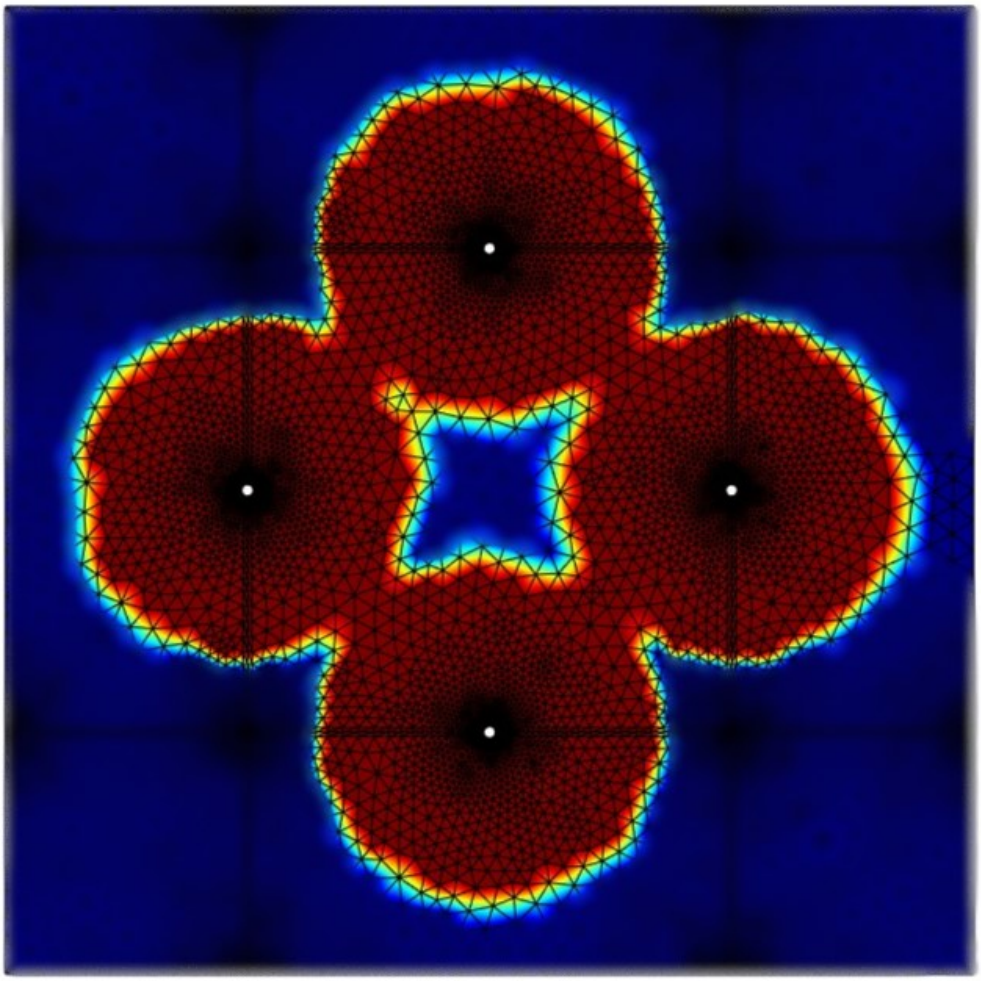


# RTM-flow

Users manual for version 1.0 June 2024

June 2024

# Contents

**Contents** **2**

**1 Introduction** **3**
1.1 Key Features of RTM-flow v1.0 . . . 3
1.2 Previous Software . . . 3
1.3 Goal of the Software . . . 3
1.4 Installation Requirements . . . 3
1.5 Installation Instructions . . . 4

**2 Structure of the Software** **5**
2.1 Code Hierarchy . . . 5
2.2 Accessing Properties and Methods . . . 5
2.3 Object-Oriented Programming in MATLAB . . . 6

**3 A Motivating Problem for Resin Transfer Molding** **7**
3.1 Mathematical and CV-FEM Formulation . . . 7
3.2 A First Example . . . 7

**4 Code Documentation** **12**
4.1 RTMFlow Class . . . 12
4.2 Physics Class . . . 13
4.3 Delaunay Mesh Classes . . . 13
4.4 Voronoi Mesh Classes . . . 14
4.5 Pressure Class . . . 15
4.6 Velocity Class . . . 16
4.7 Visualisation Class . . . 17

**5 An Example Extension: Modelling Dryspots** **19**
5.1 Introduction and Motivation . . . 19
5.2 Mathematical Framework . . . 19
5.3 Deriving a New RTM-flow Method . . . 19
5.4 Examples . . . 21

**6 Benchmark Tests** **24**
6.1 Benchmark Problem . . . 24
6.2 Error Computations . . . 24
6.3 Convergence Results . . . 26
6.4 Memory & Speed Performance . . . 26

**Bibliography** **27**

# Chapter 1

# Introduction

## 1.1 Key Features of RTM-flow v1.0

The current version of RTM-flow simulates two-dimensional experiments of resin transfer moulding (RTM) processes. The user specifies a meshed geometry, inlet and outlet pressure functions, and the material properties of the moulding tool. The software then simulates the flow of injection fluid from the inlet to the vent providing visualisations at each time increment as well as pressure, velocity and volume properties throughout the injection process.

## 1.2 Previous Software

RTM-flow is built from the code developed by M. Park and M.V. Tretyakov [5] and publicly available at `https://github.com/parkmh/MATCVFEM`. This code has been successfully used in studies of RTM processes [5, 1]. Limitations of this code include the simulation of complex domain geometries, the heavy workload required for a new user to set up a simulation, and the difficulty in making simple modifications to the code.

## 1.3 Goal of the Software

This software aims to provide researchers and industry practitioners with simple and adaptable access to the control volume finite element method (CV-FEM) for simulations in computational fluid dynamics, specifically injection moulding processes such as RTM. The software is being developed with three key objectives:

(1) The software must be completely self-contained within MATLAB, requiring no additional requirements, downloads or set-ups.
(2) The software must have a modular structure so that users can easily modify/extend the software to meet needs on a case-by-case basis.
(3) The software must have minimal coding overhead for the end user.

## 1.4 Installation Requirements

This software has been entirely written within MATLAB using version R2022b and later. A user of this software would only require an academic or industrial license for MATLAB, and we recommend installing the latest version of MATLAB at the time of reading. Instructions to install MATLAB on Windows, Mac OS, or Linux distributions can be found on the Mathworks website [2].

A simulation in RTM-flow can be executed without any additional toolboxes or third-party packages. In this manual we make use of the MATLAB Partial Differential Equations Toolbox [3] to provide a GUI to quickly generate complex meshes and to visualise the simulations, this is included for free with all MATLAB licenses. We emphasise that having this toolbox is not a requirement.

## 1.5 Installation Instructions

The CV-flow code is publicly available on GitHub [**?**]. It can be downloaded/cloned from this link `https://github.com/ChaffyHurdle/RTM-flow`. Once downloaded this software requires no further installation or initialisation steps. As with all MATLAB code the user must ensure the correct directory has been selected within MATLAB before using any of this software.

