IPVS – Universität Stuttgart WS 17/18
Abteilung Simulation großer Systeme
Prof. Miriam Mehl
Malte Brunn, Amin Totounferoush, Dr.-Ing. Nehzat Emamy

Numerical Simulation
Exercise sheet 1

Serial implementation of the flow solver
Hand in via email to amin.totounferoush@ipvs.uni-stuttgart.de until 14.11.2018, 18:00
Contact for quesions about framework malte.brunn@ipvs.uni-stuttgart.de

Introduction to flow solver

1.1 Problem definition

In the first exercise sheet, a program will be developed for the simulation and visualization of unsteady viscous laminar flows. We consider only incompressible media such as water. By such media, the density, ρ, does not change over the time. Air would be an example of a compressible medium, because the same mass of air can fill out different volumes.
The viscosity of a medium is defined by the dynamic viscosity, η. For example at 29°C, water has $\eta = 90.891\frac{Ns}{m^2}$, olive oil $\eta \sim 10^2\frac{Ns}{m^2}$ and honey $\eta \sim 10^4\frac{Ns}{m^2}$. This means that a medium with higher dynamic viscosity is more viscose.
Laminar means that the fluid does not contain any turbulence. This is not true in case of high velocities or low viscosities, which result in high Reynolds number, $Re = \frac{\rho \cdot u \cdot l}{\eta}$. An unsteady fluid can change over the time. Figure 1 shows an example of the driven cavity flow.

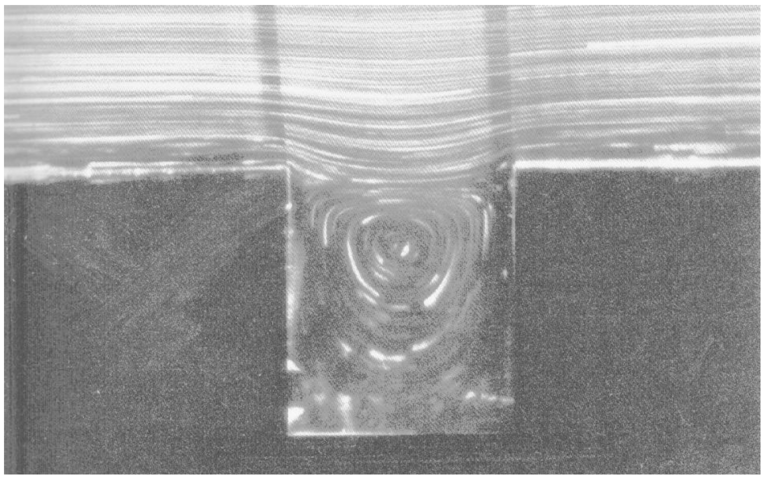


Figure 1: Snapshot from a driven cavity flow.

1.2 Plan for implementation

The whole exercise will be implemented in `C++`. You are **not allowed** to use high-level libraries. The program **must** be kept independent from the platform because on the test systems (*VISSIM-Pool* or *SGS-Pool*), only the programs under Linux could be compiled and started.
In order to change between the data types *float* and *double* define a *real* type using *typedef*.
All parts of the program must be sufficiently commented. The comments must be in appropriate forms for doxygen[1] [2].
In consideration of the next exercise sheets and easier handling of errors, you must keep on with the here suggested design for implementation. After the first structure analysis of the algorithm, the following UML class diagram (Figure 2) is sketched. In the next sections, each

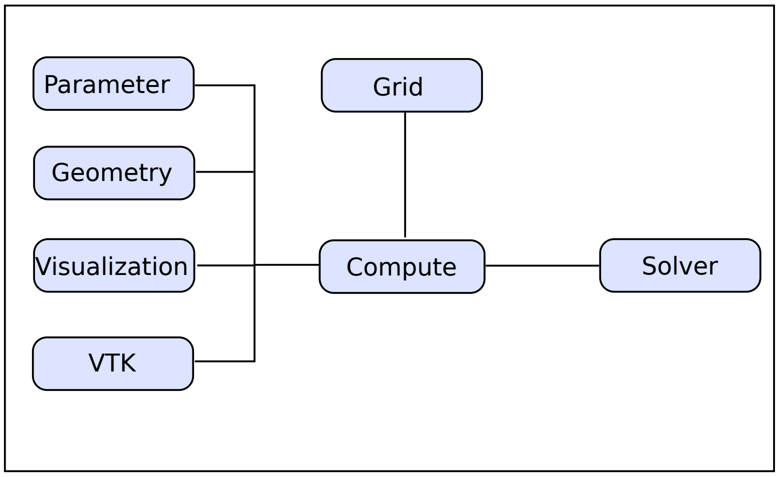


Figure 2: First UML class diagram after analysis.

