pipemesh: Preprocessing Pipe Network Generation for Use in Pipe Flow Analyses

D. Hunter

Imperial College London, South Kensington, London SW7 2AZ

Abstract

An object oriented Python package for generating pipe and pipe network meshes, *pipemesh*, has been created. The user can can create pipes in a sequential, modular fashion starting from an inlet cylinder, with individual fittings represented by Python objects, to generate a mesh with the GMSH-API. These meshes integrate with the IC-FERST pipe simulation modelling workflow, by replacing the conventional method of using GMSH. Time is saved through reducing the number of commands needed, and by providing an understandable, easy to use workflow. Generation of coarse meshes is quick, and advantageous when using software with mesh-adaptivity.

Keywords: Mesh generation, CFD, Pipe flow

Course Name	MSc Applied Computational Science and Engineering
Github Alias	Duncan-Hunter
CID	01104228
Email	duncan.hunter15@imperial.ac.uk
Date	30/08/2019
Supervisors	Dr. A. Obeysekara, Prof. Chris Pain, Dr. Andre Nicolle (BP)
Project Title	Integration of CFD Modelling Framework IC-FERST for Industrial Application in BP: Automation of pre- and post-processing using Python (Projects with IC-FERST/BP Collaboration)
Software code languages, tools, and services used & dependencies	Python, SciPy, NumPy, GMSH, IC-FERST
Link to Repository	https://github.com/msc-acse/acse-9-independent-research-project-Duncan-Hunter

Table 1: Project Metadata

1. Motivation and significance

Turbulent pipe flow in varying geometries is often studied, as transporting fluids is a key part of many industrial processes. In such processes, pipes will often form complex networks, including bends and junctions. To simulate such networks with computational fluid dynamics (CFD) software, a mesh is often required. Generating meshes for pipe flow simulations with a script rather than a graphical user interface (GUI) is a time-consuming and complex task. In an industrial or scientfic setting, this can act as a barrier for research. While using a GUI may increase usability, it doesn't facilitate scriptable generation of pipe networks with varying parameters.

IC-FERST [1, 2, 3, 4] is an open source, reservoir simulator CFD code that uses control-volume finite element methods with mesh-adaptivity. It can be used to solve turbulent flow cases, as done in [5]. It uses the .msh file format for meshes, created by GMSH [6] compiling .geo files. Creating complex .geo files conventionally involves placing a series of points to create lines and then surfaces, and then extruding surfaces to create volumes. If OpenCASCADE [7] features (discussed below) are used, individual specific volumes are directly created and fused together. In either case, the user has to manually track the locations, directions and other geometrical information of many of the objects in the model. This can lead to errors, and an iterative process of editing and compiling a .geo file several times before a correct geometry is created. Furthermore, learning the basics of GMSH can be a time-consuming task in itself. This reliance on GMSH can therefore act as a barrier for using IC-FERST for pipe network flow simulations, despite the advantages of using IC-FERST compared to commercial CFD codes. Mesh-adaptivity, not commonly found in commercial software, is the process of refining and coarsening the mesh used for simulations using a metric, which for IC-FERST is calculated from the Hessian of chosen properties in the simulation [1]. Adapting the mesh during a simulation can save computational resources, as the mesh in highly turbulent areas can be refined to gain resolution, and areas of simple flow can be coarsened. This also has the benefit of not needing to know *a-priori* the locations of computational complexity. Finally, it allows coarse meshes to be generated quickly and then refined, rather than the time-consuming process of generating a refined mesh.

This software aims to facilitate single and batch creation of complex pipe networks from Python scripts, to allow users to vary parameters of mesh geometries easily; this in turn can allow more investigations on pipe network flow, and more time spent on analysis of results. The software is designed to be used with IC-FERST, but should be applicable to any software using .msh files. As it will use scripts, this will allow direct creation of meshes on high performance computers (HPCs), which often do not have access to GUIs. Generating meshes on HPCs allows greater memory usage for very large meshes, as well as reducing time used in transferring files from workstations.

This report presents *pipemesh*, an object oriented Python package designed to facilitate sequential, modular, and batch creation of pipe network meshes for CFD simulations through Python scripts and the GMSH-API. The software development process, architecture, and implementation (including testing) is explained. Its impact in saving time in mesh creation and ability to create designs is discussed, with basic simulation results from multiple configurations presented to highlight its integration with IC-FERST. While a number of generic, open source meshing softwares are available (discussed with the GMSH-API below), as of the time of this report, there is no obvious software that creates pipe network meshes for simulating fluid flow.

2. Software description

As a pipe can be considered a sequence of fittings joined together, the software was designed so the user could add a fitting (represented by a Python class) to a pipe with a single command, and procedurally create a design, as in Figure 1. The code then integrates into the beginning of the IC-FERST modelling workflow for pipe networks. Mesh generation and MPML file configu-

ration can both be done with *pipemesh*.

2.1. Requirements

The requirements for *pipemesh* include being able to use scripting to generate batches of meshes; to be able to be run on an HPC through executable scripts; to be easy to learn and use; and to be able to create specific designs with common fittings. Further requirements include being able to set individual mesh sizes, for use with software without mesh-adaptivity. For use with IC-FERST, the software needs to be able to set physical groups, to define boundary and initial conditions for simulations [8].

Requirements for different pipe fittings can be derived from observing pipe networks, as well as from scientific analyses of different fittings. There are many published studies simulating turbulent flow in individual sections of pipes, yet simulations of large networks are not well published. One example of a network with a bend and T-junction being simulated is by [9], examining how in-pipe wedges reduce drag.

The simplest problem is that of turbulent flow in a straight cylinder, which has been studied often [10, 11, 12, 13], with experiments investigating the onset of turbulent flow at varying Reynolds numbers. The Reynolds number for pipe flow [14] is determined with:

$$Re = \frac{D\rho u}{\mu}$$

where Re is the Reynolds number, D is the pipe diameter, ρ is the fluid density, u is velocity, and μ is the fluid's kinematic viscosity. Density, viscosity and velocity are all parameters that can be edited in the MPML file for IC-FERST, and pipe diameters can be varied with *pipemesh*.

Pipe bends have been investigated by [15, 16, 17, 18, 19, 20]. Swirl switching is the oscillation of Dean vortices [21], and is one of the problems studied in pipe bends, as it can lead to mechanical fatigue from vibrations [20]. The ratio of pipe radius to bend radius (shown in Figure 2) is used to describe the geometry of a bend, and has been studied in relation to swirl switching. This highlights how bends have to be specified by their centreline radius (shown in Figure 2). Sharp "mitered" bends are less frequently simulated. This may be due to the difficulty in simulating high bend ratios, lack of use in industrial settings, or difficulty in creating the geometry. Physical experiments by [15] showed a "similar 'switching' helical flow", as well as an inner-wall separation. [22] investigated flow in helically coiled pipes. Pipes join other pipes in T-junctions. These are often simulated to analyse the mixing of fluids, and the exchange of heat, especially in nuclear power applications [23, 24, 25, 26]. These studies are all conducted with the joining pipe perpendicular to the main pipe, as this is a common configuration. Pipe reducers, where the pipe radius is reduced and flow compressed, are less simulated, but are present in many pipe networks, and provide a useful tool in joining pipes of different radii. The ability to join pipes together is crucial, and the ability to have a different joining radius to the main pipe is also a requirement. Additionally, the ability to have pipes at different joining angles could be useful for investigations into its effect.

From these studies, five fittings (with their respective *Piece* name) were identified, cylinders (*Cylinder*), curves (*Curve*), mitered bends (*Mitered*), pipe reducers (*ChangeRadius*) and T-junctions (*TJunction*). Users should be able to specify the radius of the pipe, the direction it is facing, centre-line radii of bends, bend angles, T-junction angles, and T-junction joining radii.

2.2. Software Development Process

Python was chosen for its increasingly common use [27], and ability to use object oriented programming. Both pipe fittings and networks can be considered as objects, with their own geometrical information and class methods allowing the user to not track their information.

The software is developed using the functionalities provided by the GMSH-API. This is a tool developed to allow the creation of GMSH scripts in Python, C++ or Julia, instead of .geo files. It was chosen ahead of SALOME [28], a primarily GUI based meshing tool, which has Python scripting capabilities. While SALOME has complex

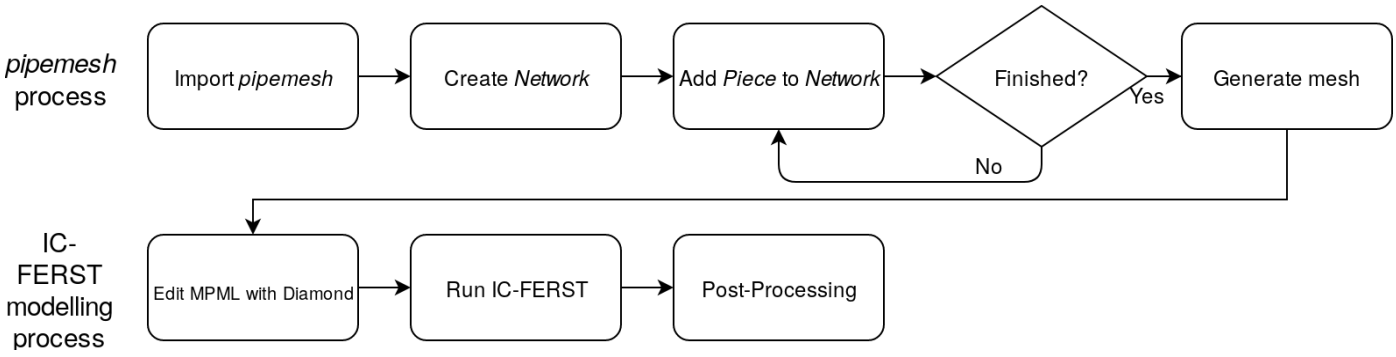


Figure 1: User process of *pipemesh*. Modular *Pieces* can be added sequentially until a design is complete. Once generated, the mesh can then be used with IC-FERST. Diamond is installed with IC-FERST and is used to change simulation settings in MPML files.

