COMPUTING METHODS IN HIGH ENERGY PHYSICS

S. Lehti

Helsinki Institute of Physics

Spring 2026

Overview

Practical 'hands-on' course for computing in high energy physics.

Credits: 5op (3ov), 13 lectures + 6 exercises + project work. Exercises split in two to have 12 x 1h exercises.

Recommended prerequisites: Introduction to particle physics and programming skills.
Literature: Recommending an advanced C++ book for reference.

Links are marked with cyan color.

Outline:

- Short review to Unix basics,latex,makefile,git
- Python
- C++
- ROOT
- Combining programming languages
- Cross section and branching ratio calculations
- Event generators
- Detector simulation and reconstruction
- Fast simulations
- Grid computing

Short review to Unix basics

The Shell is a program that runs automatically when you log in to a Unix/linux system. The Shell forms the interface between users and the rest of the system. It reads each command you type at your terminal and interprets what you have asked for. The Shell is very much like a programming language with features like

- variables
- control structures (if,while,..)
- subroutines
- parameter passing
- interruption handling

These features provide you with the capability to design your own tools. Shell scripting is ideal for any small utilities that perform relatively simple task, where efficiency is less important than easy configuration, maintenance and portability. You can use the shell to organize process control, so commands run in predetermined sequence, dependent on the successful completion of each stage.

Shell Name	History
sh (Bourne)	The original shell
csh,tcsh,zsh	The C shell.
bash	Bourne Again Shell, from the GNU project
rc	More C than **csh**

Short review to Unix basics

File Handling

Most Unix commands take input from the terminal keyboard and send output to the terminal monitor. To redirect the output use $>$ symbol:

```
$ ls > myfiles
```

Likewise you can redirect the input with the $<$ symbol:

```
$ wc -l < myfiles
```

The need for redirection becomes more apparent in shell programming.

Pipes

The standard output of one process (or program) can be the standard input of another process. When this is done a "pipeline" is formed. The pipe operator is the | symbol.

Examples:

```
$ ls | sort
$ ls | sort | more
$ ls | sort | grep btag | more
```

Here the standard output from left becomes the standard input to the right of |.

Pipelines provide a flexible and powerful mechanism for doing jobs easily and quickly, without the need to construct special purpose tools. Existing tools can be combined.

Another useful command:

```
$ find .  -name "*.cpp" | xargs grep btag
```

which searches for every file in this directory and any subdirectory with

Short review to Unix basics

ending .cpp and grepping the string btag.
Suppose you have a file called 'test' which contains unix commands.
There are different ways to get the system to execute those commands.
One is giving the filename as an argument to the shell command, or one can use the command *source* in which case the current shell is used

```
$ csh test (or sh test)
$ source test
```

One can also give executing rights to the file and then run it

```
$ chmod 755 test
$ test
```

Here the number coding in the chmod command comes from rwxrwxrwx with rwx being binary number: rwx=111=7 and r-x=101=5. You can check the rights of your files with the -l option of the ls command.
A shell script starts usually with a line which tells which program is used to execute the file. The line looks like this (for bourne shell)
#!/bin/sh

Comments start with a # and continue to the end of the line. The shell syntax depends on which shell is in use. At CERN people have been using widely csh (tcsh), but also bash as the linux default has gained ground. It's up to you which shell you prefer.

Short review to Unix basics

csh

Variables are used to hold temporary values and manage changeable information. There are two types of variables:

- Shell variables
- Environment variables

Shell variables are defined locally in the shell, whereas environment variables are defined for the shell and all the child processes that are started from it.
The set command is used to create new local variables and assign a value to them. Different ways of invoking the set command are as follows:

- set
- set *name*
- set *name* = *word*
- set *name* = (*wordlist*)
- set *name*[*index*] = *word*

The first three forms are used with scalar variables, whereas the last two are used with array variables. The setenv command is used to create new environment variables. Environment variables are passed to shell scripts and invoked commands, which can reference the variables without first defining them.

