Algorithm for voxelating spatial data obtained from a finite element multi-physics solver

Thesis Report

Submitted in partial fulfilment of the requirements of

BITS F421T, Thesis

by

Vibha Balaji
ID no 2016B5TS0385G

Under the supervison of
Dr. Deepan Balakrishnan

BITS Pilani, K K Birla Goa Campus

ACKNOWLEDGEMENTS

I would like to express my heartfelt gratitude to Dr Deepan Balakrishnan, for supervising my thesis. His constant guidance and patience kept me motivated throughout. I also extend my gratitude to Dr Duane Loh, for co-examining my work.

I also express my appreciation to Dr Swastibrata Bhattacharyya, for her support as co-supervisor.

The weekly meetings and discussions at CBIS have been a valuable source of learning for me. The diversity of topics discussed will continue to inspire me.

Finally, I would like to personally thank my family and my friends. Without their constant support, my thesis would not have been the same.

CERTIFICATE

This is to certify that the Thesis titled, ***Algorithm for voxelating spatial data obtained from a finite element multi-physics solver*** is submitted by **Vibha Balaji**, ID No **2016B5TS0385G** in partial fulfilment of the requirements of **BITS F421T**. Thesis embodies the work done by her under my supervision.

[illegible]

Dr. Deepan Balakrishnan
(CBIS, NUS)

Date : 3rd December 2019

LIST OF SYMBOLS AND ABBREVIATIONS

1. TEM - Transmission Electron Microscope
2. CFD - Computational Fluid Dynamics
3. FEM - Finite Element Method
4. FEA - Finite Element Analysis

ABSTRACT

Here we explore a starting point in understanding the collapse of nano-pillar arrays during fluid draining. Simulations from the multi-physics software ANSYS provide time-step wise data which can be sent to a Transmission Electron Microscope simulator to view the collapse as though it were from a microscope along with the intermediate steps. The processing of unstructured data from ANSYS Fluent will be processed to identify the solid pillars and filtered to obtain voxel data. The general methodology is automated to work for various types of meshes from ANSYS.
Various methods in image processing are implemented to work around challenges involving mesh density and resolution, to optimise the parameters involved and ensure that the values perform well for various values of mesh density.

”Science, like nature, must also be tamed,
with a view towards its preservation.”

Natural Science, Rush.

Contents

1 Introduction **1**
1 Capillary-induced collapse of nano-pillar arrays . . . 1
2 Pipeline . . . 1
2.1 Black-box view . . . 2

2 Processes used in Computational Fluid Dynamics **3**
1 Finite Element Method . . . 3
1.1 Discretization of the Domain . . . 3
1.2 Mesh Density . . . 3
2 Underlying Equations and Models . . . 3
2.1 Conservation equations . . . 4
2.2 Fluid models . . . 4
3 System Coupling . . . 5
4 Format of the output data from ANSYS . . . 6

3 Image Processing tools **7**
1 Packages imported for data handling . . . 7
2 Image Manipulation techniques . . . 7
2.1 Gaussian filter . . . 7
2.2 Binary opening . . . 7

4 Voxelating Algorithm - Pseudocode **9**

5 Synopsis of the Voxelating Algorithm **10**
1 Format of the input files . . . 10
2 Sorting the data into slices . . . 10
3 Gridding the scattered node coordinates . . . 10
3.1 Mesh density . . . 10
3.2 Scaling the grid . . . 11
3.3 Aligning the coordinates . . . 13
4 Extract solid from empty region . . . 13
4.1 Kernel creation . . . 14
4.2 Application of mask . . . 14
5 Filter the fluid data . . . 15

6 Parameter Analysis **17**
1 Estimating optimum values of the parameters . . . 17
1.1 Scaling factor . . . 17
1.2 Sigma for the mask filter . . . 19
1.3 Kernel size . . . 19
1.4 Sigma for the fluid filter . . . 20
1.5 Fluid threshold . . . 20

7 Results **21**

8 Conclusion **23**

A Computation Time **24**

0.1 Parallel Processing 24

B Python code of the voxelating algorithm **25**

References **29**

Chapter 1

Introduction

High aspect-ratio micro and nano pillars find increasing importance in various fields in micro and nano technology [1]. Their use in the development of cost-effective solar arrays involves semiconductor materials [2]. They also find application in anchoring neuron bodies in the culture of neural networks. Characterizing the properties of a nano-pillar array is often done using a Transmission Electron Microscope (TEM).
In TEM, an electron beam is emitted onto the sample. Interactions between the electrons and atoms of the sample at this scale result in elastic and inelastic scattering of the electrons. When the electrons are scattered elastically, they are confined to a region less than a nanometer in size. TEM images can give an idea of the relative density, and hence depth of the sample. Thus, TEM is ideal samples at the nanometer scale. [3]

1 Capillary-induced collapse of nano-pillar arrays

The high aspect-ratio of the nano-pillars compromises their structural tenacity. De-wetting the nano-pillar array while cleaning with a fluid causes the fluid's capillary forces to dominate, resulting in the rise of fluid level to form a tube. This in turn causes the pillars to collapse onto each other. In order to counter this problem, it would be valuable to observe this collapse happening under a TEM. Unfortunately, the limited resolution of the microscope cannot capture the intermediate steps of the collapse. Hence, as a preliminary step, multi-physics simulations of the nano-pillar arrays are carried out to model their wetting and de-wetting behaviour.
The data from these simulations can be passed to a TEM simulator, which yields output similar to that from the microscope. This report focuses on bridging the connection between multi-physics simulations of the nano-pillar arrays to the TEM simulator.

2 Pipeline

The user sets up the multi-physics simulation of a nano-pillar array, passing the inputs involving geometry, material, and fluid models used for the analysis. The simulation pipeline then takes the resultant unstructured nodal data.
The TEM simulator accepts the same data, which has been processed to 3D voxels, along with other inputs such as magnification, defocus, pixel length, electron energy that are required to simulate the microscope. A visualisation of this pipeline is given below.

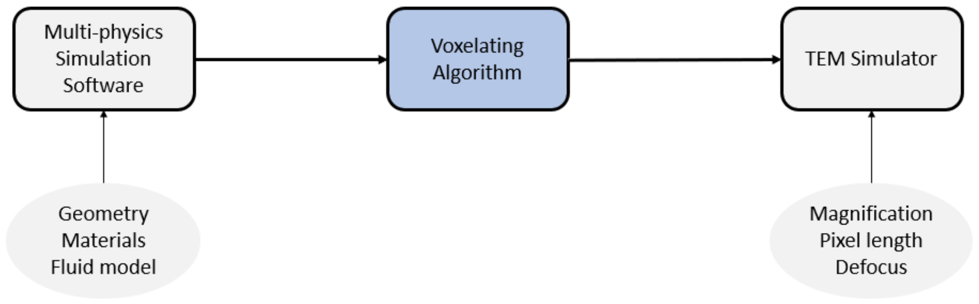


Fig 1.1. Overview of the pipeline

The following sections elucidate the steps involved in bridging the gap between unstructured nodal data from the nano-pillar simulations and the voxels sent to the TEM simulator, which is the aim of this report.

2.1 Black-box view

The pipeline takes in inputs on-the-fly from ANSYS. The files are sent as ANSYS Fluent keeps generating files for each time step.

Input

- Node number
- Node x,y,z coordinates
- Volume fraction of water in the node

Output

An array containing regularised and filtered plane-wise data of the whole structure, ready to be sent to the TEM simulator.

Our aim is to automate the pipeline. The user need not input various parameters for the voxelating algorithm, hence it acts as a black box.

Chapter 2

Processes used in Computational Fluid Dynamics

ANSYS, short for Analytical Systems, is a multi-physics software developed for engineering simulations to approximate systems in real-life. It provides the user capabilities in the numerical analysis of models involving structural design, mechanical properties, fluid interactions, electro-magnetics, and more. It employs the use of Finite Element Analysis (FEM) to converge to a solution.
For this report, we have used Transient Structural to design the nano-pillar arrays and ANSYS Fluent to simulate the fluid enclosure around the arrays.

1 Finite Element Method

Finite element method is a common numerical analysis technique used to run simulations of a variety of problems. In numerical approaches to real-life problems, various approximations and assumptions are made to reduce computation time and complexity. The basic approach to implementing FEM in simulations is given below. [4]

1.1 Discretization of the Domain

The regions of interest are divided into smaller *elements*, which can be regular or arbitrarily shaped, in 2D or 3D depending on the system. The vertices of each element are referred to as *nodes*, and each node has its own value of coordinates and variables that are used to define the system in question.
This process of dividing the domain into smaller discrete elements is known as discretization, and results in a *mesh* of finite elements. The necessary equations are solved across the boundaries of the elements.
Hence, the data after completing a simulation is in the form of nodes and their corresponding variable values.

