PixelBased Permeability Documentation

Andre Adam

Last Updated: August 6, 2024

Contents

1 Introduction

This is documentation on how to use the algorithm for predicting the permeability of 2D binary structures. This documentation will go over the requirements for running the code and basic instructions on how to use it. This document also includes more extensive documentation on the FVM formulation, inherent model assumptions, as well as technical implementation details.

The document will go over how to run the code first, as it assumes the user has already read the publication (**not yet available**) and is familiar with the physical and numerical models. If you are not familiar with the physical and numerical models, then read the formulation first, then the inputs in the input file will make more sense.

Note on the CPU version: Due to being much slower and inefficient compared to the GPU version, the CPU version has lagged behind in development. The GPU version is highly recommended. If you wish to use the CPU version and are unsure about it, contact one of the authors.

2 Requirements

To run the code, the following files are necessary in the same folder:

```
Perm2D.cpp
Perm2D.h
stb_image.h
input.txt
AnyImage.jpg
```

There are three versions of the code: the CPU version, the GPU version, and the high-performance computing (HPC) version. The CPU version uses the .cpp and .h files, while the others use .cu and .cuh files. The GPU and HPC versions currently only work with CUDA capable GPUs.

The input file must be named input.txt, and in this document we will go over a sample input file, specifications, intricacies, and how to best use it.

Important note: the GPU and HPC versions of the code have been more extensively tested than the CPU version. Therefore, it is possible the CPU version will have results that are not consistent with the other two versions or show new bugs under certain circumstances. In general, it does perform like the other versions.

2.1 Compiling

Compilation of the code is only done once, but it must be done. If the code is modified, then the user must compile again. If you alter the input file, then there is no need to compile again. From the command line on Ubuntu, the following command will compile file into a executable:

```
CPU:
g++ -fopenmp Perm2D.cpp
GPU:
nvcc Perm2D.cu
HPC:
nvcc -Xcompiler -fopenmp Perm2D.cu
```

Note that on Windows and/or with different compilers there might be different calls to load the OpenMP package, so stay alert for those. The compilation assumes the command is running in the same folder as the other files. After compilation, just run the executable either via command line or GUI.

2.2 Image Requirements

The image must be a single channel (**NOT** RGB) grayscale .jpg image for the algorithm to correctly run. If the image has any number of channels other than 1, the code returns indicating an error message telling the user to set the image to 1 channel. The file *stb_image.h*, which was sourced from open source project, is responsible for reading the specified image.

This image should have two phases: black is fluid phase, and white is solid (impenetrable) phase. As such, there should be a path connecting the left and right boundaries through the fluid, otherwise the permeability is just zero and there is no need to run the simulation.

3 Input.txt

In this section we take a look at a sample .txt file, and how to use it. The first line of the file is optional, but the name of the file must be "input.txt", verbatim. While there are three different versions of the code, the input file is the same for all three versions, except some arguments are interpreted differently in each version of the code or they are completely ignored. This section will go over all of it in details.

It must also be noted that the labels have to be exactly as shown below. The code does differentiate from lower and uppercase letters. The order in which the arguments are entered is not important. As a final note, there must always be a space after the ":" and the numbers or filenames.

```
Input File:
DomainHeight: 0.1
DomainWidth: 0.1
Dens: 1000
Visc: 1e-3
MeshAmp: 1
InputName: 00000.jpg
PL: 100
PR: 99.99
RelaxFactor: 0.1
OutputName: QSGS_GPU_Batch.csv
printMaps: 1
MaxIterGlobal: 5000
ResidualConv: 1e-11
MaxIterSolver: 100000
SolverConv: 1e-7
nCores: 4
nImg: 300
Verbose: 0
```

3.1 Domain Height and Width

This input takes the physical height and width of the domain in units of meters. In the example file, the height and width are $0.1m$.

3.2 Dens

Dens is the density of the fluid, entered in units of kg/m^3.

3.3 Visc

Visc is the dynamic viscosity, μ, in units of $Pa \cdot s$.

3.4 MeshAmp

This input is interpreted as and integer and cannot be negative. It determines how much the mesh will be increased from the pixels of the original image.

Example: if we originally have a 100×100 image, setting the MeshAmp = 2 will return a 200×200 mesh, a 4-fold increase in the number of voxels.

3.5 InputName

Verbatim name for the input image. Must be single channel grayscale .jpg image. In the HPC version, this input is ignored (read nImg for HPC version image naming convention).

3.6 PL and PR

PL and PR set the pressures at the two boundaries. We assume PL > PR, but the code should run fine without that assumption, it will just return the permeability and flow rate values as negative.

3.7 RelaxFactor

This input controls the value of the relaxation factor for the pressure-velocity coupling.

3.8 OutputName

CPU and GPU version: The output name controls the name of the output file that contains the velocity and pressure maps, if the flag "printMaps" is set to true. Otherwise, it does nothing.

HPC version: This controls the name of the output file containing the permeability and other information regarding simulation inputs and outputs. This option is never ignored in the HPC version.

3.9 printMaps

The input "printMaps" controls whether the pressure-velocity maps will be printed or not. The two possible inputs are 0 for not printing and 1 for printing.

3.10 MaxIterGlobal

The code runs on the standard under-relaxation of the consistent update technique (SUV-CUT) pressure-velocity coupling. This option controls the maximum number of iterations before the code gives up on converging. Typically, the SUV-CUT algorithm takes less iterations to converge when compared to a SIMPLE-like algorithm.

3.11 ResidualConv

This controls the residual convergence criteria for the SUV-CUT (for the algorithm, refer to section 4.6.

