

Programming 101

Pier-André Bouchard St-Amant

Queen's university

January 20, 2010

Introduction

Basic programming instructions

Flow control

- Boolean Algebra

- Flow control keywords


Functions

Programming tips

Working in group

What is this all about ?

- ▶ Get an understanding of universal building blocks for algorithms.
- ▶ I will use Matlab (or Octave¹) as a specific language implementation.
- ▶ This will not be an advanced tutorial on programming, neither a tutorial on wonders and specifics of Matlab.

¹As a first approximation, octave is a free matlab. 

What is a program/algorithm ?

- ▶ Computers understand only one language : zeros and ones.
- ▶ A program is a finite sequence of 0 and 1 (instructions), read by the computer, that produces an output (another sequence of 0 and 1).
- ▶ A program implements an algorithm : instructions that, given the same input, will always produce the same output.
- ▶ Thus, a program is specific to a machine and an algorithm is somehow the “blueprint” of a program.
- ▶ An algorithm can thus be written in plain english or in “pseudo-code”, as long as it is specific enough to fit the input/output definition above.

What is a programming language ?

- ▶ Writing sequences of 0 and 1 is unpractical. People invented abstract keywords/rules that represent some unique sequences of 0 and 1 : these keywords/rules are programming languages.
- ▶ Hence, we can write programs in text files with those keywords.
- ▶ It is the job of a compiler/interpreter to convert these text files into strings of 0 and 1.
- ▶ Some examples of programming languages : c++, java, lisp, prolog, fortran, basic, python, PHP, etc.

Meta-steps of programming.

- ▶ Understanding what is to be solved.
- ▶ Conceive the algorithm to solve it (key step)
- ▶ Divide the work.
- ▶ Type the code.
- ▶ Test the code.

What are the challenges of programming for newbies ?

Mainly three :

- ▶ Understanding the basic building blocks of algorithms.
- ▶ Understanding the basic keywords/conventions of a language.
- ▶ Conceive algorithms.

Basic programming instructions

- ▶ Computers can either store some data in memory or perform some operations on data.
- ▶ Data is stored in variables.
- ▶ Interpreters read programs from right to left (saved for parentheses priorities).

Affectation

- ▶ $x = 10$ is an example of affecting some data (the value 10) to some variable (the variable x).
- ▶ Hence, somewhere in the computer RAM, some space with a specific address has been saved for this variable and the value 10 has been stored in it.
- ▶ $x = x + 10$ is a meaningful expression : pick 10, add the value of what is stored in x and store it in x (thus, erasing the previous value).

Operations

- ▶ Most basic operators are always implemented : $+$, $-$, \backslash , $*$, \wedge .
- ▶ There are tons of operators. Which operator is implemented and what it does depends highly on the language used.
- ▶ Some operators are specific to strings of characters, files, numbers, matrices, etc.
- ▶ Thus some operations/instructions are allowed only to specific types of variables.
- ▶ Depending on how picky the programming language might be, bad usage might lead to a failure to be interpreted/compile.
- ▶ Worse, it could be interpreted/compile, but compute meaningless stuff.

Matlab variables and operators

- ▶ Any string that starts with a letter can be used as a variable : *toto*, *x*, *hello_mom*, etc.
- ▶ Matlab variables are matrices.
- ▶ $x = [1, 0.5; 0.5, 1]$ affects the matrix

$$\begin{pmatrix} 1 & 0.5 \\ 0.5 & 1 \end{pmatrix}$$

to x .

- ▶ $x = 1$ affects a scalar “matrix” to x (of value 1).
- ▶ Basic operators $+$, $-$, \backslash , $*$, \wedge applies if they follow basic algebra rules (matrices must have the correct size).
- ▶ $x(1, 1)$ refers to the top east value, etc.

Colon operator

Colons is used to indicate range. For instance :

```
y = 0:10; %generates y = [0, 1, 2, ... , 10]
```

```
y = 0:0.01:10; %generates y = [0, 0.01, ..., 9.99, 10]
```

Let alone, it is also a way to express “all values”. So if :

$$x = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

$x(:,1)$ refers to the first column and $x(1:2,1)$ the first two values of the first column.

Bracket operator

Brackets generally refer to a matrix. For instance :

```
y(:,1) = []; %Affects an empty matrix
```

```
%to the first column of y
```

```
z = [x, y]; %generates a matrix from x and y
```

```
%(which must have the same length)
```

dot operators

The “element by element” operators are very useful. They are `.*`, `.\`, `.^`. For instance, if

$$x = \begin{pmatrix} 1 & 0.5 \\ 0.5 & 1 \end{pmatrix}$$

$$y = \begin{pmatrix} 1 & 0 \\ 2 & 10 \end{pmatrix}$$

$$\Rightarrow x.^y = \begin{pmatrix} 1^1 & 0.5^0 \\ 0.5^2 & 1^{10} \end{pmatrix}$$

$$\Rightarrow x.*y = \begin{pmatrix} 1*1 & 0.5*0 \\ 0.5*2 & 1*10 \end{pmatrix}$$

The `.\` would not be allowed here since it implies division by zero for some element.

