0x7ffe7194b308
*b valeur pointee par b 23
valeur et adresse de b 23 0x7ffe7194b308 0x7ffe7194b310
b et a valeurs 5 5
```

On voit bien que **b** pointe sur la variable **a**, et que l'opérateur de déférencement agissant sur **b** (***b**) permet d'obtenir la valeur de **a**. La valeur de **b** est une adresse et **b** a elle-même à une adresse. Finalement l'instruction ***b=5.0** change la valeur de la variable **a** comme le montre l'instruction de la ligne suivante. Cette manière indirecte de modifier une variable semble bien compliquée. A l'intérieur d'un code, ce n'est pas nécessaire, mais nous verrons que cela apparaît parois dans les fonctions (enfin celles dont l'écriture est basée sur le C).

Pour ceux qui n'avaient pas encore vu cette notion de pointeur, cela semble une construction bien compliquée et pas très utile pour un programme court. Si la notion de pointeur se limitait à cet usage, cela serait bien évidemment assez peu utile, et cela se limiterait à écrire un code rapidement cryptique (certains peuvent s'en délecter en écrivant des programmes que les autres ne peuvent comprendre la signification). Si la lecture d'un code devient inutilement compliquée, on s'éloigne fortement de l'idée originale d'obtenir une interface personne-machine sympa. Le code suivant illustre ce concept avec une structure emboîtée des pointeurs. Exécutez ce code et comprenez le résultat.

```
#include <iostream>
using namespace std;
int main()
{
   double a=23;
   double *b;
   double *c;
   double **d;
   b=&a;
   d=&b;
   cout<<"a␣valeur␣et␣adresse␣"<<a<<"␣"<<&a<<endl;
   cout<<"*b␣valeur␣pointee␣par␣b␣"<<*b<<endl;
   cout<<"␣valeur␣et␣adresse␣de␣b␣"<<*b<<"␣"<<b<<"␣"<<&b<<endl;
   **d=5.0;
   cout<<"valeurs␣de␣d,␣b␣et␣a␣:␣"<<**d<<"␣"<<*b<<"␣"<<a<<endl;
   exit(0);
}
```

Comme nous allons le voir rapidement, nous allons garder les idées fortes de notion de pointeur pour comprendre les fonctions que nous allons voir, mais nous allons utiliser rapidement d'autres méthodes dont l'écriture est beaucoup plus simple et se rapproche du Python.

4.9.4 Référence d'une variable

Une dernière notion spécifique du C++ est la notion de référence d'une variable. Une référence est définie à partir de l'opérateur **&**.

```
int a;
int &b=a;
// En d'autres termes, on peut dire que b est un alias de a. On a
//deux noms pour la meme case memoire
a=3;
b++;
```

a vaut maintenant 4.

A nouveau, l'utilité de ce concept apparaîtra lors de la création de fonction et aussi de classes.

4.10 Tableaux statiques

On peut créer en C++ des tableaux à une ou plusieurs dimensions. Si on connaît à l'avance la dimension exacte du tableau à créer, il existe une déclaration simple que nous allons voir en premier. Dans un second temps, nous allons considérer le cas, où la dimension d'un tableau n'est pas connue à l'avance et où l'on souhaite à la fois le créer, éventuellement modifier sa taille et finalement le détruire. Cette seconde méthode donne un champ de possibilités remarquables, mais peut s'accompagner d'une baisse de performance de l'efficacité du calcul en cas d'utilisation répétée. Nous verrons que le C++ offre des outils pour simplifier l'écriture de la manipulation des tableaux.

4.10.1 Tableau à une dimension : vecteurs

La structure la plus simple est celle du tableau à une dimension. Comme en Python les éléments du tableau sont repérés par un indice entier par défaut. Pour déclarer un tableau avec un nombre d'éléments fixé à l'avance : il faut alors donne le type du tableau suivi du nom du tableau avec le nombre d'éléments donnés entre crochets. C'est l'occasion de revenir sur la notion de portée d'une variable. Quand on parcourt les éléments d'un tableau (en écriture ou en lecture), on a besoin d'un entier pour accéder aux différents éléments avec une boucle. Dans le cas d'un tableau statique, c'est généralement une boucle **for** puisque le nombre d'élément est connu à l'avance. La variable entière est typiquement une variable locale dont la portée doit être limitée à la boucle. En C++, on peut (et je dirais même plus, on doit) déclarer cette variable à l'intérieur de la boucle. On évite de facto de déclarer une variable en début de code qui risque d'être utilisée ou modifiée de manière fortuite par un autre partie du code sans que cela soit souhaité. A noter que cette fonctionnalité a été intégrée dans le langage C. L'exemple suivant illustre ce point.

```cpp
#include <iostream>
using namespace std;
int main()
{
    double a[10];
    for (int i=0;i<10;i++)
    {
        a[i]=2.0*i;
        cout<<"a["<<i<<"]="<<a[i]<<endl;
    }
  exit(0);
}
```

```cpp
#include <iostream>
using namespace std;
int main()
{
    double a[10];
    for (int i=0;i<10;i++)
    {
        a[i]=2.0*i;
        cout<<"a["<<i<<"]="<<a[i]<<endl;
    }
  cout<<i<<endl;
  exit(0);
}
```

Notons que le premier élément d'un tableau correspond à l'indice 0 et finit au nombre maximal moins 1, dans ce cas 9. Compiler le code de gauche, puis celui de droite, on obtient une erreur clairement identifiée dans le second cas

```
error: i was not declared in this scope
11 | cout<<i<<endl;
|
```

Cela correspond au fait que la variable i n'existe plus quand on sort de la boucle. Sa durée de vie a été limitée à son utilisation et on diminue le risque qu'elle soit utilisée de manière différente.

Comme nous sommes maintenant chaud bouillants sur les pointeurs, voici un mini-code qui illustre le fait que le nom du tableau est un pointeur sur le premier élément du tableau.

```cpp
#include <iostream>
using namespace std;
int main()
{
    double a[10];
    cout<<a<<endl;
    for (int i=0;i<10;i++)
    {
        *(a+i)=2.0*i;
        cout<<"a["<<i<<"]="<<*(a+i)<<"␣adresse␣"<<a+i<<endl;
    }
  exit(0);
}
```

Ce code réalise le remplissage du tableau de manière identique au code précédent mais utilise les pointeurs. En ajoutant **i** à l'adresse de **a** on déplace l'adresse mémoire de $8\times i$ octets et on obtient l'adresse du ième élément du tableau. Cet exemple n'est pas fait pour vous encourager dans ce type d'écriture, mais pour vous aider à comprendre sa signification quand vous la rencontrez sur des codes anciens. Le résultat de l'exécution de ce code est

```
0x7fff6bc76410
a[0]=0 0x7fff6bc76410
a[1]=2 0x7fff6bc76418
a[2]=4 0x7fff6bc76420
a[3]=6 0x7fff6bc76428
a[4]=8 0x7fff6bc76430
a[5]=10 0x7fff6bc76438
a[6]=12 0x7fff6bc76440
a[7]=14 0x7fff6bc76448
a[8]=16 0x7fff6bc76450
a[9]=18 0x7fff6bc76458
```

Dernière remarque : on voit bien que l'écart entre le premier élément et le deuxième est de 8 et apparemment de 2 entre le deuxième et le troisième. Apparence trompeuse : n'oublions pas que l'on compte en hexadécimal et cela fait bien un écart de 8 octets.

Tableau à deux dimensions : matrices

Très utiles pour définir des matrices, les tableaux à deux dimensions peuvent se définir de manière statique en utilisant une définition dans le même esprit que celles pour les tableaux à une dimensions. Pour un tableau de 6 lignes et 4 colonnes de nombres réels, on a l'instruction

```
int tab[6][8];
```

Le mini-code suivant illustre la manière de remplir un tableau d'entiers à deux dimensions et d'accéder aux valeurs de ces éléments de trois manières différentes. Bien évidemment la première manière sera privilégiée !

```
#include <iostream>
using namespace std;
int main()
{
   int tab[6][8];
   for (int i =0; i<6; i++)
      for (int j=0; j<8; j++)
      {
         tab[i][j]=i+j;
         cout<<tab[i][j]<<"␣"<<*(tab[i]+j)<<"␣"<<*(*(tab+i)+j)<<endl;
      }
   exit(0);
}
```

En modifiant simplement ce programme, vous pouvez afficher les adresses de chaque élément et vérifier que le rangement des tableaux à deux dimensions se fait par lignes.

4.11 Tableaux dynamiques

4.11.1 Tableau à une dimension

Allocation standard

Le principe est le suivant : on crée tout d'abord un pointeur de type correspondant au tableau souhaité (par souci de sécurité on initialise celui-ci à 0). On utilise l'instruction **new** suivie du type du tableau et du nombre d'éléments voulus. Le mini-code suivant illustre ce concept (la ligne commentée est la manière de créer en une seule instruction le tableau qui peut remplacer les deux lignes précédentes).

```
#include <iostream>
using namespace std;
int main()
{
    double *a=NULL;
    a=new double [10];
// double *a=new double[10];
    for (int i =0;i<10;i++)
        {
        a[i]=i*i;
        cout<<a[i]<<"␣"<<*(a+i)<<endl;
        }
exit(0);
}
```

On voit que les éléments sont rangés de manière similaire aux tableaux statiques. Toutefois, la mémoire utilisée se situe dans la région du tas (HEAP). Si la taille demandée est plus grande que la mémoire disponible, la valeur retournée est la valeur initiale et on peut sécuriser le programme en ajoutant la ligne suivante après la création du tableau **if(a==NULL) exit(1);**. Cela ordonne au programme de ne pas continuer le calcul car le tableau n'a pas pu être créé, et doit donc s'interrompre. Le chiffre 1 signifie que l'interruption du programme est liée à une erreur d'exécution, contrairement à la valeur 0 qui correspond à la fin normale d'un programme.

Avant de quitter le programme, il est nécessaire (ou préférable) de libérer la mémoire qui a été demandée par l'utilisateur, car il y a un risque que l'espace réservé par l'utilisateur ne soit pas redonné au système. L'instruction correspondante est alors

```
delete [] a;
```

En fait, ce problème existait dans les anciens systèmes d'exploitation, mais les systèmes récents font le ménage quand le programme est terminé (garbage collector en anglais) et si on oublie de libérer la mémoire, le système s'en charge (Quelqu'un qui fait le ménage quand on laisse le bazar laisse sans doute rêveur certains, mais cela n'existe qu'en informatique). Par contre, un bug classique peut survenir pour les programmeurs trop précautionneux. Si on exécute deux fois la même instruction delete sur un tableau, le programme s'interrompt brutalement sur une erreur.

Allocation dynamique par la STL

La méthode précédente est la méthode historique, mais le C++ aime (voire raffole) de mettre en place des méthodes alternatives, un comportement similaire au Python, et qui permet souvent d'avoir une écriture plus simple et aussi des fonctionnalités plus importantes. Comme le monde idéal (même en informatique) n'existe pas, le prix à payer peut être une (légère) baisse de performances. Cet effet s'estompe avec le temps et il est préférable d'utiliser les outils évolués plutôt que de fossiliser un langage.

Le mini-code suivant crée un tableau à une dimension de 10 éléments à partir de la STL. Il est nécessaire de charger le fichier d'en-tête **vector**. Pour définir ce tableau, on utilise **vector** suivi du type que l'on souhaite placé entre chevrons suivi du nom du tableau. L'argument entre parenthèses correspond aux nombres d'éléments du tableau.

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector <double> a(10);
    for (int i =0;i<10;i++)
        {
        a[i]=i*i;
       // cout<<*(a+i)<<endl;
        cout<<a[i]<<endl;

        }
exit(0);
}
```

Attention, on n'a plus accès aux adresses des éléments du tableau par cette méthode , ce qui explique que j'ai placé en parenthèses l'instruction qui ne permet pas la compilation.

On a déjà un peu gagné en lisibilité, mais on va aller plus loin pour se rapprocher du Python. Dans ce second mini-code, on crée un tableau vide (pas de parenthèses dans la déclaration) et on augmente dans la boucle la taille du tableau élément par élément. On vérifie que le résultat est le même que précédemment.

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector <double> a(10);
    for (int i =0;i<20;i++)
        {
        a.push_back(i*i);
        cout<<a[i]<<endl;
        }
exit(0);
}
```

Le choix entre ces deux dernières méthodes dépend du problème à traiter. Pour un grand tableau, mieux vaut réserver la mémoire au départ que d'augmenter celle-ci élément par élément.

Le dernier mini-code utilise les propriétés des tableaux de la bibliothèque **vector**.

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector <double> a;
    for (int i =0;i<20;i++)
        a.push_back(i*i);
    for (auto ia: a)
        cout<<ia<<endl;

exit(0);
}
```

La deuxième boucle n'utilise pas un entier pour parcourir les éléments du tableau mais les éléments du tableau eux-mêmes et a la double vertu, en utilisant le type **auto**, de sélectionner le type d'élément du

tableau (dans le cas présent un réel) et de s'arrêter sur le dernier élément sans débordement de taille. On reviendra plus largement sur cette manière d'écrire un code dans le chapitre de la programmation objet, mais on entrevoit l'avantage de cette écriture : universalité (on change de type, le code est le même) et on évite les bugs.

4.11.2 Tableaux à deux dimensions

Allocation standard

Pour avoir un tableau à deux dimensions, on commence par créer un pointeur de pointeur du type souhaité (ci-dessous le type est double). On utilise **new** pour créer un tableau de pointeurs de double de taille correspondant au nombre de lignes de la matrice, puis on fait une boucle sur les éléments du tableau de pointeurs en créant des tableaux dont le nombre d'éléments correspond au nombre de colonnes. Cela donne le mini-code suivant

```
#include <iostream>
using namespace std;
int main()
{
    double **a=NULL;
    a=new double* [10];
    for (int i=0; i<10; i++) a[i]=new double [4];

    for (int i =0; i<10; i++)
    {
        for (int j =0; j<4; j++)
        {
            a[i][j]=i+j;
            cout<<a[i][j]<<"␣";
        }
        cout<<endl;
    }
    exit(0);
}
```

Pour les tableaux à trois dimensions (que l'on n'utilise pas tous les jours), il faut créer un pointeur de pointeur de pointeur et faire les allocations dynamiques correspondantes.

Si l'on veut éviter cette complexité, on peut "aplatir" le tableau à trois dimensions et créer un vecteur avec un nombre d'éléments égal à celui de la matrice Voici un code équivalent au code précédent où la matrice est remplacée par un vecteur.

```
#include <iostream>
using namespace std;
int main()
{
    double *a=NULL;
    a=new double [10*4];

    for (int i =0;i<10;i++)
    {
        for (int j =0;j<4;j++)
        {
        a[4*i+j]=i+j;
        cout<<a[4*i+j]<<" ";
        }
        cout<<endl;
    }
exit(0);
}
```

Cette écriture respecte l'ordre du C++ avec une écriture des éléments ligne par ligne.

Allocation dynamique par la STL

Bien évidemment, on peut utiliser à nouveau la bibliothèque STL, qui permet une écriture assez simple. La logique est la suivante : une matrice est un vecteur de l vecteurs de c éléments. l et c correspondent aux nombres de lignes et colonnes de la matrice à créer. L'initialisation est un peu plus subtile car le premier argument correspond au nombre de lignes et il faut répéter le type de vecteur avec le nombre d'éléments en argument. Le mini-code suivant considère un tableau à 2 dimensions :

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector <vector <double> > a(10,vector<double> (4));
    for (int i =0;i<10;i++)
    {
        for (int j =0;j<4;j++)
        {
          a[i][j]=i+j;
          cout<<a[i][j]<<" ";
        }
        cout<<endl;
    }
exit(0);
}
```

La question inévitable qui surgit avec des différentes méthodes est de savoir quelle est la meilleure. La réponse est la même faite avec Python qui a aussi une diversité d'écriture parfois compliquée à suivre. Si le problème est simple et que les tailles de matrices sont petites, il faut choisir ce qui est le plus rapide à écrire car les accroissements de performance seront ridicules par rapport au temps consacré au développement. A l'inverse, si les manipulations à exécuter sur les matrices sont importantes, il est préférable de choisir ce qui est le plus adapté à la bibliothèque que vous utiliserez. Les bibliothèques Eigen ou Armadillo permettent de définir vecteur et matrice d'une autre manière (plus lisible que les deux proposés ci-dessus) ainsi que des outils efficaces pour faire des calculs. Il existe un développement important des bibliothèques autour de ces deux langages que sont Python et C++. Les développeurs

de bibliothèques ont la double fonction d'écrire des codes efficaces et sécurisés, mais aussi d'utiliser la possibilité de réécrire ces langages. Ainsi en utilisant une bibliothèque, il convient de s'adapter, mais le temps gagné à ne pas réécrire des algorithmes connus et généralement très bien optimisés dans les bibliothèques fait pencher la balance pratiquement toujours du même coté.

Hormis les types prédéfinis en C++ qui ont été listés dans la section 4.2.1, il est possible de définir des types plus élaborés et aussi de définir des types à l'aide de l'instruction **t**ypedef.

4.12 Typedef

Cette instruction **typedef** justifie une section par son importance et non plus par sa difficulté. Elle permet de définir (ou de redéfinir) un type que l'on peut utiliser pour alléger l'écriture ou permettre un changement global sur un code qui utilise un type redéfini. Il est alors bien entendu nécessaire de documenter son code pour rendre celui-ci compréhensible à des utilisateurs externes. Prenons l'exemple suivant où l'on souhaite avoir un code où les types usuels sont écrits en Français. En plaçant dans un fichier d'en-tête ou au début du code les instructions suivantes