# Chapter 2

# Structure of the Software

## 2.1 Code Hierarchy

The structure of RTM-flow was designed with modularity in mind. We opted to make use of the object-oriented programming features of MATLAB to clearly distinguish between different aspects of the simulation and the numerical methods. This allows for a user or developer to quickly integrate new numerical methods, models and outputs within RTM-flow. The structure of the software is illustrated in Figure 2.1. The RTMFlow class encompasses all parts of the simulation, this includes the domain discretisation in the form of mesh classes and the fluid pressure and velocity properties in the form of classes. The governing equations and properties are stored in physics or PDE classes. The graphical outputs are stored within visualisation classes.

Each class is built in a way to minimise reliance on other classes within the RTMFlow class. Any information to be shared between classes must be passed by an argument within the method of a respective class, such as computing velocity from pressure:

```
velocity_class = velocity_class.compute_velocity(pressure_class);
```

Additionally, any properties of a class, such as geometric properties of the mesh, are computed and stored within the constructor of the respective class to reduce computational overhead.

By using this structure of the software, it becomes easy to modify existing code by using sub classes and inheritance and/or replace only a small part of the software with a different methodology. Examples of this include different meshing techniques, different discretisation methods for solutions, and a different set of governing equations. We demonstrate this in Chapter 5 by including an ideal gas law into the RTM simulation to capture dryspots.

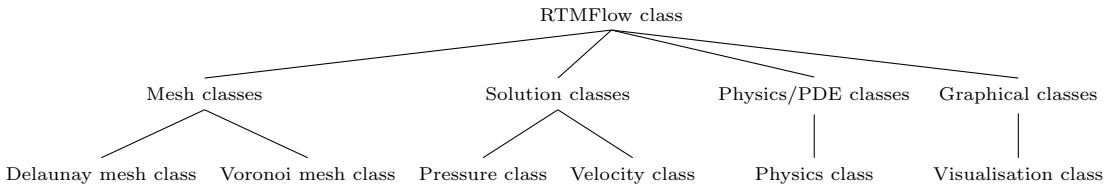



**Figure 2.1:** A diagram of the code hierarchy of RTM-flow v1.0. {fig::code_st

## 2.2 Accessing Properties and Methods

All methods and properties within each RTM-flow class are set to public so that users can access and/or modify these with ease. Methods for a class are stored either within the class file `<class name>.m` or within a folder `@<class name>`. These folders are automatically added to the MATLAB file path.

## 2.3 Object-Oriented Programming in MATLAB

Here we summarise the basics of MATLAB convention and syntax for object-oriented programming. For ease of reading we refer to an instance of a class as `obj`. To access a property of class, we simply call `obj.<property name>` whilst a method is called by `obj.<method name>(<args>)`.

```
obj.property;
obj.method(argument_1,argument_2);
```

A method defined for a class must include `obj` as the first argument, but `obj` is not part of the arguments when calling the method.

```
function method(obj,argument_1)
    disp(obj.property);
end

obj.method(argument_1)
```

If a use wishes to modify properties of a class within a class method, the class must be returned as an output of the function. This is because MATLAB does not pass by reference or value when executing a class method.

```
function obj =update_class(obj)
    obj.counter = obj.counter + 1;
end

obj = obj.update_class();
```

When a user is adding new methods within the `@<class name>` folder that have not been compiled before, the function will not be recognised because MATLAB will not automatically recompile the class. To quickly remedy this, a user can clear the class and re-instantiate it after deleting any existing instances using the IDE variables browser or `clear <my_class>` in the command window.

# Chapter 3

# A Motivating Problem for Resin Transfer Molding

## 3.1 Mathematical and CV-FEM Formulation

At a given time-level $t \in [0, t_f]$ we seek a solution to the pressure field problem of

$$-\nabla \cdot (K(\mathbf{x})\nabla p) = 0, \qquad (x, y) \in D, t > 0 \tag{3.1}$$
$$p(\mathbf{x}, t) = p_I(\mathbf{x}, t), \qquad (x, y) \in \partial D_i, t > 0 \tag{3.2}$$
$$\mathbf{n} \cdot \nabla p(t, x, y) = 0, \qquad (x, y) \in \partial D_o, t > 0 \tag{3.3}$$
$$p(0, x, y) = p_0, \qquad (x, y) \in D \tag{3.4}$$
$$\mathbf{n} \cdot \nabla p(t, x, y) = V(t, x, y), \qquad (x, y) \in \Gamma(t), t > 0 \tag{3.5}$$

## 3.2 A First Example

Here, we present a simple to code example of RTM-flow. We generate a mesh for a square domain $\Omega \subseteq [0, 1]^2$ in which we have removed a random selection of ellipses to produce a "swiss cheese" type of domain. We produced the mesh for this domain using the MATLAB Partial Differential Equation toolbox with the function `pdetool`. The mesh is saved under the name `swiss_mesh.m` and is an executable MATLAB script that will open the pdetool GUI when ran. The mesh used in this test is shown in Figure 3.1. The mesh properties are exported within the GUI as `p e t`.

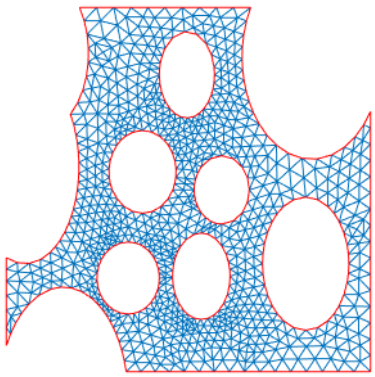


**Figure 3.1:** A mesh of the Swiss cheese domain with 876 nodes and 1476 elements. {fig::swiss_m

From this we can start writing a script to run RTM-flow. Firstly, we define the Delaunay mesh class using the exported `p e t` matrices:

```
my_mesh = DelaunayMesh(p,e,t);
```

Next we define a piecewise constant permeability of $K = 10^{-6}\mathbf{I}$ inside the circle of radius 0.25 centred at $(0.5, 0.5)$ and $K = 10^{-8}\mathbf{I}$ otherwise. In MATLAB we define the permeability as a function of a given coordinate `x`:

```
function K = permeability(x)

    if norm(x - [0.5 0.5])<0.25
        K = 1e-6 * eye(2);
    else
        K = 1e-8 * eye(2);
    end

end
```

The Physics class is then instantiated by passing the permeability as a function handle `@permeability` and setting the viscosity, porosity and thickness to $\mu = 0.1$, $\phi = 0.35$, and $H = 1$:

```
my_physics = Physics(0.1, 0.35, 1, @permeability);
```

The pressure class is then constructed by defining three functions: the boundary conditions `p_D`, the inlet condition `is_inlet`, and the vent condition `is_vent`. The boundary pressure is set to $p_0 = 10^{-5}$ and the injection pressure is set to a constant value of $p_I = 1.5 \times 10^5$. We implement this within the `P_D` function by returning a vector of pressure values for which the inlet values are set to $1.5 \times 10^5$ and the rest are set to $10^5$:

```
function p = p_D(pressure_class)

p = 0*pressure_class.pressure+1e5;
p(pressure_class.is_inlet) = 1.5e5;

end
```

