An Algorithm Report

Potential Flow: Meshing and Solver algorithm

Author
Nejc Vovk
nejc.vovk@student.um.si

Mentor
prof. dr. Jure Ravnik
jure.ravnik@um.si

Maribor, 24.02.2022

Introduction

The following report shows basic logic behind the meshing and solver algorithm for a program that solves the potential flow problem in two dimensions. The program includes a simple post-processing interface using *ParaView*, which is required to be installed. The work was inspired by an article [1]. The initial version works with *ParaView* 5.10.0.

1 Meshing Algorithm

Meshing algorithm creates points which will be used by solver to create the coefficient matrix with corresponding boundary conditions. Output of the meshing program is a text file with a **.msh** file extension. Generated file is later to be read by the solver.

1.1 Uniform Grid Generation

First, **calculateMesh()** method creates a simple uniform grid of points, spaced by the amount, specified in numerical parameters, "Mesh step". Grid size is determined by a scaling factor D, specified in "Mesh extents", with the grid aspect ratio being $1 : 2$. The uniform grid coordinates are written to **dstPike** dataset, into **dtPike** data table.

1.2 Geometry Profile Generation

Second, **NACAkoordinate()** method is activated to create points that make up geometry surface (NACA airfoil or cylinder). NACA airfoil shape is input by user as a 4-digit standardized series. The algorithm then calculates the surface points according to equations, described in [2]. For cylinder, geometry surface points are given by circle equation. Spacing between points on geometry surface is 100 times smaller than "Mesh extents", that is to ensure stability when removing mesh points from inside of geometry (discussed in later section). For NACA airfoil, camberline, thickness distribution, upper and lower surface points are written to corresponding data tables in **dstPike** dataset. Upper and lower surface points are also written to **dtPike** data table, as they present the boundary of our geometry. When **NACAkoordinate()** method is finished executing, we are left with geometry profile on top of previously generated uniform grid, as shown in Fig. 1. The next method, **drawMesh()** only takes points from **dtPike** and displays them. Figure also shows coordinate system placement used throughout mesh generation. Point index numbering, starting with 1, begins in the top left corner of the grid.

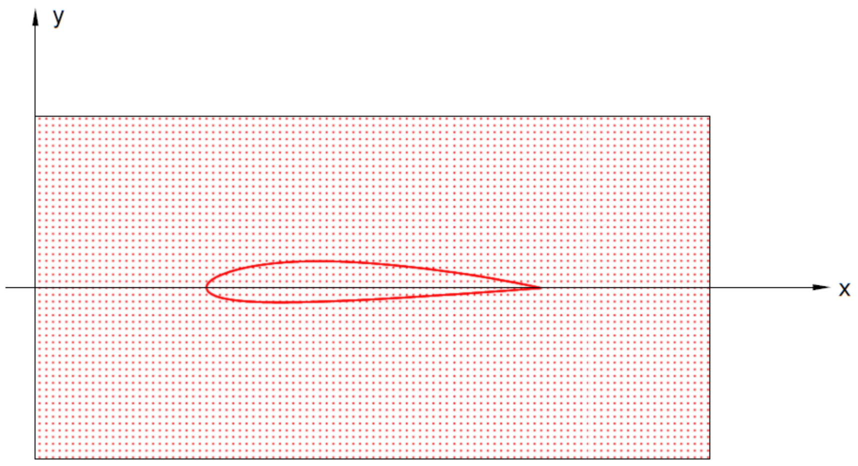


Figure 1: State after **NACAkoordiante()** is finished executing

1.3 Deleting Internal Points

Next step is to delete points that are positioned inside of our geometry. By delete we mean convert the coordinates of a point inside of geometry surface to **Double.Nan** and hence prevent them from showing up on our grid. That is achieved by method called **DeleteInner-Points1()**, the logic behind it is as follows: the algorithm first finds the grid point which is located at the leading edge of an airfoil or leftmost point on a circle if cylinder is selected[1]. It then moves along geometry upper and lower surface, until x coordinate of first grid point inside of geometry is reached. All the points on that x coordinate are then deleted, as long as they are in bounds of geometry surface. The same goes for second grid point, third and so on (Fig. 2). At high angles of attack, the middle points (for horizontal movement) are

