

Towards a Parallel, 3D Simulation of Platelet Aggregation and Blood Coagulation

Elijah Newren

December 30, 2003

Contents

1	Introduction	4
2	Model Components	6
2.1	Mechanical Interactions	6
2.2	Chemical Interactions	7
3	Quick Review of Distributed Memory Parallelism	8
4	Notation	9
5	Fluid Solver	12
5.1	Projection Method	12
5.2	Advection Terms	14
5.3	Discrete Operators	15

5.4	Approximate Projection	15
5.5	Computational Algorithm	19
5.6	Parallel Considerations	21
6	Multigrid	21
6.1	Discretization	22
6.2	Basic Iterative Methods	23
6.3	Improvements	24
6.4	Multigrid Operators Used	27
6.5	Computational and Communication Issues	28
7	Immersed Boundary Method	29
7.1	Immersed Boundary Method Equations	30
7.2	Adding Immersed Boundary Interactions to the Fluid Solver	33
7.3	Parallel Strategies	35
7.3.1	Basic Data Structure	36
7.3.2	Data Requirements	37
7.3.3	Cascade of Issues	38
7.3.4	Final Data Structures and Assumptions	47
7.4	Parallel Implementation	49
7.4.1	Calculating Forces	49
7.4.2	Spreading	50
7.4.3	Interpolation	53
7.4.4	Updating	54

7.5	Alternate Methods	57
8	Future Work	59
8.1	Immersed Interface Method	60
8.2	BDF Methods	62
8.3	Forces Between Connecting Interfaces	64
8.4	Modeling of Red Blood Cells	64
8.5	Chemical Interactions	65
8.6	Adaptive Mesh Refinement	67
8.7	Conclusion	67
A	Optimization of the Navier-Stokes Solver	68
A.1	Quantities Computed Multiple Times	68
A.2	Variable Storage	69
A.3	Optimized Numerical Method	70
B	Cell-Centered Multigrid and Robin BCs	70
B.1	Motivation	70
B.2	System of Equations	72
B.3	Approximating the Boundary Conditions	73
B.4	Filling Ghost Cells for Computing Residuals	74
B.5	Filling Ghost Cells for Interpolation	74
B.6	Filling Ghost Cells for Smoothing	76
B.6.1	Tables of Values	81

B.6.2 Summarized Formulas	82
B.6.3 Why dividing by zero does not occur	85

1 Introduction

Coupled, intricate systems exist to maintain the fluidity of the blood in the vascular system while allowing for the rapid formation of a solid clot to prevent excessive blood loss subsequent to vessel injury [10]. These systems can be invoked as part of the body’s normal defense mechanism against blood loss (a process referred to as hemostasis), but these same systems are also invoked during unwanted, pathological and perhaps life threatening clot formation known as thrombosis. Indeed, these systems can be seen as a delicate balancing act continually occurring to control clot formation and lysis in order to prevent hemorrhage without causing thrombosis [2]. Despite more than a century of research in blood biochemistry, platelet and vascular wall biology, and fluid dynamics, the complexity of blood clotting under flow has prevented quantitative and predictive modeling [13]. Yet quantitative modeling of blood function under flow could have numerous diagnostic and therapeutic uses.

When the wall of a blood vessel is injured, a variety of embedded molecules become exposed. This initiates two interacting processes known as platelet aggregation and blood coagulation. Both of these processes involve multiple subprocesses. Platelet aggregation starts when platelets suspended in the blood, which normally circulate in an inactive state, adhere to damaged tissue and undergo an activation process. During the activation of a platelet, the platelet changes from its rigid discoidal shape to a more deformable spherical form with several long, thin pseudopodia; the platelet’s surface membrane becomes sticky to other activated platelets; and the platelet begins to

release a variety of chemicals into the surrounding plasma. Other platelets then become activated by the various chemicals released by previously activated platelets. This enables them to cohere to nearby platelets, including ones which have adhered to the vessel wall. Blood coagulation involves a cascade of enzymatic reactions with multiple feedforward and feedback loops. The final enzyme in the series is thrombin, which cleaves the plasma protein fibrinogen into fibrin monomers. The fibrin monomers then polymerize and cross-link to form a fibrous mesh that helps give a blood clot its strength.

These processes are not independent; there are important interactions early in the two that couple them. These include the apparently critical role played by the surface membranes of activated platelets in promoting certain coagulation reactions and the role that the coagulation enzyme thrombin plays as a potent platelet activator.

There are also opposing processes. The fluid blood environment surrounding the membrane-bound clotting response is hostile to coagulation proteases, with very high concentrations of multiple plasma inhibitors [27]. Healthy endothelial cells secrete chemicals whose action is to suppress platelet activation [20]. As soon as fibrin polymerization has begun, the fibrinolysis process also begins to decompose the fibrin polymer.

Finally, there are also other components which serve to further couple all these processes. Chemicals diffuse through the plasma and to or from platelet or vessel surfaces; fluid flow carries platelets and chemicals downstream and can, if shear stresses are large enough, rupture a thrombus and cause embolism; and the local vessel geometry can modify the flow field.

Since there are so many highly coupled and disparate components in the blood clotting process—including coagulation and fibrinolysis proteins, activators, inhibitors, availability of receptors on platelet surfaces, fluid flow, diffusion, and local geometry—it is difficult to synthesize a coher-

ent picture of the dynamics of hemostasis using traditional laboratory approaches. This makes mathematical and computational modeling a potentially very fruitful complementary tool.

2 Model Components

Since there are several coupled processes in blood clotting, any full model of the system will have several components. From a high level overview, the basic components of my model will be: fluid (i.e. plasma, which is mostly water), individual platelets, vessel walls, and numerous chemicals—both in solution, on platelet surfaces, and on vessel surfaces. I will be neglecting red blood cells, since they are so numerous and concentrated; however, I will still try to account for them in an approximate way.

2.1 Mechanical Interactions

Platelets are advected by the fluid flow, but they can also deform, cohere to other platelets and adhere to the vessel wall. In doing so, they can exert a force on the fluid and alter the flow. The walls can deform as well and thus also alter the flow. Chemicals do not directly affect the fluid, but they can change the way platelets interact mechanically and thus indirectly change the flow.

To address the mechanical interactions, we need a method for tracking intra- and inter-structure connections, a surface representation for both platelets and the vessel walls, a method for handling the fluid-structure interactions, and a fluid solver.

From a computational viewpoint, dealing with these mechanical interactions alone can result in large and computationally intensive problems. Thus we want to use efficient solvers (such as multigrid with a structured cartesian grid), employ techniques such as adaptive mesh refinement to minimize the size of the system we need to solve, and parallelize the application in order to

distribute the work among many processors.

2.2 Chemical Interactions

Blood clotting involves a long cascade of chemical reactions, involving many chemicals both in solution and on cell or vessel surfaces. The chemicals in solution will react with each other and they will also diffuse and be advected by the fluid; handling all three of these phenomena requires the use of advection-diffusion-reaction equations in the model. These equations will be complicated by the fact that the advection and diffusion routines will need to respect the presence of platelets and walls, because chemicals should not be advected or diffused through those boundaries. This will require special treatment that may result in nonstandard stencils and may require a modified multigrid method or a different solver altogether.

To handle the reactions occurring on cell or vessel surfaces, we will need a surface mesh on each platelet and on the vessel wall, with ODEs to model the reactions at each surface mesh point. There is also the possibility of allowing for transport or diffusion of chemicals on this surface mesh to account for the diffusion of surface molecules to which the chemicals are bound.

There is also coupling between the solution and membrane chemicals. This coupling comes from the diffusion (and perhaps advection) of solution chemicals and the binding/unbinding reaction of surface chemicals. We will need to derive a boundary condition that relates the diffusive flux of the chemical in solution to the binding and unbinding of the chemical on the surface, and then determine how to treat this boundary condition numerically.

3 Quick Review of Distributed Memory Parallelism

There are a variety of multiprocessor machines. Distributed memory machines are ones for which memory is local to a processor and thus no processor has direct access to what another processor stores. This is in contrast to shared memory machines where every processor has access to every other processor's memory. (There are also hybrid distributed/shared memory systems which contain several small shared memory subunits; in fact these are the most common, but they behave most similarly to distributed memory systems.) It tends to be much easier to write parallel applications on shared memory systems, but these systems are vastly more expensive and the memory bus on these systems can become a bottleneck making it hard to scale applications to as many processors.

The *de facto* standard for implementing distributed memory programs is to use MPI (Message Passing Interface) to enable processors to communicate when they need data from each other. The way that an MPI program works is that all processors run the exact same program, but each has a unique *me* identifier. Each processor also knows the total number of processors, p , and uses both *me* and p to set up actions for itself that are different from other processors. When a processor needs information from another processor (or knows that another processor will need information from it), communication is carried out by explicitly calling receive (and send) functions.

The fact that when, where, and how to communicate must be explicitly encoded into programs is what can make distributed memory programming difficult. What adds to this difficulty is that any missing send or receive will result in an indefinite hang, and performance is abysmal if messages are sent synchronously or sequentially. These things also make debugging much more difficult than it would already otherwise be from several different processors doing many different things at once.

4 Notation

We use a regular cell centered cartesian grid which enables both finite difference and finite volume methods to be utilized (we tend to use whichever of those happens to be a more convenient way to view the current sub-problem, resulting in a mixture of methods in the overall algorithm). A sample 2D cell-centered grid can be seen in Figure 1.

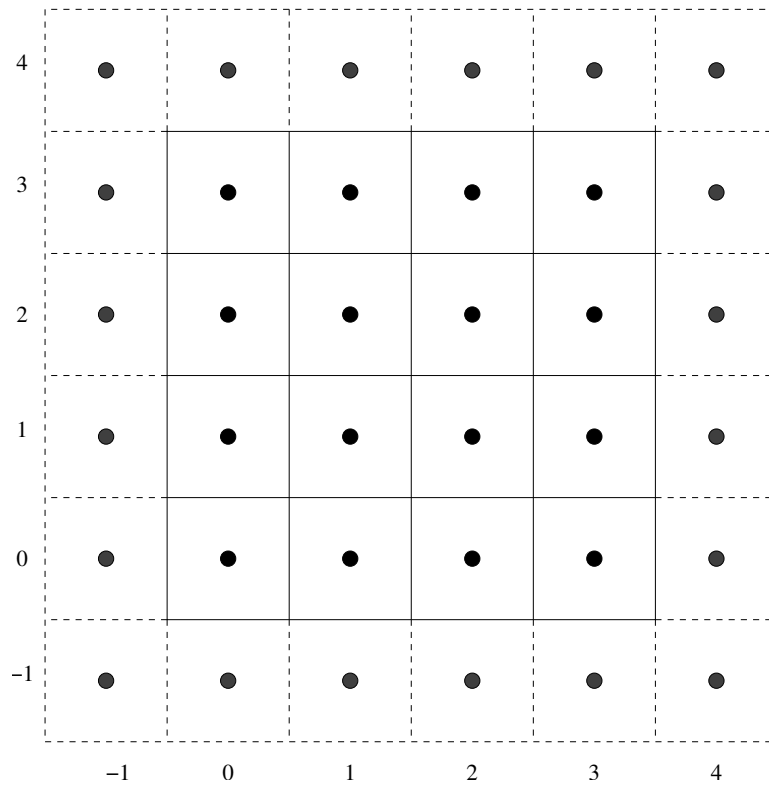


Figure 1: A 4x4 cell centered grid with 1 layer of ghost cells; indices of the various cells in the x and y direction are also shown.

Assuming that the lower left corner of the domain is at the origin, the coordinates of the (i, j)

gridpoint are

$$\left(\left(i + \frac{1}{2} \right) h_x, \left(j + \frac{1}{2} \right) h_y \right), \quad (1)$$

where h_x and h_y are the spacing between gridpoints in the horizontal and vertical directions, respectively. (Note that an x or y subscript denotes a partial derivative when it appears on a variable, but h (which is a constant) is an exception)

Computationally, it is convenient to add a layer (or perhaps even multiple layers) of ghost cells. These ghost cells are an extra layer (or layers) of computational cells surrounding the grid, as shown in Figure 1 (the ghost cells are denoted by the dashed lines). In the case of multiple processors where the domain has been divided up among the processors, most of these ghost cells will correspond to computational cells that another processor owns. These are filled by communication with the owning processor so that local computations can be made. The remainder of the ghost cells correspond to the edge of the physical boundary, and are used to apply any relevant boundary conditions.

Vectors are denoted by boldface letters. \mathbf{u} is used for velocity vectors, with the components of velocity being u , v , and w (if in 3D). \mathbf{u} is a function of both space and time and we use special subscript and superscript notation to refer to discrete values of the components of \mathbf{u} :

$$u_{ijk}^n = u \left(\left(i + \frac{1}{2} \right) h_x, \left(j + \frac{1}{2} \right) h_y, \left(k + \frac{1}{2} \right) h_z, n\Delta t \right), \quad (2)$$

where Δt is the timestep used in the problem.

We also make use of stencil notation. Stencil notation is rather simple; an example using the

discrete Laplacian (in 2D) is

$$\begin{aligned}
 -\Delta^h p(x, y) &= \frac{1}{h^2} [4p(x, y) - p(x - h, y) - p(x + h, y) - p(x, y - h) - p(x, y + h)] \\
 &= \frac{1}{h^2} \begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix}_h p(x, y)
 \end{aligned}$$

or

$$-\Delta^h = \frac{1}{h^2} \begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix}_h . \tag{3}$$

Because many of our operations can be expressed in stencil notation, we often concentrate on a given gridpoint, (i, j, k) , and frequently need to refer to neighboring gridpoints. To make this even easier, we introduce the subscripts

$$\text{up} = i, j + 1, k$$

$$\text{dn} = i, j - 1, k$$

$$\text{lt} = i - 1, j, k$$

$$\text{rt} = i + 1, j, k$$

$$\text{in} = i, j, k - 1$$

$$\text{ot} = i, j, k + 1.$$

5 Fluid Solver

To determine the fluid velocity in our model, we will need to solve the incompressible Navier-Stokes equations,

$$\mathbf{u}_t + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla p + \nu \Delta \mathbf{u} + \mathbf{f} \quad (4)$$

$$\nabla \cdot \mathbf{u} = 0. \quad (5)$$

These equations can be difficult to solve, both because of the nonlinear term, $(\mathbf{u} \cdot \nabla)\mathbf{u}$, and because we have an additional constraint on the velocities instead of an evolution equation for the pressure. In this section, we will outline an iterative method that consists of two basic steps: updating the velocities while ignoring the incompressibility constraint, and correcting the velocities to be divergence free.

5.1 Projection Method

Equation (4) can also be written in the following alternate form, which looks more like a Hodge Decomposition (any vector can be decomposed into the sum of a divergence free field and a gradient field):

$$\mathbf{u}_t + \nabla p = -(\mathbf{u} \cdot \nabla)\mathbf{u} + \nu \Delta \mathbf{u} + \mathbf{f} \quad (6)$$

Note that since \mathbf{u} is divergence free, \mathbf{u}_t will be also. Defining \mathbb{P} as the operator which produces the

divergence free portion of a vector from the Hodge decomposition, this can also be written as

$$\mathbf{u}_t = \mathbb{P}(-(\mathbf{u} \cdot \nabla)\mathbf{u} + \nu\Delta\mathbf{u} + \mathbf{f}) \quad (7)$$

This removes the pressure and the extra constraint on the velocities from the equations and leaves us with just an evolution equation for the velocities. Discretizing this directly is difficult, but it does suggest a method to solve these equations; the type of method is known as a projection method. We begin by discretizing equations (4) and (5) (written in projection form) in time in a Crank-Nicholson fashion. Doing so gives us

$$\frac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\Delta t} + \nabla p^{n+\frac{1}{2}} = -[(\mathbf{u} \cdot \nabla)\mathbf{u}]^{n+\frac{1}{2}} + \frac{\nu}{2}\Delta(\mathbf{u}^{n+1} + \mathbf{u}^n) + \mathbf{f}^{n+\frac{1}{2}} \quad (8)$$

$$\nabla \cdot \mathbf{u}^{n+1} = 0 \quad (9)$$

Rather than solve these equations directly, we instead use the following iterative scheme:

$$\frac{\mathbf{u}^{*,k} - \mathbf{u}^n}{\Delta t} + \nabla p^{n+\frac{1}{2},k} = -[(\mathbf{u} \cdot \nabla)\mathbf{u}]^{n+\frac{1}{2}} + \frac{\nu}{2}\Delta(\mathbf{u}^{*,k} + \mathbf{u}^n) + \mathbf{f}^{n+\frac{1}{2}} \quad (10)$$

$$\frac{\mathbf{u}^{n+1,k} - \mathbf{u}^n}{\Delta t} + \nabla p^{n+\frac{1}{2},k+1} = \frac{\mathbf{u}^{*,k} - \mathbf{u}^n}{\Delta t} + \nabla p^{n+\frac{1}{2},k} \quad (11)$$

$$\nabla \cdot \mathbf{u}^{n+1,k} = 0 \quad (12)$$

As explained further below, the advective term, $-[(\mathbf{u} \cdot \nabla)\mathbf{u}]^{n+\frac{1}{2}}$, is obtained by extrapolating from known values. Equation (10) can be seen as an update to the velocity field that ignores the incompressibility constraint, while equations (11) and (12) together form the projection step that corrects the updated velocity to be divergence free (or approximately so in the case of approximate

projections) while simultaneously giving us a newer corrected pressure. More details about this projection will be given in section 5.4.

5.2 Advection Terms

One of the things we need to solve these equations is a way to determine $\mathbf{u}^{n+\frac{1}{2}}$ in order to compute the advective terms. To do so, we use a method similar to Bell et al. [3], and extrapolate in time (but not in space) by using the following Taylor series expansion

$$\mathbf{u}^{n+\frac{1}{2}} = \mathbf{u}^n + \frac{\Delta t}{2}(\mathbf{u}_t^n) + O(\Delta t^2). \quad (13)$$

Using the Navier-Stokes equations to eliminate the time derivative, this becomes

$$\mathbf{u}^{n+\frac{1}{2}} = \mathbf{u}^n + \frac{\Delta t}{2}(-[\mathbf{u} \cdot \nabla \mathbf{u}]^n + \nu \Delta \mathbf{u}^n + \mathbf{f}^n - \nabla p^n). \quad (14)$$

Note that this can also be written as a decomposition like equation (6).

$$\frac{\mathbf{u}^{n+\frac{1}{2}} - \mathbf{u}^n}{\Delta t/2} + \nabla p^n = -[\mathbf{u} \cdot \nabla \mathbf{u}]^n + \nu \Delta \mathbf{u}^n + \mathbf{f}^n \quad (15)$$

Thus, we also solve for $\mathbf{u}^{n+\frac{1}{2}}$ via an (approximate) projection method as a first step to solving for \mathbf{u}^{n+1} and $p^{n+\frac{1}{2}}$.

