Numerical Project

Cyril Tsilefski

October 30, 2020

Contents

1 Functional requirement of the program **1**
1.1 The project . . . 1
1.2 Files . . . 1
1.3 Data . . . 1
1.4 Outputs . . . 2
1.5 Concerning the running time . . . 2

2 Internal structure of the program **2**
2.1 Physical model . . . 2
2.2 Scientific computation algorithms . . . 2
2.3 Constitutive elements . . . 2
2.3.1 `main.py` . . . 2
2.3.2 `data_check.py` . . . 3
2.3.3 `matrices.py` . . . 3

1 Functional requirement of the program

1.1 The project

The goal of this project is to simulate the movement of a fluid through different geometries. The program creates a box of a choosen size, builds a geometry inside it and simulates the movement of a given fluid.

1.2 Files

In order to increase readability, the project is made of several files. I made the choice to work with Object Oriented Programming.

- main.py: This file calls for the needed functions/class
- matrices.py: This file contains the class "Matrices", it builds the geometry, the different matrices to plot and stores them
- plot.py: This file plots the matrices built in "matrices.py"
- parameters.py: This file contains all the variables that can be changed by the user
- data_check.py: This file checks the variables and makes sure that the program will run

1.3 Data

This project uses several piece of data set by the user to work.

- N_x and N_y are the size of the domain
- h represents the size of a cell
- $geometry$ corresponds to the choosen geometry
- $angle$ corresponds to the angle of the widening/shrinkage geometry

- v_x is the Neuman condition
- ϕ_{ref} is the Dirichlet condition

Be careful in the case of a widening/shrinkage geometry! In order for the program to generate a domain from one end to another, there is a restriction on the angle, if the restriction is not met, the program will output a ValueError. The restriction is as follows:

$$|angle| < \arctan\left(\frac{0.5 \times N_y - 1}{N_x}\right)$$

The angle parameter should be set in degree, the program will convert it to radians for the computation.

1.4 Outputs

As of the alpha version, the program outputs 4 pdf files, one for each plot. The files are saved in a subfolder named **figures/** and the filenames are set with the following rule:

`<data>_<geometry>_Nx=<Nx>_Ny=<Ny>.pdf`

`data` stands for the plotted data (potential, velocity, streamlines, pressure).

1.5 Concerning the running time

Due to the function `numpy.linalg.solve()` being slow for big matrices, the bigger the size of the domain, the higher the running time.
For a domain size of 3600 cells (60×60), it takes around 20 seconds to run, for a domain size of 14 400 cells (120×120), it increases to 23 minuts.

I searched for a faster method to solve the linear system in vain, thus I recommend to stay on relatively low values for N_x and N_y, the graphs are easily readable for a value of 60 each.

2 Internal structure of the program

2.1 Physical model

In order to build the model, the program uses a squared structured lattice model (matrix). The values are computed at each point of the matrix.

2.2 Scientific computation algorithms

2.3 Constitutive elements

2.3.1 `main.py`

This file is executable (without parameters input in the command).
After the imports, it will first call the file `check_data.py` to check the type of the input data.
Then it will call the file `matrices.py` to initiate the values of `G`, `M`, `cell_coords`, `A` and `b`.

It then checks for the `recompute` value, if **`True`** or if there are no existing **dat** files for the current parameters, it will call the function `domain_check()` to display a warning if the domain is large. It then calls the subroutine `make_data()` to compute `phi`, `grad_y`, `grad_norm`, `pressure` and `pressure` and saves them in **dat** files.
Else, it will use the existing dat files to generate the plots.

Then it calls the function `load_data()` to read the **dat** files and stores them in a dictionary and it initiate the file `plot.py` which will be used to plot the different values.

The last lines call the subroutine `plot_graphs` with different arguments to plot and save all the wanted graphs.

2.3.2 data_check.py

This file has several subroutines and functions to rule the execution of the program.

The subroutine `data_check()` reads the value of each parameter and checks that the type and the value are correct and will not cause a crash of the program. If the value/type is incorrect, it will raise an error and stop the program.

The function `existing_data()` will check for specific files in the **dat/** subfolder. If at least one files is missing, it returns `True` and the program will recompute all the data. Else it does nothing.

The subroutine `domain_check()` is just a warning. It read the max value of the matrix `M` plus one which is the number of fluid cells to compute. If this number is higher than 5000, it will display a message warning the user that the program can take some time to run. Then it asks if the user wants to keep going. `"Yes"` will continue, `"No"` will stop the program and anything other than that will stop the program aswell.

2.3.3 matrices.py

This class is where everything is computed, from start to end.