3.12 MaxIterSolver

This controls the maximum number of iterations for each call to the implicit solver (once every SUV-CUT iteration).

3.13 SolverConv

The option controls the convergence criteria for the implicit solver.

3.14 nCores

CPU: Controls the number of cores to be used by OpenMP. These resources are used in a shared-memory environment, and are used to speed up the implicit solver. Note that OpenMP does not return an error if the user enters a number of cores greater than the amount available, and that can cause some slowing down. Make sure the OpenMP environment is set properly in the command line and you don't call more cores than are available.

GPU: This option is ignored.

HPC: This option controls the number of GPU-CPU pairs. That means if there are 4 GPUs available, the user should have at least for CPU cores available. More on how the code takes advantage of the additional resources is included in the implementation details.

3.15 nImg

CPU and GPU: this option is ignored.

HPC: This sets the number of simulations to run. The HPC version of the code is primed for running batches of simulations, and this option controls the batch size.

3.16 Verbose

The option verbose is a flag that controls printing to the console. With verbose set to 1, several things are printed to the command line/console while the code is running. If verbose is set to 0, then the code runs without printing anything. The output files are separate from this, and are not affected by this flag.

While for a single image this has a negligible impact on speed, for a batch of images on HPC or when saving the command line output, it can be non-negligible in terms of speed and storage. Therefore, it is suggested to set Verbose= 0 in HPC mode.

4 Computational Model

4.1 Assumptions

The following are the assumptions embedded into the CFD simulation and permeability calculation:

- Slow, creeping flow ($Re < 1$).
- Neglected viscous dissipation.
- No slip boundary between solid and fluid.
- Incompressible fluid.

4.2 Governing Equations

Viscous dissipation and incompressible flow are assumed, so the energy equation and/or any equations of state (for adjusting density, ρ and viscosity, μ) are neglected. The assumption of slow, creeping flow also suggests the existence of a steady-state flow regime, where there are no changes to the flow rate through the domain over time. The simplified version of the continuity and momentum equations in differential form are shown in equations (1) and (2).

$$\rho \nabla \cdot \left(\overrightarrow{\boldsymbol{u}} \right) = 0 \tag{1}$$

$$\rho \left(\overrightarrow{\boldsymbol{u}} \cdot \nabla \right) \overrightarrow{\boldsymbol{u}} = -\nabla P + \mu \nabla^2 \overrightarrow{\boldsymbol{u}} \tag{2}$$

Note that equation (2) will actually break down into a x and y momentum equations. The computational model comes down to three equations and three unknowns, P, u, and v. While continuity and momentum are different equations, they can be put into the same form [1]. It is common to introduce a general variable, ϕ, to denote the property being solved for, thus the desired shape of the equations is:

$$\rho \frac{\partial \phi}{\partial t} + \rho \left(\phi \cdot \nabla \right) \boldsymbol{u} = \Gamma \nabla^2 \boldsymbol{\phi} + S_\phi \tag{3}$$

From equation (3), each term has a physical meaning. The first term is the rate of increase of property ϕ with respect to time, the second term is the net flow of ϕ out of the fluid element. On the right hand side, the first term the rate of increase in ϕ due to diffusion, and S_ϕ is defined to account for sources or sinks of ϕ. Equation (3) is commonly known as the transport equation. Since the permeability prediction happens at a steady-state, the first term on the left hand side goes to zero, hence it is dropped from the formulation. By setting ϕ equal to 1, $\boldsymbol{u}$, or $\boldsymbol{v}$, and setting the appropriate values for $\boldsymbol{S}_\phi$ and Γ, equation (3) becomes equations (1) and (2).

In the finite volume method (FVM), the key in solving equations (1) and (2) is to integrate them over a 3D control volume (CV). In this work, the control volumes are 2D, with an arbitrary depth applied to them. That is to ensure there is no flow through the third dimension, as the problem assumes only 2D flow. The integration of (3) over the control volume, and dropping the time dependent term yields (4):

$$\int_{CV} \rho\left(\boldsymbol{\phi} \cdot \nabla\right) \boldsymbol{u} dV = \int_{CV} \Gamma \nabla^2 \boldsymbol{\phi} dV + \int_{CV} \boldsymbol{S}_\phi dV \tag{4}$$

The term for net flow out of the CV and the term for diffusive flow into the CV can be re-written using Gauss's divergence theorem, meaning the flow rates can be expressed as an integral over the surface of the control volume. The steady-state form then takes the following shape:

$$\int_A \boldsymbol{n} \cdot \rho\left(\boldsymbol{\phi} \boldsymbol{u}\right) dA = \int_A \boldsymbol{n} \cdot \Gamma \nabla \boldsymbol{\phi} dA + \int_{CV} \boldsymbol{S}_\phi dV \tag{5}$$

4.3 Grid

Equation (5) is the basis equation for the FVM formulation. A grid needs to be defined such that equation (5) can be discretized for a domain. While it is not necessary to define every single detail about the grid at this stage, we will take the opportunity to introduce the staggered grid.

Typically, when coupling pressure and velocity, special attention must be directed towards avoiding non-physical behavior of "checker-board" pressure fields. For more detail, refer to chapter 6 of Versteeg [1]. There are other ways of dealing with this issue, such as Rhie and Chow interpolation [2], but the Ghia [3] staggered grid (or multi-grid) approach is a simple and portable solution for a 2D simulation. It requires a structured grid, an assumption that is met by using the pixel resolution as the base grid (it is naturally a regular structured grid).

