
Numerical Algorithms for Physics

Andrea Mignone
Physics Department, University of Torino
AA 2018-2019

Course Purpose

- Physics is often described by equations that cannot be solved analytically or cannot be easily expressed in closed forms.
- Popular examples are
 - Pendulum: $\frac{d^2\theta}{dt^2} = -\sin\theta$
 - Parabolic motion with air drag: $\frac{d\vec{v}}{dt} = \vec{g} - \kappa v \vec{v}$
 - Atoms with many electrons, time-dependent fluid dynamics, N-body problems, etc...
- In this course, you will learn how to solve scientific problems by writing (from scratch) a computer program.
- We will use C++ as our primary language, although only in its basic form (no object-oriented programming):
 - Understand how a computer program (code) works;
 - Beware of the limitations;
 - Interpret, analyze and visualize results.

Course Objectives & Requisites

- Although far from being complete, the course aims at teaching basic numerical methodology and will cover the following topics:
 - Definite and indefinite integrals (also known as *numerical quadrature*);
 - Random numbers and (a primer to) Monte Carlo methods ;
 - Root finder methods for nonlinear equations;
 - Ordinary Differential Equations (ODE), initial & boundary value problems;
 - Numerical differentiation;
 - Linear systems of equations;
 - Elliptic partial differential equations (PDE);
- Pre-requisites:
 - Mechanics, electromagnetism, quantum physics;
 - Acquaintance with C / C++ programming language;
 - Some know-how of Linux system.

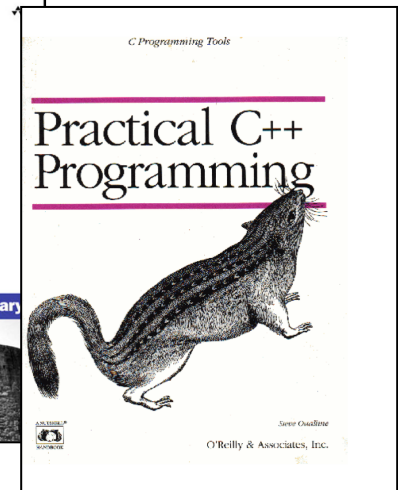
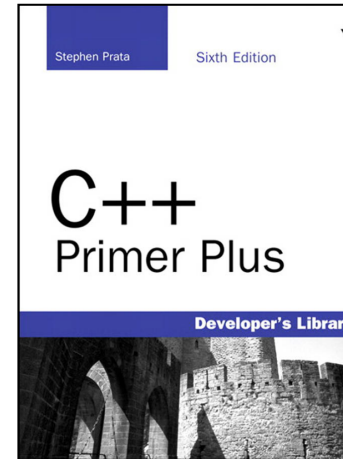
Course Evaluation & Other Info

- Attendance to classes is strongly required.
- There're few exceptions: schedule incompatibility (report to me), overlapping with other courses.
- Lectures & online material at
http://personalpages.to.infn.it/~mignone/Algoritmi_Numerici/
- Final grade will be established on: i) learning ability shown during lectures and ii) (optionally) a supplementary project of your choice to be delivered no later than one year from the beginning of the course.
- You are free to work on your laptop, if you know what you're doing.

