Programmation pour le calcul scientifique
Année : 2019-2020
Formation : L3 Ingénierie Mathématique

TP4 : Structure du code, make

1 Compilation séparée

Reprenez vos codes des exercices sur les matrices creuses et le solveur d'EDO et structurez les en fichiers en-tête `.hpp` et sources `.cpp`. Vous pourrez repartir des corrigés mis en ligne si vous n'avez pas de base de laquelle repartir. Compilez vos fichiers à la main et assurez vous du bon fonctionnement des programmes.

2 Un Makefile simple.

1. Écrire un `Makefile` pour les programmes de l'exercice précédent. Si vous n'êtes pas sûrs de ce que va faire la commande `make`, vous pouvez voir les commandes qui seront exécutées sans toutefois les lancer avec

```
$> make -n
```

2. Ajoutez une cible qui supprime les fichiers objets et les fichiers temporaires d'édition. Ajoutez une cible qui en plus de ces fichiers supprime l'exécutable.
3. Faites en sorte que chaque nom de fichier et chaque option ne soit écrit qu'à un seul endroit, au début du fichier [1].

3 Debug mode

Lorsque l'on développe un programme, il est en général pratique d'afficher plus d'information que nécessaire pour s'assurer du bon déroulement de celui-ci, ou d'effectuer des tests de sécurité (par exemple vérifier la taille de tableaux). Cependant, ces sont des opérations supplémentaires dont on n'a pas envie dans une utilisation normale du programme ou qui peuvent le ralentir considérablement.

On n'a cependant pas envie de passer son temps à éditer les fichiers sources pour commenter/décommenter les lignes correspondantes. Il est possible d'activer ou non ces instructions de façon très simple avec les directives de *preprocessing*.

1. Dans le code de résolution des EDOs, ajouter un affichage du numero de l'itération en cours qui sera placé entre deux directives

```
#ifdef DEBUG
// ... display some info here
#endif
```

1. C'est une pratique qui permet d'éviter de très nombreuses erreurs lors de modifications ultérieures. Elle n'est pas limitée à `make` : mieux vaut l'appliquer dans vos programmes, quel que soit le langage.

2. Ces instructions ne seront compilées que si la variable `DEBUG` est connue au *preprocess*. Normalement cela se fait à l'aide d'un `#define`, mais il est aussi possible de définir la variable dans la ligne de commande de compilation :

```
$> g++ -DDEBUG myFile.cpp -o myExe
```

3. Dans votre `Makefile`, créez les variables `OPTIM_FLAGS` et `DEBUG_FLAGS` qui contiennent les options de compilation correspondant à une compilation optimisée et une compilation en mode débogage.
4. Créez deux cibles différentes (qui auront les suffixes `_optim` et `_debug`) vous permettant de générer les éxecutables correspondant.

Quelques conseils :

- Développez toujours en mode *debug*. Le mode *release* n'est destiné qu'à l'utilisateur pour qu'il ait un exécutable plus léger et plus efficace.
- Ne passez en mode *release* que lorsque tous les tests que vous avez conçus sont validés.
- Il vaut mieux repartir de zéro lorsque vous changez de mode de compilation (cf exercice suivant). N'oubliez pas de faire un `make clean` entre deux versions.

4 Premiers pas avec `gdb`

`gdb` est un outil de débogage qui permet l'exécution d'un programme pas à pas pour aider à trouver les erreurs. Pour lancer `gdb`, il suffit de rentrer la commande suivante

```
$> gdb myExecutable
```

Assurez vous que l'exécutable a été compilé au préalable avec les symboles de débogage (option `-g`).