```
typedef int entier;
typedef real reel;
```

on peut par la suite définir tous les entiers et les réels de la manière suivante

```
entier i,j,k;
reel x,y,z;
```

Au delà de cet aspect nationaliste de l'écriture, cela permet par exemple de tester le code si on faisait les calculs non plus avec des réels 64 bits, mais par exemple, 32 bits. Si on remplace la deuxième ligne de la déclaration par

```
typedef float reel;
```

on peut recompiler le code pour faire une exécution presque identique à celle que l'on aurait avec un processeur 32 bits. C'est un effort très modeste par rapport à un changement sur toutes les lignes où la définition de variables réelles intervient. On va voir maintenant qu'il y a une application plus importante dans le cas des structures, et plus tard, dans la programmation objet.

4.13 Structure

Introduit dans le langage C, la notion de structure se retrouve naturellement en C++ et bien évidemment très enrichie quand on passe à la programmation objet. L'idée originelle de cette notion est d'avoir la possibilité de définir des types nouveaux qui permettent de combiner des types existants. Par exemple

```
typedef struct {
int i;
double x;
double y;
} num_pos_points;
```

définit une structure qui associe un point à deux dimensions repéré par deux coordonnées et un entier caractérisant ce point.

Ainsi défini dans l'en-tête du programme, on peut utiliser ce type comme un type habituel. Soit la déclaration de la structure cc

```
num_pos_points cc;
```

La syntaxe pour l'affectation des membres de la structure sera la suivante

```
cc.i=5;
cc.x=0.6;
cc.y=0.7;
```

Une fois que l'on a déclaré un élément de la structure, on peut assigner chacun des attributs de la structure en utilisant le point suivi du nom de l'attribut. Une fois la structure définie, on peut définir un tableau associé à cette structure On aura la syntaxe suivante

```
num_pos_points cc[10];
```

pour la déclaration d'un tableau statique.

Le mini-code suivant illustre ce concept

```
#include <iostream>
using namespace std;
typedef struct
{
    int i;
    double x;
    double y;
} num_pos_points;

int main()
{
    num_pos_points cc[10];
    for (int i =0; i<10; i++)
    {
        cc[i].i=i;
        cc[i].x=0.6*i;
        cc[i].y=0.7*i;
        cout<<cc[i].x<<"␣"<<cc[i].y<<"␣"<<
            ↪ cc[i].i<<"␣"<<endl;
    }
    exit(0);
}
```

```
#include <iostream>
using namespace std;
typedef struct
{
    int i;
    double x;
    double y;
} num_pos_points;

int main()
{
    num_pos_points *cc=new num_pos_points [10];
    for (int i =0; i<10; i++)
    {
        cc[i].i=i;
        cc[i].x=0.6*i;
        cc[i].y=0.7*i;
        cout<<cc[i].x<<"␣"<<cc[i].y<<"␣"<<cc[i].
            ↪ i<<"␣"<<endl;
    }
    exit(0);
}
```

Pour un tableau dynamique, si on utilise la première méthode, on définit tout d'abord un pointeur associé à ce nouveau type puis on utilise l'instruction **new** de manière similaire à ce que l'on a utilisé pour les tableaux d'entiers ou de réels. Cela correspond à l'instruction suivante

```
num_pos_points *cc=new num_pos_points [10];
```

à la place de **num_pos_points cc[10];**.

On a maintenant l'ensemble des éléments qui permettent de créer des structures complexes nécessaires à la programmation procédurale. L'étape suivante consiste à définir les transformations que l'on va exercer sur ces structures afin de pouvoir réaliser les calculs que l'on souhaite réaliser. Nous reviendrons sur cette notion dans le prochain chapitre sur le C++ pour aller sur la programmation objet qui est l'extension naturelle de ce que nous avons vu jusqu'à présent et qui lève les difficultés que nous rencontrons parfois dans la programmation procédurale. En attendant, le C++ offre pour la programmation procédurale de nombreux avantages que nous allons détailler par rapport aux autres langages.

4.14 Fonctions

4.14.1 Définition

Le principe d'une fonction est de transformer des données d'entrée en données de sortie modifiées, ce travail pouvant nécessiter des variables locales à l'intérieur de la fonction. Pour éviter de construire un code dans lequel on écrit les instructions les unes à la suite des autres, la notion de fonction structure un code, et permet de remplacer des séquences d'instructions par un appel à une fonction. Cela permet de réduire le programme principal avec un nombre limité d'instructions.

La définition d'une fonction comprend le type de celle-ci qui correspond aux arguments de sortie de la fonction, le nom de la fonction suivie d'une séquence de variables (placées entre parenthèses) dont le type est précisé. Dans le cas où aucun argument n'est retourné, le type de la fonction est déclaré **void**.

Souhaitant respecter au maximum la notion de portée des variables, le C et C++ ont introduit les règles suivantes : pour éviter que les variables d'entrée soient modifiées par la fonction, les valeurs de celles-ci sont recopiées dans un nouvel espace mémoire où les calculs de la fonction sont aussi exécutés. Quand la fonction retourne la variable ou la structure, celle-ci est recopiée de cet espace temporaire dans la région du programme principal.

Cette logique est très vertueuse car elle assure une protection des données qui n'existait pas à l'époque où les langages ne faisaient pas cette distinction. Le prix à payer est un (petit) ralentissement à l'exécution, lié à ces allers-retours entre différentes parties de l'espace mémoire.

On vérifie bien avec le mini-code suivant que la variable **x** n'a pas été modifiée et que **y** a bien été augmentée de la valeur 5.

```
#include <iostream>
using namespace std;
double ajoute_5(double x)
{
 double y=x+5;
 return(y);
}
int main()
{
double x=1.0;
double y=ajoute_5(x);
cout<<"␣x␣y␣"<<x<<"␣"<<y<<endl;
exit(0);
}
```

Dans la programmation scientifique, on doit très souvent transformer non pas quelques variables mais des tableaux et les allers-retours vont être très pénalisants à l'exécution si on procède élément par élément.

Il faut donc contourner les règles pour éviter des recopies et revenir dans le cas des tableaux sur une utilisation des variables directement sur le programme principal. Pour faire cela, on peut utiliser les pointeurs. En effet, on ne peut pas empêcher la fonction de recopier une variable, mais si on demande de recopier l'adresse de la variable dans une variable qui est un pointeur, on peut utiliser l'opérateur de déréférencement dans la fonction pour modifier les éléments du tableau.

```
#include <iostream>
using namespace std;
void ajoute_5(double *x, int n)
{
   for (int i=0; i<n; i++)
      x[i]+=5;
}
int main()
{
   double x[7]=
      ↪ {1.0,0.7,0.6,1.0,4.5,8,0.9};
   ajoute_5(x,7);
   for (int i=0; i<7; i++)
      cout<<"␣␣"<<x[i]<<endl;
   cout<<endl;
   exit(0);
}
```

```
#include <iostream>
using namespace std;
void ajoute_5(double *x,const int n)
{
   for (int i=0; i<n; i++)
      *(x+i)+=5;
}
int main()
{
   double x[7]=
      ↪ {1.0,0.7,0.6,1.0,4.5,8,0.9};
   ajoute_5(x,7);
   for (int i=0; i<7; i++)
      cout<<"␣␣"<<x[i]<<endl;
   cout<<endl;
   exit(0);
}
```

Le résultat est correct et il s'interprète de la manière suivante : le premier argument de la fonction est une adresse de pointeur double et le reste du code dans la fonction est écrit de manière usuelle. Le code de droite produit le même résultat, on a alors utilisé de l'algèbre de pointeur. A éviter car cela rend les codes assez obscurs !

Une variante du second programme dans la déclaration de la fonction est possible de la manière suivante

```
#include <iostream>
using namespace std;
void ajoute_5(double x[], int n)
{
   for (int i=0; i<n; i++)
      x[i]+=5;

}
int main()
{
   double x[7]=
      ↪ {1.0,0.7,0.6,1.0,4.5,8,0.9};
   ajoute_5(x,7);
   for (int i=0; i<7; i++)
      cout<<"␣␣"<<x[i]<<endl;
   cout<<endl;
   exit(0);
}
```

4.14.2 Polymorphisme

Une fonctionnalité très intéressante dans le langage C++ est de pouvoir définir une fonction plusieurs fois avec le même nom mais avec des arguments de types différents. Notons bien que seuls les arguments de la fonction peuvent être différents, mais le type de la fonction doit rester identique. Le mini-code illustre ce concept de polymorphisme.

```
#include <iostream>
using namespace std;
void affiche(const int n)
{
   cout<<n<<endl;
}
void affiche(const double *x,const unsigned n)
{
   for(unsigned i=0; i<n; ++i)
   {
      cout<<i<<' '<<x[i]<<endl;
   }
}

void affiche(const double n)
{
   cout<<n<<endl;
}

int main()
{
   double x[7]= {1e0,0.7,0.6,1.0,4.5,8,0.9};
   int a=67;
   double c=0.8;
   affiche(x,7);
   affiche(a);
   affiche(c);
}
```

Dans l'exemple ci-dessus, le compilateur choisit le type de la fonction **affiche** selon les les arguments passés.

4.15 Conclusion

Nous avons vu dans le chapitre l'essentiel (et non pas la totalité) des outils pour commencer à programmer en C++ selon une méthode procédurale. Il reste à utiliser des bibliothèques qui permettront d'utiliser des méthodes numériques. Cela sera vu lors des TP avec une présentation des outils.

CHAPITRE 5

La bibliothèque Armadillo

5.1 Introduction

Comme nous l'avons dans le chapitre précédent, le langage C++ dispose d'un très grand nombre de fonctionnalités dans les possibilités du langage (programmation structurale et objet comme nous le verrons prochainement). La notion de container avec la bibliothèque STL, celle d'itérateurs et la possibilité de surcharger des fonctions ouvrent des possibilités immenses pour écrire des programmes synthétiques. Il y a malheureusement une rançon à cette flexibilité, la syntaxe est souvent lourde et donc difficile a mettre au point. Comme nous le savons la mise au point d'un code est une étape chronophage et cela peut devenir une barrière quasi infranchissable pour l'écriture d'un code sophistiqué. Dans la logique de ce cours et de l'évolution récente des formations de physique numérique, on choisit d'écriture en python quand le temps de calcul est faible car le temps de développement l'est aussi, mais on doit utiliser un langage compilé quand les performances de calcul sont nécessaires. Comme nous l'avons vu en Python, le calcul scientifique dispose de plusieurs bibliothèques (**numpy scipy**, **scikit-learn**,...) qui permettent une bien meilleure efficacité d'exécution en conservant la flexibilité du langage Python. La bibliothèque Armadillo permet de retrouver une syntaxe proche de celle des bibliothèques de Python (ce qui permet de faire une transition plutôt rapide) et le but de ce chapitre est d'illustrer à travers de nombreux exemples le passage du Python au C++ en douceur.

Le développement de cette bibliothèque est en cours et la version actuelle est la version 12.6 (toutefois la version disponible par défaut sur Ubuntu 23.04 est est la version 11.4). De plus, cette bibliothèque est inspiré des logiciels Matlab et Octave. Sachant que **numpy** est aussi inspiré de ces mêmes logiciels, on verra donc rapidement qu'il y a une forte similarité entre les classes d'Armadillo et celle de **numpy**, ce qui donne un grand intérêt à utiliser cette bibliothèque en C++ pour minimiser le temps de développement.

5.2 Algèbre linéaire

La manipulation de vecteurs et de matrices sont au coeur de nombreux programmes scientifiques. Pour définir un tableau avec Armadillo, on dispose de différentes classes ainsi que de nombreuses fonctions membres, ce qui va permettre d'écriture des codes simples proches de la syntaxe déjà vu en Python. De plus dans un second temps, nous verrons que cette bibliothèque dispose de fonctions diverses permettant de résoudre de manière efficace de nombreux problèmes d'algèbre linéaire.

Pour utiliser la bibliothèque Armadillo, il est nécessaire de déclarer le fichier d'entête **<armadillo>** . Il est possible de déclarer un espace de nom **arma** pour utiliser simplement les différentes fonctionnalités de la bibliothèque

5.2.1 Vecteurs

Pour commencer, on s'intéresse aux tableaux à une dimension. On dispose d'un certains nombre de typedef par défaut

vec	=	Col<double>
dvec	=	Col<double>
fvec	=	Col<float>
cx_vec	=	Col<cx_double>
cx_dvec	=	Col<cx_double>
cx_fvec	=	Col<cx_float>
uvec	=	Col<uword>
ivec	=	Col<sword>

pour créer des vecteurs colonnes à une dimension (la classe Col est elle-même héritée de la classe des matrices Mat).

Pour déclarer un tableau double de 10 éléments dont tous les éléments sont initialisés à 0, on peut utiliser

```
#include<iostream>
#include<armadillo>
using namespace std;
using namespace arma;
int main()
{
    vec A=zeros(10);
    cout<<"first version"<<endl;
    cout<< A<<endl;
    cout<<"second version"<<endl;
    for (auto i: A)
        cout <<i<<endl;
    cout<<"third version"<<endl;
    for (int i=0;i<10;i++)
        cout<<A[i]<<endl;

    exit(0);
}
```

Il est bon de noter que l'affichage par défaut permet d'utiliser **cout** avec le nom du tableau (qui a été surchargée dans ce cas). On peut aussi utiliser un itérateur comme cela est donné dans la boucle avec un indice dont le type est déterminé à la compilation. Pour les nostalgiques de l'écriture à l'ancienne, il est aussi possible d'utiliser une syntaxe de type C. C'est aussi l'occasion de vérifier que les éléments du vecteur sont accessibles avec des crochets comme avec un tableau standard en C/C++.

Il est bien sur possible d'initialiser de manière différente, avec des nombres égaux à 1, ou avec des nombres aléatoires.

```
#include<iostream>
#include<armadillo>
using namespace std;
using namespace arma;
int main()
{
    vec A= randu(10);
    cout<< A<<endl;
    vec B= ones(10);
    cout<< B<<endl;
    exit(0);
}
```

Noter que si vous compilez ce programme, la compilation peut échouer à cause de l'utilisation de nombres aléatoires. Comme de nombreuses libraries C++ récentes (CGAL, STL), il n'est pas nécessaire

d'avoir une bibliothèque déjà compilé car le compilateur réalise ce travail si la bibliothèque ne comprend que des fichiers d'entêtes. Pour la bibliothèque Armadillo, ce travail n'est que partiel si vous utilisez une version inférieur à la version 11 sur laquelle ce cours est construit). Dans ce cas, il faut ajouter **-larmadillo** dans l'instruction de l'éditeur de liens. Pour continuer à utiliser Codeblocks, il est nécessaire d'aller **Setting/Compiler/Linker Settings** et d'ajouter dans la fenêtre **Link Libraries** le nom de la bibliothèque **armadillo**.

Par défaut, la fonction **randu** utilise des nombres aléatoires uniformes compris entre 0 et 1. Il est bien sur possible de changer de distribution (voir le site armadillo pour les différentes options). A noter que par défaut la séquence de nombres aléatoires n'est pas changée quand le code est relancé.

Pour avoir une grains différente à chaque exécution, il convient d'ajouter une ligne de code qui se trouve dans le code suivant. A noter que pour une distribution normale, on fait appel à la fonction **randn**

```
#include<iostream>
#include<armadillo>
using namespace std;
using namespace arma;
int main()
{
    arma_rng::set_seed_random();
    vec A= randn(10);
    cout<< A<<endl;
    exit(0);
}
```

Il existe aussi la possibilité de créer des vecteurs lignes avec des définitions similaires en susbtituant **vec** par **rowvec**. Pour des raisons de compatibilité, on peut utiliser aussi **colvec** à la place de **vec**.

5.2.2 Matrices

La deuxième étape consiste à traiter de tableaux à deux dimensions. On dispose d'un certains nombre de typedef par défaut

mat	=	Mat<double>
dmat	=	Mat<double>
fmat	=	Mat<float>
cx_mat	=	Mat<cx_double>
cx_dmat	=	Mat<cx_double>
cx_fmat	=	Mat<cx_float>
umat	=	Mat<uword>
imat	=	Mat<sword>

A noter que sword, uword, cx_double et cx_float sont aussi des typedef pour des entiers signés, non signés, des complexes doubles et réels.

Pour construire une matrice complexe, voici un exemple avec une matrice de Pauli.

```
#include <iostream>
#include<armadillo>

using namespace arma;
int main()
{
    cx_mat sigma1 = { {cx_double(0,0),cx_double(1,0)},{cx_double(1,0),cx_double(0,0)}};
    cout<<sigma1;
    exit(0);
}
```

5.2.3 Cubes

Armadillo propose un troisième container correspondant à un tableau à trois indices. C'est une utilisation moins fréquente que celle des deux précédents types, mais parfois intéressantes. On dispose à nouveau d'une série de typedef où le mot clé est cube. Par exemple cube est un typedef désignant un tableau à trois indices de type double

Le programme suivant illustre le remplissage d'un cube de dimensionx $4 \times 4 \times 4$ avec des nombres aléatoires uniformes entre 0 et 1 avec l'affichage. A noter que les fonctions **fill** ne sont pas toutes implémentées pour les cubes contrairement aux vecteurs et aux matrices. Voir la documentation en ligne pour des informations complémentaires.

