Hemodynamic Simulation with Lattice Boltzman

Harvard IACS AC 290R - Group 1
Michael S. Emanuel
Jonathan Guillotte-Blouin
Yue Sun

28-April-2019

1 Problem Statement and Motivation

AC 290R is a course on extreme computing with a focus on the application domain of fluid dynamics. In our first module we took on a prototypical problem using the continuum description of fluids governed by the Navier-Stokes equation: Rayleigh-Bénard Convection. In this module, we shift from the continuum to the mesoscale description and simulate fluids using Lattice Boltzmann methods. These methods are based on the Boltzmann Equation of thermodynamics and statistical physics.

The particular task we undertook was a hemodynamic simulation. Hemodynamics is the study of the dynamics of blood flow. It is a rich field at the intersection of anatomy and physics with a storied history. Pioneers in the field include Leonardo Da Vinci, Leonhard Euler, Thomas Young, and Jean L.M. Poieseuille.

The particular problem was as follows. We aim to model the dispersion of a therapeutic drug that is injected with a catheter to treat a stenotic artery. Stenosis is a disease of the arteries in which an artery becomes narrowed. It is frequently caused by atherosclerosis, a build-up of fatty deposits (cholesterol) on the walls of the artery. Stenotic arteries can cause serious medical problems including heart attacks and strokes. One approach to treating stenoses is to introduce a therapeutic drug with a catheter, a small tube inserted surgically into the patient that can be threaded through the circular system. The goal of our simulation was to understand how the drug molecules dispersed and to see how many of them passed through the stenotic region, and at what times.

This is a worthwhile object of study for both pedagogical and practical reasons. Pedagogically, this topic rounds out our survey of both techniques and domain knowledge in the course. We are aiming to master the techniques of extreme computing and their application to fluid dynamics. The first topic covered the continuum approach, and this topic covers the mesoscale approach. The first topic used traditional CPU-centric computations, and this topic introduces us to GPU computing.

On a practical level, heart disease has been for many years a leading cause of death in Americans alongside cancer. It is a complex disease with many treatment options and a need for accurate diagnosis. Clinical practice still often relies on human judgment about which arterial blockages look risky, and there is reason for optimism that advances in biologically realistic computer simulations could lead to materially faster and more accurate diagnosis and improve treatement selections.

2 Overview of Numerical Methods Used

The main numerical method used in this simulation is the Lattice Boltzmann Method (LBM). LBM is based on the Boltzmann Equation, which describes the statistical behavior of a thermodynamic system and dates to 1872. The idea behind the Boltzmann Equation is that particles in the system each have 6 degrees of freedom, 3 positions and 3 momenta along the 3 coordinate directions x, y and z. Molecules of a given type are physically indistinguishable from each other, so the system can be described completely by the populations of particles as a function of time and these six dimensions in the phase space. The Boltzmann Equation describes the evolution of such a system. The populations of particles change due to three terms: diffusion, collisions, and external forces.

LBM is a technique in computational fluid dynamics that uses the Boltzmann Equation to devise a numerical simulation of a fluid that can accurately capture mesoscale dynamics when it is tuned properly. The physical system is discretized, typically on a rectilinear grid and most commonly one with cubic spacing. Particle velocities are also discretized, with particles allowed to jump from one grid point only to nearby grid points over one simulation step. The most common discretization scheme for 3D fluid simulations, which we used for this problem, is called D3Q19. The label can be parsed as referring to 3 dimensions and 19 discrete velocities. The 19 discrete velocities have the following structure:

- 1 0^{th} neighbor; displacement $(0,0,0)$; weight $\frac{1}{3}$
- 6 1^{st} neighbors; displacement one of 3 combinations of $(\pm 1,0,0)$; weight $\frac{1}{18}$
- 12 2^{nd} neighbors; displacement one of 3 combinations of $\pm 1, \pm 1, 0)$; weight $\frac{1}{36}$

The 6 first neighbors have displacements $(1,0,0), (-1,0,0), (0,1,0), (0,-1,0), (0,0,1), (0,0,-1)$. The 12 second neighbors follow a similar pattern; there are $\binom{3}{2} = 3$ combinations of indices i, j, and each index has 2 choices in ± 1, leaving $4 \cdot 3 = 12$ second neighbors. The total weight of the 6 first neighbors is $6 \cdot \frac{1}{18} = \frac{1}{3}$. The total weight of the 12 second neighbors is $12 \cdot \frac{1}{36} = \frac{1}{3}$.

The state of the simulation at a time step is given by the *population* of particles, $f_p(x,t)$, where the subscript p refers to the discrete velocities above. In a system with more than one type of particle, each particle of every populations must be maintained separately. The movement of populations can be described by the Bhatnagar-Gross-Krook (BGK) update rule:

$$f_p(x + hc_p, t + h) = f_p(x,t) + \omega(x,t)h \left[f_p^{eq}(\rho, \mathbf{u} - f_p)(x,t) + w_p \frac{c_p \cdot \mathbf{g}}{c_s^2} \right]$$

Here is a brief description of all the terms appearing in this equation from left to right:

- f_p is the population of the particle introduced above.
- h is the time step, often taken to be 1 in simplified notation
- c_p is the displacement vector corresponding a given discrete velocity, e.g. $c_0 = (0,0,0)$, $c_1 = (1,0,0)$, etc.
- ω is the relaxation frequency which is related to the kinematic viscosity ν (described below)
- $\rho(x,t)$ is the density of the fluid in this cell, a macroscopic quantity; essentially the 0th moment of the velocity
- $\mathbf{u}(x,t)$ is the velocity of the fluid in this cell, a macroscopic quantity; essentially the first moment of the velocity
- w_p is the weight of particles with each velocity in the D3Q19 scheme; scalars that do not change over the simulation

- c_p is the discrete velocity
- $\mathbf{g}$ is the acceleration applied to the body by external forces; the letter g evokes gravity but it can be any external force
- c_s is the speed of sound in dimensionless units on this lattice, equal to $\sqrt{1/3}$ here. (The speed of sound of the physical medium depends on the gradient of pressure with respect to density).

The equilibrium populations can be approximated with a Taylor expansion that accounts for the 0th, 1st, and 2nd moments of the populations.

$$f_p^{eq}(\rho, \mathbf{u}) = w_p \rho \left[1 + \frac{\mathbf{u} \cdot c_p}{c_2^2} + \frac{(\mathbf{u} \cdot c_p)^2 - c_s^2 u^2}{2c_2^4} \right]$$

The resulting f_p^{eq} will match the first two moments (density ρ and velocity $\mathbf{u}$), but in general it will **not** match the second moment (energy density) unless the system is at equilibrium. This is how energy dissipation in a viscous fluid away from equilibrium is modeled. This approximation is valid as long as the system is sufficiently close to equilibrium. If the parameters are not set properly, the populations can depart from equilibrium by too much and the simulation will break down.

The macroscopic quantity density ρ is simply the sum of the fluid populations in a cell.

$$\rho(\mathbf{x}, t) = \sum_p f_p(\mathbf{x}, t)$$

The momentum density is the first moment of the particle velocities. This is equal to the density times the velocity in a cell, giving us the formula for $\mathbf{u}(\mathbf{x}, t)$:

$$\mathbf{J}(\mathbf{x}, t) = \rho(\mathbf{x}, t)\mathbf{u}(\mathbf{x}, t) = \sum_p c_p f_p(\mathbf{x}, t)$$

$$\mathbf{u}(\mathbf{x}, t) = \frac{1}{\rho(\mathbf{x}, t)} \sum_p c_p f_p(\mathbf{x}, t)$$

The relaxation frequency is related to the kinematic viscosity ν by the following relationship:

$$\nu = c_s^2 \left(\frac{1}{\omega} - \frac{1}{2} \right)$$

A Lattice Boltzmann fluid simulation can be organized into two logical phases, which are sometimes called *collision* and *streaming*. The collision step describes how particles populations in the same cell interact and how their weights move toward their equilibrium values as a result of collisions. The intuition is that when particles hit each other, momentum is conserved, but some kinetic energy is dissipated as heat or otherwise. The equation for the collision step can be written

$$f_p^* = (1 - \omega) f_p + \omega f_p^{eq}$$

where f_p^* are called the temporary post-collisional populations. We can think of these as the new populations one "instant" after the previous streaming step.

The streaming step describes how particles of the post-collisional population move forward into the next time step. This is very straightforward, the particles just move at their discrete velocities according to

$$f_p(x + c_p, t + 1) = f_p^*(x, t)$$

A key fact is that streaming is a *local* operation. This makes it ideally suited to GPU computations, which excel at simple, highly parallel tasks with memory locality.

Stability is an important concept in the numerical solution of differential equations generally. One of the best known stability criteria is the Courant-Friedrichs-Lewy (CFL) Condition. This relates the range of time steps Δt for which a numerical method is stable to the spatial discretization Δx, the speed of movement u, and a dimensionless constant that is a property of the numerical method. A rule of thumb for LBM is that optimal results are achieved when

$$|u| < \sqrt{\frac{2}{3}} \frac{\Delta x}{\Delta t}$$

A representative value of u is 0.1, leading to a guideline that stable results can be found when the kinematic viscosity ν is selected in the range $0.05 < \nu < 1$.

Like any differential equation solution method, LBM must also cope with initial conditions and boundary conditions. Common initial conditions are prescribed values for the pressure (equivalent to a prescribed density) and velocity, i.e.

$$\rho(x, t = 0) = \rho_0(x)$$
$$\mathbf{u}(x, t = 0) = \mathbf{u}_0(x)$$

A common choice is to set the initial density to be uniform and the initial velocity to be zero. Once ρ and $\mathbf{u}$ are initialized, a common modeling choice is to initialize the populations to the equilibrium implied by these macroscopic variables, i.e.

$$f_p(x, 0) = f_p^{eq}(\rho_0(x), \mathbf{u}_0(x))$$

Boundary conditions are a bit trickier. An astute reader will quickly point out that with these equations as written, any finite domain will have particles "falling off the edge" and without sensible treatment of boundary conditions, the simulation would just report that there were no particles left. That would be sad. In general, a boundary condition in an LBM simulation can be expressed as a linear combination of a constraint on the flux in a direction normal to a wall, and on the density itself. Taking ϕ to denote the quantity of interest (can be density or velocity) and n the normal direction,

$$b_1 \frac{\partial \phi}{\partial n}(x_b, t) + b_2 \phi(x_b, t) = b_3$$

A few special cases are the most common. When $\phi = \mathbf{u} = 0$, we have the **no slip** boundary condition. This is common and physically realistic. When $b_1 = 0$, we have a Dirichlet boundary condition, in which the value is imposed. When $b_2 = 0$, we have a Neumann boundary condition, in which the flux is imposed. When $b_1 \neq 0$ and $b_2 \neq 0$, the boundary condition is mixed (linear relationship between field value and flux) and is called a Robin boundary condition. In addition to no-slip boundary conditions, another common choice is for velocity to be fixed at an *inlet* or *outlet*. Finally, larger sytems can often be simulated using a *periodic* boundary condition. For the simulation we ran of an artery, we imposed a no-slip boundary condition on the side walls of the artery, and a periodic boundary condition on the longitudinal direction, effectively modeling a longer artery with periodic stenoses.

3 Description of Code

The workhorse of this simulation is a Lattice Boltzmann fliud simulator called MUPHY. MUPHY is about 10 years and designed for multi-physics simulations. We also used a newer software package called MOEBIUS. MUPHY is an open source project on which our guest instructor Simone Melchionna was one of the lead developers. MOEBIUS is a commercial package developed by a private company, Lexma, founded by Dr. Melchionna.

