UCLA
COMPUTATIONAL AND APPLIED MATHEMATICS

Incompressible Navier-Stokes Flow About Multiple Moving Bodies (Ph.D. Thesis)

Archi C. Li

December 1996

CAM Report 96-53

Department of Mathematics
University of California, Los Angeles
Los Angeles, CA. 90024-1555

Incompressible Navier-Stokes Flow About Multiple Moving Bodies

Archie C. Li

January 20, 1997

Contents

Abstract

A computational algorithm is created to solve incompressible fluid flow past multiple moving bodies. The fluid equations are solved with a projection method that is formally second-order accurate in space and fourth order in time. We present a fast method based on integral equations to solve the elliptic problems associated with the projection step. The complicated geometry is dealt with via a cut-out grid formulation and is implemented through an object-oriented design. The cut-out grids can be rapidly generated (an important feature when working with time-dependent geometry), they improve the small cell stability restriction, and they allow both moving and stationary bodies to be modeled with the same algorithm. The object-oriented approach taken has resulted in a set of computational tools (classes) that enable flows involving complex moving geometries to be simulated quickly, and it is anticipated that they will allow others to easily implement and modify algorithms for problems characterized by complicated domains. Flows involving multiple moving bodies are modeled, and computational results are presented.

Chapter 1

Introduction:

This research addresses the solution of incompressible Navier-Stokes flow about multiple moving bodies. A primary motivation for this work is a need for fluid flow solvers that can be used to develop fluid flow control procedures. The development of these procedures requires a solver that is computationally efficient (the design process requires many simulations to be run for extended periods of time) and capable of representing flow about moving objects in a computational domain with non-trivial geometry. In the problems we consider, the objects are given a prescribed motion. This is an appropriate type of simulation for fluid flow control in which the actuators are objects moved in response to a control signal. Extensions to the approach could include problems where the object motion is coupled to the fluid forces exerted upon it, such as fluid-structure interaction problems.

Our flow solver consists of a projection method [10], based on a method of lines approach, where finite differences are used to approximate the spatial derivatives and convert the partial differential equation into a system of ordinary differential equations. The ODEs are solved using fourth-order Runge-Kutta, and the incompressibility constraint is enforced by applying a projection operator. The projection operator extracts the incompressible component of a given vector field by making use of the stream function. We apply the projection using the stream function instead of the pressure for two reasons. First, the stream function approach leads us to an equation that is easier to solve. Specifically, the stream function based projection results in a Poisson problem with Dirichlet data while the pressure formulation leads to a Neumann problem. The Dirichlet problem can be solved uniquely but the Neumann problem does not have a unique solution, and it takes additional effort to deal with this non-uniqueness (In particular, the Neumann problem leads to a singular linear system, and one must deal with a non-empty nullspace when solving this system). The second reason involves the boundary data for these two Poisson problems. In the stream function approach, boundary conditions can be explicitly determined from the

known velocity boundary conditions; however, no explicit pressure boundary conditions are specified for the Navier-Stokes equations [16]. The resulting projection method is formally second-order accurate in space and fourth-order in time.

For simple rectangular geometry, this method can be made computationally efficient by using Cartesian grids, centered differences, and a standard fast Poisson solver. However, if a general, multiply connected, time-dependent domain is considered, then the situation becomes more complicated and different approaches must be developed to deal with the additional computational and theoretical issues.

One issue that arises when we allow more complicated geometry is that it becomes more difficult to discretize the domain. Some type of grid must be created in order to apply our finite differences, but the nature of our simulations limits the possible grid strategies. For example, since the domain can vary with time, we may need a different grid at every time step. This restricts the use of certain body-fitted and unstructured grids, because the construction of new grids of this type at every time step can become computationally expensive.

A cut-out grid (COG) formulation [39, 2, 7, 40, 36, 26, 12, 5] is adopted to address this issue (for additional references, an extensive list is given in [2]). A COG consists of a flagged Cartesian grid and a representation of the boundary of the domain. In our particular implementation, function values are specified on the boundary of the domain and at grid points that lie away from the boundary, while values at grid points near the boundary are defined through interpolation from these known values. This type of grid can be rapidly initialized at each time step, plus at grid points that enter the computational domain, values are naturally defined through this interpolation procedure. Thus COGs are useful in the discretization of complicated time-dependent domains.

Cartesian COGs have been widely used in compressible flow problems [7, 40, 26, 12, 5], with some simulations involving moving geometry. There have been far fewer COG approaches developed for incompressible flows, with exceptions being [2], where the incompressible Euler equations are solved, and [36] where incompressible Navier-Stokes flow is considered. However, in both of these works, only stationary geometry is considered. For our moving boundary applications, the computational demands require a simple and efficient solver, and our particular approach possesses some advantageous differences when compared to other COG formulations. For example, in [2, 36] the fluid equations are discretized using adapted high order Godunov methods. Our approach is somewhat simpler, because at most points straight centered differences are applied. Furthermore, in contrast to [2], no special redistribution techniques are needed to avoid small cell stability problems.

One could also develop solvers using overlapping grids [6, 9, 19, 31, 37, 35], where both compressible and incompressible flows have been modeled. Our COGs are in fact related to overlapping grids, but the COGs are simpler because no local grids have to be generated. Additionally, the simplicity of our procedure

makes it easy to construct our COGs from CAD data. On the other hand, the COG formulation we use does not currently allow extra refinement near the boundaries, and may be best suited for problems whose solutions require refinement throughout the domain (such as separated flow problems).

The complex geometry also causes complications in the projection step. Enforcing the incompressibility constraint requires the solution of Poisson's equation, and this elliptic problem requires a large amount of computational effort. For rectangular domains, a fast Poisson solver can be used to solve Poisson equation efficiently, but something else must be considered for the complex geometry which we will allow. Therefore, we develop a Poisson solver for complex domains by combining an iterative method with a preconditioner based on an integral equation formulation, a standard fast Poisson solver, and a relaxation step. This preconditioner results in an iterative method whose convergence is independent of the mesh width, and this performance is comparable to multigrid algorithms (although it has not been determined which method will be ultimately faster). The use of integral equations in a direct Poisson solver has been considered in [35, 24, 25]; however, this may be the first look at using integral equations in an iterative procedure.

The multiply connectedness of the domain also affects the other component of the projection step, namely determining the appropriate boundary conditions for the aforementioned Poisson problem. For our stream function based projection, the no-flow condition only determines the boundary values of the stream function up to a constant per boundary component. On a simply connected domain there is only one constant to choose and it can be arbitrarily set, but for multiply connected domains the stream function constants must be chosen to produce the correct local circulations (which are specified by the no-slip boundary conditions). In a previous work [35] these constants were chosen at a cost of solving one elliptic equation per boundary component, but this results in a large amount of computation when dealing with time-dependent, highly multiply connected domains. In this thesis, we show how an integral equation formulation can be used to simultaneously determine all of the boundary constants with a single elliptic solve, thus maintaining efficiency when the boundary possesses a large number of components.

This research required many different techniques to be brought together to form one successful procedure. To facilitate the combination and alteration of diverse methods, an object-oriented design was employed in the development of the software. One feature of object-oriented design is encapsulation, where the information and functions needed to accomplish a specific goal are grouped together with a simple interface. This is very useful when a large complicated program can be constructed from several smaller parts. This is normally considered to be a standard objective of good programming style, but it is facilitated by the use of an object-oriented language like C++. A second attribute is that of derivation and inheritance, where a fundamental base class is created and subsequent adaptations of that type are built on top of the base class. In

our implementation, all geometric entities are derived from a base class that provides a common interface for information about the geometry. This allows complex domains to be easily represented, and our fluid solvers are independent of the particular test geometries used in the simulations. For example, the current solver can have its geometry specified from a CAD package without requiring any new program modifications. It is anticipated that these software tools will allow others to easily implement algorithms for problems characterized by complicated geometry.

In summary, a numerical algorithm is presented for solving fluid flow about multiple moving bodies. A cut-out grid formulation is introduced to allow rapid regridding of the time-dependent domain, and integral equations techniques are used to efficiently solve the elliptic problems that arise. In order to ease the implementation of these different techniques, an object-oriented approach to software design was taken which resulted in the creation of computational tools for solving problems in complex domains.

In Chapter 2 we deal with issues relating to COGs. The first section introduces what a COG is and how to represent a function on a COG. The next section covers the key numerical steps involved in creating a COG representation of a function. The third section explains how to formulate Poisson's equation on a COG and how to solve that Poisson problem rapidly by incorporating an iterative method, integral equations, a fast Poisson solver, and relaxation techniques. In addition we discuss how the COG class library can be useful for problems besides those involving incompressible fluid flow.

Chapter 3 involves the solution of the incompressible Navier-Stokes equations. The details of a stream function based projection method is presented, and we show how integral equations can be used to efficiently apply the projection step in multiply connected domains. The discretization of the spatial derivatives is accomplished by incorporating COGs, and the stability benefits of the resulting formulation is explained. Finally, computational results are presented for stationary and time-dependent domains.

In Chapter 4 we include the integral equation material that is used in the previous Chapters. In the first section we review the relevant theory, cite needed results, and derive integral formulations of the Laplace problem in multiply connected domains. The second section addresses the numerical issues involved in the computational solution and evaluation of integral equations, and we discuss the fast techniques that are available to us. In the final section we develop the integral equation formulation that allows us to efficiently determine the stream function's boundary values in a multiply connected domain (these values are needed in our projection step).

Chapter 2

Cut-Out Grids:

When modeling a problem with complex geometry, a major computational issue is how to represent the domain. There are several possible choices available to us including body-fitted grids, unstructured grids, Cartesian grids, and cut-out grids (COGs). The nature of the problem will generally favor some gridding strategies over others, and one must consider how well the grid resolves the geometry, what cost is involved in generating the grid, and how easy and efficient it is to discretize the problem on that grid. For the fluid simulations addressed in this work, a cut-out formulation is employed which can be rapidly generated (even for complex domains), it easy to apply finite differences on, and whose regular structure permits the use of certain fast solvers. In this chapter, we first introduce what a COG is and discuss how to represent a function on a COG. Next, some key numerical issues associated with cut-out grids are examined, and in the final section we give a cut-out grid formulation of Poisson's equation and present ways to solve this problem efficiently. An early exploration of this type of grid was presented in [39], and we will adopt the terminology and notation that appears in that reference.

2.1 Introduction:

A COG is a discrete representation of a domain. It can be set up rapidly, even for complicated domains, and this makes it useful for problems characterized by complicated, time-dependent geometry. Consider the domain exterior to a small circle and bounded by a unit square (see Fig. 2.1). A Cartesian grid that contains the domain is created, and the grid points are flagged as either being exterior to the domain, interior to the domain, or on the boundary of the domain. Interior and exterior points are further classified as being regular or irregular points. A regular point is one whose nearest neighbors all lie on

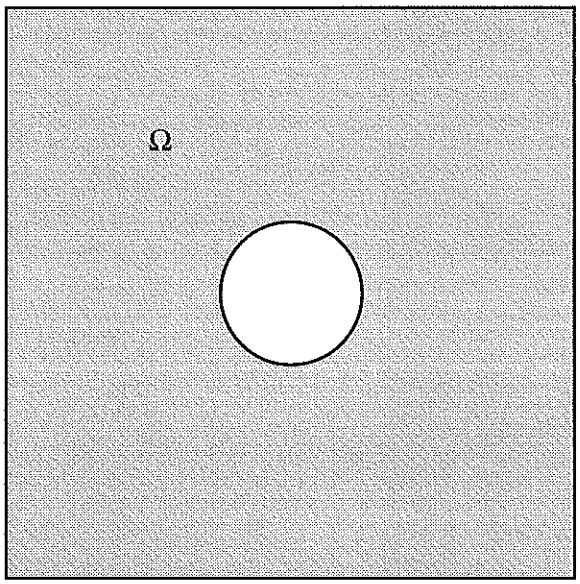

Figure 2.1: A sample domain.

the same side of the boundary, while at an irregular point some of the nearest neighbors lie on the opposite side of the boundary (or on the boundary itself). Thus, if a point is interior to the domain and all of its nearest neighbors are also interior, then that point is a regular point. If a point is interior to the domain but some of its nearest neighbors are exterior (or boundary) points, then that point is irregular. Boundary grid points must know where they lie on the boundary (with respect to the parameterization being used), and irregular points need to know where their five point stencils intersect the boundary. (For an example of a five point stencil see Figs. 4.7 and 4.8) This information is sufficient to define a COG for a given domain (see Fig. 2.2).