Une fois `gdb` lancé

- Vous pouvez ajouter des points d'arrêt (*breakpoint*) dans l'exécution du code avec

```
(gdb) break 42
(gdb) break fileName:126
```

Ici le programme se déroulera jusqu'à ce que la ligne de code 42 du fichier source principal soit atteinte. Vous pouvez préciser le nom d'un autre fichier où se trouve la ligne de code où s'arrêter.

```
(gdb) break myFunc
```

Le programme s'arrêtera à chaque appel de la fonction `myFunc`.
Vous pouvez voir la liste de tous les points d'arrêt que vous avez placés avec

```
(gdb) info break
```

Pour retirer un point d'arrêt

```
(gdb) clear 42
(gdb) clear fileName:126
(gdb) clear myFunc
```

- Démarrer/reprendre l'exécution :

```
(gdb) run <args>
```

Pour lancer le programme. Vous remplacerez `<args>` par les éventuels arguments de la ligne de commande qui aurait été utilisée.
Le programme se déroulera jusqu'au prochain point d'arrêt ou jusqu'au plantage. Une fois arrêté

```
(gdb) next
```

affiche et exécute la commande suivante après le point d'arrêt.

```
(gdb) step
```

est similaire à `next` mais s'arrête au premier appel de fonction.
Pour exécuter toutes les instructions jusqu'au prochain point d'arrêt, vous pouvez réutiliser

```
(gdb) run
```

- Examiner l'état des variables.
  Pour afficher le contenu d'une variable :

  ```
  (gdb) print myVar
  ```

  Si `myVar` n'existe pas (parce que vous êtes dans une fonction ou parce que vous êtes sortis du bloc où `myVar` est définie), vous aurez un message d'erreur.
- Savoir où on se trouve dans la hierarchie des appels. En général les codes appellent des fonctions à l'intérieur de fonctions, et ces appels imbriqués peuvent aller très loin en profondeur. Pour afficher où on se trouve dans la pile d'appels :

  ```
  (gdb) backtrace
  ```

  On peut remonter/redescendre dans la pile d'appels avec

  ```
  (gdb) up
  (gdb) down
  ```

N'hésitez pas à utiliser `gdb` ou un autre déboggeur de votre choix car ce sont des outils efficaces pour une tâche qui peut représenter une part importante du temps de développement.

Pour aller plus loin : un tutoriel pour utiliser `valgrind`, un outil qui permet de suivre l'état de la mémoire et détecter les éventuelles fuites qui peuvent amener à l'interruption du programme.

5 Arborescence classique

Dès qu'ils commencent à prendre de l'importance, les projets sont généralement structurés suivant une arborescence de ce type :

```
$> ls myProject
src/
doc/
folder1/
folder2/
...
Makefile
README
TODO
...
```

où `src` contient les fichiers sources et `doc` les fichiers qui génèrent la documentation du code. Les autres fichiers et dossiers facilitent la compilation, l'installation et l'utilisation du(des) programme(s). Lorsqu'on compile ce type de projets, une bonne pratique consiste à compiler dans un répertoire temporaire, qu'on appelle souvent `build`. Cela permet de préserver la structure des fichiers sources même si la compilation (ou le nettoyage) se passe mal.

Pour l'un des deux programmes précédents, placer vos fichiers sources dans un sous-dossier `src` et faites en sorte que `make` crée le sous-dossier `build`, s'y rende et y génère les fichiers nécessaires à la compilation. L'exécutable sera en revanche généré au niveau de la racine du projet (au même niveau que le `Makefile`, donc là où on aura tapé `"make"`). La cible `clean` supprimera le dossier `build`.

6 Espaces de nommage

1. Créez une structure `point3D` contenant les trois coordonnées cartésiennes (x, y, z). Écrivez une fonction `normalize` qui ramène ces points sur la sphère de rayon 1.
2. Écrivez une fonction `distance` qui calcule la distance euclidienne entre deux points normalisés.
3. Écrivez une fonction `distance` qui calcule la distance géodésique entre ces mêmes points. Il s'agit de calculer la longueur de l'arc sur la sphère entre les deux points. Indication : $(\boldsymbol{p_1}, \boldsymbol{p_2}) = \|\boldsymbol{p_1}\| \, \|\boldsymbol{p_2}\| \cos \alpha$.
4. Placer ces fonctions dans des `namespace` différents pour appeler la fonction désirée dans votre programme (par exemple, à la décision de l'utilisateur).