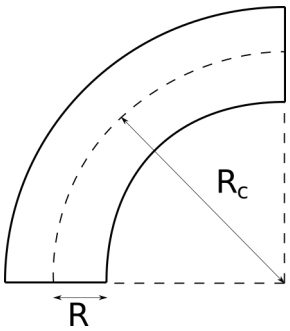


Figure 2: Bend geometry. R_c is the centreline radius. R is the pipe radius. The bend ratio is defined as R/R_c

premade pipe fittings to create in one command, it cannot export meshes to the .msh file type used by IC-FERST, or create the physical groups for boundary conditions, thus requiring a conversion tool to be developed before integration into the IC-FERST workflow. Both SALOME and GMSH have OpenCASCADE [7] features, which allows the direct creation of 3D volumes instead of placing points to turn into lines, surfaces and volumes; and boolean operations such as fuse and intersect. These respectively join volumes together and create volumes from where separate objects intersect. These are useful, as it allows individual, complex pipe fittings to be created and joined together, rather than a continuous extrusion process which restricts pieces to simple cylinders and bends. As this software is being developed in Python and for IC-FERST, the GMSH-API was chosen as the best software to base the project on.

This project used Sphinx Autodoc to automatically generate documentation, which is hosted on readthedocs[29]. Code formatting was aided by the use of yapf and pylint. Testing was performed by running a set of Python scripts to test the generation of meshes, with details included below. The use of the online integration tool TravisCI was not used, as the GMSH-API requires pre-compiled binaries that couldn't be located. The development methodology used was the Waterfall approach. The requirements above were gathered, and tools such as the GMSH-API were investigated for their feasibility. The software was designed, implemented and maintained, with additional functionality being integrated once the basic software was in place. Progress was monitored using a Kanban on Github's Project feature.

Installation uses the simple method of pip install (Listing 1), followed by downloading and copying the precompiled binary files from the GMSH-API into the site-packages directory of the Python installation. Alternatively, the user can clone the repository and add the package to their Python path. The requirements for the software are Python 3.5+, and SciPy [30] (including NumPy [31]) for rotations. It is recommended that a virtual environment is used, to locate the site-packages folder more easily. The software is open source, allowing it to be taken by users and have custom objects made, for example to have a different fitting. The package is limited to use with Linux systems, which the majority of HPCs use.

```
$python3 -m pip install --user pipemesh
```

Listing 1: pip install command

2.3. Software Architecture

pipemesh consists of three modules, *pipes*, *pieces* and *auto_mpml*. *gmsh* is the GMSH-API, and is included in the repository as it is required by *pipes* and *pieces*, along with the precompiled binaries *libgmsh.so** which *gmsh* relies on. As *pipemesh* is open-source, distributing GMSH doesn't require a license. The dependencies of each module are shown in Figure 3.

A diagram of the classes in *pipes* and *pieces* is in Figure 4. *pieces* contains a base class *PipePiece*, with 5 children classes representing the individual pipe fittings the user can create. This allows the easy addition of more complex fittings if requested by users. These are called with the geometrical information needed to create them, such as directions inlets and outlets are facing, lengths, and radii. As all pipe fittings have inlets and outlets, the geometrical information for these circular surfaces are stored as *Surface* objects. *pipes* contains the user-facing *Network* class, which has methods to add children of the base *PipePiece* classes contained in *pieces*, to the *Network*, storing the *Piece's* information to be used in the addition of other *Pieces*, and to be passed to any post-processing software. The user primarily uses the *Network* object, but can directly create *Pieces* if they use GMSH. The *Network* class and *PipePiec* classes take user inputs, and translate them into GMSH-API commands, as well as performing geometrical calculations. This layer between the user and the GMSH-API removes the need to learn how to use GMSH.

The AutoMPML class is in the 'icferst/' directory, as it will be used to set up simulations for IC-FERST. With this structure, modules for supporting other CFD softwares can be created in separate directories, with the *Network* object remaining as a common tool.

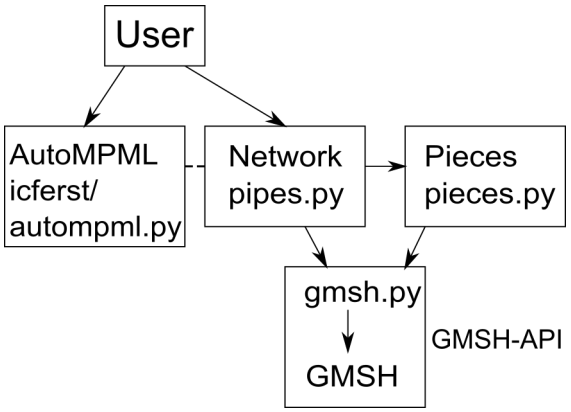


Figure 3: Dependencies of *pipemesh* modules. *pipes* depends on *pieces* and gmsh, pieces depends solely on *gmsh*. *AutoMPML* uses information obtained from the *Network* object, but can be used without accessing this information from within a script.

2.4. Functionalities and Implementation

2.4.1. GMSH-API

The GMSH-API uses *dimtags* to identify entities in GMSH models. These are tuples of dimension and a unique tag for that object in that dimension, and are assigned on creation. For example, the first volume in a model will have *dimtag (3, 1)*. By using OpenCASCADE, volumes can be created directly, with surface *dimtags* being acquired through the GMSH-API function *getBoundary* For different *Pieces*, the surface *dimtags* of the volume are assigned in a separate, consistent manner. For example, the surface *dimtags* of a cylinder will always be assigned in order of the cylindrical surface, the top surface, and then the bottom surface, which is different to the order of a smooth bend. After affine transformations, the surface *dimtags* may be assigned to a higher number (if another object is present), yet the *dimtag* of the volume remains the same, and so the consisted order of assignment of surface *dimtags* allows the stored surface tag to be updated accurately. Tracking *dimtags* is important, due to their use by the majority of GMSH-API functions. The user does not need to track these manually, as *dimtags* for inlet and outlet surfaces are stored in the *Surface* class, and the volume *dimtag* stored in the *PipePiece* class.

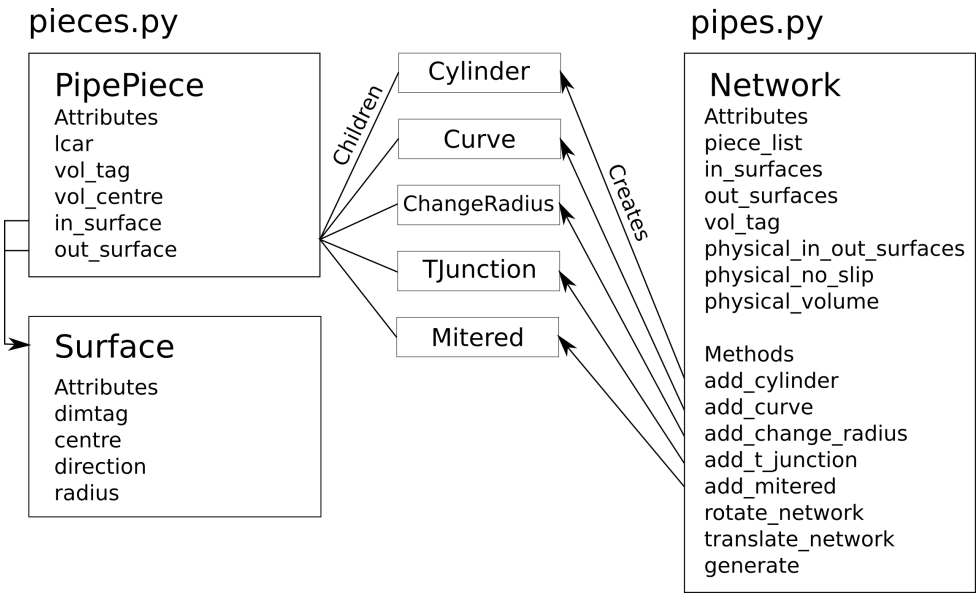


Figure 4: An overview of the classes in this project. Methods of the *Network* class create *PipePiece* objects, which are added to the *Network*. Hidden methods, are not shown.

2.4.2. PipePiece

Each *PipePiece* child has a separate init function, taking user input about the dimensions, direction and mesh size of the piece, as well as *Piece* specific properties (for example, which direction the joining pipe should face in a *TJunction*). Each *Piece* has overriding methods for updating the dimtags of the *Piece* surfaces, due to the different assignment orders for different pieces. These methods are hidden, as they are meant to be called by the *Network* class that creates the *Piece*. All pieces are created with an inlet direction (the direction the inlet should face), and a radius. *Piece* specific creation parameters for bends are the outlet direction, and for smooth bends the centreline radius. *TJunctions* require the direction that the joining pipe outlet will face. *Cylinders* require a length, with *ChangeRadius* requiring the radius to change to and the length to change it over. Conditions, such as the length has to be greater than 0, are used to check input, raising *ValueErrors* if incorrect.

