UNIVERSITY
OF WARSAW

Introduction to Computational Fluid Dynamics (CFD) simulations using OpenFOAM

Student Team Project

Summer Semester 2023

Dawid Woś
Joydeep Sarkar
Mark Passia
Mateusz Kapusta

Supervised by
Rishab Prakash Sharma

Contents

1 Introduction

OpenFoam is a Computational Fluid Dynamics (CFD) framework designed for the simulations of fluids/continuum mechanics, developed initially at Imperial Collage London. Since the creation software have been popularized as a open and easy to extend framework suitable for different type of simulations ranging from simple diffusion up to multiphase fluids and Magnetohydrodynamics (MHD). It is written in C++ language and utilizing modern features like objective oriented programing (OOP).

In this project a basic introduction to the CFD using OpenFoam was given. Report is structured as follows. In the first part practical introduction to OpenFoam will be given including mesh generation, solver creation and practical considoration will be given. In second part some interesting cases will be considered together with analitical comparison.

2 OpenFoam programing

In order to provide an easy introduction for OpenFoam a simple diffusion case will be considered in this chapter. At the end of this chapter reader should be able to reproduce this simple example from the beginning. We would like to solve a diffusion equation

$$D_T \nabla^2 U + \frac{\partial T}{\partial t} = 0, \tag{1}$$

on the block with two different temperatures at both sides.

Before diving into the specifics of each simulation step, let's take a brief overview of typical OpenFOAM case directory, which is structured into several folders and files. For a simple heat diffusion case, the directory looks as follows:

```
heatDiffusionCase/
|-- 0/
    |-- T
|-- constant/
    |-- transportProperties
|-- system/
    |-- controlDict
    |-- fvSchemes
    |-- fvSolution
```

The files are grouped such that each folder contains files associated with a specific tasks:

- **0/T**: This file contains the initial and boundary conditions for the temperature field.
- **constant/thermophysicalProperties**: This file defines the physical properties of the materials involved in the simulation.
- **system/controlDict**: This file controls the overall execution of the simulation, including start and stop times and the write interval.
- **system/fvSchemes**: This file specifies the numerical schemes used for different terms in the equations.
- **system/fvSolution**: This file defines the solvers and related settings for the simulation.

2.1 Mesh generation

In OpenFOAM, the geometry of the domain is defined using mesh files. For a simple heat diffusion simulation, we create the mesh using the `blockMesh` utility, which reads the mesh description from a file named `blockMeshDict` located in the `system` directory. In which both the geometry and the mesh of the simulated object are defined. Here are the keywords used in the exaple blockMeshDict file[1]:

- **FoamFile { ... }**: This section defines metadata for the file, including version, format, and object type, which at this point is the blockMeshDict.
- **scale 1;**: Sets the scale factor for the geometry.

[1] It is also possible to simulate in OpenFOAM using third-party CAD models. This can be done by converting the CAD models into a suitable mesh format using tools like `snappyHexMesh` or third-party software such as `gmsh`.

- **xmin**, **xmax**, **ymin**, **ymax**, **zmin**, **zmax**: Define the dimensions for the block.
- **xcells**, **ycells**, **zcells**: Define the number of cells in the x, y, and z directions, respectively.
- **vertices { ... }**: Defines the vertices of the blocks. Each vertex is given by '(x y z)'.
- **blocks { ... }**: Defines the hexahedral blocks using the vertices specified. Each block is assigned the number of cells and grading.
  - **simpleGrading (1 1 1)** specifies the mesh grading in the x, y, and z directions. Grading controls the distribution of cells within the block. Values of 1 means no expansion or contraction, resulting in uniformly sized cells.
- **boundary { ... }**: Defines the boundary patches and their types, such as 'wall'. Each boundary is associated with specific faces of the blocks.

The example file used, which can be viewed in the appendix, results in the following geometry and mesh, which will subsequently be simulated; see Fig. 1.