1.2 Mesh Density

Variable values across some parts of the domain, such as fluid boundaries or edges and curves of a solid structure. Thus, for precision, it becomes necessary to have a finer mesh in there regions. In other cases, a coarse mesh is adequate and saves computation time.

2 Underlying Equations and Models

In this section, the various methods and models assumed in the simulation of nano-pillars are elaborated.

2.1 Conservation equations

To retain precision in the solutions, it is necessary to account for physical variables that must remain constant throughout the analysis. ANSYS implements the conservation of mass and momentum in all its simulations. Since the nano-pillar simulations exhibit laminar flow, conservation equations related to laminar flow in an inertial frame are laid out below.

Conservation of Mass

The continuity equation used is as follows

$$\frac{\partial \rho}{\partial t} + \nabla . (\rho \vec{v}) = S_m$$

Here, S_m is the mass added from other user-defined sources, and v is the velocity. Since the pillars are axi-symmetric, the specific equation used is given by

$$\frac{\partial \rho}{\partial t} + \frac{\partial (\rho v_x)}{\partial x} + \frac{\partial (\rho v_r)}{\partial r} + \frac{\rho v_r}{r} = S_m$$

Here, x is the axial component and r is the radial component.

Conservation of Momentum

The general equation of momentum conservation is

$$\frac{\partial (\rho \vec{v})}{\partial t} + \nabla . (\rho \vec{v} \vec{v}) = -\nabla p + \nabla . (\overline{\tau}) + \rho \vec{g} + \vec{F}$$

where p is the static pressure, $\overline{\tau}$ is the stress tensor and $\vec{F}$ is the external force, including user defined forces. The stress tensor is given by

$$\overline{\tau} = \mu \left[\left(\nabla \vec{v} + \nabla \vec{v}^T \right) - \frac{2}{3} \nabla . \vec{v} I \right]$$

where μ is the molecular viscosity and I is the unit tensor.

2.2 Fluid models

ANSYS Fluent allows the user to select the appropriate fluid model depending on the desired nature of fluid behaviour. For the current case of draining/filling up of water in an enclosure, the models used are outlined.

Standard k-epsilon model

Accurate simulations of turbulent fluid flows, especially during their interactions with solid structures, is essential for the simulation of draining water from an array of nano-pillars. The k-epsilon model is a popular choice. It is a set of two PDEs that simulate fluid flow. The 'k' here represents the kinetic energy per unit mass of the fluid. Epsilon represents the rate of dissipation of kinetic energy. The equations to calculate k and eplison respectively are as follows.

$$\frac{\partial(\rho k)}{\partial t} + \frac{\partial(\rho k u_i)}{\partial x_i} = \frac{\partial}{\partial x_i}[\frac{\mu_t}{\sigma_k}\frac{\partial k}{\partial x_i}] + 2\mu_t E_{ij} Eij - \rho\varepsilon$$

$$\frac{\partial(\rho k)}{\partial t} + \frac{\partial(\rho \varepsilon u_i)}{\partial x_i} = \frac{\partial}{\partial x_i}[\frac{\mu_t}{\sigma_\varepsilon}\frac{\partial \varepsilon}{\partial x_i}] + C_{1\varepsilon}\frac{\varepsilon}{k}2\mu_t E_{ij} E_{ij} - C_{2\varepsilon}\rho\frac{\varepsilon^2}{k}$$

Here, u is the velocity, E is the rate of deformation, μ is the eddy viscosity. The parameters σ and C are estimated from curve fitting.

Volume of Fluid model

The VOF model is used in systems that have free surface flow, wherein the parallel shear stress is zero. This simplification allows for the use of a single equation. The aim is to track momentum and calculate volume fraction throughout the fluid domain. The volume fraction of all phases in each cell in the domain must sum to unity. Consider a system with multiple fluids. The equation used for the qth phase is

$$\frac{\partial \alpha_q}{\partial t} + \vec{v}.\nabla\alpha_q = \frac{S_{\alpha q}}{\rho_q}$$

Courant number

Devised by Courant Fletcher Levi, the Courant number is an indicator of the convergent or divergent nature of the runs. It is an estimate the speed of information travelling across a computational grid. It helps to find the approximate grid and time step size of each iteration. If the number goes about the default of 250, the solution is divergent and the system is unsolved. The Courant number is calculated as shown below, for one dimensional

$$C = \frac{u\Delta t}{\Delta x}$$

where u is the wave speed of the system.

3 System Coupling

System coupling is a numerical analysis framework employed in ANSYS Workbench that allows the individual multi-physics solvers to interact with each other. At a glance, this communication is achieved by the transfer of data between the various components at the end of each time-step. The most common type of analysis carried out is that with Fluent as one of the physics solvers. Solid structures can interact with a fluid in a number of ways.

1. Structural - Stress and strain of each part of the solid as is it displaced in the fluid.
2. Thermal - Heat flows across the boundary between solid and fluid.
3. Electromagnetic - Effect on the fluid, of the generation of electromagnetic waves by the solid.

The components interact with each other using Fluid Structure Interaction, or FSI in short.
FSI simulations require the use of coupled time-steps. At each time-step, each of the components must solve the equations with the data received from the other components. The structural case is employed in the simulations conducted for this report. The results from the structural component is mapped as displacement data to the fluid. This displacement of the solid changes the fluid boundary. When Fluent solves for the deformed fluid mesh, the volume density of various mesh elements is calculated. This causes a change in the pressure force applied on different mesh elements of the solid, which causes stress and strain, thus further displacing the solid.

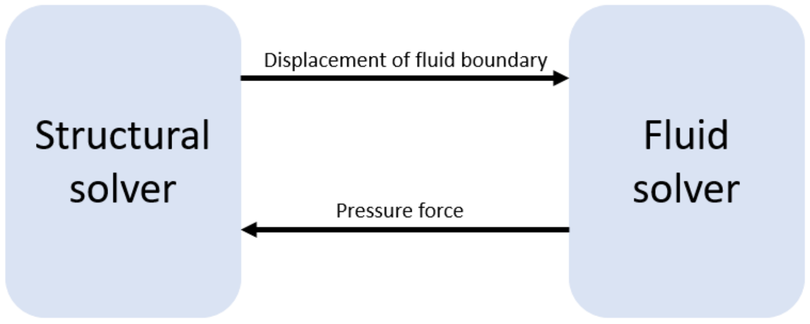


Fig 2.1. System coupling data transfer between structural and fluid solvers

At each time-step, several iterations of data transfer allow for a convergent solution.This co-simulation by combining two independent components of the workbench allow us to use various multi-physics solvers to simulate a comprehensive systems closer to those observed in real life.

4 Format of the output data from ANSYS

ANSYS Fluent relies on Finite element method to solve its equations. Hence, the variables will be given for each "element". The vertices of each element are called nodes. Each node will have a unique value of the variable in question.

ANSYS allows the user to export nodes and element data, along with any user-defined variables (such as the volume fraction, used here). Data can be exported either from CFD Post processing or during the simulation run. The former gives data only for a particular time step, while the latter gives data on-the-fly as each time step is evaluated.

Chapter 3

Image Processing tools

The coding language Python is used in our algorithm to handle the large data files from the multi-physics solvers. It's general-purpose nature ensures that the required image-processing packages are open for implementation. In this chapter, we will explore the functions and methods included in the algorithm.

1 Packages imported for data handling

The data sent from ANSYS is in the form of nodal coordinates and nodal volume fraction. Depending on the mesh density and design size, this can range from a few hundred to millions of data points. The input to the TEM simulator must be in the form of 3-dimensional arrays. Hence, it is necessary to use Python-based packages that efficiently support multi-dimensional large-scale data, along with powerful image-processing techniques.
For this purpose, the following packages are used.

1. NumPy - A scientific computing package that allows the user to handle large-scale data in the form of arrays.
2. SciPy - Includes image processing capabilities.
3. Matplotlib - A comprehensive package for plotting in Python.

2 Image Manipulation techniques

2.1 Gaussian filter

This function is a filter to be applied over an array. It can be used for multidimensional arrays as well, and uses convolution filters.
Convolution is an image processing technique in which an input array, usually 2-dimensional in connection to images, has a kernel passed over it. The kernel is a smaller array. Each element is replaced by the value obtained by placing the kernel over it and multiplying the array with the neighbours of that element, thus obtaining a weighted output of the same size as the input.

2.2 Binary opening

Binary opening is a morphological transformation, also a part of image processing. It is defined as the dilation of the erosion of the array. A user-defined structuring element is passed over the input array performing erosion and then dilation of the array, which are explained below.

Erosion

In erosion, the structure finds the local minimum over an area and replaces it with the value of the local minima. Effectively, it softens the boundary of the object and diminishes the image. Hence, the ”bright areas”, or the artefacts, get thinner.

