Programming Best Practices

R. Pastor-Satorras

Departament de Física
UPC

Eines Informàtiques Avançades / EIA

Let the computer do the work

Some obvious facts:

1. Computers are fast, people is slow
2. People gets bored/tired doing slow, repetitive things
3. Bored/tired people make more mistakes

Avoid like hell doing repetitive tasks!

In scientific research, we spend time doing repetitive things: labeling files, moving them around, computing simple things, etc

Computers are very good at doing those repetitive things

Use some time to write scripts (bash, python, etc) that automatize the boring operations that we sometimes find ourself doing

Script automatization is easier on the terminal than on a GUI

The terminal is your friend: learn how to use it!

Use the appropriate tools for each problem

You can learn a single programming language and use it for everything

This is doable, but it pays off, specially in a Unix-like environment, to know the tools available and use them effectively

Example

How to compute the average value of the third column of a data file?

Options:

Use the appropriate tools for each problem

You can learn a single programming language and use it for everything

This is doable, but it pays off, specially in a Unix-like environment, to know the tools available and use them effectively

Example

How to compute the average value of the third column of a data file?

Options:

1. Write a Fortran mini-program that does that

Use the appropriate tools for each problem

You can learn a single programming language and use it for everything

This is doable, but it pays off, specially in a Unix-like environment, to know the tools available and use them effectively

Example

How to compute the average value of the third column of a data file?

Options:

1. Write a Fortran mini-program that does that
2. Use the Unix tool `awk`
   ```
   $:> awk '{n++;sum += $3} END {print sum/n}' datafile.dat
   ```
   Just in one line!

Advice

Spend some time learning the Unix shell tools

Learn to love the command line

- `sort`: sort lines of text file, either by character or by number
- `cut`: cuts out selected portions of each line of a file. For example, select given columns in a file
- `tail`: display the last part of a file
- `time`: time command execution. Useful to benchmarking execution of programs
- `jot`: generates sequential or random numbers
- `datamash`: command-line statistical calculations on a data file. Example: compute the average value of the second column of a datafile

      :> datamash -W mean 2 < data.dat

  Simpler than awk.
  `datamash` can compute more sophisticated things, such as standard deviation, kurtosis, etc.

Do not reinvent the wheel

Many problems in scientific computing involve tasks that are common in the field

1. Generating random numbers
2. Solving equations (algebraic or differential)
3. Eigenvalues and eigenvectors
4. Special functions (Bessel, etc)
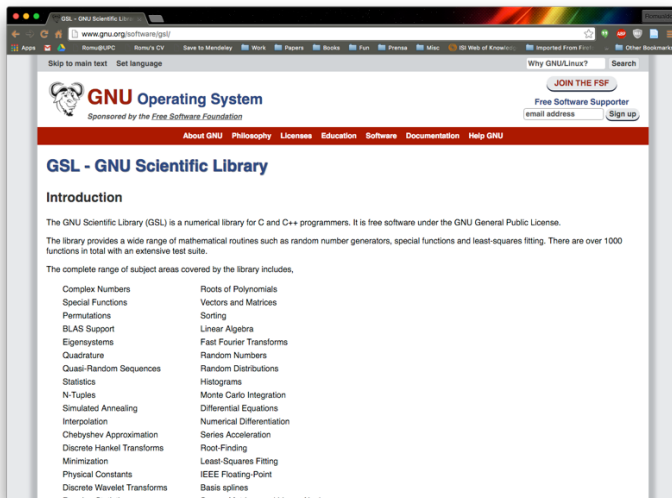5. Optimization (finding maxima or minima) ...