`__init__` stores the value of `G`, `M`, `cell_coords`, `b` and `A` which will be used several times in the class.

The subroutine `make_data{}` will call the functions that compute `phi`, `grax_x`, `grad_norm`, `pressure` and `pressure`. Each function will write the data in a **dat** file.

The function `load_data()` reads the **dat** files and return the values in a dictionary.

The function `build_g()` takes 4 input arguments.

- `Nx: int`
- `Ny: int`
- `geometry: str`
- `angle: int` of size 8 bits, in radians

It acts as a selector. Given a geometry, it will create a matrix `G` filled with zeros and call the function `build_geometry()` with different input values. There is no real point in this function in the alpha version since the straight, widening and shrinkage geometries are generated from the same function, with only the angle changing. However, it will prove useful in the beta and gold version.

The function `build_geometry()` takes 2 input arguments.

- `G: np.array`
- `angle: int` of size 8 bits, in radians

The function returns the matrix `G`.
The function starts by computing `alpha` as the tangent of the angle.
It then uses it in a `for` loop to compute an offset which will be used to build the geometry.
In case of a shrinkage geometry, the angle inputed is negative, the function will flip horizontally the matrix. It then sets the inlet and outlet of the domain.
Since the program will always compute `G` no matter what and to avoid some problem, it will save the matrix only if `recompte == True`.

The function `build_index_matrices()` takes 2 input arguments.

- `Nx: int`
- `Ny: int`

The function returns the matrix `M` and the array `cell_coords`.
It starts by creating the matrix `M` the shape of `G`, filled with zeros and initiate a counter `count`. It then parses `M` and for each fluid cell (`G[c, r] != 0`), it sets the value of `M[r, c]` to the value of the counter and increment the counter by 1. All the other values are set to -1.
Then it create an array `cell_coords` of size `(M.max() + 1, 2)` and sets the value of the counter back to zero.

Then it parses the matrix `M` and for each cell which value is not -1, it will store the coordinates of that cell in the index `count` of the array `cell_coords` and increment the counter.

The function `build_a()` takes no input, it uses the imported data from `parameters.py`.
The function returns the matrix `A`.
It starts by creating an array `A` of size `(cell_coords.shape[0], cell_coords.shape[0])`, filled with zeros. It then parses the array `cell_coords` and for each inlet and fluid cell (the oulet is not counted there), it counts the neighbors, stores their coordinates, set the value of the diagonal depending on the neighbors and sets each neighbour associated cell to 1. The diagonal cells corresponding to the outlet are all set to 1.

The function `build_b()` takes no input, it uses the imported data from `parameters.py`.
It returns the array `b`.
It starts by creating an array `b` of size `(cell_coords.shape[0], 1)`, filled with zeros.
It then parses the array. For each cell corresponding to the inlet, it sets the value to `-vx * h` and for each value corresponding to the outlet, it sets the value to `phi_ref`. The rest of the array is not modified.

The function `build_phi()` takes no input, it uses the imported data from `parameters.py`.
This function is the slowest one since it compute the solution of $Ax = b$. For that it uses the function `linalg.solve()` from `numpy`. Then it creates an empty matrix `phi` and fills it with `numpy.nan`. It then parses `x` and sets the value of each phi cell corresponding to the `x` value. The other cells are not modified.
Finally, it saves the values in a **dat** file.

The function `build_gradient()` takes no input, it uses the imported data from `parameters.py`.
This function is rather big, but it is only a variation of the `gradient` function from `numpy`. The function `numpy.gradient` does not work the way I want with the `numpy.nan` values so I made my own.
It starts by loading the **dat** file containing the values of `phi` and creates 2 empty matrices `grad_y` and `grad_norm` the same shape as `G` and fills them with `numpy.nan`. It the parse the `cell_coords` array and sets the values of `grad_y` and `grad_norm` following the centered difference with step $2h$ mentioned in the scope statement. I then found that the function `numpy.gradient` uses the same method. However, since the borders of the matrix `phi` is filled with `numpy.nan` values, the gradient is not computed at the outer edge of the domain.
To compute there values, I applied the impermeable wall condition specified in the section **5.1** of the scope statement. We consider the `numpy.nan` cell with coordinates `[i - 1, j]` as a fictious cell with the value of the cell `[i, j]`. This rule applies to the cells to the left and to the right for `grad_y` and to the top and to the right for `grad_norm`.
It the computes the norm of the gradient (used later to plot the normalized gradient).
Finally, it saves the 3 matrices in **dat** files.

The function `build_pressure()` takes no input, it uses the imported data from `parameters.py`.