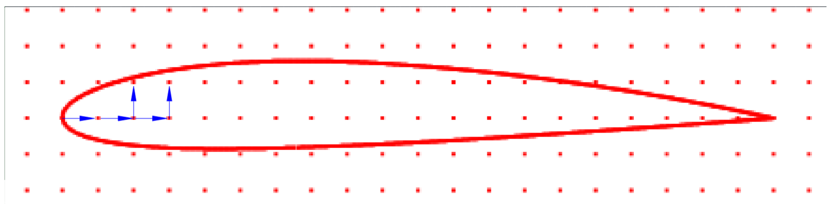


Figure 2: Deleting points inside of geometry

quickly out of bounds of geometry surface. The algorithm then moves downwards, detects upper surface and deletes points inside, until lower surface is reached, as shown in Fig. 3.

[1]Mesh step must be of such value, that a row with coordinate $y = 0$ does exist on the grid, otherwise the "mesh step not valid" error is thrown. This is a flaw of the initial version of the program and the meshing algorithm should be adapted in the future

Geometry surface points are then also deleted (converted to **Double.NaN**) and completely

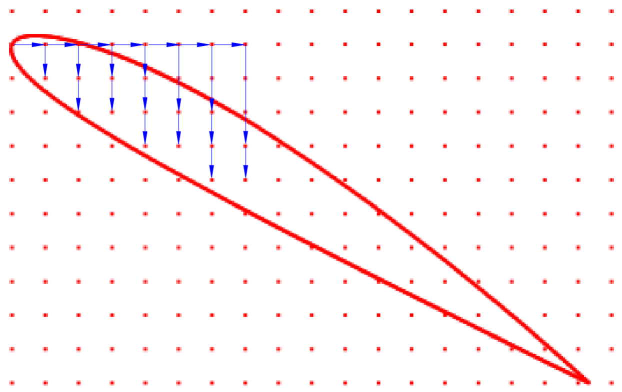


Figure 3: Deleting points inside of geometry at high angles of attack

removed from **dtPike** data table, which gives us Fig. 4. Finally, we need to mark the points

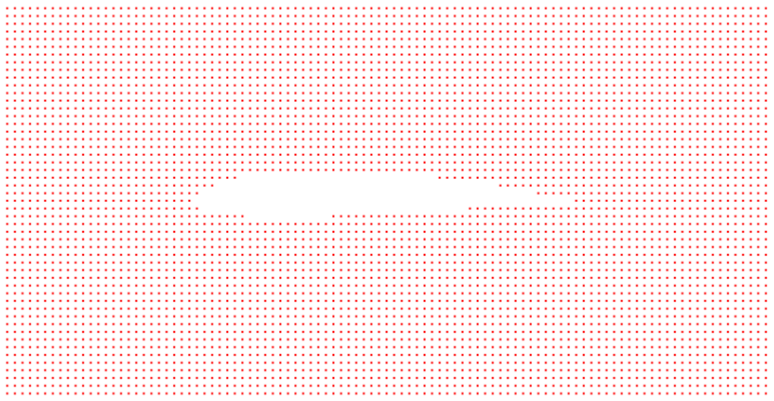


Figure 4: State after removing internal and surface points

on the grid that make up the actual surface of our airfoil, as we will later need them when determining boundary conditions. Marked surface points are stored to **dtSurfPointIndex** data table. The process of mesh generation is still not finished, as at this point, we still have our deleted internal grid points included in **dtPike** data table. Moreover, there is still no way of knowing which points make up boundaries of our domain.

1.4 Point Classification for Boundary Conditions

Next method, **SetDGV()**, has two main objectives: determine for each point, whether it is the inlet of the domain, wall, outlet or airfoil/cylinder surface. Such classification will help us set Dirichlet and Neumann boundary conditions. The algorithm takes the points from **meshChart** chart. These are uniform domain grid points, including **Double.NaN** grid points, that were converted using algorithm, explained in 1.3. Current algorithm takes each point and assigns them names, corresponding to boundary conditions, that will later be prescribed. After a point is assigned a name, it is written to **dtFinalMesh** data table. Fig. 5 shows position of each point on the mesh, with respect to its name.