- setenv
- setenv *name value*

Short review to Unix basics

To obtain the value of a variable a reference ${name} is used.
Example 2:
mv a.out anal_${CHNL}_$RUN.out

Here the variables CHNL and RUN contain some values. A reference ${#name} returns the number of elements in an array, and ${?name} returns 1 if the variable is set, 0 otherwise.
In addition to ordinary variables, a set of special variables is available.

Variable	Description
$0	Shorthand for $argv[0]
$1,$2,...,$9	Shorthand for $argv[n]
$*	Equivalent to $argv[*]
$$	Shell process number
$<	input from file

Conditional statements are created with
if(expression) then
else
endif

and loops with

foreach name (wordlist)
end

while (expression)
end

Example 3 csh script

```
#!/bin/csh

if( ! ${?ENVIRONMENT} ) setenv ENVIRONMENT
INTERACTIVE
if( $ENVIRONMENT != "BATCH" ) then
     if( ! ${?SCRATCH} ) then
          echo Setting workdir to HOME
```

Short review to Unix basics

```
        setenv WORKDIR $HOME
    else
        echo Setting workdir to SCRATCH
        setenv WORKDIR $SCRATCH
    endif
    setenv LS_SUBCWD $PWD
endif
#####################################
setenv INPUTFILE analysis.in
#setenv DATAPATH $LS_SUBCWD/data
setenv DATAPATH /mnt/data/data
#####################################

make

cd $WORKDIR
if( -f $INPUTFILE ) rm $INPUTFILE
cp -f $LS_SUBCWD/$INPUTFILE $WORKDIR
setenv LD_LIBRARY_PATH
$LS_SUBCWD/../lib:${LD_LIBRARY_PATH}

echo
echo START EXECUTION OF JOB
echo

$LS_SUBCWD/../bin/${CHANNEL}_analysis.exe >& analysis.out
if( -f $WORKDIR/analysis.out ) mv $WORKDIR/analysis.out
$LS_SUBCWD

echo
echo JOB FINISHED
echo

exit
```

bash

In bash the standard Bourne shell assignment to set variables is used

$name = value$

The array variables can be set in two ways, either by setting a single element

$name[index] = value$

or multiple elements at once

$name = (value1, .., valueN)$

Exporting variables for use in the environment

export *name*

export $name = value$

Arithmetic evaluation is performed when the following form is encountered: $((expression)). The basic if statement syntax is

Bash

if *condition* ; then
else
fi
Most often the condition given to an if statement is one or more test commands, which can be invoked by calling the test as follows:
test *expression*
[*expression*]
The other form of flow control is the case-esac block. Bash supports several types of loops: for, while, until and select loops. All loops in bash can be exit by giving the built-in break command.

Perl

Perl has become the language of choice for many Unix-based programs, including server support for WWW pages. Perl is a simple yet useful programming language that provides the convenience of shell scripts and the power and flexibility of high-level programming languages. In perl the variable can be a string, integer or floating-point number. All scalar variables start with the dollar sign $. The following assignments are all legal in perl:
$variable = 1;
$variable = "my string";
$variable = 3.14;

To do arithmetic in Perl, the following operators are supported:

Perl

+ - * / ** %. Logical operators are divided into two classes, numeric and string. Numeric logical operators are <, >, ==, <=, >=, !=, ||, && and !. Logical operators that work with strings are lt, gt, eq, le, ge and ne. The most common assignment operator is =. Autoincrement ++ and decrement - - are also available. Strings can be combined with . and .= operators, for example
$a = "be" . "witched";
$a = "be"; $a .= "witched";
The conditional statement 'if' has the following structure: if (expr) {..}. Repeating statement can be made using while and until, looping can be done with the for statement. Shell (system) commands can be run with the command system(shell command string)

More about csh(tcsh),bash and perl: man pages and www like
http://www.gnu.org/software/bash
http://www.tcsh.org
http://www.faqs.org/faqs/unix-faq/shell/csh-whynot
http://www.perl.com

