Chalmers University of Technology

Department of Applied Mechanics

Division of Fluid Dynamics

CFD with OpenSource software, assignment 3

Tutorial - shallowWaterFoam

Developed for OpenFOAM-1.7.x

Author:
Johan Pilqvist

Peer reviewed by:
Patrik Andersson
Jelena Andric

November 1, 2010

Contents

Chapter 1

shallowWaterFoam

1.1 Introduction

This tutorial aims towards explaining the fundamentals of the solver shallowWaterFoam. The main focus is to analyze the code functionality and the implementation of the *shallow water equations*. Also, to give some context to the code functionality, the standard case squareBump is followed and thoroughly analyzed.

1.2 The shallow water equations

The *shallow water equations* are a set of differential equations that describe the motion of an incompressible fluid within a domain whose depth is considered to be shallow compared with the radius of the Earth.

Assuming that the fluid is incompressible and that the effects of vertical shear of the horizontal velocity is negligble, the equations can be derived by depth-integrating[1] the continuity and Navier-Stokes equations. Depth-integrating then allows the velocity *normal to gravity* to be removed from the equations. The assumption of incompressibility is partly the reason for the name *shallow water equations*, since water is more or less incompressible. The name also derives from the vertical shear assumption, which is reasonable only if the fluid is "shallow" [1].

The momentum and continuity equations for the shallow water equations read [2]

$$\frac{\partial}{\partial t}(h\mathbf{u}) + \nabla \cdot (h\mathbf{u}^T\mathbf{u}) + f \times h\mathbf{u} = -|\mathbf{g}|h\nabla(h + h_0) + \tau^w - \tau^b \tag{1.1}$$

$$\frac{\partial}{\partial t}(h + h_0) + \nabla \cdot (h\mathbf{u}) = 0 \tag{1.2}$$

where h is the mean surface height, $\mathbf{u}$ is the velocity vector, $f = (2\Omega \cdot \hat{\mathbf{g}})\hat{\mathbf{g}}$ is the *Coriolis force* (depending on the angular rotation rate of the Earth, Ω, and $\hat{\mathbf{g}}$ being the normal vector of gravity), h_0 is the deviation from the mean surface height and τ^w and τ^b are the wind and bottom stresses respectively.

In shallowWaterFoam, the wind and bottom stresses are assumed to be zero. Also, the surface velocity flux is defined as

$$\phi_v = \phi/h = \{\phi = h\mathbf{u} \cdot \hat{\mathbf{n}}\} = \mathbf{u} \cdot \hat{\mathbf{n}} \tag{1.3}$$

where $\hat{\mathbf{n}}$ is the cell face area vector. Equation 1.1 then reduces to

$$\frac{\partial}{\partial t}(h\mathbf{u}) + \nabla \cdot (h\phi_v\mathbf{u}) + f \times h\mathbf{u} = -|\mathbf{g}|h\nabla(h + h_0) \tag{1.4}$$

These are the equations that are solved in shallowWaterFoam.

[1] i.e. integrating from the surface topography up to the free-surface

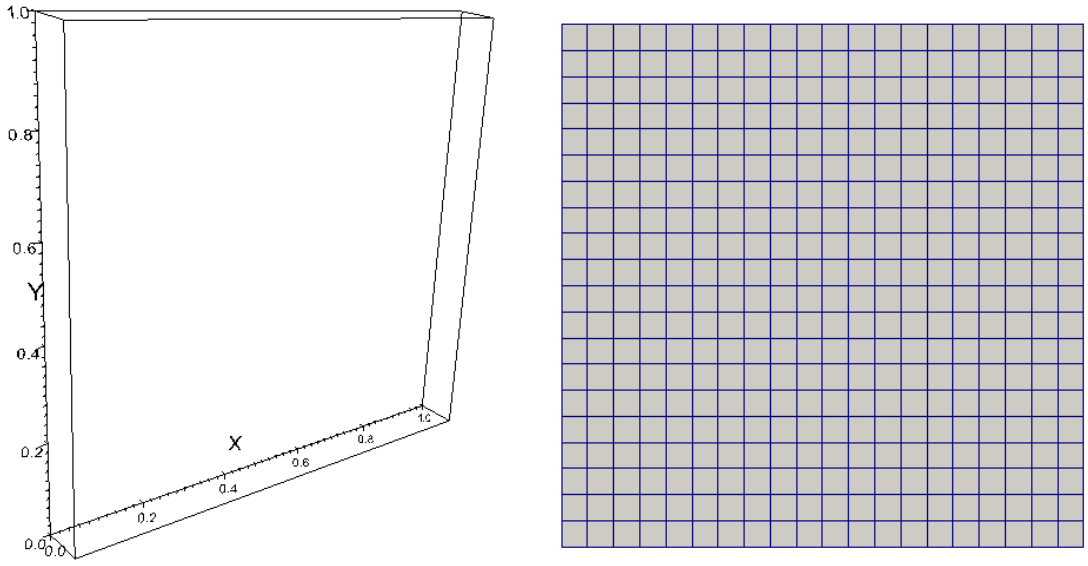
What may be noted is that although a vertical velocity term is not present in the shallow water equations, this velocity component is not necessarily zero. Consider for example a change of depth (i.e. the height of a free surface); then the vertical velocity could not possibly be zero. Once the horizontal velocities and free surface displacements have been solved for, the vertical velocities may be recovered using the equation of continuity.