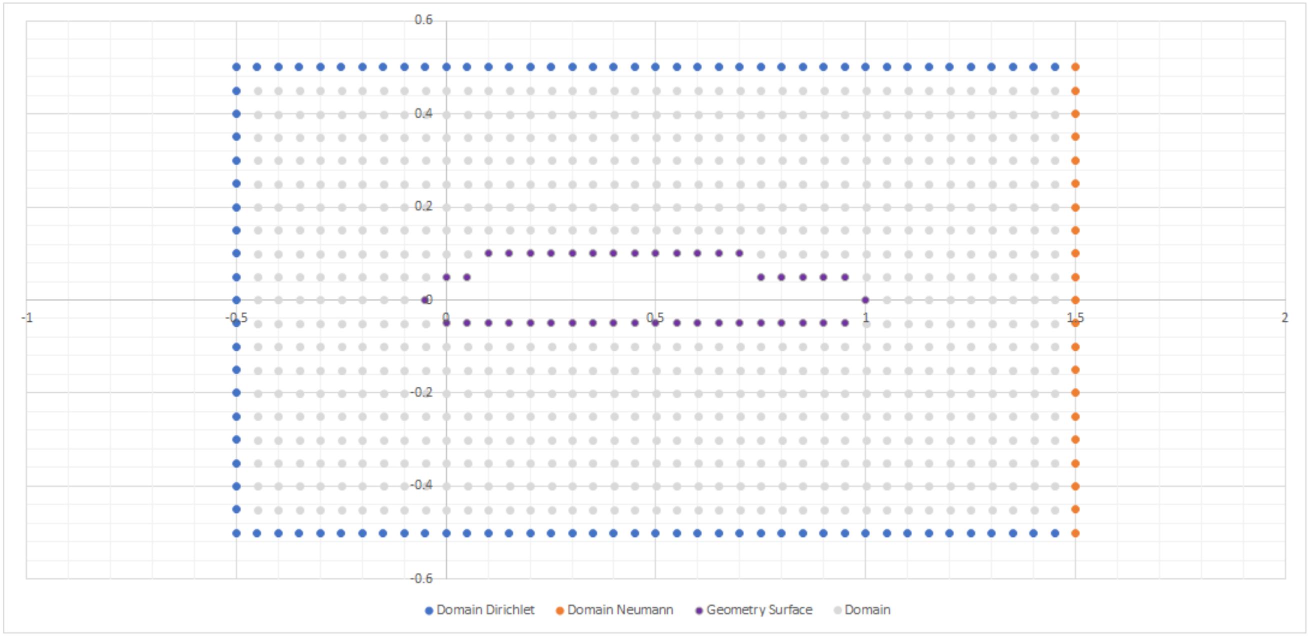


Figure 5: Points with assigned names

1.5 Connectivity

When all points with assigned names are collected to one single data table, a method called **CreateConnectivity()** creates a new data table, containing information about neighbouring points of each point in mesh. Such table will be used by solver to adequately fill the coefficient matrix. Every point without a boundary condition (see Fig. 5, Domain) must have four neighbours. **dtConnectivity** data table is filled in a way, that the indices of neighbouring points are written into rows. Points that lack one or more neighbours (geometry surface or domain boundary) are assigned a value of -1 in place of a missing neighbour.

1.6 Normal Vector Generation

At last, the method called **CreateNormalVectors()** is executed in order to create unit vectors, pointing normal to geometry surface in each point of geometry surface. The logic behind it is as follows: the algorithm detects a geometry surface point and counts the amount of surface points laying on that y coordinate. That is to determine first and last point in such row. First, a tangent vector is assigned to a point, using Eq. 1.

$$\vec{t} = \frac{1}{\sqrt{\Delta x \Delta y}} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} \tag{1}$$

Applying rotation matrix gives us a normal vector, pointing perpendicular to a tangent vector. The algorithm determines which direction is that of a normal, by checking the neighbouring points, i.e. a point on the upper surface only lacks a bottom neighbour, therefore possesses a normal vector, $\vec{n} = [1 \quad 0]^T$.