component is shortly described and in the whole sketch integrated.
To start, you will receive a framework that includes the header files of all classes and modules. A more accurate documentation and the provided functionality of each component can be found from the header files of the framework.
Furthermore, the framework provides a live-visualization of the simulation results, which will be provided by the implementation. One can find a manual for the live-visualization in section 1.2 below.
In the framework file `typedef.hpp`, you will find some **typedefs**, which you should use in your implementation.

Grid

The class `Grid` integrates the functionality for discretization of the base grid. Access to data for reading and writing the values on the grid points belong to this functionality, as well as providing the operators for computing spatial derivatives of the first and second orders. To discretize the derivatives, a compact stencil will be used that uses the information only from the neighbor cells. The internal representation of the grid is a one-dimensional field and the data access is only via appropriate Iterators possible. The discrete quantities u, v, p and other discrete fields

[1]`http://www.stack.nl/~dimitri/doxygen/`
[2]`http://www.stack.nl/~dimitri/doxygen/docblocks.html`

would be created as instances of the Grid class in Compute (as members). Implement the functions defined in the header files and interfaces of the Grid class.

Iterator

In order to access the discrete quantities at certain places easy and reliable, an Iterator will be defined. The abstract class `Iterator` possesses whole series of functions for this purpose. With `First()` and `Next()`, one can move over the whole definition zone of an Iterator. Or with functions `Up()`, `Down()`, `Right()` and `Left()`, one can access the neighbor elements. The Iterators `InteriorIterator` and `BoundaryIterator` specify the Iterator class for which the definition zone is restricted to the internal or boundary of the simulation domain, respectively.

Computation

The `Compute` class is the central unit of the flow solver. Here the actual simulations and computations takes place. The constructor initializes the simulation and requires the information about the simulation domain and its geometry as well as simulation parameters. The functions in the header file `Compute.hpp` are to be implemented according to section 3.2.6 of the script and the equations referenced there.

Solver

The arising linear system of equations for the pressure should be used iteratively. Chapter 4 of the script illustrates the function of an iterative solver in the example of Gauß-Seidel iteration and describes the extension to an SOR solver. The `Solver` class supports the base functionality of an iterative solver for linear system of equations. The `SOR`-Solver is inherited from the abstract `Solver` class and implement the `Cycle` method corresponding to equation (4.1) in the script.

Geometry

The `Geometry` class contains methods for loading or writing a rectangular geometry from or into a file, respectively. Here, the parameters `length`, `size` and `velocity` are read or written for each dimension. `length` gives the extension of the simulation domain in the corresponding spatial direction, `size` the resolution of the grid and `velocity` the value of the initial velocity. Moreover, the `Geometry` class includes methods, which set the boundary values for u, v and p. The boundary values muss be renewed at every time step. You will find notes for setting the boundary values for the u and v components of the fluid velocity in section 3.1.3 of the script. Setting of boundary values for p und der "preliminary" components F and G are clarified in section 3.2.3.

Parameter

The `Parameter` class reads the values from a parameter file into the program, save them and makes them public for other components through the Get-function. The necessary parameters are given in Table 1:

tend	end time t_{end}
tau	safety factor for time step τ
dt	time step
itermax	maximum number of iterations
eps	tolerance for pressure iteration ε
omega	relaxation factor ω
alpha	upwind difference factor α
re	Reynolds number Re

Table 1: Input parameters for the simulation.

VTK IO

The `VTK` class organizes output of the simulation results in `.vtk`-format for a later visualization with ParaView. This class is provided to you completely and must not be modified. However, you should familiarize yourself with the functionality to use the respective functions correctly.