5.3 Discrete Operators

The discrete divergence, gradient, Laplacian, and advection operators in 3D are given by

$$\left(\nabla^h \cdot \mathbf{u}\right)_{ijk} = \frac{u_{i+1,j,k} - u_{i-1,j,k}}{2h_x} + \frac{v_{i,j+1,k} - v_{i,j-1,k}}{2h_y} + \frac{w_{i,j,k+1} - w_{i,j,k-1}}{2h_z} \quad (16)$$

$$\left(\nabla^h p\right)_{ijk} = \left(\frac{p_{i+1,j,k} - p_{i-1,j,k}}{2h_x}, \frac{p_{i,j+1,k} - p_{i,j-1,k}}{2h_y}, \frac{p_{i,j,k+1} - p_{i,j,k-1}}{2h_z}\right) \quad (17)$$

$$\begin{aligned} \left(\Delta^h p\right)_{ijk} &= \frac{p_{i+1,j,k} - 2p_{i,j,k} + p_{i-1,j,k}}{h_x^2} + \frac{p_{i,j+1,k} - 2p_{i,j,k} + p_{i,j-1,k}}{h_y^2} \\ &\quad + \frac{p_{i,j,k+1} - 2p_{i,j,k} + p_{i,j,k-1}}{h_z^2} \end{aligned} \quad (18)$$

$$\begin{aligned} \left(\mathbf{u} \cdot \nabla c\right)_{ijk} &= u_{i,j,k} \left(\frac{c_{i+1,j,k} - c_{i-1,j,k}}{2h_x}\right) + v_{i,j,k} \left(\frac{c_{i,j+1,k} - c_{i,j-1,k}}{2h_y}\right) \\ &\quad + w_{i,j,k} \left(\frac{c_{i,j,k+1} - c_{i,j,k-1}}{2h_z}\right) \end{aligned} \quad (19)$$

As a quick side note, the implementation of these operators should support the choice of either taking the computed result and overwriting a certain variable, or adding a multiple of the result to a certain variable as it is computed. This is done to minimize the total number of loops over the gridpoints.

5.4 Approximate Projection

As mentioned earlier, projection methods rely on the Hodge Decomposition (any vector field can be decomposed into the sum of a divergence free field and a gradient field). These methods define a projection operator, \mathbb{P} , as being the operator which returns the divergence free portion of any vector field. The projection is performed computationally by first computing the gradient field

from the solution of a Poisson problem. For example, taking the divergence of equation (11) yields

$$\Delta p^{n+\frac{1}{2},k+1} = \frac{1}{\Delta t} \nabla \cdot \mathbf{u}^{*,k} + \Delta p^{n+\frac{1}{2},k} \quad (20)$$

Everything on the right hand side is known. Using this equation and specifying periodic boundary conditions (other boundary conditions are also possible but require the use of a derived boundary condition for the pressure), we can solve for p which in turn gives us the gradient, ∇p . Equation (11) can be rearranged in order to compute the divergence free field from

$$\mathbf{u}^{n+1,k} = \mathbf{u}^{*,k} + \Delta t (\nabla p^{n+\frac{1}{2},k} - \nabla p^{n+\frac{1}{2},k+1}). \quad (21)$$

When we apply our discrete divergence operator to equation (11) we obtain a Poisson problem where the Laplace operator is given by

$$\begin{aligned} \left(\widetilde{\Delta^h p} \right)_{ijk} &= \frac{p_{i+2,j,k} - 2p_{i,j,k} + p_{i-2,j,k}}{4h_x^2} + \frac{p_{i,j+2,k} - 2p_{i,j,k} + p_{i,j-2,k}}{4h_y^2} \\ &+ \frac{p_{i,j,k+2} - 2p_{i,j,k} + p_{i,j,k-2}}{4h_z^2}. \end{aligned} \quad (22)$$

This is similar to the Laplacian operator defined in equation (19), except that the points in the stencil are separated by $2h$. This wide stencil is problematic because it decouples the Poisson problem into four distinct subgrids. This decoupling introduces a null space containing an oscillatory mode consisting of a different constant on each subgrid. This mode is also in the null space of the divergence operator and a special filtering step is needed to remove it. Also, if multigrid is used, then the multigrid operators must take into account this decoupling.

We wish to avoid this decoupling and so instead we discretize equation (20) directly using the

Laplacian operator we defined in section 5.3. Since that Laplacian operator is not the composition of our discrete divergence and gradient operators, the resulting velocity that we get will not be discretely divergence free, but will rather be only approximately so. This is why the method is referred to as an approximate projection.

We could define different divergence and gradient operators whose composition produces the Laplacian stencil that we would like to use. Doing so would provide us with an exact projection without the decoupling of the Poisson problem. This can be achieved by employing a MAC staggered grid where scalars are located at cell centers and velocities are located at cell edges, with the divergence and gradient operators being defined through straightforward centered differences. This has been done before and the resulting projection is referred to as a MAC projection. In addition to the advantages already listed, the MAC projection also has the norm reducing property, which is used to show that the overall numerical method is stable. We will not use a MAC projection because cell centered velocities are needed for higher resolution advection methods and also make adaptive mesh refinement more straightforward. However, we will show how our approximate projection method (using the discrete operators defined in section 5.3) can be derived from a MAC projection. Making this connection between the different projection methods has the advantage that the well-posedness of the MAC projection guarantees the well-posedness of the approximate projection and it also facilitates the extension of the approximate projection to refined grids. Our approach is adapted from Minion [29].

The basic idea used in constructing an approximate projection, \mathbb{P} , from a MAC projection, $\tilde{\mathbb{P}}$, is to interpolate cell-centered quantities to cell-edges when it would be required in the MAC projection, and to interpolate quantities back from cell-edges to cell-centers when needed. Denoting cell-centered quantities with a superscript c and cell-edge quantities with a superscript e , the

interpolation of the velocity to cell-edges is given (in 3D) by the formula

$$\mathbf{u}_{i,j,k}^e = \left(\frac{u_{i+1,j,k}^c + u_{i,j,k}^c}{2}, \frac{v_{i,j+1,k}^c + v_{i,j,k}^c}{2}, \frac{w_{i,j,k+1}^c + w_{i,j,k}^c}{2} \right) \quad (23)$$

This is used when first applying the projection. As stated above, the first step in applying the projection is to apply the divergence operator $(I - \tilde{\mathbb{P}})$ to get a Poisson problem. Since the MAC divergence operator is given by

$$\tilde{D}(\mathbf{u}^e)_{i,j,k} = \frac{u_{i,j,k}^e - u_{i-1,j,k}^e}{h_x} + \frac{v_{i,j,k}^e - v_{i,j-1,k}^e}{h_y} + \frac{w_{i,j,k}^e - w_{i,j,k-1}^e}{h_z}, \quad (24)$$

the resulting divergence operator is given by

$$\begin{aligned} D(\mathbf{u}^c)_{i,j,k} &= \frac{u_{i,j,k}^e - u_{i-1,j,k}^e}{h_x} + \frac{v_{i,j,k}^e - v_{i,j-1,k}^e}{h_y} + \frac{w_{i,j,k}^e - w_{i,j,k-1}^e}{h_z} \\ &= \frac{\frac{1}{2}(u_{i+1,j,k}^c + u_{i,j,k}^c) - \frac{1}{2}(u_{i,j,k}^c + u_{i-1,j,k}^c)}{h_x} \\ &\quad + \frac{\frac{1}{2}(v_{i,j+1,k}^c + v_{i,j,k}^c) - \frac{1}{2}(v_{i,j,k}^c + v_{i,j-1,k}^c)}{h_y} \\ &\quad + \frac{\frac{1}{2}(w_{i,j,k+1}^c + w_{i,j,k}^c) - \frac{1}{2}(w_{i,j,k}^c + w_{i,j,k-1}^c)}{h_z} \\ &= \frac{u_{i+1,j,k}^c - u_{i-1,j,k}^c}{2h_x} + \frac{v_{i,j+1,k}^c - v_{i,j-1,k}^c}{2h_y} + \frac{w_{i,j,k+1}^c - w_{i,j,k-1}^c}{2h_z} \end{aligned} \quad (25)$$

which is exactly our divergence operator from section 5.3. We can then solve the resulting Poisson problem using the MAC Laplacian operator (which is also the same as the one we defined in section 5.3).

Once we have solved the Poisson problem, we will have a scalar, p to which we can apply the

MAC gradient operator,

$$\tilde{G}^e(p) = (G_1^e(p), G_2^e(p), G_3^e(p)) = \left(\frac{p_{i+1,j,k} - p_{i,j,k}}{h_x}, \frac{p_{i,j+1,k} - p_{i,j,k}}{h_y}, \frac{p_{i,j,k+1} - p_{i,j,k}}{h_z} \right) \quad (26)$$

to get a cell-edge gradient. This cell-edge gradient will then need to be subtracted from the cell centered velocities to complete the projection. In order to do so, we first interpolate back to cell centers using the interpolant

$$G^c(\phi)_{i,j,k} = \left(\frac{G_1^e(\phi)_{i,j,k} + G_1^e(\phi)_{i-1,j,k}}{2}, \frac{G_2^e(\phi)_{i,j,k} + G_2^e(\phi)_{i,j-1,k}}{2}, \frac{G_3^e(\phi)_{i,j,k} + G_3^e(\phi)_{i,j,k-1}}{2} \right). \quad (27)$$

Thus our effective gradient operator becomes

$$G^c(p)_{i,j,k} = \left(\frac{G_1^e(p)_{i,j,k} + G_1^e(p)_{i-1,j,k}}{2}, \frac{G_2^e(p)_{i,j,k} + G_2^e(p)_{i,j-1,k}}{2}, \frac{G_3^e(p)_{i,j,k} + G_3^e(p)_{i,j-1,k}}{2} \right) \quad (28)$$

$$\begin{aligned} &= \frac{1}{2} \left(\frac{p_{i+1,j,k} - p_{i,j,k}}{h_x} + \frac{p_{i,j,k} - p_{i-1,j,k}}{h_x}, \right. \\ &\quad \left. \frac{p_{i,j+1,k} - p_{i,j,k}}{h_y} + \frac{p_{i,j,k} - p_{i,j-1,k}}{h_y}, \right. \\ &\quad \left. \frac{p_{i,j,k+1} - p_{i,j,k}}{h_z} + \frac{p_{i,j,k} - p_{i,j,k-1}}{h_z} \right) \end{aligned} \quad (29)$$

$$= \left(\frac{p_{i+1,j,k} - p_{i-1,j,k}}{2h_x}, \frac{p_{i,j+1,k} - p_{i,j-1,k}}{2h_y}, \frac{p_{i,j,k+1} - p_{i,j,k-1}}{2h_y} \right). \quad (30)$$

This is exactly our gradient operator from section 5.3.

5.5 Computational Algorithm

Putting all of the above together and writing out our approximate projection method for solving the Navier Stokes equations in algorithmic form yields Algorithm 1.

Algorithm 1 Navier Stokes Solver Algorithm

$$\text{Solve } \Delta^h p^n = \nabla^h \cdot (-[\mathbf{u} \cdot \nabla \mathbf{u}]^n + \nu \Delta^h \mathbf{u}^n + \mathbf{f}^n)$$

$$\mathbf{u}^{n+\frac{1}{2}} = \mathbf{u}^n + \frac{\Delta t}{2} (-[\mathbf{u} \cdot \nabla \mathbf{u}]^n + \nu \Delta^h \mathbf{u}^n + \mathbf{f}^n - \nabla^h p^n)$$

$$\nabla^h p^{n+\frac{1}{2},0} = \nabla^h p^n$$

For $k=0, \dots$

$$\text{Solve } \frac{1}{\Delta t} \mathbf{u}^{*,k} - \frac{\nu}{2} \Delta^h \mathbf{u}^{*,k}$$

$$= \frac{1}{\Delta t} \mathbf{u}^n + \frac{\nu}{2} \Delta^h \mathbf{u}^n - [\mathbf{u} \cdot \nabla \mathbf{u}]^{n+\frac{1}{2}} - \nabla^h p^{n+\frac{1}{2},k} + \mathbf{f}^{n+\frac{1}{2}}$$

$$\text{Solve } \Delta^h p^{n+\frac{1}{2},k+1} = \frac{1}{\Delta t} \nabla^h \cdot \mathbf{u}^{*,k} + \Delta^h p^{n+\frac{1}{2},k}$$

End of for loop

$$p^{n+\frac{1}{2}} = p^{n+\frac{1}{2},k+1} - \frac{\nu}{2} \nabla^h \cdot \mathbf{u}^{*,k}$$

$$\mathbf{u}^{n+1} = \mathbf{u}^{*,k} + \Delta t (\nabla^h p^{n+\frac{1}{2},k} - \nabla^h p^{n+\frac{1}{2},k+1})$$

This algorithm can be optimized by storing computed quantities which are used more than once and finding ways to overlap storage of variables and temporaries. See Appendix A for details.

The final correction term $-\frac{\nu}{2} \nabla^h \cdot \mathbf{u}^{*,k}$ for the pressure comes from [12]. In that paper, the authors show that projection methods which do not use this correction result in pressures that are only first-order accurate and derive this correction term to ensure second-order accuracy of the pressures. Since second-order accuracy in the velocities is what is most important, this term could be dropped, but it takes very little computational effort and is very simple to implement so we include it here.

The linear system solves (two Poisson equations and three Helmholtz equations) that appear in this algorithm are solved via multigrid. More details on the multigrid method are in section 6.

5.6 Parallel Considerations

To solve these equations in parallel, the domain must be divided up among several processors. One method of doing this is discussed in [30]. However, in this work we will use the routines built into the SAMRAI package (developed at Lawrence Livermore National Laboratories) to do the domain dividing. Each processor will then own a certain region of the domain as well as some “ghost cells” corresponding to cells adjacent to the owned region but which are actually owned by other processors. Since none of our operators have a stencil involving points more than one away from the center, having a single layer of ghost cells for the various velocity and pressure variables will suffice (for the fluid solver; other components of the full solver may require more layers of ghost cells). A parallel implementation then would merely need to have routines that worked over the owned region and which had communication routines to fill the one ghost cell width of each region before executing the relevant operator. The multigrid solver will also only require a single ghost cell width, although parallel issues for it will be discussed in section 6.

6 Multigrid

Multigrid methods have emerged as a fast way to solve large systems of equations obtained from the discretization of partial differential equations. This chapter is meant as a very brief outline of multigrid for the system solves needed in Algorithm 1. To learn more about multigrid, see [11] for a good tutorial, or [35] for a more thorough treatment.

In this section I will concentrate on Poisson’s equation, but the extension to Helmholtz’s equation is trivial. I will also assume periodic boundary conditions and a cell-centered grid. When trying to deal with other boundary conditions extra issues arise. These issues are discussed in

Appendix B. Finally, I will also be assuming the problem is three-dimensional, although the two-dimensional analog is obvious.

6.1 Discretization

In order to solve the equation

$$\Delta\phi = f \tag{31}$$

numerically, we use standard finite differences. The most straightforward discretization gives us the standard 7-point stencil, so that the equation at each gridpoint (cell center) is given by

$$\frac{u_{lt} + u_{rt} + u_{up} + u_{dn} + u_{in} + u_{ot} - 6u_{i,j,k}}{h^2} = b_{i,j,k} \tag{32}$$

where $b_{i,j,k} = f(ih + \frac{h}{2}, jh + \frac{h}{2}, kh + \frac{h}{2})$ and $u_{i,j,k} = \phi(ih + \frac{h}{2}, jh + \frac{h}{2}, kh + \frac{h}{2})$. Collecting all these equations will give us the linear system

$$Au = b, \tag{33}$$

whose solution can be computed many different ways. Since these systems are extremely sparse the natural choice is to use an iterative method. The basic idea with iterative methods is to repeatedly apply some updating scheme to get a new approximation to the real solution, starting with some initial guess, $u^{(0)}$. The iterates, $u^{(n)}$, should converge to the true solution as $n \rightarrow \infty$.

6.2 Basic Iterative Methods

The basic iterative methods (see [19] for details) are formed through a splitting of the matrix; that is, a decomposition $A = M - N$ with M nonsingular. To see how this generates an iterative method, note that $Au = (M - N)u = b$ implies that $Mu = Nu + b$ or $u = M^{-1}Nu + M^{-1}b$. So we set $u^{(n+1)} = M^{-1}Nu^{(n)} + M^{-1}b$. It can be shown that the iterates will converge if and only if the absolute value of all eigenvalues of the matrix $M^{-1}N$ are less than 1. Obviously, M should be easily invertible if the method is to be feasible.

Two standard iterative methods are Jacobi and Gauss-Seidel. The Jacobi method uses the splitting $M = D$, where D is the diagonal of A , and $N = L + U$, where $-L$ and $-U$ are the strictly lower- and upper-triangular parts of A . So the entire sequence of Jacobi iterates is defined by

$$u^{(n+1)} = D^{-1}(L + U)u^{(n)} + D^{-1}b. \quad (34)$$

For Poisson's equation, this is equivalent to the following component form

$$u_{i,j,k}^{(n+1)} = \frac{1}{6}(u_{it}^{(n)} + u_{rt}^{(n)} + u_{up}^{(n)} + u_{dn}^{(n)} + u_{in}^{(n)} + u_{ot}^{(n)} + h^3 b_{i,j,k}). \quad (35)$$

Gauss-Seidel uses the splitting $M = D - L$ and $N = U$. This gives the sequence of Gauss-Seidel iterations

$$u^{(n+1)} = (D - L)^{-1}(Uu^{(n)} + b). \quad (36)$$

For Poisson's equation, if lexicographic ordering (left to right, then top to bottom, then front to back) is used to order the unknowns, then this is equivalent to the following component form (boundary conditions may change the following for boundary gridpoints)

$$u_{i,j,k}^{(n+1)} = \frac{1}{6}(u_{lt}^{(n+1)} + u_{rt}^{(n)} + u_{up}^{(n+1)} + u_{dn}^{(n)} + u_{in}^{(n+1)} + u_{ot}^{(n)} + h^3 b_{i,j,k}) \quad (37)$$

Using these iterative methods, one can solve Poisson’s equation. However, they tend to converge rather slowly despite rapid initial convergence. The reason for the slowing down of the convergence can be seen from a Fourier analysis, which shows that these iterative methods damp out the high frequency components of the error quickly, while low frequency components decay more slowly [11]. Multigrid methods came about from trying to improve this slow convergence.

6.3 Improvements

Seeing that the iterative methods given so far are rather slow, a strategy is desired to accelerate the convergence. One way to speed up the iterative method, at least initially, is to use a good initial guess for u^0 . One way to obtain a good initial guess is to first perform some iterations (referred to as relaxations or smoothings) on a coarser grid, where a good approximation can be obtained much faster. The approximate solution should then somehow be transferred to the finer grid and used as an initial guess there. Doing this quickly raises the question, “Why not obtain a good initial guess for the coarser grid as well?” Doing so, as well as getting a good initial guess for subsequent coarser grids, yields the strategy called nested iteration shown in Algorithm 2.

Note that the matrices A^h, A^{2h}, \dots never have to be formed, because equations (35) or (37) can be used to update each node individually.