2 Solver Algorithm

Solver algorithm uses the **.msh** file created by meshing program and calculates a numerical solution to a potential flow problem, with respect to chosen geometry and boundary conditions. It also includes a simple post-processing interface by running *ParaView* as a batch process [3]. Solver uses **Math.NET Numerics** package for numerical computing [4].

2.1 Importing Mesh

The algorithm reads a **.msh** file, generated by the meshing algorithm, line by line and stores the point data to **dtMeshCoordinates** data table and connectivity data to **dtConnectivity** data table.

2.2 Running the Solver

After mesh is imported, the **Run()** method is executed when user runs the solver. The **SolveSystemOfEq()** method executes the **GenerateSystemOfEq()** method which, intuitively, generates the coefficient matrix $\mathbf{A}$ and the right hand side vector $\vec{b}$.

2.2.1 Generating System of Equations

The algorithm runs a loop through **dtConnectivity** data table. It detects where the individual point is located on the grid (inlet, wall, outlet or geometry surface) and populates the row, belonging to a point in the coefficient matrix, with corresponding values. The dimension of matrix **A** and vector $\vec{b}$ corresponds to grid size and mesh step. The values and their position in the matrix **A** and $\vec{b}$ are unique for a given potential flow problem (inlet velocity, angle of attack, geometry type...).

2.2.2 Solving System of Equations

Solver uses the biconjugate gradient stabilized method [5] for solving the system of equations. Convergence criterium and maximum number of iterations is input by the user. Lastly, the resulting vector **psi** is written to **dtStreamFunction** data table[2].

2.3 Creating Velocity Field

Method called **CreateVelocityField()** takes the data from **dtStreamFunction** data table and calculates the x and y components of velocity vector in each point, using second order central difference numerical approximation for first derivative and first order left or right difference approximation for boundary points. The pressure field is generated similarly; when velocity magnitude is known in each point, relative pressure is calculated using Eq. 2.

$$p = \frac{1}{2}\rho(u_0^2 - |u|^2), \tag{2}$$

where ρ is air density, u_0 inlet velocity and u velocity magnitude in given point. The velocity and pressure data is written to **dtVelocityField** data table.

2.4 Post Processing

The results are written to a **.vtu** file which is used by *ParaView* to visualize data [6]. That is achieved by method **WriteResults()**. The method also generates cells that *ParaView* uses to interpolate data between vertices (points). The mesh primarily consists of triangular cells. If such mesh for whatever reason fails to generate (complex NACA airfoil shape etc.), square cells are generated instead, as they are simpler.

At last, *ParaView* is launched using **pvpython.exe** command in CMD. The code which *ParaView* runs as script is located in **pyCode.py** file. The process of generating images is run in the background.

[2]A big discrepancy comes from the numerical approximation of the Neumann boundary condition, $\frac{\partial\psi}{\partial n} = 0$, on geometry surface. Due to current approach to domain discretization, the normal vectors can only point in eight different directions (0°, 45°, 90°, ... , 315°) and thus cannot fully depict the complex geometry of an airfoil. A better way to prescribe such boundary condition with finite difference approximation, which will yield better results, should be implemented in the future.

References

[1] "Development of finite difference based method for potential flow simulation over aerofoils," *Kuhljevi dnevi 2021, zbornik del*, pp. 201–208, 2021.

[2] M. Hepperle, "Javafoil user's guide," *available at: www. mh-aerotools. de (accessed 17 Feb 2022)*, 2017.