```
#include <iostream>
#include<armadillo>

using namespace arma;
int main()
{
    cube AA(4,4,4,fill::randu);
    cout<<AA;
    exit(0);
}
```

5.2.4 Opérations sur les objets d'algèbre linéaire

Une des vertus importantes d'une bibliothèque comme Armadillo est la possibilité de faire des opérations en se passant de l'utilisation de boucles. Outre la simplification de l'écriture du code, c'est aussi une lisibilité renforcée qui apparaît à la lecture. Le prix à payer est de se familiariser avec la syntaxe, mais celle-ci a été inspirée par Matlab, Octave et comme **numpy** de Python. Il est donc relativement aisé de passer du Python à Armadillo.

5.2.5 Opérations simples

Parmi les opérations basiques sur les vecteurs, on peut calculer la norme d'un vecteur, le produit scalaire ou vectoriel (si les vecteurs sont de dimension 3) de deux vecteurs, cela est illustré dans le programme suivant.

```
#include <iostream>
#include<armadillo>

using namespace arma;
int main()
{
    vec AA(3,fill::ones);
    vec BB(3,fill::randu);
    cout<<AA;
    cout<<"␣norme␣de␣AA␣"<<norm(AA)<<endl;
    cout<<BB;
    cout<<"␣norme␣de␣BB␣"<<norm(BB)<<endl;
    cout<<"produit␣scalaire␣de␣AA␣et␣BB␣"<<dot(AA,BB)<<endl;
    cout<<"produit␣vectoriel␣de␣AA␣et␣BB␣"<<endl<<cross(AA,BB);
    exit(0);
}
```

On peut bien sur aussi réaliser la multiplication d'une matrice par un vecteur colonne à droite, la multiplication d'un vecteur ligne par une matrice et la multiplication de deux matrices,...

Le programme suivant illustre ces différentes possibilités. On voit que l'ensemble de ces opérations ne nécessitent aucune boucle explicite et la lecture du code est devenue plus compréhensible, Une situation analogue à celle que l'on connaît en python !

```
#include <iostream>
#include<armadillo>

using namespace arma;
int main()
{
    colvec AA(3,fill::randu);
    rowvec CC=AA.t();
    mat BB(3,3,fill::randu);
    cout<<AA;
    cout<<BB;
    cout<<"␣BB*AA␣"<<endl<<BB*AA<<endl;
    cout<<"CC"<<CC;
    cout<<"␣CC*BB␣"<<endl<<CC*BB<<endl;
    cout<< "BB*BB"<<BB*BB<<endl;
    exit(0);
}
```

5.2.6 Opérations complexes

La détermination de quantités comme les valeurs propres d'une matrice ou celle d'un déterminant est bien évidemment très utile dans le calcul numérique, mais peut s'avérer difficile à obtenir sans l'usage d'une bibliothèque bien construite. Je renvoie le lecteur au polycopié sur les méthodes numériques pour avoir plus d'informations sur les différentes méthodes utilisables en fonction des propriétés des matrices. Armadillo a implémenté de nombreuses méthodes pour obtenir des résultats numériques précis. Nous allons simplement illustrer quelques unes de ces possibilités.

Dans le programme suivant, on cherche à déterminer les valeurs propres d'une matrice symétrique tridiagonale dont la diagonale est remplie par des valeurs égales à -2 et les deux autres diagonales adjacentes sont remplies par des valeurs égales à 1.

Voici un programme très court qui utilise les fonctionnalités d'Armadillo à la fois pour la construction de la matrice, ensuite la fonction diagmat permettant de remplir les diagonales de la matrice, puis la fonction **eigval** pour obtenir à la fois le déterminant de la matrice ainsi que les valeurs propres (qui sont bien entendus réelles) ainsi que la fonction **sort** qui permet de classer les valeurs propres. Il est aisé de vérifier que le déterminant de cette matrice peut être obtenu facilement et vaut $n+1$ où n est la dimension linéaire de la matrice.

```
#include <iostream>
#include<armadillo>

using namespace arma;
int main()
{ int n=100;
    vec AA(n,fill::ones);
    vec AA2(n-1,fill::ones);
    mat BB(n,n);
    BB=2*diagmat(AA)-diagmat(AA2,1)-diagmat(AA2,-1);
    //cout<<BB;
    vec eigval;
    eig_sym(eigval, BB );
    cout<<"determinant␣de␣BB␣"<<det(BB)<<endl;
```

```
    cout<<sort(eigval)<<endl;
    exit(0);
}
```

5.3 Conclusion

Ce chapitre n'est évidemment qu'une introduction aux nombreuses possibilités données par Armadillo. Il est conseillé d'aller consulter la documentation en ligne pour l'utilisation des autres fonctions. L'utilisation du langage C++ est grandement simplifiée par ce genre de bibliothèques et devient moins rugueuse qu'avec les éléments de base du C++. Pour disposer de l'efficacité maximale du C++, il convient de compiler les programmes avec les options d'optimisation du compilateur. L'utilisation de la bibliothèque **openmp** peut être utilisée avec Armadillo et permet l'utilisation de plusieurs cœurs du processeur de votre ordinateur est aussi un point important de cette bibliothèque même si cela dépasse un peu le cadre de ce cours d'introduction.

Programmation orientée objet

La notion de programmation orientée objet repose sur quelques concepts : quand on crée une structure qui regroupe des données ayant un lien entre elles, on sait que l'on a un nombre fini de transformations qui s'appliquent à ces données et il est plus judicieux d'ajouter les fonctions qui vont transformer ces données à l'intérieur de cette structure plutôt que de les séparer et de les rendre potentiellement disponibles à tout le reste du code sans que cela présente une quelconque utilité.

Le second concept est la notion d'encapsulation qui repose sur cette idée de sécurité. On fixe dans la nouvelle structure, que l'on appellera classe, l'autorisation et/ou le refus de lecture ou d'écriture des différentes attributs de la classe aux autres parties du programme. Les classes deviennent en quelque sorte des éléments autonomes et il faut définir comment les échanges entre classes peuvent se faire.

Le troisième concept est celui d'héritage. Quand on a crée une classe, on peut créer une nouvelle classe qui va recycler les éléments de la classe précédente, et en ajouter de nouveaux. Il est possible alors de créer un système de poupées russes où les classes s'emboîtent les unes dans les autres.

On peut bien entendu utiliser les deux manières de programmer. Par exemple quand on assigne une variable, on a vu avec la commande **dir** en Python que cette variable disposait alors d'un nombre significatif de méthodes. De même en C++, l'utilisation de la bibliothèque **vector** nous a montré une fois créé qu'un vecteur dispose de méthodes permettant d'agir sur ces données. Ainsi, sans avoir écrit la moindre classe, il est possible de faire de la programmation objet. Bien évidemment, pour aller plus loin, il est nécessaire de connaître les rudiments permettant de créer des classes dans chacun de ces deux langages afin de mieux maîtriser les concepts énoncés précédemment et de penser progressivement différemment pour l'écriture d'un code. Nous verrons à cette occasion que les règles sont parfois différentes entre le Python et le C++. Concernant le C++, on va aussi présenter la notion de template qui est en quelque sorte la généralisation du polymorphisme des fonctions appliquée à celle des classes.

CHAPITRE 6

Le langage Python : programmation orientée objet

6.1 Introduction

Dans le premier chapitre consacré à Python, nous avons vu (ou revu) la syntaxe pour développer un code en Python en créant des fonctions pour transformer les données d'entrée en données de sortie. L'élément essentiel était la notion de fonction. La classe permet de combiner les deux à l'intérieur d'une même "boîte" : les données sont appelées les attributs et les fonctions les membres Nous allons construire une première classe que l'on appelle rectangle.

```
 #!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Jul 7 09:29:06 2020

@author: viot
"""

class Rectangle:
    longueur=10
    largeur=8;

rect=Rectangle()
print(rect.longueur,rect.largeur)
print("type", type(rect))
rect.largeur=6
rect.longueur=8
print(rect.longueur,rect.largeur)
rect.largeur=6
rect.longueur="inconnue"
print(rect.longueur,rect.largeur)
rect2=Rectangle()
print(rect2.longueur,rect2.largeur)
```

Le minicode définit une classe de la manière suivante : tout d'abord le mot clé suivi du nom de la classe ainsi que du caractère **:** analogue à ce que l'on a déjà pour la création de fonction. Les deux lignes suivantes définissent deux attributs dont les noms parlent d'eux-mêmes. Ils sont initialisés avec une valeur par défaut. Pour créer un objet de cette classe Rectangle, on fait un appel à la classe et on fait afficher les valeurs des deux attributs. On vérifie, avec la dernière instruction que l'on a bien créé un objet de la classe Rectangle. On peut ensuite changer les valeurs des attributs dans le programme principal. On peut le modifier en changeant les attributs de rect par une nouvelle assignation en changeant le type de l'attribut.

On voit bien que l'on a modifié les attributs de rect qui sont accessibles depuis le programme principal. Le Python étant un langage dynamique, on peut même se permettre de changer le type des attributs : dans le code précédent, on change la longueur en une chaîne de caractères. A ce stade, la classe n'a que deux attributs et pas de méthodes, mais c'est le début. Si on crée un second objet de la **class** Rectangle (**rect2**) il possède par défaut deux attributs qui sont initialisés par défaut aux valeurs données dans la classe.

6.2 Classe en Python

Pour aller plus loin dans la notion de classe, il faut pouvoir définir un ou plusieurs constructeurs et par la suite des fonctions membres de la classe. Pour créer un constructeur de classe. on définit d'une méthode que l'on appelle **__init__** et il possède au minimum l'argument self, puis la liste des variables d'instance (ou attributs de la classe). En choisissant d'initialiser par défaut les deux variables, cela permet d'avoir un nombre variable de paramètres au moment de la création comme l'illustre le code suivant.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Jul 7 09:29:06 2020

@author: viot
"""

class Rectangle:
    def __init__(self,largeur=10,longueur=8):
        self.largeur=largeur
        self.longueur=longueur

rect=Rectangle()
print(rect.longueur,rect.largeur)
rect2=Rectangle(4,6)
print(rect2.longueur,rect2.largeur)
```

Dans le programme principal, l'objet **rect** est crée en utilisant les valeurs de longueur et largeur par défaut. Par contre, pour l'objet **rect2**, les deux arguments données lors de la création permettent d'initialiser les valeurs de largeur et de longueur avec les valeurs 4 et 6 au lieu des valeurs par défaut. On a donc maintenant une plus grande flexibilité lors de la création des objets.

On va maintenant enrichir la classe précédente en ajoutant une méthode permettant de calculer l'aire d'un rectangle.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Jul 7 09:29:06 2020

@author: viot
"""

class Rectangle:
    def __init__(self,largeur=10,longueur=8):
        self.largeur=largeur
        self.longueur=longueur
    def aire(self):
        return(self.largeur*self.longueur)

rect=Rectangle(4,6)
print(rect.longueur,rect.largeur,rect.aire())
```

La fonction membre **aire** ne fait appel qu'aux variables d'instance, ce qui explique que l'argument de cette fonction ne contient que l'argument self. Dans l'appel de cette fonction membre **aire**, on voit **aire** est suivie d'une paire de parenthèses vides.

Il est facile d'ajouter une seconde méthode permettant par exemple de calculer le périmètre. Si à nouveau, vous tapez dans la console IPython l'instruction **dir(rect)**, vous verrez que la classe possède

une nouvelle fonction membre aire.

6.3 Héritage

L'héritage est une fonctionnalité essentielle de la programmation puisqu'elle permet de recycler les membres et les attributs d'une classe précédemment construite pour en définir une nouvelle. En gardant l'exemple précédent, nous allons construire une classe associée au parallélépipède. La projection de ce dernier par rapport à la hauteur représente un rectangle.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Jul 7 09:29:06 2020

@author: viot
"""

class Rectangle:
    def __init__(self,largeur=4,longueur=6):
        self.largeur=largeur
        self.longueur=longueur
    def aire(self):
        return(self.largeur*self.longueur)

class Paral(Rectangle):
    def __init__(self,largeur=2,longueur=5,hauteur=7):
        Rectangle.__init__(self,largeur,longueur)
        self.hauteur=hauteur
    def volume(self):
        return(Rectangle.aire(self)*self.hauteur)


para=Paral()
print("ex1 ",para.largeur,para.longueur,para.hauteur)
para=Paral(4)
print("ex2",para.largeur,para.longueur,para.hauteur)
print("Vol ex2",para.volume())
para=Paral(hauteur=6,longueur=7)
print("ex4",para.volume())
print(para.largeur,para.longueur,para.hauteur)
para=Paral(2,8,10)
print(para.largeur,para.longueur,para.hauteur)
print(para.volume())
```

Les première lignes correspondent à la définition de la classe Rectangle. Pour définir une classe dérivée, on place dans cette nouvelle classe **Paral** la classe parente **Rectangle**. Le constructeur de la classe Paral utilise le constructeur de la classe Rectangle pour initialiser les attributs largeur et longueur, puis il initialise l'attribut hauteur. Le membre volume calcule le volume du parallélépipède en utilisant les trois attributs de la classe Paral

Avec cette écriture, il est possible de créer des instances (objets) en passant un nombre variable de paramètres allant de 0 à 3. Cela est illustré dans les lignes qui suivent : en définissant une instance avec un nombre variable de paramètres On voit que l'ordre choisi pour assigner les attributs de l'objet correspond à la définition des différents attributs de la classe.

Comme on récupère l'ensemble des éléments de la classe Rectangle pour la classe Paral, on récupère aussi le membre **aire**. Si on exécute **para.aire()**, le résultat est incorrect pour un parallélépipède car

l'aire ne correspond pas à la définition de cette classe. Il est possible de redéfinir une fonction **aire** à l'intérieur de la classe Paral et donc d'éliminer l'ancienne définition. Selon le type d'objet, **rectangle** ou **parallélépipède**, l'interpréteur Python choisira la fonction membre correspondant à la classe et le bug précédent disparaît.

Toujours sur la base que Python est un langage dynamique, il est possible d'ajouter un attribut à un objet créé par une classe sans que cet attribut existe dans cette classe. Par exemple

```
para.comment=Parallelepipede prefere"
```

ajoute un attribut à l'objet para. C'est un moyen très rapide de faire des classes dérivées pour des instances.

La combinaison de classes ne se limite ni à une seule génération ni à une seule classe. On peut construire une classe dérivée impliquant plusieurs classes. Nous verrons quelques aspects supplémentaires lors de séances de travaux pratiques. Cette rapide introduction permet de mieux saisir la structure des modules qui repose sur ces concepts de programmation objet.

6.4 Code avec plusieurs fichiers

Dès qu'un code dépasse une centaine de lignes, il est impossible de visualiser l'ensemble du code sur un écran. Il est alors souhaitable de diviser celui-ci en plusieurs parties. Sur l'exemple précédent, il existe une manière naturelle pour réaliser cette division : la définition des classes constitue un ensemble d'outils qui va être utilisé par le programme principal. En enregistrant dans le répertoire courant cette partie du code que l'on nomme **mabib.py**, on peut ensuite utiliser ce code dans le programme principal en utilisant l'instruction import mabib et modifier la première ligne du code pour spécifier que la classe Paral se trouve dans l'espace de noms **mabib**. Cela donne

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Jul 7 09:29:06 2020

@author: viot
"""
import mabib

para=mabib.Paral(2,8,10)
print("premier␣objet",para.largeur,para.longueur,para.hauteur)
print(para.volume(),para.aire())
para.couleur="red"
```

Le module ainsi créé ressemble à un module de Python standard (avec des fonctionnalités fort réduites, je vous l'accorde). Toutefois pour être plus acceptable, il faut le commenter pour avoir des infos sur son utilisation (tant qu'il apparaissait sur l'éditeur et que le code était court, c'était moins crucial).

Pour commenter une fonction ou une classe, on utilise un triple " généralement placé après la définition. On décrit en quelques lignes les fonctionnalités et l'usage, puis on referme le commentaire par un triple ". Pour le fichier mabib.py, cela donne par exemple

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Jul 7 15:30:24 2020

@author: viot
"""

class Rectangle:
    """ class Rectangle
    -attributs: largeur et longeur
    -membres: aire
    """
    def __init__(self,largeur=8,longueur=10):
        self.largeur=largeur
        self.longueur=longueur
    def aire(self):
        return(self.largeur*self.longueur)

class Paral(Rectangle):
    """ class Paral
    -attributs: largeur,longueur et hauteur
    -membres: volume et aire
    """
    def __init__(self,largeur=8,longueur=10,hauteur=8):
        Rectangle.__init__(self,largeur,longueur)
        self.hauteur=hauteur
    def volume(self):
        return(Rectangle.aire(self)*self.hauteur)
    def aire(self):
        return(2*(Rectangle.aire(self)+ self.largeur*self.hauteur
                +self.longueur*self.hauteur))
```

Si dans la console IPython, on tape tout d'abord l'instruction **import mabib**, suivi de **dir(mabib)**, on voit apparaître les deux classes de ce petit module. Pour en savoir plus, on peut ensuite taper **help(Paral)** et on voit apparaître les commentaires.

Bien entendu, quand le code devient plus complexe, on peut séparer le programme en plusieurs fichiers.

Nous avons vu les éléments principaux de la programmation objet sous Python. Le but de ce chapitre est à la fois de pouvoir construire quelques classes simples, mais aussi de mieux appréhender la structure des modules très nombreux dans Python qui repose sur ces idées fondamentales que nous avons présenté lors de ce cours. L'utilisation de classes est plus facile que leur fabrication.