1.3 Running the squareBump case

In this section we are going to apply the shallowWaterFoam solver on the squareBump case located in the tutorials folder. Before setting up this case, copy the entire folder from the tutorials section into your own run directory;

```
cd $FOAM_RUN
cp -r $FOAM_TUTORIALS/incompressible/shallowWaterFoam/squareBump .
cd squareBump
```

1.3.1 Meshing



(a) (b)

Figure 1.1: a) Geometry of the shallowWaterFoam tutorial case squareBump and b) the default 20×20 mesh.

The geometry in the squareBump case consists of a single hexahedron block with a 1×1 meter base and a depth of 0.1 meter (Figure 1.1a). To create a mesh we use the command blockMesh which generates a mesh following the descriptions in the dictionary blockMeshDict located in the subdirectory constant/polyMesh. In the squareBump case this dictionary reads as follows:

```
convertToMeters 1;

vertices
(
```

```
    (0 0 0)
    (1 0 0)
    (1 1 0)
    (0 1 0)
    (0 0 0.1)
    (1 0 0.1)
    (1 1 0.1)
    (0 1 0.1)
);

blocks
(
    hex (0 1 2 3 4 5 6 7) (20 20 1) simpleGrading (1 1 1)
);

edges
(
);

patches
(
    patch sides
    (
        (3 7 6 2)
        (1 5 4 0)
    )
    patch inlet
    (
        (0 4 7 3)
    )
    patch outlet
    (
        (2 6 5 1)
    )
    empty frontAndBack
    (
        (0 3 2 1)
        (4 5 6 7)
    )
);

mergePatchPairs
(
);
```

From this file we can conclude that the mesh is to be built up by a single hexahedron block of 20×20 cells (Figure 1.1b). Also, the default mesh is axially equidistant, as indicated by the uniform values of simpleGrading. Note that blockMeshDict also divides the boundary areas of the domain into different patches. These are later used in order to define the boundaries in the file boundary, which is also located in the subdirectory constant/polyMesh. Now, let's generate the mesh by typing

---

```
blockMesh
```

1.3.2 Boundary and initial conditions

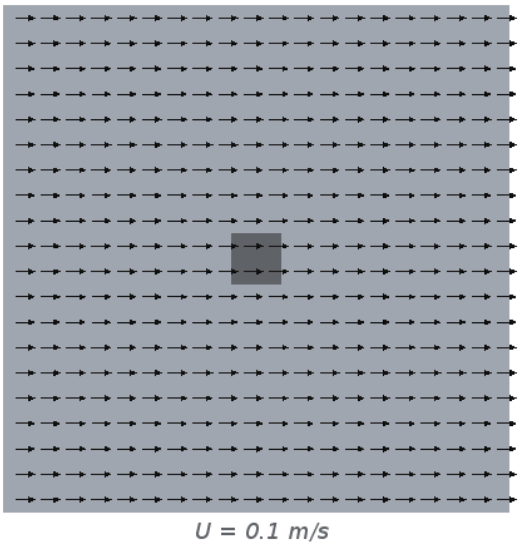


Figure 1.2: Visualization of the initial velocity vector field, $\mathbf{u} = (0.1\ 0\ 0)$ m/s, and the non-uniform initial surface height (denoted by different shades of gray).

In squareBump there are four initial conditions that need to be defined; the *mean* surface height h, the *deviation* from the mean surface height h_0, the *free-surface* height $h_{total}(= h + h_0)$ and the velocity vector field $\mathbf{u}$. The initial velocity vector field is uniformly distributed from the far left and onto the internal field of the domain, with a magnitude of 0.1 m/s. Moreover, the four midmost cells has an initial *mean* surface height of 0.009 meters (the darker square in the middle of Figure 1.2) simulating a quadratic obstacle, or a *square bump*.

The initial conditions related to the actual case should all be located in the subdirectory 0. Unfortunately, for some reason the file h0 is by default located in the wrong subdirectory, namely constant. To correct this mistake we need to move this file to the correct location (i.e. the subdirectory 0). This is done using the following command;

```
mv constant/h0 0/
```

Below, the contents of U (followed by h, hTotal and h0) have been included to show how the initial conditions are defined in the code.

```
dimensions      [0 1 -1 0 0 0 0];

internalField   uniform (0.1 0 0);

boundaryField
{
    sides
    {
        type            slip;
    }
    inlet
    {
```

```
        type            fixedValue;
        value           uniform (0.1 0 0);
    }
    outlet
    {
        type            zeroGradient;
    }
    frontAndBack
    {
        type            empty;
    }
}
```

We can see that the dimension for U is set as m/s. Since U belongs to the class volVectorField the values for U must all be given as vectors. As indicated in Figure 1.2, the internal field is set to have an initial velocity of (0.1 0 0) m/s (i.e. only velocity in the x-direction). The four boundary fields to consider in this case are (as previously mentioned) those defined in boundary, i.e. sides, inlet, outlet and frontAndBack. As can be seen the sides boundary is set as slip. This is because the flow is to be considered *irrotational* (or as a *potential flow*) without any boundary layers on the sides. Moreover, the inlet is given the same initial velocity as the internal field, the outlet is given a *homogeneous Neumann* condition defined as zeroGradient and the frontAndBack boundary is set as empty.