The inlets and vents are determined by the functions `is_inlet` and `is_vent` which return true for any given coordinate `x` that lies on the inlet and vent respectively:

```
function bool = is_inlet(x)

bool = (x(2) == 0);

end

function bool = is_vent(x)

bool = (x(2) == 1);

end
```

We then can instantiate the pressure class passing the boundary condition functions as function handles:

```
my_pressure = Pressure(my_mesh,my_physics,@is_inlet,@is_vent,@p_D);
```

Using these three classes we compile the main RTM-flow class:

```
my_RTMflow = RTMFlow(my_mesh,my_physics,my_pressure);
```

At this point we can run the simulation using `my_RTMflow.run()`. To produce any graphical output, we have to configure the settings in the `visualisation_class` before running. In this example we wish to observe the moving volume of fluid and save an animation. This is done by setting the corresponding settings to true:

```
my_RTMflow.visualise_class.is_plotting_volume = true;
my_RTMflow.visualise_class.is_animate_volume = true;
```

At this point we can run the simulation using the `run()` function, this final solution at the filling time will then returned:

```
my_RTMflow = my_RTMflow.run();
```

Upon running the MATLAB script the user should see a video of the volume moving boundary in `figure(1)` and at the conclusion of the simulation should have a file `volume.gif`.

The simulation runs until an approximate filling time of $t = 27.0254$ and the injected resin covers the entire domain. Snapshots of the moving boundary within `volume.gif` are presented in Figure 3.2. The user should also be able to see the animation within `figure(1)` while the code is running. We also include here, shown in Figure 3.3, snapshots of the pressure field and the velocity field at the filling time, these can be obtained by adding `my_RTMflow.is_plotting_pressure=true;` and `my_RTMflow.is_plotting_velocity=true;` to the script. The complete script is saved as `swiss_test.m` and we include it below for reference

```
%% Problem set up
my_mesh = DelaunayMesh(p,e,t);
my_physics = Physics(0.1, 0.35, 1, @permeability);
my_pressure = Pressure(my_mesh,my_physics,@is_inlet,@is_vent,@p_D);

%% compile RTM-flow method
my_RTMflow = RTMFlow(my_mesh,my_physics,my_pressure);

%% graphical settings
my_RTMflow.visualise_class.is_plotting_volume = true;
my_RTMflow.visualise_class.is_animate_volume = true;

%% Execute solver
my_RTMflow = my_RTMflow.run();

%% Argument set up
function K = permeability(x)

    if norm(x - [0.5 0.5])<0.25
        K = 1e-6 * eye(2);
    else
        K = 1e-8 * eye(2);
    end

end

function p = p_D(pressure_class)

p = 0*pressure_class.pressure+1e5;
p(pressure_class.is_inlet) = 1.5e5;

end

function bool = is_inlet(x)

bool = (x(2) == 0);

end
```

```
function bool = is_vent(x)

bool = (x(2) == 1);

end
```

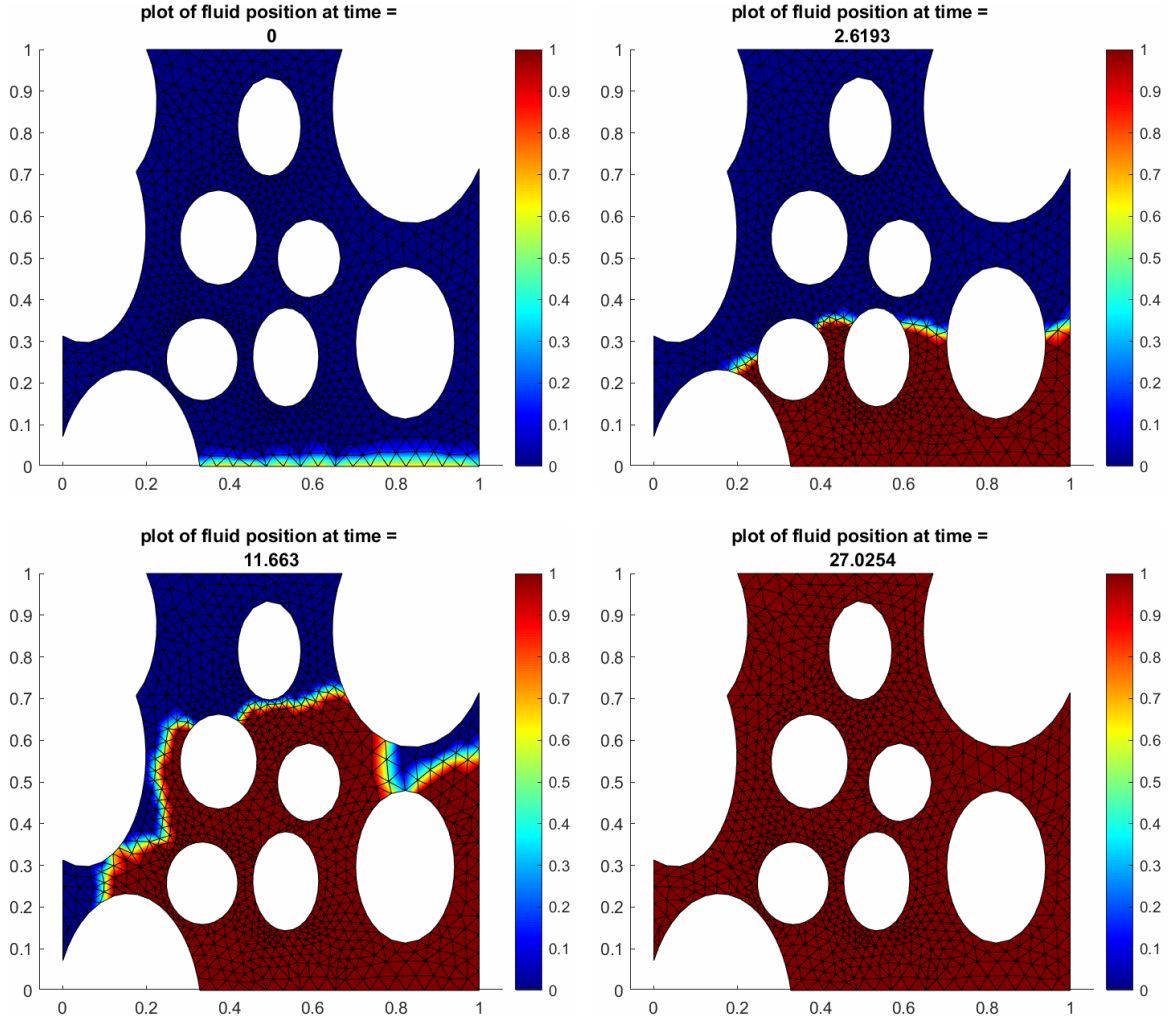
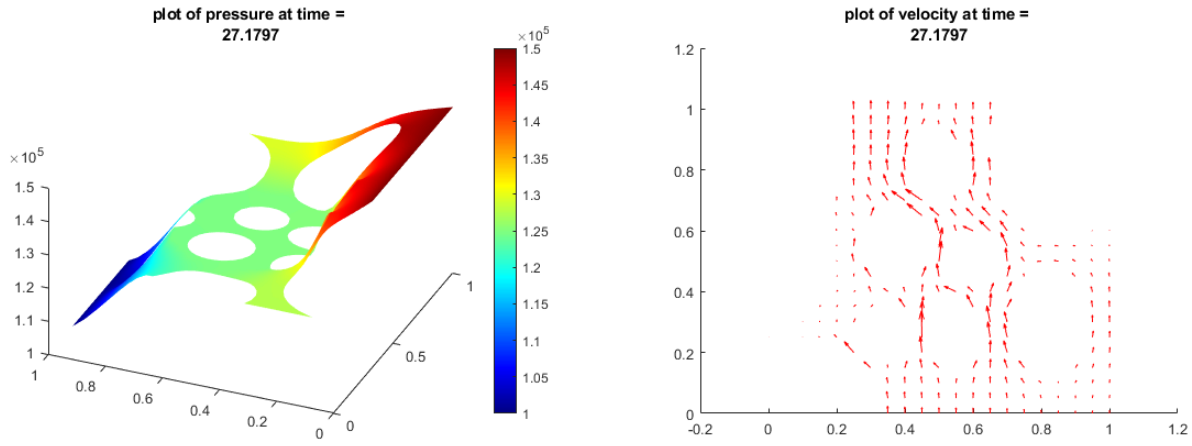