All *Pieces* are created in the x plane, facing up, as shown by the cylinder in Figure 5 a), before being rotated to the direction given by the input. This has two benefits when creating bends and junctions. In GMSH the creation of bends requires revolving a disk by an angle α about a centre of rotation and axis of rotation. By limiting this to the x plane, two directions and a radius can be used to create the bend, allowing the radius of the bend to be specified for any bend. The software finds the angle α between the directions using *vec_angle* (Listing 2) and knows the centre of rotation (radius, 0, 0) and axis of rotation (y), which it can give to GMSH to create the *Piece*. A *TJunction* is created by fusing 3 cylinders together, two in a line, and one at an angle. By creating the 3 cylinders in the x plane, the angle between the T direction and pipe direction can found and used to rotate the joining cylinder.

All *Pieces* are rotated using _rotate_inlet (Figure 5 a)). This finds the angle between up (as the *Piece* is created with direction (0, 0, 1)) and the direction that the user wants it to face. It finds the the axis of rotation using the cross product of the two directions, and uses the origin for the centre of rotation, and rotates the object with GMSH. If the *Piece's* outlet direction is different to its inlet direction, it can be found with trigonometry, as it is in the x plane. The rotation applied to the inlet is applied to the outlet direction using SciPy's Rotation class, to find the new outlet direction. The outlet direction could be found with the *getNormals* function in GMSH, but this requires the mesh to be gener-

```
def vec_angle(vec1, vec2):
    """Returns the angle between two numpy array vectors"""
    return np.arccos(
        np.dot(vec1, vec2) / (np.linalg.norm(vec1) * np.linalg.norm(vec2))
        )
```

Listing 2: Angle between vectors.

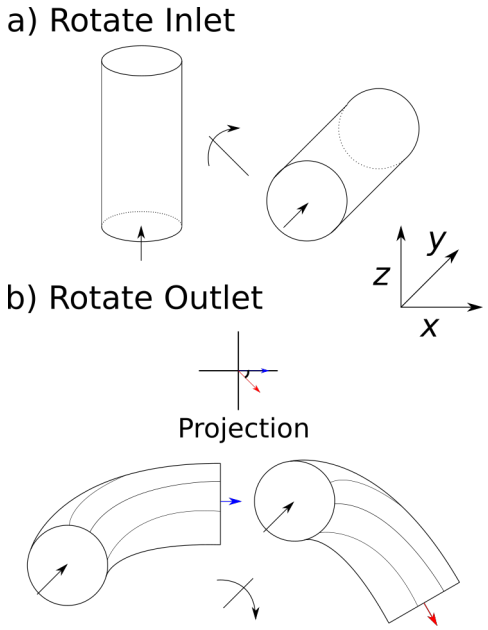


Figure 5: Piece creation. a) All pieces are created with outlets with z normal, facing up. The angle between up and the direction given is found, as well as the axis, and rotated. b) The outlet direction after the inlet rotation, and the user given outlet direction, are projected to find the angle. The object is then rotated about the inlet direction, by the angle.

ated already, and can stop the user adding further *Pieces*.

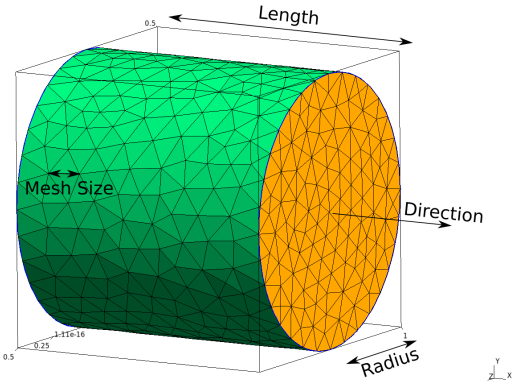
Bends and T-junctions then have to be rotated to ensure the outlets are facing the right direction (Figure 5 b)). The function *_rotate_outlet* projects the direction vector the outlet is facing, and the desired direction vector onto a plane perpendicular to the inlet direction. It then finds the angle between those projections, and uses the inlet's direction as the rotation axis to rotate the *Piece* with GMSH.

As these rotations use the origin as the centre of rotation, the post-rotation centres of mass are not always known. The OpenCASCADE feature *getCenterOfMass* and the GMSH *dimtags* (*vol_tag* is a *dimtag*) stored in the PipePiece and Surface objects, are used to find the centres of *Pieces* after rotation. This enables accurate translation of *Pieces*.

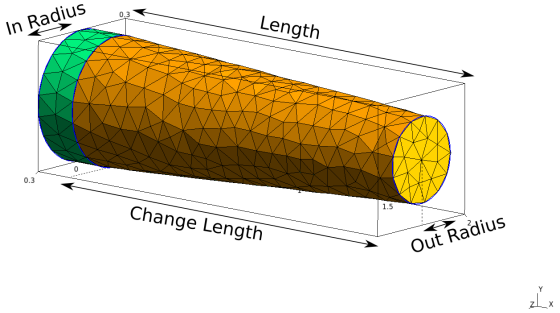
The creation of each *Piece*, during their respective *init* functions, is a different process, with some more complex than others. *Cylinders* are created with the OpenCASCADE cylinder function. The creation of smooth bends (*Curves*) is described above. A change in radius (*ChangeRadius*) is created with a cylinder, followed by a chamfer operation. *TJunctions* are created with 3 cylinders, two for the main pipe and one for the joining pipe. The *TJunction* can be created with an arbitrary joining angle. One end of the piece, and the joining pipe, have to be elongated to ensure both inlets and outlets are fully emerged. The middle joining cylinder is rotated to the correct angle, and the three cylinders fused together. Mitered pieces are created in several steps. A chamfered box is created over a cylinder, and the intersection found. This results in a cylinder with a cut face at an angle. The cylinder is then duplicated using the *symmetrize* function in GMSH. This new volume is rotated around, so as to make the cut faces join, before they are fused. Examples of the pieces, and their dimensions, are highlighted in Figure 6. The dimensions are given to allow accurate recreation of designs. For *Mitered* bends the 2.1 instead of 2 is used to ensure the chamfer operation is successful. The 1.1 in *TJunctions* is used to ensure the pipes are fully emerged from one another.

2.4.3. Network

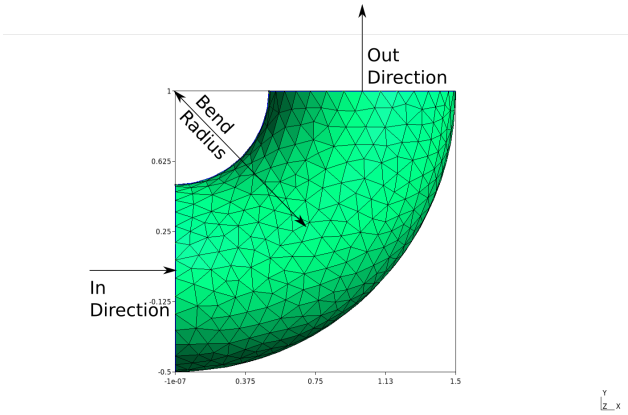
The *Network* class is intended to face the user. Its methods and respective parameters and descriptions can be found in Table 2. It is called by



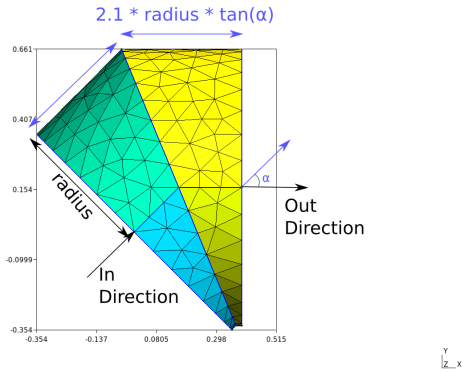
(a) *Cylinder*.



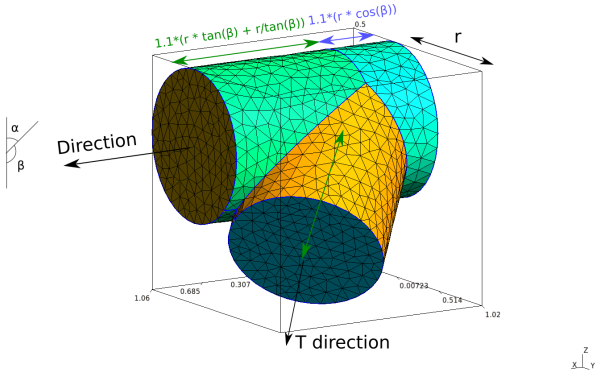
(b) *ChangeRadius*.



(c) *Curve*.



(d) *Mitered*.



(e) *TJunction*.

Figure 6: Individual *Pieces* and their dimensions. Dimensions written in black are specific parameters that are given to the *Piece*. More details can be found in the documentation.

specifying an inlet cylinder, which uses the *Cylinder* class of *pieces*. The user then selects *add_piece* methods to specify a *Piece* to add to the outlet. This *Piece* is created with the *init* of the *Piece's* class as described above, then translated so the *Piece's* inlet is at the inlet *Piece's* outlet. The new *Piece's* outlet is then made the Network's outlet, and another *Piece* can be added until the user is finished. If the user adds a *TJunction*, a *Surface* is added to *out_surfaces*, and can be added to using the out number argument of the *add_piece methods*. The default outlet has an out number of 1, with every sequential outlet incrementing this. The user can rotate or translate the *Network* using *rotate_network* or *translate_network* respectively. The user always has to add to an outlet, to limit rotations to the added *Piece* rather than an entire *Network*. There is no ability yet to add one *Network* to another *Network*. Joining two *Network* outlets can be added, but ensuring the correct relative directions of other outlets is not.

