Compilation de code Fortran sur Unix

CNRS
17 novembre 2021

Lionel GUEZ
guez@lmd.ipsl.fr
Laboratoire de météorologie dynamique (UMR 8539)

Préambule (1/3)

- Ce cours ne porte pas sur :
  - le fonctionnement interne d'un compilateur ou la création d'un compilateur
  - le langage Fortran
- Ce cours porte sur les commandes et outils de compilation d'un code en Fortran, du simple point de vue d'un utilisateur de ces commandes et outils.

Préambule (2/3)

- La compilation de code Fortran a des spécificités mais une bonne partie de ce cours (~ 50 % ?) s'applique à la compilation de code dans d'autres langages.
- Unix s'entend au sens large de la famille Unix, incluant Linux et Mac OS.

Préambule (3/3)

- Une partie de ce cours (~ 20 % ?) doit s’appliquer à la compilation sur Windows. Notamment dans la discussion sur CMake. Mais l’enseignant a très peu d’expérience sur Windows.

Plan

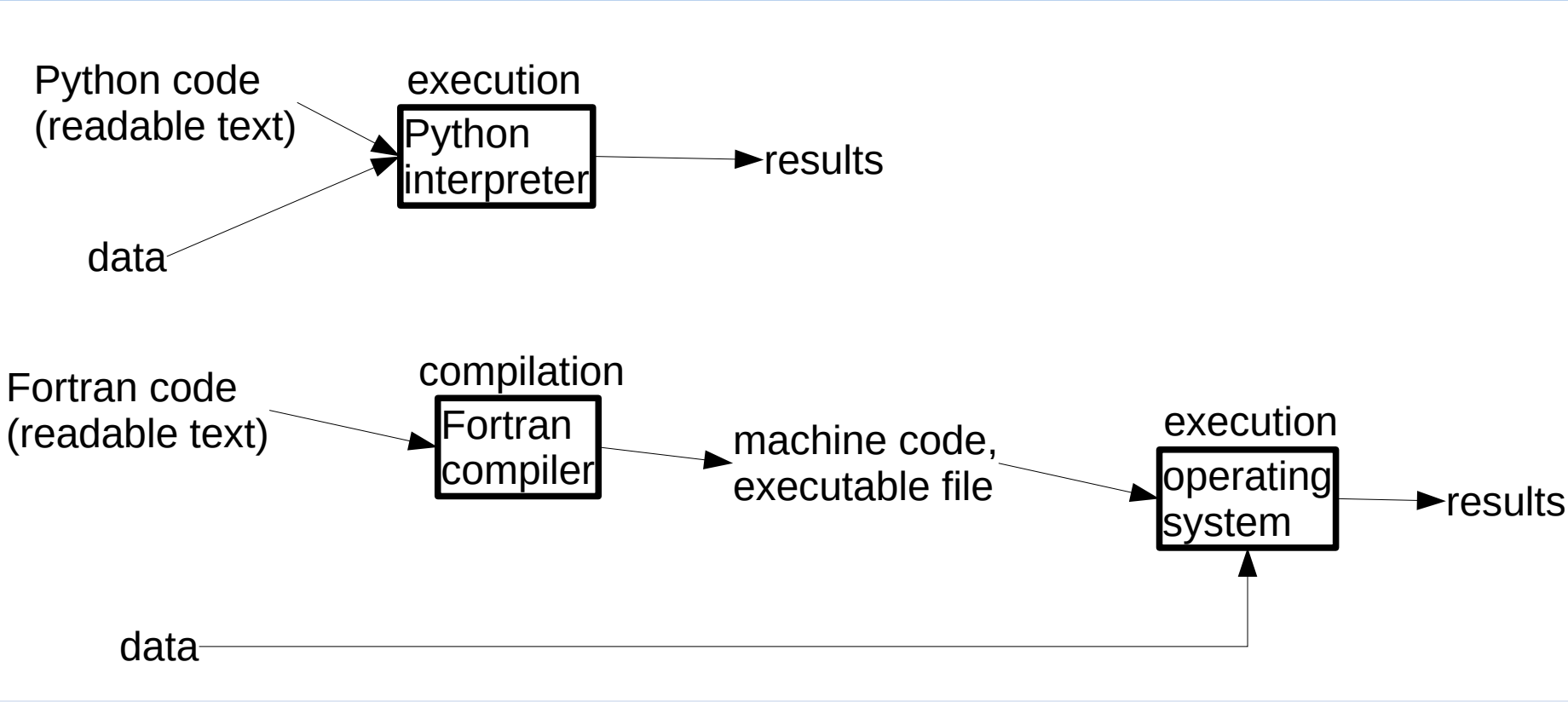
1) Introduction : la compilation
2) Invocation directe du compilateur
3) Invocation du compilateur dans un fichier shell
4) make
5) Au delà de make
6) CMake

1) Introduction : la compilation

What is compilation?

- Compilation is translation from a high-level computer language (meant to be read by a human being) to a low-level language (instructions meant to be read and executed by a machine). During compilation, errors, if any, in the grammar of the high-level language are signaled.
- The *compiler* is the piece of software that does the compilation.

Interpreted languages versus compiled languages (1/2)



Interpreted languages versus compiled languages (2/2)

- Interpreted : Python, Bash…
  Compiled : Fortran, C, C++…
- The compiler accomplishes a complex task, but it does it only once, producing an executable file that will work with any data. The interpreter accomplishes a simpler task, but has to do it again for each run with new data.
- In principle, code written in a compiled language should run faster than equivalent code written in an interpreted language.

If you want to know more about compilation

- Langages de programmation et compilation, Jean-Christophe Filliâtre. https://www.lri.fr/~filliatr/ens/compil
- Aho et al. Compilers: Principles, Techniques, and Tools, 2nd edition, Addison-Wesley, 2006

2) Invocation directe du compilateur

Basic case in Fortran

- Only one file to compile.
- Advice: do not use space, diacritical marks or any special character in the name of the file.
- La question du suffixe du nom du fichier : la plupart des compilateurs considèrent par défaut que `.f` signifie format fixe et `.f90` format libre.

Aside: free versus fixed source form

- Fixed source form: this was how Fortran programs were written in the old times (in Fortran 77 and before). Statements had to be written between columns 7 and 72. (There were also other formatting rules.)
- Free source form: appeared in Fortran 90. That is what we use now.
- We have to tell the compiler whether the file is supposed to be in fixed of free source form, either through the suffix of the filename or through a compilation option.

Manipulation 1

- Écrire un programme trivial (du type hello).
- Le compiler. Il faut connaître la commande qui invoque le compilateur : `gfortran` pour le compilateur GNU, ou `ifort` pour celui d'Intel, `nagfor` pour celui de NAG…
- L'exécuter.

Source file and executable file

- The file which contains the text of the Fortran program is called the ”source file”.
- Compilation translates Fortran into ”machine language”, that is the language of your processor. It creates an ”executable file”.

Insuffisant pour un programme non trivial

- Remarque : Fortran est devenu le langage des gros projets.
- La simple commande `gfortran plouf.f90` est insuffisante pour un programme non trivial, même contenu dans un seul fichier. Deux raisons :
  - Importance des options de débogage ou des options d'optimisation.
  - Options pour l'utilisation de bibliothèques.

Importance of compilation options (1/2)

- Do not miss the best of the compiler! You depend on the compiler for:
  - debugging your program
  - fast execution of your program
- However you cannot have both those services in a single compilation. Debugging a code and producing an efficient executable file are opposite requirements for the compiler. So you have to choose what you ask to the compiler.

Importance of compilation options (2/2)

- Two situations:
  - When you are developing your program: test it on small cases → use debuging options of the compiler.
  - When you are satisfied with your code and want to produce results for really useful cases → use compilation options producing a fast executable.

Options for debugging (1/2)

- To produce warnings for valid but suspicious statements: maybe not what you intended. For example: type conversion, truncation of a string, underflow in floating point computation, statement that can never be reached.
- To warn of ”bad” programming style: using an obsolete feature of the language, using a language extension (this extension may not work with another compiler).

Options for debugging (2/2)

- To detect errors: invalid floating point operation (division by zero, square root of a negative number, etc.), referring to an out-of-bounds subscript of an array, using in an expression a variable which has not yet been defined.

A lot of debugging options

- Example for a given version of gfortran:
  `-std=f2003 -pedantic-errors -Wall -Wconversion -Wimplicit-interface -Wunderflow -Wextra -Wunreachable-code -ffpe-trap=invalid,zero,overflow,underflow -g3 -fbounds-check -O0 -fstack-protector-all`
- `man gfortran` to see the list of compilation options (debugging options and others) with their descriptions.

Debugging options depend on the compiler

- Roughly the same functionality for debugging in all compilers but the names of the options (what you actually type on the command line) change with the compiler.
- Moreover, for a given compiler, options change a little with the version of the compiler.
- Advice: keep a list of debugging options for each major version of each compiler that you use. (A major version is usually released once per year or less frequently.)

Définition d'une bibliothèque

- Un ensemble de procédures en Fortran
- Pas de programme principal
  - La bibliothèque ne s'exécute pas toute seule. Elle s'utilise dans un programme extérieur à la bibliothèque.
- La bibliothèque s'utilise sous une forme déjà compilée.

Intérêt d'une bibliothèque (1/2)

- Une bibliothèque contient des procédures qui ont vocation à être utilisées dans différents programmes.
- Les procédures réutilisables sont ainsi regroupées. En général : les procédures d’une bibliothèque ont des contenus liés. Par exemple : bibliothèque Lapack pour l’algèbre linéaire ; bibliothèque FFTW pour la transformation de Fourier.

Intérêt d’une bibliothèque (2/2)

- Pas de duplication des fichiers sources.
- Pas de re-compilation pour chaque programme utilisateur.

3) Invocation du compilateur dans un fichier shell

Une longue commande de compilation

- Beaucoup d’options de débogage ou d’optimisation.
- Des options qui référencent des bibliothèques.
- Comment ne pas retaper une très longue commande ?
  - Rappel de commande ? Insuffisant d’une session à l’autre. Pénible de chercher dans l’historique.
  - Il faut donc stocker la commande.

Stockage de la commande telle quelle dans un fichier shell

- C’est l’idée la plus simple.
- Où ?
  - En commentaire dans le fichier source ?
    - Pas pratique : copier-coller nécessaire à chaque session.
    - Pas logique : information spécifique à un compilateur et une machine avec le code source Fortran indépendant du compilateur.
  - Mieux : dans un fichier (shell) à part.

Manipulation 2

- Créer un fichier contenant la commande de compilation du programme trivial.
- Ajouter les options de débogage.
- Se servir de ce fichier pour compiler.

Programme à plusieurs fichiers

- Intérêt de répartir le code sur plusieurs fichiers pour : la clarté ; la gestion de versions ; le travail à plusieurs ; diminuer le temps de compilation (cf. infra).
- Possibilité de compiler d'un coup tous les fichiers :
  `gfortran` [options] `foo1.f90 foo2.f90` …
  mais ce n'est pas conseillé (cf. infra).

Décomposition de la compilation (1/3)

- Possibilité de décomposer la compilation en plusieurs commandes :

```
gfortran ... -c foo1.f90
gfortran ... -c foo2.f90
gfortran ... foo1.o foo2.o
```

  Ouvre la perspective de ne recompiler que le nécessaire et donc de gagner en temps de compilation.

Décomposition de la compilation (2/3)

- La compilation d'un fichier source seul, qui est une partie seulement d'un programme, produit un "fichier objet", en langage machine, non exécutable, suffixe `.o`.
- La création du fichier exécutable à partir des fichiers objets et des bibliothèques s'appelle l'édition de liens.
- Les fichiers `.o` sont des intermédiaires produits par la compilation. Ils ne sont pas utiles à l'exécution du programme.

Décomposition de la compilation (3/3)

- Option supplémentaire `-c` (tous les compilateurs) pour la compilation d'un fichier qui est une partie seulement d'un programme (sans édition de liens).
- L'idée pour réduire le temps de compilation, après modification d'un fichier, est donc de ne re-compiler que ce fichier et refaire l'édition de liens.

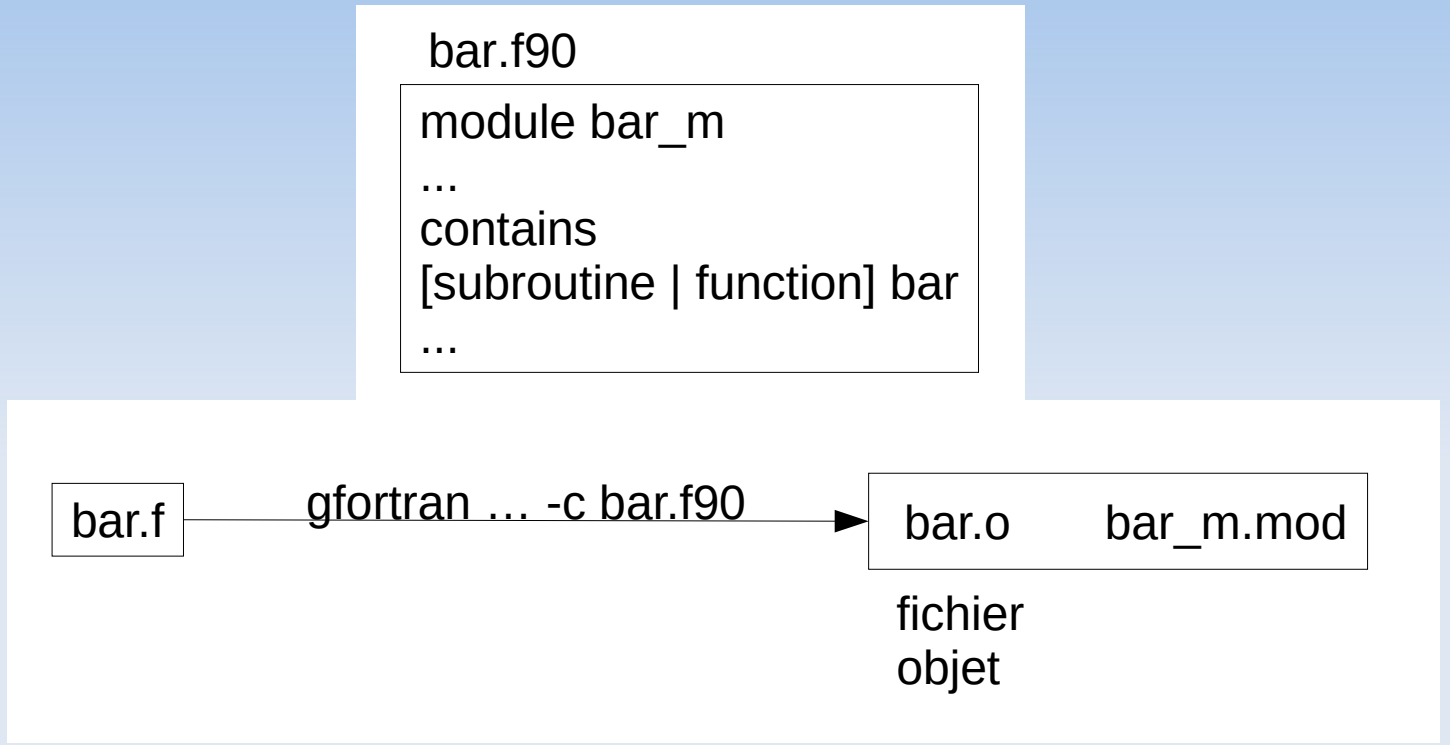
Ordre de compilation (1/2)

- L’utilisation de modules en Fortran implique un ordre de compilation. Ordre entre les fichiers s’il y a plusieurs fichiers, ou ordre d’écriture des modules à l’intérieur d’un fichier.