MUPHY and MOEBIUS (hereafter referred to simply as MUPHY to save space) follow a similar strategy to Drekar. They are object-oriented written primarily in a mix of C / C++ and Fortran, with GPU acceleration written in CUDA. The back end is implemented in lower level languages like C / C++ to meet the requirements for high performance. Another similarity to Drekar is that parallelization is proved using OpenMP. There is also a front end interface for managing jobs. This is written in Python. To run our simulations, we needed to install the MUPHY package and write scripts in Python. We did not need to write or compile any C++ or CUDA.

MUPHY was written with a particular focus on applications in life sciences. It can handle the complex geometries arising in biological systems, ranging in scales from folding proteins and cell membranes up to highly branched arteries. We saw demonstrations in class run on MUPHY of a DNA molecule passing through a cell membrane, and of a flow simulation in arteries whose geometry was based on a patient.

The first code we wrote is in the file `ShapePainter.py`. This code can be found in a repository named BUFFY our team created on Odyssey. We needed to fork the MUPHY repository so we could make edits and push changes. We named it BUFFY in homage to the hit TV show Buffy the Vampire Slayer. Access to this repository can be provided to the teaching staff on request. We took snapshot of the most relevant Python scripts and saved them in our team repository (where we submit our coursework) in the directory `/project2/BUFFY`. `ShapePainter.py` creates a geometry for the system using the `vtk` library. The artery is modeled with a simplified geometry as a cylinder extending down the z axis longitudinally. It has a radius R on the healthy region and $G = R/2$ in the stenotic region. The length of the entire artery is L and the length of the stenotic region is S. This image was provided with the problem statement and demonstrates the basic layout and parameter names:

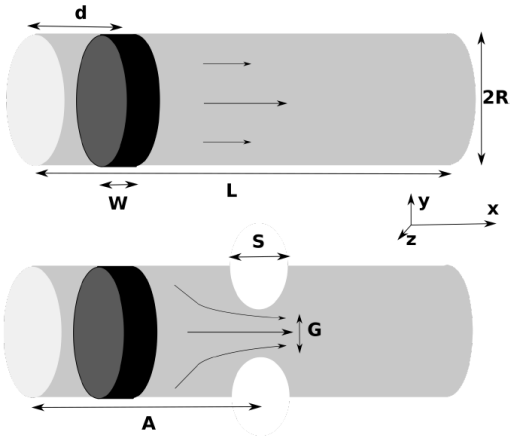


Figure 1: Artery Geometry.

We used the `ShapePainter` to create the geometry, in which there are two files of particular interest:

1. `preproc1.py`: Uses a 3d canvas to create the shape and then marching cubes to generate the surface. Change the variable Re to get the desired the radius and length of the channel. For $Re = 10$, this script takes around 10 minutes to run.

2. `preproc2.py`: Created `RBC.xyz` and `atom.inp` objects based on the channel. Originally for $Re = 10$ it took 8 hours to finished 22% of the job, which would need about 40 hours to finish the whole RBC implementation. Thanks to the update in the `MAGIC` backend `Fortan` code, it took less than 10 minutes to finish the RBC creation.

The next code we wrote is the Python script that runs the simulation on MUPHY. Each simulation run had a file, conventionally named `run2.py`, that kicks off the simulation. The script that ran our baseline simulation is located in `scratchlfs/ac290r/project2/blood_cells/BUFFY/RBC_0_Re10` (We tried to follow advice given to use organized directory names. While this is a bit long and slightly cryptic, we did at least follow a consistent scheme this time organized everything hierarchically!) We've included a copy of this file in project repository as well in `project2/BUFFY/RBC_0_Re10` for convenience in grading our submission.

```
ac290ru1906@boslogin03:/n/scratchlfs/ac290r/blood_cells/MUPHY$ tree -L 2
.
└── MAGIC
    └── BACKEND

ac290ru1906@boslogin03:/n/scratchlfs/ac290r/blood_cells/BUFFY$ tree -L 2
.
├── RBC_0_Re10
│   ├── bgkflag.dat
│   ├── bgkflag.hdr
│   ├── bgkflag.xyz
│   ├── DIRDATA_BloodFlow
│   ├── DIRDATA_Bolus
│   ├── genparalleldomains.py
│   ├── jobgpu_parallel.slurm
│   ├── jobgpu_serial.slurm
│   ├── job.slurm
│   ├── RBC.xyz
│   ├── run2.py
│   └── runrbc_0.sh
├── RBC_5_Re5_NEW
│   ├── atom.inp
│   ├── bgkflag.dat
│   ├── bgkflag.hdr
│   ├── bgkflag.xyz
│   ├── genparalleldomains.py
│   ├── jobgpu_parallel.slurm
│   ├── jobgpu_serial.slurm
│   ├── job.slurm
│   ├── RBC.xyz
│   ├── runrbc.py
│   ├── runrbc.sh
│   └── wall.xyz
└── ShapePainter
    ├── all_mod.mod
    ├── atom.inp
    ├── bgkflag.dat
    ├── bgkflag.hdr
    ├── bgkflag.xyz
    ├── EXTRAS
    ├── __init__.py
    ├── preproc1.py
    ├── preproc2.py
    ├── preproc.sh
    ├── RBC.xyz
    ├── Re2
    ├── set_modules_vtk.sh
    ├── ShapePainter.py
    ├── ShapePainter.pyc
    ├── SP.stl
    └── wall.xyz
```

```
[ac290ru1906@boslogin03:/n/scratc
.
├── 106_SSUNG_WK
├── 107_HEMO_BIFURCA.PARTITION
├── backup_107
├── CUDA-LBM
├── MAGIC
├── RBC
├── RBC_0_Re10
├── RBC_0_Re10_4
├── RBC_0_Re10_4M
├── RBC_0_Re5
├── RBC_10_Re10
├── RBC_10_Re5
├── RBC_30_Re10
├── RBC_30_Re5
├── RBC_5_Re10
├── RBC_5_Re5
├── RBC_5_Re5_NEW
├── README.md
├── set_modules.sh
├── ShapePainter
├── ShapePainterRe10
├── ShapePainterRe10RBC10
├── ShapePainterRe10RBC5
└── ShapePainterRe5

22 directories, 2 files
```

Figure 2: Sample subdirectory in `BUFFY` directory.

In summary, `scratchlfs/ac290r/project2/blood_cells/MUPHY/MAGIC` is our backend library, and in `scratchlfs/ac290r/project2/blood_cells/BUFFY/` each subdirectory represents each simulation with different RBC and Reynolds number. The basic workflow is to first create shapes in `ShapePainter/`, copy the output files into their respective `RBC_X_ReX/` folder, and submit batch scripts.

The first part of this file sets the physical parameters for the simulation, e.g. the geometry of the cylinder and the viscosity. The Python module `MagicUniverse` is the fancifully named interface to the BUFFY simulation engine. The script initializes simulation objects including:

- Universe
- Scale

- Mesh
- Fluid (for blood)
- Fluid (for drug)
- Atom (for RBC)
- Tracker (for diagnostics)

These are all class instances from the `MagicUniverse` module. Parameter values are set to the appropriate class instances. We set the boundary conditions to be periodic on the z axis, and no-slip on the x and y axes. The alternative configuration would have been to create inlets and outlets at the start and end values of z.

We want to release the drug when the blood flow is stable, at $Re = 10$ we have two relaxation time that would impact the choice of the drug release time

$$\tau_{\nu_1} = \frac{L^2}{\nu} = \frac{1000^2}{0.1} = 10^7$$
$$\tau_{\nu_2} = \frac{D^2}{\nu} = \frac{100^2}{0.1} = 10^5$$
$$\tau_u = \frac{L}{u} = \frac{1000}{0.01} = 10^5$$

Note that we choose τ_{ν_2} with D instead of L because the diffisivity phenomenon is local and we do not need the whole length of the channel to expect its motion. Since τ_{ν_2} and τ_u have the same order of magnitude, we choose the drug release time for $Re = 10$ to be $t = 100000$.

To run this code on Odyssey (Harvard's supercomputing cluster in Western Massachusetts) we also wrote shell scripts that were submitted to the Odyssey job manager `slurm`. The script to run the baseline simulation is called `runrbc_0.sh` and is located adjacent to `run2.py`. The key lines in this script set the job options and load the required modules. Important flags include:

- `-p shared` - run the job on the shared partition
- `--reservation=ac290r` use the reservation so we don't have to wait in the queue to get 1024 cores
- `-t 1200` hold the job open for up 1200 minutes = 20 hours
- `-n 512` run on a total of 512 CPU cores
- `N 16` run on 16 nodes; we are therefore requesting 16 nodes with 32 cores each
- `mem=64000` request 64,000 MB = 64 GB per node; that is a lot of memory, 1024 GB = 1 TB total
- `--job-name=RBC0RE10` the descriptive job name refers to 0 red blood cells and Reynolds Number 10
- `--output=RBC0RE10.out` the directory where output is written; matches the job name

The rest of the script loads the required modules and sets environment variables. We need modules for `gcc` and `openmpi`, provided by `gcc/7.1.0-fasrc01` and `openmpi/3.1.1-fasrc01` respectively. The environment variables to set are `MOEBIUS_PATH` and `PYTHONPATH`. These allow Python to find the MAGIC modules. In order to invoke the job as a Python 2 script with MPI, the line of the script that kicks off the actual job is

```
srun -n $SLURM_NTASKS --mpi=pmi2 python2 run2.py
```

The last major batch of code we wrote performs calculations at the post-processing stage. This is in two Python scripts `vtk2np.py` and `rbc.py` located in the folder `project2/src`. The MUPHY simulation generates output in the form of VTK files with the extensions .vtu and .pvtu. These files are well suited to

visualization, especially intensive visualizations like movies. For computations and 2D plots of one time instant, we find it more convenient to use `numpy` and `matplotlib`.

We made a strategic decision to do the computationally heavy graphics rendering remotely on Odyssey but to do the computations and 2D ploting locally on a few selected time frames. After our baseline simulation ran, we downloaded snapshots every 100,000 frames from 100,000 to 1,000,000. Each time step corresponds to 1 microsecond, so these frames were 100 milliseconds apart for the first 1.0 second of the simulation. These files weren't too large, with a total size of about 9 GB each for the blood and drug for a total of 18 GB. Downloading these files was a bit painful though because SFTP connections to Odyssey are considerably slower than data downloads from large commercial websites for whatever reason — we believe that `scratchlfs` might be behind this slowness.

The script `vtk2np.py` converts the VTK files to numpy arrays using the library `vtki`. We initially started to convert these files using the `vtk` library, but we quickly realized that it was quite verbose. We later found out that the other team was discussing with the developers of `vtki` to add the capacity of reading `.pvtu` files directly, which was added less than 2 weeks ago! A .pvtu file is essentially a wrapper around a number of .vtu files, each with one part of a larger mesh. VTK provides various types of files, like `.vti`, `.vtp`, `.vtu`, and so on, with their counterparts prepended with *p* defining the parallel wrappers. Note that these store information as XML documents. In our case, we ran on 512 nodes and each time step generated 512 .vtu files — which are vtkUnstructuredGrids — that were summarized by 1 .pvtu file. These outputs were generated for both blood and the drug. The VTK files include data that is keyed both by *points* and *cells* on the mesh. These are not the same! Our mesh for $Re = 10$ had 7,022,647 cells and 8,578,503 points. One important optimization is to realize that the geometric layout of the mesh does not change between frames. While the files contain the whole grid every time, it is only necessary to save the point and cell position data once. Only the density, velocity, and shear stress change over frames.