To represent a function on the COG, one specifies the function values at each regular interior point plus the values on the boundary of the domain. The values at boundary grid points are specified by the given boundary data, and irregular interior points have values defined by interpolation between the known values at the regular interior points and the known boundary values. This combination of geometry, grid, and data information will be referred to as a cut-out grid function.

2.2 Numerical issues:

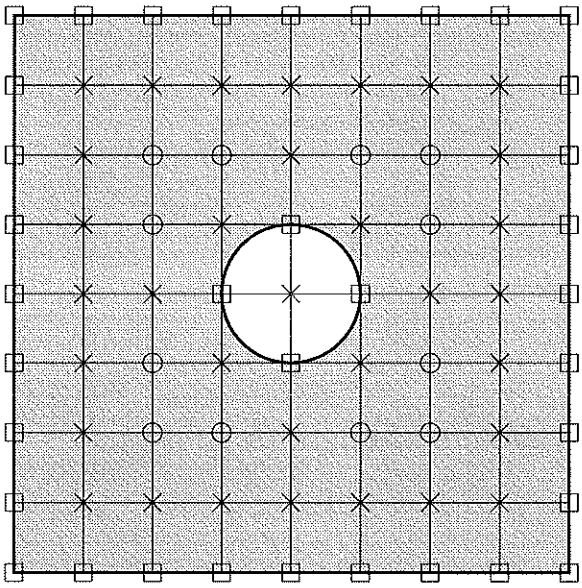

Figure 2.2: A coarse COG: regular points are marked by circles, irregular points by crosses, and boundary points by squares.

2.2.1 Identification of irregular and boundary points:

In forming a COG, an algorithm must be chosen to correctly flag the grid points. The identification of irregular, regular, and boundary points can be done efficiently by searching in a boundary-to-grid fashion. First, all grid points are initially flagged as regular points, and then the boundary is divided into intervals whose arclength is less than the minimum mesh width of the Cartesian grid. Each boundary interval is visited, and all grid points that lie near the interval are checked to see if they lie on the interval or if their five point stencils intersect that interval. Due to the structure of the Cartesian grid, the nearby points can be identified without having to search the grid (for instance, dividing the horizontal distance from a boundary point to the left edge of the grid by the mesh width produces the horizontal spatial index of a nearby grid point). Furthermore, since we only consider one boundary interval at a time, we can test if a point is an irregular or boundary point of the given interval within a constant number of operations. The specific computations will depend on what type of contour it is and how it is represented, but the local nature of the test will limit its expense. Thus by using this boundary-to-grid procedure, the cost of identifying irregular and boundary grid points increases inversely with the size of the smallest mesh width, and for a two dimensional grid with n unknowns the cost only increases like $O(n^{\frac{1}{2}})$.

2.2.2 Identification of interior and exterior points:

Another concern in creating a COG is identifying whether the grid points lie interior or exterior to the domain. For some contours a simple test exists to determine if a given point is interior or exterior. (For example, a point is interior to a circle if the distance from the point to the center of the circle is less than the radius). In these cases we can just visit each grid point and apply this simple test. For general geometries it is more difficult to efficiently identify the grid points, and something else must be tried. Two approaches to this problem will be presented, the first involving integral equations (a similar procedure can be found in [38]).

Consider the case for a bounded domain Ω whose boundary consists of one bounding contour $\partial\Omega_{M+1}$ and M inner contours $\partial\Omega_1 \cdots \partial\Omega_M$. Define $I(\boldsymbol{x})$ as a double layer potential with unit charge density.

$$I(\boldsymbol{x}) = \int_{\partial\Omega} \phi(\boldsymbol{y}) \frac{\partial}{\partial\eta(\boldsymbol{y})} \left(\frac{1}{2\pi} \log \mid \boldsymbol{x} - \boldsymbol{y} \mid\right) ds(\boldsymbol{y}) \tag{2.1}$$

here $\eta = \eta_{bdd}$ on the inner contours, and $\eta = \eta_{unbdd}$ on $\partial\Omega_{M+1}$ (this notation is defined in Section 4.2). By applying Green's theorem, we can determine the value of I for different points in the plane (In the following, Ω^c refers to the complement of the domain Ω).

$$I(\boldsymbol{x}) \quad = \begin{cases} 1 & , \ \boldsymbol{x} \in \Omega \\ \frac{1}{2} & , \ \boldsymbol{x} \in \partial\Omega \\ 0 & , \ \boldsymbol{x} \in \Omega^c \end{cases}$$

Therefore, we can determine if a grid point lies in the domain, out of the domain, or on the boundary by evaluating the double layer potential. (Note: the density is specified, not solved for, so the only cost lies in the evaluation.) A direct numerical evaluation of the potential on the grid points would require an unacceptable amount of computation; however, by using the method of local correction described in Section 4.3.3 we can do this evaluation rapidly (asymptotically, it is $O(n \log(n))$). Therefore if we are comfortable with using fast integral techniques, this double layer potential approach is a computationally viable option.

A different approach does not rely on integral equations but it is related to the local correction paradigm. The first step is to correctly flag all irregular points as either interior or exterior. There are many ways this can be done, for example the double layer potential (Eq. 2.1) could be evaluated at all irregular points (either through direct computation or by using the FMM). Another possibility uses the orientation of the boundaries: In a procedure similar to the one used to identify irregular and boundary points, the boundary is divided

into small intervals. The inner contours are traversed in a clockwise orientation and the bounding contour in a counter-clockwise fashion. For each boundary interval, nearby irregular grid points are quickly located by exploiting the structure of the Cartesian grid. Each irregular point can be identified as interior or exterior by forming the cross product of the position vectors of the endpoints of the boundary interval with respect to the irregular point. If this cross product is positive the irregular point is interior to the domain, and if negative then it is an exterior point. (see Fig. 2.3)

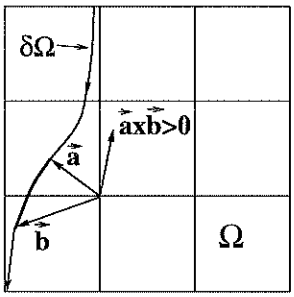

Figure 2.3: Using an oriented boundary interval to determine in/out.

Once all the irregular points have been flagged as interior or exterior, the regular grid points are traversed in a left to right, bottom to top manner. As each grid point is visited, it is flagged as interior if its left neighbor is interior, and exterior if its left neighbor is exterior. For a regular point on the left edge of the grid (which has no left neighbor), a default state of exterior will be used when the domain is bounded, and for unbounded domains the default state will be be interior. Since two adjacent (non-boundary) points cannot have different states unless the boundary lies between them, and since the irregular points have already been correctly flagged, this sequential traversal will identify the interior/exterior status of all regular points. For a Cartesian grid with n grid points, the irregular points can be checked in $O(n^{\frac{1}{2}})$ operations. Then the regular points can be flagged in $O(n)$ operations, so the overall procedure is $O(n)$.

We recognize that there are many algorithms that can identify grid points as interior or exterior to a given domain, and none of them are optimal for all situations. We have presented three strategies that can each be attractive in certain settings. If the boundary consists of simple shapes such as circles and rectangles, then a direct query can be made at each point on the grid. If an arbitrary domain is allowed, and if the user is familiar with fast integral techniques, then the double layer potential approach may be preferable (the integral approach has the added advantage that it readily extends to three dimensions). Finally if one has an arbitrary domain where it is easy to correctly flag the irregular points (for example using the cross product procedure we

present) then a sequential traversal can be used to identify all other grid points. Each procedure has its own merits and each can be a valid approach to this simple but interesting problem in computational geometry.

2.2.3 Interpolation at irregular points:

Another issue is how to obtain function values at irregular interior points when the function is known on the boundary and at the regular interior points. To solve the Poisson equation on a COG, we only need a second-order interpolant in order to maintain a second-order method [39]. However, when modeling the Navier-Stokes equations, higher order interpolation is needed, and we apply a fourth-order Lagrange interpolating polynomial. For example if an irregular interior point, $\boldsymbol{x}_{i,j}$, has three regular interior points, $\boldsymbol{x}_{i-3,j}, \boldsymbol{x}_{i-2,j}$, and $\boldsymbol{x}_{i-1,j}$, on its left and has the boundary at a distance, d_R , on its right(see Fig. 2.4), then the value at $\boldsymbol{x}_{i,j}$ will be determined by the following formula (h denotes the horizontal mesh with).

$$\begin{aligned} u(\boldsymbol{x}_{i,j}) &= u(\boldsymbol{x}_{i-3,j})\frac{d_R}{3h+d_R} - u(\boldsymbol{x}_{i-2,j})\frac{3d_R}{2h+d_R} + \\ & u(\boldsymbol{x}_{i-1,j})\frac{3d_R}{h+d_R} + u(\boldsymbol{x}_R)\frac{6h^3}{(3h+d_R)(2h+d_R)(h+d_R)} \end{aligned} \tag{2.2}$$

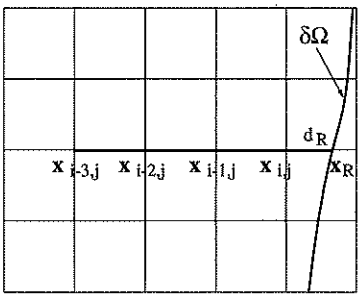

Figure 2.4: Fourth-order collinear interpolation at an irregular point.

If the geometry does not allow horizontal interpolation, then vertical interpolation will be tried. However, in some cases there will not be enough known values nearby to allow collinear interpolation in either direction. To mitigate this, we allow the interpolation to proceed in sweeps. After the given interpolation has been used to fill in all accessible irregular points, the process is repeated. Now that more of the points have values, additional irregular points can be filled in. An arbitrary number of sweeps can be prescribed, but in practice a two step process is used. There are some cases which defy this fix as well,

such as when a point is located near the tip of a sharp corner. In these cases a lower order interpolant is used, although more complicated approaches could be used to maintain high order accuracy.

2.2.4 Software design:

The preceding numerical algorithms and additional numerical concerns have been implemented and embedded in a hierarchical COG class structure. Classes have been created to represent the boundary geometry, the underlying Cartesian grids, and the data flags used to classify each Cartesian point. Simple interfaces exist for each COG component, and the operators and functions that each component needs is included in that particular class. The resulting class structure is a self contained, flexible tool for the representation of complex geometry, and it allows a user to incorporate COGs without having to know the details of the implementation. This software is used to implement the Navier-Stokes simulations involving relative motion of bodies in a flow field, yet it is not exclusively tied to the fluid solver. Potential applications involve bubble simulations and heart modeling where the the COG representation could be combined with an integral equation or level set algorithm.

2.3 Poisson's Equation:

In the fluid simulations we will consider, several Poisson equations must be solved at each time step on different domains. The solution of these elliptic equations accounts for a large portion of the computational effort involved in the overall numerical algorithm, so it is crucial that we be able to solve Poisson's equation efficiently on complex domains. To do this, we apply a COG discretization to approximate the Poisson problem by a discrete system of equations. Next, we consider how to solve this discrete system efficiently. We begin by describing a solver that combines integral equations, a fast Poisson solver, and relaxation techniques. This solver produces an approximate solution to Poisson's equation which also approximately satisfies the discrete equations. We then use this "direct" solver to precondition an iterative method. This preconditioned iterative method is used to efficiently solve the discrete Poisson equations, and its convergence rate is independent of the grid's mesh width. Computational results are given for both the "direct" and iterative methods.