The code implementation in h is somewhat more troublesome, since the *mean* surface height is not entirely homogeneous throughout the domain (c.f. Figure 1.2). Thus the value for h must be explicitly defined for each cell. This is done using a non-uniform list of scalar values.

```
dimensions      [0 1 0 0 0 0 0];

internalField   nonuniform List<scalar>
400
(
inside these parantheses the mean surface heights for all internal cells
are declared.
)
;

boundaryField
{
    sides
    {
        type            zeroGradient;
    }
    inlet
    {
        type            zeroGradient;
    }
    outlet
    {
        type            fixedValue;
        value           uniform 0.01;
    }
    frontAndBack
    {
        type            empty;
```

```
    }
}
```

The free-surface height, h_{total}, is simply set to 0.1 meter throughout the entire domain using the command uniform.

```
dimensions      [0 1 0 0 0 0 0];

internalField   uniform 0.01;

boundaryField
{
    sides
    {
        type            calculated;
        value           uniform 0.01;
    }
    inlet
    {
        type            calculated;
        value           uniform 0.01;
    }
    outlet
    {
        type            calculated;
        value           uniform 0.01;
    }
    frontAndBack
    {
        type            empty;
    }
}
```

Since h is not uniformly distributed, but h_{total} is, the equality $h_{total} = h + h_0$ then calls for h_0 to balance h_{total}. To do this, h_0 is given the value 0 for all cells except the four midmost ones, using a non-uniform list of scalars (similar to the declaration of h).

```
dimensions      [0 1 0 0 0 0 0];

internalField   nonuniform List<scalar>
400
(
inside these parantheses the deviation from the mean surface height for all
internal cells are declared.
)
;

boundaryField
{
    sides
    {
        type            zeroGradient;
```

```
    }
    inlet
    {
        type            zeroGradient;
    }
    outlet
    {
        type            zeroGradient;
    }
    frontAndBack
    {
        type            empty;
    }
}
```

Now that the *point/cell* values of the initial conditions have been defined, we also need to apply them onto the mesh as *fields*. This is done using the utility setFields. This utility requires a dictionary, setFieldsDict, which should be located in the sub-directory system. In this case the contents of this dictionary reads

---

```
defaultFieldValues
(
    volScalarFieldValue h0 0
    volScalarFieldValue h 0.01
    volVectorFieldValue U (0.1 0 0)
);

regions
(
    boxToCell
    {
        box (0.45 0.45 0) (0.55 0.55 0.1);

        fieldValues
        (
            volScalarFieldValue h0 0.001
            volScalarFieldValue h  0.009
        );
    }
);
```

Now, to execute the setFields utility, simply type

```
setFields
```

Worth noting is that had the file h0 not been moved to the correct directory, this command could not have been executed. Instead, an error message would have been received stating that setFields.C was unable to locate the file h0.

1.3.3 Physical properties

As can be seen in equation 1.1, the *shallow water equations* are dependent of the gravitational force and the rotation of the Earth. These physical properties are defined in the file gravitationalProperties located in the subfolder constant. From this file we conduct that the gravitational force $g = 9.81$ m/s^2 and that the angular rotation rate of the Earth $\Omega = 7.292 \cdot 10^{-5}$ s^{-1}.

1.3.4 Controlling the simulation

Before running a case we need to set some preferences for controlling the simulation and calculation process, as well as the output of the results. These settings are done in the files located in the subfolder system. Let us first have a look in the file controlDict.

```
--------------------------------------------------------------------------------

application     shallowWaterFoam;

startFrom       startTime;

startTime      0;

stopAt          endTime;

endTime         100;

deltaT          0.1;

writeControl    runTime;

writeInterval   1;

purgeWrite      0;

writeFormat     ascii;

writePrecision  6;

writeCompression uncompressed;

timeFormat      general;

timePrecision   6;

runTimeModifiable no;

--------------------------------------------------------------------------------
```

We can see that the squareBump case is to be simulated from 0 to 100 seconds (remember that we have indeed defined all our initial conditions for $t = 0$ s, i.e. in the 0 directory) with a timestep, Δt, of 0.1 second. Also, the writeInterval is set to 1 s, meaning that we will write to our results every *one* seconds. Hence, we will *solve* for a thousand timesteps of which we will *save* the data for every tenth timestep.

The choice of timestep should be based on a CFL number that is low enough to assure convergence (rule of thumb is to use $CFL \leq 1$). The CFL number is defined as

$$CFL = \frac{|\mathbf{u}|\Delta t}{\Delta x} \tag{1.5}$$

where Δx is the side length of a cell. The condition $CFL \leq 1$ must be valid everywhere and since we in this case have a uniform initial velocity field, $|\mathbf{u}| = 0.1$ m/s, and a uniform mesh, $\Delta x = 0.05$ m, this means that for the first time step

$$\Delta t \leq \frac{\Delta x}{|\mathbf{u}|} = 0.5s \tag{1.6}$$

Thus, the choice of $\Delta t = 0.1$ s should be more than sufficient since the magnitude of the velocity is not likely to multiply by a factor of 5.

Now, let us also have a look inside fvSchemes to see what types of discretization schemes are to be used in the squareBump case.

```
--------------------------------------------------------------------------------

ddtSchemes
{
    default         CrankNicholson 0.9;
}

gradSchemes
{
    default         Gauss linear;
}

divSchemes
{
    default         none;
    div(phiv,hU)    Gauss linear;
}

laplacianSchemes
{
    default         Gauss linear uncorrected;
}

interpolationSchemes
{
    default         linear;
}

snGradSchemes
{
    default         uncorrected;
}

fluxRequired
{
    h;
}

--------------------------------------------------------------------------------
```

Firstly, we may note that the time discretization is set to Crank-Nicholson. It is, however, given a value of 0.9, meaning that it is not fully Crank-Nicholson[2]. We may also notice that the divergence scheme is explicitly specified.