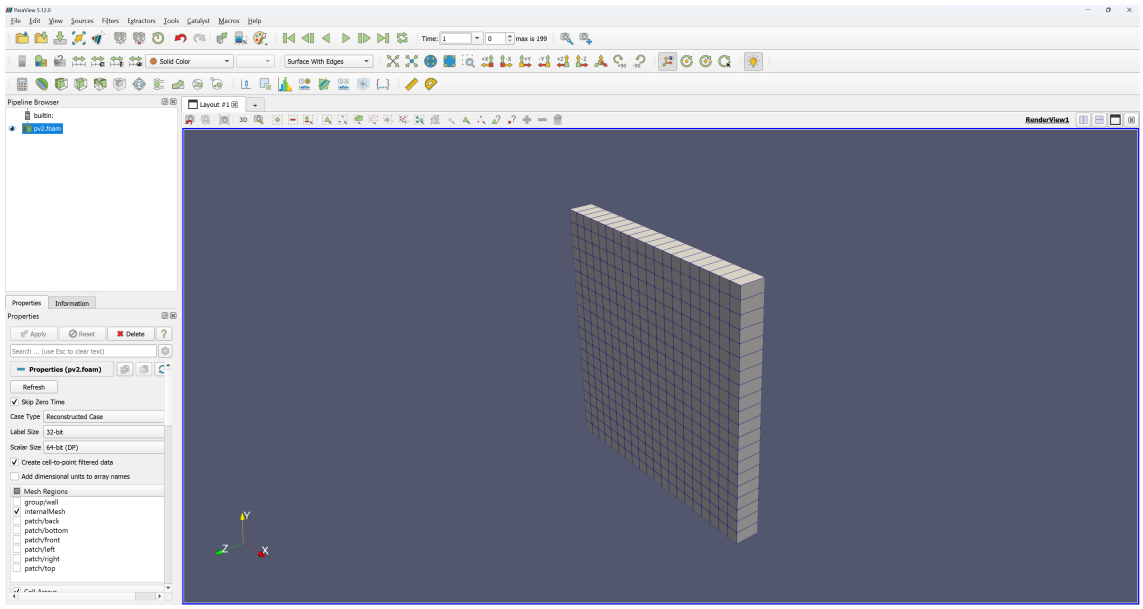

Figure 1: Created geometry and mesh shown in Paraview.

2.2 Material properties

Every simulation of a physical object requires inputting the material properties of the simulated object, be it a fluid or a solid. In OpenFOAM the material properties for the simulation are defined in the `constant/transportProperties` file. In our case the only variable which can be changed in that file is thermal diffusivity, abbreviated as `DT`. It is a measure of how quickly temperature changes occur within a material when it is subjected to a thermal gradient. The corresponding file structure is as follows:

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      transportProperties;
}

DT              0.01;
```

2.3 Boundary and initial conditions

In order to define the starting values and the boundary conditions for the simulated setup, we need to manipulate the content of the `0/T` file. Next, let's take a look at each command of the following example file and inspect what it achieves.

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      T;
}

dimensions      [0 0 0 1 0 0 0];
internalField   uniform 273;
boundaryField
{
    top{
        type            fixedValue;
        value           uniform 573;
    }

    bottom{
        type            fixedValue;
        value           uniform 273;
    }

    left{
        type            zeroGradient;
    }
    (...)
}
```

- **dimensions**: This line defines the physical dimensions of the field variable. The vector [0 0 0 1 0 0 0] indicates that the variable has the dimensions of temperature.
  - The dimensions in the `dimensions` entry of the OpenFOAM file are specified in the following order $[M\ L\ T\ \Theta\ N\ I\ J]$, where:
    - M is mass
    - L is length
    - T is time
    - Θ is temperature
    - N is quantity of substance
    - I is electric current
    - J is luminous intensity
- **internalField**: This defines the initial condition for the entire domain.
  - **uniform 273**: Sets the initial temperature to a uniform value of 273 Kelvin throughout the domain.
- **boundaryField**: This section specifies the boundary conditions for the field variable at the boundaries of the domain.
  - **top**:
    - **type fixedValue**: Specifies that the boundary condition is a fixed value.
    - **value uniform 573**: Sets the temperature at the top boundary to a uniform value of 573 Kelvin.
  - **bottom**:
    - **type fixedValue**: Specifies that the boundary condition is a fixed value.

    - **value uniform 273**: Sets the temperature at the bottom boundary to a uniform value of 273 Kelvin.
  - **left, right, front, back**:
    - **type zeroGradient**: Specifies a zero gradient boundary condition, meaning there is no change in temperature across these boundaries. This effectively makes these boundaries insulating or adiabatic, preventing heat flux through them.