2.3.1 The discrete Poisson problem on a COG:

Consider the bounded domain that lies inside a unit box and outside a small circle within the box (Fig. 2.1). The following is a statement of Poisson's equation with Dirichlet data for this domain.

$$\begin{aligned} \Delta u(\boldsymbol{x}) &= f(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega \\ \lim_{\substack{\boldsymbol{x} \to \boldsymbol{x}_o \\ \boldsymbol{x} \in \Omega}} u(\boldsymbol{x}) &= g(\boldsymbol{x}_o), \quad \boldsymbol{x}_o \in \partial\Omega \end{aligned} \tag{2.3}$$

In a COG formulation, we can satisfy the boundary conditions by creating a COG function $\tilde{u}$ and prescribing the Dirichlet data as its boundary values. To determine the regular interior values of $\tilde{u}$, we apply a finite difference approximation of the Laplacian operator to obtain a set of discrete equations that the regular interior values must satisfy. In our simulations, the standard five point discrete Laplacian is used.

$$\Delta_h \tilde{u}(\boldsymbol{x}_{i,j}) \equiv \frac{\tilde{u}(\boldsymbol{x}_{i+1,j}) + \tilde{u}(\boldsymbol{x}_{i-1,j}) + \tilde{u}(\boldsymbol{x}_{i,j-1}) + \tilde{u}(\boldsymbol{x}_{i,j+1}) - 4\tilde{u}(\boldsymbol{x}_{i,j})}{h^2} \tag{2.4}$$

There will be one discrete equation for each unknown regular interior point, so no additional equations or constraints are needed.

$$\begin{aligned} \Delta_h \tilde{u}(\boldsymbol{x}_{i,j}) &= f(\boldsymbol{x}_{i,j}), \quad \boldsymbol{x}_{i,j} \text{ is a regular interior point.} \\ \tilde{u}(\boldsymbol{x}_o) &= g(\boldsymbol{x}_o), \quad \boldsymbol{x}_o \in \partial\Omega \end{aligned} \tag{2.5}$$

We note that values at irregular points are needed in order to compute the discrete Laplacian at regular points nearest to the boundary. These values are defined (as discussed in Section 2.1) through interpolation between the values at the regular interior points and those on the boundary. Thus the discrete equations involve both the forcing and the boundary values.

If the interpolation accuracy is second-order or higher, then solving the discrete equations produces a second-order accurate solution to the Poisson problem. A convergence proof is given in [39], and we also check this result numerically. We select a test solution, $u(\boldsymbol{x}) = x^4 + \frac{1}{2}y^4$, and use it to generate the forcing and boundary values on our test geometry (a circle of radius 1/4 centered within a unit square). The discrete equations (2.5) are solved iteratively up to a relative residual error of 10^{-10}, and standard pointwise and maximum norm error analysis reveals second-order convergence. (see Table 2.1).

We now consider how to solve the discrete equations efficiently. In the next section we examine a method which combines integral equations and a fast Poisson solver to produce a solution which approximately solves both the analytic and discrete Poisson equations. In Section 2.3.3, we then use this solver as a preconditioner for an iterative method. This preconditioner increases the efficiency of the iterative method by greatly reducing the number of iterations required.

Table 2.1: Convergence study of COG formulation of Poisson's equation

Grid	$\|\tilde{u}-u\|_\infty$	Order	$\|\tilde{u}(.15,.75)-u(.15,.75)\|$	Order
20x20	0.00205		0.00026	
40x40	0.00031	2.7	5.5e-05	2.2
80x80	7.1e-05	2.1	1.1e-05	2.3

2.3.2 Solving the Poisson problem, a "direct" method:

In Chapter 4 we discuss how to use a modified double layer potential to solve Laplace's equation with Dirichlet data. In order to use these methods on Poisson's equation, we apply a fast Poisson solver to satisfy the inhomogeneous forcing terms. In Equation 2.5, the discrete forcing terms $f(\boldsymbol{x}_{i,j})$ are known only at the regular interior points of the Cartesian grid. If we fill in the values at the remaining grid points with zeroes we can call a fast Poisson solver to rapidly provide a solution to the Poisson problem on the simple rectangular domain represented by the Cartesian grid. (In the following equation, an "edge point" refers to the grid points that make up the edge of the Cartesian grid.)

$$\Delta_h u^{FPS}(\boldsymbol{x}_{i,j}) = f^{FPS}(\boldsymbol{x}_{i,j}) \tag{2.6}$$

where,

$$f^{FPS}(\boldsymbol{x}_{i,j}) = \begin{cases} f(\boldsymbol{x}_{i,j}), & \boldsymbol{x}_{i,j} \text{ is a regular interior point.} \\ 0, & \boldsymbol{x}_{i,j} \text{ is any other non-edge point.} \end{cases} \tag{2.7}$$

$$u^{FPS}(\boldsymbol{x}_{i,j}) = g(\boldsymbol{x}_{i,j}), \quad \boldsymbol{x}_{i,j} \text{ is an edge point of the Cartesian grid.}$$

Note: For geometries where the edge points, $\boldsymbol{x}_{i,j}$, of the Cartesian grid do not lie on the boundary of the domain, the values there will be set to zero ($u^{FPS}(\boldsymbol{x}_{i,j}) = 0$). This solution satisfies the discrete equations at the regular interior points; however, generally it will not satisfy the boundary conditions. Therefore, a correction term is sought by solving Laplace's equation with appropriate boundary conditions.

$$\begin{aligned} \Delta u^{IE}(\boldsymbol{x}) &= 0, \quad \boldsymbol{x} \notin \partial\Omega \\ \lim_{\substack{\boldsymbol{x}\to\boldsymbol{x}_o \\ \boldsymbol{x}\in\Omega}} u^{IE}(\boldsymbol{x}) &= g(\boldsymbol{x}_o) - u^{FPS}(\boldsymbol{x}_o), \quad \boldsymbol{x}_o \in \partial\Omega \end{aligned} \tag{2.8}$$

This problem can be solved and evaluated at the regular interior points by using the integral equation approach of Chapter 4, and an approximate solution to the Poisson problem can be formed by combining the fast Poisson solver solution with the boundary correction terms. That is, if

$$\tilde{u}(\boldsymbol{x}_{i,j}) \equiv u^{FPS}(\boldsymbol{x}_{i,j}) + u^{IE}(\boldsymbol{x}_{i,j}), \tag{2.9}$$

then

$$\begin{aligned} \Delta_h \tilde{u}(\boldsymbol{x}_{i,j}) &= f(\boldsymbol{x}_{i,j}) + O(h^2), \quad \boldsymbol{x}_{i,j} \text{ is a regular interior point.} \quad (2.10) \\ \tilde{u}(\boldsymbol{x}_o) &= g(\boldsymbol{x}_o), \quad \boldsymbol{x}_o \in \partial\Omega. \end{aligned}$$

This method was used to solve Poisson's equation in a previous work [35], and a similar approach was taken in [25]. It should be mentioned that since $u^{FPS}(\boldsymbol{x})$ is only known at the grid points, its boundary values are obtained through interpolation. Since the discrete forcing terms (in Eq. 2.6) are discontinuous near the boundary we expect a decrease in the smoothness of $u^{FPS}(\boldsymbol{x})$ across the boundary. Therefore it may be harder to interpolate onto the boundary. To remedy this, we can modify how we extend the forcing terms on the exterior and irregular grid points. Instead of filling them with zeroes, we can apply a few relaxation sweeps. That is, we fill in the values at those points with the average of their nearest neighbors. If a point Gauss-Seidel (GS) procedure is used we obtain a simple formula for updating the values.

$$\begin{aligned} & f^{FPS^{n+1}}(\boldsymbol{x}_{i,j}) \\ &= \frac{f^{FPS^{n+1}}(\boldsymbol{x}_{i-1,j}) + f^{FPS^{n}}(\boldsymbol{x}_{i+1,j}) + f^{FPS^{n+1}}(\boldsymbol{x}_{i,j-1}) + f^{FPS^{n}}(\boldsymbol{x}_{i,j+1})}{4} \quad (2.11) \\ & \boldsymbol{x}_{i,j} \text{ is an irregular or exterior point.} \quad (2.12) \end{aligned}$$

This pointwise averaging is equivalent to solving the heat equation for a short time, and it will prevent a discontinuity in the forcing terms. A different way of dealing with this problem was taken in [25] which uses the procedure given by Mayo in [24]. There, the discrete forcing terms are modified (by solving for and incorporating certain jump information) only at the irregular points, and a special interpolation formula is used which takes the loss of smoothness into account. In our problem, the forcing is not prescribed outside of our domain; therefore, we are free to extend it as we see fit. This smooth extension cannot be applied to the range of problems that Mayo's procedure can (such as true interface problems where the forcing is supposed to be discontinuous), but for our applications it is effective and simpler.

Computational results:

A numerical study of the accuracy was done for the given test geometry where (as before) the exact solution $u(\boldsymbol{x}) = x^4 + \frac{1}{2}y^4$ was used to initialize the forcing and boundary values. We placed 640 boundary points on each boundary component, evaluated the integral corrections using direct summation, and generated approximate Poisson solutions on grids of varying refinement. In Table 2.2, no smoothing steps are taken, and the forcing terms are simply extended with zeroes. In Table 2.3, four Gauss-Seidel sweeps were used to smoothly extend the discrete forcing terms. We see that the convergence rate is roughly second-order

in both cases, but extending the data with four Gauss-Seidel smoothing steps produced solutions which are approximately three times more accurate than solutions using the non smoothed data.

Table 2.2: Convergence study of FPS + integral equations, no smoothing.

Grid	$\|\tilde{u} - u\|_\infty$	Order	$\|\tilde{u}(.15,.75) - u(.15,.75)\|$	Order
20x20	0.0313		0.0071	
40x40	0.0092	1.8	0.0020	1.8
80x80	0.0024	2.0	0.0005	1.9

Table 2.3: Convergence study of FPS + integral equations, four GS smoothings.

Grid	$\|\tilde{u} - u\|_\infty$	Order	$\|\tilde{u}(.15,.75) - u(.15,.75)\|$	Order
20x20	0.0134		0.00261	
40x40	0.0031	2.0	0.00065	2.0
80x80	0.0017	0.9	0.00015	2.1

Asymptotically, the efficiency of the preceding Poisson solver is limited by the fast Poisson solver, whereas the fast integral techniques can be asymptotically optimal. In practice, however, the time required to solve the integral equation can be considerably more than the time used by the fast Poisson solver. Because of this, we would like to reduce the number of boundary points used in the integral equation discretization. This will increase the speed of the resulting Poisson solver, but if too few boundary points are used, then the accuracy of the solution will suffer. So by decreasing the number of boundary points used, we obtain a fast approximate solver, and although it may not be accurate enough to use as a solver it makes an excellent preconditioner for an iterative method. This iterative procedure will now be presented.

2.3.3 Solving the Poisson problem, an iterative method:

As we demonstrated in section 2.3.1, we can obtain a second-order accurate solution to Poisson's equation by solving the discrete COG Poisson equations. These equations (2.5) can be written in matrix form,

$$A\vec{u} = \vec{b} \tag{2.13}$$

where $\vec{u}$ represents the values at the regular interior points, $\boldsymbol{A}$ represents the effect of filling in values at irregular interior points and then applying the discrete Laplacian at all regular interior points, and $\vec{b}$ represents both the discrete forcing at the regular points and the given boundary data. In solving the matrix equation, an iterative method is often preferable to a direct method because $\boldsymbol{A}$ is sparse, it can be very large, and it can vary with time. Due to the interpolation step, $\boldsymbol{A}$ is not symmetric so the nonsymmetric iterative method FGMRES [33] is used. It is well known that for Poisson's equation, the condition number of $\boldsymbol{A}$ increases as the number of grid points increases, so as we refine the mesh the iterative method will require more iterations to achieve the same level of accuracy. This makes FGMRES costly for fine meshes, and this is usually remedied by preconditioning the matrix equation. A preconditioner is often thought of as a matrix that is applied to the original linear system in order to transform it into a better conditioned system. One can also think of a preconditioner as an approximate inverse to the matrix $\boldsymbol{A}$, or functionally as an approximate solver for the underlying partial differential equation. Therefore, we take the Poisson solver that we have introduced (in Section 2.3.2), reduce the number of boundary points used, and apply the resulting fast approximate solver as a preconditioner for FGMRES.

To facilitate this functional approach to solving the matrix equation, a data-structure-neutral implementation is used [18, 17, 4], where instead of passing in vectors and matrices, the user supplies pointers to data structures and functions that perform such basic operations as addition, scalar multiplication, and applying the matrix (and preconditioner). This approach is useful because it allows us to avoid explicitly forming the matrix and preconditioner, and it allows us to represent and store our unknowns in any form that is convenient. A brief description of this solver is given in Appendix A.

When the preconditioner is called, the forcing terms it receives are the residual errors of the iterative method, $\vec{r}_i$. These residuals can be highly oscillatory (see Fig. 2.5), and there may be difficulties when interpolating the fast Poisson solution onto the boundary (Eqs. 2.6-2.8), even if we fill in the irregular and exterior values smoothly. To address this issue, we incorporate a relaxation scheme as part of the preconditioning step. A common feature of relaxation schemes is that they result in approximate solutions with smooth errors, even after only a few iterations. Therefore, we apply our approximate Poisson solver to the smooth error equation resulting from the relaxation step, and then combine the two terms to form the preconditioned residual.