Besides obtaining an approximate solution faster, relaxing on coarser grids has another advantage. Low frequency components on one grid are represented as higher frequency components on a coarser grid. (It is common in the multigrid literature to refer to low frequency components

Algorithm 2 Nested Iteration

1. Start with $m = 2^{\text{gridnum}-1}$ and some initial guess, u^{mh} , on the grid Ω^{mh}
 2. Relax on $A^{mh}u^{mh} = b^{mh}$
 3. “Transfer” the updated guess, u^{mh} to the next finer grid, $\Omega^{\frac{m}{2}h}$
 4. If $m \neq 1$ go to step 2
 5. Relax on $A^hu^h = b^h$ to obtain an approximation to the true solution
-

as “smooth.”) This means that the smooth components of the error on the finest grid could be “killed” if they were first represented on a coarse enough grid. Performing a number of relaxations on the coarse grid will damp these high frequency components quickly, and the approximate solution obtained can be transferred to the next finer grid. Then the remainder of the nested iteration algorithm can be carried out (relax on this grid, interpolate up to the next finer one, and repeat this until the finest grid is reached; then relax until the given tolerance needed is met).

In the nested iteration algorithm, it is hoped that the relaxations on the coarser grids ensure that there are no smooth components to the error when the finest grid is reached. If any do remain, then the convergence will stall. If the convergence did start stalling, it would be nice to have some way to go back down to the coarser grids so that the smooth components could be damped again. Making use of the residual equation allows us to do that. The residual equation is simply

$$Ae = r = b - Au, \tag{38}$$

where e is the difference between u , the approximate solution, and \bar{u} , the true solution (i.e., $e = \bar{u} - u$). Note that if we can solve for e , then we can obtain the true solution by merely adding it to our current value of u . Since the error becomes smooth after a few relaxations, the error can be

represented accurately on a coarser grid. This means that we can solve a smaller problem to get a good approximation to the error and use that to obtain a more accurate approximation to the true solution. This process is known as the coarse grid correction scheme and is shown in Algorithm 3.

Algorithm 3 Coarse Grid Correction Scheme

1. Relax on $A^h u^h = b^h$ a few times
 2. Compute the residual $r^h = b^h - A^h u^h$
 3. “Transfer” r^h onto the coarser mesh Ω^{2h} to obtain r^{2h}
 4. Solve $A^{2h} e^{2h} = r^{2h}$
 5. “Transfer” e^{2h} back to the finer mesh Ω^h to obtain e^h
 6. Correct u^h by adding the correction e^h (i.e. $u^h = u^h + e^h$)
-

The “Transfer” operations will be referred to as I_{2h}^h for transfers from a coarser grid to a finer one, and I_h^{2h} for transfers from a finer grid to a coarser one. Various interpolation schemes can be used for these operators; simple ones are often chosen for the job. In Algorithm 3, it is unspecified how to do step 4. Applying a recursive idea in order to handle this step results in what is called the V-cycle scheme, shown in Algorithm 4. In the algorithm, *first_count* and *end_count* are some predefined constants, typically less than or equal to 3.

Algorithm 4 V-cycle

Vcycle(u^h, b^h)

1. Relax *first_count* times on $A^h u^h = b^h$ given some initial guess
 2. If $\Omega^h =$ coarsest grid, go to (4)
 Otherwise Vcycle($0, I_h^{2h}(b^h - A^h u^h)$)
 3. $u^h = u^h + I_{2h}^h u^{2h}$
 4. Relax *end_count* times on $A^h u^h = b^h$
-

Finally, combining the V-cycle algorithm with nested iteration (using coarser grids to obtain a

Algorithm 5 Full Multigrid V-Cycle

Multigrid_Vcycle(u^h, b^h)

1. If $\Omega^h =$ coarsest grid, go to (3)
 Otherwise Multigrid_Vcycle($0, I_h^{2h}(b^h - A^h u^h)$)
 2. $u^h = u^h + I_{2h}^h u^{2h}$
 3. Call Vcycle(u^h, b^h) *num_cycles* times
-

good initial guess for the first fine grid relaxation in the V-cycle), we arrive at the full multigrid V-cycle algorithm. This algorithm (Algorithm 5) is initially called on the finest grid (where the right hand side is defined), with an all zero approximate solution. Although the algorithm starts at the finest grid, it immediately restricts the problem down to the coarsest grid—this is done without performing any relaxations and serves only to define b on the coarser grids. Other than this detail, the algorithm starts on the coarsest grid and works its way up by performing V-cycles on a given level and then interpolating the result to the next finer grid. In the full multigrid V-Cycle algorithm, *num_cycles* is some predefined constant, typically 1.

6.4 Multigrid Operators Used

The “Transfer” operations, I_h^{2h} and I_{2h}^h , will be simple averaging of nearest neighbors and trilinear interpolation, respectively. The smoothing operator used will be Red-Black Gauss-Seidel (RBGS). RBGS only differs from regular Gauss-Seidel in the ordering of the unknowns. In RBGS, half the cells are assigned a red color and half the cells are assigned a black color, with red cells adjacent only to black cells and with black cells adjacent only to red cells (i.e. a checkerboard pattern). The red cells are all numbered before the black cells. The net effect of RBGS is that all red cells can be updated in parallel and that once those cells are updated, the black cells can then be updated

in parallel.

6.5 Computational and Communication Issues

In almost every multigrid solve for Algorithm 1, we will already have a good approximate guess for the solution (we can use a velocity or pressure from a previous time). Thus, we will not be using full multigrid (Algorithm 5) to solve in most cases as Vcycles alone (Algorithm 4) should be sufficient. The only exception will be the initial pressure for the initial timestep, since we do not have a good initial guess for it.

Since the stencils of the restriction, interpolation, and smoothing operators are all compact, a single ghost cell width is sufficient for running multigrid in parallel. Thus a parallel implementation of the multigrid solver can be obtained by writing the operators so that they work over just the owned region of cells and which have communication routines to fill the one ghost cell width of each region before operating on the region. However, three additional issues with the parallel algorithm do arise. These issues affect (1) the RBGS smoother, (2) the interpolation and restriction operators, and (3) the speedup of the parallel multigrid algorithm.

The RBGS smoother will not be able to just set the ghost cells once and then smooth over both red and black cells, because any ghost cells corresponding to red cells must be updated before adjacent black cells are updated. Thus the RBGS smooth will need to fill the ghost cells before smoothing the red cells and again before smoothing the black cells.

The cells to which values are interpolated or restricted may not be owned by the same processor depending on the domain division algorithm. In [30], the domain division algorithm ensured that the cells which were interpolated or restricted to were indeed owned by the same processor; in fact, special formulas were also derived to determine conditions under which certain ghost cells would

not need to be filled for a particular restriction or interpolation. However, that algorithm was fairly specialized and would not work well in the context of AMR. SAMRAI automatically takes care of this issue for the user by providing a temporary level for the interpolation and restriction operators to work with; SAMRAI then automatically copies values from the temporary level to the real target level, communicating if necessary when those values are not owned by the same processor.

Multigrid might actually be slower in parallel than in serial, or at least certain parts of it can be. The purpose of parallelizing an algorithm is to speed up the overall algorithm by splitting the computational work among many processors. So long as computational time dominates communication time, a reasonable speedup should be achieved. However, the coarsest grids in multigrid should always require little computational work and thus there are probably a number of coarse grids over which communication time dominates computation time. This was avoided in [30] by picking a certain coarse level that, when reached in the multigrid algorithm, would be broadcast among all processors so that all processors owned a full copy of the given level. Then, all processors would simultaneously run a “serial” version of multigrid on that level and all coarser levels. For n processors, this would result in the same work being done n times; however this redundant work would be fairly small if the given level chosen is coarse enough and it would also avoid the costly communication costs. SAMRAI does not do anything similar for the end user. However, implementing the same idea should be possible and not overly complex.

7 Immersed Boundary Method

The Immersed Boundary Method was introduced by Peskin in the early 1970’s to solve the coupled equations of motion of a viscous, incompressible fluid and one or more massless, elastic surfaces or

objects immersed in the fluid [31]. Rather than generating a curve-fitting grid for both exterior and interior regions of each surface at each timestep and using these to determine the fluid motion, Peskin instead employed a uniform cartesian grid over the entire domain and discretized the immersed boundaries by a set of points that were *not* constrained to lie on the grid. The key idea that permits this simplified discretization is the replacement of each suspended object by a suitable contribution to a force density term in the fluid dynamics equations in order to allow a single set of fluid dynamics equations to hold in the entire domain with no internal boundary conditions.

The Immersed Boundary Method was originally developed to model blood flow in the heart and through heart valves [31, 32, 33], but has since been used in a wide variety of other applications, particularly in biofluid dynamics problems where complex geometries and immersed elastic membranes or structures are prevalent and make traditional computational approaches difficult. Examples include platelet aggregation in blood clotting [17, 15], swimming of organisms [16, 15], biofilm processes [14], mechanical properties of cells [1], and cochlear dynamics [9].

7.1 Immersed Boundary Method Equations

In the Immersed Boundary method, an Eulerian description based on the Navier-Stokes equations is used for the fluid dynamics on the grid, and a Lagrangian description is used for each object immersed in the fluid. The boundary is assumed to be massless, so that all of the force is transmitted directly to the fluid. An example setup in 2D with a single immersed boundary curve Γ is shown in Figure 2. Lowercase letters are used for Eulerian variables, while uppercase letters are used for Lagrangian variables. Thus, $\mathbf{X}(s, t)$ is a vector function giving the location of points on Γ as a function of arclength (in some reference configuration), s , and time, t . The boundary is modeled by a singular forcing term, which is incorporated into the forcing term, \mathbf{f} , in the Navier-Stokes

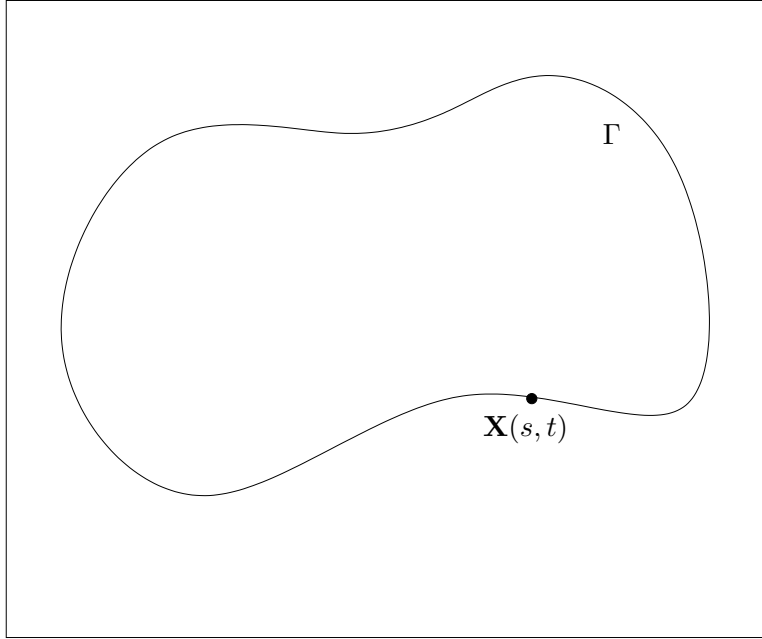


Figure 2: Example immersed boundary curve Γ described by the function $\mathbf{X}(s, t)$.

equations. The Navier-Stokes equations are then solved to determine the fluid velocity throughout the domain. Since the immersed boundary is in contact with the surrounding fluid, its velocity must be consistent with the no-slip boundary condition. Thus grid velocities are interpolated back to the immersed boundary and are used to update its location. This results in the following set of equations, in addition to the Navier-Stokes equations:

$$\mathbf{F}(s, t) = \text{Some function of } \mathbf{X}(s, t) \text{ and its derivatives} \quad (39)$$

$$\mathbf{f}(x, t) = \int_{\Gamma} \mathbf{F}(s, t) \delta(\mathbf{x} - \mathbf{X}(s, t)) ds \quad (40)$$

$$\frac{d\mathbf{X}}{dt} = \mathbf{u}(\mathbf{X}(s, t), t) = \int_{\Omega} \mathbf{u}(x, t) \delta(\mathbf{x} - \mathbf{X}(s, t)) d\mathbf{x}. \quad (41)$$

We will return to the definition of the boundary force density function, $\mathbf{F}(s, t)$, momentarily.

These equations are handled computationally by introducing a discrete delta function whose support is comparable to the mesh spacing. The immersed boundary is divided into a discrete set of points, \mathbf{X}_k . The discretized equations are then given by

$$\mathbf{f}_{ij} = \sum_k \mathbf{F}_k \delta_h(\mathbf{x}_{ij} - X_k) \quad (42)$$

$$\frac{d\mathbf{X}_k}{dt} = \mathbf{U}_k = \sum_{ij} \mathbf{u}_{ij} \delta_h(\mathbf{x}_{ij} - X_k) h^2 \quad (43)$$

$$\delta_h(x, y) = \delta_h(x) \delta_h(y) \quad (44)$$

$$\delta_h(x) = \begin{cases} \frac{1}{4h} (1 + \cos(\frac{\pi x}{2h})) & |x| \leq 2h \\ 0 & |x| \geq 2h \end{cases}. \quad (45)$$

The discrete delta function is derived from the requirement that a certain set of properties be satisfied (e.g. finite support, conservation of momentum, etc.); see [31] for more details on the derivation of this delta function. Other delta functions, such as the one derived in [34] with smaller support, could also be used.

For a boundary under tension, the strength of the force on the boundary is given by

$$\mathbf{F}(s, t) = \frac{\partial}{\partial s} (T(s, t) \boldsymbol{\tau}(s, t)), \quad (46)$$

where $T(s, t)$ is the tension at the given point and $\boldsymbol{\tau}(s, t)$ is the tangent vector to the boundary at

that point (see [33] for a derivation). The tangent vector and tension are given by

$$\boldsymbol{\tau}(s, t) = \frac{\partial \mathbf{X}}{\partial s} / \left\| \frac{\partial \mathbf{X}}{\partial s} \right\| \quad (47)$$

$$T(s, t) = T_0 \left(\left\| \frac{\partial \mathbf{X}}{\partial s} \right\| - 1 \right). \quad (48)$$

The most straightforward discretization of the force on the boundary is to compute a difference in tension on each side of a given link. This can be written as

$$\mathbf{F}_k = \sum_i T_0 (\|X_i - X_k\| - \ell_0) \frac{X_i - X_k}{\|X_i - X_k\|}, \quad (49)$$

where i ranges over all the Immersed Boundary points which are connected to Immersed Boundary point k , and ℓ_0 is the resting length of the “spring” connecting Immersed Boundary points i and k . Writing the force in this manner also makes it clear how to handle links between surfaces (which result in control points having three or more attached links).

7.2 Adding Immersed Boundary Interactions to the Fluid Solver

A variety of ways to incorporate the above equations into an algorithm to solve the Immersed Boundary equations have been used in the past. Explicit methods are perhaps the easiest, and a simple example is given in Algorithm 6.

The additional work beyond the Navier-Stokes solver required in this explicit formulation of the Immersed Boundary method falls into four steps: (1) calculation of forces on the immersed boundary, (2) spreading forces from the immersed boundary to the grid, (3) interpolating velocities from the grid to the immersed boundary, and (4) updating immersed boundary points. When we

Algorithm 6 Immersed Boundary Solver Algorithm

$$\mathbf{F}_k = \sum_i T_0(\|X_i - X_k\| - \ell_0) \frac{X_i - X_k}{\|X_i - X_k\|}$$

$$\mathbf{f}_{ij}^n = \sum_k \mathbf{F}_k \delta_h(\mathbf{x}_{ij} - X_k)$$

Solve for \mathbf{u}^{n+1} using \mathbf{u}^n and \mathbf{f}^n in Algorithm 1

$$\mathbf{U}_k^{n+1} = \sum_{ij} \mathbf{u}_{ij}^{n+1} \delta_h(\mathbf{x}_{ij} - X_k) h^2$$

$$\mathbf{X}^{n+1} = \mathbf{X}^n + \Delta t \mathbf{U}^{n+1}$$

discuss the issues in implementing this solver in parallel, we will address these four steps individually.

Note that $\mathbf{f}^{n+\frac{1}{2}}$ in the right hand side of the Helmholtz solves of Algorithm 1 must be replaced with \mathbf{f}^n in this explicit formulation, because we do not know $\mathbf{f}^{n+\frac{1}{2}}$. Also, the update of the immersed boundary is only first order in time. Getting a higher order, stable discretization is not trivial, because we use the velocities from the end of the timestep and apply them to the positions of the immersed boundary from the beginning of the timestep. Stated another way, if we denote the velocity at the immersed boundary by $\mathcal{U}(\mathbf{X})$, then the update term \mathbf{U}^{n+1} that we use is more precisely written as $\mathcal{U}^{n+1}(\mathbf{X}^n)$. Getting a higher order, stable discretization would require the use of $\mathcal{U}^{n+1}(\mathbf{X}^{n+1})$ which is difficult to obtain because it requires knowledge of the final position of the interface before computing the forces and the velocity.

Besides yielding a low order discretization, this explicit formulation can also lead to unstable results due to the inherent stiffness of the problem if the timestep is too large. Various semi-implicit and fully implicit methods have been used to avoid these problems. For example, [31] uses an approximately implicit method that computes the elastic force from an estimate of \mathbf{X}^{n+1} instead of \mathbf{X}^n ; [36] compares the approximately implicit and explicit methods with a third method

that computes the force from \mathbf{X}^{n+1} but applies it to the configuration of the boundary at the beginning of the timestep; [28] introduces a fully implicit scheme that solves the implicit equations for the immersed boundary locations through a fixed-point iteration; [34] introduces yet another fully implicit scheme that also employs a fixed-point iteration; and [22] introduces a fully implicit scheme for a hybrid Immersed Boundary/Immersed Interface (IB/II) method that uses Quasi-Newton methods to solve the implicit equations for immersed boundary locations. Most of the semi-implicit or fully implicit methods either suffer from stability problems or are prohibitively expensive. The Quasi-Newton methods, however, appear to give good results and future work will involve incorporating them to build an implicit IB/II solver.

7.3 Parallel Strategies

In order to parallelize the Immersed Boundary method, we need to decide how to divide up the data and computations among the given processors, and determine when, how, and what to communicate among the processors so that computations can go forward. The main concern is that communication can be costly and we need to minimize it in order to benefit from distributing the work. On modern machines, bandwidth is not a concern but latency is crucial. Thus, we do not need to worry about the size of the messages passed between processors, but we should closely watch both the number of messages sent and the number of other processors with which a given processor must send and receive messages.

A very simple approach to parallelizing the Immersed Boundary method would be to have all Immersed Boundary Points (IBPs) and Immersed Boundary Links (IBLs) exist on all processors. Each processor would update and compute values for local IBPs and IBLs, and then all processors would broadcast this information to all other processors (an “all-to-all” communication). The main

advantage of such a method is that it is relatively simple to understand. It turns out that this method may have other benefits as well; see section 7.5. However, the all-to-all communications could be costly.