Many of this tasks have been considered and solved by other people, in much smarter and efficient ways you can think of

Advice
Do not reinvent the wheel, google for it instead. Unless you are lerning...
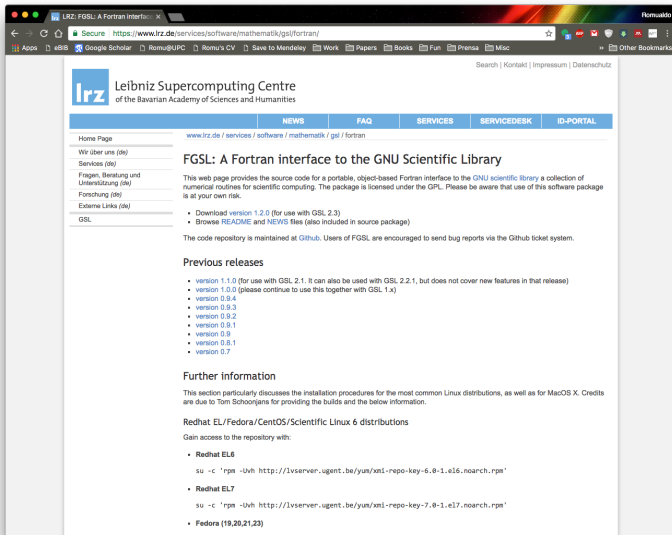
The GNU Scientific Library (GSL)

For C, C++

The GNU Scientific Library (GSL)

For Fortran

An example: Numerical integration of $\int_0^1 \frac{\sin(x)}{x}\, dx$

```fortran
module integral
  implicit none

contains
  function f( x, params ) bind(c)
    use, intrinsic :: iso_c_binding
    implicit none

    real( kind = c_double )          :: f
    real( kind = c_double ), value :: x
    type( c_ptr ), value           :: params

    f = sin( x ) / x
  end function f
end module integral
```

```fortran
program bsp
  use fgsl
  use integral
  use, intrinsic :: iso_c_binding
  implicit none

  real( kind = fgsl_double )   :: result, error
  integer( kind = fgsl_size_t) :: neval
  integer( kind = fgsl_int)    :: i
  type( fgsl_function )         :: func

  func = fgsl_function_init( f, c_null_ptr )

  i = fgsl_integration_qng ( func,           &
      0.0_fgsl_double,       &
      1.0_fgsl_double,       &
      1e-9_fgsl_double,      &
      1e-9_fgsl_double,      &
      result, error, neval )

  write( *, * ) "Result =", result

end program bsp
```

Compiling code with GSL

In Fortran, you need to use the appropriate flags to tell the compiler where the library is installed

For a correct installations, the best way is to use the `pkg-config` Unix utility:

```
gfortran -o myprog.exe `pkg-config --cflags fgsl` myprog.f08  \
      `pkg-config --libs fgsl`
```

`pkg-config` will automatically fill the necessary compiler flags

Alternatively, you can also use `libtool` to link the executable

A note: Random number generators

In many situations, we need to use random number generators (RNG) to generate sequences of random numbers

- Monte-Carlo simulations
- Random initial conditions of a molecular dynamics simulation

For a very small number of random numbers, you can use the built-in RNG in most programming languages (eg. Fortran)

Never use it for large-scale Monte-Carlo simulations, where many random numbers must be extracted

Powerful and safe RNG have been developed. In scientific simulations, the most commonly used is the Mersenne-Twister RNG

You don't have to code MT yourself (it is quite complex and performs better in C): You can link it from the GSL library

Write programs for people, not for computers

Code has to be understood by the computer, in order to produce the desired results

But, more importantly, it must be understood by other people that might look at it to learn it, improve it

Most importantly, it must be understood by **yourself**, when you look at it in a week's or month's time

Advice

Make your code readable

Making readable code

1. Indent your code properly to help identify structure blocks in the code
   - Write different blocks at different levels of indentation
     - Some languages, like Python, forces to use indentation to declare blocks
     - Other languages (C/Fortran) are inmune to white space: Use indentation for clarity

An Example: Bare code. What does it do?

```fortran
program main
implicit none
integer niter,i,j
integer seed
real*4 count
real *8 x,y,pi(100),z
real*8 rand
seed = 35791246
call srand (seed)
do j= 1,100
print *, j
niter = niter+100
count =0
do i=1,niter
x=rand()
y=rand()
z= x*x +y*y
if (z .le. 1) then
count =count+1
end if
end do
pi(j)= count/niter*4.
print *, "Iter ", niter
print *, "Value ", pi(j)
end do
end program main
```

