Advanced D. L. for Physics

Exercise 3

Exercise 3 – *Pressure update*

This exercise is to be implemented into the *mantaflow* framework which was provided in the previous exercises. All code for this exercise has to be implemented in the file ex3.cpp available on the moodle. You can either copy the relevant function declarations into source/test.cpp of your existing code, or add the ex3.cpp as a new file to the cmake projects (you can check how its done for test.cpp). Both solutions are fine, we will only refer to ex3.cpp in the following. We will test your version of the code by running your ex3.cpp, therefore do not include external libraries or change code elsewhere, as the code is not guaranteed to compile on our machines.

Overview figure:

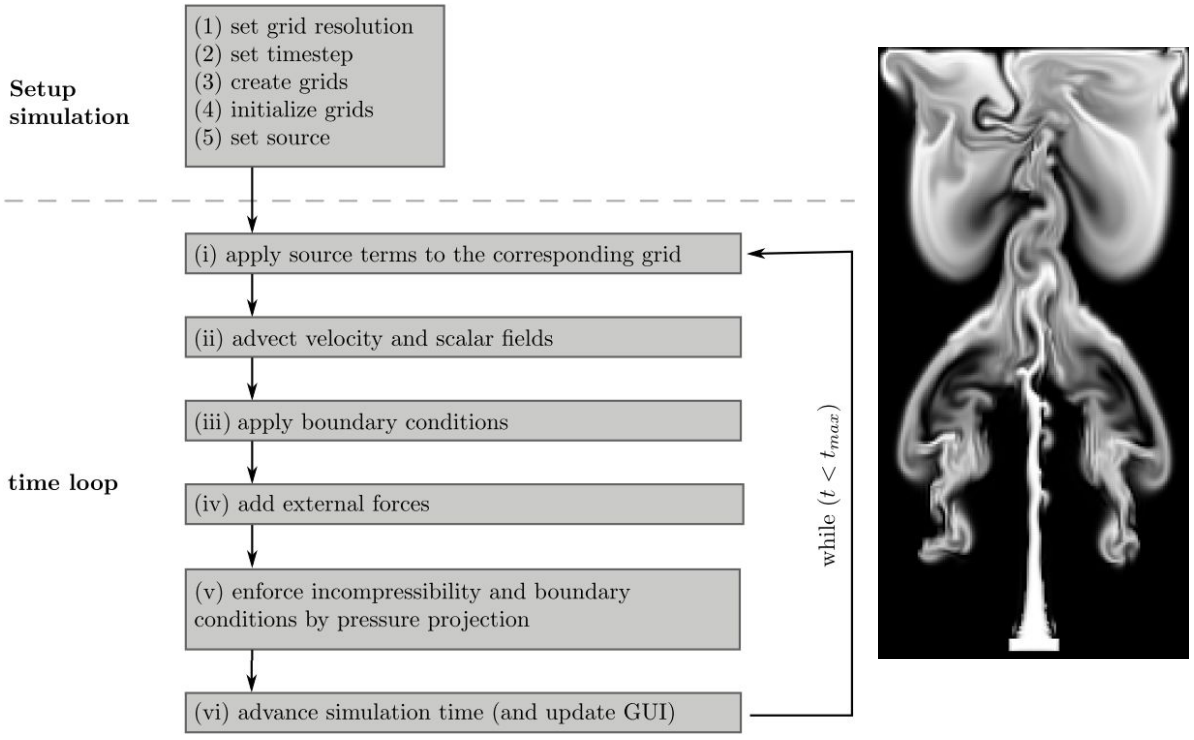

*Figure SEQ Figure * ARABIC 1: left: Flow diagram of the smoke plume simulation; right: Resulting flow of the exercise's implementation*

Hand in your exercises by sending your files to adl.phys.ex@gmail.com by **May, 31st, before midnight** (email arrival time will count)**.**

Name the zip file "**xx_lastname1_lastname2_lastname3.zip**" (xx being the group number) with **no subfolders,** please make sure to use **meaningful** filenames. The subject of the email should be [ADL Exercise03 GroupXX].

Problem description

In this exercise, you will simulate a two-dimensional, buoyancy driven smoke plume with an Eulerian fluid solver. The buoyancy is a property of the smoke scalar and is applied by the external forces step (iv). The resulting flow of this exercise is shown on the right in Figure 1.

After introducing the basics of flow simulation in the lecture, the objective of this exercise is to implement a simple version of the pressure projection step. As a reminder, the pressure projection ensures the flow's velocity field remains divergence free (=incompressible). Recall the steps involved in simulating fluids with Figure 1. All listed steps can be found by their numbering in the provided Python scene file (ex3_gsPressureSolve.py).

Setup *mantaflow* for the exercise

Two files are provided for this exercise. One is a Python case file (ex3_gsPressureSolve.py in scenes), which implements the schematic displayed in Fig. 1. The second one is an empty C++ implementation file (ex3.cpp in source) where all of this exercise's code will go.

The only method concerning the actual pressure solve which will be called from Python is solvePressureGS(). The ex3.cpp file provides additional empty optional methods. They divide the computation into the three steps listed below in "Main tasks". It is, however, possible to implement everything into a single method (note: at least one KERNEL() and one PYTHON() plugin method have to be used as KERNELs are the most efficient way to iterate over grids and only methods with the PYTHON() keyword are visible to our python scene files).