`vtk2np.py` extracts and saves the following data from the VTK files as Numpy arrays:

- `point_pos.npy` - the (x, y, z) position of each point on the mesh
- `cell_pos.npy` - the (x, y, z) position of each cell center of the mesh
- `cell_vol.npy` - the volume of each cell in the mesh
- `drug_framenum.npy` - the volume of the drug in each cell of the mesh at this frame
- `drug_point_framenum.npy` - the volume of the drug at each point on the mesh at this frame
- `vel_framenum.npy` - the blood velocity at each point on the mesh at this frame
- `vel_cell_framenum.npy` - the blood velocity at each cell on the mesh at this frame
- `rho_framenum.npy` - the blood density at each cell on the mesh; proportional to pressure

This program was run once locally to create numpy arrays, allowing the next Python program to run without worrying about VTK data structures.

`rbc.py` performs all the requested calculations and generates the 2D plots presented below that were not generated remotely using Paraview.
`drug_delivery` answers the headline question of how much drug is in the stenotic region as a function of time. We estimated this by summing up the density of drug in the cells in the stenotic region. We extracted the velocity of each cell and found that they were all equal to 1. (We knew that the interior cells should have a volume of 1 because they were cubes with side length 1, but didn't know how MUPHY and VTK handle cubes on the boundary.) The total amount of drug present in a region Ω is $\int_\Omega \rho dV$ i.e. the density integrated with respect to infinitesimal volume slices. Therefore, the discrete analog would be to sum up the density of each cell in the stenotic region multiplied by its volume. Since all the cell volumes are equal to 1, we omitted the multiplication from the calculation to improve performance. We also computed the

total quantity of drug in the system as a quality check, expecting it not to change.
`plot_speed_contour` creates a contour plot of the speed on a cross sectional slize $z = z_{plot}$. While we computed drug quantities using cells, we chose to plot speed using points. The main programming challenge is generating a contour for data that doesn't cover an entire rectangular grid. This was done by creating an augmented grid with the full square cross section containing the circular cross section of the artery. Grid points outside of the artery were marked with a speed of 0, which was excluded from the range of values in the contour.
`plot_streamlines` plots streamlines of the velocity in the xy plane (i.e. velocity components u and v) on a cross-sectional slice $z = z_{plot}$. Like the speed contour, this plot uses points rather than cells. It uses the same technique as `plot_speed` for padding the circular region with dummy entries that don't appear on the plot. Streamlines are plotted with a width proportional to their speed in the xy plane.
`plot_drug_conc` plots the concentration of the drug on a cross sectional slice $z = z_{plot}$. This plot uses cell data. A mask is created using the relationship that z at the center of a cell is equal to z at the start of the cell plus $\frac{1}{2}$. This was preserved exactly by VTK so we could create a logical mask efficiently. The conentration of drug at each (x, y) point is then read right off the masked array.
`plot_drug_profile` plots the "profile" of the drug along the z axis for a given time. This calculation is also based on cells. A mask is created for each value of z as before, and the total amount of drug at this longitude is estimated as the sum of the drug concentration array on the mask. An initial attempt at plotting this profile showed some obvious artifacts where a handful of z values had "holes" where the dropped well below their neighbors. We appplied a smoothing operation before plotting the profile $B(z)$ in which we set each b_z equal to the max of itself and its two neighbors b_{z-1} and b_{z+1}.
Plots of each type are displayed below in the Results section.

4 Parameters of the Simulation

These are the parameter values that we set at the start of the Python script for the baseline simulation with 0 red blood cells and a Reynold Number of 10.

- ν = 0.1 — this is the kinematic viscosity
- ρ = 1 — the blood density was set to 1 by convention
- $\bar{u}$ = 0.01 — this is the mean speed of the blood
- Re = 10.0 — the Reynolds number is dimensionless and characterizes the turbulence / regularity of the flow
- Pe = 10.0 — the Pectet number is dimensionless and characterizes the importance of radial to axial diffusion
- R = $Re*\nu/2\bar{u}$ — the radius is implied by Reynolds number and velocity
- DIFFUSIVITY = $\frac{\bar{u}*R}{Pe}$ — the dynamic viscosity applied to the drug fluid
- C0 = 0.01 — the baseline drug concentration away from the bolus
- C1 = 1.0 — the high drug concentration on the bolus at $t = 0$
- NSTEP = 1000000 — total number of simulation steps; each step is one microsecond
- NDIAG = 100 — interval between diagnostics
- NVTKFREQ = 1000 — interval between VTK output frames
- UNFREEZE_TIME = 100000 — number of time steps for the system to equilibrate before drug release
- GROWTIME = 2000 — number of time steps for the RBC to settle before blood flow

Our goal was to also run a big simulation. This was to be similar to the baseline, but would also include a large number of red blood cells. These red blood cells would be simulated as rigid particles with a position and orientation. Including the red blood cells makes the simulation more physiologically realistic, with increasing importance the narrower a blood vessel is. Our initial attempt was a 30% hematocrit with otherwise identical parameters to the baseline case. Unfortunately multiple attempts to simulate the system with large numbers of red blood cells failed. We will explain in detail the various failures and their causes in the section below. Fortunately, we did complete a successful run of the baseline case without red blood cells. We have made the best of a difficult situation by performing a complete analysis on the baseline case.

5 Results

5.1 Failed Runs on Odyssey with Red Blood Cells

We well understand the complexities and potential difficulties of running large parallel software on clusters, and indeed our experience greatly reflected this essence. Though we fail to get our expected results from this run, we reckon the importance of documenting the distinctive errors we received in running the jobs as a good learning experience of high performance computing.

Before we launched our two simulations on the reservation, we did some test runs on smaller systems. Since the $Re = 10$ with 30% RBC has a very large system, which would take much longer time to reach the drug release time, we used the smaller systems to test the applied forces and unfreeze time of the drug. After confirming the case without RBC case worked for a smaller case, we started to test the case with RBC, which is a much bigger system. We first tried on $Re = 5$ with 2% RBC, unfortunately it failed between $t = 901$ and $t = 1001$, which we suspected $Re = 2$ system was too small to be parallelized, therefore we stick to the test runs on $Re = 5$ and $Re = 10$. However, both case also failed between $t = 901$ and $t = 1001$ with the same memory leak issue. Luckily this memory leak was fixed on the first day of scheduled run.

```
Yues-MacBook-Pro-2:Desktop yuesun$ cat *.err
*** Error in `/usr/bin/python2': double free or corruption (!prev): 0x000000000761d160 ***
======= Backtrace: =========
/lib64/libc.so.6(+0x7c619)[0x2ae9496e5619]
/n/scratchlfs/ac290r/blood_cells/MUPHY/MAGIC/BACKEND/SHOP/libmoebius.so(__propagator_mod_MOD_spv_w_tensor+0x710)[0x2a
e95806e480]
/n/scratchlfs/ac290r/blood_cells/MUPHY/MAGIC/BACKEND/SHOP/libmoebius.so(__propagator_mod_MOD_propagate_momenta_fast+0
x150)[0x2ae958071390]
/n/scratchlfs/ac290r/blood_cells/MUPHY/MAGIC/BACKEND/SHOP/libmoebius.so(__molecular_md_cpu_mod_MOD_molecular_md_dd+0x
f45)[0x2ae958102f35]
/n/scratchlfs/ac290r/blood_cells/MUPHY/MAGIC/BACKEND/SHOP/libmoebius.so(updateAtom+0x2b8)[0x2ae9581486b8]
/lib64/libffi.so.6(ffi_call_unix64+0x4c)[0x2ae950178dcc]
/lib64/libffi.so.6(ffi_call+0x1f5)[0x2ae9501786f5]
/usr/lib64/python2.7/lib-dynload/_ctypes.so(_ctypes_callproc+0x30b)[0x2ae94ff65c8b]
/usr/lib64/python2.7/lib-dynload/_ctypes.so(+0xaa85)[0x2ae94ff5fa85]
/lib64/libpython2.7.so.1.0(PyObject_Call+0x43)[0x2ae9489c39a3]
/lib64/libpython2.7.so.1.0(PyEval_EvalFrameEx+0x2336)[0x2ae948a580f6]
/lib64/libpython2.7.so.1.0(PyEval_EvalFrameEx+0x67bd)[0x2ae948a5c57d]
/lib64/libpython2.7.so.1.0(PyEval_EvalFrameEx+0x67bd)[0x2ae948a5c57d]
/lib64/libpython2.7.so.1.0(PyEval_EvalFrameEx+0x67bd)[0x2ae948a5c57d]
/lib64/libpython2.7.so.1.0(PyEval_EvalCodeEx+0x7ed)[0x2ae948a5eefd]
/lib64/libpython2.7.so.1.0(PyEval_EvalCode+0x32)[0x2ae948a5f002]
/lib64/libpython2.7.so.1.0(+0x10043f)[0x2ae948a7843f]
/lib64/libpython2.7.so.1.0(PyRun_FileExFlags+0x7e)[0x2ae948a795fe]
/lib64/libpython2.7.so.1.0(PyRun_SimpleFileExFlags+0xe9)[0x2ae948a7a889]
/lib64/libpython2.7.so.1.0(Py_Main+0xc9f)[0x2ae948a8ba3f]
/lib64/libc.so.6(__libc_start_main+0xf5)[0x2ae94968ac05]
/usr/bin/python2[0x40071e]
======= Memory map: ========
00400000-00401000 r-xp 00000000 fd:00 141633                             /usr/bin/python2.7
00600000-00601000 r--p 00000000 fd:00 141633                             /usr/bin/python2.7
00601000-00602000 rw-p 00001000 fd:00 141633                             /usr/bin/python2.7
02366000-02ea8000 rw-p 00000000 00:00 0                                  [heap]
02ea8000-02ead000 rw-p 00000000 00:00 0                                  [heap]
02ead000-0300a000 rw-p 00000000 00:00 0                                  [heap]
0300a000-0301a000 rw-p 00000000 00:00 0                                  [heap]
0301a000-03082000 rw-p 00000000 00:00 0                                  [heap]
03082000-03083000 rw-p 00000000 00:00 0                                  [heap]
```

Figure 3: Screenshot of a portion of error message due to memory leak.

Having tested on smaller systems, our actual runs still turned into a log of struggles on Odyssey. Various exit codes emerged during the process, and some errors were very unexpected: Having changed nothing to the code, the program suddenly reported Segmentation Fault and MPI Communication error. Even though the popular belief says that "insanity is doing the same thing over and over again and expecting different results", submitting the same job multiple times ended up fixing the error (one success for every

ten failed submissions). The diagnosis is clear: we are definitely insane. On a more serious note, it seems like some part of the program (or Odyssey) displays some Byzantine behaviour.

Now, we will go through our run history. For simplicity, **Re=10, RBC=0%** represents the simulation of 0% red blood cells at Reynolds number =10.

Re=10, RBC=0%: The test runs we had were run on either 64 or 128 cores, when we used the full 1024 cores reservation, it gave continuous segmentation fault when we outputted the first VTK file at $t = 1000$. Then, we tried to specify the partition to be 7, which is partitioning in parallelepipeds for a generic number of MPI tasks. However, this still did not run. We ran on 512 cores with `m.setPartitionAlongXYZ(2,2,128)`, and the job took around 8 hours to finish.