An Example: Indenting

```fortran
program main
implicit none
integer niter,i,j
integer seed
real*4 count
real *8 x,y,pi(100),z
real*8 rand
seed = 35791246
call srand (seed)
do j= 1,100
print *, j
niter = niter+100
count =0
do i=1,niter
x=rand()
y=rand()
z= x*x +y*y
if (z .le. 1) then
count =count+1
end if
end do
pi(j)= count/niter*4.
print *, "Iter ", niter
print *, "Value ", pi(j)
end do
end program main
```

```fortran
program main
  implicit none

  integer niter,i,j
  integer seed
  real*4 count
  real *8 x,y,pi(100),z
  real*8 rand

  seed = 35791246
  call srand (seed)

  do j= 1,100
     print *, j
     niter = niter+100
     count =0
     do i=1,niter
        x=rand()
        y=rand()
        z= x*x +y*y
        if (z .le. 1) then
           count =count+1
        end if
     end do
     pi(j)= count/niter*4.
     print *, "Iter ", niter
     print *, "Value ", pi(j)
  end do

end program main
```

Making readable code

1. Use an text editor with syntax highlighting
   - Different colors for different keywords in the code
   - Helps again in identifying the different parts that compose the code

An Example: Indenting + Highlighting

```fortran
program main
implicit none
integer niter,i,j
integer seed
real*4 count
real *8 x,y,pi(100),z
real*8 rand
seed = 35791246
call srand (seed)
do j= 1,100
print *, j
niter = niter+100
count =0
do i=1,niter
x=rand()
y=rand()
z= x*x +y*y
if (z .le. 1) then
count =count+1
end if
end do
pi(j)= count/niter*4.
print *, "Iter ", niter
print *, "Value ", pi(j)
end do
end program main
```

```fortran
program main
  implicit none

  integer niter,i,j
  integer seed
  real*4 count
  real *8 x,y,pi(100),z
  real*8 rand

  seed = 35791246
  call srand (seed)

  do j= 1,100
     print *, j
     niter = niter+100
     count =0
     do i=1,niter
        x=rand()
        y=rand()
        z= x*x +y*y
        if (z .le. 1) then
           count =count+1
        end if
     end do
     pi(j)= count/niter*4.
     print *, "Iter ", niter
     print *, "Value ", pi(j)
  end do

end program main
```

```fortran
program main
  implicit none

  integer niter,i,j
  integer seed
  real*4 count
  real *8 x,y,pi(100),z
  real*8 rand

  seed = 35791246
  call srand (seed)

  do j= 1,100
     print *, j
     niter = niter+100
     count =0
     do i=1,niter
        x=rand()
        y=rand()
        z= x*x +y*y
        if (z .le. 1) then
           count =count+1
        end if
     end do
     pi(j)= count/niter*4.
     print *, "Iter ", niter
     print *, "Value ", pi(j)
  end do

end program main
```

Making readable code

1. Use meaningful variable names
   Avoid using
   1. single letter names, unless to be used in loops
   2. reusing names (in Python you can do that ...)
   3. vague names: data, input, output, do_stuff(), process_files(), params, object ...

   Use instead descriptive names which are long enough to described what is the value stored in the variable
   You can use
   1. Camel Case: EnergyPerParticle
   2. Snake Case: energy_per_uparticle

Beware of Fortran, it is not case sensitive

Advice

Incidentally: Avoid magic numbers!!
Use variables/constants instead (with capital letters, for example)

Choose wisely your text editor

Many good code editors are available for all platforms, supporting facilities to help coding. Do not stick with one that does not support

1. Syntax highlighting
2. Auto-indentation
3. Code completion [extra, extremely very helpful].
   - Do not be afraid of long variable names: Use an editor with code completion!