Suggested Textbooks

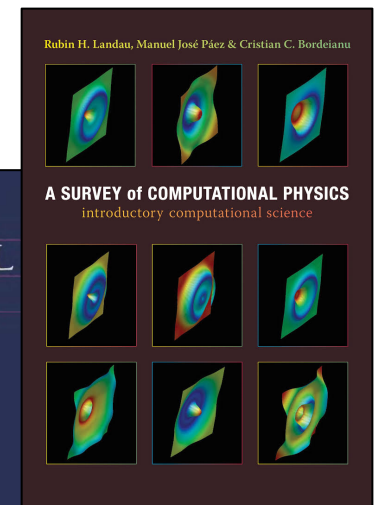
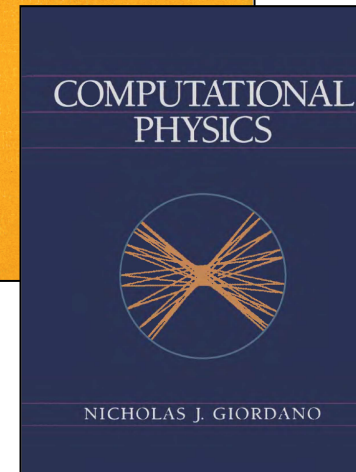
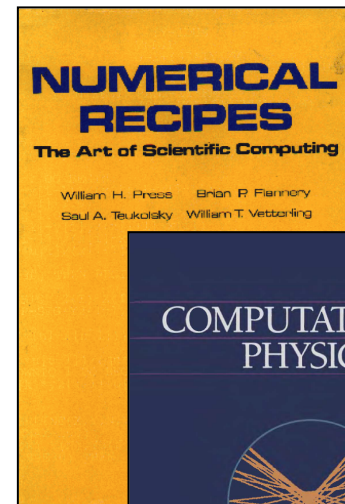
Reference **textbooks on C++:**

- “C++ Primer Plus (Developer’s Library)”
(6th edition) by S. Prata
- “Practical C++ Programming”
(2nd edition) by S.Oualline



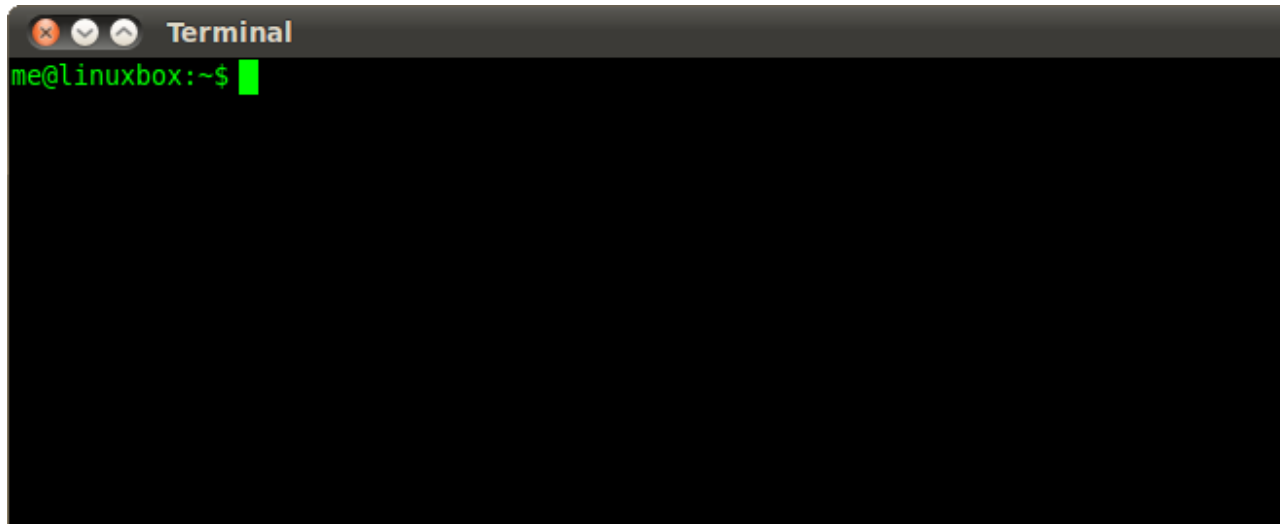
Reference **textbooks on numerical methods:**

- “Numerical Recipes” ...
- “A Survey of Computational Physics”
Landau
- “Computational Physics”



Unix Shell

- A Unix shell is a command-line interpreter (CLI) that provides an interface to the UNIX system.



- It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.
- A shell is defined by its environment in which we can run our commands, programs, and shell scripts. There are different flavors of shells with their own set of recognized commands and functions.
- Most likely, the bash shell will be the default on your system (other possibilities are csh, tcsh, zsh, etc...).