In summary, we generate a preconditioned residual, $\tilde{r}$, which approximately satisfies a Poisson problem where the forcing consists of the residual data, and zero boundary data is prescribed. That is, we seek an approximate solution to the following equations:

$$\begin{aligned} \Delta_h \tilde{r}(\boldsymbol{x}_{i,j}) &\approx r(\boldsymbol{x}_{i,j}), \quad \boldsymbol{x}_{i,j} \text{ is a regular interior point.} \\ \tilde{r}(\boldsymbol{x}_o) &= 0, \quad \boldsymbol{x}_o \in \partial\Omega \end{aligned} \tag{2.14}$$

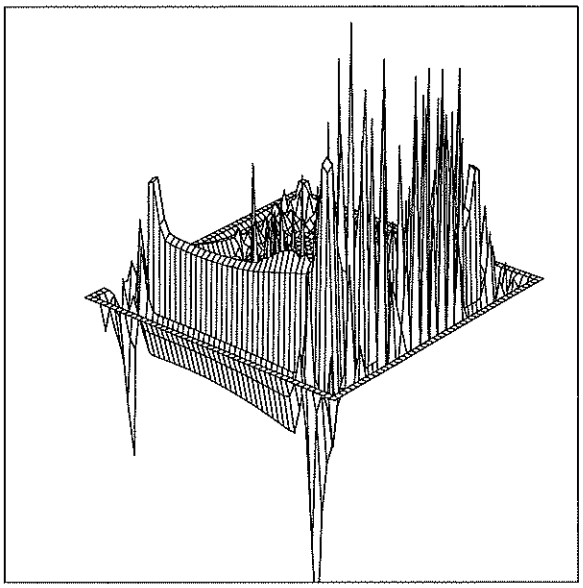

Figure 2.5: An unsmoothed residual error of FGMRES.

First, a few iterations of a Gauss-Seidel relaxation scheme are applied.

$$\begin{aligned} v^0(\boldsymbol{x}_{i,j}) &= 0 \\ v^n(\boldsymbol{x}_o) &= 0; \quad \boldsymbol{x}_o \in \partial\Omega, \quad (\forall n) \\ v^{n+1}(\boldsymbol{x}_{i,j}) & \qquad (2.15) \\ = \frac{v^{n+1}(\boldsymbol{x}_{i-1,j})+v^n(\boldsymbol{x}_{i+1,j})+v^{n+1}(\boldsymbol{x}_{i,j-1})+v^n(\boldsymbol{x}_{i,j+1})-h^2 r(\boldsymbol{x}_{i,j})}{4}; & \quad n = 0, 1, 2, \ldots \end{aligned}$$

After the relaxation step, the approximate Poisson solver of Section 2.3.2 is called to approximately solve the smooth error equation.

$$\begin{aligned} \Delta_h e(\boldsymbol{x}_{i,j}) &\approx r(\boldsymbol{x}_{i,j}) - \Delta_h v^{n+1}(\boldsymbol{x}_{i,j}) \qquad (2.16) \\ e(\boldsymbol{x}_o) &= 0, \quad \boldsymbol{x}_o \in \partial\Omega \end{aligned}$$

Finally, the iterative term is combined with its correction term to form the preconditioned residual.

$$\begin{aligned} \Delta_h(v^{n+1}(\boldsymbol{x}_{i,j}) + e(\boldsymbol{x}_{i,j})) &\approx r(\boldsymbol{x}_{i,j}), \text{ is a regular interior point.} \qquad (2.17) \\ v^{n+1}(\boldsymbol{x}_o) + e(\boldsymbol{x}_o) &= 0, \quad \boldsymbol{x}_o \in \partial\Omega \\ \Rightarrow \tilde{r} &\equiv v^{n+1} + e \qquad (2.18) \end{aligned}$$

Computational results:

We revisit our test problem where the geometry consists of a circle of radius 1/4 centered within a unit square, and the exact solution is $u(\boldsymbol{x}) = x^4 + \frac{1}{2}y^4$. The preconditioned iterative solver , FGMRES [33], is used to solve the discrete equations, and we monitor the number of iterations needed to reduce the relative residual error to a given tolerance ($\frac{\|\boldsymbol{A}\vec{u}_k - \vec{\delta}\|}{\|\vec{\delta}\|} < 10^{-10}$) for varying levels of mesh refinement. We use FGMRES over plain GMRES [34], because we are using an effective but moderately complicated preconditioner. (FGMRES only requires half as many preconditioner calls as GMRES, and we can afford the extra storage costs incurred since the total number of iterations is small.) We first apply FGMRES without any precondtioning, and then with a simple preconditioner consisting of four Gauss-Seidel sweeps. Next, we precondition the system by using our combined integral equation and fast Poisson solver approach, both with and without the added relaxation steps. In the last two cases, the integral correction is evaluated on the grid using direct summation where 320 boundary points are used to discretize each boundary components. Results are listed in Table 2.4. We see that the number of iterations needed to achieve

Table 2.4: Iteration count for different preconditioners.

	Preconditioner			
Grid	none	GS4	FPS+IE	GS4+FPS+IE
20x20	56	18	7	5
40x40	127	34	8	6
80x80	264	68	8	6

the tolerance increases with the number of unknowns when no preconditioning or Gauss-Seidel preconditioning is used. In contrast, if the fast Poisson solver with integral correction is used to precondition the system, then the convergence of the iterative solver is independent of the number of unknowns used. Furthermore, the addition of a few relaxation steps reduces the number of iterations needed by two, which seems worthwhile considering the relative cheapness of applying Gauss-Seidel.

We now consider how the effectiveness of the preconditioner depends on the accuracy of the integral equation solver. We fix the Cartesian grid to 80x80 panels, and we apply our preconditioners with varying numbers of boundary points. We see that our integral equation preconditioner continues to reduce the number of iterations needed even when the boundary is not fully resolved, and that the relaxation steps allow even fewer boundary points to be used. With the added relaxation steps, our preconditioner is just as effective when 40 points per object are used as when 320 points per object are used. Therefore, we can

greatly speed up our preconditioner (by reducing the number of boundary points used) without compromising the convergence behavior of the iterative method. Results are given on Table 2.5.

Table 2.5: Iteration count for different boundary discretization levels.

# of Boundary pts	Preconditioner	
	FPS+IE	GS4+FPS+IE
30	33	9
40	27	6
80	10	6
160	9	6
320	8	6

This preconditioner is preferable to ones based on incomplete factorization because the latter do not result in mesh width independent convergent schemes. Furthermore, in our applications the linear system can change with each time step; therefore, we cannot spread out the cost of doing the incomplete factorization over several Poisson solves. More natural competitors are the multigrid based preconditioners for they also result in refinement independent convergence, and it remains to be seen which preconditioner results in a faster method for the simulations considered in this thesis.

Chapter 3

Navier-Stokes flow:

In this chapter we are interested in solving the incompressible Navier-Stokes equations which we use to model two dimensional fluid flow. The numerical algorithm used is a projection method [10] that is based on a method of lines approach. The spatial derivatives are approximated by finite differences on a staggered COG representation of the domain. This converts the partial differential operator into a system of ordinary differential equations, and a fourth-order Runge-Kutta scheme is used to solve the ODEs in time. At each Runge-Kutta stage, we apply a projection step to ensure that the fluid remains incompressible. We base the projection step on a stream function formulation which leads to a uniquely solvable Dirichlet problem whose boundary values are determined by the given conditions on the velocity. (In contrast, a pressure based projection leads to a non-unique Neumann problem, and the pressure boundary conditions are not explicitly prescribed.) The use of COGs allows a rapid gridding of the time-dependent geometry, and an additional benefit is the avoidance of an arbitrarily small stability restriction. Other sources of difficulties lie in the projection step where computationally intensive elliptic equations must be solved in order to enforce the incompressibility constraint. Integral equations can be very useful in this projection step, and a method is developed which is efficient even in highly multiply connected domains. The overall method is formally second-order accurate in space and fourth-order in time.

3.1 Statement of the problem:

Consider the incompressible Navier-Stokes equations in a time-dependent domain $\Omega(t)$.

$$\vec{u}_t + (\vec{u} \cdot \nabla)\vec{u} = \nu \Delta \vec{u} - \nabla p \tag{3.1}$$

$$\nabla \cdot \vec{u} = 0 \tag{3.2}$$

Here, $\vec{u}$ represents the fluid velocity field, ν is the kinematic coefficient of viscosity, and p is the pressure. The first system of equations reflects the conservation of momentum, while the second represents the incompressibility constraint arising from the conservation of mass for a fluid with constant density. Due to the presence of viscosity, both no-slip and no-flow boundary conditions must be prescribed, and the initial condition must be a divergence free field that satisfies the boundary conditions.

$$\begin{aligned} \vec{u}(\boldsymbol{x},t) &= \vec{g}(\boldsymbol{x},t), \quad \boldsymbol{x} \in \partial\Omega(t) && (3.3) \\ \vec{u}(\boldsymbol{x},t=0) &= \vec{u}_o(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega(t=0) && (3.4) \end{aligned}$$

We discretize the time derivative, and for expository purposes the forward Euler approximation will be used. In practice (see Section 3.3), we use a fourth-order Runge-Kutta time approximation which can be viewed as a collection of Euler-like steps.

$$\begin{aligned} \frac{\vec{u}^{n+1}-\vec{u}^n}{\Delta t} + (\vec{u}^n \cdot \nabla)\vec{u}^n &= \nu\Delta\vec{u}^n - \nabla p^{n+1} && (3.5) \\ \nabla \cdot \vec{u}^{n+1} &= 0 && (3.6) \end{aligned}$$

The current velocity field $\vec{u}^n$ is known, and a rearrangement of terms reveals that the momentum equation produces the velocity field at the next time step plus a gradient (conservative) field.

$$\begin{aligned} \vec{v} &\equiv \vec{u}^n + \Delta t(-(\vec{u}^n \cdot \nabla)\vec{u}^n + \nu\Delta\vec{u}^n) && (3.7) \\ \vec{u}^{n+1} &= \vec{v} - \Delta t\nabla p^{n+1} && (3.8) \\ \nabla \cdot \vec{u}^{n+1} &= 0 && (3.9) \end{aligned}$$

The vector field $\vec{v}$ is a prediction of the velocity field at the next time step which is formed by applying the convection-diffusion equation to the current velocity field. This intermediate field generally will not satisfy the incompressibility constraint, and the role of the pressure is to extract the divergence free component of the intermediate velocity field $\vec{v}$. This step has been viewed as a projection onto a divergence free subspace [10, 11]. In the method we consider, the pressure will not be explicitly formed, and instead a stream function approach will be used to extract $\vec{u}^{n+1}$ from $\vec{v}$. We will create an operator $\tilde{\mathcal{P}}$ that extracts the advanced velocity field from the intermediate field (Note that this operator will not be a true orthogonal projection when inhomogeneous boundary conditions are considered). If,

$$\vec{u}^{n+1} = \vec{v} - \Delta t\nabla p^{n+1} \tag{3.10}$$

where,

$$\begin{aligned} \nabla \cdot \vec{u}^{n+1}(\boldsymbol{x}) &= 0, \quad \boldsymbol{x} \in \Omega^{n+1} && (3.11) \\ \vec{u}^{n+1}(\boldsymbol{x}) &= \vec{g}^{n+1}(\boldsymbol{x}), \quad \boldsymbol{x} \in \partial\Omega^{n+1} \end{aligned}$$

then,

$$\tilde{\mathcal{P}}(\vec{v}, \vec{g}^{n+1}, \Omega^{n+1}) \equiv \vec{u}^{n+1}. \tag{3.12}$$

We observe that the projection is applied on the advanced domain, Ω^{n+1}, which may not coincide with the current domain, Ω^n. Intuitively this is reasonable since the intermediate field is a prediction of the velocity at the advanced time, hence it should be defined on the advanced domain. Practically, we need some mechanism for transferring values from the current domain to an advanced domain, and as we will see in Section 3.5 the use of staggered COGs allows this transfer to be accomplished quite naturally.