To avoid all-to-all communications, we can distribute the IBPs and IBLs so that each processor tracks the IBPs and IBLs which are local to the region of the domain that they own. This will cause interfaces to be distributed across multiple processors. Since the steps of the IB method (calculating forces, spreading forces, interpolating velocities, and updating the IB) can require information from outside the owned region, we must specify how to handle the synchronization and communication issues and develop data structures to accommodate that work.

7.3.1 Basic Data Structure

The spreading and interpolation steps are done via delta functions, which have a compact stencil. Since the imposed CFL condition implies that IBPs cannot move more than one meshwidth per time step, the updating step also in effect has a stencil width of one. The compact stencils for these operations imply that access to IBPs and IBLs in nearby computational cells will be required. To facilitate these steps and minimize searching for relevant IBPs and IBLs which need to be passed, we want to store IBPs and IBLs according to their computational cell.

We want our data structure set up in such a way to work within the SAMRAI framework. SAMRAI has the notion of a “patch”, which is a rectangular region of the domain (consisting of computational cells) owned by a given processor. The Patch contains an array of PatchData objects, one PatchData for each variable (e.g. pressure, velocity, etc.) in the problem. The PatchData object stores the value of the relevant variable at each computational cell. Thus for cell-centered quantities such as pressure, the PatchData object (called a CellData in that case) would consist of an array of

double precision floating point values. To store the Immersed Boundary in an analogous fashion, we will have an IBData (which is a PatchData) structure and an IBCell structure. The IBCell (or IBC) structure is needed because a computational cell can contain many IBPs and IBLs. The basic data structure we have set up so far can be found in Table 1. Note that the word “list” appearing in that table does not necessarily mean linked list—it could be implemented as an array or an octree or any other number of things.

Table 1 Preliminary Immersed Boundary Data Structures

An IBData object contains	An IBCell contains
<ul style="list-style-type: none"> • Array of IBCells 	<ul style="list-style-type: none"> • List of IBPoints • List of IBLinks
An IBPoint is composed of	An IBLink is composed of
<ul style="list-style-type: none"> • Spatial coordinates (location) • Whether or not IBPoint is tethered • Force & Velocity vectors 	<ul style="list-style-type: none"> • Resting Length • Tension • Whether a periodic boundary is crossed

7.3.2 Data Requirements

Before we try to determine what other data we might need, it is useful to review the necessary computations and the data that we need in order to do them.

Calculating the IB forces is done according to equation (49). In order to be able to perform that calculation, we need the locations of both IBPs that any given IBL connects. So, to calculate the forces at all local IBPs, we need to obtain the location of each exterior IBP which is connected directly by an IBL to a local IBP. We also need the resting length and tension of the IBL.

Spreading forces from the IB to the grid is done according to equation (42). For a given cell-

centered gridpoint, we need to know two things: (1) the force at each IBP within the support of the delta function (centered at the gridpoint), and (2) the location of each of those IBPs. So to calculate the forces at all interior gridpoints of a patch, we need a copy of all IBPs within half the stencil width of the delta function of that patch.

Interpolating velocities from the grid to the IB is done according to equation (43). For a given IBP, we need to know the velocities at each gridpoint within the support of the delta function (centered at the IBP). So to calculate the velocities at all IBPs of a patch, we need a copy of all cell-centered velocities within half the stencil width of the delta function of the outermost IBPs.

As mentioned previously, the equation to update the position of the IB varies. But regardless of whether an implicit or explicit method is used (let alone the specific choice of implicit or explicit method), the basic formula is the same and it stipulates that the interface moves at the local fluid velocity. While handling this updating of the IB, we will need to know both what IBPs (and associated IBLs) enter our region from the exterior, and which IBPs (and IBLs) leave our region. Because of the CFL condition, no IBP will move more than one computational cell length. Therefore we will need one ghost cell width of IBPs and IBLs filled in order to update the interface.

7.3.3 Cascade of Issues

The data structures in Table 1 do not contain sufficient data for all computations. Many of the data fields that we need to add in order to enable a specific computation will cause a cascade effect of further requirements. The section outlines all these issues, the extra data fields we will need, and the assumptions that we will be placing on the data.

Preliminary Data Structure Questions Looking at Table 1, one of the first questions that arises is how to connect the IBLs and IBPs. In other words, given an IBL, how do we determine which IBPs it connects? Given an IBP, how do we determine what other IBPs it is connected to? We could (1) add something to each IBL to keep track of the IBPs that it connects and (2) add something to each IBP to keep track of all IBLs that connect to it, but doing both of these things results in a recursive data structure. A recursive structure would be difficult to maintain, especially since data will need to be able to “migrate” across processors as the IB moves with the fluid. So we must choose which approach to take. I believe both will work equally well, and I chose the former. This means that to each IBL I add two pointers to IBPs (which I will further abbreviate to “two PIBPs”). This choice means that there is not an efficient way to determine other IBPs to which a given IBP connects. Thus, when we compute forces at the IB, we loop through the IBLs and compute the force from that IBL and add it to both respective links instead of looping through the IBPs and computing all the forces at that given IBP.

The next question to address is what does the list of IBLs in the IBC contain? It is fairly obvious that the list of IBPs in the IBC should contain all IBPs located within that computational cell, but IBLs are not necessarily confined to a single cell. In the case that both the IBPs which an IBL connect are in separate IBCs, both IBCs will need information about the given IBL. This would make putting the IBL in just one of the two IBCs problematic because the two IBCs may belong on different processors. On the other hand, if both IBPs belong to the same IBC we do not want two IBLs in the IBC. Doing so would mean that we run the risk of double counting the forces from the IBL or that we have to carefully decide which IBP each IBL contributes force to. So the initial rule that we choose (this will be modified later as we explore the rest of the communication requirements) is that the list of IBLs in an IBC contains all IBLs for which at least one of the IBPs

to which the IBL attaches is within the IBC's list of IBPs. Another way of stating this rule is that if an IBP is contained in an IBC, then any IBL which connects to it should also be tracked in that IBC.

This rule about the list of IBLs in an IBC implies that IBLs will only be unique if both IBPs to which it attaches are contained in the same IBC. Since IBPs move during the course of a calculation, the two IBPs of an IBL may be in the same cell at certain times and not in the same cell at others. We need a way to be able to determine if two IBLs that are moving to the same IBC happen to be the same IBL so that we can remove one in this event. To enable this, we add a unique identifier to each IBL.

Calculating IB Forces Some IBLs that are connected to a local IBP will also be connected to an IBP on another processor. As the data structure currently stands, that would mean that one of the PIBPs of the IBL would be NULL. We need the location of the external IBP in order to calculate the IB forces, but how do we know which processor to get that information from? We cannot simply fill one ghost cell layer of IBCs with their respective IBPs because IBLs can span long distances. Even if IBLs did not span long distances though, we still would not know how to match up multiple IBPs in a ghost cell region with their associated IBLs. Without some basic information about the external IBPs to which our internal IBLs are connected, we cannot answer this question. Thus, we have to require that each PIBP of an IBL be non-NULL and at least point to an incomplete or dummy IBP.

Since we want to know which processor to receive the real IBP information from, we require these dummy IBPs to store the number of the processor that owns the real IBP. This will not be enough, however, since we might need several IBPs from a given processor and we need a way to

be able to distinguish them so we can associate each with their correct IBL. So, we add a unique identifier to each IBP (and have the dummies store this identifier as well).

Trying to receive the IBPs corresponding to dummy IBPs brings up another subtle issue. If one processor posts a receive call asking to obtain some information, then the processor with that information must post a corresponding send call. Therefore, each processor must be able to determine which local IBPs it needs to send and which processors it needs to send them to. This turns out to be simple due to the fact that IBLs connect exactly two IBPs—if one processor needs an IBP because one of its IBLs is connected to a dummy IBP, then that processor will also need to send the local IBP which the IBL is connected to (and the processor to which to send that local IBP can be found in the owner field of the dummy IBP).

An alternate way to handle the communication issues for calculating IB forces is to have dummy IBPs know their true location at the beginning of the timestep. This does mean extra work in the updating step to make sure that all dummies have the correct location. However, (1) there would be extra work to make sure the dummies had the correct owner field anyway (which work would be nearly identical to the work required for getting the location fields correct), (2) many of the dummy IBP locations will be known as a result of the update step (this is because the update step operates on IBPs in both the interior and the first ghost cell width), and (3) having the dummies store the correct location makes it so that we do not have to communicate before calculating IB forces. Because of all these reasons, we opt to go with the latter approach of storing locations in the dummy IBPs instead of the former approach of storing both the real owner and a unique identifier.

Spreading Forces and Interpolating Velocities The communication requirements for spreading of forces and interpolating of velocities are much simpler. The amount of data these require

was already mentioned in section 7.3.2—namely, a copy of all IBPs within half the stencil width of the delta function of any patch for the spreading of forces, and a copy of all cell-centered velocities within half the stencil width of the delta function of any patch for the interpolation of velocities. So, for spreading we will fill two ghost cells of IBCs and for interpolation we will fill two ghost cells of cell-centered velocities. Note that the ghost IBCs for spreading will only need IBP locations and forces; the rest of the IBP data fields can be ignored as can all IBLs. These requirements are enough to allow us to execute both of these steps, but some later optimizations will change these requirements slightly.

Updating the IB Location As mentioned in section 7.3.2, we need one ghost cell width of IBPs and IBLs filled in order to update the IB locations. This one ghost cell width of IBPs and IBLs has another hidden requirement. In order to maintain the consistency of the data structures (and specifically to fulfill the requirement that no PIBP of an IBL may be NULL), we need a way to find out both IBPs to which any IBL attaches. The only way to do this is to pass both IBPs of any given IBL along with the IBL. This will end up causing some heavy duplication of the IBPs being passed and the duplicate IBP info will eventually be discarded, but it is needed in order to match up IBLs with their respective IBPs on the receiving end.

Because we are passing two IBPs for every IBL, we need to be able to identify when IBPs are duplicates. Thus, we need a unique identifier for all IBPs. (When we discussed the calculation of IB forces, we presented a method that did not require these identifiers. However, these identifiers are now needed to connect IBLs to the appropriate IBP).

There is also another requirement that we need to satisfy when updating the interface in order to maintain the validity of our previous assumptions and preserve data structure consistency: we

need to ensure that all IBPs, including dummies, know their location once we are done updating. This will require more data than what the communication for the update step that we have specified so far will provide. A situation that illustrates this is in the case of IBLs that span long distances. Figure 3 shows an example of this where IBPs P_1 and P_2 are owned by CPUs C_1 and C_3 at the

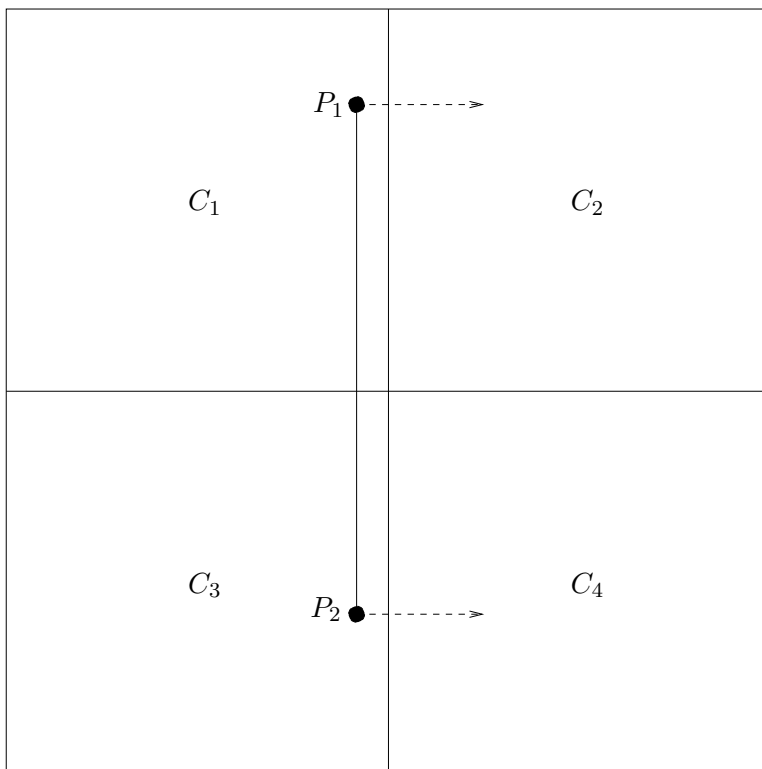


Figure 3: An example where LDC is needed—processors C_1 and C_3 own the distant IBPs P_1 and P_2 at the beginning of the timestep, while C_2 and C_4 own them at the end of the timestep.

beginning of the timestep and by CPUs C_2 and C_4 at the end of the timestep. In this example, in order to make sure that even dummy IBPs contain their correct location, both C_2 and C_4 will need to know the ending locations of both P_1 and P_2 —not just the IBP that they will own. The communication necessary to provide this information is what I call “long distance communication”

(LDC).

In order to demonstrate what needs to be done as part of LDC, we concentrate on C2 (which owns P1) in Figure 3. C2 will need to learn the location of P2. It will have no way of knowing that the new owner of the point is C4. It will be possible for it to know that C3 was the old owner (if C1 passes the right information with P1 for the update step), however C3 will not be able to know whether C2 or another processor is the one that needs the information. Therefore, we break LDC into two steps, LDC_1 and LDC_2 . For this specific example, LDC_1 will consist of C1 and C3 sending and receiving the new locations of P1 and P2 with each other (note that C1 and C3 will know these once they have updated the IBP locations as the first part of the update step). LDC_2 will consist of C1 and C3 sending the information that they receive from each other to C2 and C4, respectively.

LDC_1 will be easier if we add an owner field to all IBPs. It is not strictly necessary because the IBPs already have a location field and processors can figure out the processor that owns the region of the domain that includes a certain point. However, it will make things simpler by cutting down on the number of times that a processor has to look up the owner of a given location in the domain. So we add such a field to all IBPs.

IBLs that do not span long distances can also cause extra work in order to assure that corresponding dummy IBPs have the correct location. That extra work is a subset of what would be required for LDC, however, so I refer to all communication needed to ensure that IBPs have their correct location at the beginning of the next timestep as LDC (even though, technically speaking, “long distance” is not really accurate in all cases).

At the end of the updating step, there could be a lot of IBLs and IBPs in the ghost cells or the dummy list which are no longer needed. These correspond either to portions of the interface

in the first ghost cell layer that did not move into the interior, or to portions of the interface that started in the interior but moved outside the patch during the timestep. Since every IBL contains two PIBPs, determining whether an IBL can be deleted is fairly easy—it can be deleted if either of the two following conditions hold: (1) both IBPs that the IBL connects end up exterior to the patch at the end of the timestep, or (2) the IBL ends up in a ghost cell. Checking to see whether an IBP can be deleted is not as simple. We do not want a recursive data structure, so we cannot add data to the IBP that enables easy access to all IBLs that connect to it. We could add a routine that loops over the entire patch every few timesteps to see which IBPs still have IBLs connected to them, but I am not a fan of garbage collection. Instead, we add a simple counter to each IBP that keeps track of the number of IBLs from the local patch which connect to it and delete the IBP when the counter reaches 0.

Optimizations After all the irrelevant IBLs and IBPs have been deleted at the end of the update step, there could still be many IBPs in ghost cells. Moving all of these to the dummy list of IBPs could result in a very large list and cause slow searches. So, we leave them in the ghost cells and require that the dummy list of IBPs only contain IBPs that are outside the first ghost cell layer of a patch. This means that we will be maintaining IBCs in the first ghost cell layer as well. These IBCs will be slightly different since we delete all IBLs in ghost cells at the end of the timestep. This changes our requirement on the list of IBLs in an IBC slightly (our requirement was that the list of IBLs contain all IBLs which connect to any of the IBPs in the cell), so that the requirement only holds for interior IBCs.

These ghost cell IBPs also bring up another issue—when IBPs are passed just before the spreading of forces step, several of the passed IBPs will be the same as the ghost cell IBPs. However,

we have already added a unique identifier to each IBP so we will have enough data to handle this non-uniqueness problem.

We may be able to perform another optimization that saves a little bit of storage space, by addressing one more uniqueness question. Although we have addressed many uniqueness issues for IBLs and IBPs already, one still remains: should IBPs be unique-per-patch or unique-per-processor? Unique-per-patch is a simple consequence of the fact that IBPs cannot be in more than one location at a time. So we should definitely ensure that no patch has more than one copy of a given IBP (with a small exception that will be pointed out later). Unique-per-processor is a stronger requirement that could be used. Having IBPs be unique per processor would imply that if an IBP is located within one patch on a processor then it cannot appear in a ghost cell or the dummy list of another patch on that same processor. Unique-per-processor may save some space and time. However, the space and time saved would not be significant and it may be difficult to verify uniqueness of IBPs that appear in ghost cells and dummy lists and may also make the local link count of all IBPs harder to keep track of. Because of this, we stick with unique-per-patch.

We can also optimize our communication algorithm to remove one more communication step. Instead of passing one ghost cell of IBLs and IBPs before updating the interface and passing two ghost cells of cell-centered velocities before interpolating, we instead combine the two separate communication steps into one. To combine these steps, we have to extend the range of the velocities passed to three ghost cells so that we can still interpolate the velocities at all the ghost IBPs.

Although passing three ghost cells of cell-centered velocities along with one ghost cell of IBPs and IBLs works, we can optimize a little more. Two ghost cells of IBPs are already being passed before the spreading step, and they are not changing between the spreading and interpolation steps. Thus, we can pass the one ghost cell of IBPs and IBLs along with a second ghost cell of IBPs before

the spreading step instead of passing IBPs and IBLs before the interpolation step.

Passing the one ghost cell layer of IBPs and IBLs as part of the spreading step does raise a small issue. Spreading only requires that ghost cells being passed contain location and force (and a unique identifier to handle the non-uniqueness due to the ghost cell IBPs), because the IBPs are normally not needed after spreading is done. Since we want to keep the inner ghost cell layer of IBPs and we have IBLs in that inner layer which may have IBPs corresponding to the outer ghost cell layer, our operation will have to change a little. More than one strategy is possible, but we try to employ one that keeps spreading and updating as separate as possible. To achieve this, we delete all IBPs in the outer ghost cell layer once spreading is done (they would not normally be needed after spreading and are not part of the standard update step). This means that while we parse the IBPs and IBLs in the inner ghost layer, if any IBL is connected to an IBP that would normally be in the outer ghost cell layer, we instead store that IBP in the dummy list. This will cause some minor duplication since the IBP will appear in both the outer ghost cell layer and the dummy list, but once the spreading step is complete and the outer ghost cell layer of IBPs is deleted, IBPs will again be unique.

7.3.4 Final Data Structures and Assumptions

Section 7.3.3 presented a long cascade of issues that caused numerous changes to the data structures. Many assumptions were also made about these data structures in order to keep everything consistent. It can be hard to keep track of all these changes and assumptions, so we list the final data structures in Table 2 and all the assumptions we place on the data structures in Table 3.