1.3 Example

to start, you simulate a typical example from fluid mechanics, the driven cavity flow problem. Figure 3 shows the boundary conditions on our domain $\Omega := [0,1] \times [0,1] \subset \mathbb{R}^2$. A quadratic

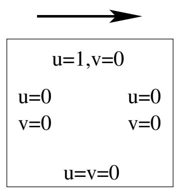


Figure 3: Randwertbedingungen für das Driven-Cavity-Problem.

container filled out with the fluid is simulated. On all boundaries no-slip boundary conditions are chosen. The upper layer is moved with a constant velocity such that the fluid (inside) moves. This effect (movement) is applied through the boundary condition values:

$$u_{i,jMax+1} = 2.0 - u_{i,jMax}, \qquad i = 1, \ldots, iMax$$

For the simulation, the following parameters will be used:

Parameter	value
$xSize = ySize$	128
$xLength = yLength$	1.0

Table 2: Geometry parameters for the driven cavity flow.

Parameter	value
Re	1000
ω	1.7
τ	0.5
eps	0.001
α	0.9
$itermax$	100

Table 3: Simulation parameters for the driven cavity flow.

Start the simulation and visualize the results with ParaView for times $t = 0.0 + i \cdot 0.2$ with $i \in [0, 82]$. You will find a manual for loading the results in ParaView in a next section.

1.4 Build system

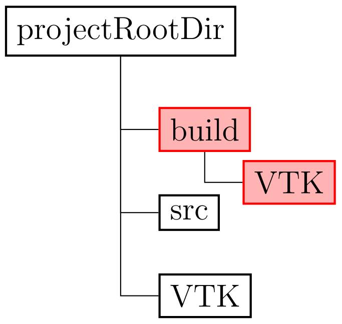
To compile your code, you have three possibilities:

a) to set up your own Makefiles

b) use of the provided CMake[3] environment

c) use of the provided SCons[4] environment

In the following, usage of the both build system will be shortly introduced. First, make sure that you have the following required folders. The build folder is optional because it is only used for CMake.



1.4.1 CMake

CMake is a popular build system, which is used by different big open-source projects, for example Deal.II (FEM library), Gmsh (library to create triangular mesh) or LAPACK (**L**inear **A**lgebra **Pack**age). In order to compile a project with CMake, you can optionally create a build folder. In this folder or the main folder of the project, you invoke cmake

```
cmake ../
```

This creates a makefile that is employed when you invoke

`make` $-j$ *number of processes*

To configure cmake you could either use the command line

`cmake -DDEBUG_VISU=ON resp. -DDEBUG_VISU=OFF` $*projectRootDir*

or the ncurses interface

`ccmake` $*projectRootDir*

The above parameter sets a flag, if the introduced visualization in section 1.5 should be compiled and shown (is realized in the source code through preprocessor directives). Besides, you could select if you want to compile a version with or without debug information.

[3] http://cmake.org/
[4] http://scons.org/

1.4.2 SCons

alternative to CMake, you could use SCons. In contrast to CMake, for which the the build system is defined in the `CMakeLists.txt` file, the SCons consists of Python file(s). The main file is called `SConstruct`. In this file, the build environment is established, hence, compiler flags are set. Afterwards, the file `SConscript` is invoked. In doing so a build folder is automatically created. In `SConscript` file, all necessary source files are looked for in conjunction, a name for the compiled file defined and finally the program is compiled. Two files are required in order to have a build folder and to look for the files not in the respective build folders but in their real places in the memory at the same time; for more information: https://bitbucket.org/scons/scons/wiki/VariantDir%28%29. To configure, if you would like to compile a visualization or not, use the file `custom.py`. In order to use the gdb, you could manage the debug information via a parameter of the command line.

```
scons debug=1 -j   number of processes.
```

The installation of SCons is done according to [5]. Afterwards, you must take the place of installation in the `${PATH}` environment:

```
export PATH=${HOME}/bin:${PATH}
```

In order not to input the command each time you login, you could create a `.bashrc` file and write the command therein. In order not to have differences when you login via ssh, you must additionally create a `.bash_profile` file and write the following there:

```
source .bashrc
```

This leads to the execution of the same commands independent from either your actual shell is a login-shell (via ssh) or an interactive shell (when you open a terminal from an arbitrary desktop environment).