Viable multiplatform code editors, with all these requirements:

1. vim [free, extremely powerful]
2. emacs [free, extremely powerful]
3. Sublime Text http://www.sublimetext.com/ [not free but usable, quite easy]
4. Atom https://atom.io/ [free, similar to Sublime Text, developed by GitHub]
5. Caret, a Sublime Text inspired editor available as a Chrome extension

Avoid deep nesting

In some pieces of code, we sometimes have to nest different constructions: do, if, etc.

Too many levels of nesting can make the code hard to read. Let us see an example

```
function do_stuff() {
    if (is_writable($folder)) {
        if ($fp = fopen($file_path,'w')) {
            if ($stuff = get_some_stuff()) {
                if (fwrite($fp,$stuff)) {

                    // DO STUFF

                } else {
                    return false;
                }
            } else {
                return false;
            }
        } else {
            return false;
        }
    } else {
        return false;
    }
}
```

```
function do_stuff() {
    if (!is_writable($folder)) {
        return false;
    }

    if (!$fp = fopen($file_path,'w')) {
        return false;
    }

    if (!$stuff = get_some_stuff()) {
        return false;
    }

    if (fwrite($fp,$stuff)) {
        // DO STUFF
    } else {
        return false;
    }
}
```

DRY: Don't repeat yourself

Don't repeat yourself

- If you see that a piece of code that keeps popping up in the same program, encapsulate it inside a function

Benefits:

- Limits the number of bugs: Less code, less bugs
- Allows to implement changes in just one piece of the code, instead of having to fiddle with the whole codebase
- Allows to reuse code
  - ▸ A useful function can be extracted from the code and reused on a different program

An Example (in Python)

```python
# Some dynamical simulation in which you look for
# nearest neighbors in a 1d lattice of size L

# Some code

# look for nearest neighbors
nearest = []
# left
x1 = x -1
if x1 == -1:
    x1 = L - 1
    nearest.append(x1)
# right
x1 = x + 1
if x1 == L:
    x1 = 0
    nearest.append(x1)

# Some more code

# look for nearest neighbors
nearest = []
# left
x1 = x -1
if x1 == -1:
    x1 = L - 1
    nearest.append(x1)
# right
x1 = x + 1
if x1 == L:
    x1 = 0
    nearest.append(x1)
```

An Example (in Python)

```python
# Some dynamical simulation in which you look for
# nearest neighbors in a 1d lattice of size L

# Some code

# look for nearest neighbors
nearest = []
# left
x1 = x -1
if x1 == -1:
    x1 = L - 1
    nearest.append(x1)
# right
x1 = x + 1
if x1 == L:
    x1 = 0
    nearest.append(x1)

# Some more code

# look for nearest neighbors
nearest = []
# left
x1 = x -1
if x1 == -1:
    x1 = L - 1
    nearest.append(x1)
# right
x1 = x + 1
if x1 == L:
    x1 = 0
    nearest.append(x1)
```

```python
def nn_pcb(x, L):
    # left
    nearest = []
    x1 = x -1
    if x1 == -1:
        x1 = L - 1
        nearest.append(x1)
    #right
    x1 = x + 1
    if x1 == L:
        x1 = 0
        nearest.append(x1)

    return nearest

# Some code

nearest = nn_pcb(x, L)

# Some code

nearest = nn_pcb(x, L)
```

KISS: Keep it simple, stupid

Try to start with the simplest implementation, in order to have a working version of your program that runs without problems

Go only for more complex things later on and if you need it, for example to speed up a particularly slow part of the code

Do so only after benchmarking your code to identify bottlenecks

Do not spend a lot of time optimizing things to go from 1 sec to 0.01 secs

KISS: Keep it simple, stupid

Try to start with the simplest implementation, in order to have a working version of your program that runs without problems

Go only for more complex things later on and if you need it, for example to speed up a particularly slow part of the code

Do so only after benchmarking your code to identify bottlenecks

Do not spend a lot of time optimizing things to go from 1 sec to 0.01 secs

"We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**"

— Donald Knuth

Comment your code

Comment your code adequately

- Consider you can be reading it in a few months: Make it easy for yourself to understand what you wrote