Example 4: a Perl script

```
#!/usr/local/bin/perl
$ibg = 2020;

$eventsPerFile = 500;
$allEvents = 100000;
$firstEvent = 1;

$pwd = $ENV{'PWD'};
$orca = rindex("$pwd","ORCA");
$slash = index("$pwd","/",$orca);
$ORCA_Version = substr("$pwd",$orca,$slash-$orca);
print "Submitting $ORCA_VERSION job(s)\n";
for ($i = 1; $i <= $allEvents/$eventsPerFile; $i++) {
    $LAST = $firstEvent + $eventsPerFile - 1;
    system("bsub -q $BatchQueue $Jobfile $ORCA_VERSION
            $runNumber $eventsPerFile $firstEvent $LAST $ibg");
    $firstEvent = $LAST + 1;
}
if($firstEvent < $allEvents){
    $LAST = $allEvents - 1;
    system("bsub -q $BatchQueue $Jobfile $ORCA_VERSION
            $runNumber $eventsPerFile $firstEvent $LAST $ibg");
}
```

Version Control Systems

In HEP experiments the software is typically written by the collaboration members. Some part of the software can be commercial programs, but most of the code must be written from scratch.

CVS is a tool to manage the source code.

- to prevent overwriting others' changes
- keeps record of the versions
- allows users and developers an easy access to releases and prereleases

Documentation for CVS can be found in WWW, e.g.:
http://ximbiot.com/cvs

Subversion (SVN) is another version control system, and it's used widely. For example the LHC Higgs XS WG is using Subversion to manage the source code and documentation source. http://subversion.apache.org

Git

Git is an open source, distributed version control system designed to handle everything from small to very large projects with speed and efficiency.
Git does not use a centralized server.

Projects using Git: Git, Linux Kernel, Perl, Ruby on Rails, Android, WINE, Fedora, X.org, VLC, Prototype

Git

Documentation about git can be found e.g from here https://git-scm.com/doc.

Cloning

If you want to get a copy of an existing Git repository the command you need is

```
git clone [url].
```

Git receives a full copy of nearly all data that the server has. Every version of every file for the history of the project is pulled down by default.

Example 5:

```
git clone
http://cmsdoc.cern.ch/~slehti/Test.git
```

creates a directory named 'Test', initializes a .git directory inside it, pulls down all the data for that repository, and checks out a working copy of the latest version.
If you want to clone the repository into a directory named something other than 'Test', you can specify that as the next command-line option

```
git clone <url> newNameForTheDir
```

Git has a number of different transfer protocols you can use. The previous example uses the https:// protocol, but you may also see git:// or user@server:path/to/repo.git, which uses the SSH transfer protocol. In a local machine or over AFS you dont need the user@server, just the path

```
git clone ~slehti/public/html/Test.git
```

Saving changes

Each file in your working directory can be in one of two states: tracked

Git

or untracked. Tracked files are files that were in the last snapshot, they can be unmodified, modified, or staged. Untracked files are everything else – any files in your working directory that were not in your last snapshot and are not in your staging area. When you first clone a repository, all of your files will be tracked and unmodified because you just checked them out and haven't edited anything.

As you edit files, Git sees them as modified, because you've changed them since your last commit. You stage these modified files and then commit all your staged changes, and the cycle repeats.

The main tool you use to determine which files are in which state is the

```
git status
```

command. Files that are staged are under the "Changes to be committed" heading. In order to begin tracking a new file, use the command

```
git add.
```

Git add is a multipurpose command which is used to begin tracking new files, to stage files, and to do other things like marking merge-conflicted files as resolved. It may be helpful to think of it more as "add this content to the next commit" rather than "add this file to the project".

Git

If you want to know exactly what you changed, not just which files were changed – you can use the

```
git diff
```

command. If you want to see what you've staged that will go into your next commit, you can use

```
git diff --staged.
```

This command compares your staged changes to your last commit. Note that if you have staged all of your changes, git diff will give you no output. If you want to compare the version before the last commit and the last commit, use

```
git diff HEAD HEAD^
```

or two commits ago

```
git diff HEAD HEAD~2.
```

When your staging area is set up the way you want it, you can commit your changes. Remember that anything that is still unstaged – any files you have created or modified that you haven't run git add on since you edited them – won't go into this commit. They will stay as modified files on your disk. When git status tells everything you want to commit was staged, you are ready to commit. Type

```
git commit -a -m 'Your commit log entry'.
```

It is mandatory to add a log

message, git won't let you commit without it.