When finished, the user calls the *generate* method of *Network*. This checks if *Pieces* are crossing by using OpenCASCADE's intersection feature. Mesh sizes are enforced by creating spherical threshold fields around the centre of each *Piece*, with a radius determined by the distance between the *out_surface* of the *Piece* and the volume centre. This can lead to overlapping meshes, where the minimum mesh size is taken. This feature is important to ensure the pipe is cylindrical, as a large mesh size leads to rough, polygonal outlet surfaces. When mesh-adaptivity is not being used, it allows the user to refine areas of predicted complexity. It would be advantageous to refine the edges of the pipe with a coarse centre, but it was found that only the reverse was possible. The *Pieces* are fused to form one volume, and physical groups assigned to each of the surfaces and the volume. The user has the option to decide the mesh format; if it is in binary; to write an information file with details of the physical surfaces; and to write an xml file with the same details. The user can also choose to run the GMSH GUI, to check the geometry, although this may stop the saving of any files as closing the GUI may end the Python process.

2.4.4. AutoMPML

A utility tool, *AutoMPML*, was developed to aid in the creation of MPML files, which are used to define simulations in IC-FERST. It is scriptable rather than automatic, allowing the settings for a pipe flow simulation with IC-FERST to be edited in Python directly, rather than navigating IC-FERST's Diamond GUI. This allows the user to focus on parameters that will directly effect the simulation, and allows scriptable variation of parameters, such as varying the Reynolds number through changing inlet velocity, or which surfaces are inlets and outlets. It is a Python class, which combines with the *get_cyl_surfs*, *get_inlet_phys_ids* and *get_velocities* methods of the *Network* class. These functions return the physical IDs of surfaces, and turn inlet physical ids into a list of velocity vectors normal to the inlets given, using either a given Reynolds number or velocity magnitude.

2.4.5. Testing

Tests were run during development after changes to the code base, to ensure the code maintains functionality. This is important as the GMSH-API is currently being updated regularly. The script pytest.py can be run during development of *pipemesh*. The tests, summarised in Figure 7, include creating and saving a mesh, and loading the .msh file to check the version and number of nodes, in both binary and ASCII format (Listing 6); testing if the mesh refinement is being applied by accessing the number of nodes before and after mesh element size is enforced (Listing 7); checking the creation of all individual *Pieces* by checking the geometries (whether the normals and centres of mass are accurate) and checking rotation by comparing surface normals before and after rotation. GMSH has functions for checking the number of nodes in a model, the centres of mass of GMSH surfaces, and normals of surfaces during creation of a model. For each *Piece* tested, the normal directions and centres of mass of the inlet and outlet surfaces are accessed through both the *Piece* and in GMSH and compared. The helper functions for IC-FERST, *get_velocities* and *get_ids* are also tested here, as they are part of the *Net-*

Table 2: Network class methods, with parameters needed and description of method.

Methods	Parameters	Description
add_change_radius	*length, new_radius, change_length, lcar, out_number=0*	Adds a *Piece* that changes the radius of the outlet.
add_curve	*new_direction, bend_radius, lcar, out_number=0*	Adds a curve to the *Network* at the outlet.
add_cylinder	*length, lcar, out_number=0*	Adds a pipe to the *Network* at the outlet.
add_mitered	*new_direction, lcar, out_number=0*	Adds a mitered bend to the *Network* at the outlet.
add_t_junction	*t_direction, lcar, t_radius=-1, out_number=0*	Adds a T junction to the *Network* at the outlet.
generate	*filename=None, binary=False, mesh_format='msh2', write_info=False, write_xml=False, run_gui=False*	Generates mesh and saves if filename.
get_cyl_phys_ids		Returns a list of physical ids of cylinder surfaces.
get_inlet_outlet_phys_ids		Returns a list of physical ids of inlets.
get_velocities_reynolds	*hysical_ids, reynolds_no, density, viscosity*	Creates velocity vectors for inlets using reynolds number.
get_velocities_vel_mag	*physical_ids, velocity_magnitude*	Returns velocity vectors for inlets using velocity magnitude.
rotate_network	*axis, angle*	Rotates the *Network* by angle about axis.
translate_network	*vector*	Translates a *Network* by vector.

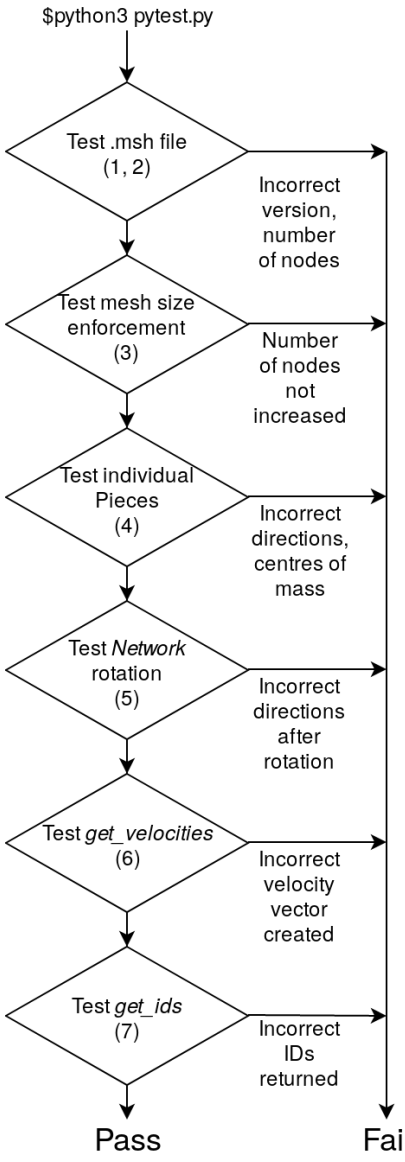


Figure 7: *pytest* process. Numbers in brackets refer to the test number in *pytest*, displayed in the Appendix as test1, test3 etc, and in the file pytest.py.

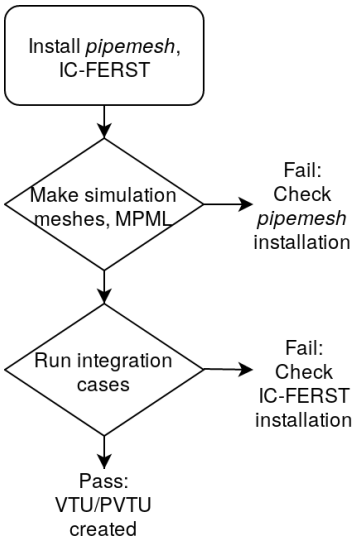


Figure 8: Integration test case workflow.

work class.

Further tests were created for *AutoMPML*, which involve changing different options, and seeing if the resulting MPML file has changed those options.

To check integration into the IC-FERST workflow, a set of directories containing a Makefile and source directory were created for each *Piece*. Figure 8 shows how these can be used to check the installation of both *pipemesh* and IC-FERST. By calling Make in a directory, *pipemesh* will generate a mesh, and the *AutoMPML* class will generate an IC-FERST settings file, shown in Listing 12 (Appendix). If Make fails to generate said files, the user can identify an issue with their installation of *pipemesh*. The simulations have one inlet with a velocity of 0.02 to ensure a low Reynolds number (for water), minimising turbulence in the solution and decreasing wall time. The test case simulations are set to end after 1 second of simulated time, and to adapt the mesh with a relatively large minimum mesh size (0.08, 0.02 smaller than created). This allows testing in a short wall time and tests if mesh-adaptivity is working, at the expense of accuracy. The MPML files can be called with mpiexec and IC-FERST, and if no VTU or PVTU files are generated, the user can limit problems to IC-FERST.

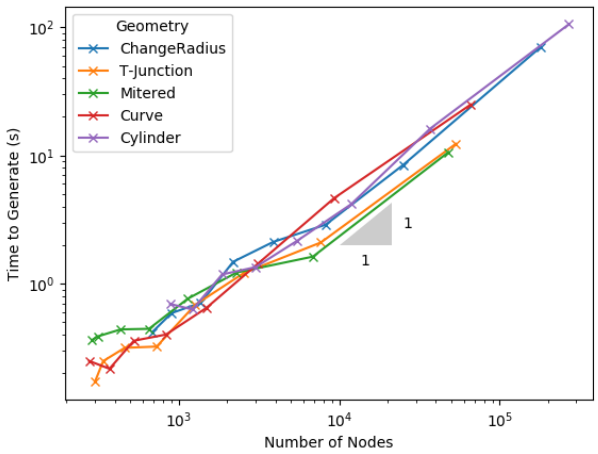


Figure 9: Mesh creation time for increasing numbers of nodes, for different geometries.