Re=10, RBC=30%: Our initial attempts to run the case with RBC were not very successful, and we waited until the memory leak was fixed to run the actual 30% job. It ran for around 10 hours and failed with ExitCode 1. It failed at the exact time (t=3000) when RBC was starting to move. The emitted signal suggested that the equilibrium was not sufficiently long and the cells collided wildly among themselves. Having read the error log, our instructor suggested this case seemed to require a long equilibrium, which defeats the purpose of the project. Therefore, we focused our RBC simulation on smaller systems.

Re=10, RBC=5,10%: When we switched to simulate on smaller systems, the program managed to go beyond the unfreeze blood flow time ($t = 1.5 \times 2000 = 3000$). However, it failed around around $t = 7001$, which might be caused by releasing the RBC too abruptly. We modified the code to add one more if statement to have a more gentle blood unfreeze process. However, the adjusted program ran for more than 2 hours and failed at $t = 5000$ when RBC was imposed to a slightly bigger force, therefore the RBC releasing process was still too abrupt.

```
elif itime == int(1.5*GROWTIME):
    if myid==0:
        print 'Unfreezing fluid'
    f.setFreeze(False)
    a.setGamma(gammaT=0.001, gammaR=0.001)
    a.setZeroVelocity()

# RBC free to move with right coupling
elif itime == int(2.*GROWTIME):
    if myid==0:
        print 'RBC at right coupling'
    a.setGamma(gammaT=0.01, gammaR=0.01)
    a.setZeroVelocity()

# allow the drug to move
elif itime == int(2.5*GROWTIME):
    if myid==0:
        print 'unfreezing drug'
    c.setFreeze(False)
```

```
elif itime == int(1.5*GROWTIME):
    if myid==0:
        print 'Unfreezing fluid'
    f.setFreeze(False)
    a.setGamma(gammaT=0.0001, gammaR=0.0001)
    a.setZeroVelocity()

# RBC free to move with right coupling
elif itime == int(2.*GROWTIME):
    if myid==0:
        print 'RBC at right coupling'
    a.setGamma(gammaT=0.001, gammaR=0.001)
    a.setZeroVelocity()

# RBC free to move with right coupling
elif itime == int(2.5*GROWTIME):
    if myid==0:
        print 'RBC at right coupling'
    a.setGamma(gammaT=0.005, gammaR=0.005)
    a.setZeroVelocity()

# allow the drug to move
elif itime == int(UNFREEZE_TIME):
    if myid==0:
        print 'unfreezing drug'
    c.setFreeze(False)
```

Figure 4: Left: Original unfreeze blood routine. Right: Updated gentle unfreeze routine.

Re=5, RBC=5% Due to time limit we decided to work on the smallest system possible for this project. However, for some reason we fail to understand, we consistently received segmentation faults when we launched the program. The error log pointed to the fact that the segmentation fault occurred in the module load part, where we never changed since the module loading worked fine for previous runs. In addition to the segmentation fault (Exit Code 139), we also received over a dozen of Exit Code 137, which pointed to MPI communication error (InfiniBand error or failure to find nodes). Though we were unable to pin down

the actual causes of both errors, we discovered that if we submitted the job multiple times, there would be one time that went through (around one in every ten submissions). The most eccentric pattern was that before we successfully ran a job, we would get Exit Code 137 and Exit Code 139 in sequence.

Our final strategy was to submit job continuously until Odyssey accepted it. There were four jobs submitted successfully, and all of them ran for a while and failed. The objective run time is 60000. The error logs were suggesting possible memory issue which we failed to trace back.

Job Name	Time Steps	Error
RBC5RE5	10001	malloc(): memory corruption
RBC5RE5	17201	free(): invalid next size (normal)
RBC5RE5	21501	corrupted double-linked list
RBC5RE5	21501	corrupted double-linked list

Table 1: Error message and elapsed time steps for Re=5, RBC=5%

In conclusion, we notice two possible sources of errors. When we first launch the job, there are chances that the cluster would emit segmentation fault on unexpected commands. And when we successfully launch the job, the software may experience a memory issue as the simulation moves further.

```
[ac290ru1906@boslogin03:/n/scratchlfs/ac290r/blood_cells/BUFFY/RBC_5_Re5$ cat RBC5RE5.err ]
[holy7c01208:04934] *** Process received signal ***
[holy7c01208:04934] Signal: Segmentation fault (11)
[holy7c01208:04934] Signal code:  (128)
[holy7c01208:04934] Failing at address: (nil)
[holy7c01208:04934] [ 0] /lib64/libpthread.so.0(+0xf5e0)[0x2b7e3fce75e0]
[holy7c01208:04934] [ 1] /lib64/libpthread.so.0(pthread_mutex_lock+0x0)[0x2b7e3fce1c30]
[holy7c01208:04934] [ 2] /n/helmod/apps/centos7/Comp/gcc/7.1.0-fasrc01/openmpi/3.1.1-fasrc01/lib64/libopen-pal.so.40(+0xceb92)
[0x2b7e51603b92]
[holy7c01208:04934] [ 3] /n/helmod/apps/centos7/Comp/gcc/7.1.0-fasrc01/openmpi/3.1.1-fasrc01/lib64/libopen-pal.so.40(+0xcf3ac)
[0x2b7e516043ac]
[holy7c01208:04934] [ 4] /n/helmod/apps/centos7/Comp/gcc/7.1.0-fasrc01/openmpi/3.1.1-fasrc01/lib64/libopen-pal.so.40(opal_libe
vent2022_event_base_loop+0x7c1)[0x2b7e5160cde1]
[holy7c01208:04934] [ 5] /n/helmod/apps/centos7/Comp/gcc/7.1.0-fasrc01/openmpi/3.1.1-fasrc01/lib64/libopen-pal.so.40(+0x603ce)
[0x2b7e515953ce]
[holy7c01208:04934] [ 6] /lib64/libpthread.so.0(+0x7e25)[0x2b7e3fcdfe25]
[holy7c01208:04934] [ 7] /lib64/libc.so.6(clone+0x6d)[0x2b7e406f534d]
[holy7c01208:04934] *** End of error message ***
srun: error: holy7c01208: task 225: Segmentation fault (core dumped)
[holy7c01209.rc.fas.harvard.edu:153242] too many retries sending message to 0x06cc:0x00001f95, giving up
[holy7c01208.rc.fas.harvard.edu:04935] too many retries sending message to 0x0577:0x000344fc, giving up
[holy7c01211.rc.fas.harvard.edu:82980] too many retries sending message to 0x06cc:0x00001fcc, giving up
[holy7c01210.rc.fas.harvard.edu:136668] too many retries sending message to 0x0692:0x00035745, giving up
[holy7c01109.rc.fas.harvard.edu:13357] too many retries sending message to 0x064e:0x000112ec, giving up
[holy7c01210.rc.fas.harvard.edu:136696] too many retries sending message to 0x062e:0x00017151, giving up
srun: error: holy7c01209: task 286: Exited with exit code 255
srun: error: holy7c01208: task 226: Exited with exit code 255
srun: error: holy7c01211: task 322: Exited with exit code 255
srun: error: holy7c01109: task 220: Exited with exit code 255
srun: error: holy7c01210: tasks 288,316: Exited with exit code 255
srun: Job step aborted: Waiting up to 32 seconds for job step to finish.
slurmstepd: error: *** STEP 8374309.0 ON holy7c01101 CANCELLED AT 2019-04-28T19:23:06 ***
srun: error: holy7c01208: tasks 224,227-255: Killed
srun: error: holy7c01109: tasks 192-219,221-223: Killed
srun: error: holy7c01210: tasks 289-315,317-319: Killed
srun: error: holy7c01211: tasks 320-321,323-351: Killed
srun: error: holy7c01209: tasks 256-285,287: Killed
srun: error: holy7c01311: tasks 480-511: Killed
srun: error: holy7c01309: tasks 416-447: Killed
srun: error: holy7c01101: tasks 0-31: Killed
srun: error: holy7c01102: tasks 32-63: Killed
srun: error: holy7c01310: tasks 448-479: Killed
srun: error: holy7c01103: tasks 64-95: Killed
srun: error: holy7c01105: tasks 96-127: Killed
srun: error: holy7c01213: tasks 384-415: Killed
srun: error: holy7c01108: tasks 160-191: Killed
srun: error: holy7c01107: tasks 128-159: Killed
srun: error: holy7c01212: tasks 352-383: Killed
```

Figure 5: Screenshot of a portion of error message due of module load segmentation fault.

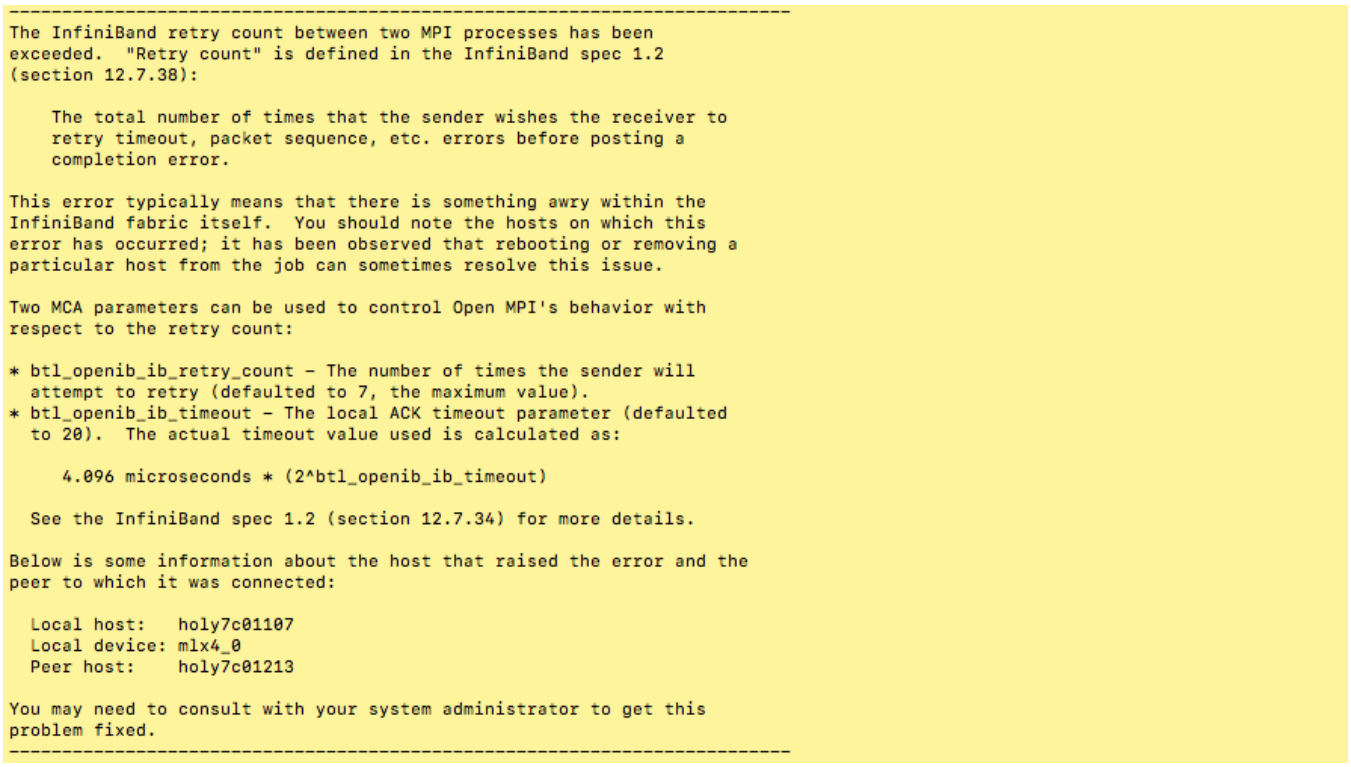


Figure 6: Screenshot of a portion of error message due of MPI communication error.

