Programming Environment for the TALENT School

N. Schunck

July 15, 2014

General Requirements for Computational Projects

- All programs must be controlled with the git control version system;
- All programs must be compiled with a Makefile;
- Your programs must be *modular* as much and as reasonably as possible;
- We ask that you try to use concepts of *object-oriented programming (OOP)*. If you use languages like C++ or Python, this is easy; in Fortran or C, it is less obvious but still doable;
- Fortran 77 is prohibited;
- As much as possible, try to use *libraries* such as BLAS, LAPACK (Fortran), Armadillo (C/C++), etc.;
- As much as possible, try to use *dynamic memory allocation*;
- All codes must be fully *documented*, both in the source code and in the form of a short manual – or a long README, explaining how to build and run the code.

Makefiles

Introducing Makefiles

Our (sensible) requirement that your project must be modular implies that you will have several files to compile, some possibly depending on one or several of the others. The `make` utility is an excellent way to automatize the compilation of such complex projects.

First things first: `make` may not be installed by default in your system, so start by installing it.

We will assume a simple C program composed of 3 files

- the file `main.c` containing the `main()` function
- the file `hello.c` containing a function displaying *Hello, World!*
- the file `hello.h` containing a header

Without any Makefile, the program could be compiled by

```
gcc -o main main.c hello.c -I.
```

C Programs Illustrating the Makefile

main.c

```
#include
int main()
{
   // call the function
   myPrintHelloMake();

   return(0);
}
```

hello.c

```
#include
#include

void myPrintHelloMake(void) {

   printf("Hello␣makefiles!\n");

   return;
}
```

hello.h

```
/*
example include file
*/

void myPrintHelloMake(void);
```

General Structure of a Makefile

The general command of a `Makefile` is

```
Target :  Dependency
[Tabulation]Command
```

The `Target` is a *label* that you choose to define a target, i.e., something that you want to obtain. In our example, the executable `main` would be our target.

The `Dependency` is anything that is required to “reach” the target. In our example, the executable `main` depends on some object files: these are our dependencies.

The target is then produced by executing the `Command`. In our example, the command would be the C compiler on our system.

Dependencies are often also targets. Again, in our example, the object file `hello.o` needed to produce the target `main` can be thought of a target, which will be produced by the C compiler.

Simple Makefile

Below is the simplest Makefile we could make out of our example:

```
hello: hello.o main.o
        gcc -o hello hello.o main.o

hello.o: hello.c
        gcc -o hello.o -c hello.c -W -Wall -ansi -pedantic

main.o: main.c hello.h
        gcc -o main.o -c main.c -W -Wall -ansi -pedantic
```

If you save this code into a file `Makefile`, you can compile the project by simply typing

```
make
```

This will produce the executable `hello` (the primary target). This primary target has two dependencies, `main.o` and `hello.o`. Each of them is the target of another command, with `main.o` depending on `hello.h`.

More Advanced Makefile

We add three more rules to: (i) list of the executables to produce (`all`), (ii) delete intermediate object files (`clean`), and (iii) remove the executable to force a complete rebuild (`mrproper`).

```
all: hello

hello: hello.o main.o
        gcc -o hello hello.o main.o

hello.o: hello.c
        gcc -o hello.o -c hello.c -W -Wall -ansi -pedantic

main.o: main.c hello.h
        gcc -o main.o -c main.c -W -Wall -ansi -pedantic

clean:
        rm -rf *.o

mrproper: clean
        rm -rf hello
```

User Variables

Instead of repeating the same command for compilation, we define our own variables.

```
CC      = gcc
CFLAGS  = -W -Wall -ansi -pedantic
LDFLAGS =
EXEC    = hello

all: $(EXEC)

hello: hello.o main.o
        $(CC) -o hello hello.o main.o $(LDFLAGS)

hello.o: hello.c
        $(CC) -o hello.o -c hello.c $(CFLAGS)

main.o: main.c hello.h
        $(CC) -o main.o -c main.c $(CFLAGS)

clean:
        rm -rf *.o

mrproper: clean
        rm -rf $(EXEC)
```

Combining Internal and User Variables

There are pre-defined variables that come handy:

$@	Name of the target
$<	The name of the first dependency
$^	The list of all dependencies
$?	The list of all dependencies that are more recent than the target
$*	The name of the file without the extension

```
CC      = gcc
CFLAGS  = -W -Wall -ansi -pedantic
LDFLAGS =
EXEC    = hello

all: $(EXEC)

hello: hello.o main.o
        $(CC) -o $@ $^ $(LDFLAGS)

hello.o: hello.c
        $(CC) -o $@ -c $< $(CFLAGS)

main.o: main.c hello.h
        $(CC) -o $@ -c $< $(CFLAGS)

clean:
        rm -rf *.o

mrproper: clean
        rm -rf $(EXEC)
```

Inference Rule

We now define generic rules to build all object files (extension .o) from source files (extension .c in our C programs). This is done with the inference rule

```
%.o :  %.c
```

We also add the requirement that object files depend on the header files (only one in our case, but they could be more).

```
CC       = gcc
CFLAGS   = -W -Wall -ansi -pedantic
LDFLAGS  =
EXEC     = hello
DEPS     = hello.h

all: $(EXEC)

hello: hello.o main.o
        $(CC) -o $@ $^ $(LDFLAGS)

%.o: %.c $(DEPS)
        $(CC) -o $@ -c $< $(CFLAGS)

clean:
        rm -rf *.o

mrproper: clean
        rm -rf $(EXEC)
```