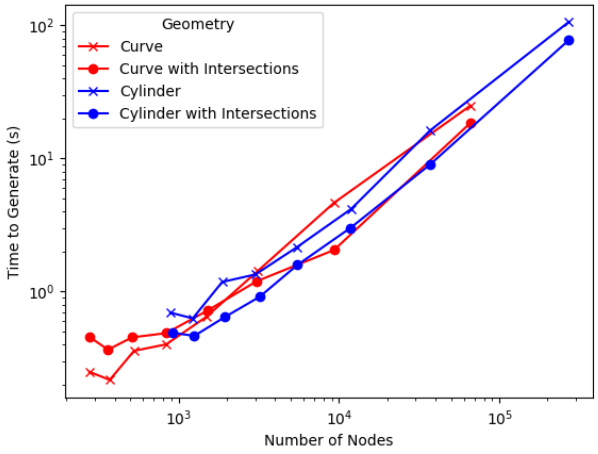


Figure 10: Mesh creation time for a cylinder and curve with intersections.

3. Results

The number of commands used in the creation of each *PipePiece* is detailed in Table 3, to show the minimum benefit to the user from the package.

The time taken to create meshes of varying number of nodes, for different geometries, is shown in Figure 9. The relationship is shown to be linear, with a log plot being used to show the relationship at both small and large numbers of nodes. The effect of using fuse operations was investigated by creating a cylinder of 7 fused cylinders, and a curve with 3 curved pieces. The results are shown in Figure 10.

A .geo script with the conventional GMSH method to create a pipe with two bends and 3 cylinders uses 35 commands, shown in Listing 5. The *pipemesh* alternative uses 7 commands, including the import command, shown in Listing 4.

To show the scripting ability of the software, pipe meshes with varying bend ratios were created with a simple script. Figure 11 shows two pipes with differing bend ratios, similar to the study undertaken in [20]. To create 5 pipe bends of differing bend ratios varying from 0.5 (sharp) to 0.1 (gentle), the script in Listing 3 (appendix) was used, with only 4 *pipemesh* commands called. To demonstrate how this could be used with a CFD code, they were simulated with IC-FERST for 7 seconds, at a Reynolds number of 80,000 to ensure turbulent flow. For all simulation results included in this report, the boundary conditions are constant velocity for inlets, no-slip for cylindrical surfaces, and 0 pressure for the outlets.

An industrial case was provided, and was recreated with 36 *pipemesh* commands. The recreation is shown in Figure 12. As this geometry, and many others, have not been created with the conventional GMSH method, there is no available comparison.

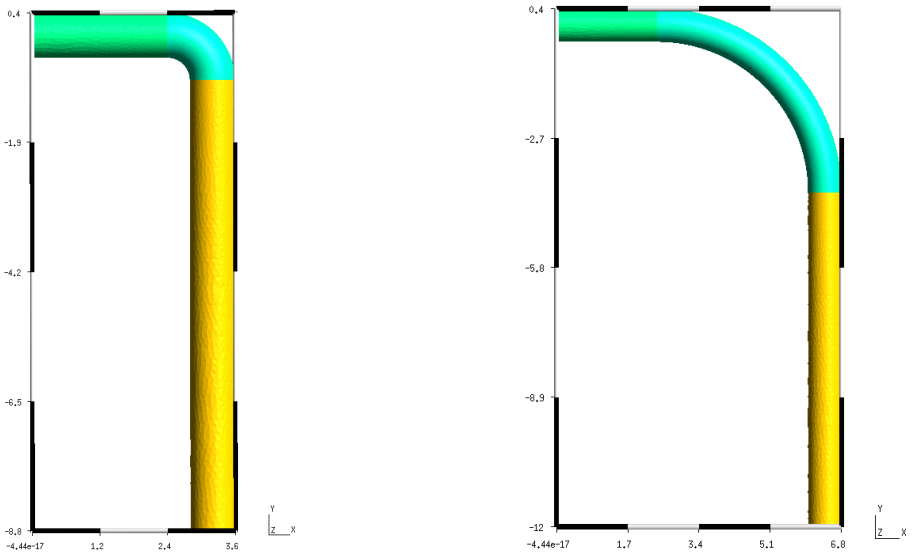
To demonstrate integration with IC-FERST, the bends of different ratios were simulated, as well as a configuration with a wider variety of *pipemesh*'s capabilities. This configuration is shown in Figure 13. To show the use of the *AutoMPML* class, an MPML file was created. For the simulation, the input Reynolds number for the top and joining pipes (on the right) was 50,000, created using the *get_velocities_reynolds* function of the *Network* class. The bottom two pipe entrances were set at 0 pressure outlets. The resolution of the result is aided by mesh-adaptivity, demonstrated in Figure 16.

4. Discussion

The software is shown to replace any alternatives effectively. Table 3 shows that the number of GMSH commands used is reduced, with between 3 and 10 commands (or more) replaced by a sin-

	Cylinder	Curve	Mitered	ChangeRadius	TJunction
Creation	1	2	7	2	5
Rotate Inlet	1	1	1	1	1
Rotate Outlet	0	1	1	0	1
Translate	1	1	1	1	1
Total Commands	3	5	10	4	8

Table 3: GMSH API commands used in adding object to Network. This does not include calculations to find translation vectors, rotation angles and rotation axes from the user inputs.



(a) A bend with bend ratio 0.5. (b) A bend with bend ratio 0.1

Figure 11: Sharpest and most gentle bends used in varying bends experiment. The Reynolds number used was 80,000 for all bend ratios. There was a preceding inlet of length 3D (2.4 m), and outlet length of 10D (8 m). Bend ratio is defined in Figure 2.

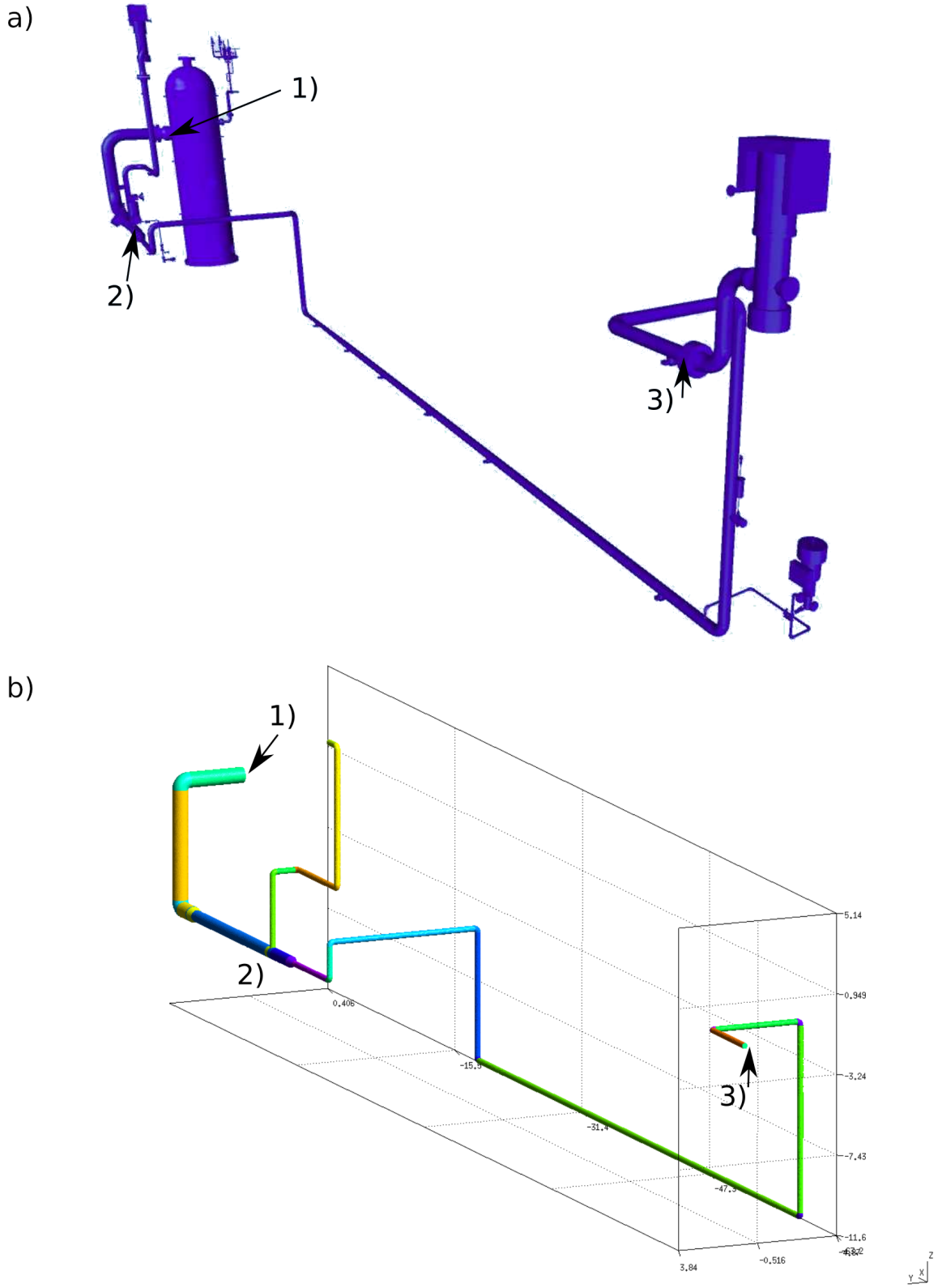


Figure 12: a) Industrial pipe flow design. 1) and 3) are inlets, 2) is the location of a T-junction, with 2 changes of radius either side. b) Design recreated with *pipemesh*. The *Pieces* used are *Curves*, *Cylinders*, *ChangeRadius* and one *TJunction*. The script to create this mesh with *pipemesh* is 40 lines long.