Table 2 Immersed Boundary Data Structures

An IBData object contains	An IBCell contains
<ul style="list-style-type: none">• Array of IBCells• List of dummy IBPoints	<ul style="list-style-type: none">• List of IBPoints• List of IBLinks
An IBPoint is composed of	An IBLink is composed of
<ul style="list-style-type: none">• Unique identifier• Spatial coordinates (location)• Force & Velocity vectors• Whether or not IBPoint is tethered• Owner & Local link count	<ul style="list-style-type: none">• Unique identifier• Resting Length• Tension• Whether a periodic boundary is crossed• Pointers to two IBPoints

Table 3 Assumptions on Immersed Boundary Data Structures

- The dummy list of IBPs contains IBPs which are outside the first ghost cell width of computational cells (IBPs located in the first ghost cell width are stored in the actual ghost cell).
 - The list of IBLs in an IBC corresponding to an interior cell contains all IBLs for which at least one of the IBPs to which the IBL attaches is within the IBC’s list of IBPs.
 - At the beginning of the timestep, no IBLs should exist in any ghost cell.
 - At the beginning of a timestep, all IBPs (including dummies) contain correct location information.
 - The local link count of each IBP specifies the number of IBLs in the local patch which link to the given IBP
 - Neither PIBP of an IBL may be NULL (however, one of them can point to an IBP in the “dummy” list of the IBData or to an IBP in the first ghost cell width)
 - IBPs are unique per patch, IBLs may or may not be (depending entirely on whether the two IBPs it attaches to are within the same IBC or not).
-

7.4 Parallel Implementation

Many of the steps to implement the IB method in parallel have already been mentioned in section 7.3, but several steps and details have not yet been covered. For completeness and to serve as a quick reference, we outline the implementation of the four relevant steps of the IB method in order: calculation of forces on the immersed boundary, spreading forces from the immersed boundary to the grid, interpolating velocities from the grid to the immersed boundary, and updating immersed boundary points.

7.4.1 Calculating Forces

In this step, each processor must determine the forces at IBPs interior to the region(s) of the domain that it owns. The force at the IBPs is a function of the position of the IB only. Since we assume that all IBPs, including dummies, contain correct location information at the beginning of a timestep, this step will not require any communication and IB forces can be computed according to Algorithm 7. While executing that algorithm, we have to be careful not to double count forces since IBLs may not be unique. If IBLs were unique we could add the force from the IBL to both IBPs to which it connects. To handle the non-unique case as well, the force due to any given IBL is only added to IBPs which are located in the same IBC as the IBL.

Algorithm 7 Calculating Forces on the IB

1. Compute the forces from each IBL within the interior of the patch using equation (49).
 2. Add the force from each IBL to the relevant IBPs.
-

7.4.2 Spreading

In this step, each processor must determine the forces at each grid point in the region that it owns from the various forces at the IBPs. In order to accomplish this, each processor will need to obtain forces from IBPs in IBCs that neighbor its region of the computational domain. The steps to do this are outlined in Algorithm 8.

Algorithm 8 Spreading Forces from the IB to the grid

1. Communicate to fill two ghost cell widths of IBPs, and one ghost cell width of IBLs.
 2. Each processor spreads forces using equation (42) from each IBP on a patch (including the ghost cell region) to cell centered forces within the interior of the patch.
 3. Delete IBPs in the outer ghost cell layer.
-

As mentioned in section 7.3.3, step 1 of Algorithm 8 is more than we really need for the spreading step but is an optimization to reduce the number of times that we communicate. The IBLs will be used in updating the IB positions, which is discussed in section 7.4.4. Making sure that all the right data gets passed in this step and that it gets parsed in such a way that the data structures remain consistent can be somewhat tricky. We list the suggested message structure to use in order to pass the needed information in Table 4.

Parsing the portion of the data structure for the outer ghost cell layer is rather simple; it consists of merely filling the outer ghost cells with the relevant IBPs. Parsing the portion of the data structure for the inner ghost cell layer, however, will require making several consistency checks to ensure that all of the following conditions are met: (1) each IBP should be unique (ignoring any IBPs in the outer ghost cell layer, of course), (2) each IBL should be linked up to two IBPs (each of which should either be in the interior, in the first ghost cell width, or in the dummy list), and (3) each IBP should have its fields set appropriately (e.g. force, location, and link count). So, we

Table 4 Suggested Message Structure

```

<Message structure>
  Number of IBCs being shared in the outer ghost cell layer
  <Per Outer Ghost Cell IBC>
    Number of IBPs within this cell
    <Per IBP>
      Relevant IBP info (location, force)
    </Per IBP>
  </Per Outer Ghost Cell IBC>
  Number of IBCs being shared in the inner ghost cell layer
  <Per Inner Ghost Cell IBC>
    Number of IBLs within this cell
    <Per IBL>
      Relevant IBL info (id, rest length, tension)
      Relevant IBP1 info (id, location, force, tethered, owner)
      Relevant IBP2 info (id, location, force, tethered, owner)
    </Per IBL>
  </Per Inner Ghost Cell IBC>
</Message structure>

```

will parse the inner ghost cell layer portion of the data structure by looping through the IBCs and IBLs, filling the relevant ghost cells as we do so, and then do special checks as we work with each IBP of an IBL. The checks vary depending on whether the IBP information corresponds to an IBP in the interior of the receiver’s domain, an IBP in the first ghost cell layer, or an IBP in the dummy list. We can determine which of those cases is relevant to a given IBP by using its location field. Once we know its location, we apply the necessary checks as found in Algorithm 9.

Step 2 of Algorithm 8 requires determining the region over which the force at an IBP would be spread and intersecting this with the interior of the relevant patch. To determine this range, we first write the equation to obtain the index of a cell, c_x , containing a given point x . In 1D this equation is

$$c_x = \left\lfloor \frac{x - x_{lo}}{dx} \right\rfloor \quad (50)$$

Algorithm 9 Parsing the Inner Ghost Cell Layer Message Structure

- For IBPs located in the interior of the receiver’s domain
 1. use the IBP location to help find the real IBP
 2. connect related IBL to the real IBP (do not increment the link count for the real IBP)
 - For IBPs located outside both the interior and inner ghost cell layer of the receiver’s domain
 1. if the IBP info does not correspond to an existing dummy IBP, create a dummy IBP (note that only the identifier and location are needed when creating this dummy IBP; the local link count should be set to 0); otherwise locate the existing dummy IBP
 2. connect related IBL to the dummy IBP
 3. increment the dummy IBP’s link count
 - For IBPs located in the inner ghost cell layer of the receiver’s domain
 1. if the IBP info corresponds to an existing IBP then
 - if the IBP is in the same IBC as the relevant IBL, then copy the passed force to the IBP
 - otherwise
 - create an IBP in the relevant IBC (note that identifier, location, force, tethered and owner are all needed when creating this IBP; the local link count should be set to 0)
 2. connect related IBL to the IBP
 3. if the IBP is in the same IBC as the related IBL then increment the IBP’s link count
-

where x_{lo} is the coordinate of the left side of the patch, dx is the spacing between gridpoints, and $[\cdot]$ denotes rounding down to the nearest integer. (Note that in higher dimensions, this equation is merely repeated for each dimension). If w is half the width of the spreading stencil, then letting $adj = w - dx/2$ makes the formula for the leftmost index over which to spread become

$$s_L = \left\lfloor \frac{x - x_{lo}}{dx} - adj \right\rfloor \quad (51)$$

and the formula for the rightmost index over which to spread become

$$s_R = \left\lfloor \frac{x - x_{lo}}{dx} + \text{adj} \right\rfloor \quad (52)$$

7.4.3 Interpolation

In this step, each processor must determine the velocity at IBPs from the relevant grid velocities. In order to accomplish this, each processor will need to obtain velocities at grid points that are adjacent to its region(s) of the computational domain. The steps to do this are outlined in Algorithm 10.

Algorithm 10 Interpolating Velocities from the grid to the IBPs

1. Copy three ghost cell widths of cell-centered velocities (u)
 2. For all IBPs in the interior of the owned region or within one ghost cell width of that region, interpolate u to these IBPs using equation (43) (to get u_{IB})
-

Step 1 again communicates more information than necessary and step 2 computes the velocity at more IBPs than necessary to reduce the number of times that communication is needed as mentioned in section 7.3.3.

As with the algorithm for spreading, step 2 of Algorithm 10 will require determining the cell indices from which to interpolate. This range turns out to be identical to the range over which to spread since the interpolation and spreading operators have the same stencil width. If bilinear interpolation was used instead (such as Lee and Leveque do in [22]), then a stencil width of 1 would be used instead to determine the value of the variable adj in formulas (51) and (52).

7.4.4 Updating

In this step, each processor must update IBP locations according to the fluid velocity, move IBPs and IBLs to the appropriate new cells, and make sure that all the data structures remain consistent. Due to the sharing of IBPs and IBLs in step 1 of Algorithm 8 and the sharing of grid velocities in step 1 of Algorithm 10, we should have enough information to compute the new locations of each IBP within the interior plus one ghost cell width of each region without further communication. However, we need to specify how and when to execute LDC during the update algorithm in order to maintain the consistency of the data structures. The basic idea of the steps to perform LDC is given in Algorithm 11. Since LDC is only needed when the two IBPs that an IBL connect are on separate processors, Algorithm 11 refers to the two CPUs that own IBPs P1 and P2 as C1 and C3, respectively. See Figure 3 in section 7.3.3 for an example situation.

Algorithm 11 Basic Idea behind LDC

1. C1 & C3 compute the new locations of IBPs P1 & P2
 2. Using the processor mapping, C1 & C3 determine which processors P1 & P2 will be owned by
 3. C1 & C3 exchange new point locations of P1 & P2 and new processor owners with each other (LDC₁)
 4. if C1 & C3 are not the new owners, then they must pass new point position & owner information on to the other processors (which also implies that the new processors must post corresponding receive calls). (LDC₂)
-

When to execute LDC is dependent on where the two IBPs for a given IBL are relative to a patch at both the beginning and end of the timestep. A diagram showing various cases can be seen in Figure 4. In that figure, each case has two IBPs connected by an IBL. The IBPs are labeled 1 and 2. The new locations for each IBP are also shown, and are labeled 3 and 4, respectively.

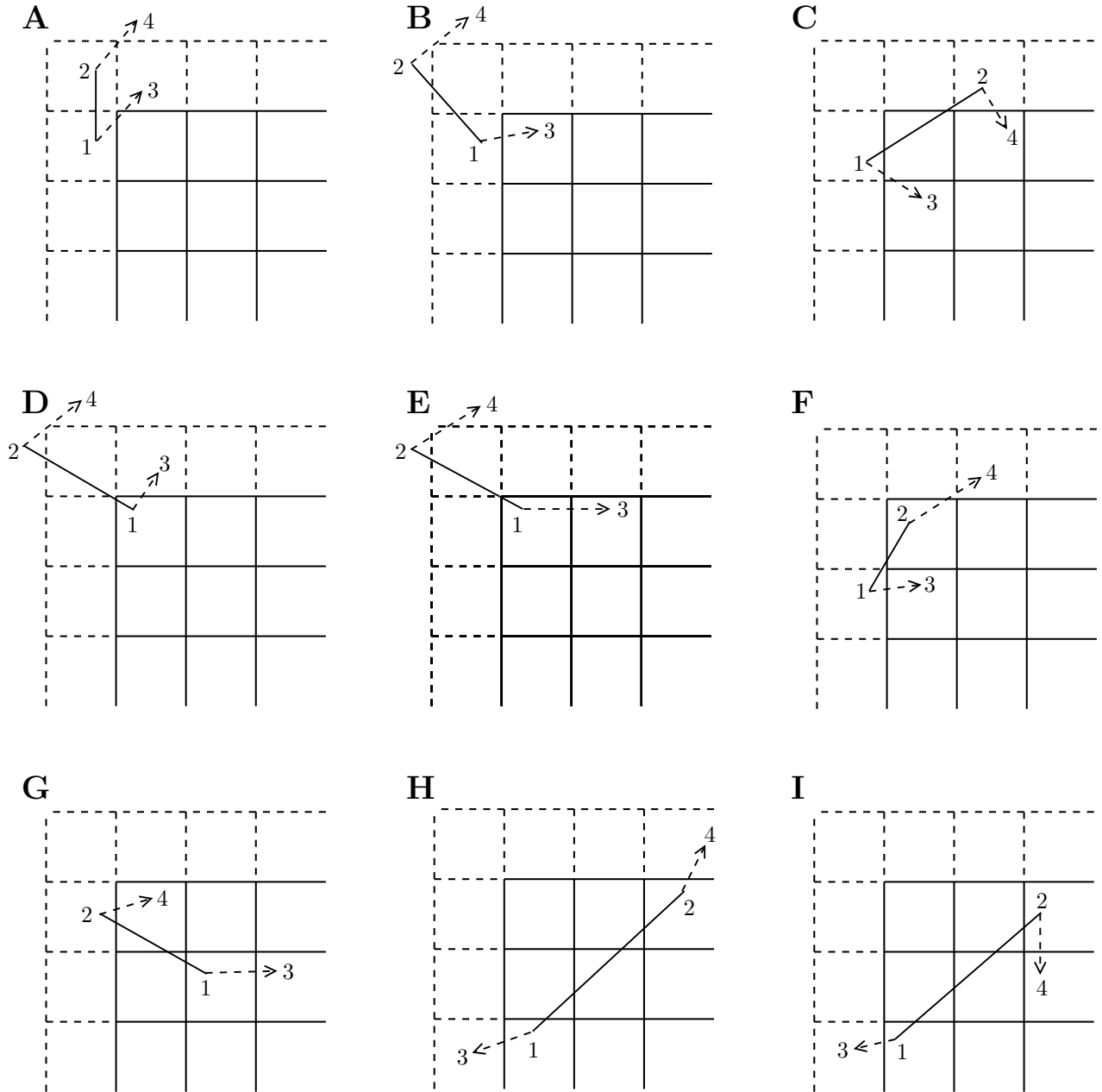


Figure 4: Various cases (labeled A-I) to consider when setting up LDC. In each case, the endpoints of the IBL at the beginning of the timestep are labeled as 1 and 2, while at the end of the timestep they are labeled as 3 and 4, respectively.

The details of when and what to communicate for each of the cases in Figure 4 are given in Table 5 and a summary is given in Table 6. In Table 5, me is used as shorthand for the processor who owns the relevant patch and O_x is used as shorthand for the processor who owns or will own the IBP when it is in position x . (For example, in case E, $O_1 = O_3 = me$)

Table 5 LDC Details

- A. LDC may be involved, but only for other processors
 - B. if me does not move both IBPs, then me needs to receive pt 4 as part of LDC₂. (if me does move both IBPs then O_1 should be able to find out that I've moved both and not attempt to send me pt 4)
 - C. no LDC necessary (O_1 and O_2 , if they don't both update both points, will need to send & receive info with each other, but can instantly discard the info they receive since they will know that $O_3 = O_4$.)
 - D.
 - 1. if O_2 does not update pt 1, then me needs to send him pt 3 (LDC₁)
 - 2. if me does not update pt 2 and $O_3 \neq O_2$ then me needs to receive pt 4 from O_2 (LDC₁)
 - 3. if O_3 does not update pt 2 and $O_4 \neq O_3$, then me needs to send pt 4 to O_3 . (LDC₂)
 - E. Steps D1 & D2
 - F. if O_4 does not update pt 1, then send him pt 3 (LDC₂)
 - G. no LDC involved
 - H. for each of O_3, O_4 that do not update the opposite point, me must send them that info (LDC₂)
 - I.
 - 1. if O_3 does not update pt 2, then send him pt 4 (LDC₂).
 - 2. if O_4 does not update pt 1, then send him pt 3 (LDC₂).
 - J. no LDC involved
-

We can now address the actual job of updating the IB data structures. To accomplish this, there are several things we need to do: perform any LDC necessary to get the new IBP position and ownership information, update the location and owner field of all the IBPs, move all IBPs and IBLs to the appropriate IBCs, delete IBLs which end up outside the patch (i.e. IBLs for which

Table 6 Different cases for LDC

IBPs in patch at start	IBPs in patch at end	Case label(s)	Possible LDC
0	0	A	none
0	1	B	LDC ₂
0	2	C	none
1	0	D	LDC ₁ & LDC ₂
1	1	E,F	E-LDC ₁ ; F-LDC ₂
1	2	G	none
2	0	H	LDC ₂
2	1	I	LDC ₂
2	2	J	none

neither linked IBP is in the patch interior), and delete IBPs in the dummy list whose link count has become 0. A method to handle all these jobs is listed in Algorithm 12.

7.5 Alternate Methods

The method outlined in the previous two sections to implement the IB method in parallel involves a lot of detailed bookkeeping. One way to avoid most of the bookkeeping would be to use the simple approach mentioned in section 7.3 of having all IBPs and IBLs be global. There are a number of trade-offs of using such a method. Some of these are that having the IBPs and IBLs be global would

- remove many details and should be easier to implement
- involve a more expensive all-to-all broadcast message
- only require two messages to be sent instead of four. (Sharing u before interpolating and an all-to-all broadcast of updated IBPs after interpolating; in contrast to sharing IBPs and IBLs before spreading, sharing u before interpolating, LDC₁, and LDC₂.)

Algorithm 12 Updating the IB

1. First, the location variable of all IBPs which started either in the interior or first ghost cell are updated according to their velocity (however, they are not moved to the appropriate cell during this step because the starting cell of each IBP is needed to determine what LDC is necessary).
 2. Move IBLs (from either the interior or first ghost cell) to the correct cell(s). Extra requirements:
 - (a) Determine and setup any LDC that needs to occur due to this IBL.
 - (b) If both IBPs for the IBL end up outside the patch interior, delete the IBL and decrement the link counter for any IBP that is in the same cell as the IBL or is in the dummy list.
 - (c) Only add the IBL to new cell(s) if the IBL will be unique.
 - (d) Remove the IBL from the old cell if neither IBP ends up in that cell or if the old cell is a ghost cell.
 3. Initiate any LDC₁ communication that needs to occur.
 4. Move IBPs from either the interior or first ghost cell to the correct cell. (The correct cell means the dummy list for any IBP that ends up outside of both the interior and first layer of ghost cells). Also update the ownership member of each IBP.
 5. Delete IBPs in the dummy list whose link count is 0.
 6. Verify that all communication from LDC₁ has completed. Execute any LDC₂ communication that needs to occur.
-

- allow the use of an Immersed Interface (II) method or a hybrid IB/II method (the 2D implementation of those methods require calculating cubic splines for the boundaries, and cubic splines require knowledge of all control points on the boundary)
- require a separate “PatchLevel” in SAMRAI since all processors would “own” all IB data objects for all cells.