Conditionnal operations/affectations

- ▶ Sometimes, we want programs to do stuff only if something is true.
- ▶ We thus need two things : keywords that instructs the program to do stuff if some propositions are true.
- ▶ We need an algebra to deal with thruthness of some propositions.
- ▶ I start with the latter (boolean algebra) and then present the former (flow control).

Boolean algebra

- ▶ Like any algebra, it has variables (propositions) and operators on these variables.
- ▶ A proposition, for instance $A > 0$, can either take two values : true or false.
- ▶ I will use letters p and q to design two distinct propositions.
- ▶ I will present the operators with the matlab/octave convention.
- ▶ Like matlab, I will use 1 for true and 0 for false.

Negation operator (\sim)

The negation operator changes the value of a proposition.

Table: Negation operator

p	$\sim p$
0	1
1	0

For instance, $\sim (x \geq 0)$ is true if $x < 0$.

And operator (&)

The and operator is true only if the two propositions are true.

Table: And operator

p	q	$p \& q$
0	0	0
0	1	0
1	0	0
1	1	1

For instance, $(x > 10) \& (x < 2)$ is true if $x \notin (2, 10)$.

Or operator (\vee)

The or operator is false only if the two propositions are false.

Table: Or operator

p	q	$p \vee q$
0	0	0
0	1	1
1	0	1
1	1	1

For instance, $(x > 10) \vee (x < 2)$ is false if $x \in (2, 10)$.

Numerical comparison propositions

The basic propositions arises from numerical comparison ($>$, \geq , $<$, \leq , $==$, $\sim=$). They are all used in their obvious way. The last two operators represent “equal” and “not equal” as they must be different from the affectation operator ($=$). For instance, if $a = 10$ and $b = 1$, then $a > b$ is true, $a \geq b$ is true, $a < b$ is false, $a \leq b$ is false, $a == b$ is false and $a \sim= b$ is true.

Composite propositions

With these, one can then construct composite propositions. For instance, $(!p) \parallel (q \& p)$. Parentheses applies as usual.

Table: Example of a composite operator

p	q	$(q \& p)$	$!p$	$(!p) \parallel (q \& p)$
0	0	0	1	1
0	1	0	1	1
1	0	0	0	0
1	1	1	0	1

Flow of instructions

- ▶ No more than one affectation by line.
- ▶ The file is read from top to bottom.
- ▶ To change the flow of instructions, we need flow control keywords : *if*, *while*, *for*, *switch*.
- ▶ I will present the operators with the matlab/octave convention.

If operator

Basic pseudo-code :

1. Beginning of code.
2. If some proposition is true
 - 2.1 execute this part of code
3. Otherwise
 - 3.1 Execute this other part of code.
4. Rest of the code.

Example

```
x = 10;  
if( x >7)  
    display('x is greater than seven');  
else  
    display('Facebook should invert some matrices');  
end  
  
x = x + 1;
```

The program will display “x is greater than seven”.

For operator

Basic pseudo-code :

1. Beginning of code.
2. For some variable in some range
 - 2.1 execute this part of code
3. Rest of the code.

Example

```
x = 0;  
for i = 1:1:10 %i starts at 1, increments by one, to ten.  
    x = x + i;  
end
```

x

The program will display "55".

While operator

Basic pseudo-code :

1. Beginning of code.
2. While some proposition is true
 - 2.1 execute this part of code
3. Rest of the code.

Example

```
x = 2;  
stop = 10^(-15);  
while( x >stop )  
    x = x / 2;  
end
```

x

The program will display “ 8.8818e-16”.

Switch operator

Basic pseudo-code :

1. Beginning of code.
2. Switch to some part, depending on the value of some variable.
 - 2.1 If the variable is equal to some first value : execute this part of code
 - 2.2 If the variable is equal to some second value : execute this part of code
 - 2.3 If the variable is equal to some third value : execute this part of code
 - 2.4 etc.
 - 2.5 If any case above has not been met : execute this part of code
3. Rest of the code.

Example

```
x = 10;
switch x

    case {1, 2, 3}
        display('We should play the game');
    case 10
        display('I am an happy person');
    otherwise
display('Facebook should invert some matrices');
end

x = x + 1;
```

The program will display "I am an happy person".