In the staggered grid approach, the pressure grid is taken as the base pixel resolution of the image. The pressures are defined at the center of each node. Then, the velocity grids are defined at the edges of the node. For a pressure $P_{i,j}$, defined in the i^{th} column and j^{th} row of nodes, the u-velocity is defined at $u_{i-\frac{1}{2},j}$, which is offset by $\frac{-\Delta x}{2}$ from the center of the node, where Δx is the size of the node. Similarly, the v-velocity is defined at $v_{i,j-\frac{1}{2}}$, $\frac{-\Delta y}{2}$ from the center of the node. It becomes clear that, for a grid with $n \times m$ nodes, there will be $n \times m$ pressures, $n+1 \times m$ u-velocities, and $n \times m+1$ v-velocities.

In terms of notation, there are three options typically employed: the first is to carry the fractions $\pm\frac{1}{2}$. Second, is to define a grid with capital I, J defined at the center of the nodes, and a grid with lowercase i, j defined at $i = I - \frac{1}{2}$ and $j = J - \frac{1}{2}$. The third option is to define a central pressure at a location P, and the adjacent pressure nodes are E, W, S, and N, respectively in the cardinal directions. With the lowercase e, w, s and n are defined the interfaces of node P. In other words, the lowercase indexes are half-steps away from P, while the uppercase indexes are full-steps away from P. The last notation is more compact, and therefore will be adopted from now on.

4.4 Integral over Control Volume

For the x-momentum equation, it is easy to show that $\phi = u$ and $\Gamma = \mu$. Hence, equation (2) takes the following shape:

$$\rho\left(u \cdot \nabla\right) \boldsymbol{u} = \mu \nabla^2 u + S_\phi \tag{6}$$

The pressure term present in equation (2) gets incorporated into S_ϕ. With the grid defined and the momentum equation in the correct shape (6), equation (5) can be integrated over the control volumes. This will yield equation (7).

$$a_e u_e = \sum_{nb} a_{nb} u_{nb} - \frac{P_P - P_W}{\Delta x} \Delta V_u + \bar{S_\phi} \Delta V_u \tag{7}$$

Equation (7) can be further simplified as follows:

$$a_e u_e = \sum_{nb} a_{nb} u_{nb} - \left(P_P - P_W\right) A_e + b_e \tag{8}$$

In equation (8), the coefficient a_{nb} are the collection of all coefficients multiplying u_{nb}. The nb subscript refers to any neighboring cell to u_e. These coefficients depend on the discretization scheme, more on that in section 4.5. The central coefficient, a_e, is the sum of all neighboring coefficients. Finally, A_e is the cross-sectional area of the current u-grid element, and b_e is the source term. Note how from equation (7) to equation (8) the source term notation was shifted from $\bar{S_\phi}$ to b_e. The new notation, b_e, is a more generic notation, and encompasses all "floating" terms that do not fit into the format of the transport equation, all volumetric effects (which can include body forces), and all source terms applicable to the current grid.

4.5 Discretization

While there are many options for discretization schemes [1], this work currently employs the hybrid discretization scheme, from Spalding [4]. The hybrid discretization is a central differencing scheme for Peclet numbers, Pe, less than 2, and a first order upwind for $Pe \geq 2$. The Pe is evaluated at the cell faces (where velocities are defined) as follows:

$$Pe = \frac{\rho u_w}{\frac{\mu}{\Delta x}} \tag{9}$$

The general hybrid discretization takes the following shape:

$$a_P \phi_P = a_W \phi_W + a_E \phi_E + a_N \phi_N + a_S \phi_S + \Delta F \tag{10}$$

Note that equation (10) is written in general terms, and the notation is not to be confused with equation (8). In fact, a_e from (8) is equal to a_P from (10), and a_{nb} are the right-hand side coefficients of equation (10). From an implementation perspective, Pe from equation (9) doesn't need to be calculated. Instead, the coefficients can be defined as follows:

$$\begin{aligned}
a_W &= max\left[F_W, D_W + \frac{F_W}{2}, 0\right] \\
a_E &= max\left[-F_E, D_E - \frac{F_E}{2}, 0\right] \\
a_S &= max\left[F_S, D_S + \frac{F_S}{2}, 0\right] \\
a_N &= max\left[-F_N, D_N - \frac{F_N}{2}, 0\right] \\
\Delta F &= F_E - F_W + F_N - F_S \\
a_P &= a_W + a_E + a_S + a_N + \Delta F
\end{aligned} \tag{11}$$

One major factor is still missing: the definition of the coefficients depend on two variables, F and D, which have not been defined yet. This is the final bridge in connecting the governing equations to the discretized form shown in equation (8). Formally, F is physically interpreted as the convective mass flux per unit area, and D is the diffusion conductance at cell faces.

To derive these coefficients, it is more descriptive to adopt the notation with lowercase and uppercase i's and j's for u and v, but will retain the uppercase cardinal notation for F's. In the context of the uniform staggered grid, and assuming incompressible flow with constant ρ and constant μ, these terms take the following form in the x-momentum equation:

$$\begin{aligned}
D &= \frac{\mu A}{\Delta x} \\
F_W &= \frac{\rho A}{2}\left(u_{i,J} + u_{i-i,J}\right) \\
F_E &= \frac{\rho A}{2}\left(u_{i,J} + u_{i+i,J}\right) \\
F_S &= \frac{\rho A}{2}\left(v_{I,j} + v_{I-1,j}\right) \\
F_N &= \frac{\rho A}{2}\left(v_{I,j+1} + v_{I-1,j+1}\right)
\end{aligned} \tag{12}$$

The coefficients from (12) are directly applied to equation (8), and the exact same process can be applied to the v-velocity discretization. Note how A, ρ, and μ are constants, which significantly simplifies the equations [1].