2.4 Numerical Schemes

After determination of the grid and equation, we need to translate differential equation to our grid. There are may different types of challanges associated with the choice of numerical schemes. All properties associated with schemes are defined in `system/fvSchemes` file. A simple example of such directory may look like this:

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSchemes;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

ddtSchemes
{
    default         Euler;
}

gradSchemes
{
    default         Gauss linear;
}

laplacianSchemes
{
    default         Gauss linear orthogonal;
    laplacian(DT,T) Gauss linear corrected;
}

interpolationSchemes
{
    default linear;
}
```

Let's start with `ddTSchemes`. This entry specifies how a time derivative is translated. In case of simple Euler scheme we are approximating

$$\frac{\partial X}{\partial t} \approx \frac{X_{i+1} - X_i}{\Delta t}, \tag{2}$$

where Δt is a timestep of our simulation. There are few other schemes that might be used such as a Crank-Nicolson time scheme. In similar manner other entries control how other part of equation are handled. Let's take a closer look at `gradSchemes`. A Gauss keyword states, that a Gauss theorem is used to compute

$$\boldsymbol{\nabla} U = \frac{1}{V} \sum_f \mathbf{S_f} U_f, \tag{3}$$

where V stands for the volume of the cell, $\mathbf{S_f}$ stands for the normal vector to the face f and U_f is a *approximate* value of field U at face f. This is where the second keyword comes in, as `linear` specifies how U_f is beeing calculated. In this case we

are using linear approximation and assuming mean value of U field in two neighbouring cells separated by face f. There are different schemes that can be used here, some can be field-dependent as a `upwind` scheme or can be adjusted with additional parameters like `LimitedLinear`.

2.5 Writing solvers

2.5.1 Diffusion

All of previous chapters explained how to write a geometry for a OpenFoam case and how to to use numerical schemes. In this chapter an introduction will be given on the solver creation. Before we dive into the details we need to take a moment to understand how fields work. Usually field are defined in the *center* of the cell (as for example temperature in diffusion case). Hence, we start our solver with

```
volScalarField T
(
    IOobject
    (
        "T",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
```

We have specified an object with type `volScalarField` that we need to read at the beginning (from file `0/T`) and which will be automatically saved. We are also specifing mesh on which our field is anchored. Than we have a simple loop

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

        while (simple.correctNonOrthogonal())
        {

            fvScalarMatrix TEqn
            (
                fvm::ddt(T)
               -fvm::laplacian(DT, T)
             ==
                fvOptions(T)
            );
            fvOptions.constrain(TEqn);
            TEqn.solve();
            fvOptions.correct(T);
        }
        #include "write.H"
        runTime.printExecutionTime(Info);
}
```

We are defining an equation with type `fvScalarMatrix` which is defined with operators `fvm::ddt` and `fvm::laplacian(DT,T)`. Both of those are translated to the discretized matrix equation using numerical schemes that we have already covered. There are also more advanced options present as constrains and corrections which we won't delve into. Usually this is enough for such solvers to work in most basic possible setup.

2.5.2 Transport equation

Now let's start with basic modification to the diffusion equation known as the transport equation:

$$\frac{\partial T}{\partial t} - D\nabla^2 T + \boldsymbol{\nabla} \cdot (T\mathbf{u}) = 0,$$

where **u** denotes the velocity of the fluid. In general velocity needs to be obtained with the help of the continuity equation and Euler equation. Here, as a simplification we are treating it as a know field (constant with time).

While diffusion equation is parabolic in nature, after including transport element it becomes hyperbolic in nature. In order to solve the problem, one defines **fluxes**, which are denoted as ϕ. In the discretization scheme we are once again defining cells, with centers and surfaces. While **u** is a vector field defined in the **center** of the cells, ϕ is defined as the $\phi_i = u_i S_i$, where u_i is a interpolated value of i-th component of the velocity, S_i is a surface of the cell. Basic idea involves using the values defined at surface instead of those defined at centers to evaluate the derivative $\frac{\partial}{\partial x}$, which makes the scheme more robust. While **u** has 3 components, ϕ has 8 defined at each of centers of the cell's surfaces. In OpenFoam there is clear distinction between the objects:

```
volVectorField U
(
IOobject
(
    "U",
    runTime.timeName(),
    mesh,
    IOobject::MUST_READ,
    IOobject::AUTO_WRITE
),
mesh
);
surfaceScalarField phi
(
IOobject
(
    "phi",
    runTime.timeName(),
    mesh,
    IOobject::NO_READ,
    IOobject::NO_WRITE
),
fvc::flux(U)
);
```