[5] http://scons.org/doc/production/HTML/scons-user.html#idm139970758218448

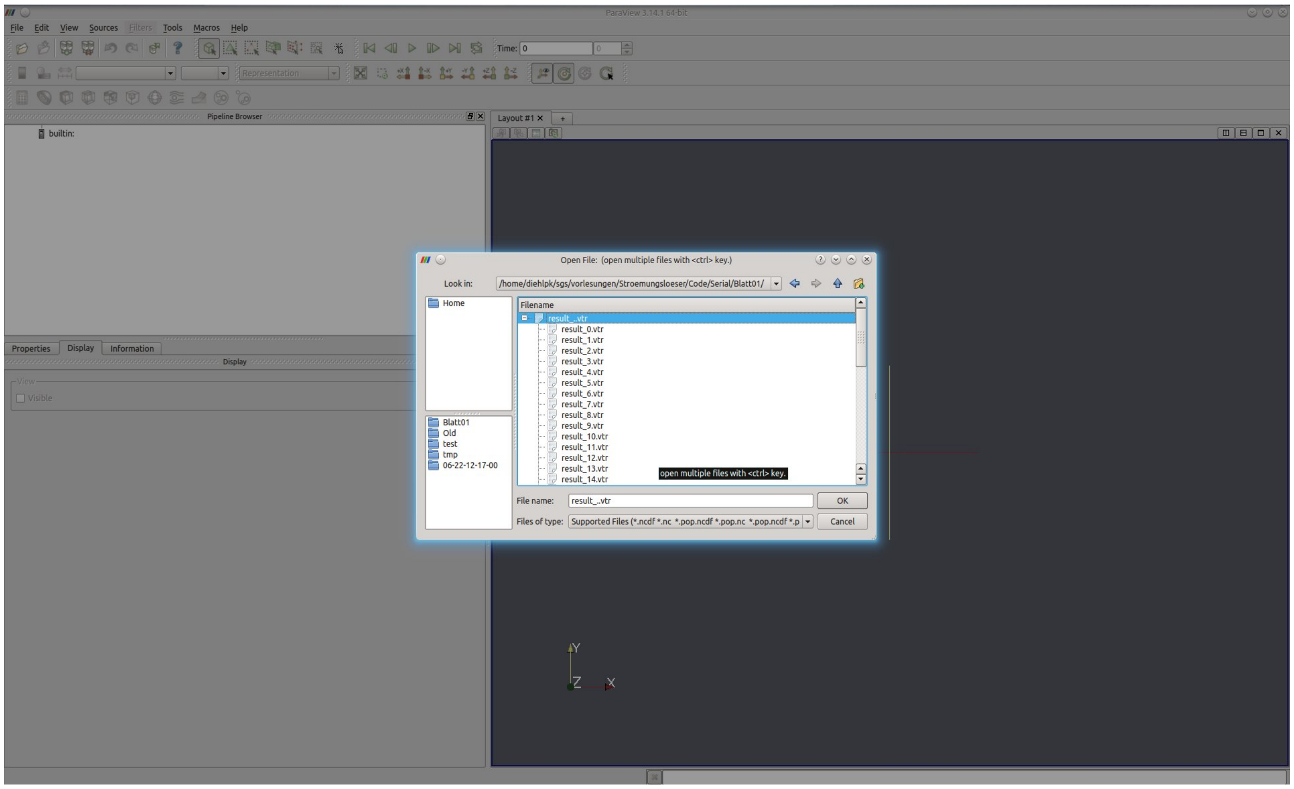
1.5 Visualization

To visualize the simulation results, there are two variants possible. One possibility is to read and visualize the `.vtk` file, containing the simulation data written by the program, with the open-source program ParaView[6]. Below, we will give a short tutorial to ParaView.
Moreover, the framework contains an SDL based live visualization of the simulation data.

Live visualization data

The whole visualization is provided in the `Renderer` class and must not be further modified. However, in order that the visualization performs without errors, the employed functions like `Grid::Interpolate()` and so on must be correctly implemented. The live visualization would be started in the main function in `main.cpp`.

Manual ParaView



Open File: (open multiple files with <ctrl> key.)