Ordre de compilation (2/2)

- S'il y a :
  `use bar_m`
  dans la procédure `foo` alors le module `bar_m` doit être compilé avant la procédure `foo` (avant le module contenant la procédure `foo`).
  - Si `bar_m` et `foo` sont dans le même fichier alors `bar_m` doit être avant `foo` dans ce fichier.
  - Sinon le fichier contenant `bar_m` doit être compilé d'abord.

Fichier `.mod` (1/3)



- `bar_m.mod` si le module s'appelle `bar_m` (le nom du fichier `.mod` n'est pas formé à partir du nom du fichier `.f90`)

Fichier `.mod` (2/3)

- `bar_m.mod` contient (sous forme lisible seulement par le compilateur) les informations sur l'interface du module, y compris l'interface des procédures publiques du module (`function` ou `subroutine`, type des arguments, scalaires ou tableaux, `intent`, etc.).

Fichier `.mod` (3/3)

- S'il y a :
  `use bar_m`
  dans la procédure `foo` alors :
  `gfortran … -c foo.f90`
  a besoin de `bar_m.mod`. D'où la contrainte d'ordre de compilation.
- Les fichiers `.mod` sont, comme les fichiers `.o`, des intermédiaires produits par la compilation. Ils ne sont pas utiles à l'exécution du programme.

Graphe de dépendances (1/2)

- En regardant les fichiers sources, on peut déduire une suite de dépendances :

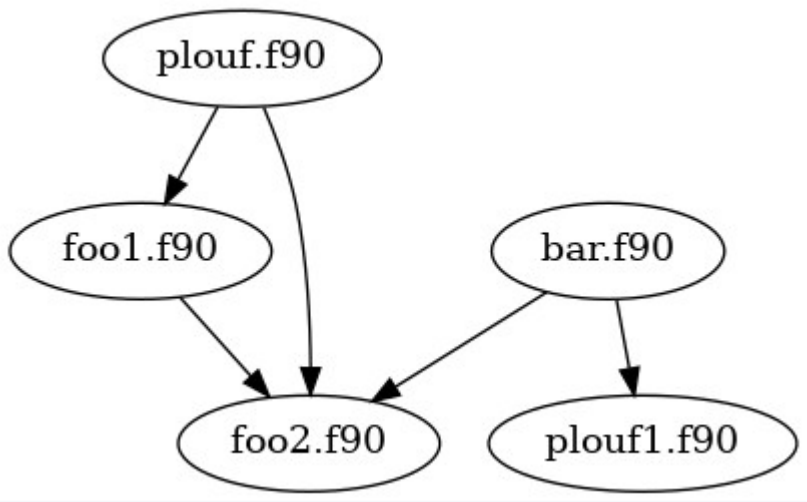
```
foo1.f90 → foo2.f90, bar.f90
```

(→ signifie ”doit être compilé après”)

```
bar.f90 → foo2.f90, plouf1.f90
plouf.f90 → foo2.f90, foo1.f90
```

Graphe de dépendances (2/2)

- Mais il reste une analyse à faire pour trouver l'ordre de compilation : créer le graphe des dépendances (un graphe directionnel acyclique) et choisir un parcours de ce graphe.



Nécessité d’un outil

Le simple listage des commandes de compilations telles quelles dans un shell est donc insuffisant.

- Ne permet pas de recompiler uniquement le nécessaire
- Ne permet pas de trouver l’ordre de compilation
- ```
  gfortran ... -c foo1.f90
  gfortran ... -c foo2.f90
  ```
  Accessoirement : idée de stocker les options de compilation dans une variable pour ne pas les répéter.
  ```

4) make

make

- Comment trouver l'ordre de compilation et ne re-compiler que le nécessaire : `make`.
- `make` is a UNIX utility (just as `ls`, `cd`, `awk`…).
- Its main intended use is as a compilation tool.
  - Note: `make` is not especially for Fortran, it is designed as a tool for any compiled language.
- À installer avec le gestionnaire de paquets de votre distribution Linux. Sur Ubuntu :
  `apt install make make-doc`

Implémentations concurrentes de make (1/2)

- make date de 1976, plusieurs implémentations concurrentes de make ont été développées. Le langage de make dépend un peu de l'implémentation.
- Nous voulons écrire un fichier pour make qui soit portable.

Implémentations concurrentes de make (2/2)

- Solution : viser une implémentation de make particulière, GNU make. GNU make est aujourd'hui disponible sur toutes les machines Unix. GNU make est d'ailleurs l'implémentation qui a le plus de fonctionnalités.

GNUmakefile (1/2)

- Au lieu de stocker dans un fichier les commandes de compilation telles quelles, on stocke dans un fichier l'information nécessaire pour créer les commandes de compilation, dans le langage de `make`.
- Les différentes implémentations de `make` cherchent par défaut un fichier appelé `makefile` ou `Makefile`. Seul GNU `make` cherche aussi un fichier appelé `GNUmakefile`.

GNUmakefile (2/2)

- Pour être sûr d’utiliser GNU `make`, choisir le nom `GNUmakefile`.

Compilation d’un programme à un seul fichier avec `make`

Même si `make` donne la pleine mesure de son utilité pour un programme à plusieurs fichiers, reprenons le cas d’un programme à un seul fichier.

The variables that `make` uses (1/2)

- `FC`: initials of *Fortran compiler*, that is the compiling command
  Examples :
  ```
  FC = gfortran
  ```
  ou
  ```
  FC = ifort
  ```
  (Nota bene : les espaces de part et d'autre du signe égal n'ont pas d'importance dans le langage de `make`, à la différence d'un shell.)

The variables that `make` uses (2/2)

- `FFLAGS`: *Fortran flags*, those are compilation options.

Suffixe `.f` ou `.f90` ? (1/2)

- Nous avons des fichiers au format libre. Quel suffixe choisir ?
- Désaccord entre `make` et les compilateurs :
  - `make` sait traiter comme un grand les fichiers avec le suffixe `.f` et non `.f90`.
  - Les compilateurs interprètent en général `.f` comme format fixe.

Suffixe `.f` ou `.f90` ? (2/2)

- Deux solutions :
  - apprendre à `make` comment traiter les fichiers `.f90`
  - ou bien mettre le suffixe `.f` et ajouter une option de compilation qui spécifie le format libre

le plus simple

Manipulation 3

- Créer un fichier `GNUmakefile`.
- Compiler le programme trivial avec `make`. Par exemple :
  `make plouf`
  si votre source s'appelle `plouf.f`.
- Ré-exécuter `make plouf`.
- Modifier le source et ré-exécuter `make plouf`.

Nota bene (1/2)

- `make` affiche la commande de compilation qu'il a créée à partir du `GNUmakefile` :
  `$(FC) $(FFLAGS) plouf.f -o plouf`
- `make` utilise les suffixes. On lui demande de créer `plouf` et il voit qu'il existe un fichier `plouf.f` présent dans le répertoire : il comprend qu'on veut créer l'exécutable `plouf` à partir du source Fortran `plouf.f`.
- `make` utilise les dates de modification des fichiers.

Nota bene (2/2)

- Les fichiers `GNUmakefile` et `plouf.f` doivent être présents dans le répertoire courant.

Cible dans le GNUmakefile

- Possibilité de spécifier la cible, `plouf`, dans le `GNUmakefile`. Ajouter la ligne :
  `plouf:`
- `make` tout court tente de créer ou mettre à jour la première cible rencontrée dans le `GNUmakefile`, inutile d'écrire `make plouf` sur la ligne de commande.

Manipulation 4

- Ajouter la cible dans le `GNUmakefile`.
- Compiler en invoquant simplement `make`, sans argument.

Compilation d’un programme à plusieurs fichiers avec `make`

- Informations supplémentaires à fournir à `make` (par rapport au cas d’un seul fichier) : la liste des fichiers à compiler ; les contraintes d’ordre de compilation.
- Ces informations sont codées sous forme de ”dépendances”.

Notion de dépendance dans `make`

Une dépendance dans `make` :

```
cible: prérequis1 prérequis2 ...
```

Ce qui signifie :

- Les prérequis doivent exister pour pouvoir créer ou mettre à jour la cible.
- La cible, si elle existe, n'est pas à jour si un prérequis a été modifié plus récemment que la cible.
- Pour créer ou mettre la cible à jour, il faut d'abord créer ou mettre à jour les prérequis, et, récursivement, les prérequis des prérequis.

Dépendance pour l'édition de liens (1/2)

- Écrire que l'exécutable dépend de tous les objets. Exemple :

```
my_program: foo.o bar.o my_program.o
```

  - Y compris l'objet correspondant au programme principal
  - Tous les objets, même si le programme principal n'utilise pas directement tous les modules
  - Utiliser comme cible le nom du fichier contenant le programme principal, sans suffixe.

Dépendance pour l'édition de liens (2/2)

- Comme l'un des prérequis est `my_program.o`, c'est-à-dire le nom de la cible avec un suffixe `.o`, make comprend qu'il doit faire une édition de liens pour créer la cible à partir de tous les prérequis.
- Nota bene : nous avons ainsi implicitement donné la liste des fichiers à compiler, ce sont les fichiers obtenus en remplaçant `.o` par `.f`.

La commande d’édition de liens

make connaît la commande d'édition de liens, mais il utilise par défaut le compilateur C. Il faut lui dire d'utiliser le compilateur Fortran pour l'édition de liens :

```
LINK.o = $(FC) $(LDFLAGS) $(TARGET_ARCH)
```

Notion de règle dans make

- Une règle est l'association d'une dépendance et d'une recette (une commande) :

```
cible : prérequis
  recette
```

  La recette est censée permettre de créer (ou mettre à jour) la cible.

- Attention : la recette doit être précédée d'une tabulation et non d'espaces.

Règles implicites

- `make` connaît déjà certaines règles utilisant les suffixes des fichiers : règles implicites, ou encore *pattern rules*.
- Les régles implicites utilisent les variables automatiques :
  - `$^` : tous les pré-requis
  - `$<` : premier pré-requis
  - `$@` : cible

Manipulation 5 : la base de données de make

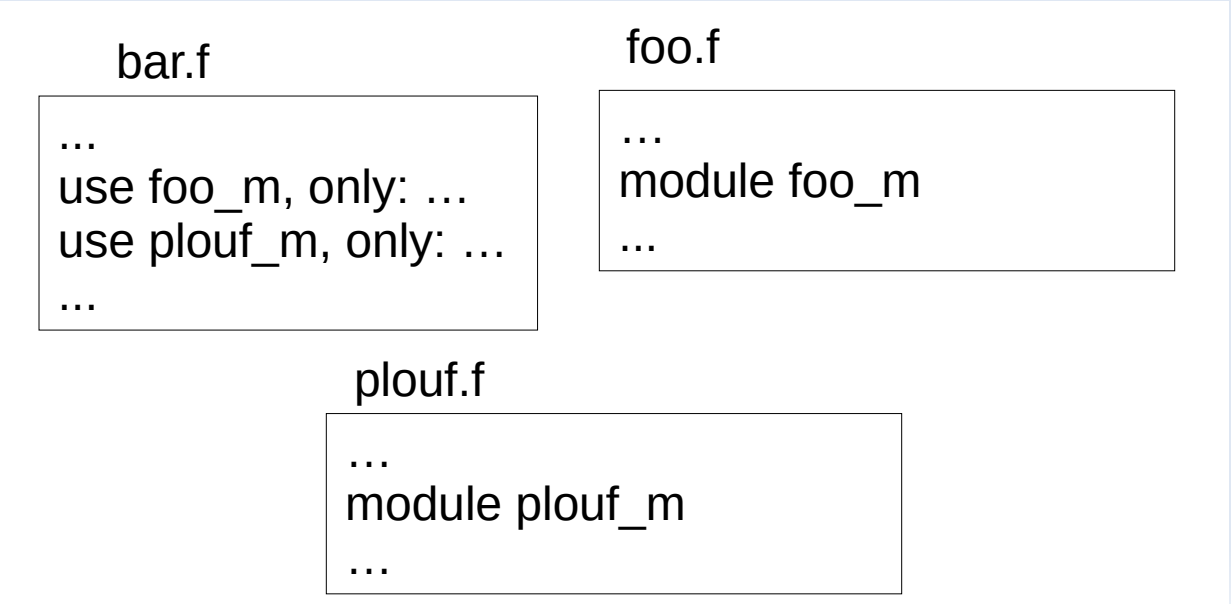
- `man make`
- Afficher la base de données de make :
  `make -p -f/dev/null >make_db.txt`
- Regarder en particulier :
  - la définition de `LINK.o`
  - la commande d'édition de liens, dans la règle `%: %.o`
  - la commande de compilation, dans la règle `%.o: %.f`

Dépendances entre les objets

Indiquer les dépendances induites par l'utilisation de modules Fortran en écrivant des dépendances entre les objets. Exemple :

```
bar.o: foo.o plouf.o
```

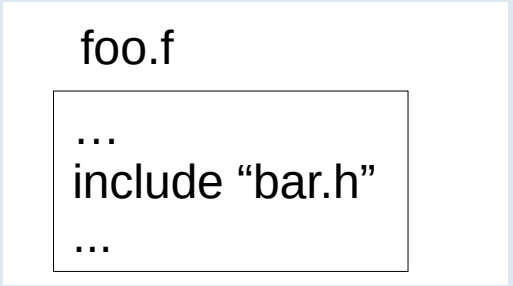
indique bien que `bar.f` doit être compilé (ou re-compilé) après `foo.f` et `plouf.f`



Lignes `include` en Fortran

- Si un source Fortran contient une ligne include alors il faut tenir compte de cette dépendance dans le makefile : on ajoute le fichier inclus dans les prérequis de l'objet.
- Exemple :