Dilation

In dilation, the structure finds the maximal pixel value over a region and replaces the region with that value. Effectively, it makes the object more prominent. Hence,

Use in noise removal

Thus, erosion thins down the artefacts and dilation spreads the background. Together they operate as binary opening and filter the random values from the background. Erosion wears down the background noise, hence it is useful in contrasting the object. However, the object boundary wears down as well. We counter this effect using dilation, thus effectively only removing the background noise and retaining the object.

Chapter 4

Voxelating Algorithm - Pseudocode

Algorithm 1 Pseudo-code

```
GET the data values
    Input: Array of all points : $nodes$
    1. $x, y, z$ coordinates
    2. Volume fraction of water
SORT $nodes$ by $z - coordinate$
procedure FIND NUMBER OF SLICES
    Input:   nodes
    Split the coordinates by 1nm resolution
    Output: $n\_layers$ - Number of slices
end procedure
procedure GRID THE DATA
    Input:   $nodes, n\_layers$
    for $z$ in n_layers,... do
        FIND $low, high$ of the layer
        for $e$ in all nodes... do
            IF node $e$ within layer $z$
                Grid $x, y$ coordinates
                Add node $e$ to $z$th slice in $z\_stack$
                Increment corresponding $z\_nums$ coordinate
        end for
    end for
    Output: $z\_stack$ - Grid array, $z\_nums$ - Number density array
end procedure
SET $factor$ of multiplication
procedure FILTER
        Input: $z\_stack$, $z\_nums$
        procedure STRUCTURE FOR THE MASK
            Dimension $n = 4 * factor$
            CREATE circle of ones of diameter $n$
        end procedure
        procedure GAUSSIAN FILTER
                for $z$ in n_layers,... do
                    Apply mask and gaussian filter
                end for
        end procedure
        Output: $filtered$ - resultant smooth grid
end procedure
```

Chapter 5

Synopsis of the Voxelating Algorithm

1 Format of the input files

The algorithm accepts input files as and when they are sent by ANSYS. Each file is sent per time-step of the simulation. The files may be in the format of .csv, .ascii or .cdat depending on how they are exported. The file contains the x,y,z coordinates of each node and the corresponding volume fraction of water at that point.
We store the coordinates and volume fractions into arrays for convenient data processing.

In this chapter, we have used images from two different data files - one of low mesh density and one of high mesh density. Most files exported from ANSYS fall within these limits of mesh density. Hence, images from both files are compared.

2 Sorting the data into slices

The unstructured data must first be segregated into layers, or slices, due to the following reasons.

1. The TEM simulator takes structured 3D coordinates and values.
2. Layers improve visual understanding of how the solid and fluid regions are arranged, thus help us find a suitable grid size.
3. Filters are applied to 2D arrays.

The first step towards dividing the data into slices is to estimate the ideal number of slices. This is generally done along the axis of the pillars, which is the z-axis. If we divide the data into too many slices, then each slice would have only a few data points. In this case, we would not get an adequate picture of the boundaries in the slice as a whole. If we divide the data into too few slices, then some data points may coincide and we risk losing information.
Hence, it is essential to find a balance between retaining information without making the data on the slice too coarse.

Fortunately, ANSYS Fluent roughly segregates the data for its own numerical calculations.

3 Gridding the scattered node coordinates

3.1 Mesh density

Although the user can set a limit on the mesh size, ANSYS Fluent meshes the fluid enclosure with varying mesh density at different points. Near the boundaries and fluid interfaces, the mesh is very fine to allow for smoother numerical calculations. At points where there is not much variation in data values, the mesh is

coarse. This provides a challenge in forming a grid over the slice.
Consider the raw data as shown. The data is from a slice near the base of the solid pillars.

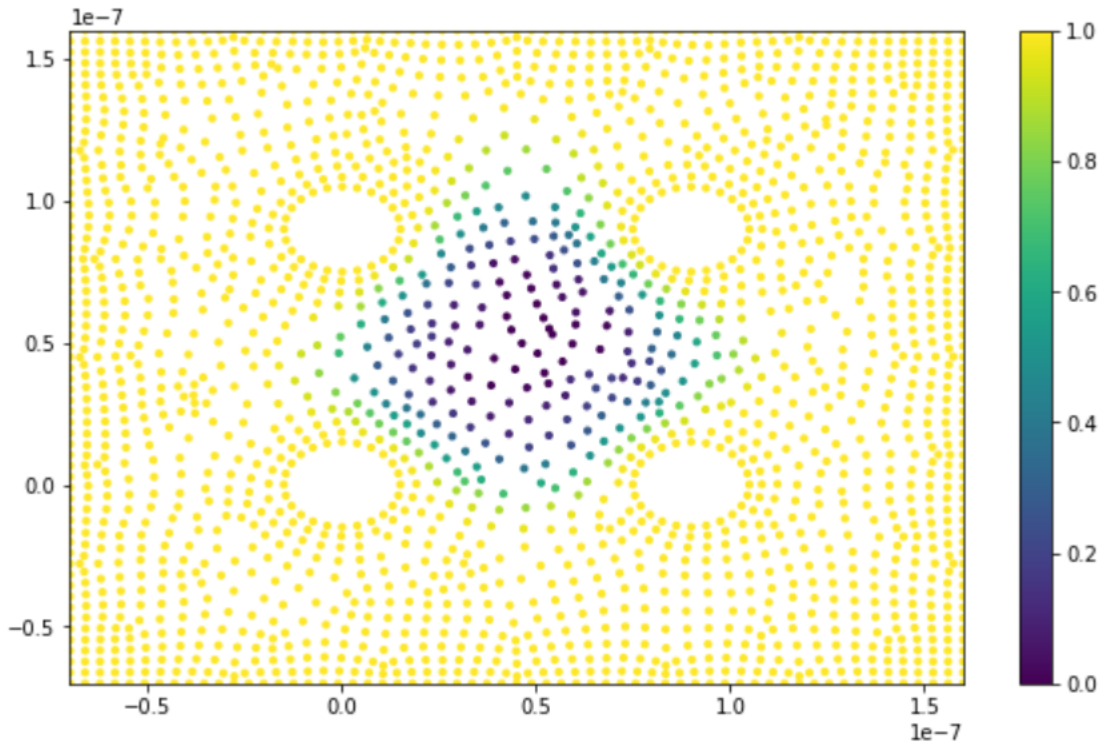


Fig 5.1. Scatter plot of raw data

As a starting step, we try to assign one pixel per point, on average. To do this, consider a regular grid of size a by b. Thus, the total number of pixels must equal ab. Since we consider one point per pixel, the total number of points in this grid is ab as well.
Now, we equate this with the number of points present in our slice, n_{points}.

$$ab = n_{points}$$

Thus, if we know :

1. Ratio of the length and width of the slice
2. Number of points on the slice

we can create a grid where the average number of points per pixel is one.
The ratio of length and width can be obtained by finding the length of the slice along x and y coordinates. This is done by subtracting the minimum value of each coordinate from the maximum. Thus, we obtain x_{length} and y_{length}.
Using this ratio, we can get the number of pixels along x and y, with the formulae

$$n_x = \sqrt{n_{points}} * \frac{x_{length}}{y_{length}}$$
$$n_y = \sqrt{n_{points}} * \frac{y_{length}}{x_{length}}$$

Thus, we have obtained a grid of pixel size n_x by n_y which has one point per pixel, on average.

3.2 Scaling the grid

Now that we have obtained an approximate grid size, we can see the effect of varying mesh density.