Velocity field is defined as `volVectorField` while ϕ is `surfaceScalarField`. In the declaration of ϕ we demand not to write down the field and do not read it. Whole field is defined using `flux` method which tells OpenFoam to automatically tie the definition with **u**. We also need to modify the equations in the main solver body.

```
phi = fvc::flux(U);
fvScalarMatrix TEqn
(
    fvm::ddt(T)
    +fvm::div(phi,T)
    -fvm::laplacian(DT, T)
    ==
    fvOptions(T)
);
```

Two of those options combined are enough to create a new solver for the OpenFoam. We also need to add additional term in the `system/fcSchemes` file as we need a scheme for a divergence.

```
divSchemes
{
    default none;
    div(phi,T) Gauss <keyword>
}
```

where `<keyword>` is a name of the method we would like to use for divergence computation. Simillar keywords are available as with the gradient as divergence calculation can be essentially reduced to the gradient computation.

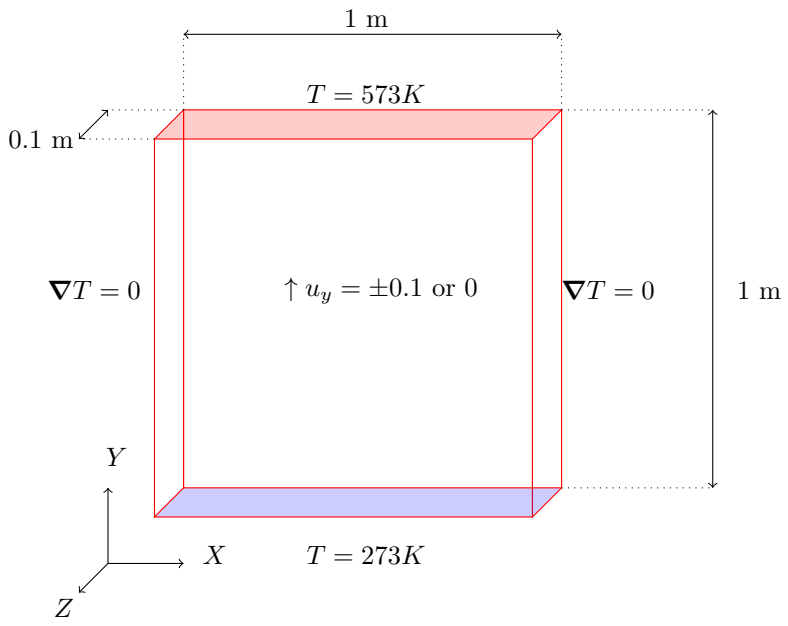

Figure 2: Diagram of the block used in the simulations

3 OpenFoam simulations

This whole section will contain various tests and cases, some with analitical solutions.

3.1 Diffusion equation + Transport

Let's start with geometry of block presented in Figure 2. Upper side of the block is kept at 573 K while lower part is kept at 273 K. We also include "wind" in the Y direction that will slightly modify our solution. We assume, that the wind is uniform in whole block. Without the wind one expects steady state that

$$\nabla^2 T = 0, \tag{4}$$

which in our case gives us linear rise from lower to upper value of temperature. This image slightly changes if we include wind. We have than

$$\frac{\partial^2 T}{\partial y^2} + \frac{u_y}{D_T}\frac{\partial T}{\partial y} = 0. \tag{5}$$

This solution has slightly different form with exponential rise/fall, with characteristic length $\lambda = \frac{D_T}{u_y}$.

Three separate cases are considered in total: no wind case and two cases with moderate velocity up and down the hill. In our simulations we assumed $D_T = 0.01$ and $u_y = \pm 0.1$ in wind cases. In order to simulate cases PBiCGStab solver with DILU preconditioner were used. All numerical schemes used simple linear interpolations together with Euler time scheme. Comparison between numerical and analitical solutions is presented in Figure 3. Paraview visulization of upwind case is presented in Figure 4. In case of zero velocity we are recovering simple linear form. In case of positive value of flow we are getting very steep temperature profile as the velocity "blocks" flow of the temperature. On the other hand if velocity is in other direction velocity is "spreading" temperature. In all cases we are getting around 0.5 K error despite only using $N = 51$ cells in Y direction.