We would like to use an II method (or at least a hybrid IB/II method) in the future, but we may not want to have IBPs and IBLs be global. Thus we might need to use a hybrid local/global approach with IBPs and IBLs shared “globally per interface” among processors. That is, if any given processor owns one IBP that is part of an interface, then it also “owns” all other IBPs on that interface. Doing so would

- allow the use of the II method or a hybrid IB/II method (again, this is related to the calculation of cubic splines, at least for 2D)
- still require a broadcast message, but this “all-to-all” message would be restricted to a subset of processors
- still require IBPs to be shared before spreading if interfaces were within a distance of two grid cells (since the delta function has a stencil width of four)
- still require LDC for links between interfaces (i.e. for cohesion)

8 Future Work

Besides debugging my current code and parallelizing the IB method already outlined, there are many additional issues that need to be addressed to produce a three dimensional simulation of platelet

aggregation and blood coagulation. These include, among other things, investigating Immersed Interface methods, modeling forces between connected interfaces, modeling the effects of red blood cells, adding a long cascade of chemical reactions that correspond to coagulation, and modeling the coupling of surface and solution chemicals.

8.1 Immersed Interface Method

As mentioned previously, I will implement an Immersed Interface solver or a hybrid Immersed Boundary/Immersed Interface solver. The II method was inspired by the IB method and was introduced to obtain solutions with second order accuracy up to and including the interface (the IB method has second order accuracy away from the interface, but only first order near the interface). The basic idea of the II method is to replace all the delta function interactions with jump conditions derived from the PDEs. The II method was first outlined for elliptic equations by Leveque and Li in [24], then extended to steady Stokes flow in [25], and was finally extended to the full Navier-Stokes equations by Li and Lai in [26].

By requiring the numerical method to produce a solution that satisfies the derived jump conditions, the II method is able to obtain sharp resolution at interfaces. In contrast, the IB method smears the values of the velocity and pressure across the interface when they are not smooth across it. This smearing not only results in less accurate solutions, it can also cause the failure to maintain certain steady states, such as the case of a pressurized circular membrane with no fluid movement (the IB method only approximately maintains this steady state and exhibits slow fluid leakage, whereas the II method maintains this discrete steady state exactly). In addition to these accuracy advantages, the II method also facilitates implicit solves by reducing the size of the system to solve; this results from the use of a cubic spline to represent the interface allowing for fewer control points.

The II method is not without its disadvantages, however. It is much more difficult to implement than the IB method. The brunt of the extra work comes in modifying the stencils for the PDEs and the right hand sides near the interface with terms from local Taylor series. This extra work could still be parallelized in the same manner as the IB method, with the possible exception of the boundary representation. The boundary representation issue arises from the fact that the II method requires the ability to evaluate the force and derivatives of the force anywhere along the interface. The way this is usually done is to run a cubic spline through the points on the interface. One drawback with this is that constructing splines requires knowledge of all control points for the spline and thus any processor which has one control point on an interface must have knowledge of all other control points on that interface as well. This “global-per-interface” requirement along with possible solutions to address it were discussed in section 7.5. Another drawback of the cubic splines is that the extension to 3D is not trivial. Thus, while the theoretical extension to 3D may be straightforward, very few implementations have appeared in the literature. The only ones that I am aware of use level set methods to represent the boundary, which precludes the use of elastic forces due to stretching since the level set boundary representation does not provide enough information to compute those kinds of forces.

Recently, Lee and Leveque introduced a hybrid II/IB method (see [22]) where the normal forces are treated with the II method and the tangential forces are treated with the IB method. This has the advantage that it is much easier to implement than the full II method and only requires a little more work than the IB method in a serial program. It does suffer from the same parallel “global per interface” issue for control points and the extension to 3D issue that affect the II method, but the simpler implementation does look intriguing especially since it shares some of the same advantages. One of these advantages is the use of fewer control points (and hence the

smaller implicit solves) that is facilitated by the use of cubic splines. Another is the maintaining of certain steady states such as a pressurized circular membrane with no fluid motion. This method also appears to achieve second order accuracy, at least for the problems Lee and Leveque tested. However, their test problems all involved cases where the normal forces on an interface are much larger than the tangential forces. Since we expect the tangential forces to be much larger for our problem than the problems they tested, we want to determine whether this hybrid method will still exhibit second order accuracy for a problem where the tangential forces are of the same magnitude as the normal forces.

8.2 BDF Methods

Another issue to explore is the type of implicit method to use in the II/IB solver. As stated in section 7.2, the implicit method used in the II papers appears to be the most promising. They replace the update formula

$$\mathbf{X}^{n+1} = \mathbf{X}^n + \Delta t \mathcal{U}^{n+1}(\mathbf{X}^n) \quad (53)$$

(which is a more precise way of writing the equation $\mathbf{X}^{n+1} = \mathbf{X}^n + \Delta t \mathbf{U}^{n+1}$ from Algorithm 6) with

$$\mathbf{X}^{n+1} = \mathbf{X}^n + \frac{1}{2} \Delta t \left(\mathcal{U}^n(\mathbf{X}^n) + \mathcal{U}^{n+1}(\mathbf{X}^{n+1}) \right). \quad (54)$$

They then solve this equation for \mathbf{X}^{n+1} via a Quasi-Newton (QN) method which searches for a zero

of the function

$$g(\mathbf{X}^*) = \mathbf{X}^* - \mathbf{X}^n - \frac{1}{2}\Delta t \left(\mathcal{U}^n(\mathbf{X}^n) + \mathcal{U}^{n+1}(\mathbf{X}^*) \right). \quad (55)$$

Equation (54) is second order in time, but we believe its stability properties may not be optimal. To see why, apply that update rule to a simple one-dimensional test problem, $x' = \mathcal{U}(x) = -\lambda x$ (The link between this test problem and the real problem is that the eigenmode of the real problem will have the same stability as the test problem with $-\lambda$ equal to the corresponding eigenvalue):

$$x^{n+1} = x^n + \frac{1}{2}\Delta t(\mathcal{U}^n(x^n) + \mathcal{U}^{n+1}(x^{n+1})) \quad (56)$$

$$x^{n+1} = x^n + \frac{1}{2}\Delta t(-\lambda x^n - \lambda x^{n+1}) \quad (57)$$

$$\left(1 + \frac{\lambda\Delta t}{2}\right)x^{n+1} = \left(1 - \frac{\lambda\Delta t}{2}\right)x^n \quad (58)$$

$$x^{n+1} = \frac{2 - \lambda\Delta t}{2 + \lambda\Delta t}x^n \quad (59)$$

The error for this test problem satisfies equation (59) as well, so it is clear that the method is stable for all $-\lambda \leq 0$. However, for very large λ , the fraction in equation (59) approaches -1, implying that the error will be oscillatory and slowly decaying. To balance large values of λ , restrictively small timesteps may be needed. Since the problems that the IB and II methods are applied to are inherently stiff, we expect that λ could be quite large. Indeed, papers on the II method have reported taking tiny timesteps on certain problems that were very stiff in early timesteps and then relaxing the timestep when the problem became less stiff later on. We suspect that using a BDF (backward differentiation formula) update method might handle these problems better since they were designed for stiff problems.

8.3 Forces Between Connecting Interfaces

In contrast to most problems to which the IB and II methods are applied, our problem will have connections between separate interfaces as platelets aggregate. Since the IB method already treats the interface as a discrete system of springs, it seems logical to also treat connections between boundaries as springs. However, there are a number of questions regarding the details of how to implement such a method. How many connections should be made? Should extra control points be added to each interface for the endpoints of each spring? If so, how should they be added?

Not adding control points for each connection would mean that the endpoints of springs are not updated by the IB/II solver and thus must be handled separately. But adding control points would increase the size of the system in the implicit solve. This would not only cause a single iteration of the solve to take longer but may also increase the condition number of the system, requiring more iterations [25].

Adding several springs per connection would amplify the problems in the previous paragraph. However, only adding a single spring would mean that the force from the spring would not depend on the angle between the interface and the spring. Since the connections between aggregating platelets consist of proteins binding to specific sites distributed over patches of the respective platelets' membranes, these connections resist rolling of one platelet with respect to the other. So if a single connection is used in our model, it should also involve some kind of force that resists rolling (i.e. it should not be a simple spring).

8.4 Modeling of Red Blood Cells

Red blood cells make up around 40-50% of blood by volume. The prevalence of these cells combined with their larger size relative to platelets results in platelets frequently being displaced due to

collisions with these cells. Trying to track individual red blood cells would be computationally infeasible, so we would like to model their overall effect. Experimentalists have observed that the motion of platelets due to interactions with red blood cells appears to resemble diffusion. We cannot just add a diffusive term since we are not using a continuum model for platelets, so we instead want to add some kind of random motion to the platelets. In doing so, we should make sure that the motion we add is somewhat uniform across the platelet instead of having random motions at each control point because red blood cells bump into the platelet cell, not into control points from the cell. This means that some kind of syncing between processors is required so that all processors moving a given interface do so in the same manner.

The method by which random motion is added is also important. The easiest method would be to add some kind of random motion after each timestep where each platelet is displaced slightly from where the fluid motion would have sent it. However, this violates the no-slip condition between the fluid and the interface. This is especially problematic when chemical reactions are taken into account, because steep gradients in chemicals could occur near the interface and moving the platelet randomly could cause chemicals near the boundary to “disappear” inside the platelet. An alternate method to add random motion would be to compute additional random forces which could be added to the forces in the IB/II method. This would allow the no-slip condition to be satisfied, but leaves the question of how to specify these forces.

8.5 Chemical Interactions

The cascade of enzymatic reactions in blood coagulation also needs to be included. Fogelson and Kuharsky present a model of a test problem related to coagulation in [18], and they extend it to a comprehensive treatment of the relevant reactions in [21]. Both papers assume spatial homogeneity

in order to simplify the problem to a set of ODEs. The latter paper includes over fifty ODEs for the full system which correspond to many different chemicals and complexes, both in solution and bound to platelet membranes. Accounting for blood coagulation requires adding spatial dependence (i.e. advection and diffusion terms) to those equations and then incorporating them into the model.

Since many of the reactions involve feed-forward and feed-back loops and the enzymes involved can accelerate reactions by several orders of magnitude, steep gradients of the chemical concentrations will occur. In order to prevent smearing of these gradients, I will use Leveque's high resolution advection method with slope limiters [23] for the advection of chemicals.

Many of the chemical reactions in coagulation occur on the surface of platelet and vessel walls. In order to handle these surface reactions, a method to define and update surface meshes on which to solve ODEs for these reactions is needed. One option for determining these meshes would be to use the control points of the surfaces. This has a drawback, however. We expect many of the interesting chemical reactions to occur on or near the boundary and since these reactions can be very fast, large spatial variation in concentrations can occur. To accurately account for those variations, we would need many control points on the surface. However, that would increase the size of our implicit system solve, slowing the convergence considerably. So control points as mesh points is not feasible, which means that another method is required.

A second challenge of the surface reactions is that a mathematical boundary condition must be derived to model the behavior of the interaction between membrane-bound and solution phase chemicals. This boundary condition must somehow account for diffusion so that these chemicals do not diffuse through the membrane. Getting the boundary condition correct is crucial to the overall accuracy.

8.6 Adaptive Mesh Refinement

Most of the phenomena in this problem are highly localized. The fluid-structure interactions between the platelets and the fluid only occur within a few grid cells of the immersed boundaries, and the most interesting fluid dynamics occurs near these interfaces. Also, important chemical reactions occur on cell surfaces; through the coupling between membrane-bound and solution phase chemicals this means that chemical reactions will also be localized near the interfaces. To handle these localized behaviors and capture the full dynamics without excessively refining the grid throughout the rest of the domain, I plan to utilize adaptive mesh refinement (AMR). The basic methods for implementing AMR for structured meshes are outlined in [4, 5, 6, 7, 8]. SAMRAI handles the implementation of the basic ideas from these papers, but leaves many details about interactions between coarse and fine grids to the application developer. To handle those details, I will draw on Minion's work combining AMR with projection methods [29], and Roma et al.'s work detailing a two-dimensional IB method utilizing AMR [34].

8.7 Conclusion

The issues involved in creating a 3D simulation of platelet aggregation and blood coagulation are numerous. They involve problems from mathematical modeling, numerical analysis, and computational implementation. This reflects upon the inherent complexity of the system that is being modeled. Such a simulation, will elucidate fundamental biological mechanisms and improve biomedical therapies and devices.

A Optimization of the Navier-Stokes Solver

Upon close examination of the Navier-Stokes solver (Algorithm 1), it becomes apparent that there are a couple of ways to optimize it. These include storing quantities that would otherwise need to be computed multiple times, and sharing storage locations for various variables and temporaries.

A.1 Quantities Computed Multiple Times

In the Navier-Stokes solver algorithm there were a number of computed quantities that appeared multiple times. For all of these quantities, plus those which serve as the right hand side (RHS) to a linear system that needs to be solved, we would like to introduce temporary variables. Doing so results in Table 7.

Table 7 Temporary variables

$$\begin{array}{ll}
 \mathbf{T}_{v0} = \mu \Delta^h \mathbf{u}^n & \mathbf{T}_{v4} = \mathbf{T}_{v2} - \mathbf{T}_{v3} + \mathbf{f}^{n+\frac{1}{2},k} \\
 \mathbf{T}_{v1} = -[\mathbf{u} \cdot \nabla \mathbf{u}]^n + \mathbf{T}_{v0} + \mathbf{f}^n & T_{s1} = \nabla^h \cdot \mathbf{u}^{*,k} \\
 T_{s0} = \nabla^h \cdot \mathbf{T}_{v1} & T_{s2} = \frac{1}{\Delta t} T_{s1} + \Delta^h p^{n+\frac{1}{2},k} \\
 \mathbf{T}_{v2} = \frac{1}{\Delta t} \mathbf{u}^n + \frac{1}{2} \mathbf{T}_{v0} - [\mathbf{u} \cdot \nabla \mathbf{u}]^{n+\frac{1}{2}} & \mathbf{T}_{v5} = \nabla^h p^{n+\frac{1}{2},k+1} \\
 \mathbf{T}_{v3} = \nabla^h p^{n+\frac{1}{2},k} &
 \end{array}$$

Some comments about the temporaries in Table 7:

- \mathbf{T}_{v5} really wouldn't be listed normally (it's not used later and isn't the RHS of a solve), but in later optimizations it becomes necessary, and I list it here simply for completeness.
- $p^{n+\frac{1}{2}}$ uses T_{s1} and \mathbf{u}^{n+1} uses both \mathbf{T}_{v3} and \mathbf{T}_{v5} .
- The Helmholtz solver will need another scalar temporary distinct from any of the above temporaries in order to define residual equations.

A.2 Variable Storage

Every temporary has to be defined at each gridpoint and since there are many gridpoints, each of these takes up a fairly large amount of memory. Thus, we would like to reuse temporaries where possible. Using $a \rightarrow b$ as shorthand for saying that a can be written to the same storage location as b , we can save temporary storage locations as follows:

$$\mathbf{T}_{v5} \rightarrow \mathbf{T}_{v2} \rightarrow \mathbf{T}_{v0}$$

$$\mathbf{T}_{v3} \rightarrow \mathbf{T}_{v1}$$

$$T_{s2} \rightarrow T_{s0}.$$

Using the same storage locations repeatedly like this, we are able to reduce our memory usage from six vector temporaries down to three (\mathbf{T}_{v4} isn't shown above) and from three scalar temporaries down to two (T_{s1} isn't shown above).

Note that of these temporaries, only \mathbf{T}_{v1} needs ghost cells (with a width of 1). Since \mathbf{T}_{v3} and \mathbf{T}_{v1} will be sharing storage, this will effectively mean that \mathbf{T}_{v3} simply has some unused storage.

We also do not need to store four different values of \mathbf{u} , four values of p , and two values of \mathbf{f} either. It is fairly easy to reduce the requirement to three different values of \mathbf{u} , two values of p and one value of \mathbf{f} . However, extreme scrutiny with the temporaries shows that with no extra temporaries, we can get away with only one value of each.

This will mean that, just as for \mathbf{T}_{v1} and \mathbf{T}_{v3} some of the \mathbf{u} and p variables will have ghost cells even though they aren't necessary.

A.3 Optimized Numerical Method

The algorithm with all the above optimizations is shown in Algorithm 13.

Algorithm 13 Optimized NS Solver Algorithm with Explanations

<p>[\mathbf{u}, p, and \mathbf{f} refer to their values at time n]</p> $\mathbf{T}_{v0} = \mu \Delta^h \mathbf{u}$ $\mathbf{T}_{v1} = -[\mathbf{u} \cdot \nabla \mathbf{u}] + \mathbf{T}_{v0} + \mathbf{f}$ <p>[\mathbf{f} now refers to $\mathbf{f}^{n+\frac{1}{2},k}$]</p> $T_{s0} = \nabla^h \cdot \mathbf{T}_{v1}$ <p>Solve $\Delta^h p = T_{s0}$</p> $\mathbf{T}_{v2} = \mathbf{u}$ $\mathbf{u} = \mathbf{u} + \frac{\Delta t}{2} \mathbf{T}_{v1}$ <p>[\mathbf{u} is part of $u^{n+\frac{1}{2}}$]</p> $\mathbf{T}_{v1} = \nabla^h p$ $\mathbf{u} = \mathbf{u} - \frac{\Delta t}{2} \mathbf{T}_{v1}$ <p>[\mathbf{u} now refers to $u^{n+\frac{1}{2}}$]</p> <p>[p now refers to $p^{n+\frac{1}{2},0}$]</p> $\mathbf{T}_{v0} = \frac{1}{2} \mathbf{T}_{v0} + \frac{1}{\Delta t} \mathbf{T}_{v2} - [\mathbf{u} \cdot \nabla \mathbf{u}]$ <p>[\mathbf{u} now refers to $\mathbf{u}^{*,k}$]</p>	<p>For $k=0, \dots$</p> <p>[\mathbf{T}_{v1} already equals $\nabla^h p^{n+\frac{1}{2},k}$]</p> $\mathbf{T}_{v2} = \mathbf{T}_{v0} - \mathbf{T}_{v1} + \mathbf{f}$ <p>Solve $\frac{1}{\Delta t} \mathbf{u} - \frac{\mu}{2} \Delta^h \mathbf{u} = \mathbf{T}_{v2}$</p> $T_{s1} = \nabla^h \cdot \mathbf{u}$ $T_{s0} = \frac{1}{\Delta t} T_{s1} + \Delta^h p$ <p>[p now refers to $p^{n+\frac{1}{2},k+1}$]</p> <p>Solve $\Delta^h p = T_{s0}$</p> <p>Exit here to terminate iteration.</p> $\mathbf{T}_{v1} = \nabla^h p$ <p>End of for loop</p> $\mathbf{T}_{v0} = \nabla^h p$ $p = p - \frac{\mu}{2} T_{s1}$ <p>[p now refers to $p^{n+\frac{1}{2}}$]</p> $\mathbf{u} = \mathbf{u} + \Delta t (\mathbf{T}_{v1} - \mathbf{T}_{v0})$ <p>[\mathbf{u} now refers to \mathbf{u}^{n+1}]</p>
---	--

B Cell-Centered Multigrid and Robin BCs

B.1 Motivation

The most common boundary conditions used with multigrid seem to be Dirichlet boundary conditions. The grid is usually vertex-centered so that the outermost (“boundary”) gridpoints coincide with the boundary of the domain. This makes the boundary conditions easy to implement, as

the outermost gridpoints can simply be set so that the boundary conditions are satisfied exactly. However, with Neumann (or even worse, Robin) boundary conditions or different cell centerings, an additional complication occurs. We then have approximations for both the interior and boundary equations:

$$\begin{aligned}Lu &= f \text{ (on } \Omega) \\ \mathcal{B}u &= g \text{ (on } \partial\Omega),\end{aligned}$$

where \mathcal{B} represents a linear combination of u and its derivatives.