CHAPITRE 7

Le langage C++ : programmation orientée objet

7.1 Introduction

Le C++ qui a été initialement créé sur la base d'être une extension du langage C. Il a depuis développé de nombreux concepts qui se retrouvent donc dans d'autres langages comme nous l'avons vu avec le langage Python dans le chapitre précédent. Loin d'être figé, le C++ continue d'évoluer pour introduire de nouveaux éléments au fil des révisions qui interviennent à un rythme soutenu depuis la dernière décennie. La périodicité de ces changements est actuellement de trois ans. Une fois les nouveautés validées, il est nécessaire que cela soit pris en compte par les compilateurs tels que le compilateur Gnu (qui suit aussi une accélération dans ses révisions). Le C++17 est aujourd'hui pris en compte par les compilateurs Gnu et Intel, et partiellement par Clang. Dans ce cours d'introduction au C++ sur la programmation, nous n'allons pas aller jusqu'à considérer les dernières nouveautés. Au mieux, quelques concepts du C++14 peuvent être utiles pour se simplifier la vie. Une première version de la programmation objet a été vue à travers le langage Python, et nous allons voir comment cela se transpose dans le cas du C++. Ultimement, l'essentiel est de comprendre les concepts et surtout permettre d'utiliser plus simplement des bibliothèques dans lesquels ces développements ont été introduits de manière intensive.

Le C++ étant un langage compilé, le typage des classes et des objets va constituer un frein par rapport au Python. Nous allons voir qu'avec le polymorphisme et la notion de template, cela introduit beaucoup de flexibilité dans l'écriture d'un programme. En résumé, à travers des exemples simples, nous allons introduire les concepts et la syntaxe spécifique du C++ afin de pouvoir faire des classes personnelles. La création de bibliothèques qui utilisent plus massivement l'ensemble des fonctionnalités du C++ va au delà de ce cours. L'ojectif principal est donc l'utilisation des bibliothèques, qui peut être plus simple dans la mesure où il s'agit d'apprendre à maîtriser un outil et non pas de le concevoir.

7.2 Structure ou classe

7.2.1 Structure

Nous avons vu la notion de structure qui était directement hérité du C et qui permettait de placer d'un ensemble de variables de types différents au sein d'une structure. La notion de classe repose sur l'idée de placer à l'intérieur de la même structure les variables que l'on appellera attributs avec les fonctions qui peuvent modifier les attributs et que l'on appellera fonctions membres. C'est à ce point la même idée que l'on a vue avec le langage Python. Le C++ va plus loin dans le sens où il permet de choisir le niveau d'accessibilité de chaque élément de classe pour le reste du programme (en particulier les classes dérivées que nous verrons au moment de l'héritage).

Cette possibilité accroît de beaucoup la sécurité des programmes (les bugs liés à l'interaction non souhaitée entre différentes parties d'un code sont un fléau quand le projet comprend de nombreux fichiers et de nombreux développeurs), mais peut devenir un frein, car cela alourdit le développement dans tous les cas. Cela peut apparaître compliqué quand la longueur du code n'est que de l'ordre de quelques centaines de lignes.

Une raison majeure en faveur de l'apprentissage de ce type de programmation est aussi liée aux bibliothèques nombreuses qui se développent en C++ qui utilisent les concepts récents de programmation. On peut avoir à créer un code court (toujours quelques dizaines de lignes) qui fait appel à des biblio-

thèques qui utilisent très efficacement le processeur et aussi le processeur graphique qui est devenu ces dernières années un concurrent très sérieux du premier.

Si on veut faire une classe qui n'utilise pas les possibilités de sécurisation du C++, il suffit d'utiliser le mot clé struct qui laisse par défaut l'accès public à tous les membres de la classe.

Nous allons reprendre le même programme de la classe rectangle que nous avions utilisé dans le chapitre précédent pour Python.

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


typedef struct{
double largeur, longueur;
double aire(void) {return(largeur*longueur);};
} Rectangle;

int main()
{
Rectangle rect;
rect.largeur=8;
rect.longueur=10;
cout<<"aire␣du␣rectangle␣"<<rect.aire()<<endl;
}
```

Comme prévu, cette façon de créer une classe fonctionne, mais c'est très laid ! Il a fallu créer un objet que l'on a initialisé à l'extérieur de la classe. Le point illustrant le concept de classe est l'utilisation de la fonction membre aire. Comme elle utilise uniquement les attributs de la classe, il n'y a aucun argument à transmettre pour la fonction aire, elle a un argument vide dans la définition.

7.2.2 Classe

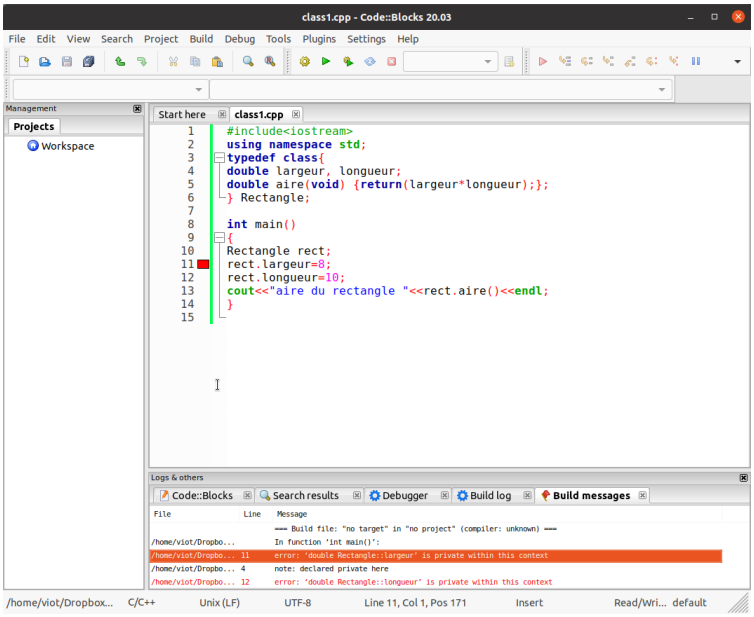


FIGURE 7.1 – La classe où tout est privé !

Nous allons maintenant remplacer le mot clé **struct** par le mot **class**. Il est donc temps de préciser les

différents niveaux d'accès des éléments de la classe : les permissions sont de trois types **p**ublic, **p**rotected ou **p**rivate. Les membres sont les fonctions membres ou les attributs de la classe. Les membres privés ne sont accessibles que par les autres membres de la classe. Les membres **p**rotected ne sont accessibles que par les membres de la classe et des classes dérivées. Les membres publics sont accessibles de partout où la classe est visible. Si aucune directive n'est précisée, tout est public avec **struct** et tout est privé avec **class**.

La figure 7.1 illustre sous Codeblocks le résultat de l'exécution, quand on est passé avec une classe où tout est privé. (A noter que la compilation s'est bien passée.)

Codeblocks indique que l'instruction de la ligne 11 ne peut pas être exécutée et dans la fenêtre du bas, la raison de cet échec est fournie ! Cela montre au passage l'intérêt des interfaces graphiques de développement clair pour identifier rapidement les problèmes d'un code (dans ce cas, il s'agit de l'exécution).

Finalement en choisissant le mot clé **class**, on peut créer un objet (ligne 10), mais on ne peut rien faire avec lui car tout est inaccessible, les attributs comme les fonctions membres. L'objet est bien protégé, mais c'est une version paranoïaque ! Un code très sécurisé et parfaitement inutile !

Il faut donc déterminer pour chaque membre le type d'accessibilité que l'on souhaite avoir. Ce n'est pas toujours évident à déterminer. Pour se donner des règles simples, nous allons nous aider en prenant une comparaison anthropomorphique : si l'on compare l'objet d'une classe à un être humain, on va dire que la partie privée correspond à des éléments personnels (donc une partie des attributs), que tout ce qui concerne les échanges avec des proches relève de la partie **protected** et que le reste relève de ce que représente notre relation avec l'extérieur et ce sera **public**.

Sur cette définition, on va dire que les attributs du rectangle doivent être privés et que la fonction membre est publique.

7.2.3 Constructeur de classe

Il reste maintenant à initialiser l'objet. Comme toujours avec les langages, on a plusieurs possibilités. On choisit une méthode qui va réaliser cela. Quand on crée un classe, celle-ci doit contenir par défaut un constructeur qui permet de créer l'objet, ce qui explique que l'exécution s'est bien passé au début, et que l'erreur apparaît quand on souhaite modifier les valeurs des attributs. Si on veut que le constructeur puisse aussi initialiser avec des valeurs données à la création de l'objet, il faut déclarer un constructeur spécifique. Pour une classe, on peut créer autant de constructeurs que l'on souhaite, et quand le nombre d'arguments change avec le constructeur, le programme choisira le bon à l'exécution. Pour définir un constructeur, il faut déjà le placer dans la rubrique public (sinon aucune chance de pouvoir l'utiliser à partir d'une autre partie du code).

Pour définir le constructeur, les règles d'écriture sont les suivantes

- Le constructeur n'a pas de type (même pas void).
- Il a le nom de la classe.
- Chacun de ses arguments a un type. Les instructions correspondent à sa tâche et sont données entre accolades. Il peut avoir une initialisation par défaut qui est donnée sur chaque argument
- On peut recopier les arguments du constructeur dans une ligne commençant par **:** et en plaçant chaque variable attribut de la classe suivi de l'argument utilisé placé entre parenthèses

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

class Rectangle
{
   double largeur, longueur;
public:
   Rectangle(double x, double y)
   {
      largeur=x;
      longueur=y;
   }
   double aire(void)
   {
      return(largeur*longueur);
   };
} ;
int main()
{
   Rectangle rect(8.,7);
   //Rectangle rect();
   cout<<rect.largeur<<endl;
   cout<<"aire du rectangle "<<rect.aire()<<endl;
}
```

On a donc maintenant créé une classe où les deux attributs sont des variables privées (si on décommente la ligne 16, le compilateur donne une erreur en indiquant que la variable est privée et donc n'est pas visible dans le programme principal).

On aimerait bien toutefois avoir la même possibilité qu'avec Python, c'est-à-dire pouvoir initialiser avec des valeurs par défaut si l'objet est créé sans argument.

Dans la version suivante, on utilise la règle 4 de la création du constructeur pour nous permettre d'initialiser un objet avec un nombre variable d'arguments.

De fait, on a aussi un bonus car notre nouveau constructeur permet la recopie d'objet et donc crée une constructeur de copie modifié par rapport au constructeur de copie par défaut qui existait.

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

class Rectangle
{
   double largeur, longueur;
public:
   Rectangle(double x=6., double y=8.): largeur(x), longueur(y) {};
   double aire(void)
   {
      return(largeur*longueur);
   };
} ;

int main()
{
   Rectangle rect;
   cout<<"aire du rectangle "<<rect.aire()<<endl;
   Rectangle rect3(4.);
   cout<<"aire du rectangle "<<rect3.aire()<<endl;
   Rectangle rect2(8.,10.);
   cout<<"aire du rectangle "<<rect2.aire()<<endl;
   Rectangle rect1=rect2;
   cout<<"aire du rectangle "<<rect1.aire()<<endl;
}
```

On a maintenant un constructeur qui permet d'initialiser avec un nombre variable d'arguments. On a aussi un constructeur de recopie permettant que l'instruction de la ligne 20 crée une parfaite copie de **rect2** dans **rect1**. Nous verrons en exercice les variantes de cette méthode qui permettent d'obtenir un résultat identique. Cette méthode a le mérite d'être synthétique et donner des possibilités analogues à celles que nous avions en Python.

7.2.4 Destructeur de classe et objet dynamique

Si le C++ crée un constructeur de classe que l'on peut enrichir par un ou plusieurs autres (comme nous l'avons vu à la section précédente), il crée aussi un destructeur de classe par défaut. Contrairement aux constructeurs qui peuvent être multiples pour une classe, on ne peut avoir qu'un destructeur par classe. Cela signifie que si on crée un destructeur, il remplace le destructeur créé par le compilateur par défaut. En général, ce n'est pas absolument nécessaire et très souvent les classes n'ont pas de destructeur explicitement donné.

Outre le fait qu'il faut savoir que ce destructeur existe, cela peut être utile d'en faire un pour mettre au point un code. Comme le destructeur sera utilisé quand l'objet est détruit, nous allons créer des objets dynamiques pour les détruire à la fin du programme.

Dans le code suivant on a ajouté un destructeur de classe dont les règles d'écriture sont plus simples que pour les constructeurs

- Le destructeur n'a pas de type (même pas void)
- Il a le nom de la classe précédé du signe ~
- Il n'a pas d'argument et le code exécuté avant la destruction est placé entre accolades.

Dans le minicode suivant, on crée dynamiquement un objet de classe Rectangle. La syntaxe est similaire à celle d'un tableau dynamique. On définit tout d'abord un pointeur associé à la classe puis une allocation dynamique d'un objet avec l'opérateur **new**.

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

class Rectangle
{
    double largeur, longueur;
public:
    Rectangle(double x=6., double y=8.): largeur(x), longueur(y) {};
    ~Rectangle()
    {
        cout<<"Salut, je suis le rectangle "<<largeur<<" x "<<longueur<<" et
            ↪ je me casse!"<<endl;
    }
    double aire(void)
    {
        return(largeur*longueur);
    };
} ;

int main()
{

    Rectangle *rect=new Rectangle(3,4);
    cout<<"aire du rectangle "<<rect->aire()<<endl;
    delete rect;;
    exit(0);
}
```

Cela donne le résultat suivant. On voit donc qu'avec le destructeur personnalisé, on a obtenu des informations supplémentaires juste avant la destruction.

```
aire du rectangle 12
Salut, je suis le rectangle 3 x 4 et je me casse!
```

7.2.5 Différentes écritures d'une classe

Contrairement aux exemples simples que nous utilisons pour illustrer les différents concepts, il est souvent intéressant de ne pas écrire la totalité du code définissant une classe à l'intérieur de celle-ci. Il suffit de garder, à l'intérieur de la classe, un prototype de chacune des fonctions membres. Les définitions de ces fonctions sont alors données extérieurement, mais pour signifier qu'elles appartiennent à une classe précise, on donne le nom de la classe suivi de **::** et du nom de la fonction membre. Dans le code suivant, on fait sortir de la classe la fonction membre aire.

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

 class Rectangle {
double largeur, longueur;
public:
Rectangle(double x=6., double y=8.): largeur(x), longueur(y) {};
double aire(void);
} ;

double Rectangle::aire(void) {return(largeur*longueur);};

int main()
{

Rectangle *rect=new Rectangle(3,4);
cout<<"aire␣du␣rectangle␣"<<rect->aire()<<endl;
delete rect;;
exit(0);}
```

Il est possible aussi de sortir de la classe la définition du ou des constructeurs de la classe, ainsi que les fonctions membres. L'exemple suivant illustre ce point ainsi que la présence de deux constructeurs de classe

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

 class Rectangle {
double largeur, longueur;
public:
Rectangle();
Rectangle(double,double);
double aire(void);
} ;

double Rectangle::aire(void) {return(largeur*longueur);};
Rectangle::Rectangle(){
   largeur=6;longueur=8;}
Rectangle::Rectangle(double x, double y)
{
   largeur=x;longueur=y;}

int main()
{

Rectangle rect;
cout<<"aire␣du␣rectangle␣"<<rect.aire()<<endl;

Rectangle rect2(3,4);
cout<<"aire␣du␣rectangle␣"<<rect2.aire()<<endl;

exit(0);}
```

7.3 Héritage

Une des propriétés intéressantes de la programmation orientée et donc du C++ est la notion d'héritage : elle permet de construire une classe à partir d'une classe déjà existante en ajoutant, soit des fonctions membres soit des attributs, soit les deux. Nous allons créer une seconde classe qui permet de définir un parallélépipède rectangle et dont on peut calculer le volume

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

class Rectangle
{
protected:
   int x, y;

public:

   Rectangle (int a=0,int b=0): x(a), y(b) {};
// ~Rectangle(){};
   int aire (void)
   {
      return (x*y);
   }
};

class Paralle_de : public Rectangle
{
protected:
   int z;
public:
   Paralle_de(int a=0, int b=0, int c=0): Rectangle(a,b), z(c) {};
// ~Rect_Complet(){}
   int volume (void)
   {
      return (aire()*z);
   }
};


int main ()
{
   Paralle_de Para;
   cout <<"volume␣"<<Para.volume()<<endl;
   Paralle_de Para2(3,4,5);
   cout <<"volume␣"<<Para2.volume()<<endl;
   Paralle_de Para3=Para2;
   cout <<"volume␣"<<Para3.volume()<<endl;
   return 0;
}
```

La classe **Paralle_de** est construite à partir de la classe **Rectangle** en introduisant un troisième attribut pour définir la hauteur après avoir défini le rectangle avec une largeur et une longueur. Le constructeur utilise une initialisation par défaut et la copie de classe en combinant celle de rectangle et celle de l'argument z. La fonction membre de la classe **Paralle_de** , **volume**, est construite à partir de la fonction membre de la classe **Rectangle**, **aire**. La fonction membre de Paralle_de hérite de la fonction membre de Rectangle.

Dans le programme principal, le premier objet donne un volume nul car ses paramètres sont nuls par défaut. Les objets suivants donnent la valeur 60 pour le volume produit des trois entiers. Le troisième élément est simplement la copie de l'objet numéro 2.

Pour être complet, notons que le programme principal peut utiliser des objets créés dynamiquement. La syntaxe est illustrée dans le programme vu plus haut où seul le programme principal avait été modifié, les autres lignes du programme n'étant pas modifiées.

7.4 Template

La notion de template, dont la traduction française est modèle permet de créer des fonctions ou des classes qui peuvent utilisées pour différents types. Cette notion est à la fois au cœur de la STL ainsi que de Boost ou Eigen. Cela permet de construire des codes plus simples. De plus, ces bibliothèques sont maintenant construites uniquement sur des fichiers d'en-tête. On crée alors une bibliothèque spécifique en fonction de la manière dont le programme utilise ces fonctionnalités. Cela induit un temps plus long de compilation, mais la (ou les) bibliothèque(s) utilisée(s) sont alors indépendante(s) de la plateforme informatique. Pour mieux cerner ce concept, nous allons créer un template de fonction très simple.