[2]A value of 1 would give fully Crank-Nicholson.

Finally, let us analyze the solution procedures by having a look in fvSolution.

```
solvers
{
    h
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-6;
        relTol          0.01;
    };

    hFinal
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-8;
        relTol          0;
    };

    hU
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-6;
        relTol          0;
    }
}

PISO
{
    nOuterCorrectors 3;
    nCorrectors      1;
    nNonOrthogonalCorrectors 0;

    momentumPredictor yes;
}
```

The solver to be used for the h and hFinal (more about this property in section 1.5.1) equation systems is the conjugate gradient solver PCG, using the preconditioner DIC (Diagonal incomplete-Cholesky, used for symmetric matrices). The hU equation system (i.e. equations 1.4) is to be solved using the bi-conjugate gradient solver PBiCG with the preconditoner DILU (Diagonal incomplete-LU, used for asymmetric matrices) [3]. Finally, fvSolution also includes some input arguments to the PISO controls;

- nOuterCorrectors
- nCorrectors
- nNonOrthogonalCorrectors
- momentumPredictor

The implementation of these arguments and the PISO algorithm are further discussed in section 1.5.1.

1.3.5 Running the simulation

Now the case has been thoroughly setup and is ready to be solved. Go to the terminal prompt and type (make sure you are located in the squareBump folder)

```
shallowWaterFoam
```

Sometimes you might want to save the output of the calculation process in a log file, so that you can open it and read it later on. To do this, and at the same time be able to monitor the process in the terminal window, you can type

```
shallowWaterFoam 2>&1 | tee log
```

This way the same output that is written to the terminal window is simultaneously written to the file log which will be located in the current folder.

1.4 Post-processing

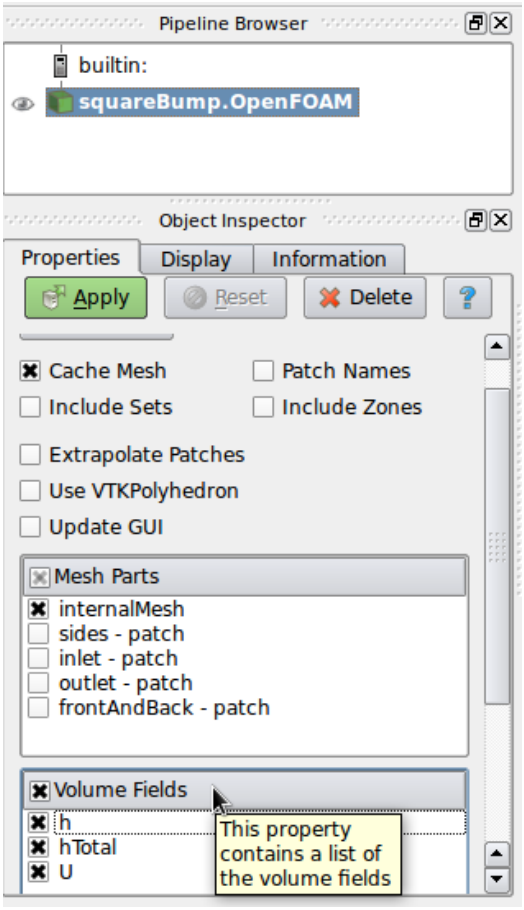


Figure 1.3: The outlines of *Pipeline Browser* and *Object Inspector* in ParaView.

The results of the simulations can be viewed using ParaView. Go to the terminal window and type

```
paraFoam
```

In the window *Pipeline Browser* you can see the name of the case being post-processed (in this case squareBump). Under the outline *Object Inspector* you can choose the variables you wish to import for analysis. Let's choose all the volume fields available and then click **Apply** (cf. Figure 1.3). Let us first have a look at the distribution of the velocity. To visualize this we simply choose U in the display options (cf. Figure 1.4).

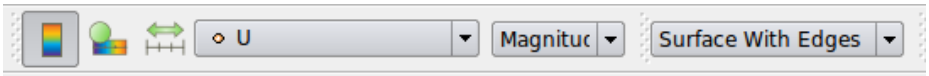


Figure 1.4: From the display options you can choose to view the velocity distribution U. In this case we have chosen to visualize the *interpolated* velocity distribution and also to show the mesh grid (*Surface With Edges*).

Figure 1.5 shows the velocity field at four different times. After 1 s, the velocity is largely influenced by the differences in surface height and continuity forces the flow to disrupt the uniform flow

conditions set for $t = 0$ s. Also, the coupling between $\mathbf{u}$ and h in the momentum equations (eqs. 1.1) causes the flow to initiate a wave motion. As time passes, the flow is decelerated and the wave motion consequently fades out (cf. Figure 1.6). Hence, the differences between, for example, 30 and 70 s in Figure 1.5 are quite small.