The problem that arises is that by alternately setting the boundary conditions (using ghost cells) and applying the smoother which we normally use for the equations on the interior, we may not actually be smoothing the full set of equations. If we update the ghost cells according to the discrete boundary conditions, then smooth, and then restrict the residual to the next coarser level, then our boundary conditions are not met and we'll be solving for the wrong boundary conditions on the coarser level. If we first smooth and then enforce the boundary conditions before restricting, then there will be "kinks" (non-smoothness) in the errors which then cause problems in transferring the defect or residual to coarser grids (since the error is no longer smooth). This degrades the convergence of multigrid considerably (in some examples we ran, we had convergence factors around 0.9 instead of around 0.1-0.2).

It turns out that if the boundary conditions are used to modify the stencil of the operator L at the boundary and the smoother is defined by using the modified stencil, then the end result is that the full equations are correctly smoothed and rapid multigrid convergence is again obtained. Maintaining a lot of different stencils is somewhat of a pain, though, so we would like to know

how to set the boundary conditions in such a way that using our normal stencils is mathematically equivalent to running multigrid with the modified stencil.

B.2 System of Equations

The system of equations we will be considering on a cell-centered grid are

$$Au - B\Delta u = r \text{ (on } \Omega)$$

$$au + bu' = c \text{ (on } \partial\Omega)$$

where Ω is the domain of interest; the values A , B , r , a , b , and c can all be spatially dependent; and u' is the derivative in the appropriate coordinate direction (e.g. $\frac{du}{dx}$, or the outward normal on the right side and the inward normal on the left side)

Since we will be dealing with several different boundaries, we adopt the convention that u_{ij} refers to the interior point closest to the boundary. Also, we will be using u_b as shorthand for the value of u at the boundary. So, for the left side of the domain, this would mean that $u_b = u_{i-\frac{1}{2},j}$, that the ghost cell is $u_{i-1,j}$, and the other point of relevance is $u_{i+1,j}$. For the right side of the domain, this would mean that $u_b = u_{i+\frac{1}{2},j}$, that the ghost cell is $u_{i+1,j}$, and that the other point of relevance is $u_{i-1,j}$. The bottom and top boundaries are similar (with j and i subscripts more or less changing places).

B.3 Approximating the Boundary Conditions

We start by working with the left side of the domain and use Taylor series expanding around the boundary point:

$$\begin{aligned} u_{i-1} &= u_b - \frac{1}{2}hu'_b + \frac{1}{4}h^2u''_b - \frac{1}{8}h^3u'''_b + \dots \\ u_i &= u_b + \frac{1}{2}hu'_b + \frac{1}{4}h^2u''_b + \frac{1}{8}h^3u'''_b + \dots \\ u_{i+1} &= u_b + \frac{3}{2}hu'_b + \frac{9}{4}h^2u''_b + \frac{27}{8}h^3u'''_b + \dots \end{aligned}$$

Using these series we take:

$$\begin{aligned} u_b &= \frac{3}{8}u_{i-1} + \frac{6}{8}u_i - \frac{1}{8}u_{i+1} + O(h^3) \\ u'_b &= \frac{1}{h}u_i - \frac{1}{h}u_{i-1} + O(h^2) \end{aligned}$$

Plugging this into the equation for the boundary conditions at the left boundary, we get

$$\begin{aligned} c &= au_b + bu'_b \\ c &= a \left(\frac{3}{8}u_{i-1} + \frac{6}{8}u_i - \frac{1}{8}u_{i+1} \right) + b \left(\frac{1}{h}u_i - \frac{1}{h}u_{i-1} \right) \\ u_{i-1} &= \frac{-\frac{6}{8}au_i + \frac{1}{8}au_{i+1} - \frac{b}{h}u_i + c}{\frac{3}{8}a - \frac{b}{h}} \end{aligned}$$

Likewise, for right boundary gives us

$$\begin{aligned}
 c &= au_b + bu'_b \\
 c &= a \left(\frac{3}{8}u_{i+1} + \frac{6}{8}u_i - \frac{1}{8}u_{i-1} \right) + b \left(\frac{1}{h}u_{i+1} - \frac{1}{h}u_i \right) \\
 u_{i+1} &= \frac{-\frac{6}{8}au_i + \frac{1}{8}au_{i-1} + \frac{b}{h}u_i + c}{\frac{3}{8}a + \frac{b}{h}}
 \end{aligned}$$

Similarly, we can figure the approximations for the bottom and top boundaries, which come out as follows:

$$\begin{aligned}
 u_{j-1} &= \frac{-\frac{6}{8}au_j + \frac{1}{8}au_{j+1} - \frac{b}{h}u_j + c}{\frac{3}{8}a - \frac{b}{h}} \\
 u_{j+1} &= \frac{-\frac{6}{8}au_j + \frac{1}{8}au_{j-1} + \frac{b}{h}u_j + c}{\frac{3}{8}a + \frac{b}{h}}
 \end{aligned}$$

B.4 Filling Ghost Cells for Computing Residuals

When computing the residual, we set the ghost cells according to the equations just derived (which is the same thing we would do if dealing with modified stencils). We do not need ghost cells corresponding to corner points when computing the residual, so the above formulas suffice.

B.5 Filling Ghost Cells for Interpolation

When interpolating the estimated error from a coarser grid to a finer one in multigrid, we use a linear interpolant (that's what we would in the case of a modified stencil, and so we do it here too).

Thus we need a different way to set the ghost cells just before performing interpolation.

We start with the ghost cells at the left hand side. There we take

$$u_b \approx \frac{1}{2}u_{i-1} + \frac{1}{2}u_i + O(h^2)$$

$$u'_b \approx \frac{1}{h}u_i - \frac{1}{h}u_{i-1} + O(h^2)$$

Using these approximations on the left hand side, we have

$$c = au_b + bu'_b$$

$$c = a \left(\frac{1}{2}u_i + \frac{1}{2}u_{i-1} \right) + b \left(\frac{1}{h}u_i - \frac{1}{h}u_{i-1} \right)$$

$$u_{i-1} = \frac{-\frac{1}{2}au_i - \frac{b}{h}u_i + c}{\frac{1}{2}a - \frac{b}{h}}$$

Likewise for the right hand side we get

$$c = au_b + bu'_b$$

$$c = a \left(\frac{1}{2}u_{i+1} + \frac{1}{2}u_i \right) + b \left(\frac{1}{h}u_{i+1} - \frac{1}{h}u_i \right)$$

$$u_{i+1} = \frac{-\frac{1}{2}au_i + \frac{b}{h}u_i + c}{\frac{1}{2}a + \frac{b}{h}}$$

Similarly we can figure the bottom and top formulas, which turn out to be:

$$u_{j-1} = \frac{-\frac{1}{2}au_j - \frac{b}{h}u_j + c}{\frac{1}{2}a - \frac{b}{h}}$$

$$u_{j+1} = \frac{-\frac{1}{2}au_j + \frac{b}{h}u_j + c}{\frac{1}{2}a + \frac{b}{h}}$$

These formulas will gives us values for all the sides, but they do not specify corner values. We

need the corner values in order to use the same 4-point bilinear interpolation stencil everywhere. We could derive formulas for these corner points using Neumann or Dirichlet boundary conditions, but trying to do so for Robin boundary conditions does not make much sense. It might be able to be done, but I don't see how to do so. Note, however, that if we didn't require using the corner point, then we could simply use a 3-point stencil to do linear interpolation and ignore the corner. Since we'd like to use the same 4-point stencil everywhere, we simply set the corner value to be whatever is required so that using the 4-point stencil involving a corner point gives us the same result as if we had just used the 3-point stencil. This gives us the following equations for the corner points:

$$\text{Bottom Left: } u_{i-1,j-1} = u_{i-1,j} + u_{i,j-1} - u_{ij}$$

$$\text{Bottom Right: } u_{i+1,j-1} = u_{i+1,j} + u_{i,j-1} - u_{ij}$$

$$\text{Top Left: } u_{i-1,j+1} = u_{i-1,j} + u_{i,j+1} - u_{ij}$$

$$\text{Top Right: } u_{i+1,j+1} = u_{i+1,j} + u_{i,j+1} - u_{ij}$$

B.6 Filling Ghost Cells for Smoothing

This case is much more difficult. The smoothing operator was defined from the modified stencil, but we want to use the smoothing operator defined from the standard stencil. To proceed, we must make heavy use of the fact that we're using a Red-Black Gauss-Seidel smoother and the fact that we are trying to solve the Helmholtz Equation. We again start with the left hand side. Recall that

the extrapolation formula for the ghost cell of u at the left of the domain is

$$u_{i-1} = \frac{-\frac{6}{8}au_i + \frac{1}{8}au_{i+1} - \frac{b}{h}u_i + c}{\frac{3}{8}a - \frac{b}{h}}$$

We substitute this into the standard Helmholtz stencil,

$$\begin{bmatrix} 0 & -B & 0 \\ -B & h^2A + 4B & -B \\ 0 & -B & 0 \end{bmatrix} u = h^2r$$

which gives us the modified Helmholtz stencil:

$$\begin{bmatrix} 0 & -B & 0 \\ 0 & h^2A + 4B + \frac{BY}{X} & -B - \frac{BZ}{X} \\ 0 & -B & 0 \end{bmatrix} u = h^2r + \frac{B}{X}c$$

where $X = (\frac{3}{8}a - \frac{b}{h})$, $Y = (\frac{6}{8}a + \frac{b}{h})$, and $Z = \frac{1}{8}a$.

Note that any (pointwise) Gauss-Seidel smoother is defined by solving for the center point of the stencil in terms of all other quantities. Since we want the end result of using the smoother defined by either of the above stencils to match, we will need to pick the ghost cell for the standard stencil (which is the value corresponding to the leftmost -B in the first stencil) appropriately. This will be easier to determine if we can first make the center coefficients of the two stencils match. To do so, we multiply the first equation through by $\frac{h^2A+4B+\frac{BY}{X}}{h^2A+4B} = 1 + \frac{BY}{h^2AX+4BX}$. To make things

easier to read, we set $\delta = \frac{BY}{h^2AX+4BX}$, so that we're multiplying through by $1 + \delta$. We get:

$$\begin{bmatrix} 0 & -B - B\delta & 0 \\ -B - B\delta & h^2A + 4B + \frac{BY}{X} & -B - B\delta \\ 0 & -B - B\delta & 0 \end{bmatrix} u = h^2r(1 + \delta)$$

In order for this equation to match the one using the modified stencil,

$$\begin{bmatrix} 0 & -B & 0 \\ 0 & h^2A + 4B + \frac{BY}{X} & -B - \frac{BZ}{X} \\ 0 & -B & 0 \end{bmatrix} u = h^2r + \frac{B}{X}c$$

we must set the ghost cell as follows:

$$\begin{aligned} u_{i-1,j} &= \frac{B\delta}{-B - B\delta}u_{i,j+1} + \frac{B\delta}{-B - B\delta}u_{i,j-1} + \frac{B\delta - \frac{BZ}{X}}{-B - B\delta}u_{i+1,j} \\ &+ \frac{\delta}{-B - B\delta}h^2r - \frac{\frac{B}{X}}{-B - B\delta}c \\ &= \frac{\delta}{-1 - \delta}u_{i,j+1} + \frac{\delta}{-1 - \delta}u_{i,j-1} + \frac{\delta - \frac{Z}{X}}{-1 - \delta}u_{i+1,j} \\ &+ \frac{\frac{\delta}{B}}{-1 - \delta}h^2r - \frac{\frac{1}{X}}{-1 - \delta}c \end{aligned}$$

It turns out that the other sides will be the same, with a change in the values of the X 's and Y 's. So we turn our concentration to the bottom left corner. First, the two relevant extrapolation

formulas:

$$u_{i-1} = \frac{-\frac{6}{8}a_L u_i + \frac{1}{8}a_L u_{i+1} - \frac{1}{h}b_L u_i + c_L}{\frac{3}{8}a_L - \frac{1}{h}b_L}$$

$$u_{j-1} = \frac{-\frac{6}{8}a_B u_j + \frac{1}{8}a_B u_{j+1} - \frac{1}{h}b_B u_j + c_B}{\frac{3}{8}a_B - \frac{1}{h}b_B}$$

Substituting these values for u_{i-1} and u_{j-1} into the standard Helmholtz stencil,

$$\begin{bmatrix} 0 & -B & 0 \\ -B & h^2 A + 4B & -B \\ 0 & -B & 0 \end{bmatrix} u = h^2 r$$

gives the modified Helmholtz stencil:

$$\begin{bmatrix} 0 & -B - \frac{BZ_B}{X_B} & 0 \\ 0 & h^2 A + 4B + \frac{BY_L}{X_L} + \frac{BY_B}{X_B} & -B - \frac{BZ_L}{X_L} \\ 0 & 0 & 0 \end{bmatrix} = h^2 r + \frac{Bc_L}{X_L} + \frac{Bc_B}{X_B}$$

where the values of $X_{L|B}$, $Y_{L|B}$, and $Z_{L|B}$ are given by

	L	B
X	$\frac{3}{8}a_L - \frac{1}{h}b_L$	$\frac{3}{8}a_B - \frac{1}{h}b_B$
Y	$\frac{6}{8}a_L + \frac{1}{h}b_L$	$\frac{6}{8}a_B + \frac{1}{h}b_B$
Z	$\frac{1}{8}a_L$	$\frac{1}{8}a_B$

We again want the center coefficients of the stencils to match, so we multiply the first equation

through by

$$\frac{h^2 A + 4B + \frac{BY_L}{X_L} + \frac{BY_B}{X_B}}{h^2 A + 4B} = 1 + \frac{BY_L}{h^2 AX_L + 4BX_L} + \frac{BY_B}{h^2 AX_B + 4BX_B}$$

First, we set

$$\delta_L = \frac{BY_L}{h^2 AX_L + 4BX_L}$$

$$\delta_B = \frac{BY_B}{h^2 AX_B + 4BX_B}$$

so that we're multiplying through by $1 + \delta_L + \delta_B$. The resulting equation is:

$$\begin{bmatrix} 0 & -B - B\delta_L - B\delta_B & 0 \\ -B - B\delta_L - B\delta_B & h^2 A + 4B + \frac{BY_L}{X_L} + \frac{BY_B}{X_B} & -B - B\delta_L - B\delta_B \\ 0 & -B - B\delta_L - B\delta_B & 0 \end{bmatrix} = h^2 r (1 + \delta_L + \delta_B)$$

In order for this last equation to match the equation with the modified stencil, we set the two ghost cells as follows:

$$u_{i-1,j} = \frac{\left(B\delta_L + B\delta_B - \frac{BZ_L}{X_L} \right) u_{i+1,j} + \delta_L h^2 r - \frac{B}{X_L} c_L}{-B - B\delta_L - B\delta_B}$$

$$u_{i,j-1} = \frac{\left(B\delta_L + B\delta_B - \frac{BZ_B}{X_B} \right) u_{i,j+1} + \delta_B h^2 r - \frac{B}{X_B} c_B}{-B - B\delta_L - B\delta_B}$$

or, simplified slightly (dividing through by B):

$$u_{i-1,j} = \frac{\left(\delta_L + \delta_B - \frac{Z_L}{X_L}\right) u_{i+1,j} + \frac{\delta_L}{B} h^2 r - \frac{1}{X_L} c_L}{-1 - \delta_L - \delta_B}$$

$$u_{i,j-1} = \frac{\left(\delta_L + \delta_B - \frac{Z_B}{X_B}\right) u_{i,j+1} + \frac{\delta_B}{B} h^2 r - \frac{1}{X_B} c_B}{-1 - \delta_L - \delta_B}$$

B.6.1 Tables of Values

We want to write down the formulas for the other sides and corners. Before we do so, it will be useful to introduce some notation. First, let us recall that we are trying to solve

$$Au - B\Delta u = r$$

$$au + bu' = c$$

We introduce a couple tables of values below, with the subscripts L , R , B , and T for the left, right, bottom, and top sides, respectively. The values of $X_{L|R|B|T}$, $Y_{L|R|B|T}$, and $Z_{L|R|B|T}$ are given by

	L	R	B	T
X	$\frac{3}{8}a_L - \frac{1}{h}b_L$	$\frac{3}{8}a_R + \frac{1}{h}b_R$	$\frac{3}{8}a_B - \frac{1}{h}b_B$	$\frac{3}{8}a_T + \frac{1}{h}b_T$
Y	$\frac{6}{8}a_L + \frac{1}{h}b_L$	$\frac{6}{8}a_R - \frac{1}{h}b_R$	$\frac{6}{8}a_B + \frac{1}{h}b_B$	$\frac{6}{8}a_T - \frac{1}{h}b_T$
Z	$\frac{1}{8}a_L$	$\frac{1}{8}a_R$	$\frac{1}{8}a_B$	$\frac{1}{8}a_T$

The values of $\delta_{L|R|B|T}$ are given by the following:

$$\begin{aligned}\delta_L &= \frac{BY_L}{h^2 AX_L + 4BX_L} \\ \delta_R &= \frac{BY_R}{h^2 AX_R + 4BX_R} \\ \delta_B &= \frac{BY_B}{h^2 AX_B + 4BX_B} \\ \delta_T &= \frac{BY_T}{h^2 AX_T + 4BX_T}\end{aligned}$$

Also, we want to add some more notation for subscripts in order to further compress the resulting equations. For values not near the corners of the domain (i.e. on the sides only), we use the subscript \perp to denote the appropriate subscript for the relevant boundary (i.e. to replace either L , R , B , or T). We use $\perp G$ as a subscript for the relevant ghost cell for that side, $\perp F$ as a subscript for the far cell (two cell lengths away from the ghost cell), and $P1$ and $P2$ as subscripts for the two points parallel to the boundary in the stencil which are not in a line with $\perp G$ and $\perp F$.

For values near the corners of the domain, we use the subscripts I and J to denote the appropriate subscript for the relevant boundaries in the I and J directions (e.g. I can replace L or R). The IG and IF subscripts denote the ghost cell and the far cell in the I direction, respectively. JG and JF are used similarly for the J direction.