If you want to commit one file instead of all your changes, use

```
git commit myFile -m 'Your commit log entry'.
```

Moving and removing files

To remove a file from Git, you have to remove it from your tracked files (more accurately, remove it from your staging area) and then commit. The

```
git rm obsoleteFile
```

command does that, and also removes the file from your working directory so you dont see it as an untracked file the next time around. If you want to rename a file in Git, use mv.

```
git mv oldName newName
```

Viewing history

The most basic tool to view old commit logs is the

```
git log
```

command. By default, with no arguments, git log lists the commits made in that repository in reverse chronological order. It lists each commit with its SHA-1 checksum, the author's name and email, the date written, and the commit message. One of the more helpful

options is -p, which shows the difference introduced in each commit. You can also use -2, which limits the output to only the last two entries

```
git log -p -2
```

This is very helpful for code review or to quickly browse what happened during a series of commits that a collaborator has added.

Undoing things

One of the common undos takes place when you commit too early and possibly forget to add some files, or you mess up your commit message. If you want to try that commit again, you can run commit with the –amend option

```
git commit --amend
```

Example 6:

```
git commit -m 'initial commit'
git add forgottenFile
git commit --amend
```

You end up with a single commit. The second commit replaces the results of the first.

If you realize that you don't want to keep your changes, use

```
git checkout modifiedFile
```

It's important to understand that git checkout [file] is a dangerous command. Any changes you made to that file are gone, you just copied another file over it. Don't ever use

this command unless you absolutely know that you don't want the file.

Remote repositories

Remote repositories are versions of your project that are hosted on the Internet or network somewhere. Collaborating with others involves managing these remote repositories and pushing and pulling data to and from them when you need to share work.

Example 7:

```
  git clone $HOME/public/html/Test.git
  git remote add public
$HOME/public/html/Test.git

  git remote
```

If you have more than one remote, the command lists them all. For example, a repository with multiple remotes for working with several collaborators might look something like this

```
lxplus0043 > git remote
alexandros
cristina
erik
joona
lauri
matti
origin
public
ritva
sami
santeri
stefan
tapio

lxplus0043 >
```

This means we can pull contributions from any of these users pretty easily. To get data from your remote projects, you can run

```
git fetch [remote-name]
```

Example 7:

I have two clones of the same repository, I've made changes in one and pushed the changes. I cant push from the second unless I fetch the changes and merge them

```
git fetch origin
git merge origin/master
```

When you have your project at a point that you want to share, you have to push it upstream. The command is

```
git push public
```

or when doing it for the first time

```
git push public
refs/heads/master:refs/heads/master
```

Branches

Branching means you diverge from the main line of development and continue to do work without messing with that main line. A branch in Git is simply a lightweight movable pointer to commits. Creating a new branch creates a new pointer for you. Let's create a new branch called 'test'

```
git branch test
```

You can see the new branch with

```
git branch -a
```

and switch into than branch

```
git checkout test
```

This moves HEAD to point to the branch 'test'. Git branch command shows the current branch with an asterisk.

```
lxplus0043 > git branch -a
* test
  master
```

Now when you commit a change, it will appear only in the test branch, not in the master branch. So from this point on the test and master branches will diverge from each other. If you run

```
git log --oneline --decorate --graph
--all
```

it will print out the history of your commits, showing where your branch pointers are and how your history has diverged.

Merging

Once you have the development in the 'test' branch ready, you can merge it with the master branch. First checkout to the master branch and the merge the development branch into it

```
git checkout master
git merge test
```

Git creates a new snapshot that results from the merge and automatically creates a new commit

that points to it. This is referred to as a merge commit, and is special in that it has more than one parent.