Automatic Generation of Object Files

Finally, we define the list of all sources files that need be compiled, put it in a variable `SRC`, and automatically define the corresponding list of object files through

```
OBJ= $(SRC:.c=.o)
```

The final Makefile looks like

```
CC       = gcc
CFLAGS   = -W -Wall -ansi -pedantic
LDFLAGS  =
EXEC     = hello
DEPS     = hello.h
SRC      = hello.c main.c
OBJ      = $(SRC:.c=.o)

all: $(EXEC)

hello: $(OBJ)
        $(CC) -o $@ $^ $(LDFLAGS)

%.o: %.c $(DEPS)
        $(CC) -o $@ -c $< $(CFLAGS)

clean:
        rm -rf *.o

mrproper: clean
        rm -rf $(EXEC)
```

The Git Version Control System

Introducing Version Control System

Git is an open source version control software (VCS).

Advantages of VCS:

- Keep a record of the entire development history of a project (can be a code or anything else, for example the versions of your Ph.D. thesis...)
- Each snapshot of the project (list and content of all files and directories) is recorded with a mandatory comment describing it
- Considerably improves the maintenance of projects involving several people

Disadvantages: None... (OK, there is a learning curve)

Installing Git

Start with downloading the software:

- Linux: use your distribution package manager to install it
- Windows: go to http://git-scm.com/downloads and download the client
- MacOs: it is a good idea to install it via Homebrew or Macports. Ask if you are not familiar with these softwares

You may want to use a GUI client. One possible choice is git-cola, available at https://git-cola.github.io/. Its main advantage is that it is multi-platform, i.e., it will work on Linux, Windows and MacOS equally well.

In a terminal window, you can access the documentation with

```
git --help
```

and the documentation on a specific topic with, e.g.,

```
git branch --help
```

First Git Project (1/2)

In the following, we will illustrate most basic features of git using the command line. We assume that you want to create a folder `dir/` that will contain a bunch of files, and you want the entire `dir/` folder to be under git control.

In the folder that contains dir/, type:

```
git init dir
```

While still in the directory `dir/` create the file `hello.f90` and write the Fortran 90 code needed to display the usual *Hello World!* message. The content of the directory has been changed, since you have added a new file.

Take a snapshot of the directory with

```
git add .
```

and record this snapshot with

```
git commit
```

First Git Project (2/2)

Note that when you commit your project, you have to type in a commit message. Be serious about it: in complex projects, commit messages are invaluable sources of information!

Under Linux, it is very likely that the commit message has to be entered with the `vi` editor: first press `i` to enter the edit mode, then type your comments, then type `ZZ` to record and close.

The initial commit creates a new *branch* in the git repository, named *master* (all git repositories have a master branch).

If you add a file to the directory, or if you remove a file, or if you modify an existing file (by editing it for instance, or by changing its attribute), you are modifying the active branch – in our simple example, `master`.

Useful Git Commands

To see if there is anything that has been modified since your last commit

```
git status
```

To see the history of the versions of your project

```
git log
```

To list all branches of your project

```
git branch
```

The asterisk lists the branch that is currently active, i.e., the branch you are “sitting on” (think ot the branches of the tree, really).

Commits are referenced with some crazy labels. You can view the difference between the current branch and a former commit with commands such as

```
git diff 655987702ea15b9daa12dc8bc37023b35fc4eb50
```

Branches

So far, the `master` branch of our git project is made of a directory `dir` with a file called `hello.f90`.

Create a new branch with

```
git branch C++
```

List the branches...

```
git branch
```

Checkout the new branch, i.e., if your git project is a tree, you are now moving to a new branch, the one named `C++`

```
git checkout C++
```

Now, delete the file `hello.f90`, and create a new one called `hello.cc` with the code to display *Hello, World!* in C++.

After checking the status of the git repository, add and commit the new file; check the history of the repository. What happens if you now checkout the `master` branch?

To go Further on Git

This short introduction should allow you to get started with Git. The topics that have not been covered:

- How to import, export and backup a Git repository on some online servers such as Bitbicket or Github, where you can create accounts for free.
- How to merge branches and resolve conflicts: this is especially important if you use Git as part of a team.

Documentation:

- http://git-scm.com/docs/gittutorial: This is a good starting point if you are totally unfamiliar with Git and version control system. In fact, I used the material in these slides as inspiration.
- http://nvie.com/posts/a-successful-git-branching-model/: For the more advanced users, this blog entry discusses a possible Git strategy for a project development.

Workplan

Today

The goal of the first session is to write a short program (in your favorite language) that achieves the following

- Read as input an integer N giving the dimension of a matrix
- Set up a matrix of real numbers and dimension $N \times N$
- Diagonalize the matrix using a call to a library
- Display the first 10 eigenvalues

If you think this is so easy that it makes you ashamed of even contemplating doing this, we suggest you start developing a module (in your favorite language) implementing Gauss-Laguerre quadrature. In particular, you could try to very numerically the orthonormality of HO radial wave functions.