Look in: /home/diehlpk/sgs/vorlesungen/Stroemungsloeser/Code/Serial/Blatt01/

Home

Blatt01
Old
test
tmp
06-22-12-17-00

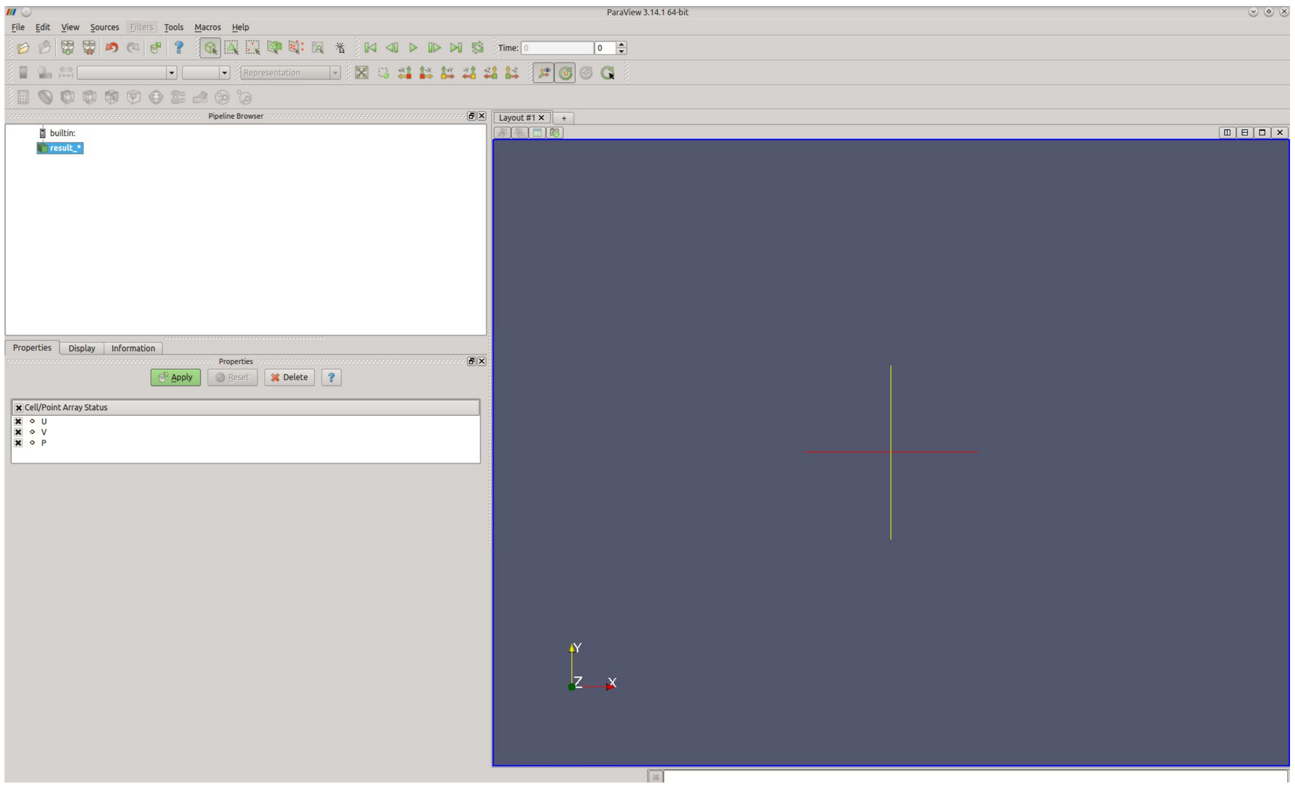
Filename
result_..vtr
result_0.vtr
result_1.vtr
result_2.vtr
result_3.vtr
result_4.vtr
result_5.vtr
result_6.vtr
result_7.vtr
result_8.vtr
result_9.vtr
result_10.vtr
result_11.vtr
result_12.vtr
result_13.vtr
result_14.vtr

open multiple files with <ctrl> key.

File name: result_..vtr
Files of type: Supported Files (*.ncdf *.nc *.pop.ncdf *.pop.nc *.pop.ncdf *.p