Sometimes, then two branches have been developed in parallel, there are changes in the same part of the code. This creates a merge conflict. In a merge conflict git won't be able to merge the branches, so the conflict must be solved by hand by the user, and then committed. When conflicting, git does not automatically create a new merge commit. Command 'git status' shows the merge conflicts which has not been resolved, they are listed as unmerged. Edit the conflicting files by hand to decide which version is correct, and type

```
git add conflictingFile
git commit
```

to finalize the merge commit. The commit message is automatically created. You can modify that message with details about how you resolved the merge if you think it would be helpful to others looking at this merge in the future.

Rebasing

In Git, there are two main ways to integrate changes from one branch into another: the merge and the

rebase. With the rebase command, you can take all the changes that were committed on one branch and replay them on another one. If you need to merge your changes not only to the current branch, but other branches as well, you can use rebase, especially if you cannot use merge.

Example 8:

I have a commit for CMSSW_7_5_X, but I need the same change also in CMSSW_7_4_X. I cant merge my 75X branch into 74X branch, I only need the changes I made for 75X.

```
git branch 74X
git rebase --onto 74X 75X

git push my-cmssw 74X
```

Cherry picking

If your changes are only in about one commit, cherry-picking is a good option.

```
git cherry-pick <commit-ID>
```

CMS-Git

Documentation about CMS specific git commands can be found in http://cms-sw.github.io/faq.html.

If you are in a CMSSW area with cmsenv set, you can checkout code by

```
git cms-addpkg <packageName>
```

or

```
git cms-addpkg -f <file>
```

with 'file' containing a list of packages. You can checkout dependent packages by

Git

```
git cms-checkdeps -a
```

By default the current branch is set to the release tag

```
lxplus0104 > git branch
* from-CMSSW_X_Y_Z
```

The name of the branch is meant to show you the tag which was used as a base for the branch. You should now create your own branch, switch to it, and continue development there. The name of the new branch should remind you what you were doing in that particular branch.

```
lxplus0104 > git checkout -b myNewBranch
Switched to a new branch 'myNewBranch'
```

Modify the code, and commit. Make sure you have registered to GitHub and that you have provided them a ssh public key to access your private repository. Push your branch to GitHub

```
lxplus0104 > git push my-cmssw myNewBranch
```

Your private repository can be found at https://github.com/<username>/cmssw . You need to push your changes to make them available to others.

A pull request is a request to merge your changes in your repository to the official CMSSW repository. A pull request is made from your GitHub page by clicking on the "Pull request" button.

Latex

LaTeX is the tool to write your papers, and at this stage of your studies you should already know how to use it. If not, then learn it now! LaTeX documentation can be found in literature and in the web, google gives you several options for getting the documentation. You might try e.g.

http://www.ctan.org/tex-archive/info/lshort/english/lshort.pdf

https://xml.web.cern.ch/textproc.html

Here we concentrate on two tools which you might need in the future: BibTeX and feynMF. The first is an easy way of managing your references, and the second one is a tool to draw Feynman graphs.

BibTeX

When using BibTeX, LaTex is managing your bibliography for you. It puts your references in the correct order, and it does not show references you have not used. This way you can have the same custom bib file for every paper you write, you just add more references when needed. You need two files, a bib file, and a style file. Here is an example style file based on JHEP style
http://cmsdoc.cern.ch/~slehti/lesHouches.b

The bib file you have to write yourself, or you can use INSPIRE automated bibliography generator http://inspirehep.net to make the

Latex

entry for yourself. Here is an example reference from a bib file:

@Article{L1_TDR,
title = "The Level-1 Trigger",
journal = "CERN/LHCC 2000-038",
volume = "CMS TDR 6.1",
year = "2000"
}

The citation (\cite{L1_TDR}) in the text works as usual. In your tex file you need to have \bibliographystyle{lesHouches}. BibTeX is run like LaTeX: latex-bibtex-latex-latex-dvips.

feynMF

feynMF is a package for easy drawing of professional quality Feynman diagrams with METAFONT. The documentation for feynMF can be found in

http://www.ctan.org/pkg/feynmf

Instructing LaTeX to use feynMF is done including the feynmf package: (files feynmf.sty and feynmf.mf needed)
\usepackage{feynmf}

Processing your document with LaTeX will generate one or more METAFONT files, which you will have to process with METAFONT. The METAFONT file name is given in the document in fmffile environment:

$\backslash$begin{fmffile}{$<$ *METAFONT − filename* $>$}
...