3.2 The projection step:

In trying to apply the projection, an important observation is that an incompressible fluid can be represented as the curl of a stream function. (For the following statements, we consider the domain to be bounded with M interior contours and one bounding contour. The unbounded case involves slight modifications.) If

$$\nabla \cdot \vec{u}^{n+1}(\boldsymbol{x}) \;=\; 0, \quad \boldsymbol{x} \in \Omega^{n+1} \tag{3.13}$$

and

$$\int_{\partial\Omega_k^{n+1}} \vec{u}^{n+1} \cdot \hat{\eta} \;=\; 0, \quad k = 1, \ldots, M+1 \tag{3.14}$$

then there exists,

$$\Psi(\boldsymbol{x}) \;\equiv\; \int_{\boldsymbol{x}_o}^{\boldsymbol{x}} \vec{u}^{n+1} \cdot \hat{\eta} \tag{3.15}$$

where

$$\vec{u}^{n+1} \;=\; \nabla\times < 0, 0, \Psi > . \tag{3.16}$$

Here, $\boldsymbol{x}_o$ represents an arbitrary point in the domain, and this means that the stream function is only determined up to an additive constant. The above result can be verified by using Green's theorem and then differentiating, and it shows that if the stream function is known, then we can obtain the desired divergence free velocity field $\vec{u}^{n+1}$ by applying the curl. The question now becomes how to determine the stream function, and we begin by taking the curl of the intermediate velocity field. If,

$$\vec{v} \;=\; \vec{u}^{n+1} + \Delta t \nabla p^{n+1} \tag{3.17}$$

then,

$$\vec{v} \;=\; \nabla\times <0,0,\Psi> +\Delta t\nabla p^{n+1}, \tag{3.18}$$

and

$$\nabla\times\vec{v} \;=\; -\Delta\Psi+0. \tag{3.19}$$

If we define the vorticity, $\vec{\omega}=<0,0,\omega>=\nabla\times\vec{v}=\nabla\times\vec{u}^{n+1}$, then we see that the stream function satisfies a Poisson equation.

$$\Delta\Psi(\boldsymbol{x})=-\omega(\boldsymbol{x}),\quad \boldsymbol{x}\in\Omega^{n+1} \tag{3.20}$$

In order to solve this problem, we need to determine a boundary condition for Ψ. We can directly verify that there is a simple relationship between the components of the velocity and the derivatives of the stream function.

$$\frac{\partial\Psi}{\partial\tau}=\vec{u}^{n+1}\cdot\hat{\eta} \tag{3.21}$$

$$\frac{\partial\Psi}{\partial\eta}=\vec{u}^{n+1}\cdot\hat{\tau} \tag{3.22}$$

From this we see that the no-flow condition determines the boundary values of the stream function up to a constant per contour.

$$\Psi(\boldsymbol{x}) \;=\; \Psi_k+\int_{\boldsymbol{x}_k}^{\boldsymbol{x}}\frac{\partial\Psi}{\partial\tau};\quad \boldsymbol{x}_k,\boldsymbol{x}\in\partial\Omega_k^{n+1} \tag{3.23}$$

$$\;=\; \Psi_k+\int_{\boldsymbol{x}_k}^{\boldsymbol{x}}\vec{g}^{n+1}\cdot\hat{\eta} \tag{3.24}$$

If the domain is simply connected, then there is only one constant and it can be set to zero since the stream function is only determined up to an additive constant. However, in a multiply connected domain these constants must be chosen to satisfy the local circulations (which are specified by the no-slip conditions). That is, we set $\Psi_{M+1}=0$ and then choose the remaining M constants to satisfy the M constraints:

$$\int_{\partial\Omega_k^{n+1}}\frac{\partial\Psi}{\partial\eta}=\int_{\partial\Omega_k^{n+1}}\vec{g}^{n+1}\cdot\hat{\tau},\quad k=1,\ldots,M.$$

This will specify the stream function up to an additive constant, and hence will produce the divergence free velocity field that satisfies the needed boundary conditions. A two step process is used to choose these constants, where we first measure the effects of setting the constants to zero. Solve,

$$\Delta\Psi^0(\boldsymbol{x}) \;=\; -\omega(\boldsymbol{x}),\quad \boldsymbol{x}\in\Omega^{n+1} \tag{3.25}$$

$$\Psi^0(\boldsymbol{x}) \;=\; \int_{\boldsymbol{x}_k}^{\boldsymbol{x}}\vec{g}^{n+1}\cdot\hat{\eta};\quad \boldsymbol{x}_k,\boldsymbol{x}\in\partial\Omega_k^{n+1}. \tag{3.26}$$

Computationally this is a challenging problem to solve both because of the complicated geometry and the large amount of computation that elliptic problems typically require. To address this problem we developed a rapid Poisson solver (see Section 2.3), where we discretize the problem on a COG and solve the resulting discrete equations using an iterative method with an integral-based preconditioner. The resulting stream function solution produces an incompressible velocity field that satisfies the no-flow boundary condition, but it may not yield the correct local circulations. (In general, $\int_{\partial\Omega_k^{n+1}} \frac{\partial\Psi^0}{\partial\eta} \neq \int_{\partial\Omega_k^{n+1}} \vec{g}^{n+1} \cdot \hat{\tau}$). Therefore, a correction term is sought to produce the desired circulation behavior: Obtain Ψ^1 and $\{\Psi_k\}$ so that,

$$\begin{aligned} \Delta\Psi^1(\boldsymbol{x}) &= 0, \quad \boldsymbol{x} \in \Omega^{n+1} \\ \Psi^1(\boldsymbol{x}) &= \Psi_k, \quad \boldsymbol{x} \in \partial\Omega_k^{n+1} \\ \int_{\partial\Omega_k^{n+1}} \frac{\partial\Psi^1}{\partial\eta} &= \int_{\partial\Omega_k^{n+1}} \vec{g}^{n+1} \cdot \hat{\tau} - \int_{\partial\Omega_k^{n+1}} \frac{\partial\Psi^0}{\partial\eta}, \quad \forall k. \end{aligned} \tag{3.27}$$

This problem has been solved previously [35] at the cost of one elliptic problem for each boundary component, but this approach becomes overly expensive when there are many boundary components. To overcome this difficulty, we developed an integral formulation (see Section 4.4) where the correction term is obtained with a single elliptic solve regardless of the number of boundary components. This formulation allows us to maintain computational efficiency even when simulating flows with a large number of bodies in the flow field. By combining Ψ^0 and Ψ^1 we obtain a stream function that satisfies the no-flow boundary condition and produces the correct local circulations and vorticity: If

$$\Psi \equiv \Psi^0 + \Psi^1 \tag{3.28}$$

then,

$$\begin{aligned} \Delta\Psi(\boldsymbol{x}) &= -\omega(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega^{n+1} \\ \frac{\partial\Psi}{\partial\tau}(\boldsymbol{x}) &= \vec{g}^{n+1} \cdot \hat{\eta}, \quad \boldsymbol{x} \in \partial\Omega^{n+1} \\ \int_{\partial\Omega_k^{n+1}} \frac{\partial\Psi}{\partial\eta} &= \int_{\partial\Omega_k^{n+1}} \vec{g}^{n+1} \cdot \hat{\tau}, \quad \forall k \end{aligned} \tag{3.29}$$

$$\Rightarrow \vec{u}^{n+1} = \nabla\times < 0, 0, \Psi > . \tag{3.30}$$

To recap the projection step, we are given the intermediate velocity field $\vec{v}$, which is defined on the domain Ω^{n+1}. We calculate the vorticity ω, and use it (plus the velocity boundary conditions) to obtain the stream function of the divergence free, advanced velocity field (Eqs. 3.25-3.28). Finally, applying the curl to the stream function recovers the velocity field, $\vec{u}^{n+1}$. We denote this three stage procedure by the symbol, $\tilde{\mathcal{P}}$, where $\vec{u}^{n+1} = \tilde{\mathcal{P}}(\vec{v}, \vec{g}^{n+1}, \Omega^{n+1})$.

3.3 The algorithm:

In the preceding algorithm, we used forward Euler to approximate the time derivative. This simple choice makes it easier to explain how the method works; however, it can lead to restrictive time steps depending on the relative magnitude of the convective and diffusive terms. This restriction stems from the size and shape of the absolute stability region of forward Euler, and larger time steps can be taken if a fourth-order Runge-Kutta [21] scheme is used to approximate the time derivative. The Runge-Kutta scheme can be viewed as a collection of Euler-like steps, therefore, the full algorithm consists of multiple applications of the algorithm we have already considered. To describe the full algorithm more concisely, we introduce some notation to represent the convection-diffusion increment.

$$\mathcal{F}(\vec{u}) \equiv \Delta t(-(\vec{u}\cdot\nabla)\vec{u} + \nu\Delta\vec{u}) \tag{3.31}$$

Given the velocity at the current time $\vec{u}^n$, we obtain the velocity at an advanced time by applying fourth-order Runge-Kutta time stepping and enforcing the incompressibility constraint at each stage.

$$\begin{aligned}
\vec{F}_1 &= \mathcal{F}(\vec{u}^n) \\
\vec{F}_2 &= \mathcal{F}(\tilde{\mathcal{P}}(\vec{u}^n + \frac{1}{2}\vec{F}_1, \vec{g}^{n+\frac{1}{2}}, \Omega^{n+\frac{1}{2}})) \\
\vec{F}_3 &= \mathcal{F}(\tilde{\mathcal{P}}(\vec{u}^n + \frac{1}{2}\vec{F}_2, \vec{g}^{n+\frac{1}{2}}, \Omega^{n+\frac{1}{2}})) \\
\vec{F}_4 &= \mathcal{F}(\tilde{\mathcal{P}}(\vec{u}^n + \vec{F}_3, \vec{g}^{n+1}, \Omega^{n+1})) \\
\vec{u}^{n+1} &= \tilde{\mathcal{P}}(\vec{u}^n + \frac{1}{6}(\vec{F}_1 + \vec{F}_2 + \vec{F}_3 + \vec{F}_4), \vec{g}^{n+1}, \Omega^{n+1})
\end{aligned} \tag{3.32}$$

This is the algorithm we will use in the fluid simulations we consider, but in order to implement it we must decide how to compute the needed spatial derivatives.

3.4 Spatial discretization:

In order to approximate the spatial derivatives in the Navier-Stokes solver we have devised, finite differences are applied on a COG discretization of the domain. The majority of our approximations consist of standard, centered differences, like those given in [30]. These difference approximations are used to advance the convection-diffusion equation, form the advanced vorticity, solve Poisson's equation, and to recover the advanced velocity by applying the curl to the stream function. One reason to use COGs is that they can be formed very quickly, and this is important for problems with time-dependent geometry. COGs also make it easy to transfer information between different time steps,

and at grid points that enter the domain (as we advance in time) we have values defined though interpolation. Finally, a third benefit is that we avoid the small cell stability problem that one might expect from a method that employs Cartesian grids.

Given the domain, Ω^n, at the current time, we form a staggered mesh consisting of three offset COGs. One grid is used to mark the points where the stream function and vorticity are known, and the other two grids represent the x and y components of the fluid velocity. As described in Section 2.3 values are given on the boundary and at all regular interior grid points, while the values at irregular points are defined through interpolation. If we represent the stream function by Ψ, the vorticity by ω, and the two velocity components as $\vec{u} =< u, v, 0 >$, then the staggered grid can be depicted as in Fig. 3.1

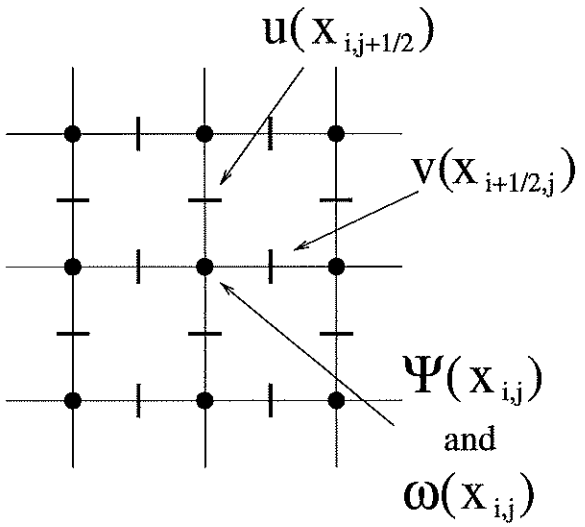

Figure 3.1: The staggered COG for the fluid variables.