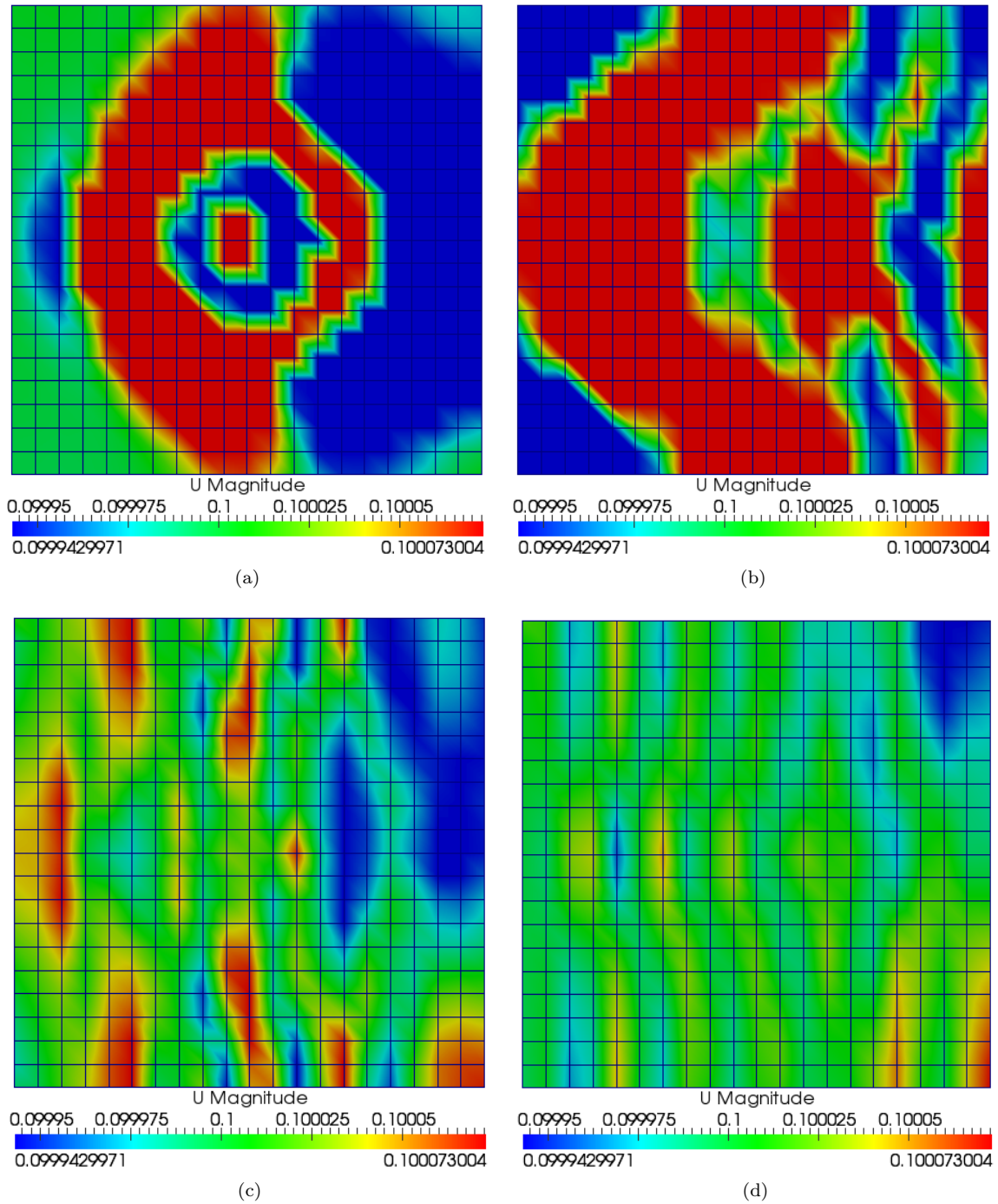


Figure 1.5: Velocity field after a) 1 s, b) 10 s, c) 30 s and d) 70 s.

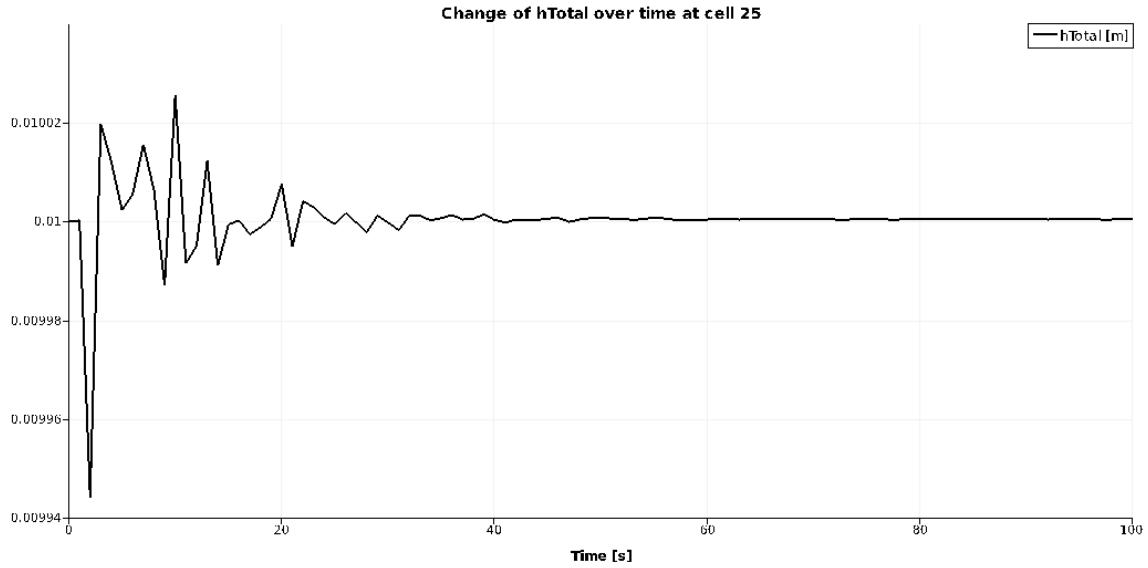


Figure 1.6: Visualization of the wave motion (i.e. change of free-surface height, h_{total}) and how it fades over time. Data plotted for cell number 25.