Avoid obvious comments: Don't write what code is doing

- Some comments are better replaced by a good variable or function name

Avoid extra-commenting

- More is not always better

Group the blocks of code using white space and comments

Programming technique:

- Comment before coding
    - ▶ Think first about what you want to code, and implement it later

An example

How **not** to comment your code

```python
def fun1(p,a,b):
    return a*np.exp(-1./(b*p)) #form of the function we will use in the fit
m=3;
time1=time.time() #start the clock
numnet=100 # number of different networks generated at each size N
netreal=100 # number of different infection outbreaks on each network
longProblist=20 # number of different probabilities considered
Problist=np.linspace(0.05,0.4,longProblist)
Nlist=np.logspace(1,3,5) # network sizes distributed distributed on logarithmic scale
for N in Nlist:
    variancelist=[]
    mplist=[]
    for Prob in Problist: # per cada probabilitat
        Rmeanlist=[]
        for inet in range(numnet): # genero diferents networks
            g = nx.barabasi_albert_graph(N, m);
            Rireal=[]
            for ireal in range(netreal): # genero diferents casos en la mateixa network
                S=set(g.nodes()); I=set(); R=set(); A=set() # A es el set d'actius
                nodeI=choice(g.nodes()) # escollim un node de la red
                I.add(nodeI) # afegeixo el node als I
                S.remove(nodeI) # trec el node dels S
                veins=set(g.neighbors(nodeI)) # actualitzo el set de ACTIUS ( veins del infectat )
                A=set.union(A,veins) # uneixo el set de actius amb els veins del infectat
                while len(I)>0: # itero fins que el numero de infectats desapareix
                    R=set.union(R,I) # els infectats son recuperats a cada pas de temps
                    I=set()
                    for node in A:   # per cada node actiu
                        dados=np.random.random((1,1))[0]
```

Program modularization and refactoring

When programs are small, it makes sense to store them in a single file

For very large programs, it makes sense to split them into separate files (modules)

Modules can contain:

1. Variables, declarations and constants that are used in all the program, allowing to modify them easily at a single point
2. Groups of related functions

Program modularization and refactoring

Modular programming is a software design technique that emphasizes separating the functionality of a program into independent interchangeable pieces (modules) that contain functions to perform only one aspect of the desired functionality

Modules do not need to know about how something else is done in other modules

- Encapsulation

Modularity, mixed with the use of functions, allows you to:

1. More easily solve a problem by breaking it down into elementary pieces
2. Fix problems and find bugs in specific sections
3. Reuse individual modules (useful functions) on other programs

An example

```fortran
module constants
  implicit none

  real, parameter :: pi = 3.1415926536
  real, parameter :: e = 2.7182818285

contains

  subroutine show_consts()
    print*, "Pi = ", pi
    print*, "e = ", e
  end subroutine show_consts

end module constants
```

```fortran
program module_example
  use constants
  implicit none

  real :: x, ePowerx, area, radius
  x = 2.0
  radius = 7.0
  ePowerx = e ** x
  area = pi * radius**2

  call show_consts()

  print*, "e  to the power 2.0 = ", ePowerx
  print*, "Area of a circle  = ", area

end program module_example
```

Bug hunting

If debugging is the process of removing software bugs, then programming must be the process of putting them in.

—Edsger Dijkstra

The truth is:

1. Programmers write code
2. Programmers aren't perfect
3. Programmer's code isn't perfect
4. It doesn't work perfectly the first time
5. Therefore, we have bugs

The situation is even worse when code is written by scientists, who are not trained programmers

Conclusion
Our code will be plagued with bugs. We better learn to deal with them

Bug hunting strategies

Some bugs are simple mistakes that are obvious to spot and easy to fix

In this best of situations, we are lucky, since just a careful reading of our code will suffice to find them

Debugging code is more an art than a science, that can only be mastered by a lot of experience

Some general strategies can be however formulated

Bug hunting strategies

- Lay traps
  1. You know a point when the code seems correct
  2. You can get to a point where it is invalid
  3. Find places in the code path between these two points, and set traps to catch the problem
  4. Add assertions or tests to check the system state
  5. Add diagnostic printouts to see what's going on