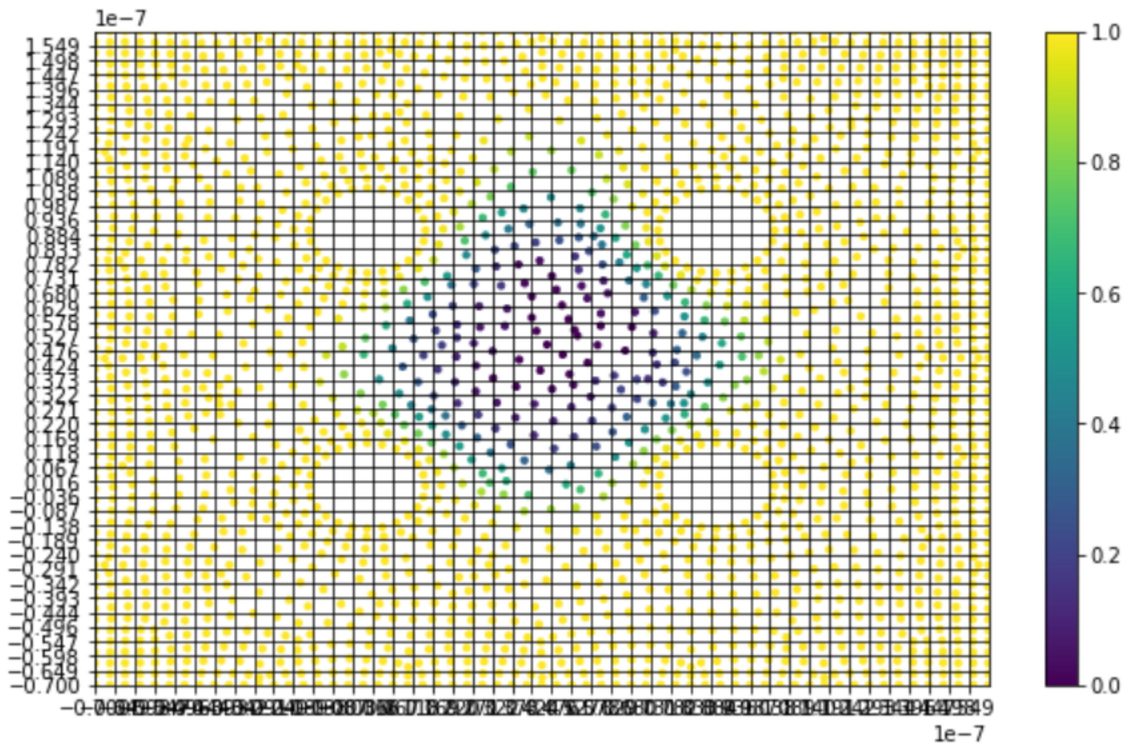


Fig 5.2. Grid scaled to 1.0 times

Low mesh density areas have empty pixels. To remove them, we can increase the pixel size (decrease the resolution). In doing so, we risk losing information about features, since there would be more points per pixel.

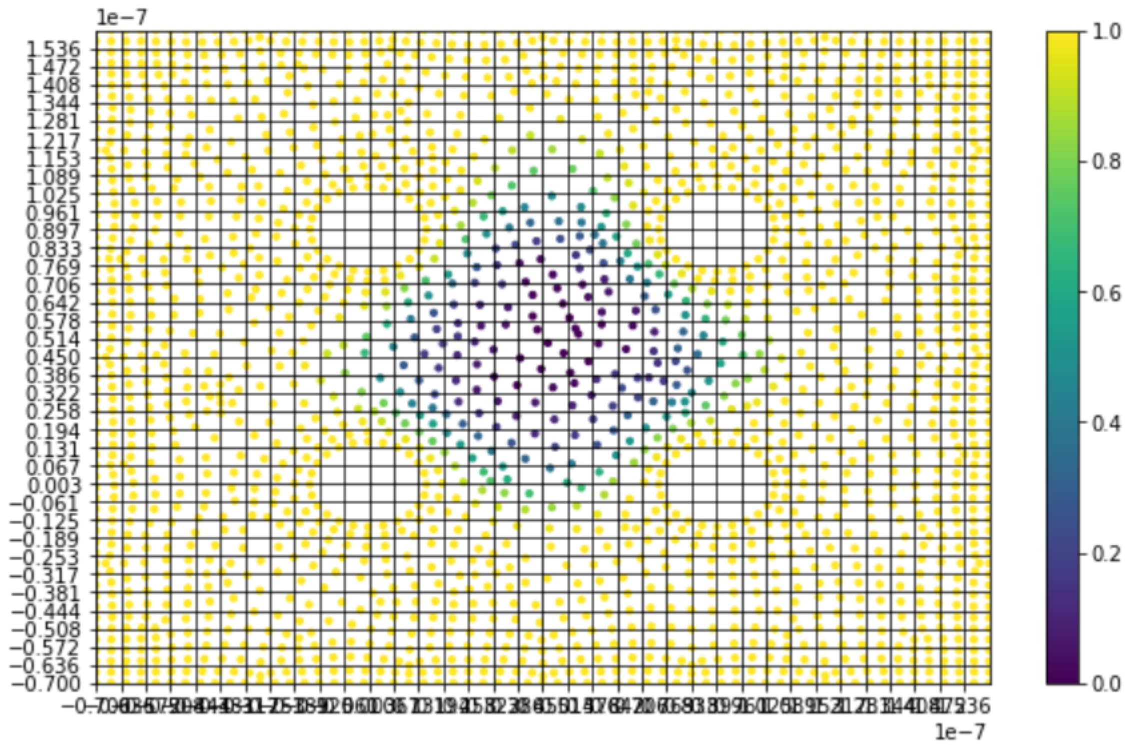


Fig 5.3. Scaled to 0.8 times. Most pixels have multiple points.

High mesh density areas have more than one point per pixel. To increase the resolution we can decrease the pixel size. However, that would create empty pixels in other regions.

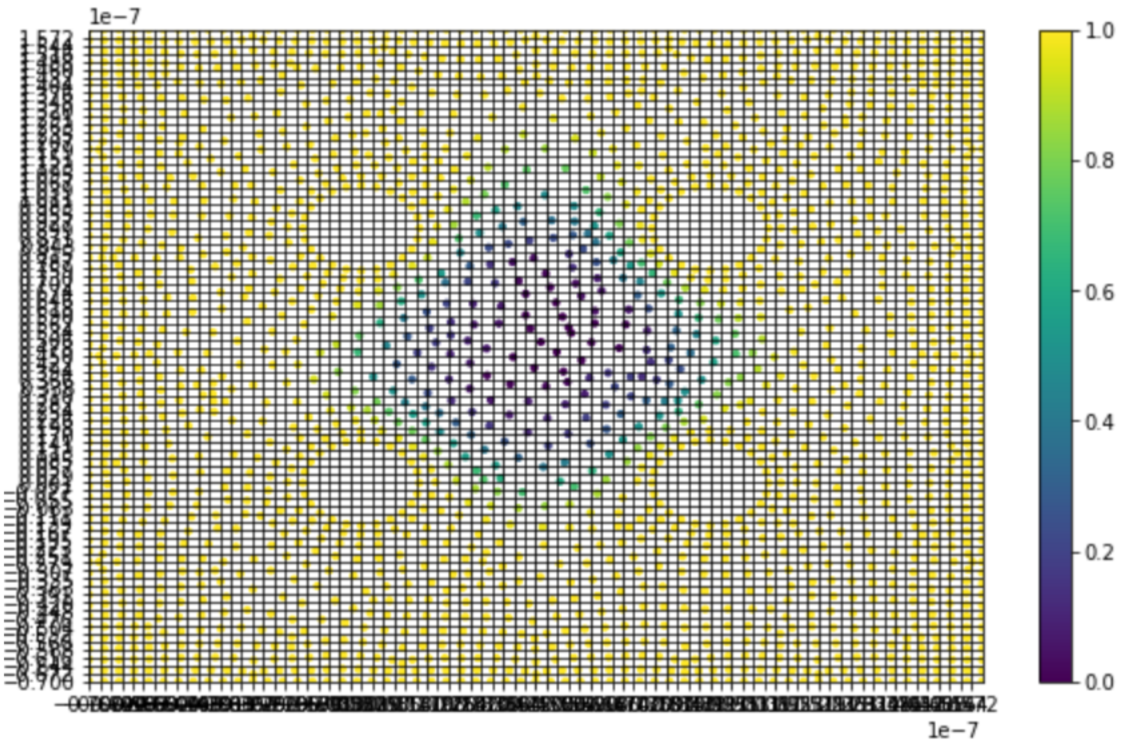


Fig 5.4. Scaled to 1.8 times. Many pixels are empty.

Empty regions are unfavourable because this could tamper with the filter application. When a Gaussian filter is applied, it averages out the empty pixels along with the filled ones. Some features may be lost with too many empty regions. Also, since the solid region is also read as an empty region, too many empty pixels will make it difficult to differentiate the solid from the background.
The current challenge is to find the optimum scaling factor for the grid, the procedure of which will be discussed.

3.3 Aligning the coordinates

Once a grid has been created, we can assign the volume fraction each point to its nearest pixel, through its coordinates. In case of there being multiple points per pixel, an average of the volume fractions is taken.
We now have a structured grid of volume fractions, with some empty pixels.

4 Extract solid from empty region

Empty pixels may represent either

1. Solid pillar

2. Regions of low mesh density

Our present task is to create a mask, by extracting just the solid pillars from the empty pixels. This mask will be the array which shows the location of the solid pillars.
To get the mask, it would be beneficial to get an idea of the distribution of density of number of points per pixel. For the grid scaled to 0.8 times and 1.8 times, this can be visualised as shown below.