This project had a very tight schedule due to a combination of the timing of the semester and the limited availability of computational resources. We worked diligently to achieve a successful simulation run with red blood cells, but unfortunately all of our efforts ended in failure. If we had more time and computational resources, we are optimistic that we could achieve a simulation along the lines that we attempted. We detail these failures in the table below. As a quick scan through this table suggests, we experienced a diverse range of problems, including software problems (bugs in the code base); incorrect parameter values that needed to be adjusted; and hardware failures.Based on our job history and SLURM notifications, we listed the errors, run time and possible diagnostics that we encountered during the failed runs for RBC.

Job Name	Run Time	Exit Code	Count	Diagnostic
RBC30RE10			2	MPI communications error.
RBC30RE10	10:30:32	1	1	Equilibration was not sufficiently long.
RBC10RE10	03:49:42	1	1	Segmentation fault (Address not mapped).
RBC10RE10		137	2	Segmentation fault (Address not mapped).
RBC30RE10	01:33:09	0	1	Node fail.
RBC10RE10	01:34:41	137	2	Releasing the cells is too abrupt.
RBC5RE10	02:56:34	1	2	Releasing the cells is too abrupt.
RBC5RE5		137	18	(MPI) InfiniBand retry count exceeded.
RBC5RE5		139	29	Segmentation fault when loading modules.
RBC5RE5	02:08:38	1	1	Adjusted cell release is still abrupt.
RBC5RE5	01:56:47		4	Corrupted double-linked list.
RBC0RE10	02:06:40		1	Segmentation fault when loading modules.

Table 2: History of failed RBC simulations on Odyssey.

Fortunately we did achieve one successful run of our baseline case without red blood cells. In the section

below, we outline our results in that baseline case.

Overall, though our communication with Odyssey on RBC simulation did not go as smoothly as we expected, we still value this project and take it as an excellent learning experience with high performance computing. Here are our key takeaways:

1. Before launch the full-size simulation, it is always good to prepare some smaller-scale test runs. With those test runs we are able to pin down some parameter tuning issue or some potential parallel computing issue. Also test runs allow us to estimate the total run time for the actual simulation.
2. `/n/scratchlfs` may be very unresponsive, and data transfer from Odyssey is very time consuming and slow. Therefore, for post-processing we need to be aware of the size of files and the time of downloading, and decide the best approach to either work remotely on Odyssey or locally.
3. Some errors are expected, therefore we need to be calm and spot the error, or test on smaller scales, or resubmitting the jobs till Odyssey accepts it. However, it seems obvious that at least some part of the whole pipeline is not resilient enough to faults, which are meant to occur in highly distributed systems like this one.

5.2 Drug Delivery Over Time

As described in greater detail in the Description of Code, we computed the quantity of drug in the stenotic region and the whole system by summing the density over cells. This was a simple sum of the density because each cell has a volume of 1.0. In general, we would need to sum over concentration times cell volume. We started by computing the total amount of drug simulated to be in the system, because it should remain constant after the simulation starts. Our first attempt at this calculation incorrectly used points rather cells, and we noticed a marked increase in drug over the first 1.0 second, greater then 10%. This was an indication something was wrong and we realized we needed to use cells. Here is the plot of drug volume in the system. The axis is scaled to start at zero so our eyes can read off the magnitude of the fluctuation in this ostensibly invariant quantity. The simulation has the total quantity increasing by 3.59%.

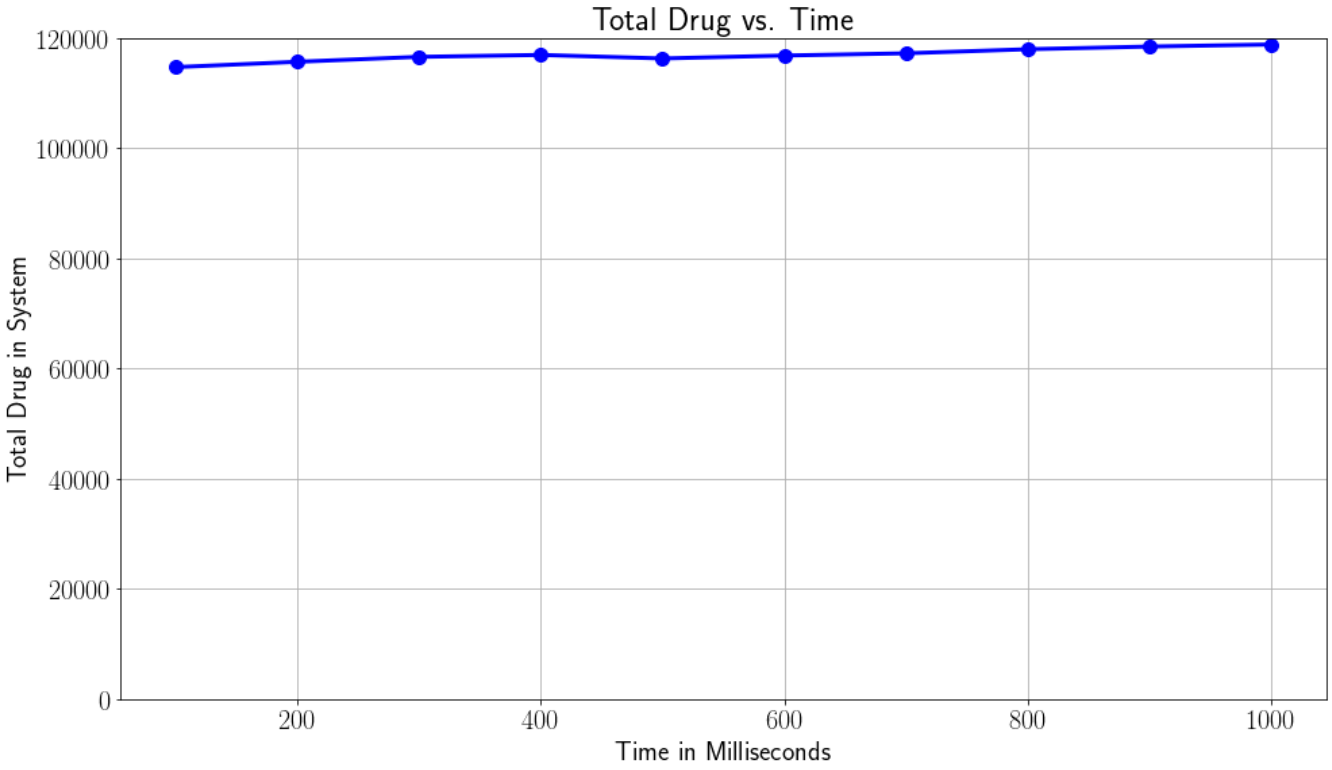


Figure 7: Total Drug Quantity in the System Over Time

Here is the requested plot showing the quantity of drug in the stenotic region. Because the absolute units in this problem are somewhat arbitrary, we are presenting the amount of drug in the system as a fraction of the total. We show the fraction in the stenotic region over time as the blue series. We compare this to a flat line in red, which is the fraction of the total volume occupied by the stenotic region. This was computed by counting the cells in the stenotic region using the same mask we used for the drug quantity. (The sum of volume is a simple cell count since each cell has volume 1.0). When the system reaches equilibrium, we would expect the drug to be completely diffused and at a uniform concentration. In that case, the fraction of drug in the stenotic region would be equal to the volume fraction. By comparing these two quantities, we can develop an intuition as to how much the drug has been delivered vs. its equilibrium. We can see that by the end of the first second, the drug delivery is approaching its equilibrium level, showing that it disperses quite rapidly.

5.3 Velocity Field - Contours of Speed and Streamlines

Here are two plots showing the contours of flow speed at time t=800 milliseconds, when the flow is fairly well established. We plot cross sections at two locations: the center of the stenosis, $z = 500$, and to the right of the stenosis at $z = 800$. At the center of the stenosis, we can see that the flow is fastest in the center, and slowest close to the walls, as we would expect. The difference in speed is less pronounced but still present away from the stenosis. There is also an interesting effect where the speed is greatest not at the center, but to one side. This is not a permanent feature of the flow; the “hot spot” moves around.

