How I learned to stop worrying and love good coding (and version control)

In this hands-on you will take a very simple and ugly code and transform it into a piece of readable, commented and modular code. Also, you will use git to keep track of all the modification you did.

So let's first create a repo on github. Once this is done, create a local repo

```
mkdir jacobi
cd jacobi
git init
```

add the remote repository to the repo list

```
git remote add origin https://github.com/gbrandino/jacobi.git
```

Add then a readme.md file. Then

```
git add readme
git commit -m 'initial commit'
git push -u origin master
```

For the entire exercise you can either C or python, that are in the corresponding folders

The Laplace problem using Jacobi iterator

The algorithm used in this program solves Laplace’s equation on an evenly spaced grid through the use of a simple Jacobi iteration technique.

The equation has the form: $\frac{\partial^2 \varphi}{\partial x^2} + \frac{\partial^2 \varphi}{\partial y^2} = 0$

A strategy to solve such an equation, given the boundary conditions, is the use of a Jacobi iteration that employs numerical second derivatives.

To tackle this, we would set up a two dimensional grid to represent the surface, and we will divide it evenly into square regions. Regarding boundary conditions, we will be setting the bottom left corner to 100.0 and with an increasing gradient toward the other corners until it is zero. Once these conditions are set, the algorithm will use numerical solutions to the second derivatives in each direction to update the current matrix elements.

The Algorithm

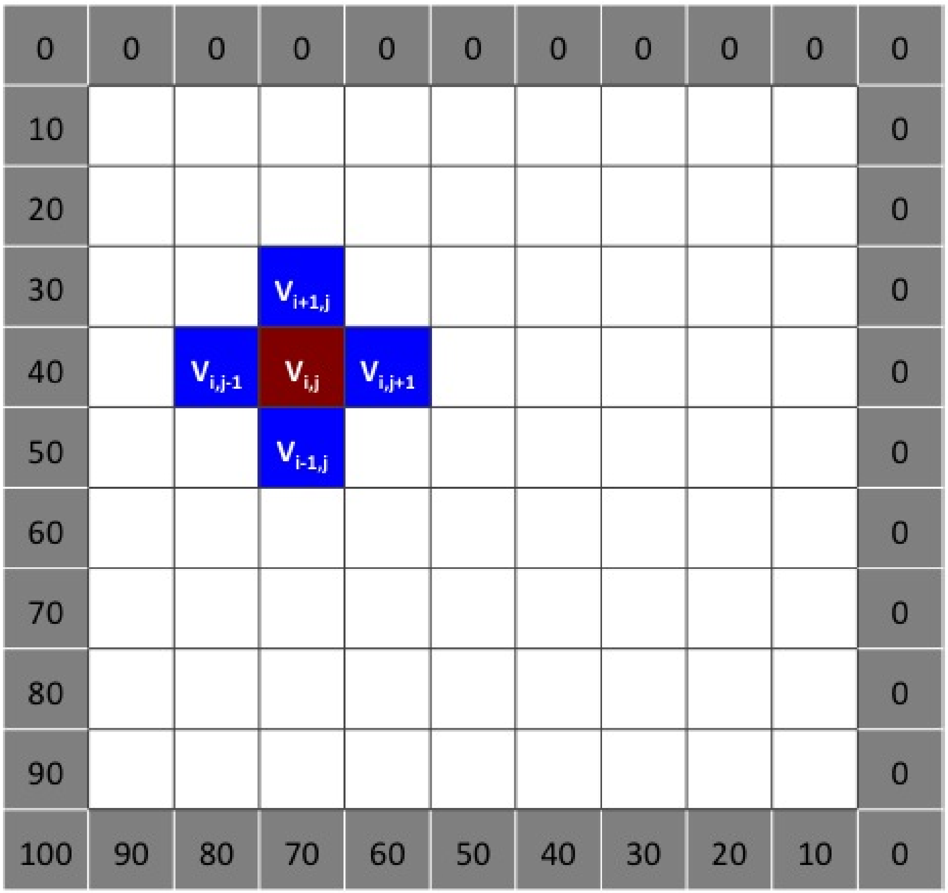
Here you can find a short description of the algorithm, of which Figure 1 shows a cartoon.