```
#include <iostream>
using namespace std;
template <typename T>
T premier(T a, T b)
{ return (a);
}
int main()
{
   double a=2.5;
   double b=3.0;
   int c=4;
   int d=5;
   cout<<premier(a,b)<<"␣"<<endl;
   cout<<premier(c,d)<<"␣"<<endl;
   exit(0);
}
```

La syntaxe pour le template est la suivante : la première ligne commence par le mot clé **template** suivi de **typename T** placé entre des chevrons. Cela signifie que l'on crée un modèle avec un type générique appelé T. A la la ligne suivante (sans qu'il y ait de séparateur ;) on définit le type de la fonction (dans l'exemple donné, ce type est le type générique T), puis les arguments de la fonction sont donné s avec chacun de leurs types (dans l'exemple, tous les arguments ont le même type générique T). Comme pour toute fonction, on retrouve l'instruction de retour **return**. Dans le programme principal, on fait appel à la fonction **premier** avec des arguments qui sont, soit réels, soit entiers. Le compilateur a préparé les deux types de fonctions dont il y a besoin pour exécuter le programme.

Une variante du programme précédent consiste à modifier les arguments de la fonction en n'utilisant pas le passage par valeur (qui consiste à recopier la valeur de la variable transmise dans une autre emplacement mémoire), mais un passage par référence. Pour signifier au compilateur que les valeurs des variables transmises ne doivent pas être modifiées à l'intérieur de la fonction, le mot clé **const** assure que le compilateur vérifiera que les arguments ne sont pas modifiés. Cela donne

```
#include <iostream>
using namespace std;
template <typename T>
T premier(const T& a,const T& b)
{ return (a);
}
int main()
{
   double a=2.5;
   double b=1.0;
   int c=4;
   int d=5;
   cout<<premier(a,b)<<"␣"<<endl;
   cout<<premier(c,d)<<"␣"<<endl;
   exit(0);
}
```

On peut poursuivre la construction de **template** avec plusieurs types génériques. Dans l'exemple suivant, on généralise la fonction précédente en utilisant deux types génériques

```
#include <iostream>
using namespace std;
template <typename T, typename U>
T premier(const T& a,const U& b)
{ return (a);
}
int main()
{
   double a=2.5;
   int b=5;
   cout<<premier(a,b)<<"␣"<<endl;
   cout<<premier(b,a)<<"␣"<<endl;
   exit(0);
}
```

Le compilateur prépare donc deux fonctions : La première fonction est réelle avec un premier argument réel et un second entier. La seconde fonction est entière avec un premier argument entier et le second réel.

7.5 Spécialisation

Après avoir créé des fonctions génériques, on peut affiner la notion de template en ajoutant celle de spécialisation. Cela permet, pour un type donné, que la fonction ait un comportement différent. Pour cela, l'écriture est la suivante : on commence par le mot clé template avec cette fois un groupe de chevrons vide. Les arguments de la fonction sont alors typés. Dans l'exemple suivant, on enrichit la fonction précédente dans le cas où les arguments sont des chaînes de caractères.

```
#include <iostream>
using namespace std;
template <typename T, typename U>
T premier(const T& a,const U& b)
{ return (a);
}
template <>
string premier (const string &a, const string &b)
{return (a+"␣devant"+b);}
int main()
{
   double a=2.5;
   int b=5;
   cout<<premier(a,b)<<"␣"<<endl;
   cout<<premier(b,a)<<"␣"<<endl;
   string c="bonjour";
   string d="au␣revoir␣";
   cout<<premier(c,d)<<"␣"<<endl;
   cout<<premier(c,b)<<"␣"<<endl;
   exit(0);
}
```

Il reste à organiser ce type de code au sein d'un programme beaucoup plus consistant. Quand on crée une fonction template avec son lot de fonctions spécialisées, elles doivent être placées dans un fichier en-tête *.h. Dans l'exemple ci-dessus, on a placé cette fonction en début de code d'un programme qui ne contenait qu'un seul fichier.

7.6 Bibliothèques standards

Nous avons déjà vu l'utilisation de classes sans le mentionner avec les vecteurs de la bibliothèques STL.

7.6.1 STL : :vector

Nous avons vu son aspect pratique, car on peut choisir le type du vecteur que l'on souhaite utiliser. On crée alors un vecteur dont le nombre d'éléments peut changer au cours de l'exécution. Avec la classe **vector**, il existe un grand nombre de fonctions membres qui permettent une nouvelle écriture d'un code. Dans l'exemple suivant, on crée un objet vecteur de double que l'on remplit au départ avec 3 éléments que l'on peut faire afficher en utilisant la fonction membre **size** qui permet de connaître la taille du vecteur, puis ensuite on continue de remplir avec de nouveaux éléments et on fait afficher à nouveau les différents éléments de ce vecteur.

On a donc cette possibilité de changer simplement la taille d'un vecteur au cours de l'exécution.

On peut aussi insérer un nouvel élément à un emplacement choisi. Dans l'exemple suivant, on insère un nouvel élément comme premier élément du vecteur (alors que les trois premières instructions ont initialisé les trois premiers éléments avec la méthode **push_back**). La méthode **insert** demande la position comme premier argument à laquelle on introduit un nouvel élément. Dans le cas présent on souhaite insérer comme premier élément, et en utilisant la fonction membre **vec.begin()**, on obtient le résultat souhaité.

```
#include <iostream>
#include<vector>
using namespace std;
int main(){
    vector <double> vec;
    vec.push_back(2.0);
    vec.push_back(3.0);
    vec.push_back(4.0);

    vec.insert(vec.begin(),5.0);

    for(int i=0;i<vec.size();i++)
        {cout<<vec[i]<<endl;}
    exit(0);
}
```

7.7 STL : : iterateur

Dans l'exemple précédent il apparaît la notion d'itérateurs qui remplace la notion d'indice de boucle pour des objets plus complexes que les tableaux. Pour de nombreuses classes (de la STL, de Boost,..), cela permet d'unifier l'écriture des structures itératives pour des vecteurs et pour des conteneurs plus complexes. Sachant que la fonction membre **begin()** fournit un itérateur pointant sur le premier élément d'un vecteur, et que **end()** sur le dernier élément, on obtient alors le code équivalent suivant :

```
#include <iostream>
#include<vector>
using namespace std;
int main(){
    vector <double> vec;
    vec.push_back(2.0);
    vec.push_back(3.0);
    vec.push_back(4.0);

    vec.insert(vec.begin(),5.0);

    for(vector<double>::iterator i=vec.begin(); i!=vec.end()); i++)
        {cout<<*i<<endl;}
    exit(0);
}
```

Vu l'écriture de la boucle, on constate qu'un itérateur est une adresse et que la valeur correspondant à cette adresse est obtenue par l'opérateur de déférencement. A ce stade, cela semble bien lourd, mais cela sera très utile pour les autres objets que l'on souhaite utiliser.

Il est bien entendu possible à la fois d'utiliser d'autres classes de cette bibliothèque. Il faut alors placer le fichier d'en-tête définissant le container au début du programme principal de manière similaire au fichier que l'on a utilisé tout au long de ce cours.

7.8 Conclusion

Nous avons vu quelques unes des nombreuses possibilités qu'offrent le C++ pour développer un code permettant une programmation objet, en utilisant les nombreux outils disponibles à travers la STL. On ne peut pas cacher que le développement de templates personnels nécessite très souvent une expertise que l'on acquiert progressivement. Par contre, les éléments développés dans ce chapitre devraient

permettre de prendre en main une utilisation de la programmation objet via l'usage de bibliothèques, et progressivement, de mettre en place une nouvelle manière de programmer avec un niveau d'abstraction permettant de traiter des problèmes jusque là peu abordés avec les langages traditionnels. De plus, on a vu que Python et C++ qui font partie des langages évoluant rapidement ces dernières années, ont des points de convergence grandissants pour obtenir des résultats qui minimisent à la fois le temps de développement et celui du calcul.

CHAPITRE 8

Introduction à la parallélisation

8.1 Motivation

Les processeurs disposent depuis plus de 20 ans plusieurs unités de calcul appelés cœurs qui sont disponibles pour exécuter un code. Par défaut, seul un cœur est utilisé alors que le nombre de cœurs disponibles est souvent supérieur à 4 et peut d'atteindre 96 sur les derniers CPU AMD. Répartir le calcul sur l'ensemble des cœurs est donc un gage d'efficacité. A nouveau, Python et C++ disposent de modules pour le premier et de bibliothèques pour le second qui permettent de réaliser ce travail. Pour éviter de rentrer dans des détails techniques, nous allons prendre une image de ce que représente la réalisation d'un calcul en parallèle.

Si on considère que chaque cœur du processeur représente une personne qui réaliser une partie du travail, le travail du développeur est celui de l'organisation ou la répartition des différentes tâches pour éviter que certaines personnes attendent les autres car ils ne peuvent rien faire. Si chacun réalise une part du travail, il faut aussi s'assurer qu'à la fin il y ait une collecte des informations pour avoir un assemblage complet fournissant le résultat attendu.

Si on poursuit la construction de cette image, il faut décider si chaque personne dispose de la totalité des outils pour réaliser sa tache ou si les outils sont partagés. Dans le premier cas, il n'y a pas de conflits entre les différentes personnes car c'est seulement à la fin que tout est rassemblé mais cela implique un peu un gaspillage des ressources. Dans le second cas, on peut optimiser l'usage des ressources, mais il faut organiser entre les différentes personnes l'ordre de l'utilisation des outils et c'est bien sûr plus complexe. Ces situations correspondent par exemple à la transformation d'un tableau ou chaque un cœur va s'occuper d'une partie des données à calculer. Le tableau se trouvant en mémoire, il faut s'assurer que chaque cœur n'accède pas en même temps au même endroit de la mémoire car cela va créer un conflit d'utilisation. Dans la première méthode, si on a dupliqué le tableau original pour que chaque cœur ait un espace de travail séparé des autres, on a une bonne sécurité, mais si le tableau original est important, la multiplication des copies peut conduite à un fort ralentissement du système voire un dépassement de capacité du système. La moralité de cette histoire est la suivante. Si le travail de la répartition est simple, on choisira une méthode de type deux, et inversement pour un problème complexe, il est inévitable de passer à un travail plus sophistiqué par une méthode de type 1.

Pour compléter ces différents schémas, il est nécessaire de rappeler que deux autres types de parallélisation existent : le premier consiste à utiliser un seul ordinateur et les transferts d'informations se font à l'intérieur de la mémoire. Le second type consiste à répartir les travaux entre plusieurs ordinateurs et donc la collecte d'informations nécessite l'usage d'un réseau rapide entre les ordinateurs pour éviter de consommer du temps pour le transfert de données.

Dans ce chapitre d'introduction à la programmation parallèle, nous ne considérons pas la situation avec plusieurs ordinateurs.

Pour limiter les difficultés, on va considérer un exemple simple de parallélisation. On considère un mouvement Brownien à une dimension d'une particule soumise au potentiel suivant.

$$V(x) = k\left(-\frac{x^2}{2} + \frac{x^4}{4}\right) \tag{8.1}$$

Ce potentiel est illustré par la Figure (8.1). La particule peut se déplacer du coté des x positifs vers les x négatifs en franchissant le maximum de potentiel situé à l'origine.

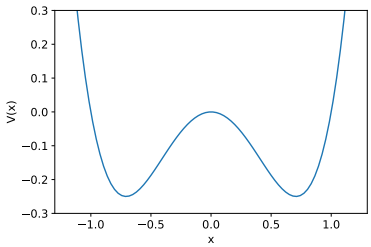


FIGURE 8.1 – Potentiel $V(x)$ avec $k = 1$

Dans la limite suramortie, l'équation du mouvement est donnée par

$$\gamma \dot{x} = k(x - x^3) + \eta(t) \tag{8.2}$$

où $\eta(t)$ est une force aléatoire telle que la valeur moyenne est nulle et que le corrélateur $\langle \eta(t)\eta(t') \rangle = A\delta(t - t')$ où A est une constante.

Il est possible de montrer qu'une particule atteint un état équilibre qui est donnée par la distribution de Gibbs $P_{eq}(x)$

$$P_{eq}(x) = \frac{e^{-\frac{k}{T}(-x^2/2+x^4/4)}}{\int_{-\infty}^{\infty} dx e^{-\frac{k}{T}(-x^2/2+x^4/4)}} \tag{8.3}$$

Pour la résolution numérique, on utilise l'algorithme d'Euler, et avec un choix d'unité approprié, on peut prendre $\gamma = 1$ et $k = 1$, ce qui donne

$$x(t + \Delta t) = x(t) + \Delta t(x - x^3) + \sqrt{(2T\Delta t)}\nu(t) \tag{8.4}$$

où Δt est le pas d'intégration que l'on choisit égal à 0.001, T est la température que l'on choisit égale à 1, et $\nu(t)$ est un nombre aléatoire tiré d'une distribution gaussienne centrée en 0 et de variance 1.

En calculant plusieurs trajectoires et en gardant dans un tableau l'ensemble des positions, on peut construire la distribution d'équilibre avec un histogramme que l'on comparera à la valeur exacte donnée ci-dessus.

8.2 Python

Le langage Python fournit plusieurs modules pour réaliser un calcul parallèle. Le module **joblib** permet de réaliser des taches d'un programme de manière parallèle en utilisant les deux types de méthodes précédemment évoquées. Le second module mpi4py est une manière simple d'utiliser une bibliothèque qui s'appelle Message Passing Interface (MPI) qui peut s'appliquer aussi bien sur un seul ordinateur que sur plusieurs reliés par un réseau. C'est une méthode de second type car cette bibliothèque construit des environnements séparées pour chaque cœur, et une partie importante dans l'utilisation de ce module consiste à implémenter la communication entre les différents cœurs.

Pour bien souligner le changement de code quand on implémente la parallélisation, voici une première version séquentielle de la résolution des équations différentielles stochastiques avec la construction de la distribution des positions que l'on compare à la valeur exacte.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Aug 5 14:28:52 2022

@author: viot
"""

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

def force(x):
    return(x-x**3)

def traj(x,npoints,dt,T,noise):
    for i in range(1,npoints):
        x[i]=x[i-1]+dt*force(x[i-1]) +np.sqrt(2*T*dt)*noise[i-1]

    return(x)

npoints=100_000
start=time.time()
dt=0.001
T=1
nrep=100
x=np.zeros(npoints)
y=np.empty(0)

for jrep in np.arange(nrep):
    x[0]=0
    noise=np.random.normal(size=npoints)
    x=traj(x,npoints,dt,T,noise)
    y=np.append(y,x)

print(time.time()-start)
plt.hist(y,density=True,bins=100)
x=np.linspace(-2.5,2.5,100)
plt.plot(x,np.exp(x**2/2-x**4/4)/3.905137170)
```

Le code résout l'équation différentielle stochastique pour 100000 pas et pour 100 trajectoires indépendantes. La condition initiale est choisie telle que la particule soit à l'origine. Le temps de calcul sur un portable personnel est d'environ 14s.

8.2.1 Module joblib

Sur l'exemple précédent, on voit que la boucle sur les différentes réalisations correspond à des calculs de trajectoires indépendantes. Le module **Joblib** va permettre avec une seule instruction de transformer ce code avec la création de processus indépendants qui vont calculer une partie de la boucle. Le code correspondant est situé ci-dessous, il a fallu faire quelques petits changements. La boucle initiale avait une instruction qui calculait le bruit, elle a été intégré dans la fonction **traj**. la variable y est maintenant la sortie du module **joblib** et contient les 100000 positions des 100 trajectoires sous la forme d'une liste de liste. Pour accélérer la transformation d'une liste de liste en une liste, on utilise une astuce qui permet une transformation rapide (avec le module **itertools**).

Concernant la syntaxe de la commande **joblib**, on précise le nombre de cœurs que l'on souhaite

utiliser. Les résultats sont assez clairs pour l'ordinateur disposant de 4 cœurs : pour un nombre de cœurs égal à 2, le temps est environ de $7s$, pour 4 cœurs, le temps est de $3.5s$. Au delà, plus d'amélioration, et même une décroissance. Pour un nombre de cœurs de 8, le temps est de $5s$.

Pour vérifier que **joblib** a bien créé un nombre de processus égal à ceux demandés, on peut sous Linux ouvrir une fenêtre terminal et lancer la commande top. Quand le code python est exécuté, on voit apparaître un nombre de processus python égal au nombre de cœurs demandé qui disparaisse une fois le calcul effectué. L'option **verbose=10** est optionnelle et permet de récupérer des informations sur le calcul exécuté parallèlement.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Aug 5 14:28:52 2022

@author: viot
"""

import numpy as np
import matplotlib.pyplot as plt
from joblib import Parallel,delayed
import time
import itertools

def force(x):
    return(x-x**3)

def traj(x,npoints,dt,T):
    noise=np.random.normal(size=npoints)
    for i in range(1,npoints):
        x[i]=x[i-1]+dt*force(x[i-1]) +np.sqrt(2*T*dt)*noise[i-1]

    return(x)

npoints=100_000
start=time.time()
dt=0.001
T=1
nrep=100
x=np.zeros(npoints)

y= Parallel(n_jobs=4,verbose=10,backend='loky')(delayed (traj)(x,npoints ,
    ↪ dt,T) for i in np.arange(nrep))

print(time.time()-start)
y= list(itertools.chain.from_iterable(y))
plt.hist(y,density=True,bins=100)
x=np.linspace(-2.5,2.5,100)
plt.plot(x,np.exp(x**2/2-x**4/4)/3.905137170)
```

8.2.2 Module mpi4py

Transformer un programme avec le module **mpi4py** nécessite plus de complexité dans un premier cas, mais il est utilisable non seulement sur l'ensemble des cœurs d'un ordinateur mais aussi sur plusieurs ordinateurs connectés par un réseau.

Avant de passer à la transformation du code précédent, nous allons voir le principe d'un code mini-

mal

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sat Aug 19 16:09:17 2023

@author: viot
"""

from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
if rank == 0:
    print("hello World ",rank)
else:
    print("Bonjour Monde ",rank)
```

Après avoir importé MPI à partir du module **mpi4py** La première instruction définit pour tous les processus un mode noté **comm** dans ce code (théoriquement, on peut en définir plusieurs, mais je ne l'ai jamais utilisé depuis 30 ans!). La variable size récupère le nombre de processus qui ont été demandé au démarrage de l'exécution. La variable rank récupère le numéro attribué à chaque cœur et qui va de 0 à size-1. Avec ce numéro chaque processus va exécuter la partie du code le concernant.