```
foo.o: bar.h
```



Forme compilée d'une bibliothèque statique

La bibliothèque "statique" compilée se présente sous la forme de :

- Un fichier “archive”, dont le nom commence par “lib” et dont le suffixe est `.a`.
  Exemple : `libnr_util.a`
  Plus pratique que l'ensemble de fichiers objets (`.o`), contient cet ensemble.
- Un ou plusieurs fichiers `.mod`.
  Exemple : `nr_util.mod`

Compilation d'un programme utilisant une bibliothèque (1/2)

- Le compilateur a besoin des fichiers `.mod` de la bibliothèque au moment de la compilation des objets. Trouver le répertoire contenant les `.mod` de la bibliothèque.
- Dans le GNUmakefile, compléter `FFLAGS` avec l'option supplémentaire :
  `-I`répertoire
  où " répertoire " est le répertoire trouvé à l'étape précédente.

Compilation d'un programme utilisant une bibliothèque (2/2)

- Le compilateur a besoin de l'archive `lib….a` au moment de l'édition de liens. Trouver le répertoire contenant cette archive.
- make utilise la variable `LDLIBS` à l'édition de liens. Cf. la règle :
  `%: %.o`
  Exemple :ajouter dans le `GNUmakefile` la ligne :
  `LDLIBS = -L`répertoire `-lnr_util`
  où " répertoire " est le répertoire contenant `libnr_util.a`.

Utilisation de plusieurs bibliothèques (1/2)

- Les archives et les `.mod` des bibliothèques peuvent se trouver dans plusieurs répertoires. On ajoute autant d'options `-L` et `-I` que nécessaire.
- L'ordre des options `-L` et `-I` n'a pas d'importance.
- Une option `-l` pour chaque bibliothèque.

Utilisation de plusieurs bibliothèques (2/2)

- Si une bibliothèque fait appel à une autre bibliothèque alors l'ordre des options `-l` est important : les bibliothèques doivent apparaître dans l'ordre "appelant appelé".
  Exemple : si la bibliothèque plouf1 appelle plouf2 alors `-l`plouf1 doit être avant `-l`plouf2 dans `LDLIBS`.

Manipulation 6

- Écrire un makefile pour le programme Coriolis sachant que :
  - le programme utilise les bibliothèques Numer_Rec_95 et Jumble
  - les bibliothèques sont dans `~/.local`
  - Numer_Rec_95 utilise Jumble
  - L'unité de programme principale est dans le fichier `coriolis.f`
- Compiler
- `touch coriolis.f`
  Recompiler.

Nota bene

- Mettre la dépendance de l’exécutable avant les dépendances entre les objets. Rappel : make traite par défaut la première cible rencontrée.
- La simple commande make produit une série de commandes : compilations des objets, édition de liens.
- make a déduit un ordre de compilation à partir des dépendances entre objets.
- make ne recompile que le nécessaire.

Génération automatique des dépendances (1/2)

- make ne regarde pas l'intérieur des fichiers, ne connaît pas les langages de programmation, ne peut donc pas générer tout seul les dépendances entre objets induites par l'utilisation de modules Fortran et les dépendances induites par les include Fortran. Nécessité de lui adjoindre un autre outil.

Génération automatique des dépendances (2/2)

- makedepf90
  Dépôt des sources de makedepf90
  Installer via le gestionnaire de paquet de votre distribution Linux. Sur Ubuntu :
  ```
  apt install makedepf90
  ```

Manipulation 7

- Avec le programme Coriolis, sur la ligne de commande :

```
makedepf90 -free -Wmissing -Wconfused \
    -nosrc -u numer_rec_95 -u jumble \
    -u nr_util *.f
```

- Nota bene : l'option `-u` indique qu'un module doit être ignoré. Elle est utilisée pour un module de bibliothèque, qui ne fait donc pas partie des sources.

Combinaison de make et makedepf90 (1/6)

- Idée : nous redirigeons la sortie standard de makedepf90 vers un fichier, que nous appelons `depend.mk`, et nous faisons référence à ce fichier dans le makefile :
  `include depend.mk`

Combinaison de make et makedepf90 (2/6)

- Il est inutile de re-créer la liste des dépendances lorsqu'on modifie le moindre caractère dans un fichier source. Nous voulons donc que `depend.mk` ne soit créé que lorsqu'il n'existe pas ou sur demande explicite de l'utilisateur (dans le cas où une nouvelle instruction `use` ou une ligne `include` est insérée dans le programme).

Combinaison de make et makedepf90 (3/6)

- Si nous mettons les fichiers sources en prérequis de la liste des dépendances :

  `sources =` liste des fichiers `.f`

```
depend.mk: ${sources}
    makedepf90 ${sources} >depend.mk
```

  alors le fichier `depend.mk` est recréé automatiquement à chaque modification d'un fichier source. Ce n'est pas ce que nous voulons.

Combinaison de make et makedepf90 (4/6)

- Il faut donc une règle sans prérequis :

```
depend.mk:
    makedepf90 ${sources} >depend.mk
```

Permet bien la création automatique de `depend.mk` s'il n'existe pas. Mais pas la mise à jour. Si `depend.mk` existe, la commande :

```
make depend.mk
```

donne :

```
make: « depend.mk » est à jour.
```

Digression : notion de cible fictive dans make

- Une cible fictive (*phony target*) ne désigne pas un fichier.
- Si make essaie de fabriquer cette cible, il la considère donc toujours non à jour, et exécute la recette correspondante.
- make ne perd pas de temps à chercher une règle implicite qui permettrait de créer cette cible.
- Syntaxe :

```
.PHONY: nom de la cible
```

Combinaison de make et makedepf90 (5/6)

- Avec seulement une cible fictive :

```
.PHONY: depend
depend:
   makedepf90 ${sources} >depend.mk
```

`depend.mk` n'est pas créé automatiquement lorsqu'il n'existe pas. Le makefile ne marche pas. Il affiche simplement le message d'erreur :

```
GNUmakefile:ligne: depend.mk: No such file or directory
make: *** Pas de règle pour fabriquer la cible «
depend.mk ». Arrêt.
```

Combinaison de make et makedepf90 (6/6)

- Il faut donc les deux cibles, réelle et fictive :

```
.PHONY: depend
depend depend.mk:
    makedepf90 [options] ${sources} >depend.mk
```

- Rappel : s'assurer que l'exécutable reste la première cible du makefile.

Création de la liste des objets à partir de la liste des sources (1/2)

- Nous avons besoin de la liste des objets comme prérequis de l'exécutable et de la liste des sources comme prérequis de depend.
- Ce sont les mêmes listes au suffixe près. Ne pas les dupliquer.
- Utiliser une fonctionnalité de make qui modifie la valeur d'une variable :

```
sources = liste des fichiers .f
objects := $(sources:.f=.o)
```

Création de la liste des objets à partir de la liste des sources (2/2)

- Nota bene : même s'il existe des fichiers inclus par une ligne `include` Fortran, ils n'ont pas à apparaître dans la variable sources.

Manipulation 8

Générer automatiquement les dépendances dans le makefile du programme Coriolis. Tester la création automatique et la mise à jour.

Règle `clean` (1/2)

- Convention : la cible `clean` existe dans la plupart des makefiles et permet d'effacer les produits du makefile. `clean` ne devrait effacer *que* ce que le makefile est capable de regénérer. Ne pas utiliser donc de wildcard dans la recette.

```
.PHONY: clean
clean:
    rm -f nom de l'exécutable ${objects}
```

Règle `clean` (2/2)

- Nota bene : make n'a pas la liste des `.mod`, il n'y a donc pas de moyen simple de les supprimer dans `clean`. Mais ce n'est pas grave : si on veut s'assurer de tout recompiler (si par exemple on a changé une option de compilation), clean répond bien à ce besoin.
- Rappel : s'assurer que l'exécutable reste la première cible du makefile.

Manipulation 9

Ajouter une règle clean au makefile du programme Coriolis. Tester.

Pré-processeur

- Possibilité d’inclure dans le code Fortran des lignes destinées au pré-processeur du langage C.
- Le pré-processeur passe avant la compilation proprement dite. Il filtre des lignes de texte ou remplace des chaînes de caractères.

Pré-processeur : exemples

- ```
  #ifdef HAVE_MPI
      use mpi_f08
  #endif
  ```
  Une option de compilation dit si la ”macro” `HAVE_MPI` est définie ou non.
- ```
  integer, parameter:: wp = CPP_WP
  ! working precision for real type
  real(wp) ...
  ```
  Une option de compilation dit par quoi la macro `CPP_WP` doit être remplacée.

Pré-processeur : impact sur le système de compilation (1/2)

- La possibilité d'utiliser le pré-processeur ne change presque rien au système de compilation.
- Essentiellement, la question se règle via les suffixes des noms de fichiers et des options de compilation supplémentaires.
- Le suffixe `.F` (ou `.F90`) au lieu de `.f` (ou `.f90`) indique au compilateur que le fichier doit passer par le pré-processeur.

Pré-processeur : impact sur le système de compilation (2/2)

- Si le programme est constitué d'un mélange de fichiers .f et .F alors la création de la liste des objets par :

```
objects := $(sources:.f=.o)
```

ou bien :

```
objects := $(sources:.F=.o)
```

ne marche plus. Utiliser les fonctions `addsuffix` et `basename` de make :

```
objects := $(addsuffix .o, $(basename ${sources}))
```

Options de définition de macros (1/2)

- make reconnaît la variable `CPPFLAGS` comme contenant les options à passer au préprocesseur. Cf. dans la base de données de make la règle `%.o: %.F` et la variable `compile.F`, à comparer à `%.o: %.f` et la variable `compile.f`.

Options de définition de macros (2/2)

- L'option définissant une macro est (probablement tous les compilateurs) `-D`.
  Exemples :
  ```
  -DHAVE_MPI
  -DCPP_WP='kind(0.)'
  ```
- makedepf90 a aussi l'option `-D`. Il suffit donc en général de passer `${CPPFLAGS}` à makedepf90.

Compilation d'une bibliothèque statique (1/2)

- Au lieu de créer un exécutable à partir des objets, il faut créer l'archive `lib....a`. Le programme qui mène à bien cette opération est l'utilitaire Unix `ar`.
- make connaît cette opération. Il faut juste lui indiquer quels objets l'archive doit contenir : `lib....a: lib....a(`liste des objets`)`
  (Analogue à la ligne de dépendance d'un exécutable.)

Compilation d'une bibliothèque statique (2/2)

- Nota bene : il n’y a pas d’édition de liens lors de la compilation d’une bibliothèque. La variable `LDLIBS` n’a donc pas à être définie.

Mode déterministe ou non de `ar` (1/2)

- Selon le système de votre machine, il est possible que `ar` fonctionne par défaut en mode déterministe : il re-crée la bibliothèque à chaque invocation, même si les objets n’ont pas changé.

Mode déterministe ou non de `ar` (2/2)

- Si l'on veut éviter ce comportement, on peut passer une option à `ar`. make reconnaît la variable `ARFLAGS`. Sa valeur par défaut est :
  `ARFLAGS = rv`
  Cf. base de données de make.
- `ARFLAGS = rvU`
  demande à `ar` de passer en mode non déterministe.

Bibliothèque statique versus bibliothèque dynamique (1/3)

- À l'édition de liens d'un exécutable avec une bibliothèque statique, le code de la bibliothèque est copié dans le fichier exécutable, qui est donc auto-suffisant.
- Il est aussi possible de créer et faire l'édition de liens avec une bibliothèque "dynamique", ou encore "partagée".
  - Le code de la bibliothèque n'est pas dans l'exécutable.

Bibliothèque statique versus bibliothèque dynamique (2/3)

- Le code de la bibliothéque est chargé au moment de l'exécution.
- Le suffixe est `.so` au lieu de `.a`.
- L'outil qui crée la bibliothèque dynamique n'est pas `ar` comme pour une bibliothèque statique, c'est le compilateur.

- Les bibliothèques installées par votre système sont souvent dynamiques : économie de place ; la mise à jour ne nécessite pas de recompilation du code utilisateur.

Bibliothèque statique versus bibliothèque dynamique (3/3)

- Pour vos propres bibliothèques, compilez plutôt des bibliothèques statiques. Intérêt pour la reproductibilité.

Manipulation 10 (1/3)

- Écrire le makefile pour la bibliothèque `NR_util` sachant que :
  - Cette bibliothèque n'utilise aucune autre bibliothèque.
  - Un des fichiers doit passer par le préprocesseur et la macro `CCP_WP` doit être définie. Choisir par exemple :
    `CPP_WP='kind(0.)'`
  - Les fichiers `test_ifirstloc.f` et `test_nr_util.f` contiennent des programmes principaux de test et ne font donc pas partie de la bibliothèque.

Manipulation 10 (2/3)

- Tester.
- Nota bene : les sources Fortran contiennent des lignes include de fichiers `.h` mais il n'y a rien de plus à faire dans le makefile. makedepf90 tient compte de ces `.h` automatiquement. Cf. le fichier `depend.mk` créé.
- Nota bene : l'option `CPPFLAGS` n'est utilisée que pour le fichier `nrtype.F`. Cf. les commandes créées par make.

Manipulation 10 (3/3)

- Passer en mode non déterministe de `ar` et tester.

Plusieurs cibles (non fictives)

- make ne limite pas le nombre de cibles (exécutables, bibliothèques etc.) insérées dans un makefile.
- Convention : créer une cible fictive `all` qui regroupe les principales cibles du makefile. Cette cible devrait être la première du makefile.

Manipulation 11

- Ajouter dans le makefile de `NR_util` les cibles pour les programmes de test : faire dépendre les exécutables de la bibliothèque.
- Ajouter une cible fictive `all`.
- Ne pas oublier d’ajouter les exécutables tests dans la recette de `clean`.

Plusieurs répertoires sources (1/3)

- Cas où les fichiers composant le programme ou la bibliothèque sont répartis dans plusieurs répertoires.
- Peut être traité simplement avec make à condition qu'il n'y ait pas de fichiers homonymes.
- Créer le makefile à la racine de l'arborescence du code.

Plusieurs répertoires sources (2/3)

- Lister les répertoires et sous-répertoires relatifs à la racine de l'arborescence du code (sans inclure le répertoire racine), séparés par des espaces, dans la variable VPATH. VPATH pour *view path*.
- Les produits de la compilation sont créés à la racine de l'arborescence.

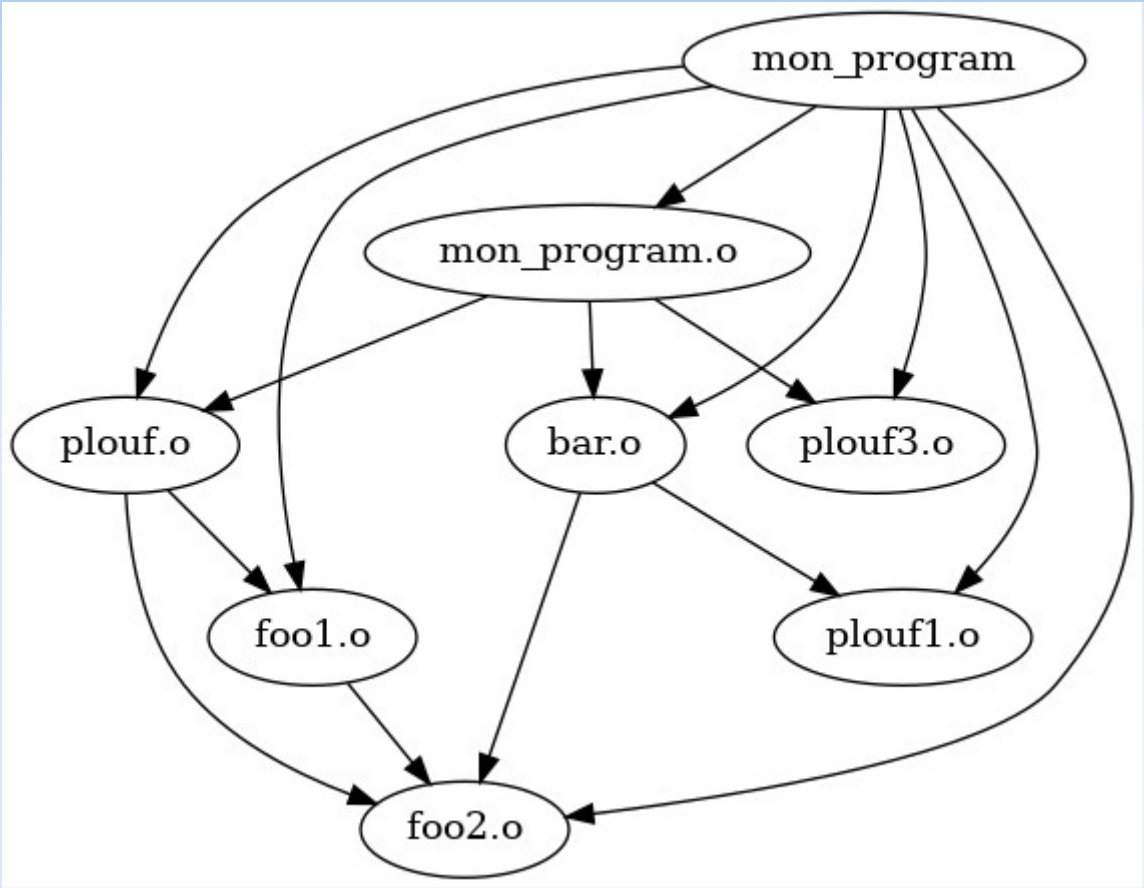
Plusieurs répertoires sources (3/3)

- Il faut indiquer à makedepf90 de chercher les sources dans les différents répertoires, avec l'option `-I` de makedepf90. Utiliser la fonction addprefix de make :
  `$(addprefix -I, ${VPATH})`

Manipulation 12

- Écrire le makefile pour la bibliothèque Jumble, sachant que :
  - Jumble utilise la bibliothèque `NR_util`
  - Les fichiers `test_eigval.f` et `test_greg2jd.f` contiennent des programmes principaux de test et ne font donc pas partie de la bibliothèque.
- Tester la compilation.
- Nota bene : make crée automatiquement les commandes de compilation avec le bon chemin.

Compilation parallèle (1/2)



- On voit sur cet exemple la possibilité de principe de compilation de certains fichiers en parallèle : fou2, plouf3 et plouf1, puis, dès que foo2.o existe, foo1 et bar, etc.

Compilation parallèle (2/2)

- make crée le graphe de dépendance et sait exploiter cette possibilité.
- Utiliser l'option `-j` de make pour spécifier un nombre de tâches parallèles. Exemple :
  ```
  make -j 4
  ```
  On gagne en temps de compilation si suffisamment de coeurs sont disponibles.

Annexe : pour utiliser le suffix `.f90` (1/3)

Si vous souhaitez que vos sources Fortran aient les suffixes `.f90` ou `.F90` et non `.f` ou `.F`, ajoutez dans votre makefile les règles implicites suivantes, qui sont calquées sur les règles de la base de données de make pour `.f` et `.F`.

Annexe : pour utiliser le suffix `.f90` (2/3)

- Pour le makefile d'un programme à un seul fichier créant directement l'exécutable à partir du source (pas de séparation entre compilation de l'objet et édition de lien), ajouter :