[3] Executing paraview in batch. [Online]. Available: https://forgeanalytics.io/blog/executing-paraview-in-batch/

[4] Math.net numerics. [Online]. Available: https://numerics.mathdotnet.com

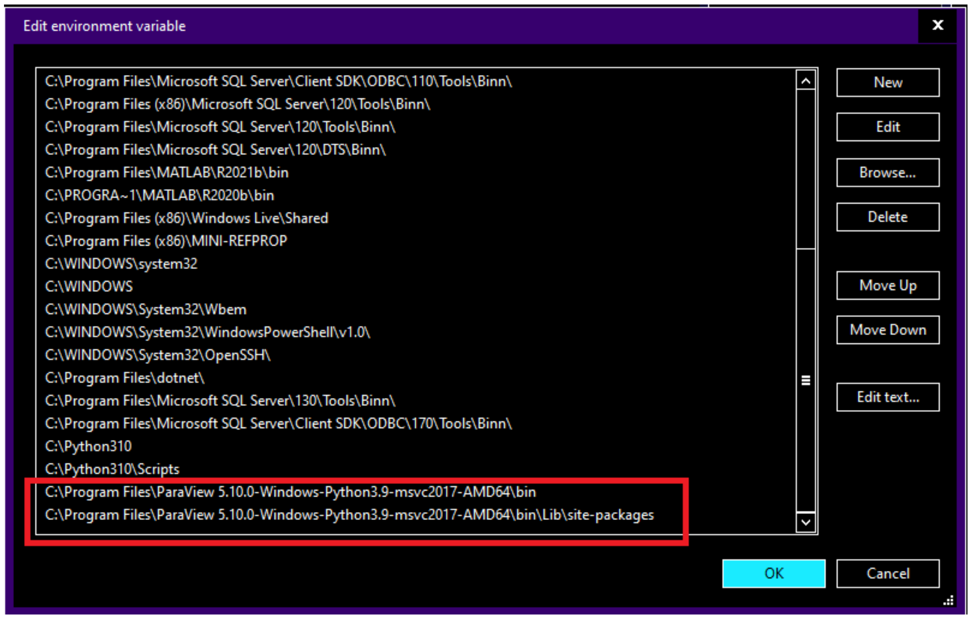
[5] H. A. Van der Vorst, "Bi-cgstab: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems," *SIAM Journal on scientific and Statistical Computing*, vol. 13, no. 2, pp. 631–644, 1992.

[6] *The VTK User's Guide.* Kitware, Inc., pp. 469–492.

Instructions

The following section will give an example on how to create a simulation of potential flow in the case of a cylinder.

Post processing currently only works with *ParaView* version 5.10.0, by running as a batch process. In order for it to function, the user must add the *ParaView* path to environment variables: in Windows settings find **Edit the System Environment Varibles**. Then click **Environment Variables...** –> under **System Variables** add two new values to the **Path** variable. The path to *ParaView* installation folder; *bin* and *site-packages* locations (i.e. *C:\Program Files\ParaView 5.10.0-Windows-Python3.9-msvc2017-AMD64\bin* and *C:\Program Files\ParaView 5.10.0-Windows-Python3.9-msvc2017-AMD64\bin\Lib\site-packages*, Fig. 6).