**Figure 3.2:** Snapshots of the volume saturation. The snapshots are taken at times $t = 0$ (top left), $t = 2.6193$ (top right), $t = 11.663$ (bottom left) and $t = 27.0254$ (bottom right).

ss_snapshots}




wiss_p_and_u}

**Figure 3.3:** Snapshots of the pressure field (left) and the velocity field at the filling time $t = 27.0254$.

# Chapter 4

# Code Documentation

## 4.1 RTMFlow Class

The RTMFlow class stores all components needed to simulate an RTM process with CV-FEMs. Classes for the physics, domain meshes, and solutions components are stored in separated classes. An additional class is included to store any user-specified graphical options.

### Instantiating

The constructor for this mesh class allows two possible input types. The first takes three inputs `RTM_flow_cls = RTMFlow(mesh_class,physics_class,pressure_class)` where
`mesh_class`, `physics_class`, and `pressure_class` are instantiated classes described in Sections 4.3, 4.2, and 4.5 respectively. The Voronoi mesh, velocity, and visualisation classes are instantiated within the constructor.

```
RTM_flow_cls = RTMFlow(mesh_class,physics_class,pressure_class);
```

### Properties

`Delaunay_mesh_class`: The Delaunay triangulation mesh class.

`pressure_class`: The fluid pressure class.

`Voronoi_mesh_class`: The Voronoi tessellation class.

`velocity_class`: The fluid velocity class.

`physics_class`: The governing physics/PDE class.

`visualise_class`: The graphical options class.

`time`: A scalar to track the time of the simulation.

`time_step`: A scalar to track the time-stepping size.

`volume_fill_percentage`: An $N_p$ length vector that tracks the percentage of each control volume filled with fluid.

`volume_filling_times`: An $N_p$ length vector for the time required to fill each control volume given the current fluid velocity.

`volume_rates_of_flow`: An $N_p$ length vector for the rate of change of volume in each control volume.

`new_filled_volume`: A vector of control volumes that have been filled in the current time-level.

`active_elements`: An $N_t$ length boolean which is true for each Delaunay mesh triangle that is in the fluid domain.

### Methods

`compute_flow_rates()`: This function computes the `volume_rates_of_flow` property using the current `velocity_class` solution.

`compute_time_increment()`: This function sets the `time_step` property to the minimum time to fully saturate the next control volume.

`is_fully_saturated()`: This function returns `true` if all vent connected control volumes are fully saturated.

`run()`: This function runs the whole simulation until the `is_fully_saturated()` function returns `true`.

`update_computational_domain()`: This function updates the computational elements used in the `pressure_class` as well as updating the Dirichlet boundary conditions for the pressure.

`update_filling_percentage()`: This function updates the `volume_filling_percentage` after a time-step and updates the `new_filled_volume` property.

`update_time_level()`: This function updates the `time` properties in the `RTMFlow` class as well as the `pressure_class` and `velocity_class`.

## 4.2 Physics Class

{sec::physics

The physics class stores key physical properties of the domain material used in any governing laws a user wishes to prescribe. Currently Darcy's Law is used to define boundary movement through the domain material and the important material properties are stored within this class.

### Instantiating

The class constructor currently takes material properties viscosity, porosity, thickness and permeability as inputs. Each input is expected to be constant besides permeability which should be passed as a function `K = permeability(x)` where `x` is a $2 \times 1$ vector of a point position and `K` is a $2 \times 2$ matrix. Non-constant viscosity, porosity and thickness will be added in future updates.

```
physics_class =  Physics(viscosity, porosity, thickness, permeability);
```

### Properties

`viscosity`: A constant value for the fluid viscosity of the domain.

`porosity`: A constant porosity value for the domain.

`thickness`: A constant thickness used to approximate a three-dimensional problem as a two-dimensional one.

`permeability`: A function handle that returns a $2 \times 2$ matrix for the permeability of the domain at a point.

### Methods

There are currently no methods for this class.

## 4.3 Delaunay Mesh Classes

{sec::mesh_cl

The Delaunay mesh class stores geometric information and methods for a Delaunay triangulation of a user specified domain. This class does not contain any problem specific information.

### Instantiating

The constructor for this mesh class allows two possible input types. The first takes three inputs of `p`, `e`, and `t` which are the points, edges and triangle matrices exported by the Partial Differential Equations toolbox within MATLAB. Alternatively, the user can provide a single input `.mat` file containing a structure with `p`, `e`, and `t` stored within it.

```
Delaunay_mesh_class = DelaunayMesh(p, e, t);
Delaunay_mesh_class = DelaunayMesh('mesh_file.mat');
```

### Properties

`nodes`: An $N_p \times 2$ matrix containing the coordinates of the $N$ nodes of the mesh.

`elements`: An $N_t \times 3$ matrix containing the indices of nodes that form each mesh triangle.

`boundary_nodes`: An $N_b$ length vector containing the indices of nodes that are lying on the mesh boundary.

`centroids`: An $N_t \times 2$ matrix of centroid coordinates for each of the $N_t$ triangles of the mesh.

`element_areas`: An $N_t$ length vector containing the areas of each of the $N_t$ triangles of the mesh.

`num_nodes`: The number of mesh nodes.

`num_elements`: The number of triangles in the mesh.

`num_boundary_nodes`: The number of boundary nodes in the mesh.