The plot in Figure 1.6 was created using Plot Selection Over Time from the menu Filters in ParaView. The *Selection* in this case was the 25^{th} cell (cf. Figure 1.7). To make a single cell selection, use the Select Cells On tool from the menu bar (cf. Figure 1.8) and simply click on the cell[3] of your choice.

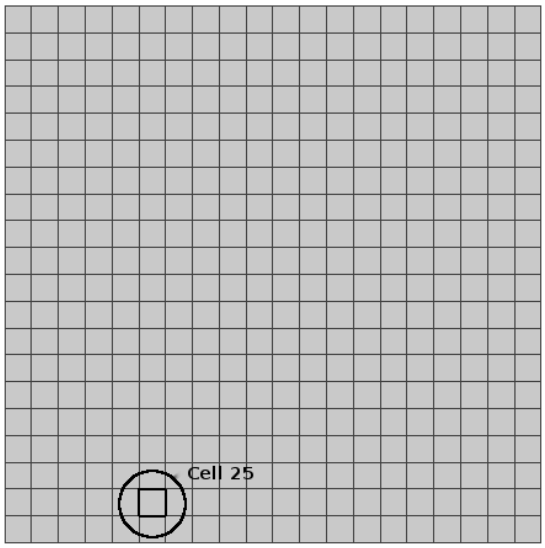


Figure 1.7: Location of cell number 25.

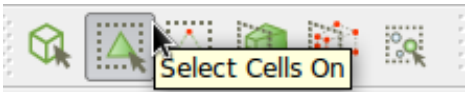


Figure 1.8: The tool Select Cells On as seen from the menu bar.

[3]It is also possible to mark multiple cells

1.5 The solver

To start the analysis of the solver, let us have a look in its .C-file. The path to this file is

```
$FOAM_APP/solvers/incompressible/shallowWaterFoam/shallowWaterFoam.C
```

but for convenience its contents can also be seen in full in Appendix A.

The very first line in shallowWaterFoam.C reads

```
#include "fvCFD.H".
```

This type of command is used to treat the contents of the header file fvCFD.H as if they had instead appeared at this point in the .C-file. In other words; instead of typing

```
#ifndef fvCFD_H
#define fvCFD_H

#include "parRun.H"

#include "Time.H"
#include "fvMesh.H"
#include "fvc.H"
#include "fvMatrices.H"
#include "fvm.H"
#include "linear.H"
#include "uniformDimensionedFields.H"
#include "calculatedFvPatchFields.H"
#include "fixedValueFvPatchFields.H"
#include "adjustPhi.H"
#include "findRefCell.H"
#include "mathematicalConstants.H"

#include "OSspecific.H"
#include "argList.H"
#include "timeSelector.H"

#ifndef namespaceFoam
#define namespaceFoam
    using namespace Foam;
#endif

#endif
```

(i.e. the contents of fvCFD.H) in the beginning of shallowWaterFoam.C, we simply type

```
#include "fvCFD.H".
```

As can be seen, fvCFD.H does in turn *include* several other .H-files, of which most subsequently *include* additional ones. All these inclusions are needed in order to define and/or declare different classes, types, functions and variables used in shallowWaterFoam.C and its subfiles.

When all fundamental inclusions have been made, the solver executes the first function main, which (if completed succesfully) returns the integer value 0 (see the last line of shallowWaterFoam.C). As is the case for the .C-file itself, the main-function starts off with some inclusions;

- setRootCase.H makes sure that shallowWaterFoam is executed in a valid directory. If not, it returns the message "`FOAM FATAL IO ERROR`".
- createTime.H defines the time properties according to the settings in the controlDict file.
- createMesh.H reads the mesh generated by blockMesh for which the equations are to be solved.
- readGravitationalAcceleration.H reads the gravitational constant, g, and the angular rotation rate of the Earth, Ω, to later be used in the equations.
- createFields.H reads the initial values for the different scalar and vector fields (e.g. h, $\mathbf{u}$ etc.) defined in the current case. It also calculates the *Coriolis* force and names it F.

1.5.1 Solving the equations

When the initial conditions, geometry, mesh and time properties have all been defined it is time to start the iteration process to solve equations 1.4 (i.e. the velocity distribution, $\mathbf{u}$, and the free-surface height, $h_{total} = h + h_0$). This is done using a while loop which runs over all the time steps defined in controlDict. The shallowWaterFoam solver uses the PISO algorithm. PISO stands for Pressure Implicit with Splitting of Operators and was *originally* a pressure-velocity calculation procedure developed for non-iterative computation of unsteady compressible flows, using one predictor step and two corrector steps. It has however been successfully adapted for the iterative solution of steady state problems and may be seen as an extension of the SIMPLE algorithm [4].

The inclusion of CourantNo.H in the beginning of the while loop is needed to calculate the *Courant* (or CFL, eq. 1.5) number used to evaluate the time step. Provided that the squareBump case was run using the command

```
shallowWaterFoam 2>&1 | tee log
```

the reader could easily confirm that $CFL \leq 1$ at all times, i.e. that the *Courant* number did not exceed the value *one* at any time step, by having a look in the file log.