Main tasks

1. Compute the divergence

First, the divergence of the current velocity field needs to be computed. Use central differences to compute the divergence field of the velocity and store it in a scalar grid. The velocity is stored on a MAC grid and scalar quantities such as the divergence are saved on a collocated grid. The MAC (colored) and collocated (black) indices can be seen in Figure 2 on the left including the discrete divergence. The corresponding divergence equation can be found in the lecture slides.

mantaflow normalizes the size of all grid cells to be 1.0 in all directions. The time step is set to 1.0 (check the Python file). The density ρ can also be assumed to be 1.0.

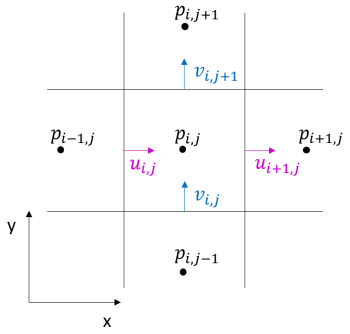

*Figure SEQ Figure * ARABIC 2: mantaflow indices for staggered (blue and purple velocity components) and collocated (black pressure value) quantities.*

2. Solve the Poisson equation

Use the Gauss Seidel algorithm presented in the lecture starting on slide 29 05_FluidSimulationNumericalSolve.pdf to solve the pressure equation. Update the pressure grid in place (use old and new values at the same time which is characteristic for the Gauss-Seidel algorithm). Implement two versions of this procedure:

1. Implement a version which calculates the pressure update and a residual in two separate loops.
2. Implement a second version that uses the residual computed for a cell for the cell's pressure update.

To compute an average residual of the entire domain, use the L2 norm. The summation of residual values for the L2 norm can be done either manually or with the KERNEL(reduce=+) macro. Here is an example for the KERNEL(reduce=+) macro:

```
KERNEL(idx, reduce=+) returns(double result=0.0)
double knGridTotalSum(const Grid<Real>& a, FlagGrid* flags) {
        if(flags) {         if(flags->isFluid(idx)) result += a[idx]; }
        else    { result += a[idx]; }
}
```

Print the residual for both (2.1 and 2.2) versions during each iteration to the console. If the algorithm is implemented correctly, the residual will decrease fast in the first iteration steps and slow during late iterations, see Figure 3.

Set a maximum iteration count to stop iterating if the target accuracy (gsAccuracy attribute of the solvePressureGS() method) is not reached within this maximum number of iterations.

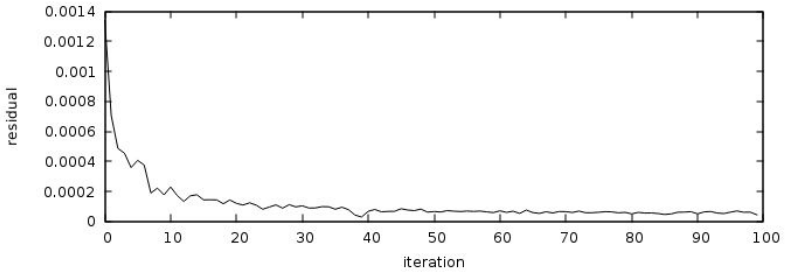

*Figure SEQ Figure * ARABIC 3: Example residual of a Gauss Seidel Poisson solver*

3. Update the velocity field

Add the negative gradient of the computed pressure update field to the velocity field in the cells which contain fluid (use the FlagGrid).

4. Test your pressure solve

If your pressure solve and the velocity update were successful, the divergence of the update velocity field should converge to zero (scaling with the accuracy gsAccuracy used for the pressure solve).

Implementation, hints & ideas

Show your solutions (or as much as possible) in the mantaflow UI. That's what it's for. You can create (and pass) as many helper grids as you like to show your solutions, intermediate variables and so on.

General:

If you are wondering what the constant factor c should be: it is a scalar value since we consider some equations for one cell only, not for the whole grid.

Boundary conditions:

For simplicity, all boundaries are frictionless walls in this exercise. The FlagGrid provides information on which cells are fluid and which aren't (this is already implemented, nothing do be done here). Therefore the coefficients matrix A0 (off-diagonals: either -1 or 0 = fluid or no fluid) and matrix A1 (diagonal: absolute sum of the four surrounding A0 values) should be computed before solving the pressure equation (remember cell size, ρ and time step are 1.0).

Additional items:

1. Remember that the KERNEL() methods automatically implement loops. This means you do not need to explicitly loop over the grid. You can access a grid cell over the indices i, j and k. The following code when implemented in a KERNEL() method will be automatically executed for the entire grid:

   ```
   vel(i,j,k).x = 1.0; //set the velocity in x-direction to 1.0
   ```

2. Additional grids need a solver object associated with them and therefore need to be created like this:

   ```
   FluidSolver *parent = flags.getParent();//do this only once
   Grid<Real> mygrid(parent);          //automatically initialized to 0
   ```

Demonstration tasks

The following items of your implementation will be checked:

- Correct computation of the divergence
- Correct implementation of both Gauss Seidel iterative procedures
- Correct computation of the velocity update
- Stable simulation of the smoke plume, residual is converging
- Compute the **final 2D results with a resolution of 128**.

Files to hand in

- Screenshots of the smoke density and the pressure field at time steps **250**, **500** and **750**

  Make sure they look like this:

- Your version of the ex3.cpp implementation

Bonus task (does not influence the grade bonus)

- Extend your code to three dimensions.