Handling directories: `ls`, `pwd`, `cd`

- The command `ls` is used to list the content of a directory. Additional command line options (such as `-l`, `-lt`, `-lh`) can be given, see the man page (`man ls`)

```
> ls -l
-rw-r--r--  1 mignone  staff   298 Apr 16 16:22 conta.cpp
-rwxrwxrwx  1 mignone  staff 1288 Apr 12 19:27 factorial.cpp
-rwxrwxrwx  1 mignone  staff  979 Apr 12 18:59 guess.cpp
-rwxrwxrwx  1 mignone  staff  745 Apr 16 16:45 operations.cpp
-rwxrwxrwx  1 mignone  staff  574 Apr 16 15:35 welcome.cpp
```

- The `pwd` command prints the current working directory (`$HOME`);
- The `cd` command is used to change directory:

```
> cd My_Folder  # change directory to My_Folder
> cd ../        # go one level up
```

- The `mkdir` command is used to create a new directory:

```
> mkdir New_Folder  # create directory New_Folder
```

Handling files: cp, mv, rm

- cp can be used to copy files and directories from one location to another, e.g.

```
> cp source dest           # copy 'source' to 'dest'  
> cp file ../              # copy 'file' one level up  
> cp file1 file2 file3 My_Dir/ # copy three files to the directory 'My_Dir'
```

- mv can be used to rename or move a file from one location to another, e.g.

```
> mv source dest           # rename 'source' to 'dest'  
> mv ...                   # copy three files to the directory 'My_Dir'
```

- rm is used to delete files:

```
> rm file1                 # delete 'file1'  
> rm ./*                   # remove all files in the current direcoty
```

Attention: remove command cannot be undone !

Using wildcards

- Wildcards are used to work with multiple files:

- ? (*question mark*): this represents any single character, example

```
> ls fo?.c      # may list fo1.c fo2.c fo3.c, foo.c, etc...
```

- * (*asterisk*): this represents any number of characters (including zero). Example:

```
> ls fo*.c      # may list fo12.c fool.c fokker.c, etc...
```

Useful Tips & Tricks

- **Linux command are case-sensitive.**
- **Finding previous command:** pressing the Up keyboard key will cycle through the last Linux commands we successfully used, in order. No failed commands will show here.
- **Use Tab to autocomplete:** the <Tab> button on the keyboard automatically fills in partially typed commands (or files names). It save huge time.

Example:

- If we want to delete a file named "whydidIgivethisfilesuchalongname", we just need to type "rm w" and pressing Tab will automatically complete the rest of the filename.

Note: If there are more than files that begin with the same letters, e.g.

"whydidIgivethisfilesuchalongname" and "whydidIeatsomuch", pressing Tab on "rm w" will autocomplete the common "whydidI".

Customizing Bash Environment

- Bash environment can be customized by editing '.bashrc' file in your \$HOME directory (on Mac OS, this may be found under '.profile')
- For instance,

```
source $HOME/Lib/Scripts/utility.sh

export PS1="\[\e[32;1m\][\u@\h] \w\[\e[0m\]\n> "

export PATH= <new paths here>:$PATH

alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'
```

Your 1st program: Hello World!

- Use your editor to open a new file 'welcome.cpp' and type

```
#include <iostream>           // Preprocessor directive

int main()                   // Function Header
{
    using namespace std;     // Make definitions visible
    cout << std::endl;
    cout << "-----+" << endl;
    cout << "          Hello world !          +" << endl;
    cout << "          +" << endl;
    cout << "+ Welcome to the Numerical Algorithm Class +" << endl;
    cout << "+-----+" << endl;
    return 0;
}
```

- Save files and compile with

```
> g++ welcome.cpp -o welcome
```

- This will compile 'welcome.cpp' and produce the executable 'welcome'.
- Run the program by typing, at the terminal,

```
> ./welcome
```

Program Structure

- C++ programs are constructed from building blocks called *functions*.
- Organize a program into major tasks and then design separate functions to handle those tasks.
- The previous program consists of a single function named `main()` with the following elements:

- Comments, indicated by the `//` prefix
- A preprocessor `#include` directive
- A function header: `int main()`
- A `using namespace` directive
- A function body delimited by `{` and `}`
- Statements that uses the C++ `cout` facility to display a message
- A `return` statement to terminate the `main()` function

```
#include ...
```

```
Func1(){  
    statements  
}
```

```
Func2(){  
    statements  
}
```

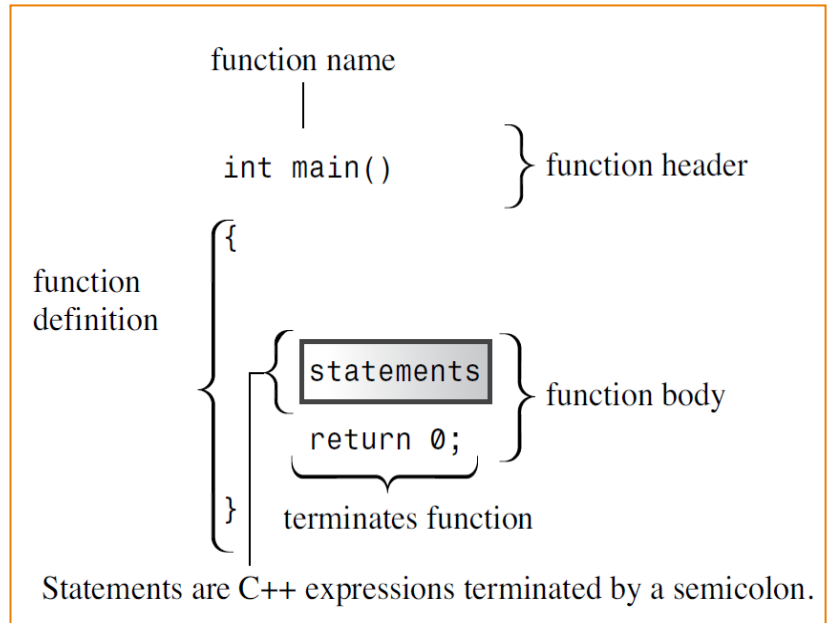
```
main(){  
    statements  
}
```

The main() function

- The main() function has the following fundamental structure:
- These lines state that there is a function called main() and describe how the function behaves.: together they constitute a *function definition*.

```
int main()
{
    statements
    return 0;
}
```

- It consists of
 - the *function header* `int main()`: a capsule summary of the function's interface with the rest of the program,
 - the *function body*: the portion enclosed in braces ({ and }), which represents instructions to the computer about what the function should do.



- In C++ each complete instruction is called a statement.
- Statement must terminate with a semicolon,

The C++ Preprocessor and the *iostream* File

- C++ (like C) uses a *preprocessor*: a program that processes a source file before the main compilation takes place.

- Preprocessor directives begin with #:

```
#include <iostream> // a PREPROCESSOR directive
```

- The `#include` directive causes the preprocessor to add the contents of the `iostream` file to your program at compilation time: in practice the content of the `iostream` file will replace the `#include <iostream>` line.
- This directive causes, a typical **preprocessor** action which **adds** or **replaces** text in the source code before it's compiled.
- The content of the `iostream` is added for communication between the program and the outside world. The I/O in `iostream` refers to input - information brought into the program - and to output - information sent out from the program.
- C++' I/O scheme involves several definitions found in the `iostream` file.

Practice Session #1

- count.cpp: write a program that prints the integers numbers from 1 to 10, using for and while loops.
- Modify the previous code to print only odd numbers.
- operations.cpp: given two float numbers and two integer numbers, perform simple arithmetic operations (+, -, *, /) → watch out integer division !

Functions

- Consider a simple program that computes the sum of two double precision numbers and write a separate function to perform the sum.
- An example is given here:

Function declaration (or prototype): does for functions what a variable declaration does for variables: it tells what types are involved.