B.6.2 Summarized Formulas

Using all the above notation, we can summarize the equations for filling ghost cells before smoothing as follows:

Sides:

$$u_{\perp G} = \frac{\delta_{\perp}}{-1 - \delta_{\perp}} u_{P1} + \frac{\delta_{\perp}}{-1 - \delta_{\perp}} u_{P2} + \frac{\delta_{\perp} - \frac{Z_{\perp}}{X_{\perp}}}{-1 - \delta_{\perp}} u_{\perp F}$$

$$+ \frac{\frac{\delta_{\perp}}{B}}{-1 - \delta_{\perp}} h^2 r - \frac{\frac{1}{X_{\perp}}}{-1 - \delta_{\perp}} c_{\perp}$$

Corners:

$$u_{IG} = \frac{\left(\delta_I + \delta_J - \frac{Z_I}{X_I}\right) u_{IF} + \frac{\delta_I}{B} h^2 r - \frac{1}{X_I} c_I}{-1 - \delta_I - \delta_J}$$

$$u_{JG} = \frac{\left(\delta_I + \delta_J - \frac{Z_J}{X_J}\right) u_{JF} + \frac{\delta_J}{B} h^2 r - \frac{1}{X_J} c_J}{-1 - \delta_I - \delta_J}$$

The compact notation can be somewhat difficult to read and compare to previous work, so here

I write out the formulas without the special subscripts.

Sides:

$$u_{i-1,j} = \frac{\delta_L}{-1 - \delta_L} u_{i,j+1} + \frac{\delta_L}{-1 - \delta_L} u_{i,j-1} + \frac{\delta_L - \frac{Z_L}{X_L}}{-1 - \delta_L} u_{i+1,j}$$

$$+ \frac{\frac{\delta_L}{B}}{-1 - \delta_L} h^2 r - \frac{\frac{1}{X_L}}{-1 - \delta_L} c_L$$

$$u_{i+1,j} = \frac{\delta_R}{-1 - \delta_R} u_{i,j+1} + \frac{\delta_R}{-1 - \delta_R} u_{i,j-1} + \frac{\delta_R - \frac{Z_R}{X_R}}{-1 - \delta_R} u_{i-1,j}$$

$$+ \frac{\frac{\delta_R}{B}}{-1 - \delta_R} h^2 r - \frac{\frac{1}{X_R}}{-1 - \delta_R} c_R$$

$$u_{i,j-1} = \frac{\delta_B}{-1 - \delta_B} u_{i+1,j} + \frac{\delta_B}{-1 - \delta_B} u_{i-1,j} + \frac{\delta_B - \frac{Z_B}{X_B}}{-1 - \delta_B} u_{i,j+1}$$

$$+ \frac{\frac{\delta_B}{B}}{-1 - \delta_B} h^2 r - \frac{\frac{1}{X_B}}{-1 - \delta_B} c_B$$

$$u_{i,j+1} = \frac{\delta_T}{-1 - \delta_T} u_{i+1,j} + \frac{\delta_T}{-1 - \delta_T} u_{i-1,j} + \frac{\delta_T - \frac{Z_T}{X_T}}{-1 - \delta_T} u_{i,j-1} \\ + \frac{\frac{\delta_T}{B}}{-1 - \delta_T} h^2 r - \frac{\frac{1}{X_T}}{-1 - \delta_T} c_T$$

Bottom Left corner

$$u_{i-1,j} = \frac{\left(\delta_L + \delta_B - \frac{Z_L}{X_L} \right) u_{i+1,j} + \frac{\delta_L}{B} h^2 r - \frac{1}{X_L} c_L}{-1 - \delta_L - \delta_B}$$

$$u_{i,j-1} = \frac{\left(\delta_L + \delta_B - \frac{Z_B}{X_B} \right) u_{i,j+1} + \frac{\delta_B}{B} h^2 r - \frac{1}{X_B} c_B}{-1 - \delta_L - \delta_B}$$

Bottom Right corner

$$u_{i+1,j} = \frac{\left(\delta_R + \delta_B - \frac{Z_R}{X_R} \right) u_{i-1,j} + \frac{\delta_R}{B} h^2 r - \frac{1}{X_R} c_R}{-1 - \delta_R - \delta_B}$$

$$u_{i,j-1} = \frac{\left(\delta_R + \delta_B - \frac{Z_B}{X_B} \right) u_{i,j+1} + \frac{\delta_B}{B} h^2 r - \frac{1}{X_B} c_B}{-1 - \delta_R - \delta_B}$$

Top Left corner

$$u_{i-1,j} = \frac{\left(\delta_L + \delta_T - \frac{Z_L}{X_L} \right) u_{i+1,j} + \frac{\delta_L}{B} h^2 r - \frac{1}{X_L} c_L}{-1 - \delta_L - \delta_T}$$

$$u_{i,j+1} = \frac{\left(\delta_L + \delta_T - \frac{Z_T}{X_T}\right) u_{i,j-1} + \frac{\delta_T}{B} h^2 r - \frac{1}{X_T} c_T}{-1 - \delta_L - \delta_T}$$

Top Right corner

$$u_{i+1,j} = \frac{\left(\delta_R + \delta_T - \frac{Z_R}{X_R}\right) u_{i-1,j} + \frac{\delta_R}{B} h^2 r - \frac{1}{X_R} c_R}{-1 - \delta_R - \delta_T}$$

$$u_{i,j+1} = \frac{\left(\delta_R + \delta_T - \frac{Z_T}{X_T}\right) u_{i,j-1} + \frac{\delta_T}{B} h^2 r - \frac{1}{X_T} c_T}{-1 - \delta_R - \delta_T}$$

B.6.3 Why dividing by zero does not occur

As you may have noticed, the above formulas involve many fractions. A question that quickly arises is whether any of them cause a divide by zero to occur. If you look through all of them, the following are the quantities that appear in a denominator somewhere:

$$B$$

$$h^2 A + 4B$$

$$X_{\dagger}$$

$$-1 - \delta_{\dagger}$$

$$-1 - \delta_{\dagger} - \delta_{\ddagger}$$

where \dagger and \ddagger are one of L , R , B , or T .

B is never zero—if it were, this would be a pretty trivial Helmholtz equation to solve.

$h^2A + 4B$ won't be zero either. For our problems, either both A and B are positive, or else B is negative when A is zero. . (Also note that if A and B are not of the same sign, then the diagonal dominance of the discrete Helmholtz equation comes into question and multigrid may not be an appropriate solver)

To determine the range of values of X_{\dagger} , we have to look more closely at our specific problem. X_{\dagger} is defined by the values of the constants a and b in the boundary conditions of the problem,

$$au + bu' = c$$

For us, this boundary condition will come from the coupling of the diffusion of a chemical species with the binding and unbinding reaction rates of the same chemical. The form of the equation will be

$$-D \frac{\partial c}{\partial n} = k_{\text{on}}c(t - c^b) + k_{\text{off}}c^b$$

where c is the concentration of some chemical species in solution, c^b is the concentration of the same species bound to a binding site at the boundary, and t is the total binding-site density at the given boundary. Rearranging this equation to the form for a and b gives

$$k_{\text{on}}c(s - c^b) + D \frac{\partial c}{\partial n} = -k_{\text{off}}c^b$$

which, for example, at the left side is:

$$k_{\text{on}}c(s - c^b) - D\frac{\partial c}{\partial x} = -k_{\text{off}}c^b$$

So that

$$a_L = k_{\text{on}}(s - c^b) \geq 0$$

$$b_L = -D < 0$$

Since $a_L \geq 0$ and $b_L < 0$, we find that $X_L = \frac{3}{8}a_L - \frac{1}{h}b_L > 0$. For the right side, the equation for X_R has a plus instead of a minus between the a 's and b 's. However, the direction of the outward normal also changes, which results in a change of sign for b . Thus we find that $X_R > 0$ also. The bottom and top are similar to the left and right.

Now, to take care of the two cases $-1 - \delta_{\dagger}$ and $-1 - \delta_{\dagger} - \delta_{\ddagger}$, we have to see how small the different values of δ can become. The values of δ are defined by

$$\delta = \frac{B}{h^2A + 4B} \frac{Y}{X}$$

We seek a lower bound on the value of δ :

$$\begin{aligned} \delta &= \frac{B}{h^2A + 4B} \frac{Y}{X} \\ &\geq \frac{B}{h^2A + 4B} \cdot \min\left(\frac{Y}{X}\right) \end{aligned}$$

which is true since the fraction on the left is known to be positive. We can bound the ratio of Y

and X , which depend on the values of a and b . We look at the left hand side (all the other sides are similar):

$$\frac{Y_L}{X_L} = \frac{\frac{6}{8}a_L + \frac{b_L}{h}}{\frac{3}{8}a_L - \frac{b_L}{h}}$$

We obtain the bound by looking at some simple limits. For h large,

$$\frac{Y_L}{X_L} = \frac{\frac{6}{8}a_L + \frac{b_L}{h}}{\frac{3}{8}a_L - \frac{b_L}{h}} \approx \frac{\frac{6}{8}a_L}{\frac{3}{8}a_L} = 2$$

whereas for h small,

$$\frac{Y_L}{X_L} = \frac{\frac{6}{8}a_L + \frac{b_L}{h}}{\frac{3}{8}a_L - \frac{b_L}{h}} \approx \frac{\frac{b_L}{h}}{-\frac{b_L}{h}} = -1$$

So $\frac{Y_L}{X_L}$ will always lie in the interval $[-1, 2]$. It turns out the ratios of the Y 's and X 's for the other sides lie in the same interval. Thus, continuing with trying to bound δ , we find that

$$\begin{aligned} \delta &\geq \frac{B}{h^2A + 4B} \cdot \min\left(\frac{Y}{X}\right) \\ &= -\frac{B}{h^2A + 4B} \\ &\geq -\frac{B}{4B} \\ &= -\frac{1}{4} \end{aligned}$$

Since $\delta \geq -\frac{1}{4}$, we see that $-1 - \delta_{\dagger} \leq -\frac{3}{4}$ and $-1 - \delta_{\dagger} - \delta_{\ddagger} \leq -\frac{1}{2}$. Thus all the cases are covered and no divide by zero will occur with any of the possible denominators listed.

There is one other case to worry about that hasn't been covered. We haven't explicitly written

out the smoothing step of the Red-Black Gauss-Seidel smoother. The smoother is defined by taking the equations with the Helmholtz stencils that we had and solving for the values corresponding to the center stencil point. The only division that will occur in doing this (other than cases that are covered above) is in dividing through by the coefficient of the center value of the stencil. So we want to make sure that this value is not zero. For the stencil near the sides, this coefficient is

$$h^2A + 4B + \frac{BY_{\dagger}}{X_{\dagger}}$$

whereas for the stencil near the corners, this coefficient was

$$h^2A + 4B + \frac{BY_{\dagger}}{X_{\dagger}} + \frac{BY_{\ddagger}}{X_{\ddagger}}$$

We noted before that the smallest that the ratio $\frac{Y_{\dagger}}{X_{\dagger}}$ could be is -1. Therefore, the smallest that the center coefficient for stencils near the side can be is

$$h^2A + 3B$$

whereas the smallest that the center coefficient for stencils near the corners can be is

$$h^2A + 2B$$

We end with one final note that these center coefficients in the stencils also maintain diagonal dominance. For diagonal dominance, we need the center point of the stencil to be at least as large as the sum of the absolute values of the other entries in the stencil (note that all the other entries

in the stencil happen to be negative). For a stencil near this side, this requirement is:

$$h^2 A + 4B + \frac{BY}{X} \geq B + B + B + B \frac{Z}{X}$$

$$h^2 A + B + B \frac{Y-Z}{X} \geq 0$$

which is certainly true so long as $\frac{Y-Z}{X} \geq -1$. Looking at the definitions for X , Y , and Z at the left hand side (the other sides are similar):

$$\frac{Y_L - Z_L}{X_L} = \frac{\frac{6}{8}a_L + \frac{1}{h}b_L - \frac{1}{8}a_L}{\frac{3}{8}a_L - \frac{1}{h}b_L}$$

$$= \frac{\frac{5}{8}a_L + \frac{1}{h}b_L}{\frac{3}{8}a_L - \frac{1}{h}b_L}$$

Again, $X_L > 0$, and by looking at the limits for h large and for h small (just as we did before for the ratio of Y to X) show that this ratio is always within the interval $[-1, \frac{5}{3}]$. The ratios of the other $\frac{Y-Z}{X}$'s are also within this same interval. Thus diagonal dominance for the stencils near the side hold.

For a stencil near the corner, in order to have diagonal dominance, we need that:

$$h^2 A + 4B + \frac{BY_{\dagger}}{X_{\dagger}} + \frac{BY_{\ddagger}}{X_{\ddagger}} \geq B + B \frac{Z_{\dagger}}{X_{\dagger}} + B + B \frac{Z_{\ddagger}}{X_{\ddagger}}$$

$$h^2 A + 2B + B \frac{Y_{\dagger} - Z_{\dagger}}{X_{\dagger}} + B \frac{Y_{\ddagger} - Z_{\ddagger}}{X_{\ddagger}} \geq 0$$

which is certainly true since $\frac{Y-Z}{X} \geq -1$.

References

- [1] G. AGRESAR, J. J. LINDERMAN, G. TRYGGVASON, AND K. G. POWELL, *An adaptive, cartesian, front-tracking method for the motion, deformation and adhesion of circulating cells*, Journal of Computational Physics, 143 (1998), pp. 346–380.
- [2] *A Balancing Act*. Website, 2003. <http://www.haemoscope.com/balance/>.
- [3] J. B. BELL, P. COLELLA, AND H. M. GLAZ, *A second-order projection method for the incompressible Navier-Stokes equations*, Journal of Computational Physics, 85 (1989), pp. 257–283.
- [4] M. J. BERGER, *Adaptive mesh refinement for hyperbolic partial differential equations*, Journal of Computational Physics, 53 (1984), pp. 484–512.
- [5] ———, *Stability of interfaces with mesh refinement*, Mathematics of Computation, 45 (1985), pp. 301–318.
- [6] ———, *Data structures for adaptive grid generation*, SIAM Journal of Scientific Statistical Computing, 7 (1986), pp. 904–916.
- [7] ———, *Local adaptive mesh refinement for shock hydrodynamics*, Journal of Computational Physics, 82 (1989), pp. 64–84.
- [8] M. J. BERGER AND I. RIGOUTSOS, *An algorithm for point clustering and grid generation*, IEEE Transactions on Systems, Man, and Cybernetics, 21 (1991), pp. 1278–1286.
- [9] R. P. BEYER, *A computational model of the cochlea using the immersed boundary method*, Journal of Computational Physics, 98 (1992), pp. 145–162.

- [10] E. G. BOVILL, *Coagulation and Hemostasis*. Website, 2003. http://cats.med.uvm.edu/cats_teachingmod/pathology/path301/hemostasis/home/hemoindex.html.
- [11] W. L. BRIGGS, *A Multigrid Tutorial*, SIAM, Philadelphia, Pennsylvania, 1987.
- [12] D. L. BROWN, R. CORTEZ, AND M. MINION, *Accurate projection methods for the incompressible Navier-Stokes equations*, *Journal of Computational Physics*, 168 (2001), pp. 464–499.
- [13] S. L. DIAMOND, *Reaction complexity of flowing human blood*, *Biophysical Journal*, 80 (2001), pp. 1031–1032.
- [14] R. DILLON, L. FAUCI, A. FOGELSON, AND D. GAVER, *Modeling biofilm processes using the immersed boundary method*, *Journal of Computational Physics*, 129 (1996), pp. 85–108.
- [15] L. J. FAUCI AND A. L. FOGELSON, *Truncated newton methods and the modeling of complex immersed elastic structures*, *Communications on Pure and Applied Mathematics*, 66 (1993), pp. 787–818.
- [16] L. J. FAUCI AND C. S. PESKIN, *A computational model of aquatic animal locomotion*, *Journal of Computational Physics*, 77 (1988), pp. 85–108.
- [17] A. L. FOGELSON, *A mathematical model and numerical method for studying platelet adhesion and aggregation during blood clotting*, *Journal of Computational Physics*, 1 (1984), pp. 111–134.
- [18] A. L. FOGELSON AND A. L. KUHARSKY, *Membrane binding-site density can modulate activation thresholds in enzyme systems*, *Journal of Theoretical Biology*, 193 (1998), pp. 1–18.
- [19] G. H. GOLUB AND C. F. V. LOAN, *Matrix Computations*, John Hopkins University Press, Baltimore, Maryland, 1996.

- [20] D. S. HOUSTON, P. ROBINSON, AND J. M. GERRARD, *Inhibition of intravascular platelet aggregation by endothelium-derived relaxing factor: Reversal by red blood cells*, *Blood*, 76 (1990), pp. 953–958.
- [21] A. L. KUHARSKY AND A. L. FOGELSON, *Surface-mediated control of blood coagulation: The role of binding site densities and platelet deposition*, *Biophysical Journal*, 80 (2001), pp. 1050–1074.
- [22] L. LEE AND R. LEVEQUE, *An Immersed Interface method for incompressible Navier-Stokes equations*, *SIAM Journal of Scientific Computing*, 25 (2003), pp. 832–856.
- [23] R. J. LEVEQUE, *High-resolution conservative algorithms for advection in incompressible flow*, *SIAM Journal of Numerical Analysis*, 33 (1996), pp. 627–665.
- [24] R. J. LEVEQUE AND Z. LI, *The Immersed Interface method for elliptic equations with discontinuous coefficients and singular sources*, *SIAM Journal of Numerical Analysis*, 31 (1994), pp. 1019–1044.
- [25] ———, *Immersed Interface methods for stokes flow with elastic boundaries or surface tension*, *SIAM Journal of Scientific Computing*, 18 (1997), pp. 709–735.
- [26] LI AND LAI, *IIM for Navier Stokes eqs with singular forces*, *Journal of Computational Physics*, 171 (2001), pp. 822–842.
- [27] K. G. MANN, S. KRISHNASWAMY, AND J. H. LAWSON, *Surface-dependent hemostatis*, *Seminars in Hematology*, 29 (1992), pp. 213–226.
- [28] A. A. MAYO AND C. S. PESKIN, *An implicit numerical method for fluid dynamics problems with immersed elastic boundaries*, *Contemporary Mathematics*, 141 (1993), pp. 261–277.

- [29] M. L. MINION, *A projection method for locally refined grids*, Journal of Computational Physics, 127 (1996), pp. 158–178.
- [30] E. NEWREN, *Multilevel distributed memory solutions of poisson's equation*, Master's thesis, University of Utah, 1998.
- [31] C. S. PESKIN, *Numerical analysis of blood flow in the heart*, Journal of Computational Physics, 25 (1977), pp. 220–252.
- [32] C. S. PESKIN AND D. M. MCQUEEN, *Modeling prosthetic heart valves for numerical analysis of blood flow in the heart*, Journal of Computational Physics, 37 (1980), pp. 113–132.
- [33] ———, *A three-dimensional computational method for blood flow in the heart: I. immersed elastic fibers in a viscous incompressible fluid*, Journal of Computational Physics, 81 (1989), pp. 372–405.
- [34] A. M. ROMA, C. S. PESKIN, AND M. J. BERGER, *An adaptive version of the Immersed Boundary method*, Journal of Computational Physics, 153 (1999), pp. 509–534.
- [35] U. TROTTEBERG, C. OOSTERLEE, AND A. SCHULLER, *Multigrid*, Academic Press, 2001.
- [36] C. TU AND C. S. PESKIN, *Stability and instability in the computation of flows with moving immersed boundaries: A comparison of three methods*, SIAM Journal on Scientific and Statistical Computing, 13 (1992), pp. 1361–1376.