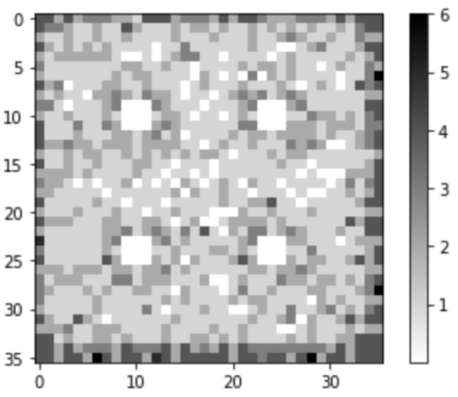
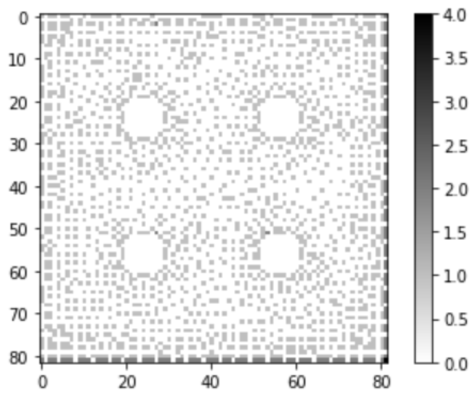



Fig 5.5.a. *Fig 5.5.b.*

a. Scaled to 0.8 times, b. Scaled to 1.8 times

The plots of point density of pixels gives a visual representation of where the empty pixels are. Clearly, when the grid size is scaled up, there are far more pixels to be occupied by the same number of points, and hence many pixels are empty (as represented by the mostly-white *Fig. b.*).
To extract only the four pillar cross-sections, we can apply a similar-sized kernel across the entire plane. This would ensure that any empty regions that are much smaller will be erased, and the solid regions will remain.
Hence, our current goal is to find a way to create a kernel of similar shape and size of the pillars, for different values of scaling factor, that we can sweep across the array to retain only the pillars.

4.1 Kernel creation

Initially, we use trial-and-error to find a suitable kernel size for each value of the scaling factor. The kernel can be represented in terms of pixels. As we scale the grid up, the solid would occupy a higher number of pixels, and hence the kernel becomes larger.
Empirically, the kernel size was found to be 4 x $scalingfactor$.

4.2 Application of mask

Now, to extract the solid pillars, we will first apply a Gaussian filter to identify the empty pixels amongst the filled pixels. Then we will apply binary opening with the kernel (the size of which is almost equal to that of the solid pillars) and retain only the solid pillars. Given below are illustrations of these two steps, comparing the arrays from data of low mesh density and high mesh density.

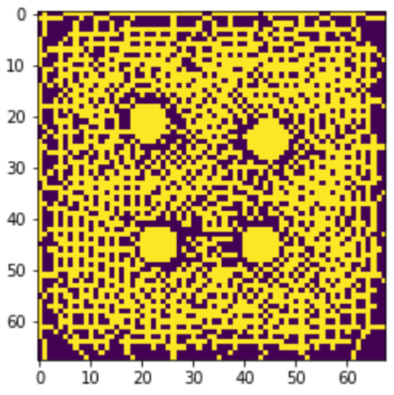
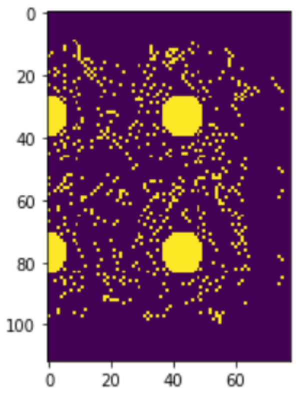
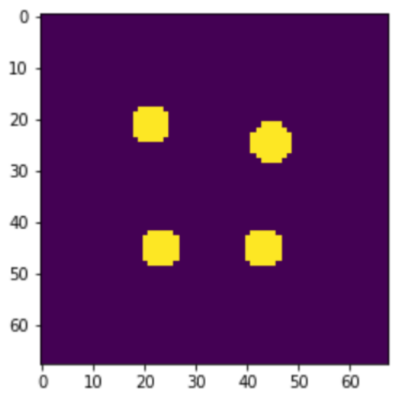



Fig 5.6.a. *Fig 5.6.b.*



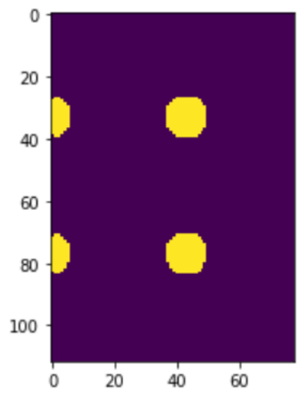


Fig 5.6.c. *Fig 5.6.d.*

description

Figures *a. and b.* represent the number density plots filtered to denote each pixel as either filled (purple) or unfilled (yellow), of the low and high mesh density cases respectively. We can see that as the mesh density increases, there is less background noise, as expected. Figures *c. and d.* represent the correspondng plots after binary opening. The background noise has been removed to retain the solid pillars.
Now that we have obtained an array containing data of the location of the solid regions, we can use it over the filtered array of the fluid region.

5 Filter the fluid data

Once the mask of the solid region is obtained, we can filter out the rest of the fluid region. This can be done with a simple Gaussian filter.
Once again, we compare the working with planes from low and high density of mesh.
Our aim is to apply a gaussian filter over the entire region, except for the solid areas (from the mask). The problem here is that the mask is an array of 0s and 1s, and the volume fraction of water ranges from 0 to 1 as well. To counter this problem, we scale up the volume factor to be stretched from 0 to 2. Then the empty regions from the mask will be denoted by 1. There will be very few points equal to 1 in the scaled up volume fraction array, since they represent the air-water boundary.
Shown below are the raw data plots, after scaling the volume fraction values. The non-mask blue pixels are empty and need to be filtered out.

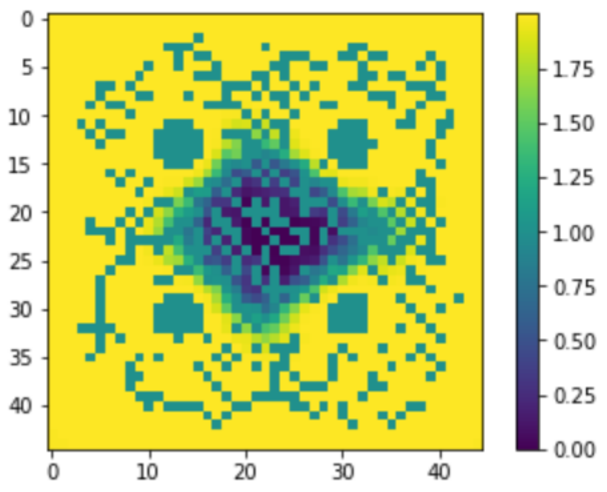


Fig 5.7.a.

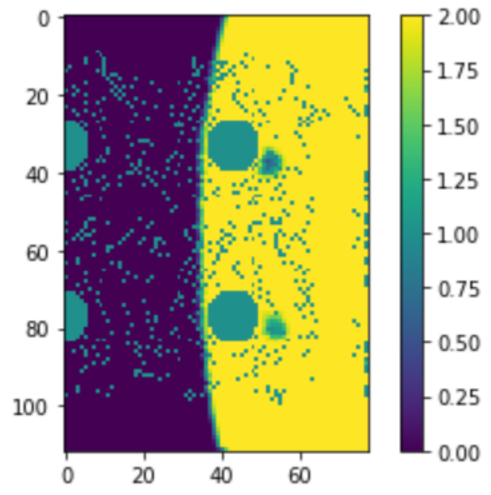


Fig. 5.7.b.

We then partition all the volume fraction into two values - 0 (for air) and 2 (for water).We do this by denoting 1 as the threshold value parameter. Now the data has a very clear-cut boundary between the two fluids. This separation is indicated by the histograms for the two mesh density cases. Clearly, almost all the points have migrated to either 0 or 2.

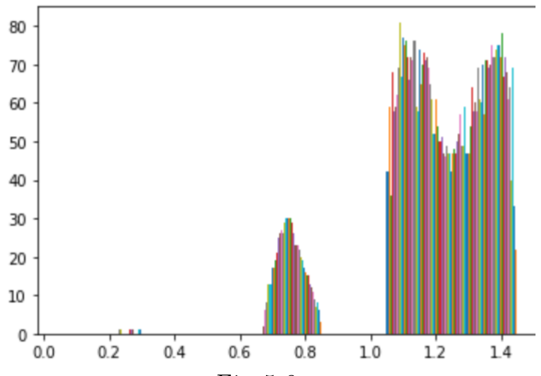


Fig 5.8.a.

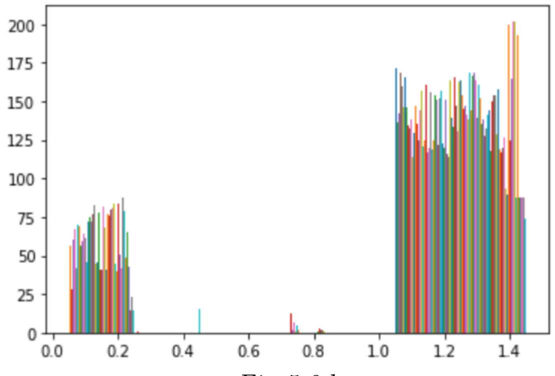


Fig 5.8.b.