Il faut maintenant lancer l'exécution en tapant dans une fenêtre terminal :

```
mpirun -np 4 python3 hellompi.py
```

Cette commande demande 4 cœurs pour l'exécution du code python. La commande mpirun est un script qui vient de la bibliothèque OPENMPI qui est aussi utilisée en C++.

Le résultat obtenue est

```
Bonjour Monde 3
Bonjour Monde 1
Bonjour Monde 2
hello World 0
```

On voit que seul le cœur avec l'indice 0 a fait une réponse en anglais. Par ailleurs, il es possible de changer le nombre de cœurs au moment de l'exécution et non pas dans le code.

Passons maintenant au code modifié pour le calcul des trajectoires

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Aug 5 14:28:52 2022

@author: viot
"""

import numpy as np
import matplotlib.pyplot as plt

import itertools

from mpi4py import MPI

comm = MPI.COMM_WORLD
```

```
size = comm.Get_size()
rank = comm.Get_rank()

def force(x):
    return(x-x**3)

def traj(x,npoints,dt,T,noise):
    for i in range(1,npoints):
        x[i]=x[i-1]+dt*force(x[i-1]) +np.sqrt(2*T*dt)*noise[i-1]

    return(x)

npoints=100_000

dt=0.001
T=1
nrep=100
x=np.zeros(npoints)
y=np.empty(0)

if rank==0: wt=MPI.Wtime()
for jrep in np.arange(rank,nrep,size):
    x[0]=0
    noise=np.random.normal(size=npoints)
    x=traj(x,npoints,dt,T,noise)
    y=np.append(y,x)

Ycollect = comm.gather(y,root=0)

print('tableau final', type(Ycollect))

if rank==0:
    print('temps ecoule',MPI.Wtime()-wt)
    Ycollect= list(itertools.chain.from_iterable(Ycollect))
    plt.hist(Ycollect,density=True,bins=100)
    x=np.linspace(-2.5,2.5,100)
    plt.plot(x,np.exp(x**2/2-x**4/4)/3.905137170)
    plt.show()
```

Le code commence bien entendu par les variables usuelles dont le programme a besoin comme dans le code précédent. La boucle est modifiée de telle manière que chaque cœur réalise une fraction identique du travail (ou presque car il faut que 100 soit divisible par le nombre de cœurs). Une fois réalisée la commande **gather** permet de collecter tous les résultats sur le cœur 0, y compris celui du cœur 0. La variable Ycollect est une liste de liste comme dans le cas de **joblib**, mais uniquement sur le cœur 0, les autres ont une variable vide. En conséquence, on réalise la transformation de liste de liste en liste sur le cœur 0 ainsi que le graphe que l'on souhaite obtenir. Au final les résultats sont proches de ceux obtenus avec **joblib**, par exemple $3.6s$ pour 4 cœurs. Je rappelle que pour exécuter le code dont le nom du fichier python est **quarticmpi.py** il faut entrer dans une fenêtre terminal la commande :

```
mpirun -np 4 python3 quarticmpi.py
```

Il existe bien évidemment de nombreux autres instructions pour transmettre des données entre processus avec MPI. Il est nécessaire a minima d'aller voir la documentation sur le site de mpi4py.

8.3 C++

```
#include<iostream>
#include<fstream>
#include <vector>
#include<random>
#include<algorithm>
#include<ctime>
using namespace std;
double force(double x) {return(x-x*x*x);
vector <double> traj(vector <double> &x,int npoints,double dt, double T)
{
    random_device rd; //Will be used to obtain a seed for the random number engine with Linux
        ↪ and MacOx
    mt19937 gen(rd()); //Standard mersenne_twister_engine seeded with rd
    normal_distribution<double> dis{0,1}; // loi normale (mu=0, sigma=1)
    for(int i=1; i<npoints; i++)
        x[i]=x[i-1]+dt*force(x[i-1]) +sqrt(2*T*dt)*dis(gen);
    return(x);
}
double exact(double x,double T){return(exp((x*x/2-pow(x,4)/4/T))/3.90515170) ;}
int main()
{
    const int npoints=100000;
    const double T=1.0;
    const double Deltat=0.001;
    const int nrep=100;
    const int nbins=200;
    vector <double> x(npoints,0);
    vector <double> y;
    clock_t t1,t2; //clocks for timing information
    t1=clock();
    for (int j=0; j<nrep; j++)
    {
        x[0]=0;
        x=traj(x,npoints,Deltat,T);
        for (int i=0; i<npoints; i++)
            y.push_back(x[i]);
    }
    t2=clock();
    cout<<"temps ecoule "<<(t2-t1)/(double)CLOCKS_PER_SEC<<endl;
    sort(y.begin(),y.end());
    vector <double> histogram(nbins,0);
    double bin = y[0]; //Choose the starting point with the minimum value
    const double bin_width = (y[npoints*nrep-1]-y[0])/nbins; //Choose the bin interval
    for (auto e : y) ++histogram[int((e-y[0])/bin_width) ];
    double sumh=accumulate(histogram.begin(),histogram.end(),0.0);
    for (auto &ih: histogram) ih/=sumh*bin_width;

    ofstream fichier;
    fichier.open("quartic.dat");
    for (int i=0; i<nbins; i++)
    {
        double xpos= bin+(i+0.5)*bin_width;
        fichier<<xpos<<" "<<histogram[i]<<" "<<exact(xpos,T)<<endl;
    }
```

```
    fichier.close();
    exit(0);
}
```

Pour le langage C++, on a deux possibilités pour paralléliser un code. Si on ne dispose que d'un seul ordinateur, on peut assez facilement utiliser une méthode de type 1 comme cela a été défini au début du chapitre. Pour comparer à nouveau un code séquentiel avec un code parallèle, on va donner tout d'abord le code séquentiel.

Quelques commentaires concernant ce code. Le calcul du bruit est réalisé dans la fonction **traj**, car il n'y a pas de gain a créer un tableau contrairement au langage Python avec **numpy**. Pour calculer l'histogramme, on utilise l'instruction **sort** de la STL pour accélérer le calcul du remplissage des histogrammes. La compilation se fait avec l'option d'optimisation.

```
g++ quartic.cpp -o quartic -Ofast
```

Le temps de calcul est déjà descendu à $0.4s$.

8.3.1 Bibliothèque OPENMP

La bibliothèque permet de faire une partie du code en C++. L'avantage de celle-ci est qu'il suffit d'ajouter une ligne de code de type **pragma** avant la partie concernée et de compiler avec une option **-fopenmp** pour générer un code qui peut s'exécuter en parallèle. Il reste à choisir le nombre de cœurs que l'on souhaite utiliser. Pour cela, il faut donner une valeur à une variable spécifique. Taper dans une fenêtre terminal la commande suivante pour utiliser 4 cœurs :

```
export OMP_NUM_THREADS=4
```

Attention, la valeur de la variable est fixée ici dans cette fenêtre terminal mais aucunement dans les autres que l'on peut ouvrir (ou qui sont déjà ouvertes).

```
#include<iostream>
#include<fstream>
#include <vector>
#include<random>
#include<algorithm>
#include<ctime>
#include<omp.h>
using namespace std;
double force(double x) {return(x-x*x*x);}
 vector<double> traj(vector <double> x,int npoints,double dt, double T)
{
    random_device rd; //Will be used to obtain a seed for the random number engine with Linux
        ↪ and MacOx
    mt19937 gen(rd()); //Standard mersenne_twister_engine seeded with rd
    normal_distribution<double> dis{0,1}; // loi normale (mu=0, sigma=1)
    //cout<<dis(gen)<<" "<<omp_get_thread_num()<<endl;
    for(int i=1; i<npoints; i++)
        x[i]=x[i-1]+dt*force(x[i-1]) +sqrt(2*T*dt)*dis(gen);
        return x;
}
double exact(double x,double T)
{
    return( exp((x*x/2-pow(x,4)/4/T))/ 3.90515170) ;
}
int main()
{
```

```
    const int npoints=100000;
    const double T=1.0;
    const double Deltat=0.001;
    const int nrep=100;
    const int nbins=200;
    vector <double> y(npoints*nrep,0);;
    clock_t t1,t2; //clocks for timing information
vector <double> x;

t1=clock();
#pragma omp parallel
cout<<"threads " <<omp_get_thread_num()<<endl;
#pragma omp parallel for private(x) shared(y)
for (int j=0; j<nrep; j++)
    { x.resize(npoints);
    x[0]=0;
    cout<<" j "<<j<<" rank "<<omp_get_thread_num()<<endl;
      x= traj(x,npoints,Deltat,T);
      for (int i=0; i<npoints; i++)
           y[j*npoints+i]=x[i];
    }
    t2=clock();
    cout<<"temps ecoule "<<(t2-t1)/(double)CLOCKS_PER_SEC<<endl;
    cout<<"taille "<<y.size()<<endl;
    sort(y.begin(),y.end());
    vector <double> histogram(nbins+1,0);

    double bin = y[0]; //Choose the starting point with the minimum value
    const double bin_width = (y[y.size()-1]-y[0])/nbins; //Choose the bin interval

   for (auto e : y)
    {
       ++histogram[int((e-y[0])/bin_width) ];
    }
    double sumh=accumulate(histogram.begin(),histogram.end(),0.0);
    for (auto &ih: histogram)
    {
       ih/=sumh*bin_width;
    }
    ofstream fichier;
    fichier.open("quartic.dat");
    for (int i=0; i<nbins; i++)
    {
       double xpos= bin+(i+0.5)*bin_width;
       fichier<<xpos<<" "<<histogram[i]<<" "<<exact(xpos,T)<<endl;
    }
    fichier.close();
    exit(0);
}
```

8.3.2 Bibliothèque OPENMPI

Avec la bibliothèque MPI en C++, on retrouve les mêmes idées que l'on avait vu en Python. Dans une première étape, on définit un monde puis on récupère le nombre de cœurs fixé au lancement du programme ainsi que le numéro du processus. Une fois les boucles réalisées, on utilise comme précé-

demment la fonction **MPI_Gather**.

```
#include<iostream>
#include<fstream>
#include <vector>
#include<random>
#include<algorithm>
#include<ctime>
#include<mpi.h>
using namespace std;
double force(double x) {return(x-x*x*x);
vector <double> traj(vector <double> &x,int npoints,double dt, double T)
{
    random_device rd; //Will be used to obtain a seed for the random number engine with Linux
        ↪ and MacOx
    mt19937 gen(rd()); //Standard mersenne_twister_engine seeded with rd
    normal_distribution<double> dis{0,1}; // loi normale (mu=0, sigma=1)
    for(int i=1; i<npoints; i++)
        x[i]=x[i-1]+dt*force(x[i-1]) +sqrt(2*T*dt)*dis(gen);
    return(x);
}
double exact(double x,double T)
{
    return(exp((x*x/2-pow(x,4)/4/T))/3.90515170) ;
}
int main(int argc, char **argv)
{
//! parallelization
    MPI_Init(&argc, &argv);
    int numprocs;
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    int myrank;
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    //! no broadcasting is necessary
    const int npoints=100000;
    const double T=1.0;
    const double Deltat=0.001;
    const int nrep=100;
    const int nbins=200;
    vector <double> x(npoints,0);
    vector <double> y;
    vector <double> ycollect(npoints*nrep,0);
    clock_t t1,t2; //clocks for timing information
    t1=clock();
    for (int j=0; j<nrep; j+=numprocs)
    {
        x[0]=0;
        x=traj(x,npoints,Deltat,T);
        for (int i=0; i<npoints; i++)
        y.push_back(x[i]);
    }
    t2=clock();
     cout<<"temps␣ecoul␣"<<(t2-t1)/(double)CLOCKS_PER_SEC<<endl;
     cout<<"myrank␣"<<myrank<<"␣taile␣de␣y␣"<<y.size()<<endl;
    MPI_Gather(&y[0],y.size(),MPI_DOUBLE,&ycollect[0],y.size(),MPI_DOUBLE,0,MPI_COMM_WORLD);
    MPI_Barrier;
    if (myrank==0)
```

```
    {
    sort(ycollect.begin(),ycollect.end());
    vector <double> histogram(nbins,0);
    double bin = ycollect[0]; //Choose the starting point with the minimum value
    const double bin_width = (ycollect[npoints*nrep-1]-ycollect[0])/nbins; //Choose the bin
        ↪ interval
    for (auto e : ycollect)
    {
       ++histogram[int((e-ycollect[0])/bin_width) ];
    }
    double sumh=accumulate(histogram.begin(),histogram.end(),0.0);
    for (auto &ih: histogram)
    {
       ih/=sumh*bin_width;
    }
    ofstream fichier;
    fichier.open("quartic.dat");
    for (int i=0; i<nbins; i++)
    {
       double xpos= bin+(i+0.5)*bin_width;
       fichier<<xpos<<" "<<histogram[i]<<" "<<exact(xpos,T)<<endl;
    }
    fichier.close();
     }
      MPI_Finalize();

    exit(0);
}
```

Le temps d'exécution décroît bien comme le nombre de **cœurs**. Typiquement, on arrive a avoir un temps de l'ordre de $0.1s$ pour 4 cœurs. Rappelons que le code séquentiel en python donnait $14s$. Cela donne un facteur 140. Si on n'optimise pas le Python, le facteur 1000 peut être dépassé.

8.4 Conclusion

La parallélisation permet comme nous l'avons vu dans ce chapitre de réduire le temps physique de calcul avec une utilisation de l'ensemble des cœurs disponibles. Bien entendu, cela ne dispense de commencer par une écriture pertinente du code dans la version séquentiel, en particulier en Python où l'utilisation des modules **numpy** et **scipy** peuvent améliorer les performances de plus d'un ordre de grandeur. Toutefois pour des calculs longs, la parallélisation est un outil puissant dont il convient d'apprendre les méthodes de programmation pour obtenir un accroissement des performances et surtout une utilisation rationnelle des ressources.

CHAPITRE A

Python et C++ : la réunification

A.1 Introduction

Nous avons vu dans les chapitres précédents que Python et C++ avaient très souvent de grandes similitudes. En revanche, écrire un code en C++ demande toujours plus de temps pour la mise au point que le même code en Python. Inversement, le temps d'exécution est généralement en faveur du C++ de manière très importante par rapport au Python. Si on a besoin de visualiser les résultats ou de faire un post-traitement des résultats, les modules de Python sont nombreux et évitent d'écrire de longs code en C++. En résumé, choisir entre les deux langages est souvent déchirant car il est nécessaire de garder les avantages des deux langages pour optimiser le temps de développement et celui d'exécution.

Si on se rappelle que les modules de Python concernant les calculs scientifiques sont des codes qui ont été écrits en Fortran, C ou C++, on voit donc une solution conduisant à la meilleure optimisation, celle qui consiste à combiner les deux langages.

Il existe actuellement plusieurs méthodes pour permettre une utilisation conjointe d'un code écrit en partie en C/C++ avec une autre partie en Python. Sans être exhaustif, on peut citer : SWIG, Pybind11, CFFI, Cython, Boost.Python, Cppyy ...

Dans le cadre de ce cours introductif, on va se limiter à une interface qui sera SWIG (et/ou PyBind11) et nous allons illustrer son utilisation sur les deux concepts phares de ce cours, à savoir dans un premier temps, la programmation procédurale et dans un second temps la programmation objet. Avec un niveau de difficultés croissant, nous allons considérer plusieurs exemples et vérifier l'accélération d'exécution engendrée par cette combinaison de deux langages.

Il faut reconnaître que la création d'une interface est singulièrement moins simple que de mélanger des parties de code C et C++, voire C et Fortran. L'une des raisons à l'origine de cette difficulté est liée au fait que Python est un langage scripté et C++ (ou C) est un langage compilé.

Je rappelle que le but de ce mélange est d'avoir une meilleure efficacité en Python. Cela signifie que les parties de code en C++ vont être transformées en module de python et non pas l'inverse !

A.2 Programmation procédurale

Dans ce premier exemple, nous allons construire une interface pour appeler une fonction C++ en Python. Il faut donc créer un fichier d'interface entre le python et le C++ puis ensuite SWIG crée un fichier C++ pour créer une interface avec le python. Ce fichier ainsi que le fichier C++ de la fonction doivent être alors compilés et on crée une bibliothèque qui sera utilisée par l'interpréteur Python.

Comme on le voit, ce n'est pas très simple, mais si on respecte assez scrupuleusement les règles, on verra que cela fonctionne correctement. L'exemple choisi pour illustrer la méthode est de créer une fonction factorielle en C++ qui puisse être utilisée en Python. On sait qu'il existe une fonction avec le module **numpy**, mais on vise ici de montrer la faisabilité de la méthode.

Voici la fonction factorielle écrite en C++ (Je laisse en exercice de transformer cette fonction avec une structure récursive) :

```
/* File: example.cpp */

#include "example.h"

double fact(int n) {
    if (n < 0){ /* This should probably return an error, but this is simpler */
        return 0;
    }
    if (n == 0) {
        return 1;
    }
    else {
          double sum=1;
          for (int m=n;m>0;m--)
                sum*=m;

        return sum;
    }
}
```

Le fichier **example.h** donne simplement le prototype de la fonction :

```
/* File: example.h */

double fact(int n);
```

Le fichier d'interface de SWIG a la structure suivante :

```
/* File: example.i */
%module example

%{
#define SWIG_FILE_WITH_INIT
#include "example.h"
%}

double fact(int n);
```

Il faut alors exécuter la commande suivante

```
swig -python -c++ example.i
```

On a alors créé un fichier **example_wrap.cxx** qui est nécessaire pour l'interface C++/Python

Il faut alors compiler chaque fichier C++ et créer la bibliothèque partagée que l'on utilisera avec Python.