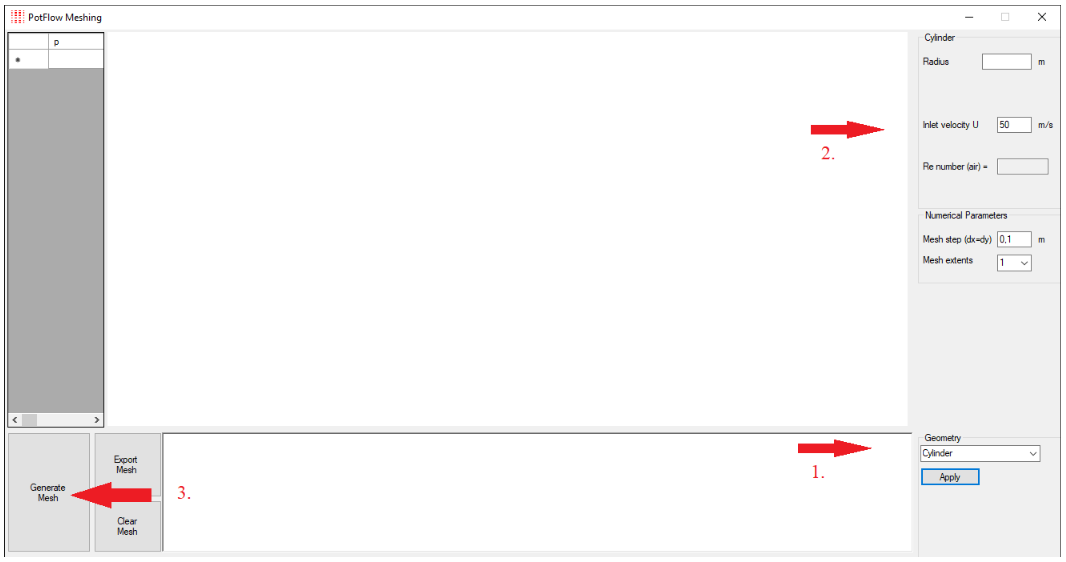


```
C:\Program Files\Microsoft SQL Server\Client SDK\ODBC\110\Tools\Binn\
C:\Program Files (x86)\Microsoft SQL Server\120\Tools\Binn\
C:\Program Files\Microsoft SQL Server\120\Tools\Binn\
C:\Program Files\Microsoft SQL Server\120\DTS\Binn\
C:\Program Files\MATLAB\R2021b\bin
C:\PROGRA~1\MATLAB\R2020b\bin
C:\Program Files (x86)\Windows Live\Shared
C:\Program Files (x86)\MINI-REFPROP
C:\WINDOWS\system32
C:\WINDOWS
C:\WINDOWS\System32\Wbem
C:\WINDOWS\System32\WindowsPowerShell\v1.0\
C:\WINDOWS\System32\OpenSSH\
C:\Program Files\dotnet\
C:\Program Files\Microsoft SQL Server\130\Tools\Binn\
C:\Program Files\Microsoft SQL Server\Client SDK\ODBC\170\Tools\Binn\
C:\Python310
C:\Python310\Scripts
C:\Program Files\ParaView 5.10.0-Windows-Python3.9-msvc2017-AMD64\bin
C:\Program Files\ParaView 5.10.0-Windows-Python3.9-msvc2017-AMD64\bin\Lib\site-packages
```

Figure 6: Added *ParaView* paths in Windows environment variables

First, the user must create a mesh using **potFlow Meshing.exe** as shown in Fig. 7. From the drop down menu in bottom right corner choose geometry type "cylinder" and click apply. Enter radius of the cylinder and inlet velocity. The user must also specify the coarseness of mesh and domain size ("scale factor"). Click "Generate Mesh" to start mesh generation process[3]. The box at the bottom shows mesh generation progress and basic mesh information.

[3]If a large or a very fine mesh is being generated, the Windows program will likely display the *Not Responding* message. That is because the algorithm runs on the UI thread of the program. In the future, the code should be modified to run on another thread, in the background. The user should ignore the *Not*



Inlet velocity U 50 m/s

Re number (air) =

Mesh step (dx=dy) 0,1 m

Mesh extents 1

Geometry: Cylinder

Figure 7: Generate mesh

Successful mesh generation yields Fig. 8. The view on the left shows numbered points and their coordinates. In order to use the mesh in solver, the user must export it by clicking "Export Mesh". This allows the user to save the mesh to an arbitrary location, when satisfied[4]. By clicking "Clear Mesh", the mesh and its info gets deleted and the user can create a new one.