To advance the fluid velocity, we use the time discretized convection-diffusion equation (Eq. 3.7) where a finite difference approximation is applied to the convection and diffusion terms.

$$\begin{aligned}
(\vec{u}\cdot\nabla)u(\boldsymbol{x}_{i,j+1/2}) &\approx u(\boldsymbol{x}_{i,j+1/2})D_h^x u(\boldsymbol{x}_{i,j+1/2}) + v(\boldsymbol{x}_{i,j+1/2})D_h^y u(\boldsymbol{x}_{i,j+1/2}) \\
(\vec{u}\cdot\nabla)v(\boldsymbol{x}_{i+1/2,j}) &\approx u(\boldsymbol{x}_{i+1/2,j})D_h^x v(\boldsymbol{x}_{i+1/2,j}) + v(\boldsymbol{x}_{i+1/2,j})D_h^y v(\boldsymbol{x}_{i+1/2,j}) \\
\Delta u(\boldsymbol{x}_{i,j+1/2}) &\approx \Delta_h u(\boldsymbol{x}_{i,j+1/2}) \\
\Delta v(\boldsymbol{x}_{i+1/2,j}) &\approx \Delta_h v(\boldsymbol{x}_{i+1/2,j})
\end{aligned} \tag{3.33}$$

We advance the velocity at all grid points in the domain, and the finite differences (D_h^x, D_h^y, and Δ_h) use a stencil appropriate to the type (regular or irregular) of grid point it is acting on. For example consider differencing the y

component of velocity at the grid point $\boldsymbol{x}_{i+1/2,j}$. We denote $\boldsymbol{x}_L, \boldsymbol{x}_R, \boldsymbol{x}_D$, and $\boldsymbol{x}_U$ as the nearest points (to the left, right, down, and up) where values of v are known (either at a grid point or a boundary point), and we denote d_L, d_R, d_D, and d_U as the corresponding stencil arm lengths (see Fig. 3.2). With this

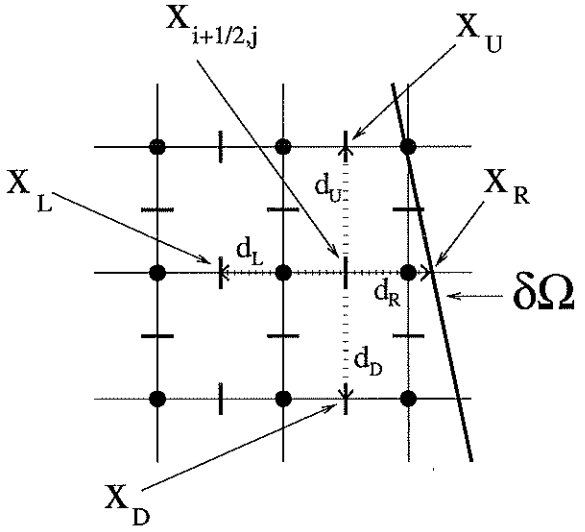

Figure 3.2: Differencing at a Y velocity point.

notation, the following finite differences are used:

$$
\begin{aligned}
D_h^x v(\boldsymbol{x}_{i+1/2,j}) &= \frac{\frac{v(\boldsymbol{x}_R)-v(\boldsymbol{x}_{i+1/2,j})}{d_R} d_L + \frac{v(\boldsymbol{x}_{i+1/2,j})-v(\boldsymbol{x}_L)}{d_L} d_R}{d_L + d_R} \\
D_h^y v(\boldsymbol{x}_{i+1/2,j}) &= \frac{\frac{v(\boldsymbol{x}_U)-v(\boldsymbol{x}_{i+1/2,j})}{d_U} d_D + \frac{v(\boldsymbol{x}_{i+1/2,j})-v(\boldsymbol{x}_D)}{d_D} d_U}{d_D + d_U} \qquad (3.34) \\
\Delta_h v(\boldsymbol{x}_{i,j+1/2}) &= \frac{\frac{v(\boldsymbol{x}_R)-v(\boldsymbol{x}_{i+1/2,j})}{d_R} - \frac{v(\boldsymbol{x}_{i+1/2,j})-v(\boldsymbol{x}_L)}{d_L}}{\frac{1}{2}(d_L + d_R)} \\
&\quad + \frac{\frac{v(\boldsymbol{x}_U)-v(\boldsymbol{x}_{i+1/2,j})}{d_U} - \frac{v(\boldsymbol{x}_{i+1/2,j})-v(\boldsymbol{x}_D)}{d_D}}{\frac{1}{2}(d_D + d_U)}
\end{aligned}
$$

Since these approximations are applied at all grid points in the domain, it would appear that they might lead to stability problems. However, in our projection step, we never use information that is generated by stencil arms that are smaller than half a mesh width (a fourth a mesh width in the moving boundary case), therefore we never encounter any arbitrarily small stencil stability problems. This is explained further in Section 3.6.

The convective terms couple the two components of velocity (which are defined on separate, offset grids), and in order to obtain a value for $u(\boldsymbol{x}_{i+1/2,j})$, we fit a bilinear interpolant through the known values at the appropriate grid and boundary points. If we use the notation shown in Fig. 3.3 , we can write down an equation for this interpolation.

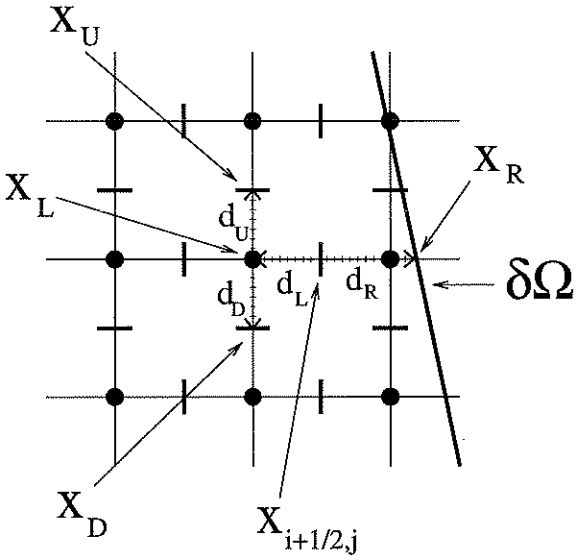

Figure 3.3: Interpolation of X velocity values at a Y velocity grid point.

$$u(\boldsymbol{x}_L) = \frac{u(\boldsymbol{x}_U)d_D + u(\boldsymbol{x}_D)d_U}{d_D + d_U} \tag{3.35}$$

$$u(\boldsymbol{x}_{i+1/2,j}) = \frac{u(\boldsymbol{x}_R)d_L + u(\boldsymbol{x}_L)d_R}{d_L + d_R} \tag{3.36}$$

We note that for points away from the boundary, the given approximate derivatives reduce to the standard centered finite differences, and the bilinear interpolation becomes equivalent to averaging the four nearest values ([30] pps. 146-147). The formulas for the x component of velocity are completely analogous.

Once the convection and diffusion terms have been used to advance the velocity field to an intermediate state, $\vec{v} =< v_1, v_2, 0 >$, the projection step is applied. As described in Eqs. 3.17- 3.19 the first step is to compute the vorticity by talking the curl of the intermediate velocity. We apply the curl by using centered finite differences.

$$\omega(\boldsymbol{x}_{i,j}) \quad = \quad (\nabla \times \vec{v})(\boldsymbol{x}_{i,j})$$

$$
\begin{aligned}
(\nabla \times \vec{v})(\boldsymbol{x}_{i,j}) &\approx (\nabla \times_h \vec{v})(\boldsymbol{x}_{i,j}) \\
&= D_h^x v_2(\boldsymbol{x}_{i,j}) - D_h^y v_1(\boldsymbol{x}_{i,j}) \qquad (3.37) \\
D_h^x v_2(\boldsymbol{x}_{i,j}) &= \frac{v_2(\boldsymbol{x}_{i+1/2,j}) - v_2(\boldsymbol{x}_{i-1/2,j})}{h} \\
D_h^y v_1(\boldsymbol{x}_{i,j}) &= \frac{v_1(\boldsymbol{x}_{i,j+1/2}) - v_1(\boldsymbol{x}_{i,j-1/2})}{h}
\end{aligned}
$$

When we enforce the incompressibility constraint, the vorticity becomes an inhomogeneous forcing term for Poisson's equation defined on a COG discretization of the domain at the advanced time. The spatial derivatives in the Poisson problem are approximated using the standard five point discrete Laplacian, as discussed in Section 2.3. Representing Poisson's equation on a COG has several consequences: First, the vorticity is only needed at regular points, and as we will explain in Section 3.5 this lets us model moving boundaries. Second, we can apply the fast techniques developed in Section 2.3 to solve the resulting discrete equations, and finally we avoid the small cell stability problem that a plain Cartesian grid method would encounter. This stability improvement may seem surprising, so we will analyze it in more detail in Section 3.6.

After the incompressibility constraint has been applied, the projected velocity field is recovered by taking the curl of the stream function. The curl is only applied at regular points, so once again the partial derivatives are approximated with simple, centered differences.

$$
\begin{aligned}
\vec{u}^{n+1} &= \nabla \times \Psi \\
&\approx < D_h^y \Psi, -D_h^x \Psi, 0 > \qquad (3.38) \\
u^{n+1}(\boldsymbol{x}_{i,j+1/2}) &= \frac{\Psi(\boldsymbol{x}_{i,j+1}) - \Psi(\boldsymbol{x}_{i,j})}{h} \\
v^{n+1}(\boldsymbol{x}_{i+1/2,j}) &= \frac{\Psi(\boldsymbol{x}_{i+1,j}) - \Psi(\boldsymbol{x}_{i,j})}{h}
\end{aligned}
$$

This determines the velocity at the regular points, and since the velocity is represented on a COG, the values at the irregular points are filled in with a third-order interpolant. At this stage, the advanced velocity is known at all grid points, and the next time step can be taken.

3.5 A moving boundary issue:

At first glance, moving boundaries appear to cause difficulties in our Navier-Stokes solver. In our algorithm (Eq. 3.32) we form the intermediate velocity field on the current domain, Ω^n, while the projection is applied to this velocity on the advanced domain, $\Omega^{n+1/2}$ or Ω^{n+1}. For time-dependent geometry these two domains will not coincide, so it seems that there will be a problem in transferring

information from one domain to the other. However, if we choose time steps so that the boundary does not move too far in any one step, then it turns out that our algorithm can handle moving boundaries without modifications. This feature is a result of the use of staggered COGs: First, recall that the intermediate velocity is calculated for both regular and irregular points of Ω^n. However, in the projection step (due to the use of COGs) we only need the vorticity at the regular points of Ω^{n+1} (which by definition are at least a full mesh width away from $\partial\Omega^{n+1}$). We obtain these vorticity values by taking the curl of the intermediate velocity, and (due to use of staggered grids) each vorticity value is formed from intermediate velocity values that lie only one half a mesh width from the regular vorticity point (Eq. 3.37). Therefore, we only need intermediate velocity values at points at least one half a mesh width away from $\partial\Omega^{n+1}$, and if the boundary moves less that half a mesh width, then these intermediate values must be regular or irregular points of Ω^n (hence are known, see Fig. 3.4). Thus moving boundaries can be modeled with exactly the same

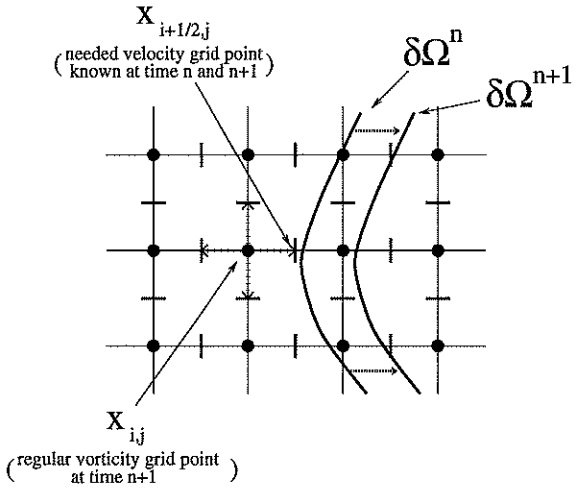

Figure 3.4: Transfer of information between the non-overlapping domains resulting from a moving boundary.

algorithm as long as reasonable time steps are chosen. In the next section we will see that stability concerns impose a more severe time step restriction on our algorithm than the non-overlapping domains do.