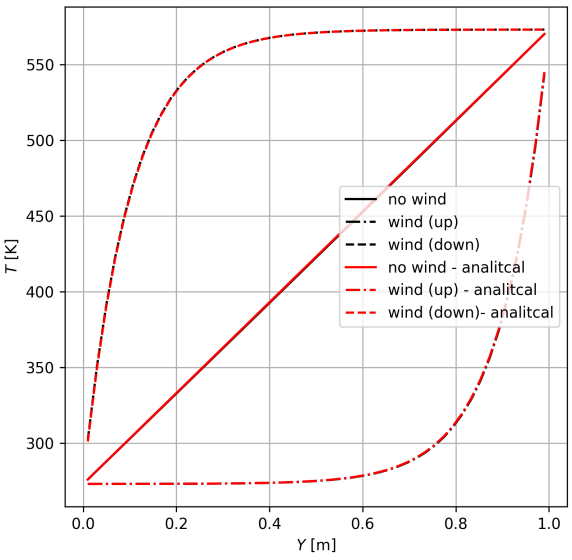

Figure 3: Comparison of velocity profile with and without velocity field (up stands for the positive u_y).

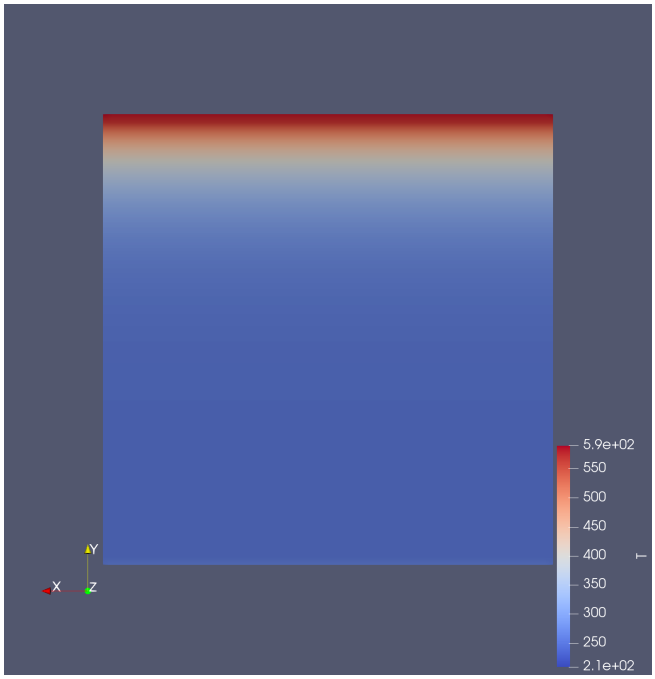

Figure 4: Temperature profile in the Y direction after reaching steady state with nonzero upflow velocity.

3.2 Couette flow

Now let's move to the Coutte flow in pipe. Here, a viscous laminar flow will be considered with incompressible solver called `icoFoam`. There are two variables $\mathbf{u}$ and p and two equations

$$\boldsymbol{\nabla} \cdot \mathbf{u} = 0, \tag{6}$$

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \boldsymbol{\nabla})\mathbf{u} = \nu \nabla^2 \mathbf{u} - \boldsymbol{\nabla} p. \tag{7}$$

$$\tag{8}$$

Here, pressure p represents in fact pressure divided by the density (pressure density). Incompressible flow is a good approximation as long as considered velocities are smaller than local speed of sound, which is important to keep in mind. We are considering flow in pipe, which is purely in Z direction. Moreover, we are assuming flow is symmetrical and hence depends only on r variable. It's possible to prove in such case, that velocity profile inside pip should be parabolic. Here is where the things get a little bit tricky. Usually in such cases we are assuming, that we have translational symmetry. This usually can be realized in case of very long system. Here, we can employ a different approach. OpenFoam provides us with a **cyclic** boundary condition. Those allow for realizing various for of translational boundaries. First we need to start with `blockMeshDict`.

```
boundary
(
    inlet
    {
        type  cyclic;
        neighbourPatch outlet;
        faces
        (
            <faces>
        );
    }
    outlet
    {
```

```
            type  cyclic;
            neighbourPatch inlet;
            faces
            (
                <faces>
            );
        }
    <pipe>
    )
```

We are writing down a different type of boundary called `cyclic`. We also need to specify boundary condition in our fields, here in case of velocity **u**

```
boundaryField
{
    pipe
    {
        type noSlip;
    }
    inlet
    {
        type cyclic;
        neighbourPatch outlet;
        transform translational;
    }
    outlet
    {
        type cyclic;
        neighbourPatch inlet;
        transform translational;
    }
}
```

This guaranties that our case will have translational symmetry and hence we do not need a long pipe.