```
%: %.f90
   $(LINK.f) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

ou bien :

```
%: %.F90
   $(LINK.F) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

Nota bene : vous n'avez pas à redéfinir `LINK.f` ni `LINK.F`, ni à créer de variable supplémentaire.

Annexe : pour utiliser le suffix `.f90` (3/3)

- Pour le makefile d'un code à plusieurs fichiers (avec séparation entre compilations des objets et édition de lien), ajouter :

```
%.o: %.f90
   $(COMPILE.f) $(OUTPUT_OPTION) $<
```

ou :

```
%.o: %.F90
   $(COMPILE.F) $(OUTPUT_OPTION) $<
```

Nota bene : vous n'avez pas à redéfinir `COMPILE.f` ni `COMPILE.F`, ni à créer de variable supplémentaire.

Annexe: création de TAGS (1/3)

- Pour les utilisateurs d'Emacs ou Vi, possibilité de créer un index des identificateurs dans un ensemble de fichiers sources, pour pouvoir sauter à la définition d'un identificateur.
- Le programme qui crée l'index s'appelle Exuberant Ctags. L'installer avec le gestionnaire de paquets de votre distribution Linux. Sur Ubuntu :

```
apt install exuberant-ctags
```

Annexe: création de TAGS (2/3)

- Le fichier contenant l'index est appelé TAGS.
- Le makefile a la liste des sources et est donc le bon endroit pour créer ou mettre à jour le fichier TAGS.

Annexe: création de TAGS (3/3)

- Pour un fichier TAGS pour Emacs, ajouter dans le makefile la règle :

```
TAGS: ${sources}
    ctags -e --language-force=fortran $^
```

  Pour Vi, enlever l'option `-e` de ctags.

- `$^` est une "variable automatique" de make : l'ensemble des prérequis de la règle courante. Donc ici `${sources}`.

Annexe : génération automatique de graphiques PDF pour LaTeX

Références sur make

- GNU Make manual
- Paul’s Rules of Makefiles
- Recursive Make Considered Harmful

5) Au delà de make

Que désirer de plus ? (1/5)

Ce qui sort du ”domaine de confort” de make :

- Compiler dans un répertoire indépendant des sources (*out of source build*)
  - Sur une machine d’un centre de calcul, en général, un utilisateur a un espace sauvegardé de petite taille, où placer les fichiers sources. Il faut donc placer les produits de la compilation dans un autre espace.
  - Même sur une machine personnelle, intérêt de distinguer une arborescence à sauvegarder et donc de placer les produits de compilation ailleurs.

Que désirer de plus ? (2/5)

- Basculer facilement entre des options de débogage et des options de rapidité. Autrement qu’en commentant et décommentant des lignes d’options dans le makefile.

Que désirer de plus ? (3/5)

- Changer de machine ou de compilateur. Autrement qu'en commentant et décommentant des lignes de makefile. Noter que le changement de compilateur implique en général le changement des options et que le changement de machine, même à compilateur identique, peut impliquer le changement des chemins des bibliothèques utilisées.

Que désirer de plus ? (4/5)

- Plusieurs exécutables ou bibliothèques à partir de plusieurs fichiers sources chacun, dans un projet. Pénible parce qu'il faut dans le makefile créer des variables pour les sources de chaque exécutable ou bibliothèque, des variables correspondantes pour les objets, des variables pour l'ensemble des sources et des objets.

Que désirer de plus ? (5/5)

- Partage du code, avec un système de gestion de version. Souhaitable de partager un système de compilation général : indépendant du compilateur et de la machine. Mais la présence d'un tel makefile général dans les sources gêne la présence d'un makefile opérationnel sur la machine de travail.

Configuration (1/2)

- Une idée puissante : distinguer une étape préalable à la compilation, la configuration. On enregistre dans les sources les informations intrinsèques au programme : liste des fichiers sources, bibliothèques à trouver, cibles à créer, etc. L'utilisateur choisit au moment de la configuration les informations de circonstance : compilateur, chemin des bibliothèques, etc.

Configuration (2/2)

- Cette idée de distinguer la configuration est mise en oeuvre par plusieurs outils qui dépassent ou englobent make.

Un petit tour d’horizon des outils au delà de make (1/2)

Outils de compilation (*build tools*) ”importants” qui annoncent pouvoir gérer du code Fortran :

- GNU Autotools : autoconf, automake et libtool.
  Depuis ~ 1991. L’ancêtre, mais toujours développé. Non essayé.
- Scons
  Depuis 2000. Interface en Python. Échec personnel en Fortran, pour des projets non triviaux (essayé en 2018).

Un petit tour d’horizon des outils au delà de make (2/2)

- CMake
  Depuis 2000. Probablement le plus populaire aujourd’hui. Multi-plateformes.
- Waf
  Depuis 2010. En Python. Échec personnel. Documentation insuffisante.
- Meson
  Depuis 2013.

6) CMake

Installation de CMake

- Avec le gestionnaire de paquets de votre distribution Linux. Sur Ubuntu :

```
apt install cmake cmake-curses-gui \
    cmake-qt-gui
```

- Il est facile aussi d'installer la dernière version sur le site de CMake, au format binaire.

Principe (1/4)

- CMake met en œuvre l'idée de configuration.
- On écrit les informations décrivant le code (liste des fichiers, bibliothèques nécessaires etc.) dans un fichier appelé `CMakeLists.txt`.
- On n'indique pas le compilateur, les options de compilation, le chemin des bibliothèques dans `CMakeLists.txt`. Ces choix seront faits au moment de la configuration.

Principe (2/4)

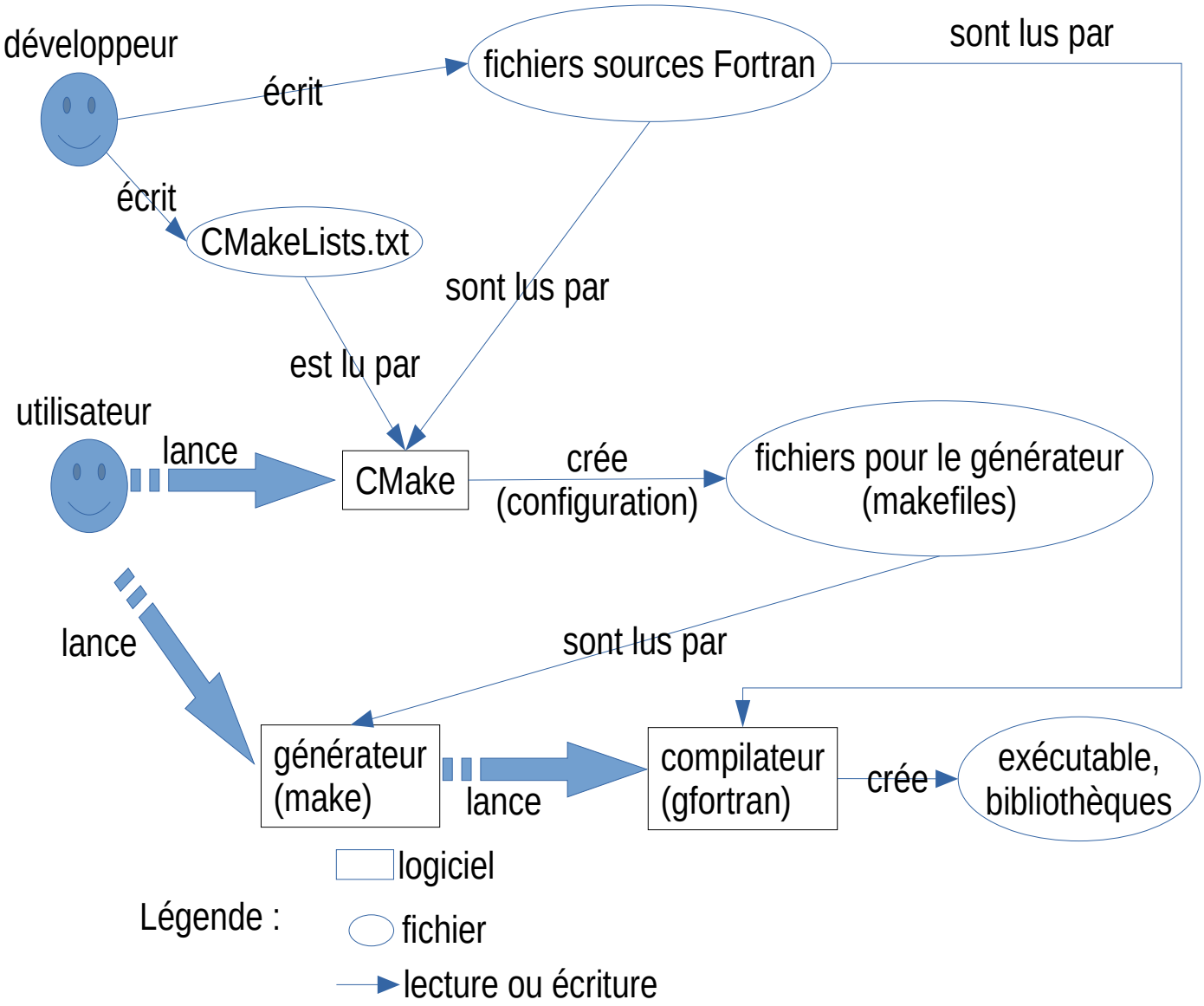
- CMake gère la configuration et nécessite un autre outil pour la compilation. CMake peut travailler avec plusieurs outils de compilation au choix de l'utilisateur, et selon le système d'exploitation de la machine de compilation (on dit encore : selon la "plateforme"). CMake appelle ces outils des générateurs. Pour voir la liste disponible sur sa machine :