3.6 A stability issue:

In forming the intermediate field, we computed the finite differences at irregular grid points using non-standard stencils (Eq. 3.34), where some stencil arms were shorter than the mesh width of the Cartesian grid. Generally these non-standard stencils imply that at the irregular points, information cannot travel as far in one time step as at the regular points. This adversely affects the stability of the method, and forces us to use a smaller time step (this is referred to as the small cell stability limit). Even worse, since the boundary can lie anywhere on the Cartesian grid there is no limit to how short a stencil arm could be, and for some domains an arbitrarily small time step would be needed. In our COG formulation, since the vorticity is only computed at the regular points we never need to use information that was obtained from a very small stencil arm. For example, consider the time-independent domain shown in Fig. 3.5 where the vorticity at the regular point $\boldsymbol{x}_{i,j}$ is computed using the y component of the

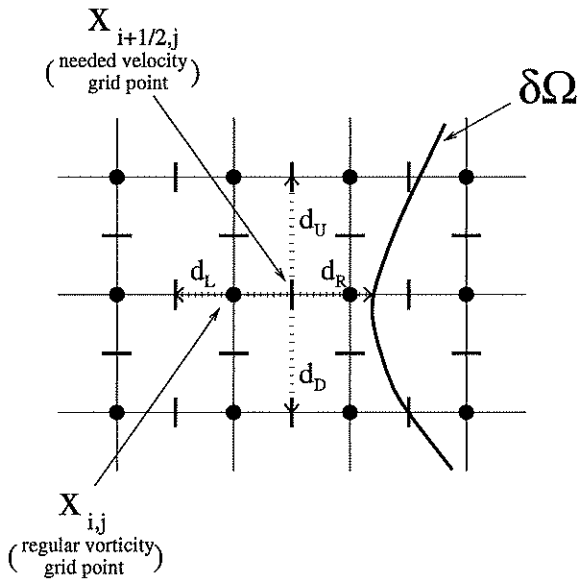

Figure 3.5: Avoidance of arbitrarily small stencil arms (Static boundary: Part 1, note that $d_R > 1/2$ a mesh width).

velocity field at the irregular point $\boldsymbol{x}_{i+1/2,j}$. Let the left, right, down, and up stencil arms at $\boldsymbol{x}_{i+1/2,j}$ have lengths denoted by d_L, d_R, d_D, d_U, where d_R is less that a full mesh width long but greater than half a mesh width. If we changed the geometry so that d_R was less than half a mesh width (see Fig. 3.6) then we see that $\boldsymbol{x}_{i,j}$ would no longer be a regular point of the vorticity, and so we would not use the intermediate velocity component generated with this small

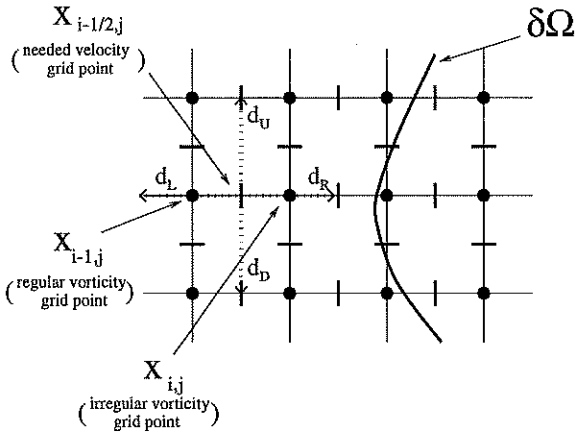

Figure 3.6: Avoidance of arbitrarily small stencil arms (Static boundary: Part 2, note that $d_R > 1/2$ a mesh width).

stencil arm. When moving geometry is considered, we see a similar behavior if we choose the time step so that the boundary moves less than one-fourth a mesh width per time step. In this case we will never use information that is generated with a stencil arm shorter than one-fourth a mesh width (see Fig 3.7).

In summary, using a staggered COG representation will prevent the use of arbitrarily small cells for any given domain (including cases with moving geometry), and therefore avoid the arbitrarily small time step restriction that would otherwise occur.

3.7 Computational results:

3.7.1 A moving cylinder in a fixed box:

In our first example, we consider an infinite rectangular tube containing an infinite cylinder. The tube is filled with a fluid with kinematic viscosity coefficient 0.0025. The cross section is represented by a unit square (with (0,0) and (1,1) being the minimum and maximum coordinates), containing a circle of radius 0.09. At time t=0, the cylinder is centered at (0.25,0.5), and the fluid is at rest. The cylinder moves horizontally with a velocity $\frac{3(1-(t-1)^2)}{8}$, which causes the cylinder to accelerate for one second and then decelerate by the same amount, bringing the cylinder to a rest at the point (0.75,0.5). In Fig 3.8 and 3.9, we observe the results of the simulation when computed on a 40x40 grid with 40 points placed on each boundary component. A uniform time step of 0.009 was

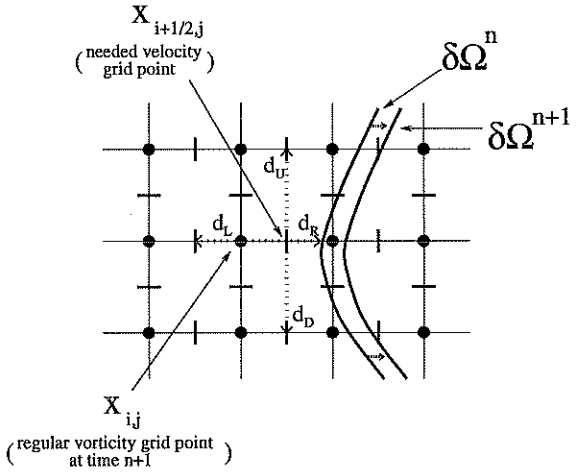

Figure 3.7: Avoidance of arbitrarily small stencil arms (Moving boundary, note that $d_R > 1/4$ a mesh width).

used, which satisfies the stability restriction throughout the simulation. The stream function and vorticity are shown up to t=2.0 in increments of 0.5 seconds. Since the cylinder is contained within the tube, the boundary of the domain undergoes relative motion, and therefore this flow cannot be modeled using a static boundary in a moving reference frame. We observe that as the cylinder moves, the viscous effects generate vorticity at the boundary, and a small wake is formed.

As a check of the long term convergence behavior, we consider the moving cylinder problem for multiple grid refinements. We prescribe the same cylinder velocity and the same initial conditions as before, but in order to allow the coarse grid to resolve the flow, we increase the kinematic viscosity to 0.01. We run our simulation up to time 1.0 on a 20x20, 40x40, and 80x80 grid, with a uniform time step of 0.000225. The resulting stream functions are shown in Fig. 3.10, and the vorticities are shown in Fig. 3.11. We see that the solution on the 20x20 grid is noticeably different from the 40x40 solution, however the 40x40 solution nearly matches the 80x80 result. This shows that the solutions are converging as the refinement increases. The stream function seems to be fully converged on the 40x40 grid, while the vorticity still seems to be changing slightly as we go from the 40x40 to the 80x80 grid. This behavior is not surprising since the stream function is an integrated quantity (with respect to the velocity), while the vorticity is a differentiated one.

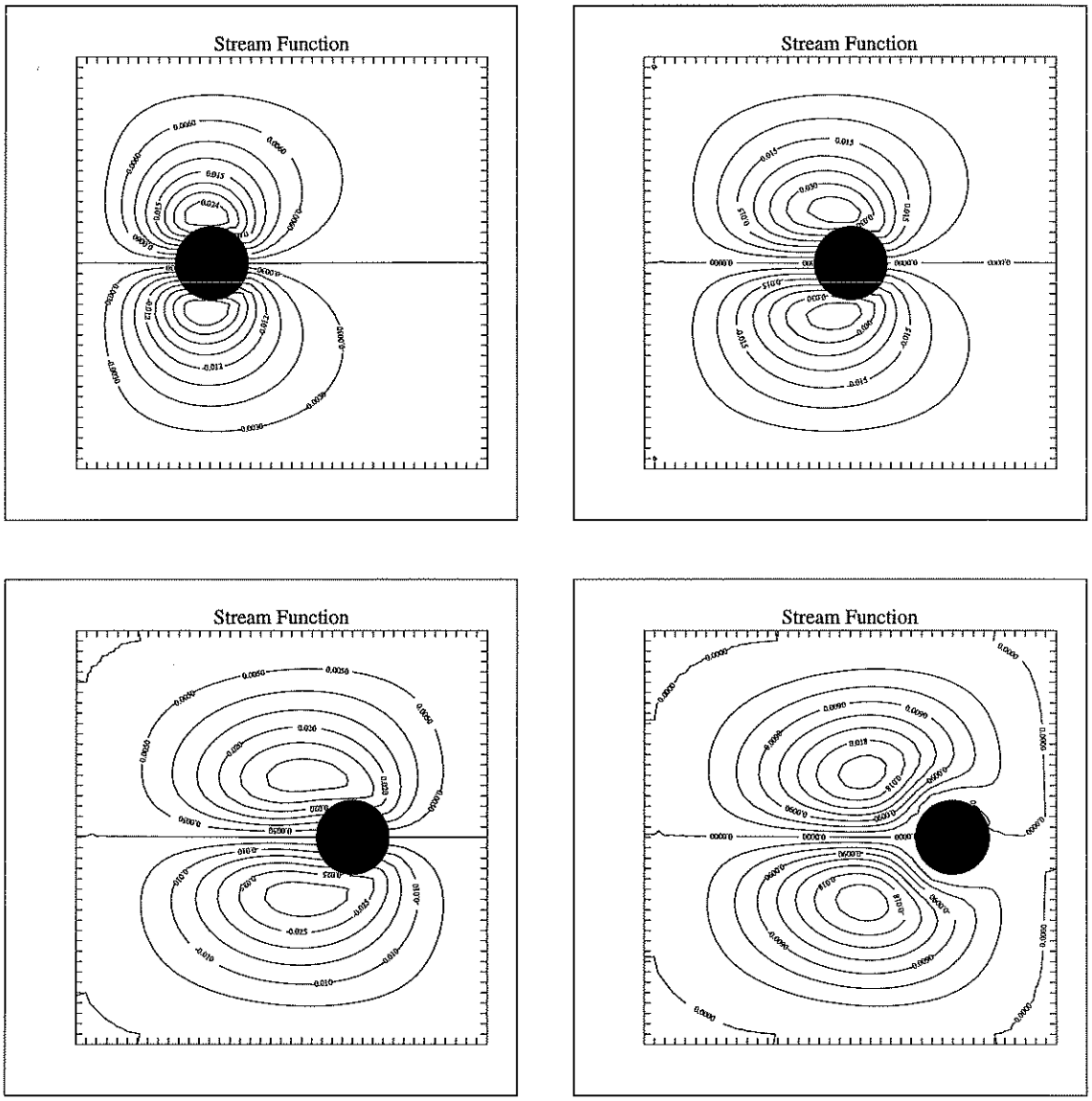

Figure 3.8: Stream function of a moving cylinder in a fixed box.

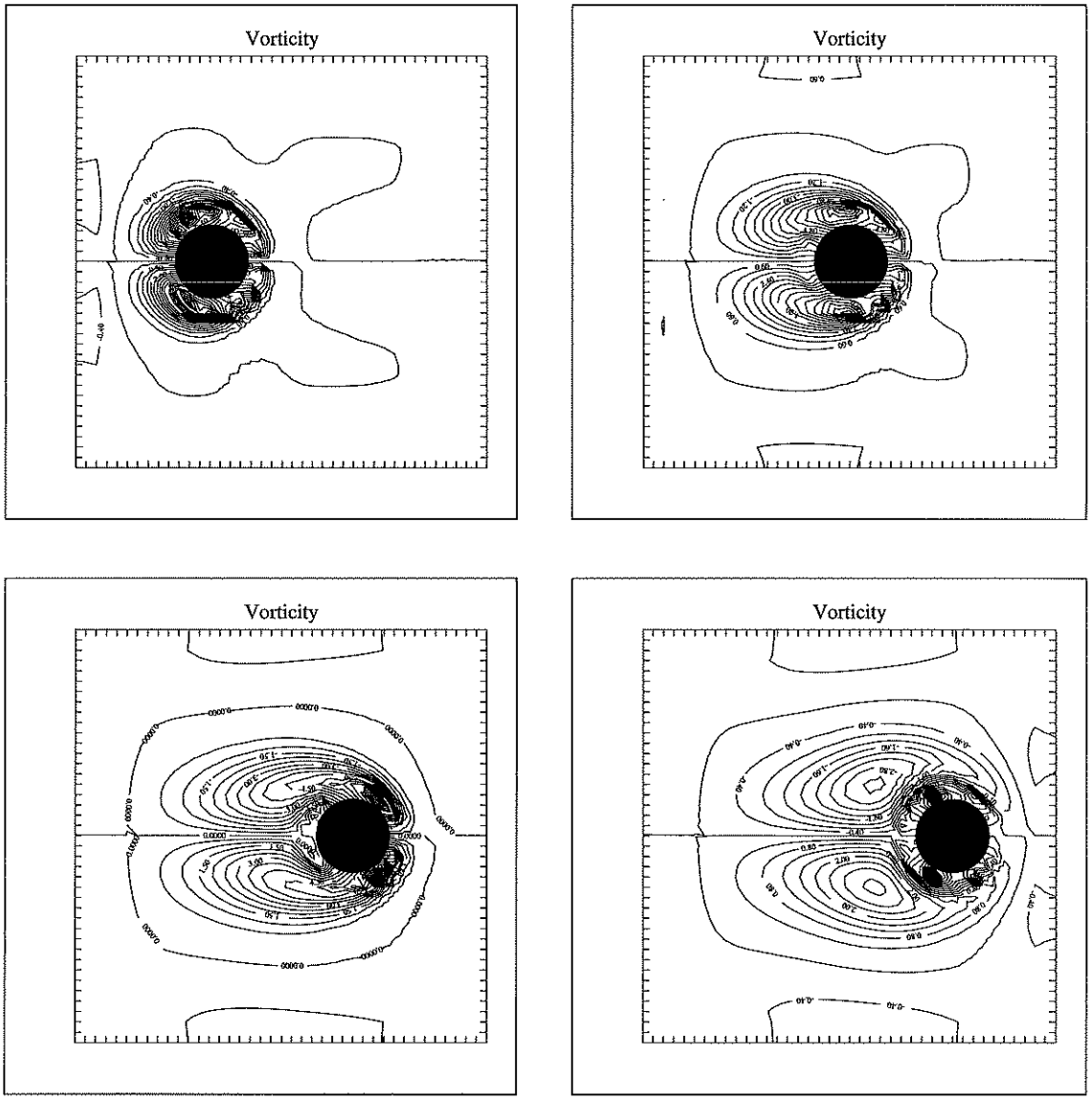

Figure 3.9: Vorticity of a moving cylinder in a fixed box.