4.6 Pressure-Velocity Coupling

From section 4.5, the momentum equations have been discretized, but there are still 3 unknowns for 2 equations, and there are many avenues for incorporating the continuity equation (1) to solve for pressure. However, due to the non-linear relationship between pressure and velocity, it is not as simple as solving a linear system with 3 equations, and therefore there is a need for special mechanism to address the nonlinearity at hand.

By far the most common method is the Semi-Implicit Method for Pressure-Linked Equations (SIMPLE) algorithm, put forth by Patankar and Spalding [5], which is used as basis for a family of algorithms (the SIMPLE-like algorithms), the most prominent of which are the "revised" version (SIMPLER) [6], and the "consistent" version (SIMPLEC) [7].

Most recently, a new algorithm, the consistent update technique (CUT) [8], has emerged as an alternative to the SIMPLE-like algorithms with faster convergence in several benchmark tests. Furthermore, another work [9] identified a better way to introduce relaxation factors to the original CUT algorithm, and called it the standard under-relaxation version of the consistent update technique (SUV-CUT), and that is the formulation that is adopted for pressure-velocity coupling in this work. This method is chosen because of the advantage in convergence speed and the ease in implementation.

The basic idea of the SUV-CUT algorithm is to, at any iterative step, match the momentum equations explicitly and the continuity equation implicitly. Equation (8) can be re-written generically for any iterative level with the under-relaxation, α, included as follows:

$$u_e = (1-\alpha)\, u_e^0 + \frac{\alpha}{a_e}\left(\sum a_{nb} u_{nb}^0 + b\right) + \frac{\alpha A}{a_e}\left(P_P - P_E\right) \tag{13}$$

The superscript in u_e^0 indicates that the u values are from the previous iterative step (or, in the first iteration of the algorithm, u^0 is the initial guess at the solution). Then, it is convenient to define the explicit velocity, $\tilde{u_e}$, as the velocity component that is solely dependent on the values of velocity calculated at an earlier iterative step and material properties.

$$\tilde{u_e} = (1-\alpha)\, u_e^0 + \frac{\alpha}{a_e}\left(\sum a_{nb} u_{nb}^0 + b\right) \tag{14}$$

The computation of (14) is completely explicit, and similarly for $\tilde{v_e}$.

Substituting velocities from equation (13) into continuity equation (1), an implicit equation for pressure is obtained in the following format:

$$a_P P_P = a_E P_E + a_W P_W + a_S P_S + a_N P_N + b_P \tag{15}$$

In equation (15), none of the pressure values are known, therefore the equation is arranged in a linear system of the form $A\boldsymbol{x} = \boldsymbol{b}$. From the substitution, the coefficient that arise take the following shape:

$$\begin{aligned}
a_E &= \frac{\rho A^2}{a_e} \\
a_W &= \frac{\rho A^2}{a_w} \\
a_S &= \frac{\rho A^2}{a_s} \\
a_N &= \frac{\rho A^2}{a_n} \\
b_P &= -\left[(\rho \tilde{u} A)_e - (\rho \tilde{u} A)_w + (\rho \tilde{v} A)_n - (\rho \tilde{v} A)_s\right]
\end{aligned} \tag{16}$$

The lowercase coefficients in the denominator are the central coefficients from equation (12). The central uppercase coefficient is the sum of neighboring coefficients, $a_P = a_E + a_W + a_S + a_N$. From equation (16), it is obvious that the a coefficients populate the left-hand side matrix A, vector $\boldsymbol{x}$ are the implicit pressures to solve for, and b_P is the right-hand side (along with boundary conditions, where pressure is known). More on the boundary conditions in section **Insert reference to section here** and more detail about the solution of the implicit linear equations in section **another one**.

With the implicit solution for the pressures, the momentum equation (13) can be solved, using both the results from equation (14) and equation (15). This concludes one iteration of the SUV-CUT algorithm. The process repeats until the stipulated convergence criteria is reached (from section 3.11) or the maximum number of iterations 3.10 is reached. Below 1 is the pseudocode for the SUV-CUT algorithm.

Algorithm 1 The basic SUV-CUT pseudocode.

```
Set $\boldsymbol{u^0}$, $\boldsymbol{v^0}$, $\boldsymbol{P^0}$
while conv > tol or iter < MaxIter do
    Step 1: Explicit solve for $\tilde{\boldsymbol{u}}$, $\tilde{\boldsymbol{v}}$
    Step 2: Create $A$
    Step 3: Implicit solve $A\boldsymbol{p} = \boldsymbol{b}$
    Step 4: Explicit solve for $\boldsymbol{u}$ and $\boldsymbol{v}$
    Step 5: Evaluate conv
end while
```

4.7 Boundary Conditions: The Physical System

The physical model from which the boundary conditions are derived is shown in figure 1.

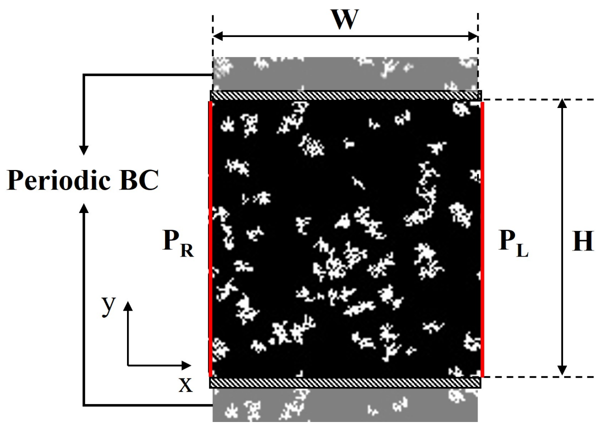

Figure 1: The boundary condition reflecting the physical model.