```
cmake --help
```

Principe (3/4)

- Sur Linux, les générateurs sont make et Ninja. Avec make, par exemple, CMake va créer des makefiles (très compliqués).

Principe (4/4)


développeur
écrit
fichiers sources Fortran
sont lus par
écrit
CMakeLists.txt
sont lus par
est lu par
utilisateur
lance
CMake
crée
(configuration)
fichiers pour le générateur
(makefiles)
lance
sont lus par
générateur
(make)
lance
compilateur
(gfortran)
crée
exécutable,
bibliothèques
Légende :
logiciel
fichier
lecture ou écriture

Répertoire de compilation

- CMake crée des fichiers pour le générateur (des makefiles si le générateur est make, par exemple) dans un répertoire quelconque, choisi par l'utilisateur.
- Le générateur (make par exemple) est invoqué depuis ce répertoire et les produits de compilation sont créés dans ce répertoire.
- Répertoire de compilation = *build dir* = *binary dir*

La question du suffixe à nouveau

- CMake a la même convention que la plupart des compilateurs : il suppose par défaut que `.f90` et `.F90` indiquent le format libre, `.f` et `.F` le format fixe.
- Il n'y a donc plus de raison de faire un autre choix.

Langage de CMake (1/2)

(Le point noir de CMake)

- Un fichier `CMakeLists.txt` est écrit dans un langage spécifique à CMake.
- Des variables, des structures de contrôle (alternative, itération), des commandes, des fonctions…
- Les noms de commandes, de fonctions sont insensibles à la casse. Les noms de variables sont sensibles à la casse.

Langage de CMake (2/2)

- Commentaires : #
- Valeur d’une variable : `${myVar}`
- Les arguments d’une commande ou d’une fonction sont entre parenthèses séparés par des espaces ou des retours à la ligne.

Manipulation 13 : programme trivial (1/4)

- Écriture de `CMakeLists.txt` :
  - `cmake_minimum_required(VERSION` x.x`)`
    Conseil : dans le doute, mettez le numéro de votre version. Ce qui entre en ligne de compte : partage du code, version disponible dans les distributions Linux, fonctionnalités de CMake utilisées.
  - `project(`<nom de projet> `LANGUAGES Fortran)`
    (La casse importe pour ces arguments.)
  - `add_executable(`<nom de cible> <liste des fichiers sources>`)`

Manipulation 13 : programme trivial (2/4)

- Le nom de cible est un identificateur dans CMake. Le nom du fichier exécutable est créé à partir du nom de la cible, selon la convention du système d’exploitation. Ne pas mettre de suffixe.

Manipulation 13 : programme trivial (3/4)

- Configuration :

```
mkdir build
cd build
```

Répertoire de compilation quelconque, éventuellement en dehors de l'arborescence des sources. On peut rester dans le répertoire source mais cela est déconseillé : CMake va créer beaucoup de fichiers pour le générateur, en plus des produits de la compilation. Par exemple si le générateur est make, CMake va créer beaucoup plus qu'un makefile.

Manipulation 13 : programme trivial (4/4)

- `cmake` <chemin vers la racine de l'arborescence source> (contenant CMakeLists.txt)
- Compilation :
  `make`
- Essayer `make` une seconde fois.

Manipulation 14 : arguments possibles de make

- Les cibles de make
  `make help`
- Voir les commandes invoquées par make :
  `make VERBOSE=1`

Manipulation 15 : cmake-gui (1/2)

Alternative à la commande cmake dans le terminal, pour choisir interactivement la configuration avec une interface graphique.

```
cd build
rm -r *
cmake-gui <chemin vers la racine de l'arborescence source>
```

Manipulation 15 : cmake-gui (2/2)

- Cliquer sur ”Configure”. Choix :
  - du générateur
  - du compilateur
  - du type de compilation : débogage, rapidité (*Release*), etc.
- Cliquer sur ”Generate”.
- `make`

Configuration et génération (1/2)

- Le travail de CMake est en fait divisé en deux étapes :
  - Configuration : CMake se construit une représentation interne du projet, indépendante du générateur choisi.
  - Génération : CMake crée les fichiers pour le générateur choisi.
- Vocabulaire source de confusion : la *génération* est une opération faite par CMake et non par le *générateur*.

Configuration et génération (2/2)

- cmake sur la ligne de commande : configuration et génération enchaînées automatiquement.
  cmake-gui : deux boutons.

Manipulation 16 : équivalence de cmake et cmake-gui

Tous les choix avec cmake-gui peuvent aussi être faits avec cmake sur la ligne de commande. Par exemple :

```
cd build
rm -r *
cmake -DCMAKE_BUILD_TYPE=Debug chemin
make VERBOSE=1
```

Manipulation 17 : voir la configuration après coup

- Après cmake ou cmake-gui, pour voir la configuration qui a été choisie, regarder le fichier `CMakeCache.txt` créé dans le répertoire de compilation. Notamment : `CMAKE_BUILD_TYPE`, `CMAKE_Fortran_COMPILER`
- Pour voir les options de compilation et la commande d'édition de liens qui vont être utilisées par make : `CMakeFiles/`cible`.dir/flags.make` `CMakeFiles/`cible`.dir/link.txt`

Modifier la configuration (1/2)

- On peut relancer cmake ou cmake-gui.
- Une fois que le fichier `CMakeCache.txt` existe, il n'est plus nécessaire de répéter sur la ligne de commande le chemin des sources :
  `cmake .`
  ou
  `cmake-gui .`
  suffit.

Modifier la configuration (2/2)

- Avec cmake, il n'est plus nécessaire non plus de répéter les options (`-D`…). On peut en modifier une partie ou en mettre de nouvelles.
- Pour voir les options : option `-L` de cmake.
- Si on modifie le fichier `CMakeLists.txt`, il n'est même pas nécessaire de relancer cmake. On peut directement lancer make. La modification de `CMakeLists.txt` est détectée et la configuration est enchaînée automatiquement avec la compilation.

Manipulation 18 : plusieurs configurations

```
cd ..
mkdir build_release
cd build_release
cmake -DCMAKE_BUILD_TYPE=Release chemin
make VERBOSE=1
```

Un système prometteur (1/3)

Le système de CMake : créer des fichiers de générateur (des makefiles) dans un répertoire indépendant des sources, répond à nos attentes :

- Placer les produits de compilation dans un répertoire indépendant des sources.

Un système prometteur (2/3)

- Basculer facilement entre des options de débogage et des options de rapidité, ou changer de compilateur : il suffit de créer des répertoires pour différentes configurations. Pas de recompilation inutile lorsqu'on passe de l'un à l'autre. Aucune autre opération à faire que changer de répertoire.

Un système prometteur (3/3)

- Changer de machine. On peut laisser sur une machine le répertoire de compilation et déplacer les sources sur une autre machine. Si on revient à une machine, pourvu que l'on rapporte les sources au même endroit, la configuration de compilation est prête : rien à refaire.
- Partage du code : on partage `CMakeLists.txt`, indépendant du compilateur et de la machine.

Compilation d'une bibliothèque (1/2)

- La commande est :
  `add_library(`<nom de cible> <liste de fichiers sources>`)`
- En général, il est préférable de laisser le choix entre bibliothèque statique et dynamique sera fait par l'utilisateur. CMake crée par défaut une bibliothèque statique.
- La liste de fichiers sources doit contenir les fichiers visés par les lignes `include`.

Compilation d’une bibliothèque (2/2)

- Le nom de la cible est utilisée pour former le nom de fichier contenant la bibliothèque compilée, selon les conventions du système d’exploitation. Ne pas mettre le préfixe `lib` dans le nom de la cible, ne pas mettre de suffixe `.a` ou `.so`.

Propriétés d’une cible (1/4)

- La cible n’est pas seulement un identificateur utilisé pour former un nom de fichier (bibliothèque ou exécutable). Une cible a des propriétés.

Propriétés d’une cible (2/4)

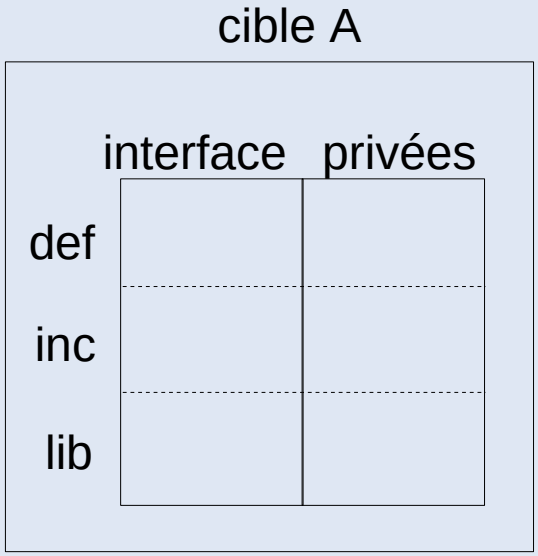
- Propriétés d’interface ou de ”non interface”. On peut aussi appeler ”privées” les propriétés de ”non interface”.
  - Les propriétés privées sont utilisées pour compiler la cible elle-même.
  - Les propriétés d’interface d’une cible A sont destinées aux cibles ”consommatrices” de A : les cibles qui vont utiliser A.

Propriétés d’une cible (3/4)

- Propriétés importantes (mais il en existe d’autres) :
  - `INTERFACE_COMPILE_DEFINITIONS` et `COMPILE_DEFINITIONS` : les macros pour le préprocesseur
  - `INTERFACE_INCLUDE_DIRECTORIES` et `INCLUDE_DIRECTORIES` : les répertoires dans lesquels chercher les ”en-têtes”, c’est-à-dire les fichiers inclus et les fichiers `.mod`

Propriétés d’une cible (4/4)

- `INTERFACE_LINK_LIBRARIES` et `LINK_LIBRARIES` : les dépendances, c’est-à-dire typiquement des cibles qui désignent des bibliothèques
- Celles qui n’ont pas le préfixe `INTERFACE_` sont les propriétés privées.



Définition des propriétés (1/2)

- Trois commandes :
  - `target_compile_definitions` pour `INTERFACE_COMPILE_DEFINITIONS` et `COMPILE_DEFINITIONS`
  - `target_include_directories` pour `INTERFACE_INCLUDE_DIRECTORIES` et `INCLUDE_DIRECTORIES`
  - `target_link_libraries` pour `INTERFACE_LINK_LIBRARIES` et `LINK_LIBRARIES`

Définition des propriétés (2/2)

- Ces commandes prennent un argument PRIVATE, INTERFACE ou PUBLIC.
- PUBLIC : la commande définit deux propriétés d’un coup, la propriété privée et la propriété interface.
  PUBLIC = PRIVATE + INTERFACE

Définition d'une macro du préprocesseur

- `target_compile_definitions(`<nom de cible> `<INTERFACE|PUBLIC|PRIVATE>` <macro>`)`
  ou
  `target_compile_definitions(`<nom de cible> `<INTERFACE|PUBLIC|PRIVATE>`
  <macro>`=`<valeur>`)`
- Pas de `-D`

Ajout d’un chemin d’en-tête

Ajout d’un répertoire à la liste des répertoires dans lesquels chercher un fichier inclus ou un fichier `.mod` :

`target_include_directories(`<nom de cible> `<INTERFACE|PUBLIC|PRIVATE>` répertoire`)`

Ajout d’une dépendance à une bibliothèque

- `target_link_libraries(`<nom de cible A> <`INTERFACE`|`PUBLIC`|`PRIVATE`> <nom de cible B>`)`
- Nota bene : cette commande est différente des deux précédentes en ce qu’elle met en relation deux cibles.
- Elle indique en particulier que B est nécessaire à l’édition de liens de A.

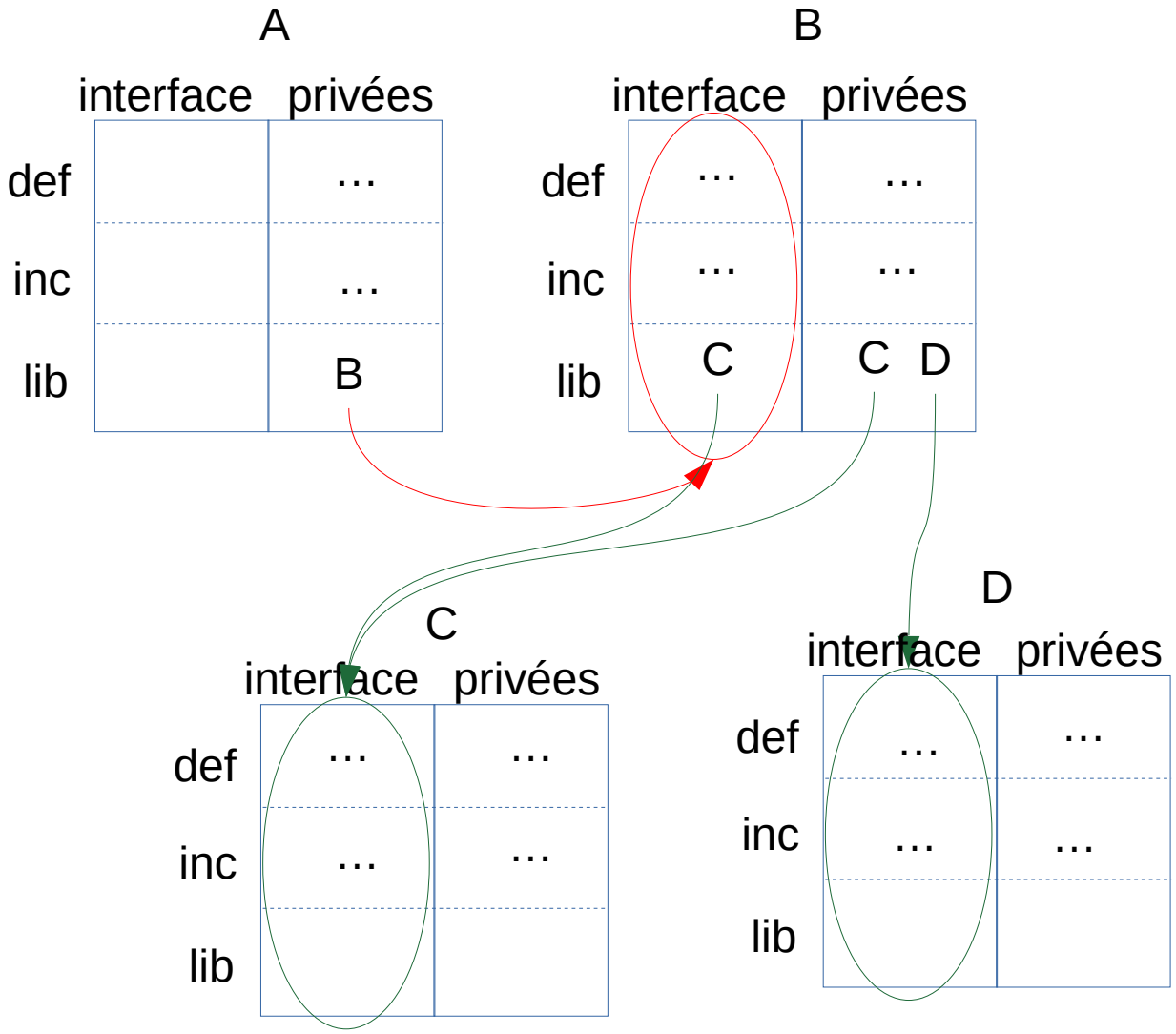
Transitivité des contraintes (1/2)

- Les propriétés d'interface d'une cible C sont prises en compte non seulement pour la compilation de toute cible B consommant C mais aussi de toute cible A consommant B, si C est dans l'interface de B. La documentation de CMake parle de "propagation le long de la chaîne de consommation", ou encore de "transitivité des contraintes".

Transitivité des contraintes (2/2)

- Si CMake crée des bibliothèques statiques et non dynamiques, toutes les bibliothèques dans la chaîne de dépendance d'une cible A sont bien prises en compte à l'édition de liens de A, même si elles ne sont pas en interface.
- À l'édition de liens, CMake met automatiquement les bibliothèques dans le bon ordre en tenant compte des dépendances entre cibles.

Exemple de suivi des dépendances (1/3)



Exemple de suivi des dépendances (2/3)

- Compilation de A :
  - def privées A + def interface B + def interface C
  - inc privés A + inc interface B + inc interface C
  - édition de liens :
    - bibliothèques dynamiques : lib B + lib C
    - bibliothèques statiques : lib B + lib C + lib D

Exemple de suivi des dépendances (3/3)

- Compilation de B :
  - def privées B + def interface C + def interface D
  - inc privés B + inc interface C + inc interface D
  - édition de liens, bibliothèques statiques ou dynamiques : lib C + lib D

`target_link_libraries` avec PUBLIC

Le choix de PUBLIC dans :
`target_link_libraries(A PUBLIC B)`
est le bon si par exemple les procédures de A prennent toujours au moins un argument qui est d'un type dérivé défini dans B. Alors toute cible consommatrice de A doit faire directement référence à B.

Manipulation 19

- Écrire le fichier `CMakeLists.txt` pour `NR_util`. Ne pas oublier les cibles pour les programmes de test.
- Tester.

nagfor et les fichiers inclus

Le compilateur NAG cherche automatiquement les fichiers visés par les lignes `include` dans le répertoire courant mais pas dans le répertoire contenant le fichier incluant, si le fichier incluant n'est pas dans le répertoire courant. Donc, si le code (bibliothèque ou exécutable) contient des fichiers inclus :