After having partitioned the volume fraction values, we can apply a gaussian filter to remove the artefacts and smoothen the data out.

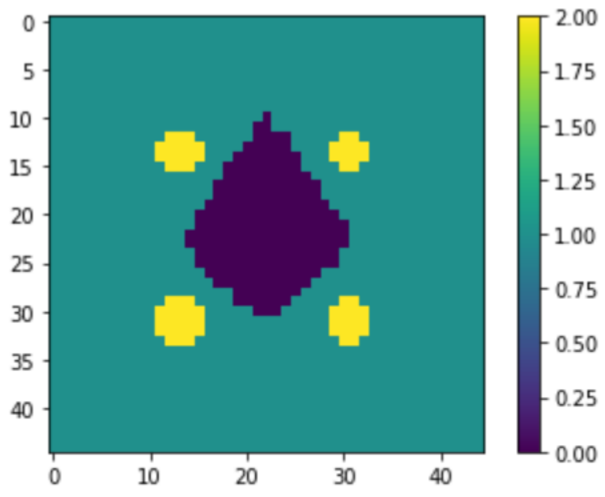


Fig 5.9.a.

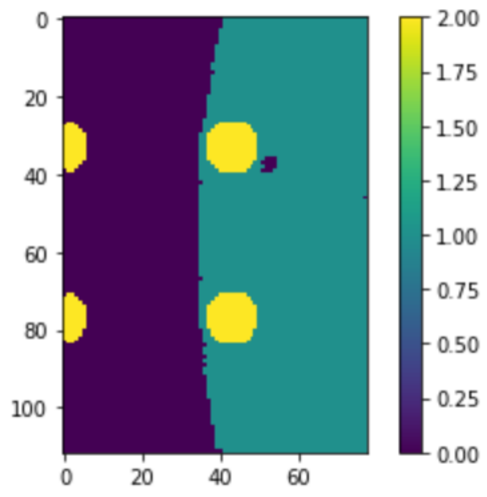


Fig 5.9.b.

Chapter 6

Parameter Analysis

The parameters used in this algorithm,in order of precedence, are

1. **Scaling factor** : For scaling the grid
2. **Sigma for the mask filter** : To remove the empty regions around the solid structures
3. **Kernel** : For creating a structure to identify the pillars
4. **Sigma for the fluid filter** : To smoothen out the fluid and remove artefacts
5. **Fluid Threshold** : To remove intermediate volume fractions and obtain only zero(air) and one(water) values

Our aim is to automate the algorithm. This involves finding the values of the parameters that would work for all, or atleast for a wide range, of nodal data.

1 Estimating optimum values of the parameters

We will go through each of the parameters and find how to either automate or optimise them.

1.1 Scaling factor

Scaling up the grid size involves finding the right balance between retaining features and minimising empty spaces. A good estimation would be to compare a case of scaling the grid down with a case of scaling it up, as shown. We will compare each of the two cases for low and high mesh density.
For the case of low mesh density, the pixel number density array is shown below. Clearly, scaling by 0.8 times decreases the resolution, and pixels have more data points.

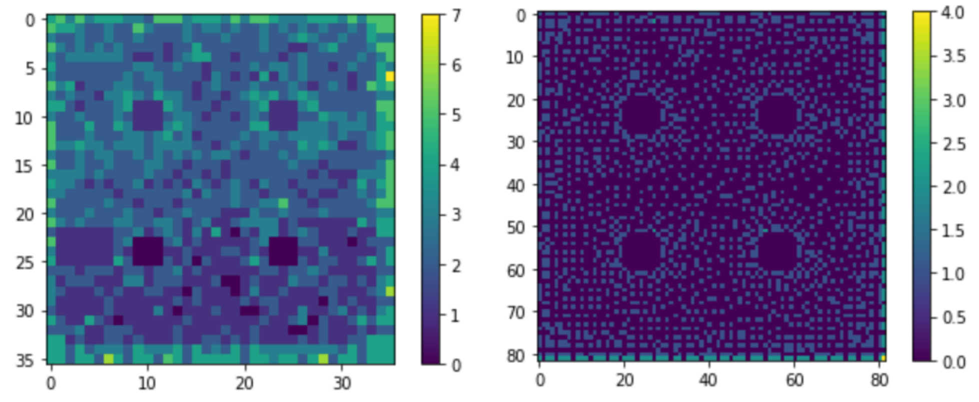


Fig. 6.1.a. and 6.1.b.

At these scales, the final outputs are shown.

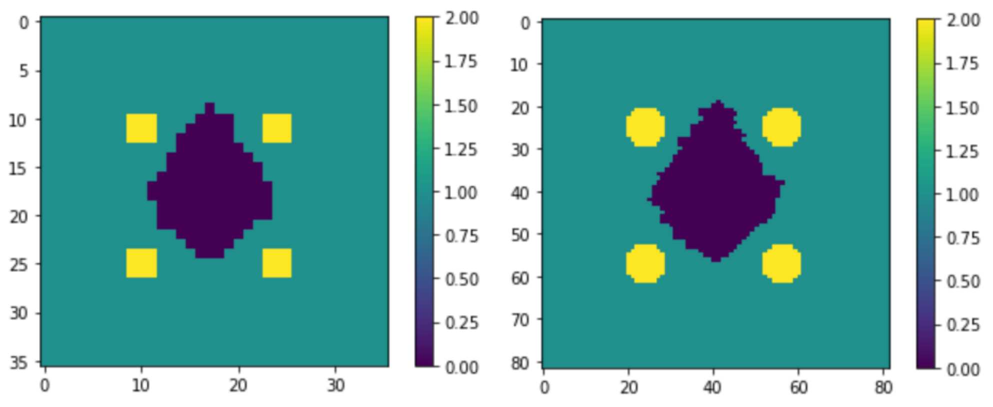


Fig. 6.2.a. and 6.2.b.

The loss of resolution by scaling the grid down is evident here. The kernel is not round, resulting in square pillars. As the resolution improves, features are not lost and pillar shape is retained.
A similar comparison is done for the case of high mesh density for scaling factors 0.8 and 1.8.

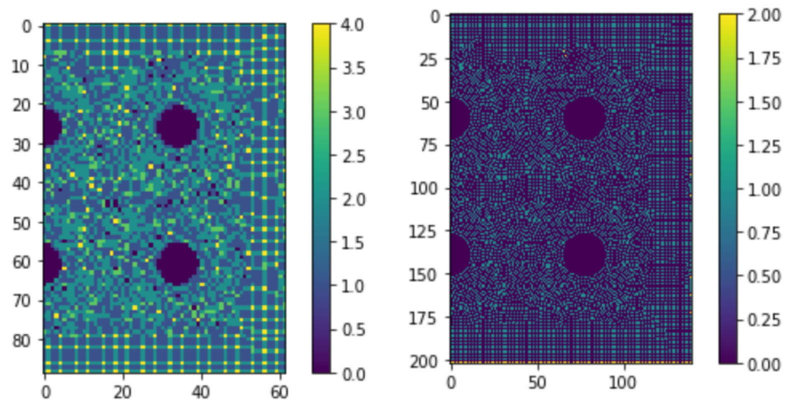


Fig. 6.3.a. and 6.3.b.

At these scales, the final outputs are shown.

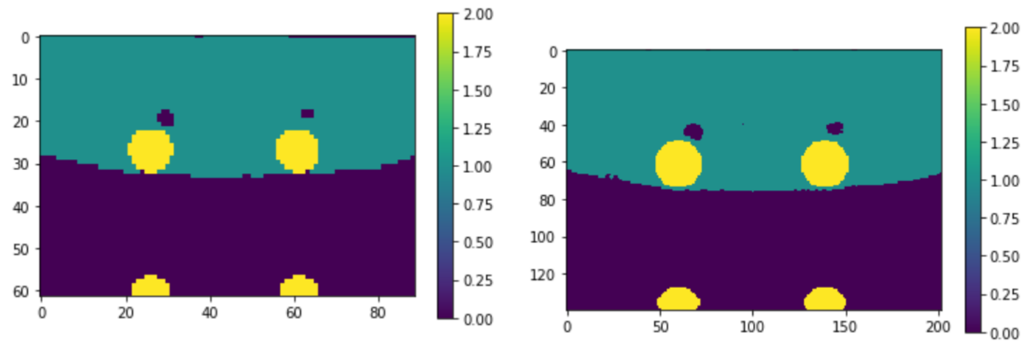


Fig. 6.4.a. and 6.4.b.

Here we can see once again that a higher value of the scaling factor results in a much smoother plot. However, increasing the factor too much could result in unnecessary increase in computation time, as well as running the risk of having too many empty regions which could be mistaken for solid pillars. Thus, ideally the grid should be scaled between 1.1 to 1.4 times.