The physical model requires a set pressures at $x = 0$ and $x = W$. These boundaries are enforced directly as shown in the equation (17).

$$\begin{aligned} P(x = 0) &= P_R \\ P(x = W) &= P_L \end{aligned} \tag{17}$$

The fixed pressure boundary condition, as implemented in the current algorithm, is defined to be the fixed pressure just outside of the domain, and not inside of the domain. The only assumption made about the u-velocity is that the u-velocity outside the domain is the same as the u-velocity just at the boundary, which is valid for both x-boundaries.

The y-boundaries assume a periodic domain. The physical interpretation of this boundary condition is that the domain being simulated is a representative volume element (RVE). It follows from the RVE assumption that it is surrounded by RVEs, hence the north and south boundary are "connected" (in reality, they are connected to other identical RVEs).

4.8 Domain Voxel Classification: The Flood Fill Algorithm

The algorithm accepts grayscale (binary) images as input. From the input, it is really easy to classify between solid and fluid, any simple thresholding/voxel counting algorithm can do that. However, this physical model demands two special physical treatments. First, the code must determine if there is a valid path, entirely through the fluid domain, connecting the left and right boundaries. If the two boundaries are not connected then there is no flow and the permeability is just zero. Second, pockets of fluid that are completely isolated by solids, and do not interact with any fluid coming from a boundary or going to a boundary, must receive special treatment.

The most efficient way to do this that has been implemented to date is a flood-fill algorithm. Consider a grid Ω with $m \times n$ voxels. The first step is classifying every single element $\bar{\Omega}_{m,n} \in \Omega$ in either solid of fluid. Solid voxels are assigned $\bar{\Omega}_{m,n} = 1$ and fluid is assigned $\bar{\Omega}_{m,n} = -1$. The second step, is identifying all fluid voxels at either boundaries $x = 0$ and $x = W$. Those voxels are assigned a different flag, $\bar{\Omega}_{m,n} = 0$, indicating that they are connected to at least on boundary, and therefore need no special treatment. Here, a list is defined: *OpenList*(). Each entry on this list is a tuple containing the coordinates of a voxel. All fluid voxels tagged with a 0 flag are added to the *OpenList* automatically upon assigning the flag to 0.

The algorithm now repeats a loop until there are no more voxels in the *OpenList*. The first step inside the loop is to pop each voxel $\bar{\Omega}_{m,n}$ currently inside the the *OpenList*, remove the entry from the *OpenList*. Then, each voxel $\bar{\Omega}_{m,n}$ has four neighboring voxels, q_{nb}, where the subscript *nb* are the cardinal directions E, W, S, and N. Then, if any $q_{nb} == -1$, then $q_{nb} = 0$ and the coordinates of q_{nb} get added to the open list. This process is repeated until there are no more items in the open list. This search applies the periodic boundary condition to the north and south.

The final step is to perform another full scan. If there are any $\bar{\Omega}_{m,n} == -1$, then these are non-participating fluid. In other words, they are not connected to any fluid channel that connects to a boundary, and are simply surrounded by solid. These voxel receive a separate flag, $\bar{\Omega}_{m,n} = 2$, and will receive special treatment in the discretization. A pseudocode of this algorithm is available 2.

Algorithm 2 Pseudocode for the Flood Fill algorithm, as is used in this study.

```
for $\forall \quad \bar{\Omega} \in \Omega$ do
    if $\bar{\Omega} == Solid$ then
        $\bar{\Omega} = 1$
    else
        $\bar{\Omega} = -1$
    end if
end for
Declare $OpenList()$
for $\forall \quad \bar{\Omega}|_{x=0}$ or $|_{x=W}$ do
    if $\bar{\Omega}_{m,n} == -1$ then
        $OpenList.append([m, n])$
    end if
end for
while $OpenList() \mathrel{!=} \varnothing$ do
    $[m, n] = OpenList.begin()$
    $OpenList.Erase(OpenList.begin)$
    if $u_{nb} == -1$ then
        $OpenList.Append([nb])$
    end if
end while
for $\forall \quad \bar{\Omega} \in \Omega$ do
    if $\bar{\Omega} == -1$ then
        $\bar{\Omega} = 2$
    end if
end for
```

4.8.1 Solid Interface Treatment

After the Flood Fill algorithm 2, all pixels have a flag of 0 for fluid, 1 for solid, and 2 for non-participating fluid. There are several commonly accepted ways to deal with the solid fluid interfaces to ensure no penetration of fluid and no slip condition at the interface. To avoid complications in the numerical solution, Wang et al. [9] proposed a simple and elegant solution: instead of treating the interface as a boundary and eliminating the solid from the computational domain, the solid is kept as part of the simulation domain and receives the following (18) discretization:

$$\phi_P = 0 \tag{18}$$

Where ϕ stands for u, v, and P. This treatment is called the pressure-decoupled solid velocity correction. It is a great method because it doesn't involve large numbers (such as switch-off or Darcy source term), hence there are no complications with the solver.