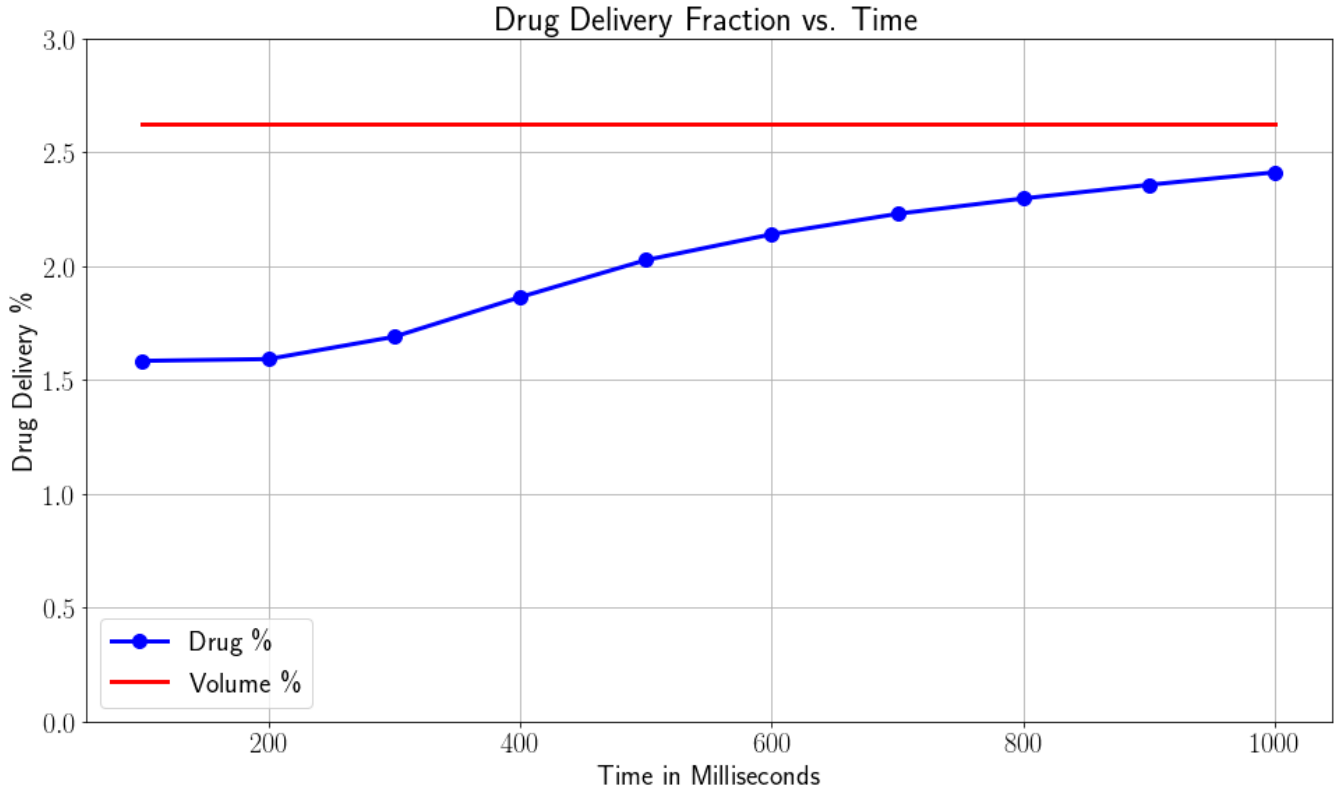
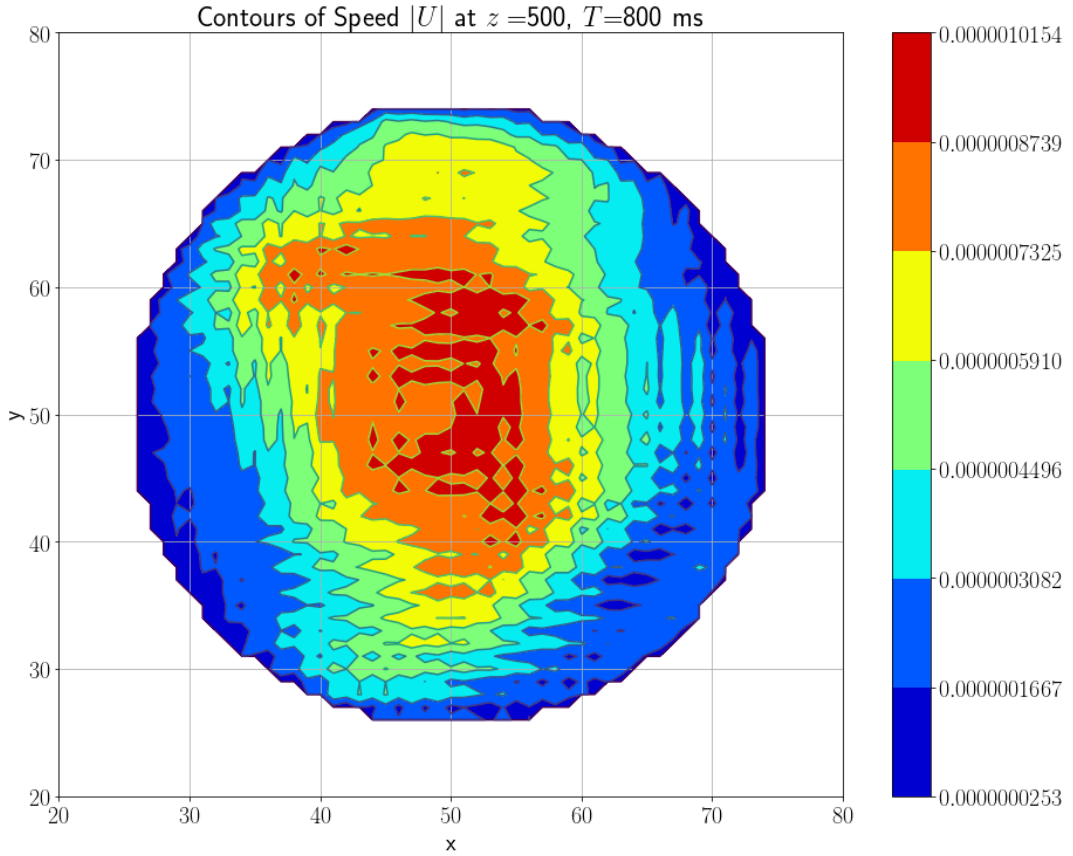


Figure 8: Relative Drug Delivery to Stenotic Region Over Time



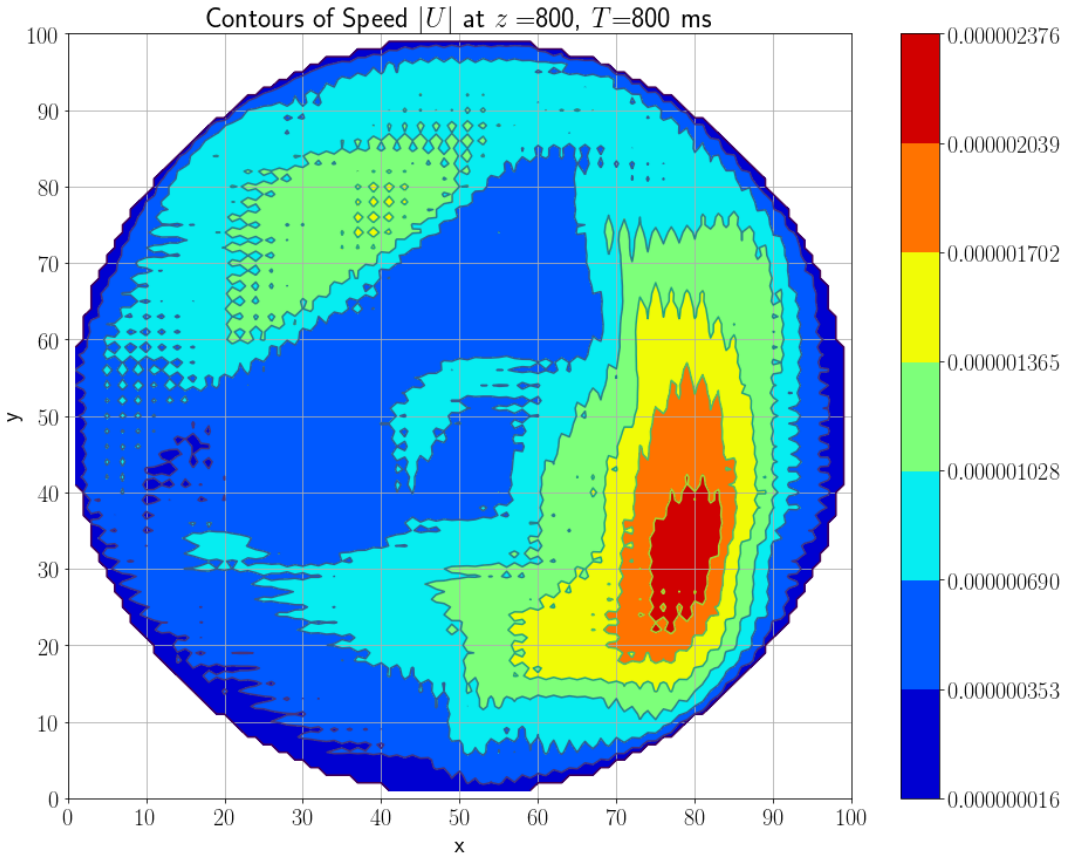
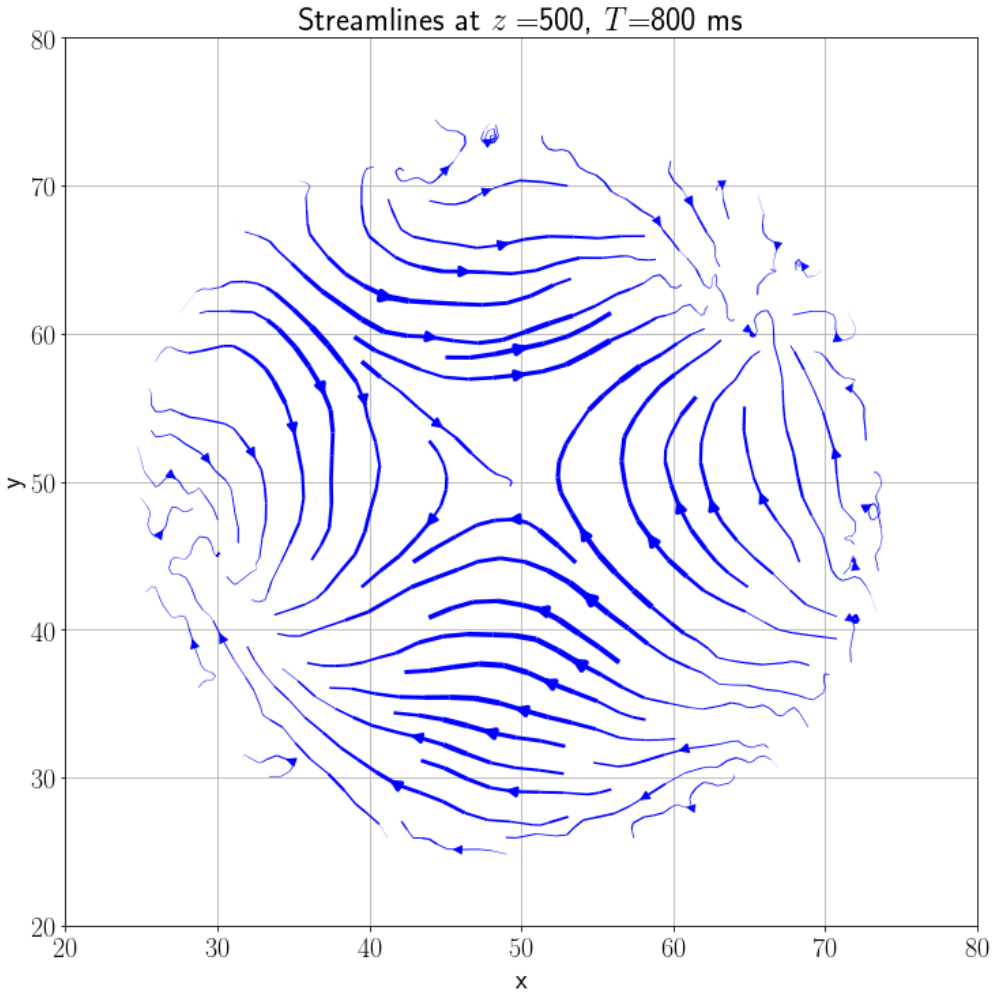


Figure 9: Contours of Speed at time t=800 ms; at two locations, center and right of stenosis

Here are two analogous plots for the same two time instants and z cross sections showing streamlines in the XY plane instead. In looking at these plots, we are becoming suspicious that there may have been a conceptual error in the boundary conditions. Dr. Melchionna suggested that we set the periodicity flag to '111' indicating that all three components x, y and z have periodic boundaries. We had initially planned on setting this parameter of '001' indicating that the z axis was periodic, but x and y were not.

In looking at the last streamline plot above, it seems as if a strong flow has developed in the XY plane that is wrapping around from the southeast corner to the northwest corner. We are writing this remark too late in the development process to re-run our simulation and post-processing pipeline. As scientists we prefer to be honest and express some reservations about one element of this computation than to try to sweep difficulties under the rug. For the purposes of a course project under these time constraints, we believe this is a solid effort that meets the requested requirements. If this were a paper to be submitted to peer review or an analysis that would be used in treating patients, we would need to do additional work to get a definitive answer about whether the periodic boundary conditions on the x and y axes were incorrect.