`icoFoam` solvers is a little bit more elaborate than laplacian solver and detailed description of it is beyond the scope of the report. As it is one of most popular solvers available, we won't delve into the details of `fvSchemes` and `fcSolution` files. Unfortunately fluid simulations are quite slow compared to the simple laplacian solver so simulations were accelerated using OpenFoam MPI interface. In simulations viscosity $\nu = 0.01$ was used with initial uniform velocity profile, with velocity $u_z = 0.1$ m/s. After 400 s final velocity profile was plotted and is presented with Paraview block rendering in Figure 5 and Figure 6. As one can clearly see velocity profile is parabolic as one expects from the theoretical point of view.

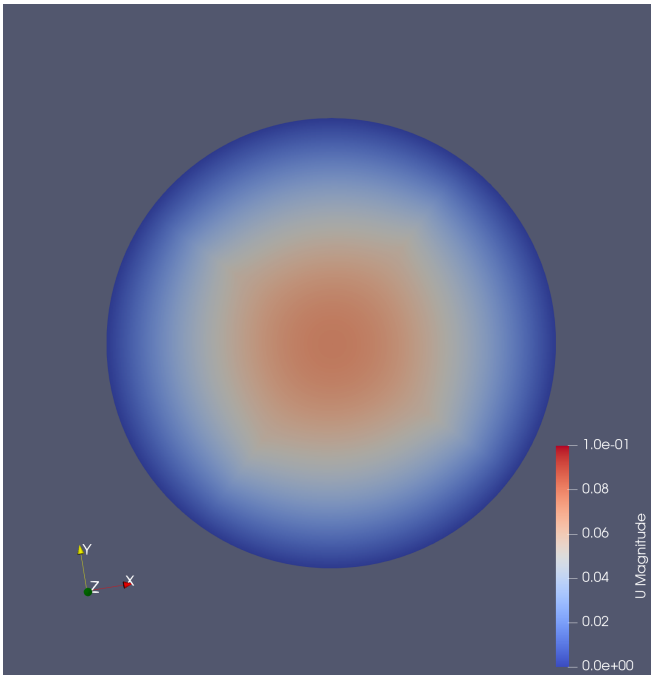

Figure 5: Paraview rendering of velocity in block.

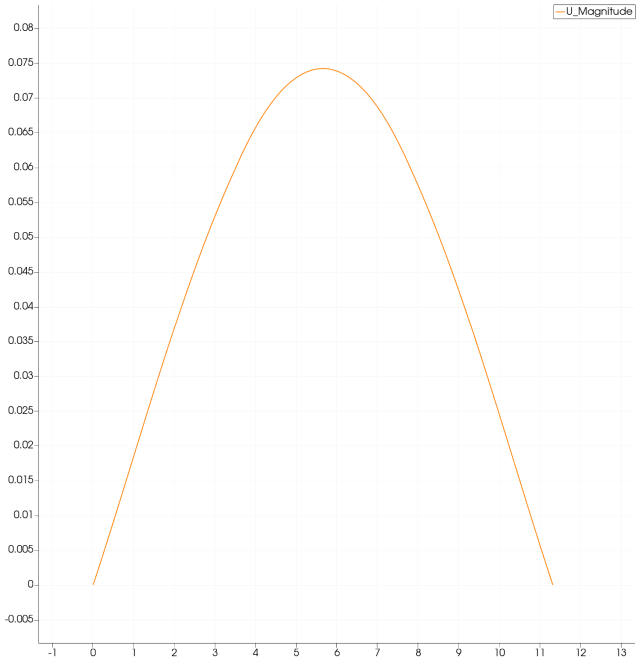

Figure 6: Velocity profile inside the pipe.