Function can be called from anywhere now

The Function definition implements the actual body of the function.

```
#include <iostream>
#include <cmath>

double Sum(double, double);

int main() {
    using namespace std;
    double a, b, c;


    a = 1.5;
    b = 3.7;
    c = Sum(a,b);
    return 0;
}

double Sum(double x, double y)
{
    return x+y;
}
```

Passing Arguments to Functions

- Arguments to function can be passed by value or by reference.
- By value: arguments to the function are copied into the variables represented by the function parameters. For example, take:

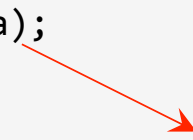
```
c = Sum(a,b);  
  
double Sum(double x, double y)  
{  
    ...  
}
```



In this case, x and y are copies of the variables a and b: any modification of these variables within the function has no effect on the original value.

- By reference: what is passed is no longer a copy but the variable itself: any modification to the arguments is reflected in the variables passed as arguments in the call:

```
AddOne(a);  
  
void AddOne(double& x)  
{  
    x = x + 1.0;  
}
```



In this case, a (= x) is modified and after the call it has been incremented by one.

Practice Session #2

- sum.cpp: compute the sum of two integers (or floats)
- quotient.c: write a function that computes the quotient and remainder of the division of two integers, a and b. Example
 - $10/7 = 1$ remainder 3
 - $13/6 = 2$ remainder 1, etc..
- add_one.cpp: add one to a number.
- factorial.cpp: compute the factorial of a number, using integer and floating point variables.
- arrays.cpp: compute the average, variance and standard deviation of an array.

Typical Errors, *good and bad programming habits*

I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code



Indentation

```
int i;  
  for (i=0; i < 10; i++){  
if (i == 5)  
    {  
    cout << "I am number five" << endl;  
    }  
}
```

NO !!

```
int i;  
  
for (i=0; i < 10; i++){  
    if (i == 5) {  
        cout << "I am number five" << endl;  
    }  
}
```

YES !!



*Indentation clarifies the link between control flow constructs such as conditions or loops, and code contained within and outside of them.
The code will be i) easy to read, ii) easy to understand, iii) easy to modify, iv) easy to maintain.*

Type Casting ($1/2 = 0$)

```
int i,n;  
double q;  
  
for (i = 0; i < n; i++) {  
    q = i/n;  
    q = (double)(i/n);  
}
```

NO !!



```
int i,n;  
double q;  
  
for (i = 0; i < n; i++) {  
    q = (double)i/(double)n;  
}
```

YES !!



Mixing lower- and upper-case

```
int I;  
double a[i];
```

```
Cin >> I;
```

NO !!

```
int i;
```

```
cin >> i;  
double a[i];
```

YES !!



The == sign and the = sign

```
for (i = 0; i < 10; i++){  
    if (i = 5) break;  
}
```

NO !!

```
for (i = 0; i < 10; i++){  
    if (i == 5) break;  
}
```

YES !!



Floating Point arithmetic

```
double x=0.0;

while (1){
    x += 0.1;
    if (x == 10.0) break;
}
cout << "x = %f\n" << x << endl;
return 0;
```

NO !!



```
double x=0.0;

while (1){
    x += 0.1;
    if ( fabs(x-10.0) < 1.e-6) break;
}
cout << "x = %f\n" << x << endl;
return 0;
```

YES !!



Rounding error

```
double x, dx;  
for (x = 0.0; x <= xend; x +=dx) {  
    ...  
}
```

```
// This will yield (xend = 1.0, dx = 0.1):  
//  
// x = 0.0, 0.1, 0.19999, 0.200003
```

NO !!



```
int i;  
  
for (i = 0; i < N; i++){  
    x = x0 + i*dx  
    ...  
}
```

YES !!



Errori Tipici

```
int i;  
for (i=0; err > acc; i++){  
  
}
```

NO

```
int i=0;  
err = 1.e40;  
while (err > acc){  
    ...  
    err =...  
    i++  
}
```

```
while (1){  
    ...  
    err =...  
    if (err < acc) break;  
    i++  
    if (i > MAX_ITER) exit  
}
```

YES