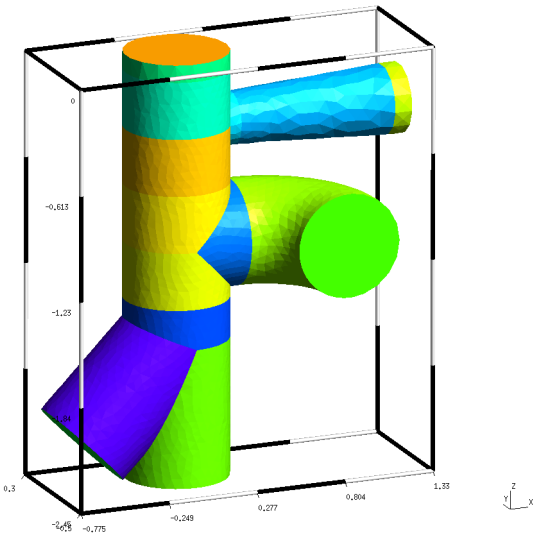


Figure 13: Configuration with variety of *pipemesh Pieces*.

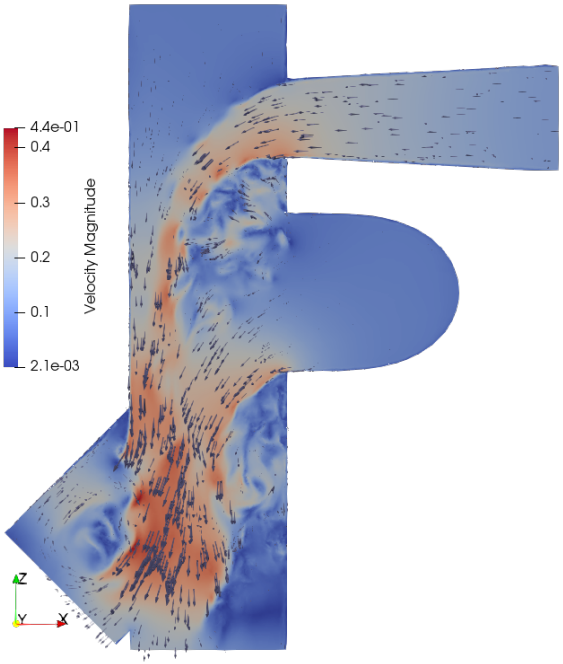


Figure 15: Slice of simulation results from configuration shown in Figure 13. Arrows are velocity vectors.

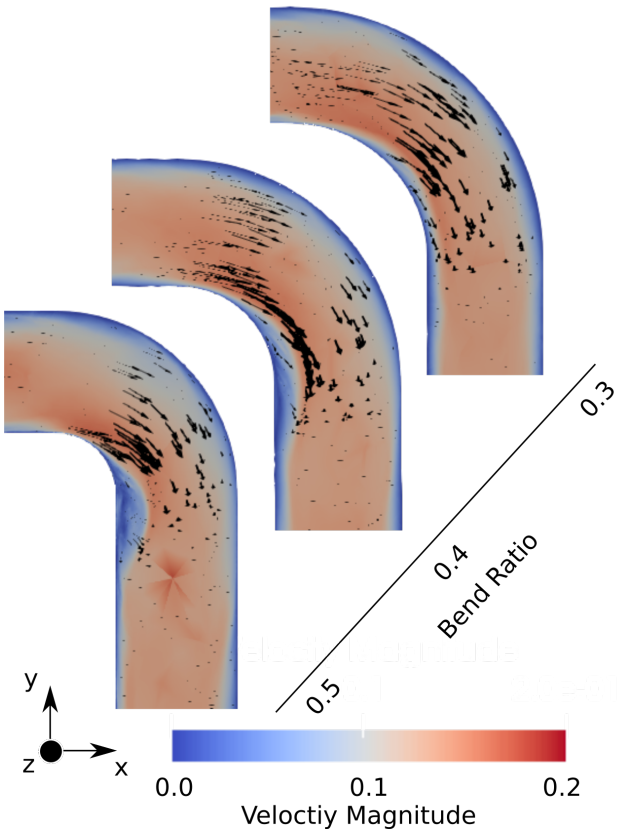


Figure 14: Slices of simulation results from the 3 sharpest bend ratios created. Flow separation is shown to decrease when the bend is smoother. Arrows are velocity vectors.

gle command. This assumes the creation method would be the same, and doesn't include the tracking of geometrical information or GMSH dimtags, of which the impact is hard to assess, but qualitatively it can be said that there is less tracking of information needed. The 35 commands needed by GMSH in Listing 5 to recreate a simple pipe design compared to *pipemesh*'s 7 is a quantitative example of its utility. Listing 3 is an example of how no GMSH commands are called directly, which removes the need to learn GMSH if the user is content to not develop the package further. If the user needs to develop a specific fitting, they would need to write a *PipePiece* class with an *init* function that calls the required GMSH commands. A drawback for the user is the repetitive process of creating designs, one that is hard to avoid if creating a design manually. If there is a standard method of specifying pipe network designs, a tool could be developed to convert instructions into designs. A further usability problem may arise from converting designs into geometrical parameters, for example finding direction vectors, and ensuring *TJunctions* and *Mitered Pieces* are accurate. The method of cre-

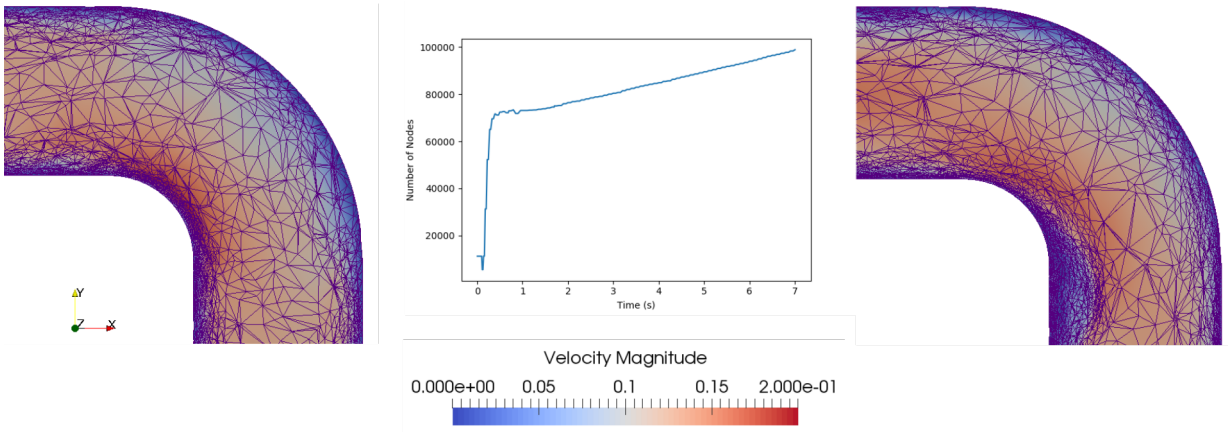


Figure 16: Left: Mesh at 0.5 seconds. Middle: Number of nodes varies during the simulation. Right: Mesh at 7 seconds. Images are slices through 3D mesh. Colour is velocity magnitude. The bend ratio is 0.5.

ating pipes sequentially is an easy to understand idea, but may become complex when multiple junctions are used, as the outlet numbers need to be tracked, and the joining pipes built 'backwards'. This could be solved by the ability add a *Network* to another *Network*, yet was not implemented as it is hard to ensure both *Networks* are facing in the correct directions, and is also likely to lead to many invalid *Networks* from intersections.

Figure 12 shows that industrial designs can be replicated by *pipemesh*. The accuracy of the recreation is only limited in *TJunctions*, due to the *Piece's* minimum dimensions. The available *Pieces* are the most common fittings seen in pipe networks, yet is still a small variety, with none having in-pipe elements which can be present in real life.

As the software only uses executable scripts, it is able to be run on most HPCs. Listing 3 demonstrates how batches of geometries varying a geometrical parameter such as bend ratio can be easily set up. Additionally, with IC-FERST and *AutoMPML*, scripts can be created to generate simulations with varying parameters such as the Reynolds number. The ability to create large numbers of meshes and simulations has potential for applications in machine learning, as large, varied datasets can be created easily through simulation for training models, which could be used in turn for reducing simulation times.

Integration with IC-FERST is proven with the results in Figures 14 and 15 showing qualitative agreement with physical turbulence. [20] states that for bend ratios greater than 0.3 flow separation occurs, which agrees well with the results shown by Figure 14, with the blue area to the inside of the bend increasing in size as bend ratios increase. Turbulent flow is demonstrated in Figure 15, where turbulence appears immediately downstream of the sharp changes in direction resulting from T-junctions. This turbulence limits flow from entering one of the outlets. The meshes can be adapted successfully and retain their geometry, as shown in Figure 16, where simple flow through the middle of the bend is seen to be less refined, and the area of flow separation towards the outside has an increased number of nodes. *pipemesh* creates a geometry with 10,000 nodes, and is adapted by IC-FERST to almost 100,000 as turbulence develops.

pipemesh itself is not designed in parallel, but the underlying GMSH-API is parellelised. The time to run *pipemesh* is strongly dependent on the size of mesh being generated, as shown in Figure 9, where a large mesh with 100,000 nodes can take up to 100 seconds to run. This trend is less clear at smaller mesh sizes, where the individual operations of GMSH have a more significant impact, yet not enough to increase run time to much more than 1 second. As simulation time also increases with number of nodes, added to the time consuming process of direct numerical simulation of turbulent flow, it is likely that the time spent generating meshes will be much less than

the time spent simulating them. Furthermore, as IC-FERST has mesh-adaptivity, a coarse mesh can be generated quickly and refined during the simulation. It is shown that the method of creating individual pipes and fusing them together using boolean operations does not heavily impact generation time. Figure 10 shows that for two similar cylinders, one consisting of one cylinder and one of 7 fused cylinders, the time to generate a mesh is less for that with intersections. For the curve, the trend is the same at larger number of nodes, but reverses at small number of nodes. This behaviour isn't fully explainable, but is beneficial to *pipemesh*, as the difference in mesh generation time at small numbers of nodes is almost inconsequential compared to the difference at large numbers of nodes.