### Methods

`compute_element_areas()`: This function computes the area for each mesh triangle and stores the result in the class property `element_areas`.

`compute_element_centroids()`: This function computes the barycentric coordinates for each element and stores the result in the class property `centroids`.

## 4.4 Voronoi Mesh Classes

The Voronoi mesh class stores geometric information and methods for a Voronoi mesh/tessellation that is the dual mesh of the Delaunay mesh class. As with the Delaunay mesh class, this class does not contain any problem specific information.

### Instantiating

The constructor for this class only requires a Delaunay mesh class object and a physics class object to instantiate.

```
Voronoi_mesh_class = VoronoiMesh(Delaunay_mesh_class,physics_class);
```

### Properties

`Delaunay_mesh_class`: The dual Delaunay triangulation of the Voronoi mesh is stored for reference.

`volume_measures`: An $N_p$ length vector containing a three-dimensional volume approximation of each Voronoi cell.

`volume_outflow_vectors`: An $N_t \times 6$ that stores the outward weighted normal vectors for each Voronoi cell that intersects each Delaunay triangular element.

`node_connectivity`: An $N_p \times N_p$ boolean matrix which returns true if two Delaunay mesh nodes connect to form a Voronoi cell.

`element_connectivity`: An $N_t \times N_t$ boolean matrix which returns true if two Delaunay mesh triangles are connected.

`has_node_i`: An $N_p \times 9$ matrix that stores the triangle connected Delaunay mesh nodes.

`connected_polygons`: An $N_p$ cell array containing the Voronoi cells that intersect the boundary of a given cell.

### Methods

`compute_connectivity()`: This function computes the connectivity properties `node_connectivity`, `element_connectivity`, `has_node_i`, and `connected_polygons`.

`compute_volume_measures()`: This function computes the three-dimensional volume approximation property `volume_measures`.

`compute_volume_outflow_vectors()`: This function computes the property `volume_outflow_vectors`.

## 4.5 Pressure Class

{sec::pressur

The pressure class stores the numerical method to compute the pressure field (the finite element method) along with the corresponding system of equations, solvers and solution properties that a user may wish to access, modify or couple up with other solution classes. Our implementation of the pressure class is specific to the RTM example documented in Chapter 2 however, extensions/modifications to alternative problems can be easily done.

### Instantiating

The class constructor requires the Delaunay mesh class and physics class along with 3 function handles; `inlet_func`, `vent_func`, `p_D`. The `inlet_func` and `vent_func` must be user specified functions that take a 2 vector input of a point coordinate and return a boolean if the point lies on the inlet or vent boundary respectively. The `p_D` function takes the entire Pressure class as an input and must return an $N_p \times 1$ vector of pressure values which coincides with the inlet and vent pressure values (interior values are discarded by the solver later on).

```
pressure_class = Pressure(Delaunay_mesh_class, physics_class, inlet_func,
    vent_func, p_D);
```

### Properties

`mesh_class`: The Delaunay mesh class on which the pressure is approximated is stored for reference.

`physics_class`: The PDE/physics class governing the pressure equations is stored for reference and computation of systems of equations.

`inlet_func`: This function takes a coordinate (vector of length 2) and returns a boolean depending on if it lies on the pressure inlet boundary.

`vent_func`: This function takes a coordinate (vector of length 2) and returns a boolean depending on if it lies on the pressure vent boundary.

`p_D`: This function takes the class and returns an $N_p$ lengthed vector of pressure values which has the correct inlet and vent boundary values.

`time`: A floating point value to track the current time in the simulation.

`stiffness_matrix`: An $N_p \times N_p$ sparse matrix containing the finite element stiffness matrix of the pressure problem.

`load_vector`: An $N_p$ length vector containing the finite element load vector of the the pressure problem.

`pressure`: An $N_p$ length vector that contains the pressure values at each Delaunay mesh node.

`shape_fun_gradients`: An $N_t$ length cell, each entry containing a $2 \times 3$ matrix of gradient values for the finite element shape/basis functions defined on each triangle of the mesh.

`pressure_gradient`: An $N_t \times 2$ matrix where each row contains the gradient values of the pressure on each triangle of the mesh.

`is_Dirichlet`: An $N_p$ length boolean which is true for any mesh node that has a fixed pressure value, prescribed by a Dirichlet boundary condition.

`is_inlet`: An $N_p$ length boolean which is true for any mesh node that lies on the pressure inlet.

`is_vent`: An $N_p$ length boolean which is true for any mesh node that lies on the pressure vent.

`is_Neumann`: An $N_p$ length boolean which is true for any mesh node that lies on a sealed boundary with no inlet or vent.

`is_node_active`: An $N_p$ length boolean which is true for any mesh node that lies within the fluid phase.

`new_active_elements`: An $N_t$ length boolean which is true for any mesh triangles that lie within the fluid phase but is not yet included in the finite element method.

### Methods

`compute_inlets_outlets()`: This function uses the `mesh_class` properties along with `inlet_func`, `vent_func`, and `p_D` to compute `is_inlet`, `is_vent`, `is_Nuemann`, `is_Dirichlet`, `is_node_active`.

`compute_shape_fun_gradients()`: This function computes the gradient of each finite element shape function and stores them in `shape_fun_gradients`.

`assemble_stiffness_matrix()`: This function computes local finite element stiffness matrices for any `new_active_elements` and adds them to the global `stiffness_matrix`.

`compute_pressure_gradient()`: This function computes the gradient values of the pressure solution and stores them for each element in `pressure_gradient`.

`solve()`: This function solves the matrix equations for the pressure problem and saves the solution in `pressure`. The Dirichlet boundary conditions must be imposed before this function is used.

## 4.6 Velocity Class

The velocity class stores any information regarding the fluid velocity field. Currently, the velocity is computed from Darcy's Law.

### Instantiating

The class constructor requires the Vonoroi mesh class and physics class to instantiate.

```
velocity_class = Velocity(Voronoi_mesh_class, physics_class);
```

### Properties

`voronoi_mesh_class`: The Voronoi mesh class on which the velocity is approximated is stored for reference.

`physics_class`: The physics class governing the Darcy equations is stored for reference and computation of the velocity field.

`time`: A floating point value to track the current time in the simulation.

`velocity`: An $N_p \times 2$ matrix to store the piecewise constant velocity field over each Voronoi cell.

### Methods

`compute_velocity(pressure_class)`: This function computes the `velocity` using the Darcy Law provided in the `physics_class` and the `pressure_class`.

## 4.7 Visualisation Class

The Visualisation class stores the user settings and functions to produce and export graphics from the RTM-Flow simulations. Visualisations require the PDE Toolbox add-on to be installed. Figures are produced in separate windows depending on the user-specified settings. Animations are exported to unedited GIF files under the default filenames.

### Instantiating

To keep things simple for the user, the constructor for the Visualisation class has no input arguments. Instead the settings can be easily turned on or off by the user on a case-by-case basis. By default, all graphical settings are turned off in the visualisation class.

```
visualisation_class = Visualisation();
```

### Properties

`is_plotting_volume`: A boolean that when set to `true` will plot the saturated volumes at each time level in `figure(1)`.