4.8.2 Non-Participating Fluid Correction

The non-participating fluid receive a similar treatment to the pressure-decoupled solid velocity correction. However, while there is no physical meaning to applying a pressure to the solid, it makes sense to have a pressure applied on the non-participating fluid. While that doesn't contribute or affect the solution, it does make a difference when pressures of 0 (solid) are masked off when plotting the pressure and velocity maps. Therefore, the correction for non-participating fluid is as follows 19:

$$\begin{aligned} P &= P_R \\ u &= 0 \\ v &= 0 \end{aligned} \tag{19}$$

4.9 Permeability Calculation

The permeability is calculated using Darcy's equation for permeability (20).

$$\dot{Q}_{avg} = -\frac{KA}{\mu}\frac{\Delta P}{W} \tag{20}$$

The pressure drop $\frac{\Delta P}{W}$ is one of the user inputs, μ is a material property, K is Darcy's permeability, A is the cross-sectional area, and Q_{avg} is the average flow rate through the domain. Therefore, from the converged simulation output Q_{avg} can be calculated, and by manipulating equation (20) the permeability can be obtained. On the pixel-based grid of dimensions $N \times N$, the calculation of Q_{avg} is shown in equation (21).

$$\dot{Q}_{avg} = \frac{1}{N+1}\sum_{j=0}^{N+1}\sum_{i=0}^{N} u_{i,j}A_{i,j} \tag{21}$$

Finally, the Darcy permeability, K, has units of length squared (i.e. 1 darcy = $9.87 \times 10^{-13} m^2$), but generally a dimensionless measure is preferred. As such, an arbitrary depth, dz, is assigned to the simulation such that $dx = dy = dz$. Then, equation (20) can be rearranged to yield K^*, a dimensionless form of Darcy's permeability.

$$K^* = -\frac{\dot{Q}_{avg}\mu W}{A^2 \Delta P} \tag{22}$$

Note from equation (22) that K^* is both independent of the width, W, and independent of the mesh refinement (arbitrary depth), dz.

5 Code Outputs

5.1 PUV Map

As discussed in section 4.1, this model doesn't necessarily need a $Re < 1$. The main output of this code is not actually the permeability, as the permeabilities and flow rates can all be derived from a map of the velocity components.

As explained earlier, the printMaps option controls whether or not these maps will be printed. As a general recommendation, always leave that flag as true. The maps get printed to a .csv file in the following format:

```
P,U,V,x,y
99.999,0.000000,0.01,0,0
99.999,0.000000,0.01,1,0
99.999,0.000000,0.00,2,0
```

This is just an example, the numerical values are not important. What is noteworthy is that the first line in the csv is the labels, and after that all of the numerical values. In any given row, the output is the pressure, x-velocity, and y-velocity a given x and y location.

Another important note is that the code was written in C++, so it is very convenient to use the top left corner as $y = 0$. That is important when reading the output for plotting. A sample plot is shown below. A python code to read the maps and plot them is included in the Github repository.

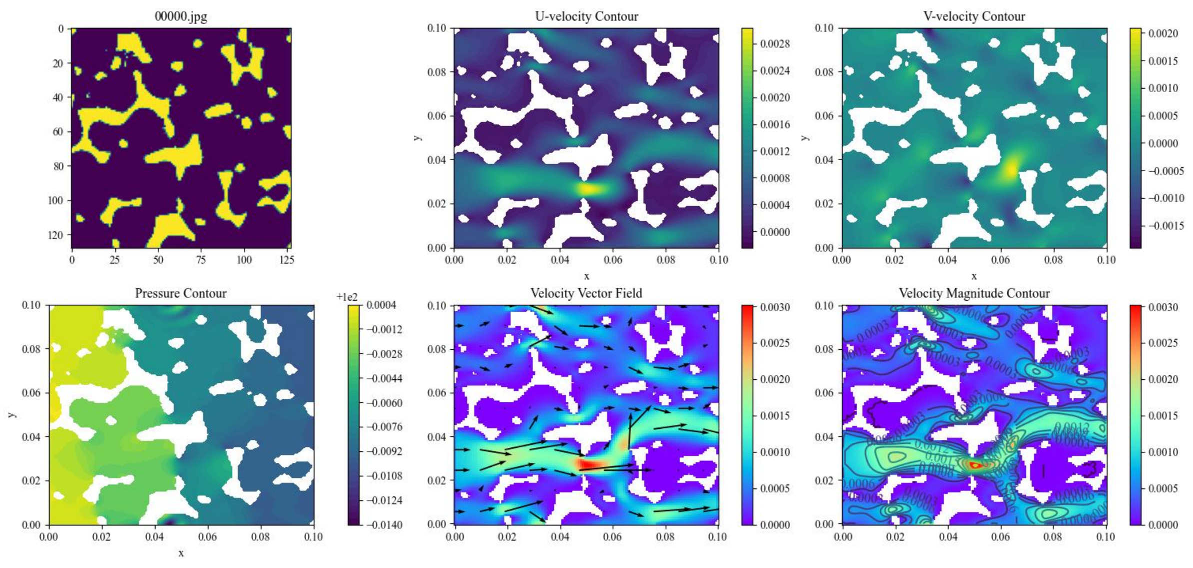

Figure 2: Sample contours of a simulation.

5.2 HPC Output File

The output file contains a lot of information about the simulation. It is also output in a csv format. See an example below:

```
imgNum,RMS,K,iter,porosity,time
0,3.553e-08,4.701e-04,701,0.800354,233.697374
75,4.929e-08,2.501e-05,1301,0.342163,433.826034
76,5.962e-08,4.687e-05,1601,0.466431,535.013514
```

The time in the HPC version is the total execution time calculated with the OpenMP packages. The HPC version is great for batches of images and is highly recommended. Even if only one GPU is available, still run the HPC version by selecting nCores = 1.