```
if(CMAKE_Fortran_COMPILER_ID MATCHES NAG)
  target_include_directories(<cible> PRIVATE
      ${CMAKE_CURRENT_LIST_DIR})
endif()
```

Variable normale ou de cache (1/2)

- Deux types de variables dans CMake : normales et de cache.
- Variables normales : définies à l'intérieur de `CMakeLists.txt`, le temps de l'éxécution de cmake (ou cmake-gui), puis perdues. Syntaxe :

```
set(varName value)
```

Variable normale ou de cache (2/2)

- Variables de cache : peuvent être modifiées à l'extérieur de `CMakeLists.txt`, au moment de la configuration, valeur conservée d'une exécution de cmake à l'autre, dans `CMakeCache.txt`.
- L'idée de variable de cache est de donner une option à l'utilisateur sans qu'il ait besoin de modifier `CMakeLists.txt`.

Affectation à une variable de cache (1/3)

- On définit habituellement une variable de cache dans `CMakeLists.txt`, avec la syntaxe :

```
set(varName <valeur> CACHE <type> "docstring")
```

  - La valeur doit être comprise comme une valeur par défaut.
  - Le type peut être : `BOOL`, `FILEPATH`, `PATH` ou `STRING`.
  - Conseil : quand on définit ainsi une variable de cache, préfixer son nom avec le nom du projet. Idée d'éviter un conflit si le projet est inclus dans un autre projet.

Affectation à une variable de cache (2/3)

- La valeur peut être modifiée sur la ligne de commande avec l'option `-D` de cmake. Cette option définit toujours une variable de cache.
  - Si la variable est aussi définie dans `CMakeLists.txt` ou prédéfinie par CMake alors la donnée de la valeur suffit :
    `-D`<var>`=`<value>
    Par exemple : `CMAKE_BUILD_TYPE` est une variable de cache prédéfinie de CMake, donc on peut juste écrire :
    `-DCMAKE_BUILD_TYPE=Debug`

Affectation à une variable de cache (3/3)

- Sinon, il est très conseillé de donner un type avec la syntaxe :
  `-D<var>:<type>=<value>`
- La valeur peut être modifiée dans cmake-gui, avec le bouton ”Add Entry”. La variable ainsi définie est toujours une variable de cache.

Manipulation 20

- Pour la bibliothèque `NR_util`, ajouter la prise en compte du compilateur NAG.
- Donner l'option à l'utilisateur de choisir la valeur de la macro `CPP_WP` à la configuration (sans qu'il ait besoin de modifier `CMakeLists.txt`).
- Tester en changeant la valeur par défaut (par exemple à `kind(0d0)`).

Bibliothèque dynamique

Pour créer une bibliothèque dynamique :

```
cmake -DBUILD_SHARED_LIBS:BOOL=True …
```

(Ou définir `BUILD_SHARED_LIBS` dans cmake-gui.)

Messages (1/3)

- Pour afficher des messages à la configuration :
  `message(`<mode> <un message>`)`
  où mode peut être, dans l'ordre du plus "haut niveau" au plus "bas niveau" : `FATAL_ERROR`, `WARNING`, `STATUS`, `VERBOSE`. `FATAL_ERROR` arrête l'exécution.

Messages (2/3)

- Choix par l'utilisateur du niveau de message à afficher :
  - avec l'option `--log-level` de cmake, seulement pour cette exécution de cmake (pas de création de variable de cache)
  - ou en définissant la variable de cache `CMAKE_MESSAGE_LOG_LEVEL`. Nota bene : cette variable n'est pas prédéfinie par CMake donc si vous la définissez sur la ligne de commande de cmake, donnez-lui son type, `STRING`. Exemple : `-DCMAKE_MESSAGE_LOG_LEVEL:STRING=VERBOSE`

Messages (3/3)

- Les messages affichés sont ceux du niveau spécifié et ceux de niveau supérieur.
- Par défaut, le niveau de log est STATUS.

Manipulation 21

À la configuration de la bibliothèque NR_util, ajouter un message annonçant la précision (valeur de `CPP_WP`) utilisée.

Modification des options de compilation (1/3)

- CMake ajoute tout seul des options correspondant aux choix : `CMAKE_BUILD_TYPE`=Release ou Debug
- Par exemple avec gfortran : `-O3` pour Release et `-g` pour Debug.

Modification des options de compilation (2/3)

- Les variables de cache prédéfinies `CMAKE_Fortran_FLAGS`, `CMAKE_Fortran_FLAGS_DEBUG` et `CMAKE_Fortran_FLAGS_RELEASE` contiennent les options de base et les options supplémentaires pour Debug et Release. On peut les modifier à la configuration. Nota bene : ces variables ne sont pas censées contenir d'option `-I` (recherche des fichiers inclus et des modules) ni `-D` (définition de macros).

Modification des options de compilation (3/3)

- Dans cmake-gui, cocher ”Advanced” pour voir `CMAKE_Fortran_FLAGS`, `CMAKE_Fortran_FLAGS_DEBUG` et `CMAKE_Fortran_FLAGS_RELEASE`.

Toolchain (1/6)

- Pour ajouter d’un seul coup de nombreuses options (notamment pour le débogage), il est pratique de collecter ces options dans un fichier, destiné à être lu directement par CMake.

Toolchain (2/6)

- Mais c’est un choix qui doit être laissé à l’utilisateur et non à l’auteur du fichier `CMakeLists.txt`. Ce n’est donc pas une bonne idée d’inclure le fichier d’options de compilation dans `CMakeLists.txt`. Il faut un autre mécanisme qui puisse prendre en compte un tel fichier à la configuration.

Toolchain (3/6)

- On utilise la variable de cache `CMAKE_TOOLCHAIN_FILE`, qui donne le chemin du fichier. Le chemin peut être absolu ou relatif à la racine du répertoire source ou du répertoire de compilation.
- Définir `CMAKE_TOOLCHAIN_FILE` à la première exécution de cmake ou cmake-gui. Ne pas le modifier ensuite. Si vous utilisez cmake-gui, cocher l'option "Specify toolchain file for cross-compiling".

Toolchain (4/6)

- Les variables à renseigner (avec la syntaxe de CMake) dans ce fichier sont `CMAKE_Fortran_FLAGS_INIT`, `CMAKE_Fortran_FLAGS_DEBUG_INIT` ou `CMAKE_Fortran_FLAGS_RELEASE_INIT`.
- Ces variables sont utilisées pour initialiser `CMAKE_Fortran_FLAGS`, `CMAKE_Fortran_FLAGS_DEBUG` ou `CMAKE_Fortran_FLAGS_RELEASE` à la première exécution de cmake ou cmake-gui seulement.

Toolchain (5/6)

- On peut éventuellement encore modifier ensuite `CMAKE_Fortran_FLAGS`, `CMAKE_Fortran_FLAGS_DEBUG` ou `CMAKE_Fortran_FLAGS_RELEASE` à la configuration. Le plus pratique est alors de le faire via cmake-gui.

Toolchain (6/6)

- Nota bene : l'esprit du fichier toolchain est d'être spécifique à la machine mais indépendant des projets. Conseils :
  - ne pas mettre de logique spécifique à un projet dedans ;
  - ne pas le placer dans le répertoire source ;
  - ne pas le placer dans le répertoire de compilation.

Manipulation 22

- Configurer NR_util pour une compilation de type Debug, en utilisant le fichier `gfortran_7.cmake`, qui contient (entre autres) des options de débogage.
- Re-configurer en supprimant la sous-option invalid de `-ffpe-trap`.

Projet à plusieurs répertoires (1/2)

- Dans le répertoire racine, créer la cible (exécutable ou bibliothèque) en référençant seulement les fichiers sources du répertoire racine.
- Pour chaque sous-répertoire : `add_subdirectory(source_dir)`
- Chaque sous-répertoire doit contenir un fichier `CMakeLists.txt`.

Projet à plusieurs répertoires (2/2)

- La lecture du `CMakeLists.txt` parent est suspendue pour lire le `CMakeLists.txt` du sous-répertoire.
- Dans chaque sous-répertoire :
  `target_sources(`<target> `PRIVATE` liste de fichiers sources`)`
  avec les fichiers qui sont à ce niveau de l'arborescence uniquement.

Projet englobant

On peut aussi inclure dans un projet d’autres projets avec la commande :

```
add_subdirectory(source_dir)
```

Portée des variables

`add_subdirectory` crée un nouvelle portée (*scope*) pour les variables du `CMakeLists.txt` du sous-répertoire.

- Les variables du `CMakeLists.txt` parent peuvent être lues. Si une telle variable est modifiée, la modification n'est pas visible par le parent.
- Les variables du `CMakeLists.txt` du sous-répertoire sont invisibles par le parent.

Notion de liste dans CMake (1/2)

- Une chaîne de caractères peut être interprétée comme une liste, avec le caractère `;` comme séparateur.
- `set(`varName value ...`)`
  joint les valeurs avec `;` si bien que le contenu de varName peut être interprété comme la liste des valeurs en argument de `set`.

```
set(myVar a b c) # myVar = "a;b;c"
```

Notion de liste dans CMake (2/2)

- La commande `list` permet diverses opérations :
  `list(`<opération> varName …`)`
  où opération peut être `LENGTH`, `GET`, `INSERT`, `APPEND`, `PREPEND` etc.

Contexte de message (1/4)

- Possibilité de définir un contexte pour chaque message en modifiant la variable normale (pas de cache) `CMAKE_MESSAGE_CONTEXT` dans `CMakeLists.txt`.
- Le contenu de `CMAKE_MESSAGE_CONTEXT` est interprété comme une liste dont les éléments sont affichés, séparés par des points, avant chaque message.

Contexte de message (2/4)

- Chaque élément de `CMAKE_MESSAGE_CONTEXT` ne peut contenir que lettres, chiffres et blanc souligné (_).
- Utilisation habituelle :
  - On ajoute à `CMAKE_MESSAGE_CONTEXT` un élément indiquant le projet courant :
    `list(APPEND CMAKE_MESSAGE_CONTEXT` <projet>`)`

Contexte de message (3/4)

- On place cette commande `list` avant la commande `project` parce que la commande `project` déclenche les tests du compilateur (avec des affichages correspondants).
- Si le projet est englobé, comme chaque projet ajoute un élément au contexte, on voit dans le contexte la pile d'appels.
- Si le projet est englobé avec `add_subdirectory`, la modification de `CMAKE_MESSAGE_CONTEXT` est locale.

Contexte de message (4/4)

- Choix par l'utilisateur de l'affichage ou non du contexte :
  - avec l'option `--log-context` de cmake, seulement pour cette exécution de cmake (pas de création de variable de cache)
  - ou en définissant la variable de cache `CMAKE_MESSAGE_CONTEXT_SHOW`. Nota bene : cette variable n'est pas prédéfinie par CMake donc si vous la définissez sur la ligne de commande de cmake, donnez-lui son type, `BOOL`. Exemple : `-DCMAKE_MESSAGE_CONTEXT_SHOW:BOOL=True`

Manipulation 23

- Créer un projet englobant les bibliothèques `NR_util` et Jumble.
- Compléter `CMakeLists.txt` dans `NR_util` :

```
target_include_directories(nr_util
   INTERFACE ${PROJECT_BINARY_DIR})
```

- Compléter `CMakeLists.txt` à la racine de Jumble, écrire les `CMakeLists.txt` dans les sous-répertoires.
- Tester.

Manipulation 24

- Créer un projet englobant les bibliothèques `NR_util`, Jumble, Numer_Rec_95 et Coriolis.
- Compléter `CMakeLists.txt` aux racines de Jumble et Numer_Rec_95.
- Écrire `CMakeLists.txt` pour Coriolis.
- Tester.

Englober ou non un projet dans un autre (1/3)

- Il est intéressant d'englober un projet dans un autre projet dans certains cas particuliers. Exemples :
  - Compiler d'un seul coup un ensemble de bibliothèques, en changeant de machine, ou en changeant de version de compilateur, ou en changeant d'options de compilation (avec options d'analyse de performance, options de débogage, etc.).

Englober ou non un projet dans un autre (2/3)

- Quand on distribue le code d'un programme, offrir à l'utilisateur la possibilité de compiler d'un seul coup le programme et ses dépendances, même si les dépendances sont dans des dépôts indépendants. Cf. plus loin `FetchContent`.

Englober ou non un projet dans un autre (3/3)

- Mais en général, on ne veut pas inclure le projet bibliothèque dans tous les projets qui utilisent cette bibliothèque. Cela signifierait à chaque fois déplacer les sources et recompiler : on perdrait une partie de l'intérêt d'une bibliothèque. On préfère utiliser la bibliothèque déjà compilée.

Rendre une bibliothèque ”consommable”

L’utilisation de la bibliothèque par un projet indépendant (sans que le projet bibliothèque soit englobé dans l’autre projet) nécessite une préparation. Il faut créer un fichier <packageName>`Config.cmake` :

- <packageName> est le nom sous lequel la bibliothèque sera cherchée par le projet consommateur. En général, on choisit le nom du projet bibliothèque.
- Le fichier contient les propriétés d’interface de la bibliothèque.

Création du fichier config, cas sans dépendances, sans installation

Dans le cas simple où la bibliothèque ne dépend pas elle-même d'autres bibliothèques, et où on ne veut pas installer la bibliothèque (voir plus loin pour la notion d'installation), le fichier peut être créé par la commande suivante, à ajouter dans `CMakeLists.txt` :