1. Allocate and specify a 2D array defining an evenly spaced grid of square dimension, leaving a space for the boundaries, as they do not belong to the main grid (i.e. a 1024 x 1024 matrix would need to be allocated as 1026x1026 to leave room for the borders.
2. Setup the initial constant boundary conditions. The value at the lower left hand corner of the of the grid will be fixed at 100.00, and the value ascending and to the right will be set to a linear gradient reaching zero at the opposite corners (see Figure 1). The rest of the borders will be fixed at zero. Please note, these boundaries will remain constant throughout the simulation.

3. Setup the initial condition of the inner grid elements as 0.5.
4. Begin and continue for a fixed number of cycles the iterative process. At each iteration, the value of each inner matrix element needs to be recomputed from elements of the current iteration. The updating formula, based on numerical computation of second derivatives, is:

$$V_{i,j}^{new} = 0.25(V_{i+1,j} + V_{i-1,j} + V_{i,j+1} + V_{i,j-1})$$

5. After updating, copy the new matrix into the old's memory and continue iterations until completion.



0	0	0	0	0	0	0	0	0	0	0
10										0
20										0
30			$V_{i+1,j}$							0
40		$V_{i,j-1}$	$V_{i,j}$	$V_{i,j+1}$						0
50			$V_{i-1,j}$							0
60										0
70										0
80										0
90										0
100	90	80	70	60	50	40	30	20	10	0

Figure 1: A diagram of the Jacobi Relaxation for Solving the Laplace's Equation on an evenly spaced 9x9 grid with the boundary conditions outlined in the text above.

Compile the code

```
gcc -O2 jacobi.c -o jacobi
```

and run it

```
jacobi 10 5
```

To plot the results, (you will need matplotlib python package)

```
python plot.py
```

1 - Basic coding style

First, copy the jacobi_initial.c/jacobi_initial.py file to your repo and put under version control. (git add & git commit).

Now have a look at it. The goal of the first part is to clean it up, enforcing

- space between groups of lines
- space between elements in a line
- indentation,2 or 4 spaces, your choice. Remember to set your editor to expand tabs to spaces (well, in python this is part of the language...)
- 80 columns length
- a brief explanatory comment at the start of every program/file
- comment inside the code to explain non trivial parts

The exercise is rather tedious so, for your convenience you can find the clean up version in the folder solution/1. Nevertheless, we encourage you to do a little clean up, just to get the hang of it.

Once you are done (or you have copied the solution...) remember to commit and push your changes.

2 - Variable naming and modularity

The second step is to improve further readability and enforce modularity. About the former, the task is to substitute all the variable names with meaningful names, such that it is clear what they represent. For composite names (such as the one holding the dimension in byte) you can choose to use underscore (dimension_byte) or camel-case (dimensionByte).

Then, make the code modular, by creating three functions from, respectively

- the part that sets the boundary conditions
- the part that updates the solution
- the part that prints the output

Once again, the solution is given in the folder solution/2

Once you are done, compare the changes with the commited code using

```
git diff <filename>
```

3 - Separate in different files

Finally separate the code in different files, having the three functions created before in a different file. In C, this requires the creation of a header and a source file. In python you will be creating a module. Since we are making a big change to the code, it may be a good idea to create a new branch and work there

```
git checkout -b restructuring
```

The solution is given in the folder solution/3. For the C code, the solution folder contains also a minimal makefile for your convenience.

Once you are done, you may decide to merge the newly created branch into the master. First switch to master

```
git checkout master
```

then merge the restructuring branch

```
git merge restructuring
```