5.3 Convergence Data

HPC and GPU versions have an additional output option, not yet implemented to the CPU version. The decision on whether or not to print the convergence data output file is hard-coded into line 10 of the ".cu" file. It is set to "true" by default, and if the user wants to change it, then compilation is needed again.

This file simply contains an assessment of the convergence criteria, the residual continuity, and permeability calculated at every iterative step. A python script is included to readily plot these convergence maps.

6 Implementation Details

6.1 Sparsity Analysis

As has been emphasized multiple times in this document, the main point of writing this algorithm is to have a more efficient alternative to commercial CFD packages for generating pressure-velocity maps and permeability data for training machine learning models. Hence, the emphasis is in the efficiency of the model for being implemented in large batches of data.

The GPU and HPC versions of the code use CUDA-enabled GPUs, so why not just use one of the solvers from cuSPARSE library? Per cuSPARSE documentation, the library targets matrices with sparsity ratios in the range between 70-99.9%. If we account for a sample 2D system with 128×128 nodes, the coefficient matrix A has a size of $16,384 \times 16,384$. That is over 260 million elements. With the current hybrid discretization, there are 5 main diagonals in the system. Without accounting for any solid in the domain (which would yield more zero entries in the matrix), there are a maximum of $81,920$ non-zero entries. From this calculation, the matrix is 99.97% sparse, and therefore is outside of the target range for cuSPARSE usage. Furthermore, if the size is increased from 128×128 to 256×256, the matrix is even more sparse.

So, another method is needed for efficiently solving this extremely sparse matrix, which ultimately determines how fast the simulation itself is. Additionally, the coefficient matrix needs to be store in an efficient way, since there is no point in allocating massive amounts of memory for a matrix that is 99% sparse.

The cuSPARSE methods for storing matrices are the Coordinate (COO) storage and the Compressed Sparse Row (CSR) method. THE COO stores three vectors of size nnz, where nnz is the number of non-zero elements. One vector stores the row indexes, one vector stores the column indexes, and a third vectors stores the values of all non-zero elements. This method stores $3 \cdot nnz$ elements, which using the example described about, would mean storing $3 \cdot 81,920 = 245,760$ elements. Without going into much detail about the CSR, it stores the row offsets, the column indexes, and the non-zero entries. A rough estimate of the CSR for large systems is $2 \times nnz + nRows$, where $nRows$ is the number of rows of the coefficient matrix. With this method applied to the example above, the computer would have to store $2 \times 81,920 + 16,384 = 180,224$. The CSR method, in this example, saves 30% of the space used by the COO method.

On the other hand, our coefficient matrix A is really well structured, all non-zero entries occur within the 5 main diagonals, and we know the exact position of each of them. Thus, we can store only these five diagonal in a $16,384 \times 5$ matrix, using only $81,920$ elements in the computer memory. The matrix, which has 5 columns and 16,384 rows, is arranged as follows:

```
/* Create the "A" matrix in Ax = b system

Indexing is as follows:

P = 0
W = 1
E = 2
S = 3
N = 4
*/
```

Then, A can be called as follows:

```
A[index + j] = coeff;
```

where "*index*" is the row number of the original coefficient matrix A, and "*j*" is the offset. The offsets control which main diagonal we are calling. This indexing, besides saving enormous amounts of memory, is very efficient in the implementation of the Jacobi method, which will be the solver of choice for this work.

6.2 Implicit Numerical Solution: GPU and HPC

The chosen method for solving the implicit $\boldsymbol{Ax} = \boldsymbol{b}$ system is the successive over-relaxation of the Jacobi (SOR-Jacobi) method. The Jacobi method is one of the standards, and won't be covered here.

To apply this method in a GPU programming environment, a GPU kernel needs to be defined. All operations related to copying arrays, assessing convergence, and printing are handled by the CPU. The only thing that happens inside the GPU is the actual computation of the main loop in the Jacobi method. For a system with the indexing shown in the section above, the kernel updates the x-vector from $\boldsymbol{Ax} = \boldsymbol{b}$ as follows:

```
__global__ void updateX_SOR(float* A, float* x, float* b, float* xNew, int numCellsX, int numCellsY)
{
 // Figure out the index
 unsigned int myIdx = blockIdx.x * blockDim.x + threadIdx.x;
 // break index into row and col
 int myRow = myIdx/numCellsX;
 int myCol = myIdx % numCellsX;
 float w = 2.0/3.0;

 int nElements = numCellsX*numCellsY;

 // do the Jacobi Iteration with periodic BC

 if (myIdx < nElements){
  float sigma = 0;
  for(int j = 1; j<5; j++){
   if(A[myIdx*5 + j] !=0){
    if(j == 1){
     sigma += A[myIdx*5 + j]*x[myIdx - 1];
    } else if(j == 2){
     sigma += A[myIdx*5 + j]*x[myIdx + 1];
    } else if(j == 3){
     if(myRow == numCellsY - 1){
      sigma += A[myIdx*5 + j]*x[myCol];
     }else{
      sigma += A[myIdx*5 + j]*x[myIdx + numCellsX];
     }
    } else if(j == 4){
     if(myRow == 0){
      sigma += A[myIdx*5 + j]*x[(numCellsY - 1)*numCellsX + myCol];
     }else{
      sigma += A[myIdx*5 + j]*x[myIdx - numCellsX];
     }
    }
   }
  }
  xNew[myIdx] = (1.0-w)*x[myIdx] + w/A[myIdx*5 + 0] * (b[myIdx] - sigma);
 }
}
```

Some research has suggested using a weight of 2/3 for the successive over-relaxation, but any number of $0 < w \leq 2$ should be stable. The number of threads per block was not extensively tested, but as it stands it works fine with 160 threads per block.