Streamlines at z =500, T=800 ms
80
70
60
50
40
30
20
y
20
30
40
50
60
70
80
x

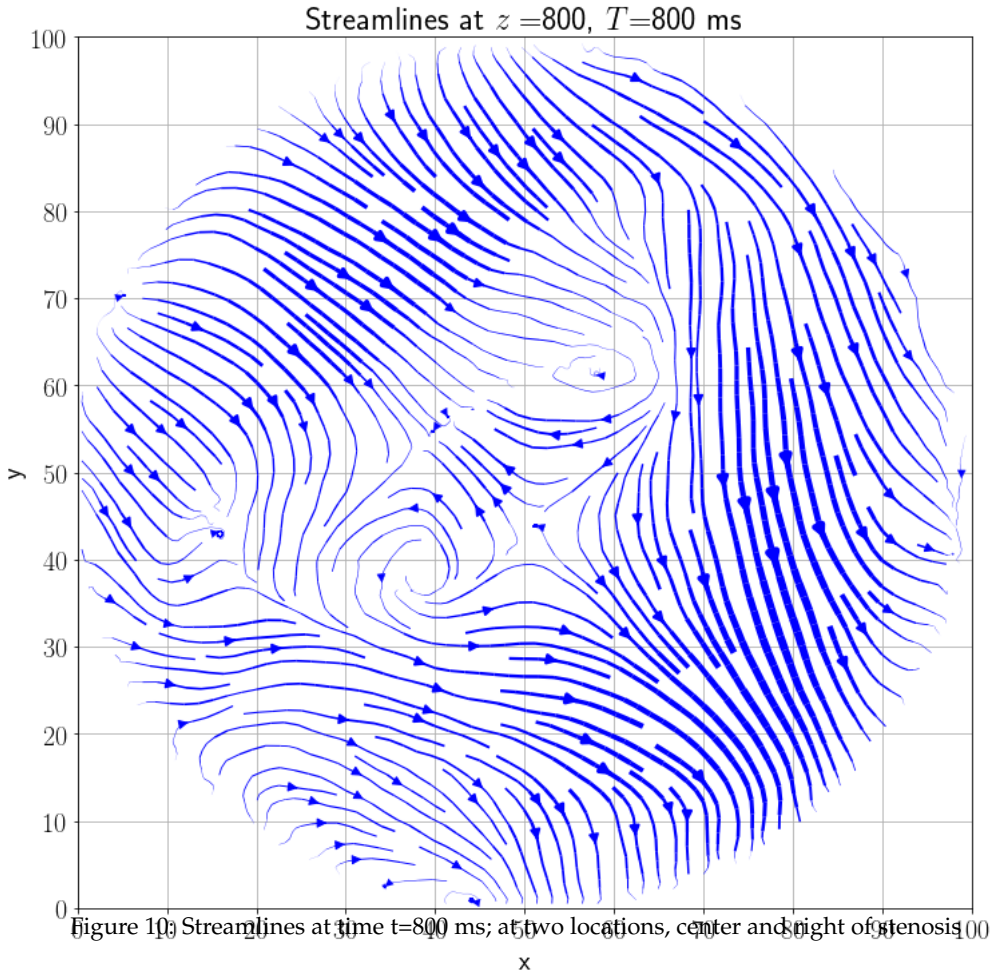
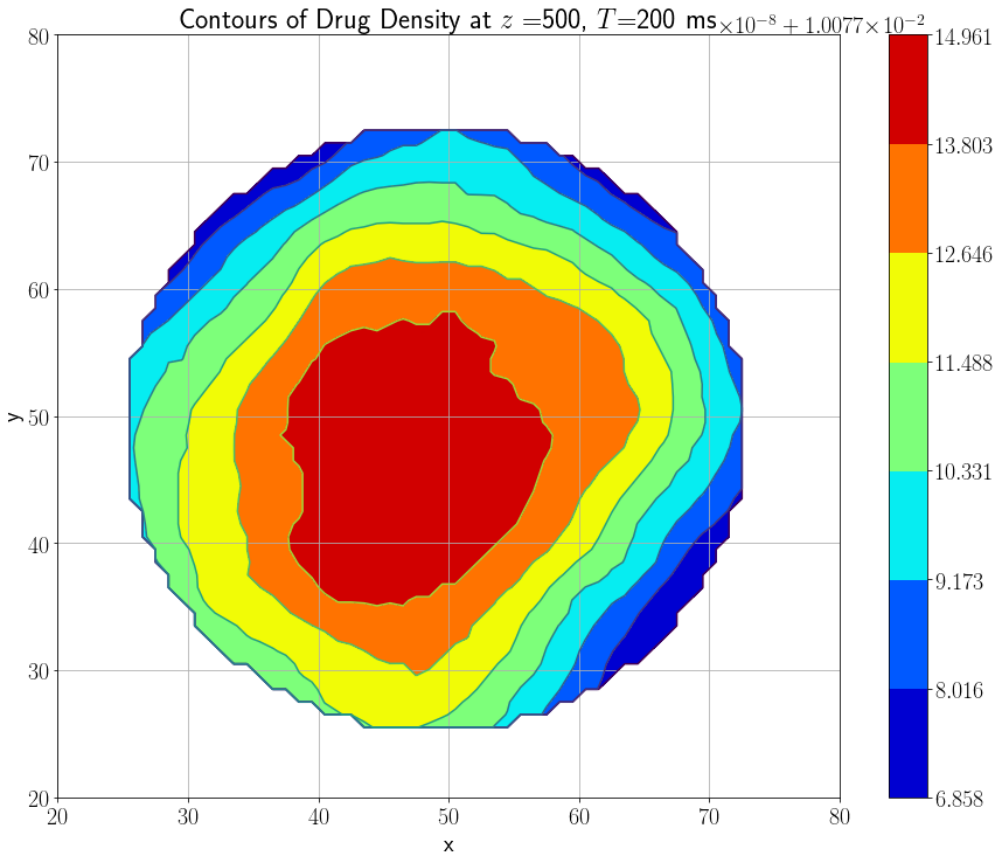


Figure 10: Streamlines at time t=800 ms; at two locations, center and right of stenosis

5.4 Drug Concentration - Contours

Here are two contour plots for the concentration of the drug at the center of the stenosis at $z = 500$. They are run at times of 200 and 800 milliseconds. The scale on these charts is tricky. There is only a very small variation in concentration between the lowest and highest end of the scales. Still the visual pattern is clear, and we can see that the drug concentration is highest in the center and lower on the edges.


Contours of Drug Density at z =500, T=200 ms
$\times 10^{-8} + 1.0077 \times 10^{-2}$
14.961
13.803
12.646
11.488
10.331
9.173
8.016
6.858
80
70
60
50
40
30
20
y
20
30
40
50
60
70
80
x

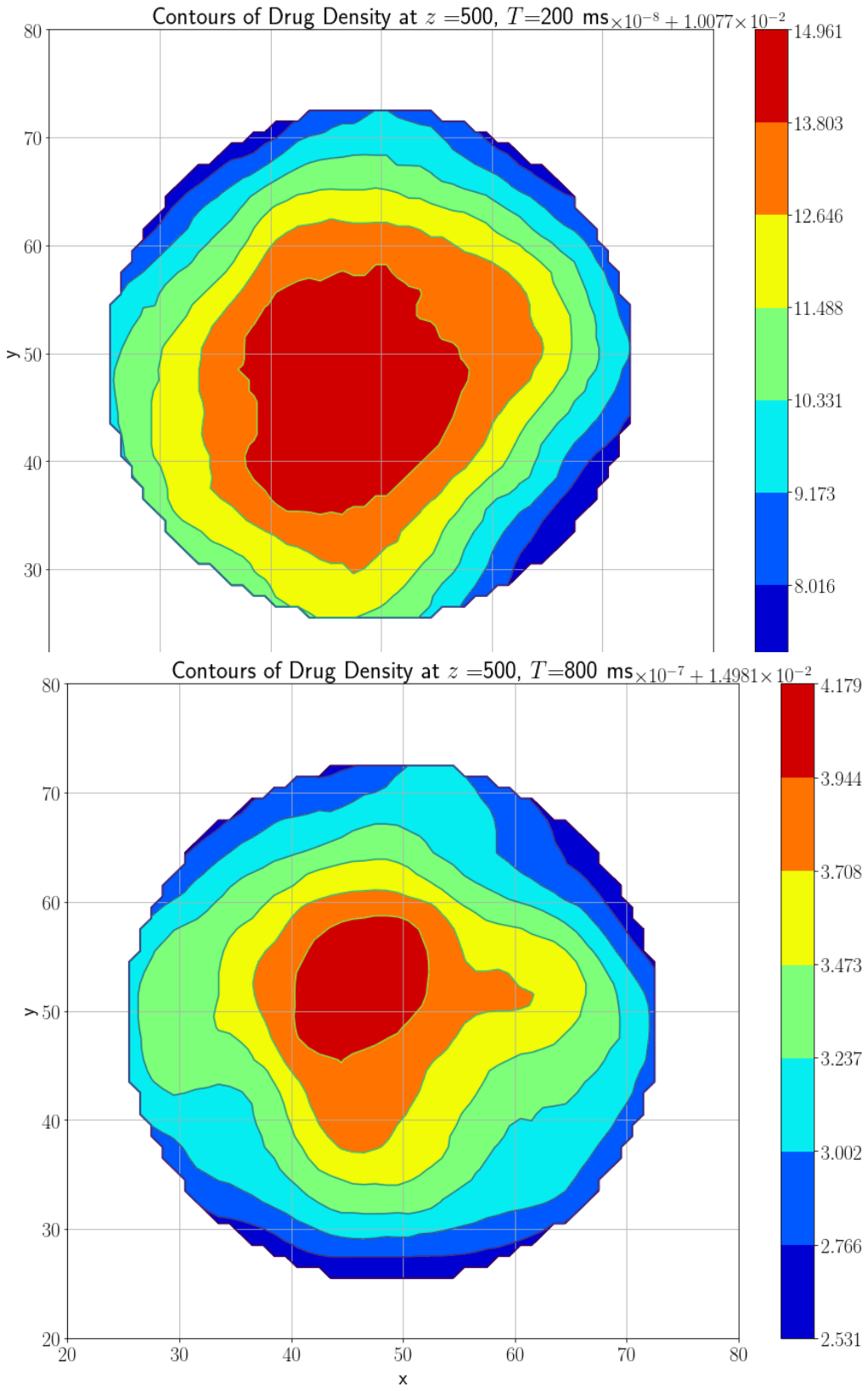
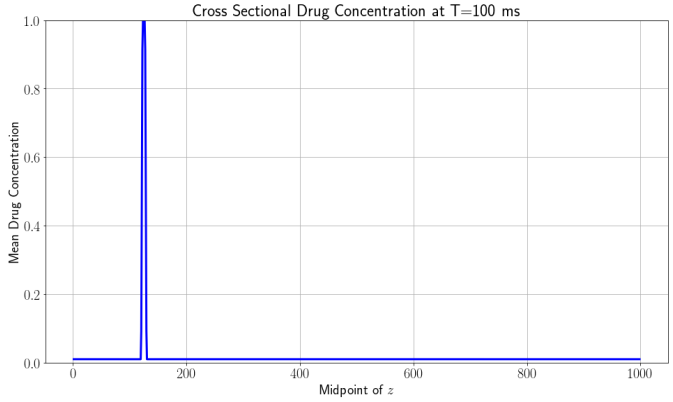
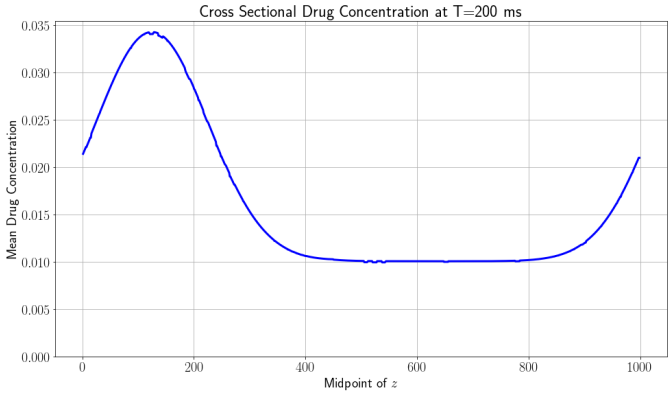