`is_plotting_pressure`: A boolean that when set to `true` will plot the pressure profile at each time level in `figure(2)`.

`is_plotting_volume_flow`: A boolean that when set to `true` will plot the rate of volume increase/decrease of the control volumes at each time level in `figure(3)`.

`is_plotting_velocity`: A boolean that when set to `true` will plot velocity streamline arrows at each time level in `figure(4)`.

`is_plotting_volume`: A boolean that when set to `true` will export a GIF animation named `volume.GIF` of saturated volumes at each time level.

`is_animate_pressure`: A boolean that when set to `true` will export a GIF animation named `pressure.GIF` of the pressure profile at each time level.

`is_animate_volume_flow`: A boolean that when set to `true` will export a GIF animation named `volume_flow.GIF` of the rate of volume increase/decrease of the control volumes at each time level.

`is_animate_velocity`: A boolean that when set to `true` will export a GIF animation named `velocity.GIF` of the velocity streamline arrows at each time level.

### Methods

`plot(RTMFlow_class)`: This function plots and animates flow features depending on the settings specified in the Visualisation class.

# Chapter 5

# An Example Extension: Modelling Dryspots

{chap::dryspo

## 5.1 Introduction and Motivation

We demonstrate an easy extension to the RTM-flow software to simulate dryspots in RTM processes. Dryspots/ macrosopic voids can form behind the moving boundary when, for example, the fluid passes through regions of mixed permeability. This is a common cause of defective parts in composite manufacturing. As such software to detect and aid in the prevention of possible dryspots is of great value. We directly adapt the numerical method from the MATCVFEM software [4, 5] to demonstrate a quick and easy extension to the RTM-flow which provides extra complexity to the model.

## 5.2 Mathematical Framework

The behaviour of dry spots is modelled using an ideal gas law to simulate the entrapment of air in each dryspot. This is done with a two-phase model with one phase (resin) being an incompressible fluid and the other (air) being a compressible gas. It is assumed that for a given dryspot volume $v_i(t)$ that

$$\int_{v_i(t)} p(\mathbf{x}, t)\ d\mathbf{x} = C_i \tag{5.1}$$

for a time-independent constant $C_i > 0$. Upon formation of a closed pocket of air at time $t$, the constant $C_i$ is computed (assuming constant vent pressure $p_0$) via

$$C_i = p_0 * H * \sum_{CV_j \subset v_i(t)} |CV_j|, \tag{5.2}$$

where $CV_j$ denotes a control volume contained within $v_i(t)$ and $H$ the material thickness. A constant pressure inside the dryspot can then be computed at future time levels according to this conservation law. The Dirichlet boundary conditions are modified accordingly in the fluid phase pressure solver. We refer to [5] for further details on formulation and implementation.

## 5.3 Deriving a New RTM-flow Method

In simulating a dryspot in the RTM process, we follow the main steps as the original RTM-flow `run()` method. The only additions required are the implementation of Dirichlet boundary conditions on the pressure and the information on dryspots in the simulation. A derived class of `RTMFlow` called `RTMFlowDryspot` is designed to easily include the dryspot simulation code from MATCVFEM [4]. The new and overidden properties and methods of this class are detailed below.

### New Properties

`num_voids = 0`: An integer that counts the total number of dryspots in the simulation.

`void_volume`: An $N_p \times 2$ matrix that contains the volume of a dryspot in the first column and for a node contained within a dryspot and the indexed dryspot number in the second column.

`is_volume_void`: An $N_p$ length boolean which is true if the node is part of a dryspot.

`is_dryspots`: A boolean that is true if a dryspot is present in the simulation.

### New Methods

`apply_ideal_gas_law()`: This function modifies the Dirichlet boundary conditions for the pressure for nodes contained in dryspots.

`find_dryspots()`: This function checks for dryspots in the simulation and updates the `is_dryspots` and `is_volume_void` properties.

`void_partition()`: This function computes the volume of each identified dryspot and updates the `num_voids` and `void_volume` properties.

`run()`: This function executes the full RTM simulation tracking dryspots.

We include the overridden `run()` function below for reference:

```
function obj = run(obj)

while ~obj.is_fully_saturated()

    %% Solve pressure & velocity Problem
    obj.pressure_class = obj.pressure_class.solve();
    obj.velocity_class = obj.velocity_class.compute_velocity(obj.
       pressure_class);

    %% Solve flow problem
    obj = obj.compute_flow_rates();

    %% Visualise
    obj.visualise_class.plot(obj);

    %% Increment to new time
    obj = obj.update_time_level();

    %% Update flow volumes and moving boundaries
    obj = obj.update_filling_percentage();
    obj = obj.update_computational_domain();

    %% Compute any voids/vacuum
    obj = obj.find_dryspots();
    obj = obj.void_partition();
    obj = obj.apply_ideal_gas_law();

end

disp("end")

end
```

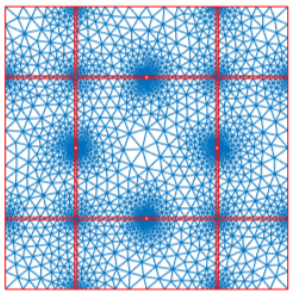


**Figure 5.1:** Plot of the domain mesh with 2413 nodes and 4632 elements. {fig::lab_mes

## 5.4 Examples

The modifications were tested with a model problem based on a tooling design for RTM processes. A $418mm \times 418mm \times IDK$ tool is approximated by creating a two-dimensional domain $\Omega = [0, 0.418]^2$ with a thickness parameter of $H = 0.2$. An example mesh of this tool is provided in Figure 5.1. The domain is subdivided into channels to create 9 rectangular regions. Each region is separated by four symmetric channels defined by $0.1025 \leq x \leq 0.165$, $0.3125 \leq x \leq 0.3155$, $0.1025 \leq y \leq 0.165$, $0.3125 \leq y \leq 0.3155$. Four equidistant injection holes of diameter $2.5mm$ are created at the midpoints of each channel. The tool is set up such that the fluid drains on the boundary $\partial\Omega$.

The permeability in each region is set to $K = 10^{-14}\mathbf{I}$ and within each channel it is set to $K = 10^{-10}\mathbf{I}$. The fluid viscosity is set to $\mu = 0.1$ and the porosity of the material is set to $\phi = 0.35$. The inlet pressure is kept at a constant $p_I = 1.5 \times 10^5$ Pa with a steady vent pressure of $p_0 = 1 \times 10^5$ Pa.

Snapshots of the simulation are presented in Figure 5.2 in which we observe the race-tracking phenomena where the fluid is rapidly transported between the channels of the tool. A macro void is produced when the fluid encompasses the inner square region which slowly compresses over time. Finally, we remark that the difference in permeability between the channels and the regions results in a substantial amount of time for the resin to saturate the entire boundary of the domain.