```
export(TARGETS <cible bibliothèque>
   FILE ${PROJECT_NAME}Config.cmake)
```

Création du fichier config, cas avec dépendances, sans installation (1/4)

Si la bibliothèque A dépend elle-même d'autres bibliothèques, le fichier
A`Config.cmake`
doit vérifier la disponibilité de ces bibliothèques.

- Ajouter la commande dans `CMakeLists.txt` :

```
export(TARGETS A
   FILE ${PROJECT_NAME}Targets.cmake)
```

Création du fichier config, cas avec dépendances, sans installation (2/4)

- Noter la différence avec `FILE ${PROJECT_NAME}Config.cmake` qui crée directement le fichier `${PROJECT_NAME}Config.cmake` dans le cas sans dépendances. Ici, nous allons inclure `${PROJECT_NAME}Targets.cmake` dans `${PROJECT_NAME}Config.cmake`.

Création du fichier config, cas avec dépendances, sans installation (3/4)

- Créer manuellement un fichier AConfig.cmake.in avec le contenu :

```
include(CMakeFindDependencyMacro)
find_dependency(B)
find_dependency(C)
...
include(
${CMAKE_CURRENT_LIST_DIR}/@PROJECT_NAME@Targets.cmake
)
```

Création du fichier config, cas avec dépendances, sans installation (4/4)

- Ajouter dans `CMakeLists.txt` la commande :

```
configure_file(${PROJECT_NAME}Config.cmake.in
   ${PROJECT_NAME}Config.cmake @ONLY)
```

qui copie le fichier dans le répertoire de compilation en remplaçant les chaînes de caractères entre `@`.

Utilisation d’une bibliothèque extérieure au projet (1/3)

- Ajouter dans `CMakeLists.txt` du projet utilisateur :

```
find_package(<PackageName> CONFIG
    REQUIRED)
```

Utilisation d'une bibliothèque extérieure au projet (2/3)

- À l'exécution de cmake ou cmake-gui, si la bibliothèque utilisée n'est pas trouvée automatiquement, donner le chemin du répertoire parent de son répertoire de compilation avec la variable de cache `CMAKE_PREFIX_PATH`. Nota bene :
  - Lui donner le type `PATH`.
  - `CMAKE_PREFIX_PATH` est interprété comme une liste : vous pouvez donner plusieurs chemins (en général pour plusieurs bibliothèques) séparés par des points-virgules.

Utilisation d’une bibliothèque extérieure au projet (3/3)

- Donner des chemins absolus.

Manipulation 25

- Compiler NR_util et Jumble avec deux projets indépendants. À la configuration de Jumble, utilisez NR_util que vous venez de compiler et non la version installée dans `~/.local`.
- Préparer aussi la consommation de Jumble même si vous ne l'utilisez pas tout de suite.

Namespace (1/2)

- Conseil : ajouter une option `NAMESPACE` à la commande export :

```
export(TARGETS <cible>
  NAMESPACE <namespace>
  FILE ${PROJECT_NAME}....cmake)
```

Namespace (2/2)

- L'effet est de préfixer le nom de la cible avec namespace dans :
  `${PROJECT_NAME}...cmake`
  Les projets consommateurs devront faire référence à la cible préfixée avec namespace.
- Terminer namespace par "`::`".
- Typiquement, on choisit pour namespace :
  `${PROJECT_NAME}::`

Namespace : exemple

Dans le `CMakeLists.txt` de NR_util :

```
export(TARGETS nr_util
    NAMESPACE ${PROJECT_NAME}::
    FILE ${PROJECT_NAME}Config.cmake)
```

et dans le `CMakeLists.txt` d'un projet consommateur de NR_util :

```
find_package(NR_util CONFIG REQUIRED)
target_link_libraries(Jumble PRIVATE
    NR_util::nr_util)
```

Intérêts de namespace (1/3)

- Intérêt principal : si un nom contient `::` alors CMake le traite comme le nom d'une cible importée (typiquement avec `find_package`) ou d'un alias (cf. plus loin). Si `find_package` a été oublié ou s'il y a une faute de frappe dans le nom, une erreur est signalée à la génération.
- Si le nom ne contient pas `::`, CMake admet la possibilité que ce soit le nom d'une bibliothèque à trouver dans les répertoires du système, au moment de la compilation.

Intérêts de namespace (2/3)

- Exemple :
  ```
  target_link_libraries(Jumble PRIVATE
      nr_util)
  ```
  et on a oublié `find_package`. CMake ajoute juste `-lnr_util` dans les options d'édition de liens et l'erreur n'apparaîtra qu'à la compilation.

Intérêts de namespace (3/3)

- Autre exemple, on n’a pas oublié `find_package` mais on a une faute de frappe dans le nom de la cible consommée :

```
target_link_libraries(Jumble PRIVATE
    NR_util)
```

  Là encore, l’erreur n’apparaîtra qu’à la compilation.
- Intérêt secondaire de namespace : éviter une collision entre cibles consommées de noms identiques, venant de différents projets.

Laisser le choix entre englober le projet ou non : le problème de `find_package` (1/3)

- Si les projets A et B sont englobés alors la commande
  `find_package(A CONFIG REQUIRED)`
  depuis B :
  - d'une part est inutile, puisque la cible A est définie dans une partie du projet global ;
  - d'autre part échoue (le fichier A`Config.cmake` n'est pas encore créé au moment de la configuration).
- Il faut d'une façon ou d'une autre tester l'opportunité de l'appel à `find_package`.

Laisser le choix entre englober le projet ou non : le problème de `find_package` (2/3)

- Première méthode :

```
if(CMAKE_PROJECT_NAME STREQUAL PROJECT_NAME)
   find_package(A CONFIG REQUIRED)
endif()
```

  Inconvénient : suppose que le projet englobant B englobe aussi A. Ou, en d'autres termes, que tout projet englobant B, englobera logiquement aussi A.

Laisser le choix entre englober le projet ou non : le problème de `find_package` (3/3)

- Seconde méthode :

```
if(NOT TARGET A)
   find_package(A CONFIG REQUIRED)
endif()
```

  Plus robuste. Le projet englobant B peut ne pas englober A. Inconvénient : il faut faire attention à l'ordre des appels à `add_subdirectory` dans le projet englobant, `add_subdirectory(A)` doit être avant `add_subdirectory(B)`.

Laisser le choix entre englober le projet ou non : le problème du namespace (1/2)

- Supposons que A soit utilisé par B :

  ```
  target_link(B ... Project_A::A)
  ```

  et que A et B soient englobés.
- Le namespace `Projet_A::` n'est ajouté que dans le fichier A`Config.cmake`, à l'intention seule de `find_package`.

Laisser le choix entre englober le projet ou non : le problème du namespace (2/2)

- Pour que la commande `target_link` ci-dessus n'échoue pas, il faut ajouter un alias dans le projet A :

```
add_library(PROJECT_A::A ALIAS A)
```

Manipulation 26

Modifier les `CMakeLists.txt` de NR_util, Jumble, Numer_Rec_95 et Coriolis pour que ces codes puissent être compilés aussi bien indépendamment qu'englobés dans un super-projet. Penser à mettre des namespaces. Tester.

L’idée d’installation (1/3)

- Installation : après la compilation, copie des exécutables ou des bibliothèques compilées vers une autre arborescence, avec d’autres exécutables ou bibliothèques venant de projets indépendants.
- Nous avons donc 3 arborescences pour un projet : source, compilation et installation.

L’idée d’installation (2/3)

- Intérêts de l’installation :
  - Regrouper pour mieux trouver. Le répertoire d’installation des exécutables peut faire partie de la variable `PATH`. Les bibliothèques et les `.mod` sont aussi cherchés dans un endroit conventionnel, sans avoir besoin de connaître les divers répertoires de compilation.
  - Avoir une version ”stable” installée et continuer à développer et compiler ailleurs la version en développement.

L’idée d’installation (3/3)

- Ne garder que le nécessaire. Possibilité de détruire après installation les répertoires source et de compilation. Les fichiers installés sont en général beaucoup moins nombreux que ceux du répertoire de compilation (fichiers objets, fichiers générés par CMake etc.).

Chemins d’installation conventionnels (1/2)

- `/usr/local` pour les installations communes à tous les utilisateurs (si on a l’autorisation).
- `~/.local` pour soi-même seulement, est un choix assez courant.
- D’autres choix sont possibles.

Chemins d'installation conventionnels (2/2)

- Le chemin à choisir ci-dessus est la racine d'une arborescence. En dessous, on trouve les répertoires :
  - `bin` pour les exécutables
  - `lib` pour les bibliothèques
  - `include` pour les fichiers inclus (surtout en C ou C++) et les fichiers `.mod`
  - `man`, fichiers lus par la commande `man`
  - `libexec` pour les exécutables non destinés à être appelés directement

Installation d’un exécutable (1/2)

- Ajouter dans `CMakeLists.txt` la commande :

  `install(TARGETS` target`)`

- À la configuration, choisir la racine de l’arborescence d’installation avec la variable de cache prédéfinie `CMAKE_INSTALL_PREFIX`. L’exécutable sera installé dans le sous-répertoire bin sous cette racine.

Installation d’un exécutable (2/2)

- Ni le répertoire désigné par `CMAKE_INSTALL_PREFIX` ni ses sous-répertoires (bin, etc.) n’ont besoin d’exister à l’avance.
- Pour installer :

```
make install
```

Manipulation 27

Ajouter dans le projet Coriolis une commande d’installation. Tester.

Installation d'une bibliothèque : `install(TARGETS)` (1/2)

```
install(TARGETS target EXPORT
    ${PROJECT_NAME}Targets INCLUDES
    DESTINATION include)
```

déclare :

- que la cible indiquée doit être installée lors de la commande `make install`
- que la cible fait partie d'un ensemble de cibles à installer appelé `${PROJECT_NAME}Targets`
  (on pourrait choisir ici un nom arbitraire)

Installation d'une bibliothèque : `install(TARGETS)` (2/2)

- que le sous-répertoire `include` de `CMAKE_INSTALL_PREFIX` doit être ajouté à la propriété `INTERFACE_INCLUDE_DIRECTORIES` de la cible installée.

Installation d'une bibliothèque : le fichier `.mod`

```
install(FILES
   ${PROJECT_BINARY_DIR}/ma_bibli.mod
   TYPE INCLUDE)
```

déclare que le fichier `ma_bibli.mod` doit être installé, que ce fichier est de type `INCLUDE` et doit donc aller dans le répertoire correspondant de `CMAKE_INSTALL_PREFIX`.

Installation d'une bibliothèque : le fichier `config`, sans dépendances (1/2)

```
install(EXPORT ${PROJECT_NAME}Targets
    DESTINATION lib/cmake/${PROJECT_NAME}
    NAMESPACE ${PROJECT_NAME}:: FILE
    ${PROJECT_NAME}Config.cmake)
```

- Demande la création, lors de `make install`, d'un fichier `${PROJECT_NAME}Config.cmake`, dans le répertoire `lib/cmake/${PROJECT_NAME}` sous `CMAKE_INSTALL_PREFIX`.
- Ce fichier est destiné à `find_package` et permet de générer les cibles avec toutes leurs propriétés d'interface.

Installation d'une bibliothèque : le fichier `config`, sans dépendances (2/2)

- Les cibles concernées sont celles qui font partie de `${PROJECT_NAME}Targets` (ensemble de cibles défini par l'option `EXPORT` de `install(TARGETS)`).
- Les cibles générées auront le préfixe spécifié après l'option `NAMESPACE`.

Installation d'une bibliothèque : le fichier `config`, avec dépendances (1/2)

- `install(EXPORT ${PROJECT_NAME}Targets`
  `DESTINATION lib/cmake/${PROJECT_NAME}`
  `NAMESPACE ${PROJECT_NAME}::)`
  - La seule différence avec le cas avec dépendances est l'absence de l'option FILE. Le nom du fichier est alors le nom par défaut : `${PROJECT_NAME}Targets.cmake`
  - Ce fichier sera créé, lors de `make install`, dans le répertoire `lib/cmake/${PROJECT_NAME}` sous `CMAKE_INSTALL_PREFIX`.
  - Il sera inclus dans `${PROJECT_NAME}Config.cmake`.

Installation d'une bibliothèque : le fichier `config`, avec dépendances (2/2)

- ```
  install(FILES
      ${CMAKE_CURRENT_BINARY_DIR}/${PROJECT_NAME}Config.cmake
      DESTINATION
      lib/cmake/${PROJECT_NAME})
  ```
- Rappel : le fichier `${PROJECT_NAME}Config.cmake` a été créé par `configure_file`.
  ```

Utilisation d’une bibliothèque dans le répertoire de compilation (1/2)

- Même si la bibliothèque est installée, on peut vouloir utiliser la bibliothèque dans son répertoire de compilation (par exemple pour utiliser la bibliothèque compilée en mode `DEBUG`, alors que celle installée est compilée en mode `RELEASE`, ou pour tester une version en développement).
- Nous avons vu que l’utilisation dans le répertoire de compilation était possible grâce à la commande `export(TARGETS)`.

Utilisation d’une bibliothèque dans le répertoire de compilation (2/2)

- Si on a ajouté dans `CMakeLists.txt` des commandes d’installation, il est plus cohérent de remplacer `export(TARGETS)` par `export(EXPORT)`.

export(EXPORT) (1/3)

- Cas sans dépendances, remplacer :

```
export(TARGETS target
  NAMESPACE ${PROJECT_NAME}::
  FILE ${PROJECT_NAME}Config.cmake)
```

par :

```
export(EXPORT ${PROJECT_NAME}Targets
    NAMESPACE ${PROJECT_NAME}::
    FILE ${PROJECT_NAME}Config.cmake)
```

export(EXPORT) (2/3)

- Cas avec dépendances, remplacer :

```
export(TARGETS target
  NAMESPACE ${PROJECT_NAME}::
  FILE ${PROJECT_NAME}Targets.cmake)
```

par :

```
export(EXPORT ${PROJECT_NAME}Targets
    NAMESPACE ${PROJECT_NAME}::)
```

(sans option `FILE`)

`export(EXPORT)` (3/3)

- `${PROJECT_NAME}Targets` a été défini par l'option `EXPORT` de `install(TARGETS)`.
- La commande `export(EXPORT)` a pour effet, comme `export(TARGETS)`, de créer `${PROJECT_NAME}Config.cmake` (cas sans dépendances) ou `${PROJECT_NAME}Targets.cmake` (cas avec dépendances) dans le répertoire de compilation.