Also note how j controls the indexing of the $\boldsymbol{x}$: $j = 1$ means the west neighbor, so that would be the lower main diagonal $myIndex - 1$. In C++ language, the array is stored from top left to bottom right, so note that North $j = 4$ translated to $myIdx - numCellsX$, where $numCellsX$ is the number of columns. That means the north diagonal is offset by the number of columns from the main diagonal. Finally, note how the periodic boundary condition is applied at this stage for $j == 3$ and $j == 4$ when we are at the edges of the domain.

6.3 Implicit Numerical Solution: CPU

As stated prior, the CPU version has lagged behind the other versions in testing and development. The reason for this is that one relatively cheap GPU (specifically, most of the testing was done on a GeForece RTX 3070) can be hundreds of times faster than a singular CPU core. While most modern computers have upwards of 16 CPU cores, the GPU still saves at least one order of magnitude in computational demand. Also, NVIDIA along with a study by the National Energy Research Scientific Computing Center (NERSC) claims GPUs to be significantly more energy efficient when compared to CPUs.

The CPU version of the code still employs a parallel CPU acceleration of a Jacobi method without the SOR. The parallel construct is an acceleration of the inner loop of the Jacobi method (for updating the $\boldsymbol{x}$ vector) using the following calls:

```
    #pragma omp parallel private(index, sigma)
    #pragma omp for
```

The first call sets the variable σ and the index as private to each CPU core, ensuring the CPU cores aren't modifying each others' variables. Since the matrix with the coefficients and the x-vector are very large in comparison with the number of cores available, it is very unlikely that two cores will access the same portion of the x-vector at the same time (this is the main source of inefficiency in parallel OpenMP constructs).

6.4 Data Structures

The GPU and HPC versions have two main data structures that are essential for functionality. The HPC version has an additional data structure to handle the convergence data for each image, while the GPU version handles that in-code. The data structures are reffered to as structs. These structs play a vital role organization wise, as using the structs ensures that all the functions in the code receive the same essential information in a very compact manner.

The first essential struct is the "options" struct, defined below:

```
// Structure for saving user entered options
typedef struct
{
 float density;
 float viscosity;
 float PL;
 float PR;
 int MeshAmp;
 char* inputFilename;
 char* outputFilename;
 bool printMaps;
 bool verbose;
 float alphaRelax;
 long int MaxIterSolver;
 float ConvergenceSolver;
 long int MaxIterGlobal;
 float ConvergenceRMS;
 float DomainHeight;
 float DomainWidth;
 int nCores;
}options;
```

This struct handles all of the user entered options and the input file. Pretty much every function in the entire code takes this struct as an input.

The second struct is the "simulationInfo". It is defined as follows:

```
// Struct to hold constants, variables, and results intrinsic to the simulation
typedef struct{
 int numCellsX;
 int numCellsY;
 int nElements;
 float dx;
 float dy;
 float porosity;
 float gpuTime;
 float Perm;
 float Flowrate;
 float ResidualX;
 float ResidualY;
 float ResidualP;
}simulationInfo;
```

This is a struct that contains information that is pertinent to each and every simulation. The HPC version declares one of those for every single simulation, while the GPU version only needs one. The only noteworthy aspect about the “simulationInfo” is that the GPU time variable is deprecated and is not used anywhere anymore.

References

[1] Versteeg, H. & Malalasekera, W. *An introduction to computational fluid dynamics* (Prentice Hall, Philadelphia, PA, 2007), 2 edn.

[2] Rhie, C. M. & Chow, W. L. Numerical study of the turbulent flow past an airfoil with trailing edge separation. *AIAA Journal* **21**, 1525–1532 (1983).

[3] Ghia, U., Ghia, K. N. & Shin, C. T. High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method. *Journal of Computational Physics* **48**, 387–411 (1982).

[4] Spalding, D. B. A novel finite difference formulation for differential expressions involving both first and second derivatives. *International Journal for Numerical Methods in Engineering* **4**, 551–559 (1972). URL `http://dx.doi.org/10.1002/nme.1620040409`.

[5] Patankar, S. & Spalding, D. A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows. *International Journal of Heat and Mass Transfer* **15**, 1787–1806 (1972). URL `https://www.sciencedirect.com/science/article/pii/0017931072900543`.

[6] Patankar, S. V. *Numerical heat transfer and fluid flow*. Series in computational and physical processes in mechanics and thermal sciences (CRC Press, Boca Raton, FL, 1980).

[7] Van Doormaal, J. P. & Raithby, G. D. Enhancements of the simple method for predicting incompressible fluid flows. *Numerical Heat Transfer* **7**, 147–163 (1984). URL `http://dx.doi.org/10.1080/01495728408961817`.

[8] Jin, W. W., Tao, W. Q., He, Y. L. & Li, Z. Y. Analysis of inconsistency of SIMPLE-like algorithms and an entirely consistent update technique - The CUT algorithm. *Numerical Heat Transfer, Part B: Fundamentals* **53**, 289–312 (2008).

[9] Wang, S., Faghri, A. & Bergman, T. L. A comprehensive numerical model for melting with natural convection. *International Journal of Heat and Mass Transfer* **53**, 1986–2000 (2010). URL `http://dx.doi.org/10.1016/j.ijheatmasstransfer.2009.12.057`.