1.2 Sigma for the mask filter

The Gaussian filter for the mask has been found to perform adequately for a wide range of sigmas, since the binary opening aids in removing the non-solid empty regions as well. Hence, its value can be anywhere between 0.0 and 0.5.

1.3 Kernel size

The kernel is a square matrix of zeros. The size has been empirically determined to be 4 x $scaling factor$. Since the pillars are circular, some zeros are replaced by ones to create the largest circle possible in this matrix.
Thus, we effectively have a circular array a bit smaller than the solid pillars. We apply this across the grid, and binary opening removes the empty spaces that are smaller than the kernel.
Thus, this parameter is automated, when the scaling factor is known.
The effect of the size of the kernel is shown below.

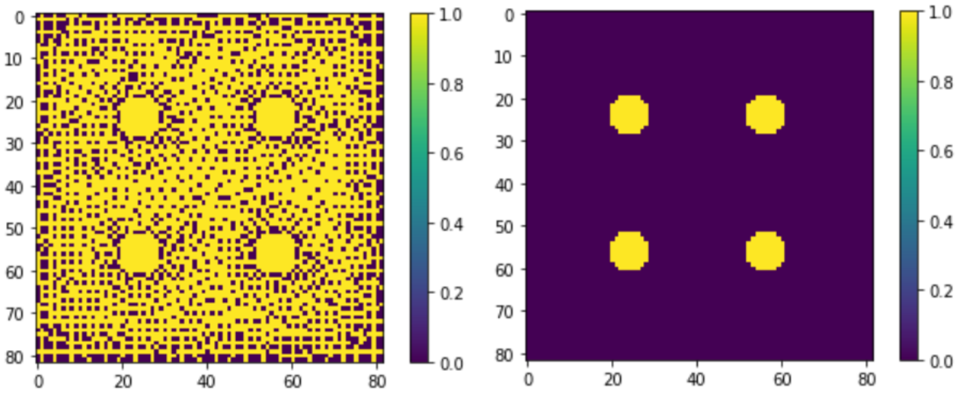


Fig. 6.5.a. and 6.5.b.

Fig. a shows the mesh density of grid where most of the pixels are empty (yellow). The scaling factor here is 1.8, hence the kernel size would be 4 x 1.8, or roughly 7. Hence, we create a circle of ones inside a 7x7 zero array. This kernel is applied across the slice to obtain *Fig. b*.
When the kernel size deviates from 7x7, we may observe the two cases below

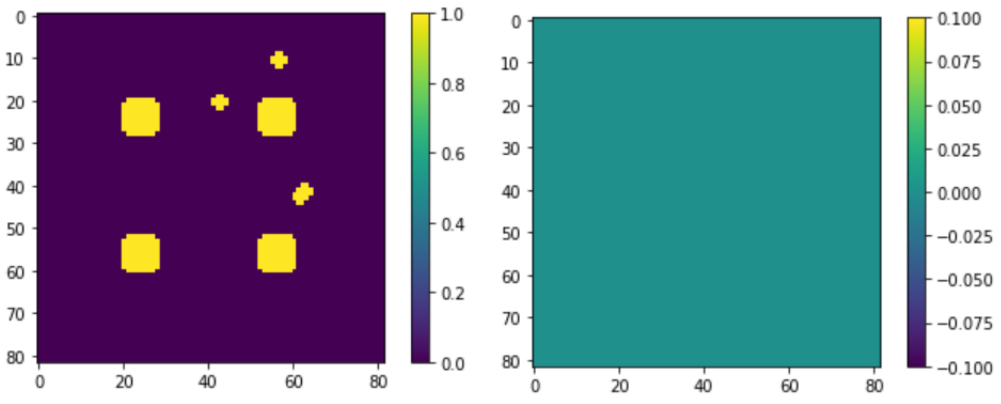


Fig. 6.6.a. and 6.6.b.

In *Fig. a*, the kernel is of size 4x4. Hence, the opening function also picks up on empty regions that are smaller than the pillars.
In *Fig. b*, the kernel is of size 10x10. Hence, the opening function being much larger than the solid pillars, reads them as noise and eradicates them as well, leaving an empty array behind.

1.4 Sigma for the fluid filter

This parameter is used to filter out the artefacts present in the non-empty pixels, which represents the fluid. A simple Gaussian filter is used. The defining parameter here is sigma, the value of which depends on the value of the scaling factor. As the factor increases, the resolution increases as well, and hence a larger pixel size is needed to filter out the holes. This means that sigma must increase with the scaling factor. However, we still face the problem of estimating the ideal combination of sigma and scaling factor. Thus, we use the ratio of total number of pixels with two data points to the total number of pixels with one data point as a measure of the average pixel density, which is expected to decrease as the scaling factor increases. Both the parameters are plotted and compared, for the cases of low mesh density and high mesh density.

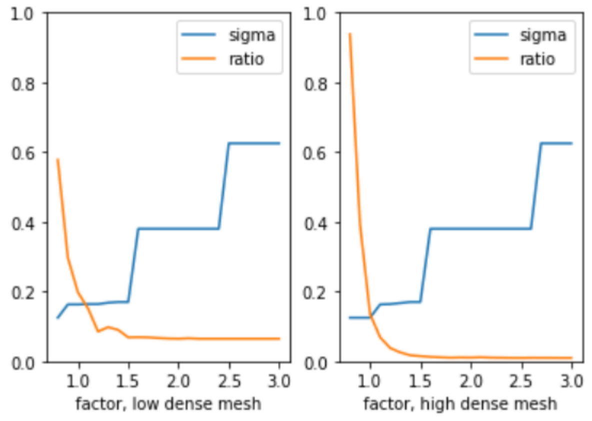


Fig. 6.7. Comparing sigma for various values of factor, low and high mesh resolution

From the above plots, we can see that for grids scaled by the same factor value, the sigma value remains roughly the same for both low and high mesh density. Thus, these values can provide a general guideline for estimating which value of sigma to use for a given value of the scaling factor.
The plot of ratio of pixels with two data points to one data point decreases and stays roughly constant, as is expected, since beyond a resolution, only empty pixels are created. The elbow points of the curve represent the best values of scaling factor to be used. Beyond these points, increasing the factor does increase the resolution but sigma also needs to be increased to result in the same final voxel array, which does not justify the extra computation time.

1.5 Fluid threshold

This parameter is essential to create a clear distinction between air and water.
Ideally, air should have a value of 0, and water, 1. However, since ANSYS Fluent is a numerical simulation software that relies on finite element analysis, boundaries tend to have intermediate values. Thus, to get clearly defined values, we segregate all points into either air or water. All volume fraction values less than 0.5 are air, while the others are assigned to water.

Chapter 7

Results

3D visualisations of the final voxel structures of both low and high mesh density are shown below. Also the data after TEM.
The final visualisation of the entire array can be viewed using a 3D projection plot. The azimuthal angle and elevation can be modified to view the structure from different points of view.
For the case of low mesh density, we have the following plots.

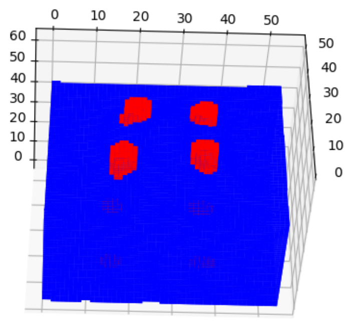
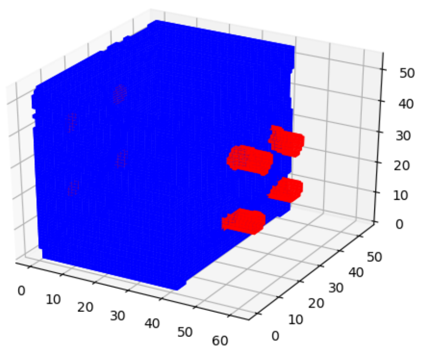



Fig 6.1.a and 6.1.b : Low mesh density

Similarly, for high mesh density, we plot the volume fraction.

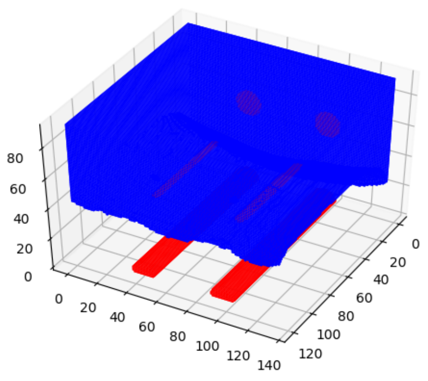
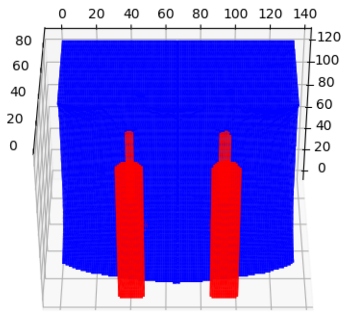



Fig 6.2.a and 6.2.b : High mesh density

For these cases, we can also view the TEM simulations after the 3d voxels data has been passed through the TEM simulator.