We include the `lab_test.m` script below for reference (the animation produced is saved as `lab_test.gif`):

```
%% Problem set up
my_mesh = DelaunayMesh(p,e,t);
my_physics = Physics(0.1, 0.35, 0.2, @permeability);
my_pressure = Pressure(my_mesh,my_physics,@is_inlet,@is_vent,@p_D);

%% compile CVFEM method
my_RTMFlow = RTMFlowDryspot(my_mesh,my_physics,my_pressure);
my_RTMFlow.visualise_class.is_plotting_volume = true;

%% Execute solver
my_RTMFlow.run()

%% Argument set up
function K = permeability(x)

    K = 1e-14 * eye(2);

    %% is point in channels
    is_left_channel = abs(x(1)-0.1045)<= 0.002;
    is_right_channel = abs(x(1)-0.3135)<= 0.002;
    is_lower_channel = abs(x(2)-0.1045)<= 0.002;
```

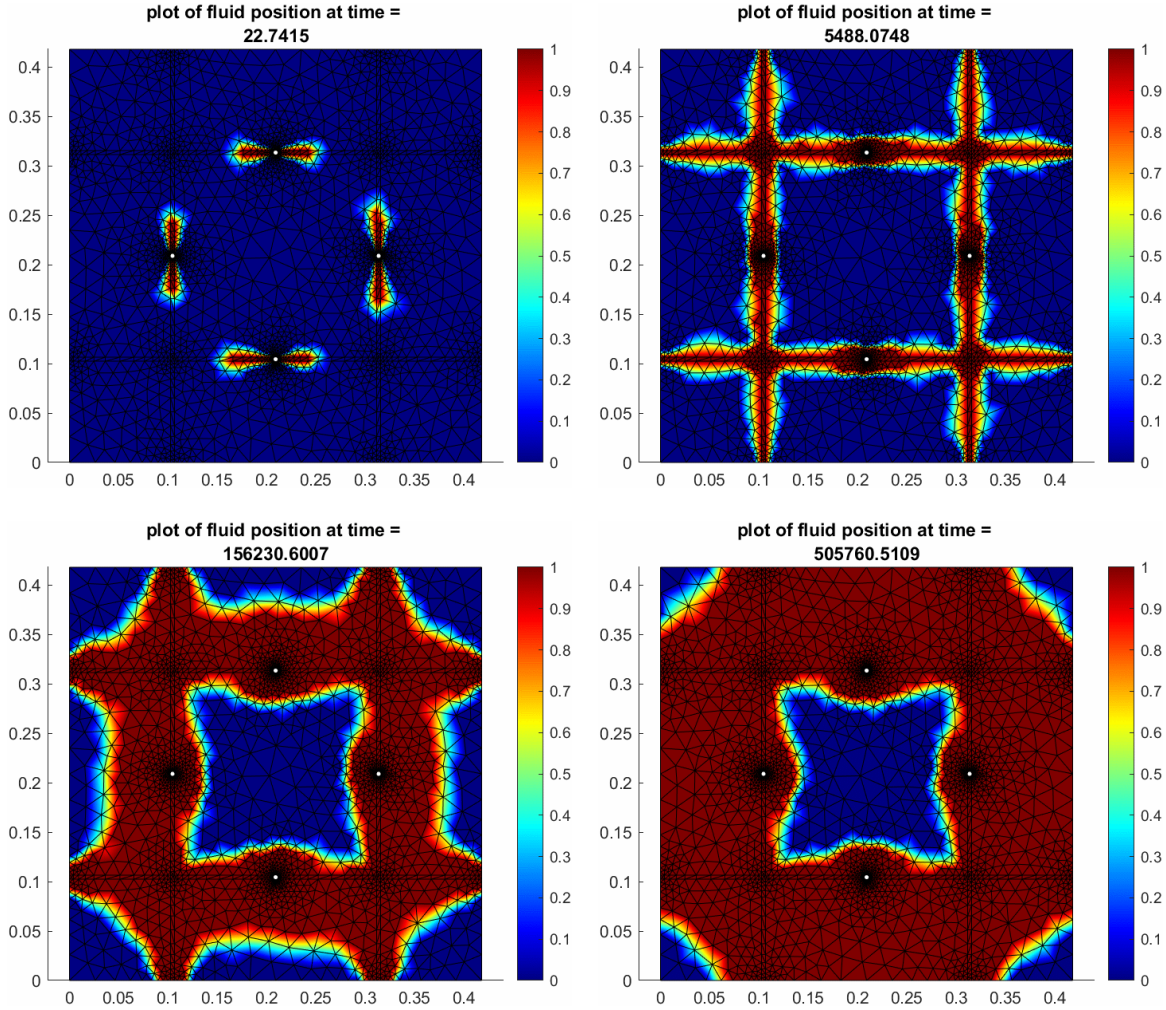



**Figure 5.2:** Snapshots of the volume saturation. The snapshots are taken at times $t = 22.7415$ (top left), $t = 5488.0748$ (top right), $t = 156230.6007$ (bottom left) and $t = 505760.5109$ (bottom right). {fig::lab_sna

```
    is_upper_channel = abs(x(2)-0.3135)<= 0.002;

    if  is_left_channel || is_right_channel || is_upper_channel ||
        is_lower_channel

        K = 1e-10*eye(2);
    end
end

function p = p_D(pressure_class)

p = 0*pressure_class.pressure + 1e5;
p(pressure_class.is_inlet) = 1.5e5;

end

function bool = is_inlet(node)

%% inlets on central circles (eps neeeded for rounding error)
```

```
is_north_inlet = norm(node-[0.209, 0.3135]) < 0.0025+eps;
is_south_inlet = norm(node-[0.209, 0.1045]) < 0.0025+eps;
is_east_inlet = norm(node-[0.3135, 0.209]) < 0.0025+eps;
is_west_inlet = norm(node-[0.1045, 0.209]) < 0.0025+eps;

bool = is_north_inlet || is_south_inlet || is_east_inlet || is_west_inlet;

end

function bool = is_vent(node)

bool = (node(1) == 0 || node(2) == 0 || node(1) == 0.418 || node(2) ==
   0.418);

end
```

# Chapter 6

# Benchmark Tests

## 6.1 Benchmark Problem

We base our convergence rate tests against the analytical solution presented in [6] and references therein. The solution is an extension of a one-dimensional solution extended to two dimensions. The solution equations are given by

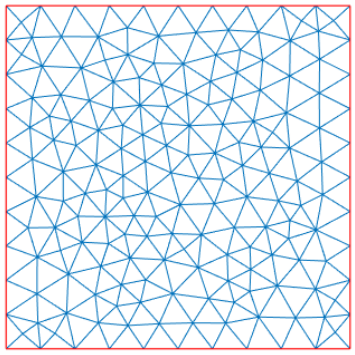
:benchmark_1}
$$x_\Gamma(t) = \sqrt{\frac{2Kp_0t}{\phi\mu}} \tag{6.1}$$

:benchmark_2}
$$p(x,t) = p_0\left(1 - \frac{x}{x_\Gamma(t)}\right) \tag{6.2}$$

:benchmark_3}
$$t_f = \frac{\phi\mu L^2}{2KP_0}, \tag{6.3}$$

where $L$ is the length of the domain, $x_\Gamma(t)$ denotes the $x$-coordinate of the moving boundary, $t_f$ is the time to fill the domain, $p_0$ a constant inlet pressure, and $K$ is the constant permeability. The material parameters are set to constant values of $\mu = 0.1$, $\phi = 0.35$, and $H = 1$ for viscosity, porosity and thickness respectively. The permeability tensor is set to $K = 10^{-10}\,\mathbf{I}$.