From fvSolution (section 1.3.4) we can recall the input arguments to the PISO loop. For clarity, these have also been tabulated in Table 1.1.

Table 1.1: The input arguments to the PISO loop defined in fvSolution.

PISO argument	Value	Description
nOuterCorrectors	3	Input value to the outer for loop solving the entire equation system
nCorrectors	1	Input value to the correction loop for ϕ (i.e. the face flux field)
nNonOrthogonalCorrectors	0	Input value to the correction loop for h (i.e. the mean surface height)
momentumPredictor	yes	Activation of momentum predictor

The PISO arguments are read into the solver code via the inclusion of readPISOControls.H. Through this file some of the arguments experience a slight change of name, or rather become abbreviated. Still, they are easily recognized in the solver code as nOuterCorr, nCorr and nNonOrthCorr.

By having a quick look at the conditions of the for loops in the solver code (Appendix A) and Table 1.1, we can conclude that

- the hU equation system (i.e. equations 1.4) is solved *three* times for each time step
- the correction of ϕ and h is done *once* every time the hU equation system is being solved.

Hence, in shallowWaterFoam, the PISO algorithm has been modified to use only *one* correction step for the face flux field, ϕ, and the mean surface height, h. Instead it uses three *outer* correction steps for the entire momentum equation system.

What may be noted is that the for loop concerning the non-orthogonal grids, i.e.

```
for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
```

will indeed be executed even though nNonOrthogonalCorrectors was given the value 0 in the PISO controls. Moreover, the code within

```
if (momentumPredictor){}
```

will be executed since momentumPredictor was set to yes. This step is usually incorporated to generate a good initial guess for the PISO loop in order to speed up the convergence. This is not always the case though. As a matter of fact, the squareBump case converges faster when momentumPredictor is inactivated (i.e. set to no). This is because it is (in this case) actually more time-consuming to execute the momentumPredictor-code than it is to have a less correct start guess[4].

The reocurring condition

```
if (rotating)
```

means that the solver shallowWaterFoam has been written so that it allows for the user to disregard the rotation of the Earth. Since rotating was set to true in gravitationalProperties, equations 1.4 were solved in their entirety (i.e. also accounting for the Earth's rotation). Setting rotating to false would instead mean that the *Coriolis* force would be ignored.

The left-hand side[5] of the momentum equation system (named hUEqn in the code) is defined as

```
fvVectorMatrix hUEqn
(
    fvm::ddt(hU)
  + fvm::div(phiv, hU)
);
```

where ddt denotes the time-derivative $\partial/\partial t$ and div denotes the divergence (i.e. $\nabla\cdot$). The operators ddt and div use the discretization schemes that were defined in fvSchemes.

As an attempt to get faster convergence, the shallowWaterFoam solver uses under-relaxation to solve the hU equation system. This is done using the command

```
hUEqn.relax();
```

[4]The reader might verify this simply by setting momentumPredictor to no in fvSolution, re-run the case and compare the final values for ClockTime.

[5]Except the *Coriolis* part which, as previously mentioned, is optional to take into consideration

The full hU equation system is now defined and solved as follows (provided that the rotation of the Earth is accounted for);

```
hUEqn + (F ^ hU) == -magg*h*fvc::grad(h + h0)
```

where F is the *Coriolis* force, magg denotes $|\mathbf{g}|$ and grad denotes the gradient operator (∇). Just as ddt and div the operator grad uses the discretization scheme defined in fvSchemes.

One part of the description section of shallowWaterFoam.C reads

```
If the geometry is 3D then it is assumed to be one layers of cells and the
component of the velocity normal to gravity is removed.
```

This corresponds well with the derivation of equations 1.1 and 1.2 in section 1.2, where depth integration allowed the velocity normal to gravity to be removed. In the code, removing the velocity normal to gravity is arranged by the reocurring lines

```
if (mesh.nGeometricD() == 3)
{
    hU -= (gHat & hU)*gHat;
}
```

where the "-=" operator means that the left-hand side is equal to the left-hand side *minus* the right-hand side and gHat refers to $\hat{\mathbf{g}}$ (i.e. the normal vector of gravity).

Recall, once again from fvSolution (section 1.3.4), the distinction made between h and hFinal. These are, in fact, both the same as h, with the slight difference that hFinal is the mean surface height *only for the third* calculation at each time step. The distinction is probably made to allow the tougher tolerance that is set for hFinal, i.e. the final calculation of h.

As aforementioned, ϕ and h are corrected once for every outer loop. The corrections of these parameters are done within the PISO loop in the code. For ϕ the correction implementation reads

```
surfaceScalarField hf = fvc::interpolate(h);
volScalarField rUA = 1.0/hUEqn.A();
surfaceScalarField ghrUAf = magg*fvc::interpolate(h*rUA);

surfaceScalarField phih0 = ghrUAf*mesh.magSf()*fvc::snGrad(h0);

if (rotating)
{
    hU = rUA*(hUEqn.H() - (F ^ hU));
}
else
{
    hU = rUA*hUEqn.H();
}
phi = (fvc::interpolate(hU) & mesh.Sf())
    + fvc::ddtPhiCorr(rUA, h, hU, phi)
    - phih0;
```

where

- hf is the interpolated mean surface height,
- rUA is the reciprocals of the coefficient matrix for the hU equation system,
- ghrUAf is rUA times the magnitude of $\mathbf{g}$ and the interpolated mean surface height,
- phih0 is ghrUAf times the cell face area and the normal gradient of h_0 ($\mathbf{n} \cdot \nabla h_0$) using the corresponding discretization scheme defined in fvSchemes.