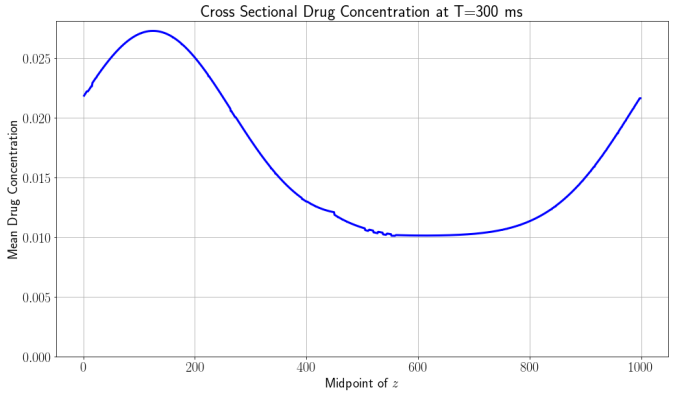
Figure 11: Drug concentration at center of stenosis at two times, 200 and 800 ms

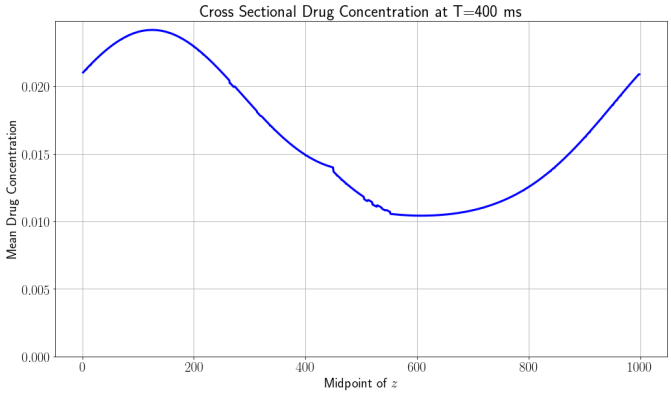
5.5 Longitudinal Drug Profile vs. Time

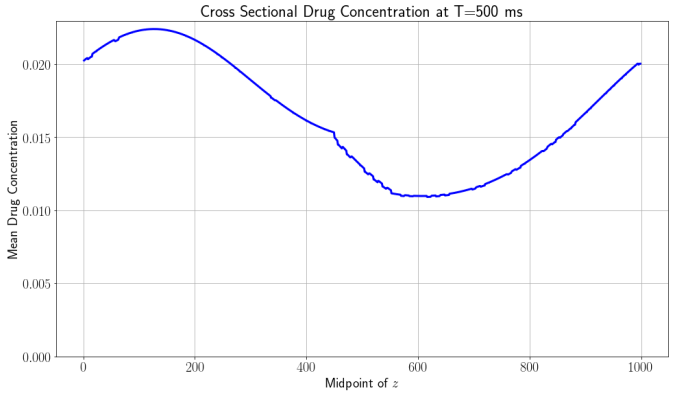
We would like to reiterate that we needed to apply a small degree of smoothing to generate interpretable plots for the drug profile. We computed the concentration as a function of z and t using the finest resolution possible, $z = 1$. Mostly the results were good, with just a few artifacts. We applied a simple filter with a width of $z = 1$ to get a picture for what is really happening in the system. All the plots below are scaled with the y-axis starting at zero. This allows us to compare profiles at different times and develop an intuition about how the bolus migrates left to right and diffuses outward over time. Without a common scaling, all the charts look very similar, and it is much harder to understand the interplay between advection and diffusion. We plot the profile at times of 200, 300, 400, 600, 800 and 1000 milliseconds. The profile at 100 shows a spike with the density at 1.0 in the bolus and 0.01 elsewhere. By 100 milliseconds it has diffused and looks like a normal distribution. Because of the periodic boundary, the left tail of the distribution wraps around to the right side of the plot. Over the succeeding time steps we can see the peak widening out and gradually drifting to the right with the advection. By the time we reach 1.0 seconds the profile has leveled out substantially, though we can still see that concentration in the stenotic region is lower.

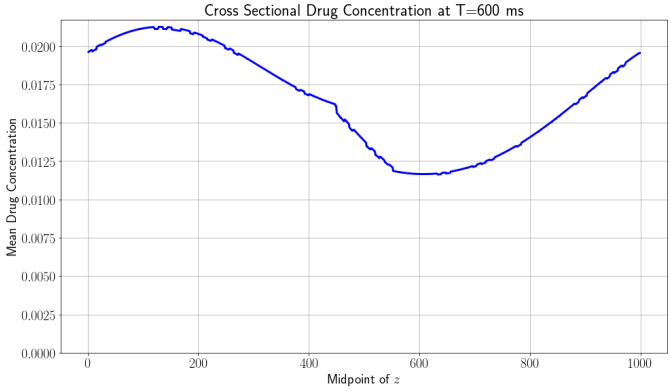

Cross Sectional Drug Concentration at T=100 ms
Mean Drug Concentration
Midpoint of z

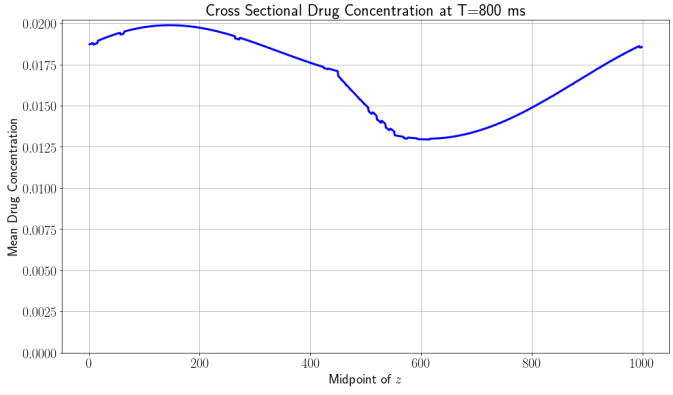

Cross Sectional Drug Concentration at T=200 ms
Mean Drug Concentration
Midpoint of z

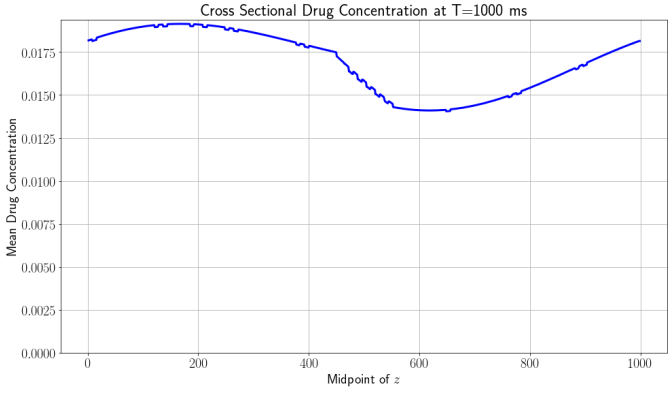

Cross Sectional Drug Concentration at T=300 ms
Mean Drug Concentration
Midpoint of z


Cross Sectional Drug Concentration at T=400 ms
Mean Drug Concentration
Midpoint of z


Cross Sectional Drug Concentration at T=500 ms
Mean Drug Concentration
Midpoint of z


Cross Sectional Drug Concentration at T=600 ms
Mean Drug Concentration
Midpoint of z


Cross Sectional Drug Concentration at T=800 ms
Mean Drug Concentration
Midpoint of z


Cross Sectional Drug Concentration at T=1000 ms
Mean Drug Concentration
Midpoint of z

6 Conclusions and Future Work

6.1 Things we Learned About This Hemodynamic System and Future Work

In our baseline simulation, we found that a system with these physical parameters will get fairly close to equilibrium concentrations within a time horizon of 1.0 second. If these parameters are a reasonably accurate approximation to the hemodynamics of a large artery receiving treatment via a bolus, the high level conclusion is that the specific geometry and flows are probably not that important. The drug molecules are going to diffuse into the stenotic region, and the concentration will be near equilibrium levels by the patient's first heartbeat. The main conclusion from the base case may be that if we want to make useful predictions of drug delivery, we have to use a more realistic model of a human circulatory system with a more complete geometry of the arteries and veins.

In light of our difficulties with the RBC simulation, and the relative lack of importance in larger arterial systems, the most fruitful direction for future work in this specific problem area would probably be for a larger, more physiologically accurate mapping of the circulatory system. Computationally, the optimal direction would most likely be to shift from CPU to GPU simulations.

At this point we must address the elephant in the room: all of our simulations including red blood cells ended in failure. Clearly, these failed simulations didn't tell us anything about the dynamics of this system. They did teach us about Extreme Computing.

6.2 Things We Learned About Extreme Computing

At the risk of a *cliché*, the first conclusion is that Extreme Computing is ... hard. This is the second module, and the second time we experienced major failures on our most ambitious computing job and needed to analyze a smaller job as a fallback plan. While running the computational jobs, we learned some common-sense principles that were also suggested by our instructors.

It is advisable to run smaller test cases before launching a full scale simulation. Some errors may manifest themselves at the smaller scale, and it is less expensive in both time and computing resources to troubleshoot them on a smaller experimental run. A second lesson learned was less obvious to us. Some errors have a non-deterministic (i.e. random) aspect to them, and submitting the same job repeatedly may help. This is quite different from our experience running regular-sized programs on a single PC, which almost always behave the same way. Albert Einstein is said to have quipped that the definition of insanity is doing the same thing over and over again and expecting different results. A corollary may be that only insane people should try Extreme Computing.

As we reach the end of the course, we will close with some more philosophical comments about Extreme Computing. People are more likely to say they do something at the limits than to actually do it. Talk is cheap. What does it mean to do any endeavor in the extreme? The sine qua non is the risk of failure.

In honor of Simone's past work with the Ferrari F1 racing team, we can make this point by analogy. What is the difference between driving a car and racing a car? When we drive, we expect to get from point A to point B, with a preference to go faster, but we are NOT willing to crash. When we watch a professional race car driver on TV, we expect them to go insanely fast, and accept that sometimes they WILL crash (we don't accept that they will get injured anymore). This weekend at the Azerbaijan Gran Prix, the young star on the Ferrari Team Charles Leclerc posted the best time in the last practice session, then proceeded to put his multi-million euro SF90 race car into the wall.

After the crash, Leclerc said “I am so stupid” on the team radio twice. But is he stupid? Of course not! Otherwise he wouldn’t be getting paid millions of euros a year to race cars by the most iconic team in the sport. In order to achieve a lap time fast enough for a shot at pole position, Leclerc needed to take risks and take the preceding corner very close to the wall. He was the second driver of the day to crash in almost the exact same spot. Here is a more hopeful image. It shows Leclerc being congratulated by Lewis Hamilton after his third place finish in Bahrain.

Looking back on this module and this course, we feel like Charles Leclerc, but maybe a bit more like he felt on Saturday than he did on the podium. We've pushed hard and done our best, but we ate the wall more than once.

We will conclude with a page from the playbook of all the best drivers, who always thank their team at the end of the race. We give our sincere thanks to the course staff and the team at Odyssey for sharing their knowledge and responding to our questions at all hours.

"Grazie mille, grazie ragazzi."

References

[1] Succi, Sauro: The Lattice Boltmann Equation for Fluid Dynamics and Beyond