We tested this analytical solution against a simple a unit square domain $[0,1]^2$, a mesh of this domain is presented in Figure 6.1. We extend the Equations (6.1)-(6.3) on the interval $[0,1]$ to two-dimensions by maintaining a constant inlet pressure on the boundary at $x = 0$, resulting in a uniform flow across the domain.



hmark_meshes}
**Figure 6.1:** Plot of a unit square mesh.

## 6.2 Error Computations

In these tests we measure the error for three variables. The filling time error is measured as the absolute difference between the theoretical filling time defined in Equation (6.3) and the end time of the simulation

`RTMFlow_class.time`. The pressure error is measured by a nodal average error (or a discrete $l^1$ norm) at fixed time $t \in (0, t_f)$

$$||p(\mathbf{x}, t) - p_h(\mathbf{x}, t)|| \approx \frac{1}{N_p} \sum_{i=1}^{N_p} |p(\mathbf{x}_i, t) - p_h(\mathbf{x}_i, t)|. \tag{6.4}$$

Similarly, the boundary error is measured as a nodal average error of outward normal position difference over the $N_\Gamma$ mesh points on the moving boundary $\Gamma_h(t)$.

$$||\Gamma(t) - \Gamma_h(t)|| \approx \frac{1}{N_\Gamma} \sum_{i=1}^{N_\Gamma} |\mathbf{x}_i \cdot \mathbf{n} - x_\Gamma(t)|. \tag{6.5}$$

We sample the pressure and boundary solutions at approximately half-way through the process. This information is captured by introducing a new sub-class `RTMFlowConvergence` of `RTMFlow`. Within `RTMFlowConvergence` , we have modified the `RTMflow_class.run()` function as follows.

```
function [sample_obj, obj] = run(obj,sample_time)

sample_obj = [];

while ~obj.is_fully_saturated()

    %% Solve pressure & velocity Problem
    obj.pressure_class = obj.pressure_class.solve();
    obj.velocity_class = obj.velocity_class.compute_velocity(obj.
       pressure_class);


    %% Solve flow problem
    obj = obj.compute_flow_rates();

    %% Visualise
    obj.visualise_class.plot(obj);

    %% save simulation at first instance over time point
    if obj.pressure_class.time >= sample_time && isempty(sample_obj)
        sample_obj = obj;
    end

    %% Increment to new time
    obj = obj.update_time_level();

    %% Update flow volumes and moving boundaries
    obj = obj.update_filling_percentage();
    obj = obj.update_computational_domain();

end

disp("end")

end
```

| $N_p$ | filling time error | pressure error | boundary error |
|---|---|---|---|
| 177 | 0.01256 | 0.001488 | 0.018471 |
| 665 | 0.0065623 | 0.0014121 | 0.0093403 |
| 2577 | 0.0028396 | 0.0004952 | 0.0047723 |
| 10145 | 0.00076614 | 0.00025699 | 0.0027939 |

it_sqr_error}

**Table 6.1:** Numerical errors for the unit square benchmark test

| $N_p$ | run time (s) | Memory usage (GB) |
|---|---|---|
| 177 | 0.6239 | 0.4845 |
| 665 | 4.2704 | 1.8485 |
| 2577 | 59.1520 | 7.3221 |
| 10145 | 943.1400 | 29.1480 |

_performance}

**Table 6.2:** A table of run times and memory usage for the RTM-flow simulation on the benchmark test.

| $N_p$ | run time (s) | Memory usage (GB) |
|---|---|---|
| 177 | 27.0880 | 0.4705 |
| 665 | 94.6290 | 1.8471 |
| 2577 | 468.5300 | 7.3221 |
| 10145 | 3449.7000 | 29.1480 |

_performance}

**Table 6.3:** A table of run times and memory usage for the RTM-flow simulation on the benchmark test with the volume plotting setting turned on.

## 6.3 Convergence Results

We assess the errors by the asymptotic rates of convergence, noting that the mesh size $h \sim N_p^{-1/2}$. The numerical errors are presented in Table 6.1. The error in the filling time, pressure and moving boundary scale like $O(h)$. This observed rate of convergence is likely a consequence of a piecewise constant approximation of the fluid velocity and a linear time-stepping scheme. Improvements to these rates of convergence would require higher-order discretisation schemes. In [5], when the sequence of meshes correlated to the direction of fluid travel and the solution was compared to a finer mesh approximation, $O(h^2)$ accuracy is observed.

## 6.4 Memory & Speed Performance

We record the memory and speed performance of RTM-flow on the benchmark test. The experiment was done on an Intel Xeon W-2123 CPU @3.60GHz, Windows 10 device with 31.7GB of available RAM. The timing is measured on each test using the `tic` and `toc` functions and the memory usage is assessed using the `whos` function.

In Table 6.2, we record the simulation time and memory usage for each mesh refinement in which we produce no graphical outputs. It is observed that the run time of RTM-flow scales like $O(h^{-4})$ whilst the memory usage scales like $O(h^{-2})$.

For comparison, we re-run the test with the `is_plotting_volume` set to `true`. Memory usage is negligibly different from that in Table 6.2. The run time scales like $O(h^{-4})$ as before but we observe substantial increases in the run time compared to Table 6.2. The multiplication in run time between producing graphical outputs and not scales like $O(h)$. However, given the substantial increase in run time on coarser meshes we recommend avoiding graphical outputs if they are not required in simulations.

# Bibliography


[1] M. Iglesias, M. Park, and M. V. Tretyakov. Bayesian inversion in resin transfer molding. *Inverse Problems*, 34(10):105002, 2018.

[2] MathWorks Inc. Help center: Install products. `https://uk.mathworks.com/help/install/install-products.html`, 2024.

[3] MathWorks Inc. Partial differential equation toolbox. `https://uk.mathworks.com/products/pde.html`, 2024.

[4] M. Park. Matlab toolbox for the control volume finite element method (cvfem). `https://github.com/parkmh/MATCVFEM`, 2024.

[5] M. Park and M. Tretyakov. Stochastic resin transfer molding process. *SIAM/ASA Journal on Uncertainty Quantification*, 5, 06 2016.

[6] A. Saad, A. Echchelh, M. Hattabi, M. El Ganaoui, and F. Lahlou. Numerical simulation and analysis of flow in resin transfer moulding process. *Fluid Dynamics & Materials Processing*, 8(3):277–294, 2012.