The correction implementation for h reads as follows

```
---------------------------------------------------------------------------

fvScalarMatrix hEqn
(
    fvm::ddt(h)
  + fvc::div(phi)
  - fvm::laplacian(ghrUAf, h)
);

if (ucorr < nOuterCorr-1 || corr < nCorr-1)
{
    hEqn.solve();
}
else
{
    hEqn.solve(mesh.solver(h.name() + "Final"));
}

if (nonOrth == nNonOrthCorr)
{
    phi += hEqn.flux();
}

---------------------------------------------------------------------------
```

where laplacian denotes the laplace operator, $\Delta = \nabla^2 = \nabla \cdot \nabla$, which accordingly uses the dicretization scheme defined in fvSchemes.

The corrected ϕ and h are finally implemented in the momentum equation via the command

```
---------------------------------------------------------------------------

hU.correctBoundaryConditions();

---------------------------------------------------------------------------
```

Bibliography

[1] D. A. Randall, "The Shallow Water Equations," tech. rep., Department of Atmospheric Science, Colorado State University, 2006. `http://kiwi.atmos.colostate.edu/group/dave/pdf/ShallowWater.pdf`.

[2] Rutgers University, Institute of Marine and Coastal Sciences, "Shallow Water Equations." `http://marine.rutgers.edu/po/tests/soliton/eqs.php`, October 2010.

[3] OpenCFD Ltd., "Solution and algoritm control." `http://www.openfoam.com/docs/user/fvSolution.php`, October 2010.

[4] H. K. Versteeg and W. Malalasekera, *An Introduction to Computational Fluid Dynamics, the finite volume method*. Harlow, England: Pearson Education Limited, 2nd ed., 2007.

Appendix A

shallowWaterFoam.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 1991-2010 OpenCFD Ltd.
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

Application
    shallowWaterFoam

Description
    Transient solver for inviscid shallow-water equations with rotation.

    If the geometry is 3D then it is assumed to be one layers of cells and the
    component of the velocity normal to gravity is removed.

\*---------------------------------------------------------------------------*/

#include "fvCFD.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

int main(int argc, char *argv[])
```

```
{
    #include "setRootCase.H"
    #include "createTime.H"
    #include "createMesh.H"
    #include "readGravitationalAcceleration.H"
    #include "createFields.H"

    // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

    Info<< "\nStarting time loop\n" << endl;

    while (runTime.loop())
    {
        Info<< "\n Time = " << runTime.timeName() << nl << endl;

        #include "readPISOControls.H"
        #include "CourantNo.H"

        for (int ucorr=0; ucorr<nOuterCorr; ucorr++)
        {
            surfaceScalarField phiv("phiv", phi/fvc::interpolate(h));

            fvVectorMatrix hUEqn
            (
                fvm::ddt(hU)
              + fvm::div(phiv, hU)
            );

            hUEqn.relax();

            if (momentumPredictor)
            {
                if (rotating)
                {
                    solve(hUEqn + (F ^ hU) == -magg*h*fvc::grad(h + h0));
                }
                else
                {
                    solve(hUEqn == -magg*h*fvc::grad(h + h0));
                }

                // Constrain the momentum to be in the geometry if 3D geometry
                if (mesh.nGeometricD() == 3)
                {
                    hU -= (gHat & hU)*gHat;
                    hU.correctBoundaryConditions();
                }
            }

            // --- PISO loop
            for (int corr=0; corr<nCorr; corr++)
            {
                surfaceScalarField hf = fvc::interpolate(h);
                volScalarField rUA = 1.0/hUEqn.A();
```

```
        surfaceScalarField ghrUAf = magg*fvc::interpolate(h*rUA);

        surfaceScalarField phih0 = ghrUAf*mesh.magSf()*fvc::snGrad(h0);

        if (rotating)
        {
            hU = rUA*(hUEqn.H() - (F ^ hU));
        }
        else
        {
            hU = rUA*hUEqn.H();
        }

        phi = (fvc::interpolate(hU) & mesh.Sf())
            + fvc::ddtPhiCorr(rUA, h, hU, phi)
            - phih0;

        for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
        {
            fvScalarMatrix hEqn
            (
                fvm::ddt(h)
              + fvc::div(phi)
              - fvm::laplacian(ghrUAf, h)
            );

            if (ucorr < nOuterCorr-1 || corr < nCorr-1)
            {
                hEqn.solve();
            }
            else
            {
                hEqn.solve(mesh.solver(h.name() + "Final"));
            }

            if (nonOrth == nNonOrthCorr)
            {
                phi += hEqn.flux();
            }
        }

        hU -= rUA*h*magg*fvc::grad(h + h0);

        // Constrain the momentum to be in the geometry if 3D geometry
        if (mesh.nGeometricD() == 3)
        {
            hU -= (gHat & hU)*gHat;
        }

        hU.correctBoundaryConditions();
    }
}

U == hU/h;
```

```
        hTotal == h + h0;

        runTime.write();

        Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
            << "  ClockTime = " << runTime.elapsedClockTime() << " s"
            << nl << endl;
    }

    Info<< "End\n" << endl;

    return 0;
}


// ************************************************************************* //
```