Installation of *pipemesh* is very simple, as it is able to be pip-installed. By using a testing script, the user can test if their installation is correct, and that the version of the GMSH-API they have used is working correctly.

5. Conclusions

The *pipemesh* package has been shown in this report. It is a tool enabling sequential, modular and batch generation of pipe and pipe network meshes, with the user starting from an inlet and building in a simple method. The tool utilises the GMSH-API, and provides a layer between the user and API with the *Network* and *PipePiece* classes, reducing the required knowledge of GMSH. Specific industrial designs can be created using five different types of fittings, represented as Python objects: cylinders, smooth bends, T-junctions, sharp bends and pipe reducers. The software does not rely on a GUI, and can be run on an HPC through executable scripts. The time to generate meshes is dependent on the size of the mesh, and is not affected by fusing objects together. The software integrates with IC-FERST, allowing coarse meshes to be generated quickly by *pipemesh*, and refined during simulation. Simulation results show that the turbulence observed correlates qualitatively to real world phenomena.

Acknowledgements

Thanks are given to BP for providing funding for investigating pipe network flow in the AMCG department, facilitating this project.

[1] C. Pain, A. Umpleby, C. de Oliveira, A. Goddard, Tetrahedral mesh optimisation and adaptivity for steady-state and transient finite element calculations, Computer Methods in Applied Mechanics and Engineering 190 (29-30) (2001) 3771–3796. doi:10.1016/S0045-7825(00)00294-2.

[2] J. L. M. A. Gomes, D. Pavlidis, P. Salinas, Z. Xie, J. R. Percival, Y. Melnikova, C. C. Pain, M. D. Jackson, A force-balanced control volume finite element method for multi-phase porous media flow modelling, International Journal for Numerical Methods in Fluids 83 (5) (2017) 431–445. doi:10.1002/fld.4275.
URL http://doi.wiley.com/10.1002/fld.4275

[3] P. Salinas, D. Pavlidis, Z. Xie, C. Jacquemyn, Y. Melnikova, M. D. Jackson, C. C. Pain, Improving the robustness of the control volume finite element method with application to multiphase porous media flow, International Journal for Numerical Methods in Fluids 85 (4) (2017) 235–246. doi:10.1002/fld.4381.
URL http://doi.wiley.com/10.1002/fld.4381

[4] P. Salinas, D. Pavlidis, Z. Xie, A. Adam, C. C. Pain, M. D. Jackson, Improving the convergence behaviour of a fixed-point-iteration solver for multiphase flow in porous media, International Journal for Numerical Methods in Fluids 84 (8) (2017) 466–476. doi:10.1002/fld.4357.

[5] Z. Xie, D. Pavlidis, P. Salinas, J. R. Percival, C. C. Pain, O. K. Matar, A balanced-force control volume finite element method for interfacial flows with surface tension using adaptive anisotropic unstructured meshes, Computers & Fluids 138 (2016) 38–50.

[6] C. Geuzaine, J.-F. Remacle, Gmsh: A 3-d finite element mesh generator with built-in pre-and post-processing facilities, International journal for numerical methods in engineering 79 (11) (2009) 1309–1331.

[7] Open cascade.
URL https://www.opencascade.com/

[8] FluidityProject, Fluidityproject/fluidity.
URL https://github.com/FluidityProject/fluidity/wiki

[9] A. Li, X. Chen, L. Chen, R. Gao, Study on local drag reduction effects of wedge-shaped components in elbow and t-junction close-coupled pipes, in: Building Simulation, Vol. 7, Springer, 2014, pp. 175–184.

[10] J. G. M. Eggels, F. Unger, M. H. Weiss, J. Westerweel, R. J. Adrian, R. Friedrich, F. T. M. Nieuwstadt, Fully developed turbulent pipe flow: a comparison between direct numerical simulation and experiment, Journal of Fluid Mechanics 268 (1994) 175–210. doi:10.1017/S002211209400131X.

[11] K. Avila, D. Moxey, A. de Lozar, M. Avila, D. Barkley, B. Hof, The Onset of Turbulence in Pipe Flow, Science 333 (6039) (2011) 192–196. doi:10.1126/SCIENCE.1203223.

[12] X. Wu, P. Moin, A direct numerical simulation study on the mean velocity characteristics in turbulent pipe flow, Journal of Fluid Mechanics 608 (2008) 81–112. doi:10.1017/S0022112008002085.

[13] G. K. El Khoury, P. Schlatter, A. Noorani, P. F. Fischer, G. Brethouwer, A. V. Johansson, Direct Numerical Simulation of Turbulent Pipe Flow at Moderately High Reynolds Numbers, Flow, Turbulence and Combustion 91 (3) (2013) 475–495. doi:10.1007/s10494-013-9482-8.

[14] Reynolds number.
URL https://www.engineeringtoolbox.com/reynolds-number-d_237.html

[15] M. J. Tunstall, J. K. Harvey, On the effect of a sharp bend in a fully developed turbulent pipe-flow, Journal of Fluid Mechanics 34 (3) (1968) 595–608. doi:10.1017/S0022112068002107.

[16] H. Takamura, S. Ebara, H. Hashizume, K. Aizawa, H. Yamano, Flow Visualization and Frequency Characteristics of Velocity Fluctuations of Complex Turbulent Flow in a Short Elbow Piping Under High Reynolds Number Condition, Journal of Fluids Engineering 134 (10) (2012) 101201. doi:10.1115/1.4007436.

[17] F. Rütten , M. Meinke, W. Schröder, Large-eddy simulations of 90 pipe bend flows, Journal of Turbulence 2 (2001) N3. doi:10.1088/1468-5248/2/1/003.

[18] F. Rütten, W. Schröder, M. Meinke, Large-eddy simulation of low frequency oscillations of the Dean vortices in turbulent pipe bend flows, Physics of Fluids 17 (3) (2005) 035107. doi:10.1063/1.1852573.

[19] A. Kalpakli Vester, R. Örlü, P. H. Alfredsson, POD analysis of the turbulent flow downstream a mild and sharp bend, Experiments in Fluids 56 (3) (2015) 57. doi:10.1007/s00348-015-1926-6.

[20] L. Hufnagel, On the swirl-switching in developing bent pipe flow with direct numerical simulation.

[21] W. R. Dean, Xvi. note on the motion of fluid in a curved pipe, The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science 4 (20) (1927) 208–223.

[22] T. Hüttl, R. Friedrich, Direct numerical simulation of turbulent flows in curved and helically coiled pipes, Computers & Fluids 30 (5) (2001) 591–605. doi:10.1016/S0045-7930(01)00008-1.

[23] F. Sierra-Espinosa, C. Bates, T. O'Doherty, Turbulent flow in a 90 pipe junction. Part 2., Computers & Fluids 29 (2) (2000) 215–233. doi:10.1016/S0045-7930(99)00005-5.

[24] F. Sierra-Espinosa, C. Bates, T. O'Doherty, Turbulent flow in a 90 pipe junction. Part 1., Computers & Fluids 29 (2) (2000) 197–213. doi:10.1016/S0045-7930(99)00004-3.

[25] A. Kuczaj, E. Komen, M. Loginov, Large-Eddy Simulation study of turbulent mixing in a T-junction, Nuclear Engineering and Design 240 (9) (2010) 2116–2122. doi:10.1016/J.NUCENGDES.2009.11.027.

[26] A. Sakowitz, M. Mihaescu, L. Fuchs, Turbulent flow mechanisms in mixing T-junctions by Large Eddy Simulations, International Journal of Heat and Fluid Flowdoi:10.1016/j.ijheatfluidflow.2013.06.014.

[27] Stack overflow.
URL https://insights.stackoverflow.com/trends?tags=python

[28] A. Ribes, C. Caremoli, Salome platform component model for numerical simulation, in: 31st Annual International Computer Software and Applications Conference (COMPSAC 2007), Vol. 2, IEEE, 2007, pp. 553–564.