Functions

- ▶ Programming is prone to mistakes.
- ▶ Hence, if A, B, C are segments of code $(A + B)C$ is a better way to write code than $AC + BC$.
- ▶ Since C is written only once, there is only one place for possible errors.
- ▶ Moreover, some segments of code may be programs by themselves.
- ▶ Functions allows to write code that can be called by other segments of code.

Functions

Basic call to a function:

```
[out1, out2, ...]= func_name(arg1, arg2, ...)
```

There exists a lot of built-in functions.

Examples of built-in functions

```
x = eye(10, 10); %10x10 identity matrix
x = rand(10, 10); %10x10 uniform matrix
[n,m] = size(x); %Size of matrix x
lx = log(x) %Element by element logarithm
clear %Clear all variables
clc %Clears the screen
help [keyword] %help file
why %answer the the meaning of life
%(matlab only)
```

Creating your own functions

- ▶ You can create your own functions.
- ▶ They must be in a separate file if you want them accessible by many programs.
- ▶ They must be in the same directory than the file calling them or in a known directory to Matlab.
- ▶ The file must have the same name as the function name.

Function declaration

In the file “myFunction.m” :

```
function [out1, out2]= myFunction(arg1, arg2)
```

```
%some code
```

```
out1 = some_result;
```

```
out2 = some_other_result;
```

```
end
```

A simple example

Let the function be :

```
function Tx= toperator(x)

    a = (1, 1);
    b = 0.9;
    P = [0.3, 0.7; 0.2, 0.8];
    Tx = a + b * P * x;
end
```

A simple example

Then, one could do :

```
stop = 10(-15);
x_k = [0;0];
x_kp = [10; 10];
i = 1;
while( max( abs( x_k - x_kp ) ) > stop)
    if(i == 1)
        i = i + 1;
    else
        x_k = x_kp;
    end

    x_kp = toperator( x_k );
end
```

Programming tips

- ▶ Think before you type.
- ▶ Divide to reign.
- ▶ Clarity over speed.
- ▶ One line of code = one line of comments.
- ▶ Bug free code \nrightarrow it does what you want.

Working in group

- ▶ There are two things worse than reading your code : reading your code one month later and reading somebody else's code.
- ▶ Hence, working in groups is much better when you deal in abstract algorithms, specifications and tests.

Working in group

- ▶ Algorithms and pseudo-code allows to deal with general ideas to implement without looking at code.
- ▶ Functions allows to hide code. Once inputs, outputs and general working conditions are specified in group, one can code on his own.
- ▶ Hence, someone can program the main file while someone else programs some functions used by it.
- ▶ Testing allows to look for errors in functions.

An example

An implementation of the GNR for some non-linear regression model.

1. Load data.
2. Initialise β_1 and set $\beta_0 = 0$
3. While $\max(\nabla^{-2}\nabla > 10^{-8})$
 - 3.1 Compute the gradient ∇ .
 - 3.2 Compute the hessian ∇^2 .
 - 3.3 Set $\beta_i = \beta_i - \nabla^{-2}\nabla$ and $\beta_{i-1} = \beta_i$
4. Compute the SSR and report values.

Work can be roughly divided in four tasks : programming the main file, loading data, computing the gradient and computing the hessian.

An example (contd)

This leads to the following functions specifications.

```
[y, x1, x2, x3, x4] = load_data('path_to_file')  
g = gradient(b, y, x1, x2, x3, x4)  
h = hessian(b, y, x1, x2, x3, x4)
```

Working in group

People must also convey of the behavior of functions.

- ▶ How is the file with data organised ? What if loading the data fails ?
- ▶ What if the input of some function are empty vectors ?
Vectors of different size ?
- ▶ Is the hessian guaranteed to be invertible or should it be tested before attempting inversion ?

Broadly :

- ▶ What are the input cases and the output given these cases ?
- ▶ Under what circumstances will the function deliver ?

Working in group

An example of a more detailed function spec :

```
% Function Lorentz : draw a lorentz curve.
% Input : An nx2 matrix.
%           In the first column, the total revenue by category.
%           In the second column, the number of persons by category.
% The matrix must look like
%
%                                     | 10$      15 pers |
%                                     | 12$      12 pers |
%                                     | etc.      etc      |
% Note :           1- The matrix does not have to be sorted by $/pers.
%           2- Some numbers may be zero.
%
% Output :   On screen : Lorentz curve
% exit : An nx2 matrix with the cumulative distribution
%           First column : the share of revenues
%           Second column : the share of people
% Algorithm :
% 1- Delete all categories with zero persons in it.
% 2- Sort categories by $/pop.
% 3- Compute revenues and population in percent.
% 4- Compute cumulative distributions
% 5- Draw curves
function exit = lorentz(matrix)
```