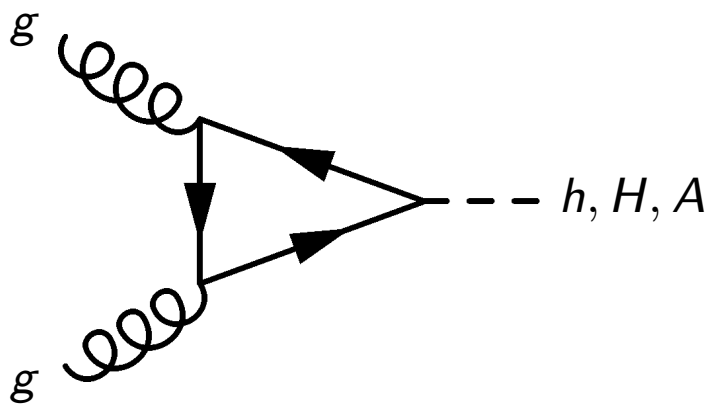
$\backslash$end{fmffile}

Latex

Example 9:

The following code produces the figure below:

```
\begin{fmffile}{triangle_loop}
\begin{figure}{h}
\centering
\parbox{50mm}{
\begin{fmfchar*}(100,60)
    \fmfleft{i1,i2}
    \fmfright{o1}
    \fmf{curly,tension=0.5}{i1,v1}
    \fmf{curly,tension=0.5}{i2,v2}
    \fmf{fermion,tension=0.1}{v2,v1}
    \fmf{fermion,tension=0.3}{v1,v3}
    \fmf{fermion,tension=0.3}{v3,v2}
    \fmf{dashes}{v3,o1}
      \put(0,60){\small $g$}
      \put(0,-5){\small $g$}
      \put(100,28){\small $h$, $H$, $A$}
\end{fmfchar*}}
\end{figure}
\end{fmffile}
```



Make

The make utility automatically determines which pieces of a large program need to be recompiled and issues command to recompile them. However, make is not limited to programs, it can be used to describe any task where some files must be updated automatically from others whenever the others change.

To run make, you must have a makefile. Makefile contains the rules which make executes. The default names for the makefiles are GNUmakefile, makefile and Makefile (in this order). If some other name is preferred, it can used with command make -f myMakefile

Make

A documentation about make can be found in

http://www.gnu.org/software/make/manual/make.html

A makefile is an ASCII text file containing any of the four types of lines:

- target lines
- shell command lines
- macro lines
- make directive lines (such as include)

Rules

Target lines tell make what can be built. Target lines consist of a list of targets, followed by a colon (:), followed by a list of dependencies.

Although the target list can contain multiple targets, typically only one target is listed.

Example 10:

Makefile

```
clean:
        rm -f *.o
```

The clean command is executed by typing

```
$ make clean
```

Notice that the shell command line after the semicolon has a tab in front of the command. If this is replaced by blanks, it won't work! This applies to every line which the target is supposed to execute.

Make

Dependencies are used to ensure that components are built before the overall executable file. Target must never be newer than any of its dependent targets. If any of the dependent targets are newer than the current target, the dependent targets must be made, and then the current target must be made.
Example 11:

```
foo.o: foo.cc
        g++ -c foo.cc
```

If foo.o is newer than foo.cc, nothing is done, but if foo.cc has been changed so that the timestamp of file foo.cc is newer than the timestamp of foo.o, then foo.o target is remade.