```
g++ -O2 -fPIC -c example.cpp
g++ -O2 -fPIC -c example_wrap.cxx -I/usr/include/python3.9
g++ -shared example.o example_wrap.o -o _example.so
```

Quelques remarques :

1. SWIG génère des codes C++ avec le suffixe cxx au lieu de cpp. Cela fonctionne très bien avec le compilateur g++. Il faut par contre ne pas oublier ce détail dans la commande de compilation.
2. Ne pas oublier l'option **-fPIC** qui permet d'avoir un code dit de position indépendante.
3. Ne pas oublier l'option qui précise où se trouvent les fichiers d'entête de la bibliothèque Python (sur votre ordinateur, vérifier le numéro de version pour que cela fonctionne)
4. la troisième commande crée la bibliothèque et noter qu'il est indispensable que le premier caractère soit un _

Il apparaît un fichier **_example.so**.

Pour utiliser cette fonction sous python, créer le script python suivant :

```
import example
example.fact(5)
```

Le résultat obtenu est correct et donc cela fonctionne.

Comme on le voit clairement, les différentes étapes sont un peu longues et on peut bien sûr choisir une autre procédure pour aboutir à un résultat identique. Les deux première étapes qui créent les fichiers **example.cpp** et **example.i** ainsi que la création du fichier C++ d'interface sont inchangés. Si l'on veut automatiser un peu le reste, on peut créer un script de configuration avec les outils du module de gestion des distributions Python (**distutils**) :

```
#!/usr/bin/env python

"""
setup.py file for SWIG example
"""

from distutils.core import setup, Extension


example_module = Extension('_example',
                           sources=['example_wrap.cxx', 'example.cpp'],
                           )

setup (name = 'example',
       version = '0.1',
       author = "SWIG Docs",
       description = """ exemple basique avec SWIG""",
       ext_modules = [example_module],
       py_modules = ["example"],
       )
```

Lorsque Python interprète ce script, l'instruction **setup** va réaliser les compilations précédentes et créer un fichier de bibliothèque **_example.cpython-39-x86_64-linux-gnu.so** . Malgré un nom compilé différent, le script précédent fonctionne de la même manière.

A.3 Programmation objet

Après cet exemple somme toute trivial, il est temps de voir des situations plus réalistes permettant un échange entre le script python et les parties codées en C++ plus significatives. Typiquement, un programme dont le calcul nécessite d'utiliser le C++ à la place du python intervient dans deux types de situations : soit le programme C++ fait l'essentiel du travail et nous allons voir comment utiliser la classe vector de la STL sous python ce qui conduira a utiliser des tableaux dont la structure est définie en C++. La deuxième situation correspond à un programme python dont une partie du calcul nécessite de passer en C++. Cela signifie généralement que le code utilise principalement des tableaux définis sous numpy et dont il convient de les transmettre à la partie C++.

A.3.1 STL

SWIG dispose de fichiers d'interface déjà présents et qui permettent de faire une interface entre le Python et le C++. On peut donc inclure des fichiers d'interface existants à l'intérieur d'un fichier d'interface personnalisé. En conséquence le fichier d'enrobage (le *wrapper*) que SWIG construit devient assez important et je ne peux pas cacher que la mise au point des codes n'est pas une tâche toujours très

simple. Il est toujours possible de se référer au manuel de SWIG qui est un ouvrage de 500 pages pour la dernière version (4.0), mais dans l'optique de ce cours introductif, on peut utiliser les exemples que l'on va donner comme des recettes de cuisine sans entrer dans le détail de leur construction.

Comme le niveau de difficulté augmente, on ne présentera que la construction de cette interface en utilisant un fichier de configuration **setup.py** .

Voici la fonction **imprime** écrite en C++ :

```
/* File: vector.cpp */

#include "vector.h"

void imprime(std::vector<double> v) {
       for (auto i :v )
       std::cout<<i<<std::endl;
return;
};
```

Le fichier **vector.h** donne simplement le prototype de la fonction

```
#include<iostream>
#include<vector>

void imprime(std::vector<double> v);
```

Le fichier d'interface de SWIG a la structure suivante

```
%module Vector
%{
#define SWIG_FILE_WITH_INIT
#include "vector.h"
%}

%include "std_vector.i"

namespace std {
%template(DoubleVector) vector<double>;
}
using namespace std;

void imprime(std::vector<double> v);
```

Concernant ce dernier fichier, on note que la présence de deux éléments nouveaux par rapport à l'exemple précédent. L'inclusion du fichier **std_vector.i** . Il contient tous les éléments pour permettre l'utilisation de vecteurs de la STL sous python. Le deuxième élément nouveau est la présence de template qui définit un nom à partir duquel on pourra sous python créer un vecteur de doubles avec les fonctionnalités de la classe de la STL [1].

Il faut alors exécuter la commande suivante

```
swig -python -c++ vector.i
```

On a alors créé un fichier **vector_wrap.cxx** qui est nécessaire pour l'interface C++/Python. Pour ceux qui ont la curiosité de voir ce que ce fichier inclut, on peut noter que le fichier **vector_wrap.cxx** généré une fois que l'on a utilisé la commande **swig** possède plus de 8000 lignes !

Le script Python utilise la commande **setup** du module **distutils**, qui va lancer la compilation des différents fichiers objets ainsi que produire la bibliothèque de l'extension (argument *build_ext*). Elle sera

1. L'ensemble de ces fichiers reste de volume modeste, mais pour converger vers un résultat, cela nécessite de nombreux essais.

placée dans le répertoire courant (option *inplace*).

```
python setup.py build_ext --inplace
```

Pour tester cette nouvelle bibliothèque, on a le script python suivant :

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Jul 22 21:02:39 2021

@author: viot
"""
import Vector
AA=Vector.DoubleVector(10)
Vector.imprime(AA)
AA=Vector.DoubleVector(10,2) #utilisation d'un second constructeur de vector de la
    ↪ STL
Vector.imprime(AA)
AA.size()
AA.push_back(6) #methode de vector
Vector.imprime(AA) #le tableau contient maintenant 11 termes
#en utilisant la fonction membre de vector a parir de python. La classe!
```

A.3.2 Numpy

La seconde situation consiste à transmettre les tableaux numpy au langage C++. À nouveau ce point va être facilité car SWIG dispose d'un fichier d'interface **numpy.i** que l'on peut inclure dans le fichier d'interface personnalisé.

Contrairement à ce qu'indique le site de numpy, le fichier **numpy.i** n'est pas présent dans les distributions Ubuntu et Fedora que j'ai testé. Avant le script suivant, il est possible de télécharger ce fichier qui va alors se trouver dans le répertoire où le script est utilisé, par exemple avec le script **fetch.py** :

```
import re
import requests
import numpy

np_version = re.compile(r'(?P<MAJOR>[0-9]+)\.'
                    '(?P<MINOR>[0-9]+)') \
                    .search(numpy.__version__)
np_version_string = np_version.group()
np_version_info = {key: int(value)
                for key, value in np_version.groupdict().items()}

np_file_name = 'numpy.i'
np_file_url = 'https://raw.githubusercontent.com/numpy/numpy/maintenance/' + \
            np_version_string + '.x/tools/swig/' + np_file_name
if(np_version_info['MAJOR'] == 1 and np_version_info['MINOR'] < 9):
    np_file_url = np_file_url.replace('tools', 'doc')

chunk_size = 8196
with open(np_file_name, 'wb') as file:
    for chunk in requests.get(np_file_url,
                          stream=True).iter_content(chunk_size):
        file.write(chunk)
```

Voici une fonction **fill** d'initialisation d'un tableau, écrite en C++ :

```
/* File: ndarray.cpp */
#include "ndarray.h"
void fill(double *vec, int n)
{
   for (int i=0; i< n; i++)
       vec[i] = 0.1*i;
}
```

Le fichier **ndarray.h** donne simplement le prototype de la fonction :

```
void fill(double *vec, int n);
```

Le fichier d'interface **ndarray.i** de SWIG a la structure suivante :

```
%module Ndarray
%{
#define SWIG_FILE_WITH_INIT
#include "ndarray.h"
%}
%include "numpy.i"

%init %{
import_array();
%}

%numpy_typemaps(double, NPY_DOUBLE, int)
%apply (double* ARGOUT_ARRAY1, int DIM1) {(double* vec, int n)}
void fill(double *vec, int n);
```

Concernant ce dernier fichier, il m'est difficile de commenter l'ensemble de ces lignes et on les utilisera comme une recette de cuisine.

À nouveau, on exécute la commande SWIG suivie de la commande python qui génère la bibliothèque, en ayant adapté le fichier **setup.py** pour travailler sur **ndarray**.

Pour tester ce nouveau module, on a créé le script python suivant :

```
#!/usr/bin/env python

"""
setup.py file for SWIG example
"""

from distutils.core import setup, Extension


ndarray_module = Extension('_Ndarray',
                           sources=['ndarray_wrap.cxx', 'ndarray.cpp'],
                           )

setup (name = 'ndarray',
       version = '0.1',
       author = "Pascal␣V␣",
       description = """Exemple avec numpy """,
       ext_modules = [ndarray_module],
       py_modules = ["ndarray"],
       )
```

Il reste un petit problème puisque le test passe très bien en utilisant Ipython mais s'arrête sur Spyder.

CHAPITRE B

Installation des logiciels

B.1 Introduction

Les deux interfaces pour les langages Python et C++ ont été choisies pour leur disponibilité sur les trois plateformes (Linux, Windows, MacOX) sur les ordinateurs personnels. Concernant Python, il existe d'autres interfaces telles que Jupyter et Pycharm qui ont des fonctionnalités au moins comparables voire supérieures. Si vous avez déjà une expérience sur ces interfaces, il n'est pas interdit de continuer à les utiliser, mais vous avez aussi l'occasion de découvrir un autre produit, où vous retrouverez sous une forme légèrement différente, vos habitudes. L'utilisation d'interface graphique pour le développement de code accélère l'écriture et sa vérification grâce aux outils facilement accessibles. De la même manière, Codeblocks n'est pas la seule interface disponible (Eclipse est par exemple très utilisée), mais peut être utilisée simplement avec un programme unique et court (ce qui sera une grande partie de son usage dans ce cours d'introduction sur le C++). Même si les interfaces graphiques permettent rapidement d'être efficaces, cela nécessite une période de prise en main du logiciel que l'on ne peut pas éliminer. Le premier appendice est justement dédié à ces produits. Si vous avez d'un ordinateur personnel, il est fortement recommandé de compléter vos séances de TP par un apprentissage personnel. Les deux produits choisis ont le privilège d'être disponibles sous quasiment toutes les plateformes. Il est probable que vous trouverez des petites différences entre les logiciels installés dans les salles de l'UFR et ceux sur votre ordinateur personnel (des couleurs différentes et des menus légèrement modifiés), mais très vite on peut passer de l'un à l'autre sans difficulté.

Nous allons donc voir comment installer ces logiciels sur chaque plateforme pour que vous puissiez avoir les mêmes outils pour travailler dans les salles informatiques de l'UFR et sur votre ordinateur personnel.

B.2 Spyder5

Python2 n'est plus maintenu depuis Janvier 2020 et donc nous travaillerons exclusivement sur Python3. Spyder5 est l'interface correspondant au Python3. Si vous avez une ancienne version de Spyder, il faut donc installer Spyder4 au moins. Si vous avez une version de Python 3.10 où au dela, il est nécessaire d'avoir la version 5 de Spyder.

B.2.1 Linux

Les distributions Linux les plus courantes proposent des paquets d'installation de Spyder5 ainsi que Python 3 : Ubuntu, Debian, Fedora, ... Il est possible d'installer les paquets de la distribution. Mais une installation à partir d'Anaconda est possible sans détruire l'installation de la distribution.

Selon l'interface graphique sur laquelle vous travaillez (gnome, Kde Plasma, XFCE,..), une fois installé, le logiciel devrait se trouver dans le menu des applications. Vous avez alors la possibilité de lancer le logiciel par ce moyen. Une alternative consiste à ouvrir une fenêtre terminal dans laquelle vous tapez la commande **Spyder5** suivi de la touche d'entrée.

B.2.2 Windows

: Pour télécharger **Spyder**, vous utilisez votre navigateur et moteur de recherche préféré en tapant le mot clé Spyder5. Vous trouverez le site https://Spyder-ide.org à partir duquel vous pouvez obtenir le logiciel avec une installation de Python via **Anaconda**. Une fois installée, l'application **Spyder5** apparaît dans un sous menu de **Anaconda**. Vous pouvez lancer alors l'application.

B.2.3 MacOx

Pour télécharger Spyder, vous suivez la même procédure que pour l'installation sous Windows. Vous téléchargez le paquet Anaconda pour MacOx (situé au bas de la page https://www.anaconda.com/products/individual). Une fois installée, vous pouvez utiliser finder pour lancer l'application.

Par contre, il est très important d'avoir une installation correspondant au Python3 et une version de Spyder correspondante. En effet, les versions de Python 2 sont très souvent incompatibles avec le Python 3.

B.3 Codeblocks

B.3.1 Linux

A nouveau, les distributions Linux les plus courantes proposent des paquets d'installation de Codeblocks ainsi que le compilateur g++ de Gnu : Ubuntu, Debian, Fedora, ... Il est préférable d'installer les paquets de la distribution par rapport à une installation directe. En effet, le système de paquets d'une distribution vérifie la présence de bibliothèques nécessaire au bon fonctionnement du logiciel et installe les paquets supplémentaires automatiquement. Vous avez même la possibilité d'utiliser d'autres compilateurs tels que Clang, voire le compilateur Intel disponible gratuitement sous Linux

Selon l'interface graphique sur laquelle vous travaillez (Gnome, Kde Plasma, XFCE,..), une fois installé, le logiciel devrait se trouver dans le menu des applications. Vous avez alors la possibilité de lancer le logiciel par ce moyen. Une alternative consiste à ouvrir une fenêtre terminal dans laquelle vous tapez la commande **codeblocks** suivi de la touche d'entrée. La version installée sous Ubuntu 20.04 est la dernière version 20.03 et cela correspond aussi aux versions que vous pouvez installer sur les autres plateformes, hormis MacOx.

B.3.2 Windows

Pour télécharger Codeblocks, vous utilisez votre navigateur et votre moteur de recherche préféré en tapant le mot clé Codeblocks. Vous trouverez le site . Une fois installée, l'application Codeblocks apparaît dans la liste de applications présentes dans le menu de Windows. Vous pouvez lancer l'application.

B.3.3 MacOx

Pour télécharger Codeblocks, vous suivez la même procédure que pour l'installation sous Windows. Vous téléchargez le paquet Anaconda pour MacOx (situé sur la page). La version est plus ancienne car il y a actuellement un manque de développeurs pour maintenir cette plateforme (avis aux amateurs). Je n'ai pas testé cette version sous MacOx, mais elle devrait convenir pour le cours et sinon les utilisateurs des Mac ont souvent l'habitude d'utiliser Xcode qui présente des fonctionnalités largement comparables.

CHAPITRE C

Structure simplifiée du fonctionnement d'un ordinateur

C.1 Système d'exploitation

L'apparition des ordinateurs étant relativement récente et l'on la peut situer environ à la fin de la seconde guerre mondiale pour une petite communauté de scientifiques, et au début des années 80, pour un grand public. La montée en puissance des ordinateurs avec le développement d'internet dans les années 90, a contribué à renouveler les langages. Durant la première période de la création de l'informatique, il a fallu développer des programmes qui permettent à la fois de piloter la machine physique puis de fournir des outils pour que l'utilisateur puisse commander la machine. La première partie correspond au système d'exploitation, même si celui-ci doit être aidé au démarrage de l'ordinateur par un plus petit programme pouvant installer le système d'exploitation dans la mémoire centrale de l'ordinateur. C'est un peu le paradoxe de l'œuf et de la poule car pour mettre en marche les fonctions de base d'un ordinateur et donc utiliser le système d'exploitation, il faut un premier programme qui le permette (c'est le rôle du BIOS historiquement sur les PC). Nous n'irons pas plus loin sur cette partie ô combien délicate pour la sécurité. Bon nombre de programmes malveillants cherchent à entamer la sécurité de l'ordinateur en s'activant au démarrage de l'ordinateur.