Installation d'une bibliothèque : BUILD_INTERFACE (1/2)

- La commande :

```
target_include_directories(nr_util
    INTERFACE ${PROJECT_BINARY_DIR})
```

  qui permet l'utilisation de la bibliothèque englobée, ou non installée, n'est plus possible si des commandes d'installation sont présentes.

- On ne peut ajouter `${PROJECT_BINARY_DIR}` à l'interface de la cible installée : c'est un chemin absolu dans le répertoire de compilation.

Installation d'une bibliothèque : BUILD_INTERFACE (2/2)

- Solution :

```
target_include_directories(nr_util
   INTERFACE
   $<BUILD_INTERFACE:${PROJECT_BINARY_DIR}>)
```

`$<BUILD_INTERFACE:${PROJECT_BINARY_DIR}>` vaut `${PROJECT_BINARY_DIR}`
pour les commandes de compilation et rien pour les commandes d'installation.

Manipulation 28

- Ajouter les commandes d’installation aux bibliothèques NR_util, Jumble et Numer_Rec_95. Installer les bibliothèques.
- Tester l’utilisation des bibliothèques installées avec le programme Coriolis.

Module regroupeur d'une bibliothèque (1/2)

- Une bibliothèque peut proposer un module unique d'interface avec l'utilisateur, qui ne fait que regrouper les interfaces des modules de la bibliothèque :

```
module ma_bibli
   use foo_m
   use bar_m
   use plouf_m
end module ma_bibli
```

Module regroupeur d'une bibliothèque (2/2)

- Dans le programme utilisateur de la bibliothèque, le seul module utilisé est le regroupeur :

```
use ma_bibli, only: foo
```

- Lors de la compilation de la bibliothèque, des fichiers `.mod` vont être créés pour les modules contenant les procédures et pour le module regroupeur.

Fichiers `.mod` d'une bibliothèque nécessaires au programme utilisateur

- Quels fichiers `.mod` sont nécessaires à la compilation du programme utilisateur ? Seulement le `.mod` regroupeur, `ma_bibli.mod`, ou tous les `.mod` de la bibliothèque ?
- La réponse dépend du compilateur (et de sa version). Avec gfortran 9 et ifort 15, le fichier `.mod` regroupeur contient toute l'information, lui seul est utile. Avec ifort 19, nagfor 6 et pgfortran 2016, tous les `.mod` doivent être présents.

Conséquences pour le `CMakeLists.txt` de la bibliothèque (1/2)

- Le fait que la compilation du programme utilisateur d'une bibliothèque puisse nécessiter l'accès à tous les `.mod` de la bibliothèque n'apporte pas de nouvelle contrainte si la bibliothèque est utilisée dans son répertoire de compilation : la propriété `INTERFACE_INCLUDE_DIRECTORIES` de la bibliothèque, qui permet de trouver le module regroupeur, permet aussi de trouver dans le même répertoire les autres `.mod`.

Conséquences pour le `CMakeLists.txt` de la bibliothèque (2/2)

- Par contre, c'est une nouvelle contrainte si la bibliothèque est utilisée dans son répertoire d'installation : tous les `.mod` doivent avoir été installés, pas seulement le `.mod` regroupeur.

Installation de tous les `.mod` d'une bibliothèque (1/6)

- Créer tous les `.mod` dans un répertoire séparé :

```
set_target_properties(target PROPERTIES
   Fortran_MODULE_DIRECTORY
   ${PROJECT_BINARY_DIR}/modules)
```

Installation de tous les .mod d'une bibliothèque (2/6)

- Ajouter ce répertoire à la propriété INCLUDE_DIRECTORIES de la bibliothèque dans le répertoire de compilation :

```
target_include_directories(target PUBLIC
   $<BUILD_INTERFACE:${PROJECT_BINARY_DIR}/modules>)
```

au lieu de :

```
target_include_directories(target INTERFACE
$<BUILD_INTERFACE:${PROJECT_BINARY_DIR}>)
```

Installation de tous les `.mod` d'une bibliothèque (3/6)

- Installer le répertoire dans un sous-répertoire de `include` (plutôt que de mettre en vrac dans `include` tous les `.mod` de toutes les bibliothèques) :

```
install(DIRECTORY
   ${PROJECT_BINARY_DIR}/modules/
   DESTINATION include/${PROJECT_NAME})
```

au lieu de :

```
install(FILES
   ${PROJECT_BINARY_DIR}/ma_bibli.mod
   TYPE INCLUDE)
```

Installation de tous les `.mod` d'une bibliothèque (4/6)

- Nota bene : le `/` après `modules` dans la commande `install(DIRECTORY)` ci-dessus, pour copier le contenu du répertoire et non le répertoire lui-même.

Installation de tous les .mod d'une bibliothèque (5/6)

- Indiquer que `include/${PROJECT_NAME}` est dans la propriété `INTERFACE_INCLUDE_DIRECTORIES` de la cible installée :

```
install(TARGETS target EXPORT
   ${PROJECT_NAME}Targets INCLUDES DESTINATION
   include/${PROJECT_NAME})
```

au lieu de :

```
install(TARGETS target EXPORT
   ${PROJECT_NAME}Targets INCLUDES DESTINATION
   include)
```

Installation de tous les `.mod` d'une bibliothèque (6/6)

- Possibilité d'ajouter dans `CMakeLists.txt` un test pour décider selon le compilateur si on installe seulement le `.mod` regroupeur ou tous les `.mod`.

Manipulation 29

Modifier les `CMakeLists.txt` des bibliothèques pour que l'installation et l'utilisation des bibliothèques installées avec les compilateurs NAG, Intel récent ou PGI soient possibles.

Comment utiliser une bibliothèque compilée sans CMake ?

La meilleure façon est de passer par un ”module de recherche”, en anglais *find module*, de CMake.

Modules de recherche de CMake (1/3)

- Même si la bibliothèque n’a pas été compilée avec CMake, il peut exister un ”module de recherche” de CMake pour cette bibliothèque.

Modules de recherche de CMake (2/3)

- Le module de recherche contient du code CMake qui :
  - cherche la bibliothèque dans des endroits standard, utilisant éventuellement d'autres logiciels comme pkg-config pour trouver la bibliothèque ;
  - crée une cible dans CMake, avec des propriétés d'interface, cible qui peut être utilisée comme si la bibliothèque correspondante avait été compilée avec CMake.

Modules de recherche de CMake (3/3)

- Le module de recherche est un fichier dont le nom est de la forme `Find`<PackageName>`.cmake`.
- Un ensemble de modules de recherche est distribué avec CMake lui-même.
- Le module de recherche peut être écrit par un tiers (chercher sur internet, sur Github etc. si quelqu'un en a écrit un pour la bibliothèque qui vous intéresse).

Utilisation d’un module CMake pour trouver une bibliothèque (1/2)

- Si ce n’est pas un module de recherche distribué avec CMake, le télécharger dans votre arborescence source.
- Dans `CMakeLists.txt`, donner le chemin de ce module. Par exemple :

```
list(APPEND CMAKE_MODULE_PATH
   ${CMAKE_CURRENT_SOURCE_DIR})
```

Utilisation d’un module CMake pour trouver une bibliothèque (2/2)

- Faire appel au module avec la commande : `find_package(`PackageName `REQUIRED)` sans le mot-clef `CONFIG` (à la différence de la recherche d’une bibliothèque compilée avec CMake).

Manipulation 30

- Écrire le fichier `CMakeLists.txt` pour le programme `test_netcdf90.f90`.
- Utiliser les modules `FindNetCDF.cmake` et `FindNetCDF_Fortran.cmake` du dépôt lguez/cmake.

Rappel : modes module et config de `find_package`

Attention :

- mode module : `find_package` sans le mot-clef `CONFIG`, utilise `Find<PackageName>.cmake` dans le répertoire source du projet utilisateur de PackageName.
- mode config : `find_package` avec le mot-clef `CONFIG`, utilise `<PackageName>Config.cmake` dans le répertoire de compilation de PackageName ou dans un répertoire d'installation.

Utilisation d’une bibliothèque sans module de recherche (1/4)

- Pour utiliser une bibliothèque non compilée par CMake et pour laquelle vous n’avez pas de module de recherche, utilisez les commandes :
- `find_path(`<VAR> name `REQUIRED)`
  Cherche le répertoire contenant le fichier name, résultat dans <VAR>. Xxx`_INCLUDE_DIR` est un nom conventionnel pour <VAR>.

Utilisation d'une bibliothèque sans module de recherche (2/4)

- `find_library(`<VAR> name `REQUIRED)`
  Cherche la bibliothèque name. Donner name sans préfixe (`lib`) ni suffixe (`.a` ou `.so`). En sortie, <VAR> contient le nom complet, avec le chemin absolu et le préfixe et le suffixe. Xxx_LIBRARY est un nom conventionnel pour <VAR>.
- Ces deux commandes cherchent dans les répertoires standard.

Utilisation d’une bibliothèque sans module de recherche (3/4)

- Vous pouvez donner un répertoire supplémentaire où chercher avec la variable `CMAKE_PREFIX_PATH`.
- Les variables définies par les deux commandes sont des variables de cache.
- Vous pouvez imposer leur valeur à l’installation.

Utilisation d'une bibliothèque sans module de recherche (4/4)

- Utilisez les variables obtenues :

```
target_include_directories(<target>
    <INTERFACE|PUBLIC|PRIVATE>
    ${Xxx_INCLUDE_DIR})

target_link_libraries(<target>
    <PRIVATE|PUBLIC|INTERFACE>
    ${Xxx_LIBRARY})
```

Manipulation 31

- Écrire le `CMakeLists.txt` pour le programme `max_diff_rect`, qui utilise la bibliothèque Jumble. Remarquez que Jumble va automatiquement chercher NR_util, dont il a besoin.
- Récrire le `CMakeLists.txt` en faisant comme si Jumble et NR_util n'étaient pas compilés avec CMake (c'est-à-dire en n'utilisant pas `find_package`).

FetchContent (1/3)

- La problématique :
  - Vous avez sur des dépôts un programme et des bibliothèques dont dépend ce programme.
  - Vous voulez simplifier la vie de l'utilisateur : qu'il n'ait pas à télécharger et compiler séparément chaque bibliothèque (et dans le bon ordre s'il y a des dépendances entre elles).
- Une solution si les bibliothèques peuvent être compilées avec CMake : le module FetchContent.

FetchContent (2/3)

- ```
  include(FetchContent)
  FetchContent_Declare(<name>
     GIT_REPOSITORY <url>)
  ...
  FetchContent_MakeAvailable(<name1>
     [<name2>...])
  ```
- L’ordre peut être quelconque même s’il y a des dépendances entre les bibliothèques téléchargées.
  ```

FetchContent (3/3)

- Les bibliothèques sont téléchargées au moment de la configuration. Elles sont ajoutées au projet comme avec la commande `add_subdirectory`.

La commande `option`

- `option(`optVar helpString`)`

  définit une variable de cache optVar, de type logique, dont la valeur par défaut est ”False”.

- C’est un raccourci pour :

  `set(`optVar `False CACHE BOOL` helpString`)`

Manipulation 32

- Modifiez le fichier `CMakeLists.txt` de `max_diff_rect` pour offrir l'option de télécharger automatiquement les bibliothèques nécessaires. Adresses :
  https://github.com/lguez/NR_util
  https://github.com/lguez/Jumble
- Remarquez que si l'utilisateur choisit le téléchargement alors, après la configuration, le répertoire `_deps` est créé dans le répertoire de compilation.

MPI (1/2)

- Pour compiler un programme utilisant MPI :

```
find_package(MPI REQUIRED)
target_link_libraries(<target> … PRIVATE
   MPI::MPI_Fortran)
```

- Nota bene : CMake n’utilise pas le wrapper du compilateur pour MPI. Il est possible d’avoir un exécutable compilé sans MPI dans le même projet.

MPI (2/2)

- Si vous utilisez le module `mpi_f08` (conseillé) dans votre code source Fortran, vous pouvez vérifier à la configuration que votre version de MPI est suffisamment récente. Le module FindMPI, qui est utilisé par `find_package(MPI)`, définit la variable logique `MPI_Fortran_HAVE_F08_MODULE`.

Manipulation 33

Écrivez le fichier `CMakeLists.txt` pour le programme `hello_MPI_Fortran`. Vérifiez la disponibilité de `mpi_f08`.

Problèmes de richesse (1/4)

- Si vous avez plusieurs compilateurs sur votre machine, ou plusieurs versions de bibliothèques, comment orienter CMake vers l'un ou l'autre ?
- Si votre environnement est contrôlé par la commande module, chargez les modules que vous souhaitez utiliser, et de préférence seulement ceux-là, *avant* de lancer CMake.

Problèmes de richesse (2/4)

- Pour choisir un compilateur, utilisez la variable d'environnement FC. Par exemple :

```
FC = ifort cmake ..
```

ou

```
FC = ifort cmake-gui ..
```

- *Vous ne pouvez pas* relancer cmake avec un autre compilateur, en gardant le reste de votre configuration. Pour changer de compilateur, vous devez repartir de zéro : tout effacer dans le répertoire de compilation (ou créer un autre répertoire de compilation).

Problèmes de richesse (3/4)

- Si CMake ne choisit pas la bonne version d'une bibliothèque, utilisez la variable `CMAKE_PREFIX_PATH`.
- Vous n'avez pas besoin de repartir de zéro (c'est-à-dire d'effacer le répertoire de compilation) pour changer de version d'une bibliothèque. Supprimez la variable pointant vers cette bibliothèque : bouton dans cmake-gui ou option `-U` de cmake.

Problèmes de richesse (4/4)

- Si CMake ne choisit pas la bonne version de la bibliothèque MPI, définir la variable de cache `MPIEXEC_EXECUTABLE` (type `FILEPATH`) pointant vers le mpiexec de la bonne bibliothèque.

”Bibliothèques d’objets” de CMake

Autres options de cmake

- Nous avons déjà vu les options de cmake `-D` et `-U` qui définissent ou rendent indéfinie une variable de cache.
- Options -B et -S
- Option --build
- Option -L
- Autres options

Problème de placement des .mod