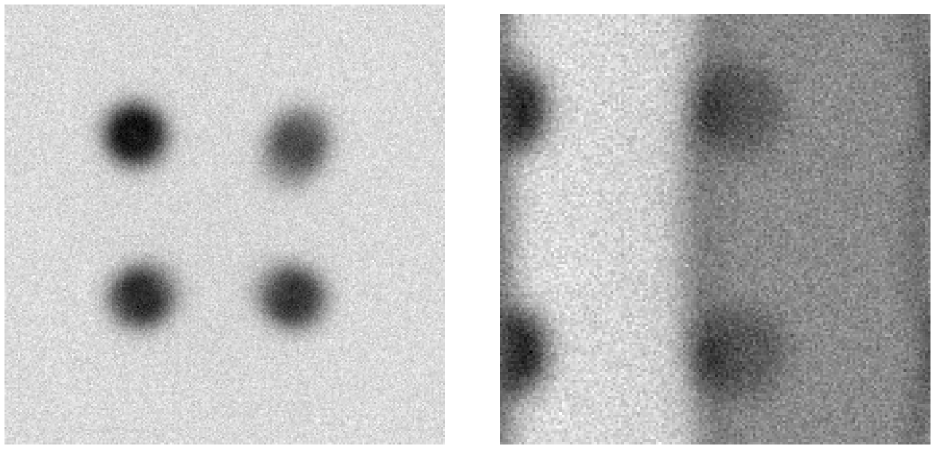


Fig 6.3.a and 6.3.b : Low and high dense mesh TEM simulator outputs.

The presence of more depth or matter in a particular region makes it darker. Hence, the pillars, which are tall, are seen to be almost black, while the fluids are lighter.
In the low mesh density case, one of the pillars is a bit lighter than the other three. This represents displacement of the top due to filling up of water.
In the high mesh density case, we can clearly differentiate between the regions with air and water.

Chapter 8

Conclusion

The voxelating algorithm introduced in this report bridges the gap between unstructured nodal data from ANSYS multi-physics simulations, and the voxelated multi-slice arrays as required by the TEM simulator.

Automation of this code ensures smooth processing of the pipeline, hence it is important to use the right set of parameter values, with which the code will run for various mesh densities of the input file.

The algorithm also applies a gaussian filter to counter the gaps in the data file. Solid pillar regions are identified prior to doing so. The algorithm relies on various image processing tools for its implementation.

The time taken by the code to run depends on the size of the input file, and can be run faster using parallel processing.

Appendix A

Computation Time

Since the voxelating algorithm is present in the middle of the pipeline, it must not cause a bottleneck in processing the data. Hence, for the data from each time-step, the voxelating algorithm must not take the most amount of time to process the data, out of all three parts of the pipeline.
The computation times taken for each of the parts, is as follows :

1. **Multi-physics Simulation** : 5 minutes for 10 time-steps of 0.08ns each.
2. **Voxelating Algorithm** : 150s for high dense mesh, 11s for low dense mesh
3. **TEM Simulator** : 100s

The multi-physics simulations and the TEM simulator take about the same time for both low and high mesh density cases. In this case, the high mesh density case runs for longer time than the TEM simulator and could hold up the data being sent. However, large data files are usually processed on multiple cores, while the above figures utilise just one core.

0.1 Parallel Processing

Increasing the efficiency of running codes involving large-scale data can be achieved using multiple nodes in a multi-core processor.
The algorithm being discussed is a perfect candidate for parallel processing since the operations are performed on each slice independently.

Appendix B

Python code of the voxelating algorithm

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


from sklearn import mixture
import scipy.stats
import matplotlib.pyplot as plt
from scipy import ndimage
import numpy as np
import csv
import math


# Read the ASCII/CSV data

# In[2]:


mesh_data = r"C:\Users\vibha\Desktop\Thesis\ANSYS practice\Pillars\Drain from sides fluent
with open(mesh_data) as csvfile:
    data_list = list(csv.reader(csvfile))
data_array = np.array(data_list[6:])


# In[3]:


nodeslist = []
vf = []
count = 0
elements_start = 0
for i in range(len(data_array)):
    if count == 0:
        nodeslist.append( [float(data_array[i][0]), float(data_array[i][1]), float(data_ar
        if data_array[i+1] == []:
            count = 1
nodes = np.array(nodeslist)


# In[4]:


nodes = nodes[nodes[:,3].argsort()]


# Processing the data

# In[5]:


def gridify(coords, spacing):
    np.array(coords)
    return np.round(coords / spacing) * spacing


# In[6]:


k=np.zeros(len(nodes))
init=0
i=2
while i<len(nodes)-1:
    if(nodes[i+1,3]-nodes[i,3]>1e-9):
        i+=1
        init=i
        k[init:]+=1
        i+=1

    i+=1
n_layers = len(np.unique(k))


# In[7]:


factor = 1.2


# In[8]:


minx = min(np.array(nodes)[:,[1]])
maxx = max(np.array(nodes)[:,[1]])
miny = min(np.array(nodes)[:,[2]])
maxy = max(np.array(nodes)[:,[2]])

base = np.min(nodes[:,[3]])
top = np.max(nodes[:,[3]])
```

```
# In[9]:


n = len(nodes)/n_layers #approximate number of points per layer
xlen = maxx - minx
ylen = maxy - miny
ratio = xlen/ylen

nx = int((np.sqrt(n*ratio))*factor)
ny = int((np.sqrt(n/ratio))*factor)

deltax = (maxx-minx)/nx
deltay = (maxy-miny)/ny

deltaz = (top-base)/n_layers


# In[10]:


planes = [[] for i in range(n_layers)] #to store data of the planes before regularising by
z_stack = np.empty((nx, ny, n_layers))
z_stack[:] = 1
z_nums = np.empty((nx, ny, n_layers))

for z in range(n_layers):
    low_z = base+z*deltaz
    high_z = base+(z+1)*deltaz
    for e in range(len(nodes)):
        if int(k[e])==z:
            #pre-regularised planes
            planes[z].append((nodes[e,1],nodes[e,2],nodes[e,4]))

            #regularised planes
            coord = [gridify(nodes[e,1],deltax),gridify(nodes[e,2],deltay)]
            i = int(np.round((coord[0]-minx)/deltax)) -1
            j = int(np.round((coord[1]-miny)/deltay)) -1
            z_stack[i][j][z] = (z_nums[i][j][z]*z_stack[i][j][z] + nodes[e,4])/(z_nums[i][
            z_nums[i][j][z] += 1



# Analysing the processed data

# In[26]:


#get the kernel for the mask. Input, n - related to pillar diameter.
n = int(4*factor)
r = math.ceil(n/2)
struct = np.zeros((n,n))
for i in range(n):
    for j in range(n):
        if n%2==0:
            if (i+0.5-r)**2 + (j+0.5-r)**2 <= n*n/4:
                struct[i,j] = 1
        else:
            if (i+1-r)**2 + (j+1-r)**2 <= n*n/4:
                struct[i,j] = 1


# In[24]:


filtered = [[] for i in range(len(planes))]
for i in range(len(planes)):
    pl_array = z_stack[:,:,i]
    pl_array_msk=1*(ndimage.gaussian_filter(z_nums[:,:,i],0.1)<0.6)
    pl_array_msk_open = scipy.ndimage.binary_opening(pl_array_msk, structure=struct)
    pl_conv=ndimage.gaussian_filter(pl_array,1.0)>1.0
    pl_multilayer2=((pl_conv)*1)*(pl_array_msk_open==0)+(pl_array_msk_open==1)*2
    filtered[i]=pl_multilayer2
```

References

1. Xie, Chong and Hanson, Lindsey and Xie, Wenjun and Lin, Ziliang and Cui, Bianxiao and Cui, Yi*Noninvasive Neuron Pinning with Nanopillar Arrays*
doi:10.1021/nl101950x

2. Fan, Zhiyong and Razavi, Haleh and Do, Jae-Won and Moriwaki, Aimee and Ergen, Onur and Chueh, Yu-Lun and Leu, Paul W. and Ho, Johnny C. and Takahashi, Toshitake and Reichertz, Lothar A. and et al.*Three-dimensional nanopillar-array photovoltaics on low-cost and flexible substrates*
doi:10.1038/nmat2493

3. Reimer, Ludwig and Kohl, H. *Transmission electron microscopy: physics of image formation*

4. Alawadhi, Esam M.*Finite Element Simulations Using ANSYS*
doi:10.1201/9781439801611