NS-EOF – Overview

November 28, 2023

1 Purpose & Prerequisites

NS-EOF can be used to simulate two- and three-dimensional incompressible Navier-Stokes problems on Cartesian grids. The solver follows the descriptions from Griebel and thus uses explicit Euler time-stepping for the evolution of the flow velocities and an implicit formulation for the pressure. The latter results in a Poisson equation that needs to be solved in each time step. The Poisson equation is solved using the PETSc library. Since the implementation makes use of MPI, you should further install OpenMPI. On Ubuntu systems, you may just use the library and headers from the Ubuntu repositories.

2 The Configuration File

The configuration is provided via an XML file. The outer node is called `configuration`. As an example, the file looks somewhat like this:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <category1 attribute1="value1" atribute2="value2" />
    <category2 attribute3="value3"> Some text </node2>
    :
    :
</configuration>
```

The hierarchy looks like this:

configuration

flow

- `Re`: Reynolds number.

simulation : Contains text defining the type of simulation to run

- `finalTime`: Goal final time for the simulation.

timestep

- `dt`: Default time step
- `tau`: Time step security factor

`dt` is only used if `tau` is set to a negative number. Otherwise, the time step will be computed to keep the simulation stable.

solver

- `gamma`: Weighting factor between finite differences and donor cell scheme.

geometry

- `dx/dy/dz`: Dimensions of a cell.
- `lengthX/lengthY/lengthZ`: Dimensions of the domain.
- `sizeX/sizeY/sizeZ`: Number of cells in each direction.

Either the length of the cell or the length of the domain has to be set, not both at once.

environment

- `gx/gy/gz`: gravity components.

walls

- `periodicX/periodicY/periodicZ`: (boolean) Periodic boundaries in each direction.

left/right/bottom/top/front/back

vector

- `x/y/z`: Components of wall velocity.

vtk : Has text with the prefix for the output files

- `interval`: number of steps between writing of VTK files.

stdOut

- `interval`: interval for printing of output to the screen.

parallel

- `numProcessorsX/numProcessorsY/numProcessorsZ`: Number of processors in each direction.

The meanings of the parameters `tau` for the time step and `gamma` for the solver are better explained in section 3 of Griebel.

3 Code Structure

In the following, we step through the typical ingredients of a fluid simulation. We do this "hierarchically": we first consider the overall simulation flow and then dive into the different technical ingredients that are required.

3.1 The Main

The starting point of every program is the Main.cpp. In the current software, the main reads the configuration file, stores the global flow parameters (such as Reynolds number, boundary conditions etc.) in a `Parameters` object, initialises the correct Simulation object and triggers the time stepping. You can thus find the respective time loop in the Main.cpp.

3.2 Algorithmic Phases

A simulation time step consists of different algorithmic steps. In our case, for example, we need to

1. determine the current time step size
2. assemble the right hand side of the Poisson equation,
3. solve the Poisson equation,
4. update the velocity values for the next time step and
5. set suitable boundary conditions.

The sequence of these steps is implemented and triggered from the class `Simulation`, cf. the method `solveTimestep()` in Simulation.hpp. The latter method is virtual; you can thus inherit from the `Simulation` class and define your own time stepping scheme.

3.3 Iterators and Stencils

3.3.1 Iterators

In most of the algorithmic steps, we need to traverse all cells of the grid (or subsets of the grid) and carry out an operation on each cell. We typically do not have to deal with all kinds of different cell traversals (such as randomised cell access or particular ordering of the cells) and respective grid decompositions, but only need to perform operations on

- all grid cells
- all inner grid cells or
- all cells close or at a global or parallel boundary.

We hence use a simple lexicographic ordering of the cells. This concept of traversing the cells is encapsulated in the class `Iterator` and inherited classes, cf. Iterators.hpp. Given a certain type of cellwise operation in form of a `Stencil` object (we come to this class in a second) and a compatible data set `DataField` associated to these operations, an iterator loops over all cells and applies the cellwise operation of the stencil. Considering the implementation of the iterators, you may observe that the type of the data field, i.e. the kind of information that is stored in each grid cell, is handed over as *template parameter*. Thus, it does not matter which kind of data field we plug into the iterator. We only request that the data field and the stencil are compatible.

3.3.2 Stencils

A cellwise operation is encapsulated in form of a *stencil*, see also the header Stencil.h for different stencil classes. Let's consider the `FieldStencil` in more detail. This stencil class can be used to execute cellwise operations on all cells of our data structure. Looking into the definition of the `FieldStencil`, you can observe that

- the stencil is also a template and depends on the data field type `FlowField` via the template's argument.
- the stencil has abstract functions `void apply(FlowField&, int i, int j)` and `void apply(FlowField&, int i, int j, int k)`. Implementing one of these interface definitions (for 2D or 3D, you may choose one or the other method) is thus sufficient to define a cellwise operation.

If we want do define a cellwise operation on a particular data field, we can thus inherit from the respective stencil (e.g. the FieldStencil <MyDataField> where MyDataField is a respective data structure built from the available data structures in DataStructures.h) and implement the cellwise operation using the apply(...) methods. Since we have access to a respective data structure and the indices i,j,k of the current cell under consideration in the apply(...) method as well as access to a `Parameters` object with the global simulation data (such as time step size, Reynolds number etc.) , we have everything that we need for our purposes.

3.3.3 Connecting Iterators and Stencils

We discussed how to iterate over the Cartesian grid and how to implement a cellwise operation. Let's bring the two concepts together. For this purpose, we consider the class `RHSStencil` that can be found in the directory `stencils`. The latter directory contains all stencil implementations that work on the data structure `FlowField` which is a combination of all data fields (such as velocity, pressure and flagfield) required for the standard Navier-Stokes simulation program from Griebel. The `RHSStencil` is used after evaluating the functions F,G and H and assembles the right hand side of the Poisson equation. An object of this stencil is thus defined and initialised—together with an iterator—in the `Simulation` class as follows:

```
private:
  RHSStencil rhsStencil_;
  FieldIterator<FlowField> rhsIterator_;
  :
  :
  :
public:
  Simulation(Parameters &parameters, FlowField &flowField):
    parameters_(parameters),
    flowField_(flowField),
    :
    rhsStencil_(parameters),
    rhsIterator_(flowField_, rhsStencil_),
    :

  :
  :
  virtual void solveTimestep(){
    :
    rhsIterator_.iterate();
    :
  }
```

We thus first define an object of our stencil. Then, we further define an object for a field iterator which should work on our data structure `FlowField`. In the constructor of the simulation class, we now instantiate and initialise both objects: we first initialise the stencil and subsequently, we initialise the iterator which depends on the respective stencil. When we want to assemble the right hand side of the Poisson equation during one time step (see `solveTimestep()`-part), we only trigger the iterator. This results in the following sequence of events:

- We call iterate() on our iterator `rhsIterator_`
- The iterator loops over the respective domain. Since the `rhsIterator_` is a `FieldIterator`, we will loop over the whole computational grid.
- On each grid cell indexed by i,j,k, the iterator triggers a call to its stencil's apply(...) method. In the current example, `rhsStencil_.apply(i,j,k)` is called for each grid cell.
- After traversing the grid, the iterator returns and the function call ends.

A word on the compatibility of stencils and iterators: obviously, we can only implement cellwise operations on data structures that are available during the grid traversal. Hence, the data structure (e.g. the `FlowField` in the current example) must match for both the iterator and the stencil. Looking into the definition of the `RHSStencil`, we see that this stencil inherits from `FieldStencil<FlowField>`. Thus, everything is fine: the stencil wants to apply an operation on the `FlowField` *and* the iterator traverses the correct data structure `FlowField` as well. Incompatibilities should thus be avoided at this point.

3.4 Boundary Treatment

In order to supply boundary conditions, different iterators than the pure `Field-Iterator` are required. We further distinguish two types of boundaries: *global* and *parallel boundaries*.

Global boundaries are located at the very outer sides of our simulation domain. The global simulation domain has a box-like shape, resulting in 4 (2D)/ 6 (3D) global boundaries. The *GlobalBoundaryIterator* can be used to traverse these regions and set appropriate (scenario-dependent) boundary conditions.

Besides, *parallel boundaries* may occur in distributed simulations. Assume we can work with 8 processes to handle a 3D simulation. Dividing the global simulation domain into $2 \times 2 \times 2$ sub-domains, we observe that each sub-domain has three contributions to global boundaries *and* three local boundaries which separate this sub-domain and a neighbouring sub-domain on another process. The latter boundaries are referred to as *parallel boundaries*. A `ParallelBoundaryIterator` is available, cf. Iterators.hpp.

4 Parallelisation

The parallelisation works by domain decomposition. The global rectangular domain is split into several subdomains by planes parallel to one of its faces. This also means that the subdomains will be rectangles themselves. Each of these subdomains is assigned to one of the processors. The assignment follows lexicographically, according to the position of the subdomain.

In order to share information, each processor requires additional information, such as a definition of its subdomain and the ranks of its neighbors for communication. The `Configuration` class should take care of initializing these values so that the program runs correctly with a single processor, but to work in parallel, the `PetscParallelConfiguration` class should be used, which will properly set the `Parameters` object.

The communication is performed by the `ParallelManager` class. The class has methods to communicate velocity and pressure between processors. When called, they will load output buffers, transmit information using MPI routines, read input buffers and set the data locally. Once the ghost cells are properly set, the simulation can continue using the other iterators.

5 Solving the Poisson Equation Using PETSc

In the algorithm, we need to solve a Poisson equation in each time step. The setup and solving of the respective linear system is accomplished using PETSc. The respective solver can be found in `Solvers/PetscSolver.hpp`. This solver can be considered as *black box*, we are thus neither further interested in the solving methodology at this point nor in its parallel implementation.

However, note that you may also use a simple PETSc-independent SOR-solver to solve the Poisson problem in sequential mode (see `Solvers/SOR-`

`Solver.hpp`). In this case you may, however, adapt some lines in your `Simulation` class to initialise the new solver. But be prepared for slower simulations.