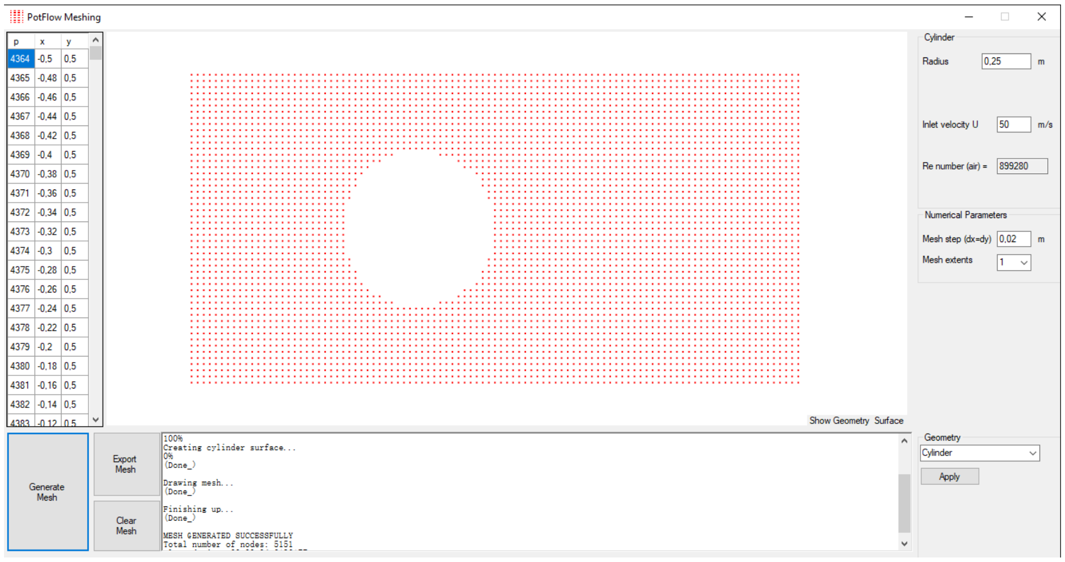
Next, run **potFlow Solver.exe** and click "Import Mesh". From the open file dialog window, choose the **.msh** file from location the mesh has previously been saved to. When mesh is done importing, specify parameters for numerical simulation, then click "Start Run" (Fig. 9). At the end of the run, save file dialog will appear. The resulting **.vtu** file must be saved to the same folder as mesh[5]. The top text box will provide basic information about geometry and mesh. Bottom text box will display info about the current run and whether it has converged or not. The user can display results by clicking wanted results on the right (Fig. 10).

If further, more complex post processing is needed, the user can open the resulting **.vtu** file in *ParaView*, from previously specified directory.

Responding message, as the algorithm is indeed running.

[4]The location on user's hard drive must only include letters of the English alphabet and must exclude spaces.

[5]Most Windows versions will automatically recommend that folder, as the path was remembered by saving the mesh file. Regardless of that, the user should check if **.vtu** file is being saved to a correct folder and change if necessary.



p	x	y
4364	-0,5	0,5
4365	-0,48	0,5
4366	-0,46	0,5
4367	-0,44	0,5
4368	-0,42	0,5
4369	-0,4	0,5
4370	-0,38	0,5
4371	-0,36	0,5
4372	-0,34	0,5
4373	-0,32	0,5
4374	-0,3	0,5
4375	-0,28	0,5
4376	-0,26	0,5
4377	-0,24	0,5
4378	-0,22	0,5
4379	-0,2	0,5
4380	-0,18	0,5
4381	-0,16	0,5
4382	-0,14	0,5
4383	-0,12	0,5