One of the powerful features of the make utility is its capability to specify generic targets. Suppose you have several cc files in your program. Instead of writing every one object file a separate rule, one can use a suffix rule:

```
.cc.o:
        g++ -c $<
main: a.o b.o c.o d.o
        g++ a.o b.o c.o d.o
```

Here $< is a special built-in macro, which substitutes the current source file in the body of a rule. When main is executed, and the timestamp of the object files is newer than that of exe file main, the dependent

Make

targets a.o b.o c.o d.o are made using the suffix rule: if the corresponding cc file is newer, new object file is made. When all object files needing updating are made, then the exe file main is made.

Macros

In the above example the object files were typed in two places. One could use a macro instead:

```
OBJECTS = a.o b.o c.o d.o
main: $(OBJECTS)
        g++ $(OBJECTS)
```

The function call syntax is $(*ftion args*). There are several functions available for various purposes: text and file name manipulation, conditional functions etc.

Example 12: let's assume that the program containing the source files a.cc, b.cc, c.cc and d.cc are located in a directory where there are no other cc files. So every file ending .cc must be compiled into the executable. One can use functions wildcard, basename and addsuffix.

```
files = $(wildcard *.cc)
filebasenames = $(basename
$(files))
OBJECTS = $(addsuffix
.o,$(filebasenames))
```

Here the function wildcard gives a space separated list of any files in the local directory matching the

Make

pattern *.cc. If you now modify your program to include a fifth file e.cc, your Makefile will work without modifications.

Example 13: a Makefile

Notice the usage of macros $(CXX) and $(MAKE). What does the @ do? If new line is needed, one can make the break with \.

```
files = $(wildcard ../src/*.cc ../src/*.cpp *.cc *.cpp)
OBJS = $(addsuffix .o,$(basename $(files)))

OPT = -O -Wall -fPIC -D_REENTRANT

INC = -I$(ROOTSYS)/include -I. -I../src \
-I../src/HiggsAnalysis/MssmA2tau2l/interface

LIBS = -L$(ROOTSYS)/lib -lCore -lCint -lHist -lGraf -lGraf3d
-lGpad \
-lTree -lRint -lPostscript -lMatrix -lPhysics -lpthread -lm -ldl \
-rdynamic

.cc.o:
        $(CXX) $(OPT) $(INC) -c $^  -o $@

.cpp.o:
        $(CXX) $(OPT) $(INC) -c $^  -o $@

all:
        @$(MAKE) –no-print-directory All
All:
        @$(MAKE) compile; $(MAKE) analysis.exe

compile: $(OBJS)

analysis.exe: $(OBJS)
        $(CXX) $(LIBS) -O $(OBJS) -o analysis.exe

clean:

        rm -f $(OBJS)
```

Sed

Sed is a stream editor. A stream editor is used to perform basic text transformations on an input stream, which in most cases is a file. Normally sed is invoked like this:

```
sed 's/hello/world/' input.txt
```

Here sed is searching from file input.txt a pattern 'hello' and

sed

changing it into 'world'. You can easily practise to get the script working by writing the result on screen, and after it works, change the file by using option -i

```
sed -i 's/hello/world/' input.txt
```

Example 14: You have downloaded a tarball, and installing it with ./configure && make. You discover a problem with the makefile, a compiler option -fPIC is missing. You dont want to edit the file 'configure' by hand, since all changes are overwritten when unpacking the tarball. You want to add the correction in *your* Makefile (which is downloading and compiling the package in question). So we use sed.

Your original Makefile contains:

```
thepackage:
        wget http://url.to/thepackage.tgz
        tar xfz thepackage.tgz
        cd thepackage
        ./configure
        make
```

In the file 'configure' you want to change the line starting with string 'CFLAGS', and replace from the same line string '-Wall' with '-Wall -fPIC', so you add the sed command in your Makefile before running ./configure:

```
thepackage:
        wget http://url.to/thepackage.tgz
        tar xfz thepackage.tgz
        cd thepackage
        sed -i '/CFLAGS/ s/-Wall/-Wall -fPIC/' configure
        ./configure
        make
```

If the file 'configure' sets flags in only one place, the sed command can be simplified:

```
sed -i 's/-Wall/-Wall -fPIC/' configure
```