On suppose que l'on arrive à l'étape où l'ordinateur a chargé en mémoire le système d'exploitation (en fait une partie, d'autres outils restent stockés sur le support de stockage en attendant d'être sollicités par le système lui-même ou un utilisateur.) Après quelques secondes de plus, les ordinateurs présentent une interface graphique qui propose une invite de connexion. Cela signifie qu'une partie de la mémoire est aussi maintenant occupée par le système graphique. Une fois passée l'étape de connexion, vous avez enfin la possibilité d'utiliser l'ordinateur en lançant des applications, et pour ce que ce qui nous intéresse, lancer les interfaces graphiques. Ceux ci permettent d'écrire des codes et ensuite de les exécuter.

Il faut savoir que l'occupation de la mémoire commence par le système d'exploitation et que son usage sera soumis à des droits. C'est nécessaire pour la sécurité et le bon fonctionnement. Les programmes liés au système possèdent des droits supérieurs à ceux lancés par l'utilisateur. En effet, supposons que votre programme cherche à écrire sur une partie occupée par le système, cela va entraîner, soit un dysfonctionnement grave, soit un arrêt de l'ordinateur. Normalement tous les programmes reliés au système sont protégés mais il existe souvent des failles (de sécurité) que les programmes malveillants exploitent pour non pas stopper le fonctionnement de l'ordinateur, mais pour utiliser les privilèges du système pour récupérer des données ou modifier profondément le fonctionnement de l'ordinateur. Ce cours n'a pas l'ambition de vous proposer les méthodes pour prendre le contrôle de l'ordinateur, quand vous programmez, il se peut qu'à l'exécution vous cherchiez à écrire sur une région interdite et dans le cas, le système va vous l'interdire en stoppant votre code. Cela explique une raison fréquente dans l'interruption d'un programme que l'on développe.

C.2 Stockage

Un élément important qu'il est nécessaire de connaître concerne l'organisation du stockage sur l'ordinateur. Typiquement, celle-ci repose sur une structure que l'on peut représenter sous la forme d'un arbre. Le nœud initial s'appelle la racine et à partir de ce noeud, on crée des branches secondaires (si le système l'autorise) et ainsi de suite. Les figures suivantes montrent la structure de la configuration sous Linux puis sous Windows de cette structure à partir de la racine de chaque système d'exploita-

tion. L'interface graphique utilisée donne un aperçu sous la forme de camemberts concentriques et dont l'importance est donnée par le volume d'occupation.

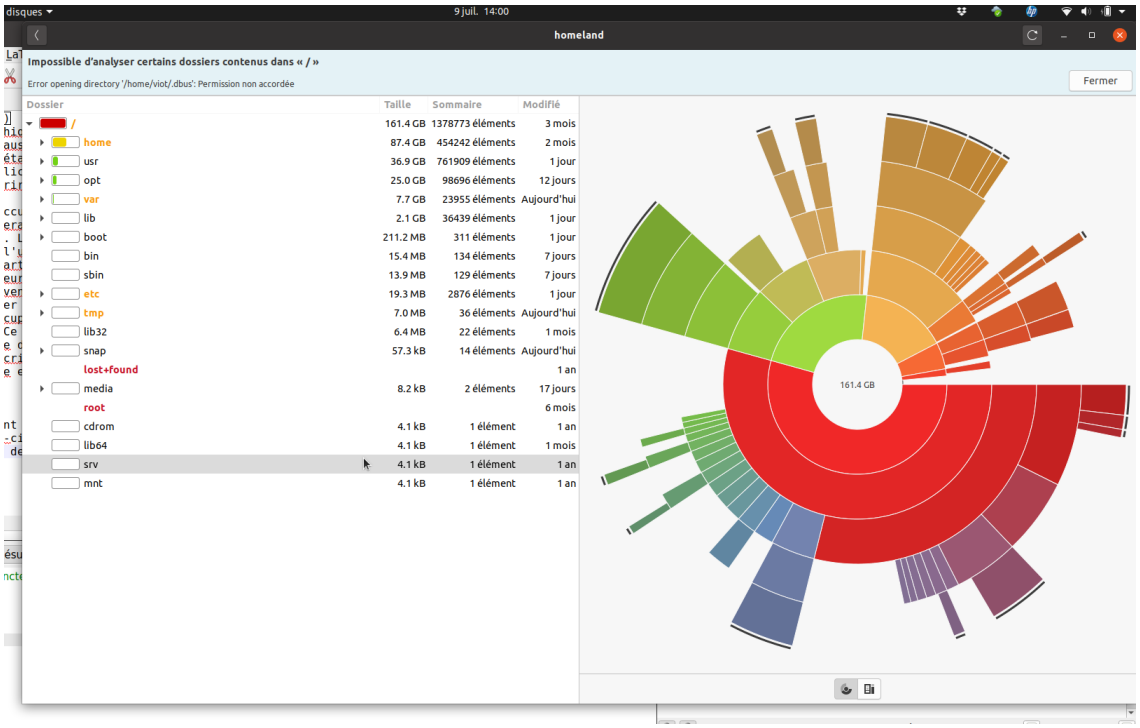


FIGURE C.1 – Structure du disque dur sous Linux

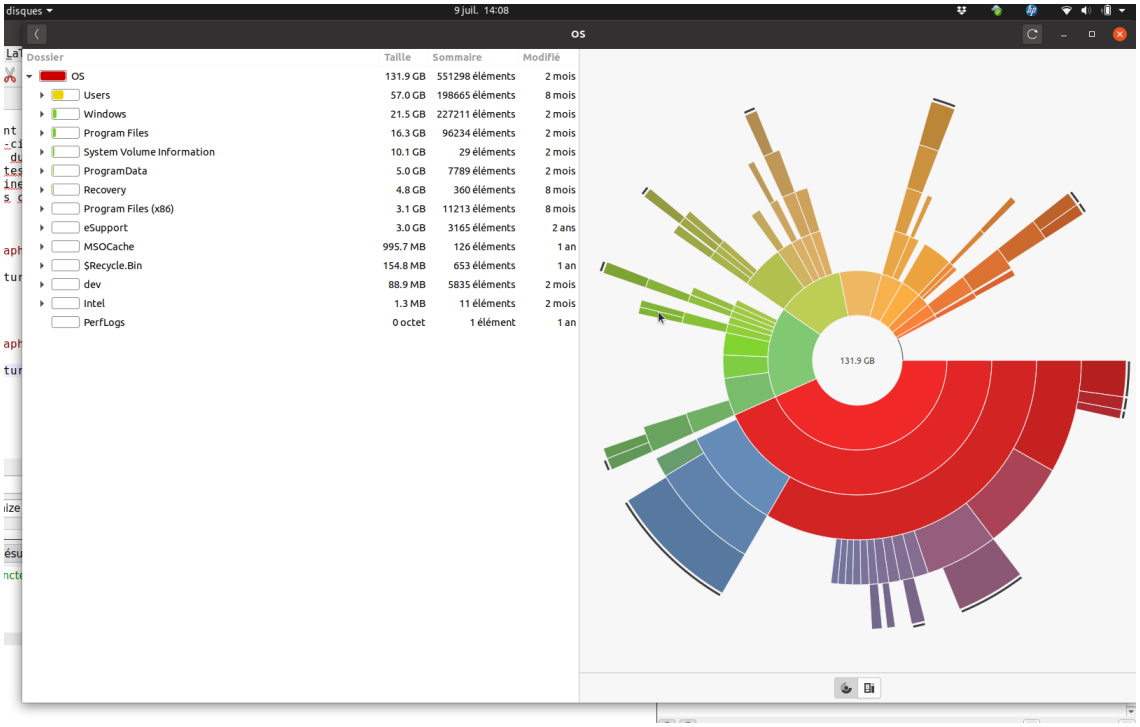


FIGURE C.2 – Structure du disque dur sous Windows

A ce niveau de la hiérarchie, la totalité des branches appartient par défaut au système. Cela signifie que vous ne pouvez pas modifier le contenu du tout : vous ne pouvez rien ajouter, retirer ou modifier. Notez aussi, qu'en haut de la figure, il apparaît un message d'alerte que **baobab** qui a été lancé à partir d'un compte utilisateur n'est pas en mesure d'explorer la totalité du disque dur car certaines parties ne sont lisibles ou accessibles qu'à l'administrateur, mais pas à un utilisateur lambda. Sur votre ordinateur personnel, vous avez la possibilité de passer en mode administrateur à partir de votre compte utilisateur. Il faut savoir qu'à partir du moment où vous êtes administrateur, vous pouvez corrompre votre système et donc prudence ! Sur les ordinateurs de l'UFR, vous n'avez pas ce privilège.

C.2.1 Linux

Examinons tout d'abord l'image du disque sous Linux :

- La racine est dénotée par le nom **/** Tous les fichiers et dossiers peuvent être identifiés à partir de cet élément
- Le premier élément (sur cet ordinateur et compte tenu que l'ordre d'apparition est relié à un tri par taux d'occupation est le répertoire **/home**. Les branches (non affichées sur cette figure) sont les répertoires racines de chaque utilisateur). Quand un utilisateur devient autorisé à se connecter sur un ordinateur, le système crée une branche avec le nom de connexion et cette branche ainsi que toutes les branches dérivées appartiennent par défaut à cet utilisateur. Cela donne la possibilité de créer une structure personnelle. Selon la manière dont le compte a été crée, la branche /home/-nom/nom_utilisateur appartient à chaque utilisateur mais elle est interdite en écriture aux autres et parfois interdite aussi à la lecture. Bien entendu, l'administrateur du système qui a le privilège de se connecter avec les droits identiques au système d'exploitation peut lire, modifier et supprimer une partie ou la totalité des fichiers et des branches de chaque utilisateur. Attention, quand vous disposez d'un ordinateur personnel, il peut être tentant de se mettre en mode administrateur pour avoir toute la liberté d'intervenir. Cela est un très mauvaise idée. Si vous êtes administrateur, vous pouvez modifier et détruire tous les fichiers y compris ceux du système. Le résultat est que sur un malentendu ou une fausse manœuvre, vous pouvez détruire des fichiers sans vous en rendre compte et cela devient catastrophique pour l'ordinateur. Si vous reprenez un programme dans lequel il y avait une erreur qui était interceptée par le système, en tant qu'administrateur du système, l'erreur pourra s'exécuter et votre ordinateur va avoir un système d'exploitation détruit ou amputé sévèrement. C'est la raison pour laquelle les systèmes récents vous demandent de passer en mode administrateur que pour des tâches bien spécifiques.
- Le répertoire suivant **/usr** appartient au système (ce qui signifie que vous ne pouvez modifier le contenu). Deux sous-répertoires **/usr/bin** et **/usr/lib** contiennent les fichiers exécutables du système ainsi que de nombreuses applications et les fichiers de bibliothèques respectivement.
- Le répertoire **/opt** est un répertoire où des applications sont installées à partir d'une seule branche. A travers le système des paquetages, lors de l'installation d'un logiciel, plusieurs fichiers sont installés et très souvent répartis entre plusieurs répertoires : (**/usr/bin/** pour l'exécutable **/usr/lib/** pour les bibliothèques **/usr/share/doc** pour la documentation,...)
- Le répertoire **/var** contient essentiellement (dans les sous-répertoires) les fichiers de log, c'est-dire les fichiers où les exécutables écrivent des messages qui correspondent, soit à des erreurs, soit simplement des commentaires liés à la réalisation de taches. Quand on analyse le fonctionnement d'un ordinateur, ces fichiers sont une source d'information très précieuse pour comprendre les problèmes ou pour collecter des informations de fonctionnement.
- Le répertoire **/lib** est un répertoire technique lié à la distribution et quasi-vide. Le vrai répertoire contenant les bibliothèques est le répertoire **/usr/lib/**.
- le répertoire **/bin** contient les exécutables du système, en particulier les utilitaires que nous discuterons de manière plus détaillée par la suite.

- Le répertoire **/sbin** contient les exécutables que seul un administrateur peut utiliser (protection de l'ordinateur avec la séparation des usages).
- Le répertoire **/etc** contient l'ensemble des fichiers de configuration du système. Il s'agit aussi d'un répertoire qui appartient aussi au système. Certains fichiers sont accessibles en lecture à l'utilisateur, mais pour des raisons de sécurité une autre partie ne l'est pas.
- Le répertoire **/lib32** est un répertoire technique lié à la distribution et quasi-vide. Le vrai répertoire contenant les bibliothèques 32 bits est le répertoire **/usr/lib32/**. Ces bibliothèques correspondent à des anciens programmes où les processeurs n'étaient capables que d'avoir des fichiers compilés pour des registres 32 bits. Les processeurs sont depuis plus d'une dizaine d'années des processeurs 64 bits et à terme ce répertoire disparaîtra. La plupart des logiciels ont été réécrits et recompilés. Toutefois, certains ne l'ont pas été, ce qui explique cette survie. A nouveau, cela est appelé à disparaitre prochainement.
- Le répertoire **/snap** est propre à Ubuntu (et quelques autres distributions) pour un système de paquetages.
- Le répertoire **/lost+found** est un répertoire où peuvent se trouver des fichiers récupérés, en cas de crash sévère du système.
- Le répertoire **/media** est un répertoire qui est crée dynamiquement quand on attache un périphérique au système (clé USB, disque dur supplémentaire)
- Le répertoire **/root** est le répertoire principal de l'administrateur
- Le répertoire **/cdrom** est le répertoire de rattachement lors d'un branchement d'un lecteur/graveur amovible. (Pour ceux qui disposent d'un ordinateur avec un graveur permanent, ce répertoire est en permanence attaché à ce graveur).
- Le répertoire **/srv** est un répertoire historique pouvant servir aux transferts de fichiers par ftp par exemple
- Le dernier répertoire **/mnt** est un répertoire réservé à l'administrateur pour monter un système de fichiers

Il serait très intéressant de poursuivre cette découverte de l'organisation de la structure du stockage du système en allant dans les branches successives, mais cela dépasse de beaucoup le cadre de ce cours. Notons que seul ce premier niveau ne contient que des branches, si on examine chacune des sous-branches, il y a a la fois des fichiers et des sous-répertoires.

Il y a deux leçons à tirer sur cette description hiérarchique du stockage. La complexité, le très grand nombre de fichiers et la gestion des accès a conduit à structurer l'organisation. Il est aussi recommandé à un utilisateur de suivre une telle démarche, c'est à dire de structurer l'organisation du stockage de ses fichiers. Par exemple, quand on développe un projet, on crée un nouveau répertoire. Chaque utilisateur a un répertoire "racine" à partir duquel il peut créer une structure. Il faut donc créer des sous-répertoires, pour éviter de transformer son lieu de stockage en chambre d'adolescent.

C.2.2 Windows

Nous allons répéter l'exercice précédent avec la structure que l'on a sous le système d'exploitation Windows.

- Le répertoire racine s'appelle **OS** parce qu'il s'agit d'une partition monté sous Linux. Sous Windows, il apparaît comme **c :**. Le séparateur sous Windows est \ et non pas **/** comme sous les systèmes Unix (Linux, MacOx, Android..)
- Le répertoire**Users** est le répertoire à partir duquel les répertoires utilisateurs sont créés. (l'équivalent de **/home** sous Linux)
- Le répertoire **Windows** contient l'ensemble du système Windows

- Le répertoire **Program Files** contient les applications installées.
- Le répertoire **System Volume Information** est un répertoire caché qui contient les éléments d'information et de restauration en cas de crash system
- Le répertoire **ProgramData** est un répertoire système utilisé en cas de restauration.
- Le répertoire **Recovery** est un répertoire système utilisé quand un répertoire qui a été perdu doit être restauré.
- Le répertoire **Program Files(X86)** contient les applications qui sont en mode 32bits, un répertoire appelé à disparaître sur le long terme.
- Le répertoire **eSupport** est un répertoire lié au constructeur de l'ordinateur
- Le répertoire **MSOCache** est un répertoire créé par l'installation de Microsoft Office, permettant la restauration de l'installation du logiciel en cas de crash.
- Le répertoire **$Recyle.bin** est encore un répertoire utilisé pour la restauration (Active Directory)
- Le répertoire **dev** a été créée lors de l'installation d'une bibliothèque
- Le répertoire **Intel** a été créée lors de l'installation d'un driver Intel
- Le répertoire **PerfLogs** est un répertoire pouvant contenir des fichiers enregistrant les performances de certains outils du système

A nouveau, on peut continuer la description en explorant chaque répertoire, mais cela dépasse le cadre de ce cours. On voit que l'on retrouve des points communs, mais des différences importantes. [1]

1. A titre personnel, Il est surprenant de voir autant de répertoires de récupération car bien que les plantages aient été très nombreux dans le passé sous Windows (il y a eu des progrès ces dernière années !), la récupération ne marche pratiquement jamais.

CHAPITRE D

Bibliographie Internet

Compte tenu de la forte activité autour du développement, il existe un grand nombre de documents sur internet qui permettent d'aller plus loin que ces notes d'introduction. On ne peut que recommander que d'aller les consulter afin de poursuivre l'apprentissage entamé par ce cours, et évidemment, pour avoir des explications supplémentaires aux différents éléments constituant ce cours. Par souci de simplicité, ces liens sont regroupés par groupe :

D.1 Interfaces graphiques pour le C++

- Site Codeblocks
- Site Visual Studio Code
- e Site Eclips

D.2 Interfaces graphiques pour le Python

- Site Spyder
- Site Jupyter (autre interface de Python)
- Site Pycharm (autre interface de Python)

D.3 Python3

- Site Python
- Site Numpy
- Site lecture notes de Scipy
- Site Scipy
- Site Matplotlib
- Site Apprentissage Python

D.4 C++

- Compilateur Gnu G++
- Compilateur Clang
- Compilateur Intel
- Site de la bibliothèque Eigen
- Site de la bibliothèque Boost
- Site Apprentissage C++

Table des matières