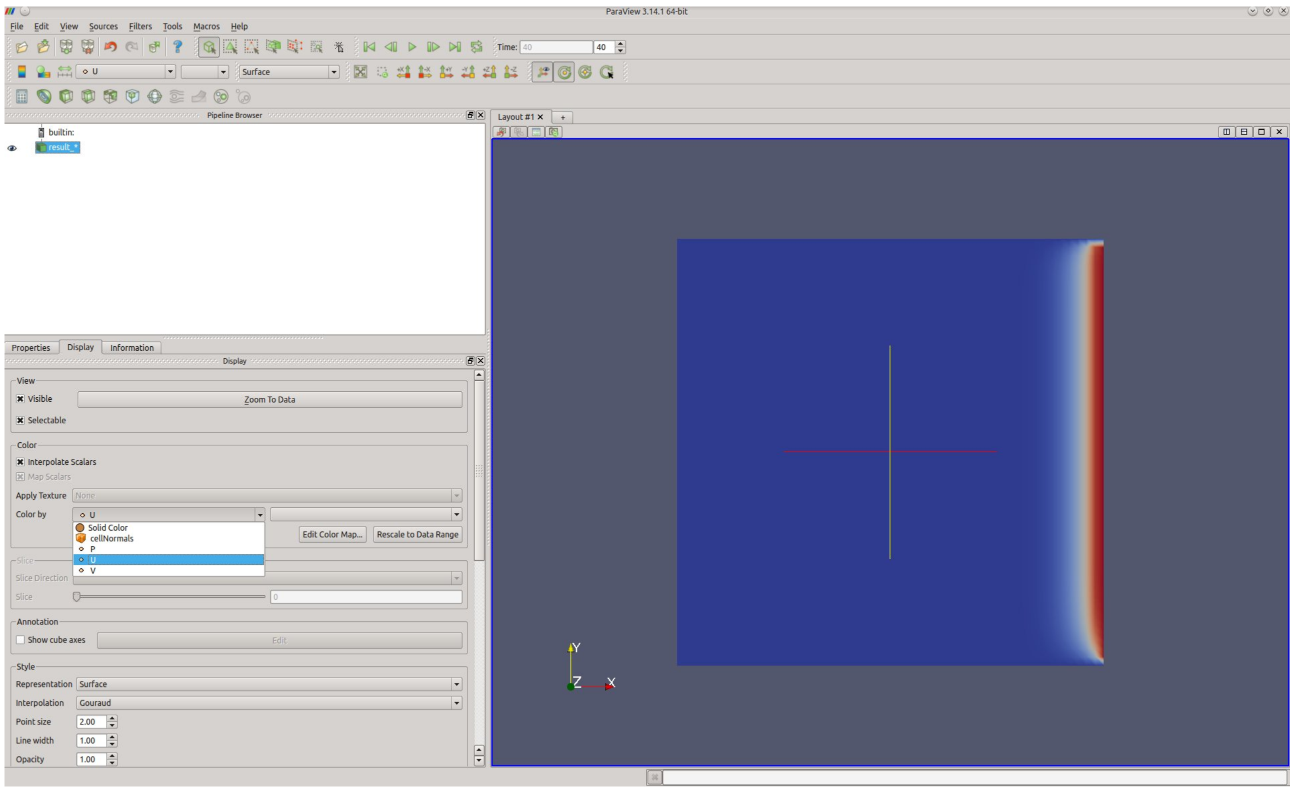
Figure 4: In order to open the simulation results altogether, click on *_..vts.

[6] http://www.paraview.org/



Cell/Point Array Status
- ☒ U
- ☒ V
- ☒ P

Figure 5: In tab *Properties* click on the button Apply to visualize the results.



View
- ☒ Visible
- ☒ Selectable

Color
- ☒ Interpolate Scalars
- ☒ Map Scalars

Apply Texture: None
Color by: U

Show cube axes: ☐

Style
- Representation: Surface
- Interpolation: Gouraud
- Point size: 2.00
- Line width: 1.00
- Opacity: 1.00

Figure 6: In tab *Display* you could choose via *Color by* which data set should be used for coloring the quadrilateral grid.

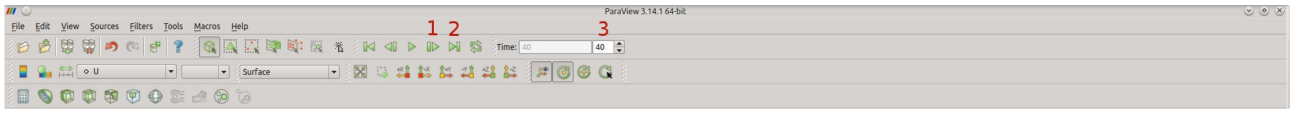


Figure 7: With button 1, one time step is gone forwards, Button 2 jumps to the last time step and the text field 3 shows the actual time step.

1.6 Questions

- Choose the time steps manually. For which time steps dt is the algorithm stable?
- What happen when the Reynolds number is changed, $(Re = 100, 500, 2000, 10000)$?
- We have claimed in the lecture that our discretization in space second order (by using central differences), respectively, first order (by using the Donner-Cell schemes) and in time first order accurate is. How could you verify that using your simulation program?
- Make a table for the run times of different grid resolutions. Do the results match our theoretical considerations from the lecture? Why / Why not?