Example

A typical example is given by integer overflow, for example when counting occurrences of a given thing

Check the value of the counter with a `print` statement

Bug hunting strategies

- Use binary chop
    1. Work out the start and end of a chain of events
    2. Partition the code in two halves and work out if the middle point is good or bad
    3. Based on this information, you can narrow down the size of the problematic code to roughly half the size
    4. Repeat this process until you pinpoint the position of the bug
    5. This procedure can speed up the localization of the bug
        - Binary search in time $\mathcal{O}(\log n)$

Bug hunting strategies

- Software archaeology
  1. Look at the historical record of your code, as given by previous versions
  2. Determine the point in the near past when the bug didn't exist
  3. Step forward in time to determined the code change that caused the introduction of the bug.
     - ⋆ Binary chop is best used here
  4. This is a compelling reason to write your code making a series of small changes, using a test-driven approach
     - ⋆ Using a control version system becomes an asset here

When everything else fails: Use a debugger

A debugger is a tool used to test and debug programs

It allows the programmer to stop a program at any point and examine and change the values of the variables

Debuggers are extremely powerful tools, but they are complex to master

They should be used as the last resource to find a bug that is impossible to locate otherwise

Here we will examine the GNU debugger `gdb`

Warning

`gdb` works with no problem on linux or Unix machines
It is quite tricky even to get it installed in Mac OSX
No idea on Windows ...
So run it in the following on the Linux partition

Running gdb

```
:> gdb -help
This is the GNU debugger.  Usage:

    gdb [options] [executable-file [core-file or process-id]]
    gdb [options] --args executable-file [inferior-arguments ...]

Selection of debuggee and its files:

  --args              Arguments after executable-file are passed to inferior
  --core=COREFILE     Analyze the core dump COREFILE.
  --exec=EXECFILE     Use EXECFILE as the executable.
  --pid=PID           Attach to running process PID.
  --directory=DIR     Search for source files in DIR.
  --se=FILE           Use FILE as symbol file and executable file.
  --symbols=SYMFILE   Read symbols from SYMFILE.
  --readnow           Fully read symbol files on first access.
  --write             Set writing into executable and core files.

Initial commands and command files:

  --command=FILE, -x Execute GDB commands from FILE.
  --init-command=FILE, -ix
                      Like -x but execute commands before loading inferior.
  --eval-command=COMMAND, -ex
                      Execute a single GDB command.
                      May be used multiple times and in conjunction
                      with --command.
  --init-eval-command=COMMAND, -iex
                      Like -ex but before loading inferior.
  --nh                Do not read ~/.gdbinit.
  --nx                Do not read any .gdbinit files in any directory.
```

Running gdb

In order to run `gdb` in a program, it must be compiled with the debugging -g option:

```
gfortran -g -ffpe-trap=zero,invalid,overflow,underflow program.f09
```

We can then rub the debugger on our code

```
:> gdb example.exe
GNU gdb (Ubuntu 7.10-1ubuntu2) 7.10
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from example.exe...done.
(gdb)
```

At the (gdb) prompt, we can enter our commands

gdb commands

gdb has many commands. The most important of them are the following:

- `help`: help on gdb commands
- `break <line number>`: Stop the execution at the given line number
- `step`: runs one line of code from the point of the program in which we are
- `run`: run the program being debugged from the beginning
- `print <variable>`: print the value of a variable
- `set <variable = value>`: set the value of a variable
- `watch <variable>`: print a commentary whenever the variable changes its value
- `list`: prints the lines of code around the place in which we are in the program

Exercise

Let us try to debug a very simple program

1. Download the example codes from the Campus Virtual
   - `example_1.f90`
   - `example_2.f90`
   - `example_3.f90`
2. Compile and execute them normally to see how they fail
3. Compile them using the debugging options and run them through `gdb`
4. Investigate the problem in the code using `gdb`. Try to see what is the problem and how you can fix it