[29] pipemesh's documentation.
URL https://pipemesh.readthedocs.io/en/latest/

[30] E. Jones, T. Oliphant, P. Peterson, et al., SciPy: Open source scientific tools for Python (2001–).
URL http://www.scipy.org/

[31] T. E. Oliphant, A guide to NumPy, Vol. 1, Trelgol Publishing USA, 2006.

```
from pipemesh import pipes
import os, sys

pipe_radius = 0.4
diameter = 2*pipe_radius

bend_ratios = [0.1, 0.2, 0.3, 0.4, 0.5]
bend_radii = [pipe_radius/i for i in bend_ratios]

for bend_radius in bend_radii:
    network = pipes.Network(3*diameter, pipe_radius, [1, 0, 0], 0.1)
    network.add_curve([0, -1, 0], bend_radius, 0.05)
    network.add_cylinder(10*diameter, 0.1)
    os.mkdir("radius_{}".format(bend_radius))
    os.mkdir("radius_{}/src".format(bend_radius))
    network.generate(filename="radius_{}/src/pipe".format(bend_radius),
                     write_info=True,
                     binary=True,
                     run_gui=False)
```

Listing 3: Code to generate pipes of different bend ratios

```
from pipemesh import pipes

network = pipes.Network(10, 0.25, [0,0,-1], 0.1)
network.add_curve([0,-1, 0], 0.5, 0.1)
network.add_cylinder(5, 0.2)
network.add_curve([0, 0, -1], 0.5, 0.1)
network.add_cylinder(10,0.2)

network.generate(filename="src/pipe",
                 binary=True,
                 write_info=True,
                 run_gui=False)
```

Listing 4: Pipemesh creation of downward pipe with two bends. With the import statement, 7 commands are used.

```
charLen = 0.2;
charLen2 = 0.1;
r=0.25;
ex1=10;
ex2=5;
ex3=10;
Point(1) = {0,0,0,charLen};
Point(2) = {r,0,0,charLen};
Point(3) = {0,r,0,charLen};
Point(4) = {-r,0,0,charLen};
Point(5) = {0,-r,0,charLen};
Circle(1) = {2,1,3};
Circle(2) = {3,1,4};
Circle(3) = {4,1,5};
Circle(4) = {5,1,2};
Line Loop(5) = {1,2,3,4};
Plane Surface(6) = {5};
extr1[] = Extrude {0,0,ex1} {
    Surface{6};
};
extr2[] = Extrude { {1,0,0}, // direction of rotation axis
                    {0,2*r,ex1}, // a point on the rotation axis
                    -Pi/2 } { // the rotation angle
    Surface{extr1[0]};
};
extr3[] = Extrude {0,ex2,0} {
   Surface{extr2[0]};
};
extr4[] = Extrude { {1,0,0}, // direction of rotation axis
                    {0,ex2+2*r,ex1+4*r}, // a point on the rotation axis
                    Pi/2 } { // the rotation angle
    Surface{extr3[0]};
};
Extrude {0,0,ex3} {
   Surface{extr4[0]};
}
Physical Surface(1) = {116};
Physical Surface(2) = {6};
Physical Surface(3) = {19, 15, 23, 27, 49, 45, 41, 37, 59, 63, 71, 67, 81, 85,
                       93, 89, 115, 111, 103, 107};
Physical Volume(4) = {5, 4, 3, 2, 1}; //4 and 2 are curves
Characteristic Length {13, 8, 6, 19, 21, 26, 57, 69, 64, 70, 82, 86} = charLen2;
```

Listing 5: .geo file to create a downward pipe with two bends. Without variable setting, 35 commands are used.

```python
def test1():
    """Tests if save is correct."""
    network = pipes.Network(1, 0.25, [1, 0, 0], 0.1)  # Generate network
    gmsh.model.mesh.generate(3)
    n_nodes = len(gmsh.model.mesh.getNodes()[0])  # Access number of nodes in GMSH
    network.generate(filename="test")  # Write file
    with open("test.msh", 'r') as testFile:
        content = testFile.readlines()
        assert(content[1][:3] == "2.2")  # Check version
        assert(content[4][:3] == "{}".format(n_nodes))  # Check number of nodes
        assert(len(content) == 1736)  # Check number of lines
    os.remove("test.msh")  # Clean
    print("ASCII msh file working correctly.")
```

Listing 6: Test save function. A separate test works similarly for binary .msh files.

```python
def test3():
    """Tests if the mesh size is being changed."""
    network = pipes.Network(1, 0.25, [1, 0, 0], 0.2)  # Generate network
    network.add_cylinder(1, 0.1)
    gmsh.model.mesh.generate(3)
    n_nodes_before = len(gmsh.model.mesh.getNodes()[0])  # Access nodes before
    network.generate(filename="test", binary=False)  # Enforce refined mesh
    with open("test.msh", 'r') as testFile:
        content = testFile.readlines()
        n_nodes_after = int(content[4][:3])  # Access nodes after
    assert(n_nodes_after > n_nodes_before)  # Check there are more nodes
    print("Mesh size set correctly.")
```

Listing 7: Test mesh size enforcement.

```
def test4():
    """Tests individual pieces."""
    gmsh.initialize()
    cyl = pieces.Cylinder(1, 0.5, [1, 0, 0], 0.1)
    assert(np.allclose(cyl.out_surface.centre, [1, 0, 0]))
    assert(np.allclose(cyl.in_surface.centre, [0, 0, 0]))
    gmsh.finalize()

    gmsh.initialize()
    change_radius = pieces.ChangeRadius(
        1, 0.8, 0.3, 0.2, [1, 0, 0], 0.1
    )
    assert(np.allclose(change_radius.out_surface.centre, [1, 0, 0]))
    assert(np.allclose(change_radius.in_surface.centre, [0, 0, 0]))
    gmsh.finalize()

    gmsh.initialize()
    curve = pieces.Curve(
        0.25, [1, 0, 0], [0, 0, 1], 1, 0.1
    )
    assert(np.allclose(curve.out_surface.centre, np.array([1, 0, 1])))
    assert(np.allclose(curve.in_surface.centre, np.array([0, 0, 0])))
    assert(np.allclose(curve.out_surface.direction, np.array([0, 0, 1])))
    assert(np.allclose(curve.in_surface.direction, np.array([1, 0, 0])))
    gmsh.finalize()

    gmsh.initialize()
    mitered = pieces.Mitered(
        0.25, [1, 1, 0], [0, 0, 1], 0.1
        )
    assert(np.allclose(mitered.in_surface.direction, np.array([1, 1, 0])))
    assert(np.allclose(mitered.out_surface.direction, np.array([0, 0, 1])))
    gmsh.finalize()

    gmsh.initialize()
    t_junc = pieces.TJunction(
        0.3, 0.3, [0, 0, 1], [1, 0, 0], 0.1
    )
    assert(np.allclose(t_junc.in_surface.direction, np.array([0, 0, 1])))
    assert(np.allclose(t_junc.t_surface.direction, np.array([1, 0, 0])))
    gmsh.finalize()
    print("Indiviual pieces created correctly.")
```

Listing 8: Tests for individual *Pieces*.

```
def test5():
    """Tests if network updates after rotation."""
    network = pipes.Network(
        1, 0.25, [1, 0, 0], 0.1
    )
    network.add_curve([0, 0, 1], 1, 0.1)
    network.rotate_network([0, 1, 0], -np.pi/2)
    network.generate(run_gui=False)
    assert(np.allclose(
        network.out_surfaces[0].direction, np.array([-1, 0, 0])
        ))
    assert(np.allclose(
        network.in_surfaces[0].direction, np.array([0, 0, -1])
        ))
    print("Rotate whole network works correctly.")
```

Listing 9: Test for rotation of *Network*.

```
def test6():
    """Tests creation of velocities."""
    network = pipes.Network(
        1, 0.25, [1, 1, 1], 0.1
    )
    network.add_t_junction([1,0,0], 0.1)
    network.generate(run_gui=False)
    velos = network.get_velocities_reynolds([1, 3], 10000, 1000, 1e-3)
    assert(np.allclose(velos[1], np.array([-0.02, 0, 0])))
    velos_2 = network.get_velocities_vel_mag([1, 3], 0.02)
    assert(np.allclose(velos_2[1], np.array([-0.02, 0, 0])))
    print("Get velocities methods working correctly.")
```

Listing 10: Test for *get_velocities*.

```
def test7():
    """Tests get_ids methods."""
    network = pipes.Network(
        1, 0.25, [1, 1, 1], 0.1
    )
    network.add_t_junction([1, 0, 0], 0.1)
    network.generate(run_gui=False)
    inlet_phys_ids = np.array(network.get_inlet_outlet_phys_ids())
    assert(np.allclose(inlet_phys_ids, np.array([1, 2, 3])))
    cyl_phys_ids = network.get_cyl_phys_ids()
    assert(np.allclose(np.array([cyl_phys_ids]), np.array([4, 5, 6, 7])))
    print("Get IDs method working correctly.")
```

Listing 11: Test for *get_ids*.

```
from pipemesh import pipes
from pipemesh.icferst import auto_mpml

network = pipes.Network(0.5, 0.3, [0, 0, -1], 0.1)
network.add_curve([0, -1, 0], 0.5, 0.1)
network.add_cylinder(0.5, 0.1)

network.generate(filename="pipe", binary=True, write_info=True, write_xml=True)

entry_phys_ids = network.get_inlet_outlet_phys_ids()
cyl_phys_ids = network.get_cyl_phys_ids()
inlets = entry_phys_ids[:1]
outlets = entry_phys_ids[1:]

vel = 0.02
inlet_velocities = network.get_velocities_vel_mag(inlets, vel, 1000, 1e-3)

options = auto_mpml.AutoMPML()
options.set_all(sim_name="3d_pipe_elbow_test_case",
                msh_file="src/pipe",
                dump_ids=entry_phys_ids,
                density=1000,
                viscosity=1e-3,
                inlet_phys_ids=inlets, inlet_velocities=inlet_velocities,
                outlet_phys_ids=outlets,
                cyl_phys_ids=cyl_phys_ids,
                max_no_nodes=10000,
                min_mesh_size=0.08,
                finish_time=1.0,
                t_adapt_delay=0.5
                )
options.write_mpml("../3D_pipe_FEM")
```

Listing 12: Generate script used to check integration with IC-FERST.