Cylinder
Radius 0,25 m
Inlet velocity U 50 m/s
Re number (air) = 899280

Numerical Parameters
Mesh step (dx=dy) 0,02 m
Mesh extents 1

Geometry
Cylinder

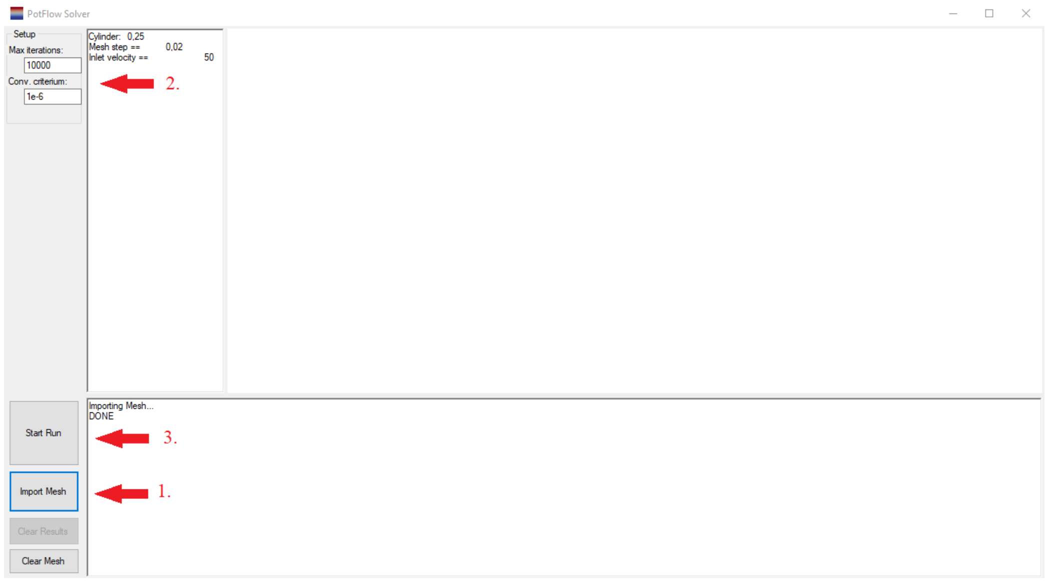
```
100%
Creating cylinder surface...
0%
(Done_)

Drawing mesh...
(Done_)

Finishing up...
(Done_)

MESH GENERATED SUCCESSFULLY
Total number of nodes: 5151
```

Figure 8: Mesh for a cylinder

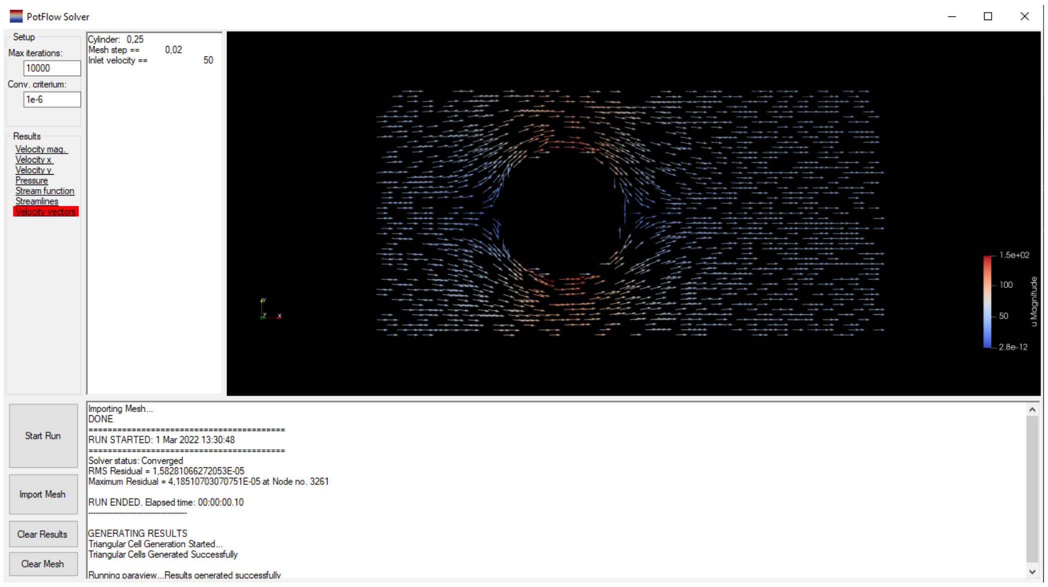


Setup
Max iterations: 10000
Conv. criterium: 1e-6

```
Cylinder:  0,25
Mesh step ==     0,02
Inlet velocity ==      50
```

```
Importing Mesh...
DONE
```

Figure 9: Pre-processing



Max iterations: 10000
Conv. criterium: 1e-6

```
Cylinder:  0,25
Mesh step ==        0,02
Inlet velocity ==              50
```

```
Importing Mesh...
DONE
==============================================
RUN STARTED: 1 Mar 2022 13:30:48
==============================================
Solver status: Converged
RMS Residual = 1,58281066272053E-05
Maximum Residual = 4,18510703070751E-05 at Node no. 3261

RUN ENDED. Elapsed time: 00:00:00.10
----------------------------------------------


GENERATING RESULTS
Triangular Cell Generation Started...
Triangular Cells Generated Successfully

Running paraview...Results generated successfully
```

Figure 10: Post-processing