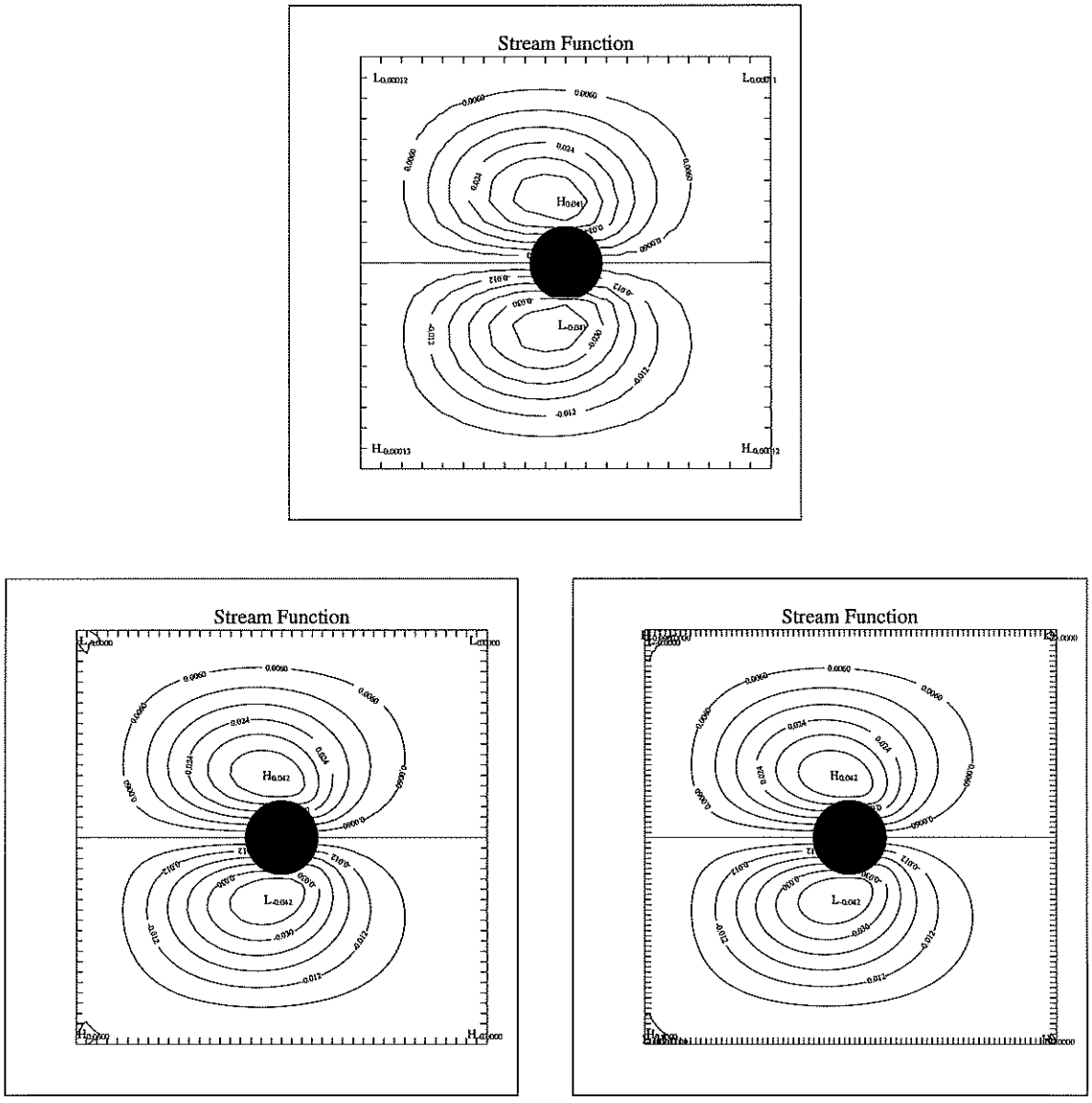

Figure 3.10: Stream function(at t= 1.0), calculated using three grid refinements: 20x20, 40x40, and 80x80.

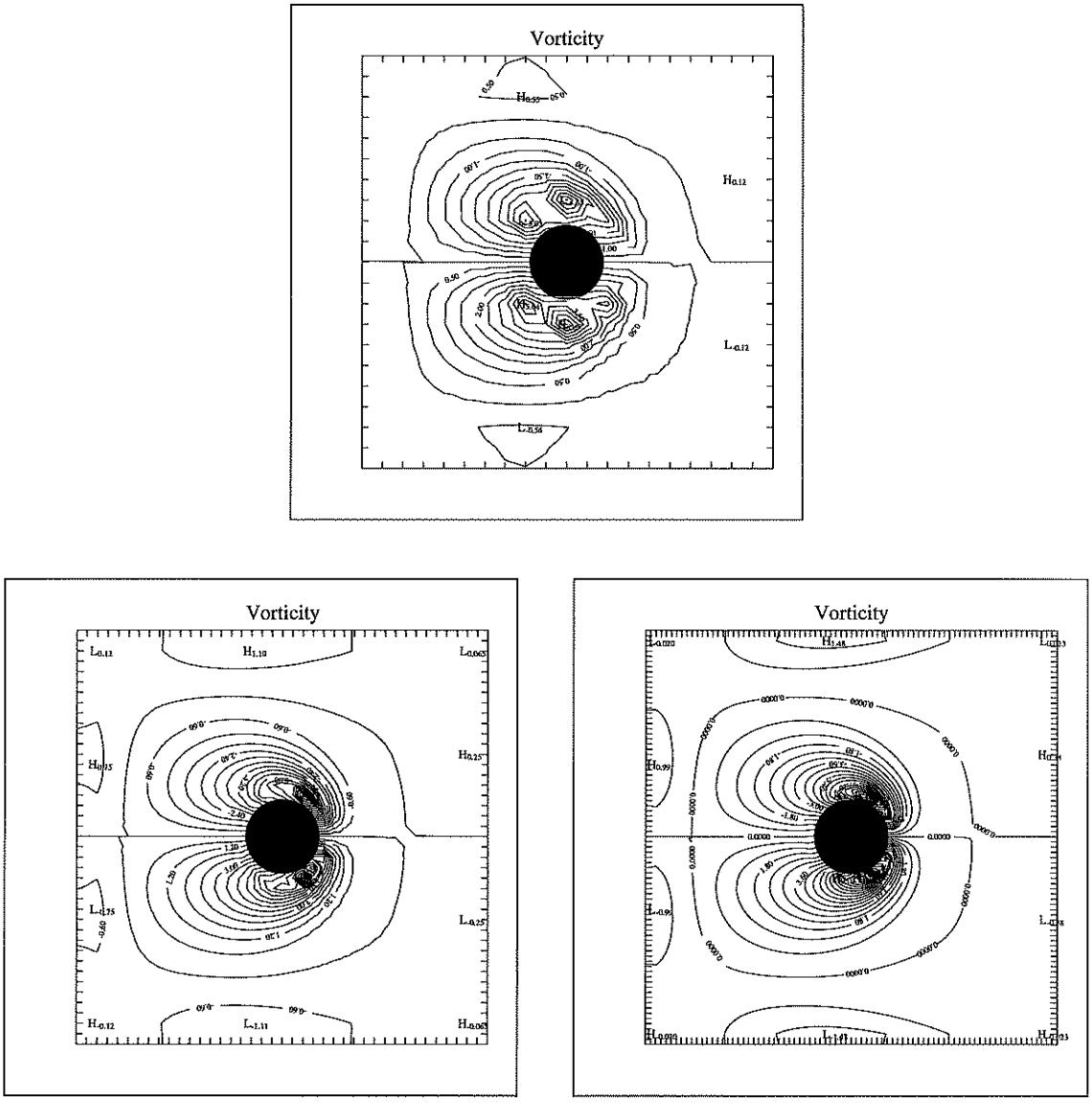

Figure 3.11: Vorticity(at t= 1.0), calculated using three grid refinements: 20x20, 40x40, and 80x80.

3.7.2 A convergence study:

In a quantitative check of the convergence of the method, we ran the moving cylinder simulation on multiple grid refinements (20x20, 40x40, and 80x80), and measured the L2 norm of the relative error (of the stream function, velocity, and vorticity) between successive grids. We used a kinematic vorticity of 0.01, and advanced the solution up to t=0.09 using a uniform time step of 0.0009. Results are summarized in Table. 3.1. (Note: staggered velocity grid points do not coincide when the grids are refined. Therefore we averaged the values of the two bordering points to compare values on different grids)

Table 3.1: Convergence study of fluid solver: t=0.09

	$\|E_{h,\frac{h}{2}}\|_2$	$\|E_{\frac{h}{2},\frac{h}{4}}\|_2$	Order
Stream fnc	0.000351	3.69e-05	3.3
X velocity	0.002385	0.000596	2.0
Y velocity	0.002627	0.000526	2.3
vorticity	0.121288	0.039931	1.6

3.7.3 Flow past a staggered array of cylinders:

As a demonstration of the solver's ability to model flow about multiple bodies, we consider a flow past an array of cylinders. Eight cylinders of radius 0.09 are placed within the unit box. The box is filled with fluid with a kinematic viscosity coefficient of 0.01, and the fluid is initially at rest. The top and bottom walls of the box are treated as impermeable, fixed boundaries, while a horizontal cross flow is induced by prescribing a fluid velocity at the left and right walls. At both sides, the x component of fluid velocity linearly increases from zero to one in the span of one second. The computation is performed on a 40x40 grid with 30 points placed on each boundary component (eight cylinders and the bounding box). A time step of 0.0037 was used to advance the solution, and Fig. 3.12 shows the stream function after 1/4 a second has passed. This test problem can be considered an idealization of a cooling simulation for a nuclear reactor (where coolant is flowing about an array of control rods), and could be a starting point for modeling cross flow induced vibration.

3.7.4 Stir-up of an infinite cup of coffee:

The next example involves a flow problem where both the moving object and the bounding contour are non rectangular. Consider a coffee cup consisting

of an infinite cylinder. We place an infinite straw inside the fluid (kinematic viscosity of 0.01) filled cup, and stir the coffee with a circular motion. The cup is centered at the origin with a radius of 3.81 cm, and a straw of radius 0.31 cm revolves about the center of the cup at a radius of 2.51 cm. Initially, the coffee is at rest and the straw is centered at (2.51,0). The straw is moved in a counter-clockwise orbit with an angular velocity that linearly increases from rest to two revolutions per second. 40 points are used to sample each of the two boundary components, and the computational domain is embedded in a 60x60 square grid (whose sides are 5.0 cm long). The time step is picked adaptively to satisfy the stability constraint, and ranges from a step of 0.043 when t=0 to a step of 0.0085 when the time approaches 1 second. The stream function for the fluid is plotted in increments of 0.25 seconds, the results are shown in Fig. 3.13. Since the stream function is only defined at interior points of the grid, we attempted to fill in the remaining points smoothly to help the contour plotter. This resulted in artificial streamlines in the exterior region of our images which are not actually present in our solution.

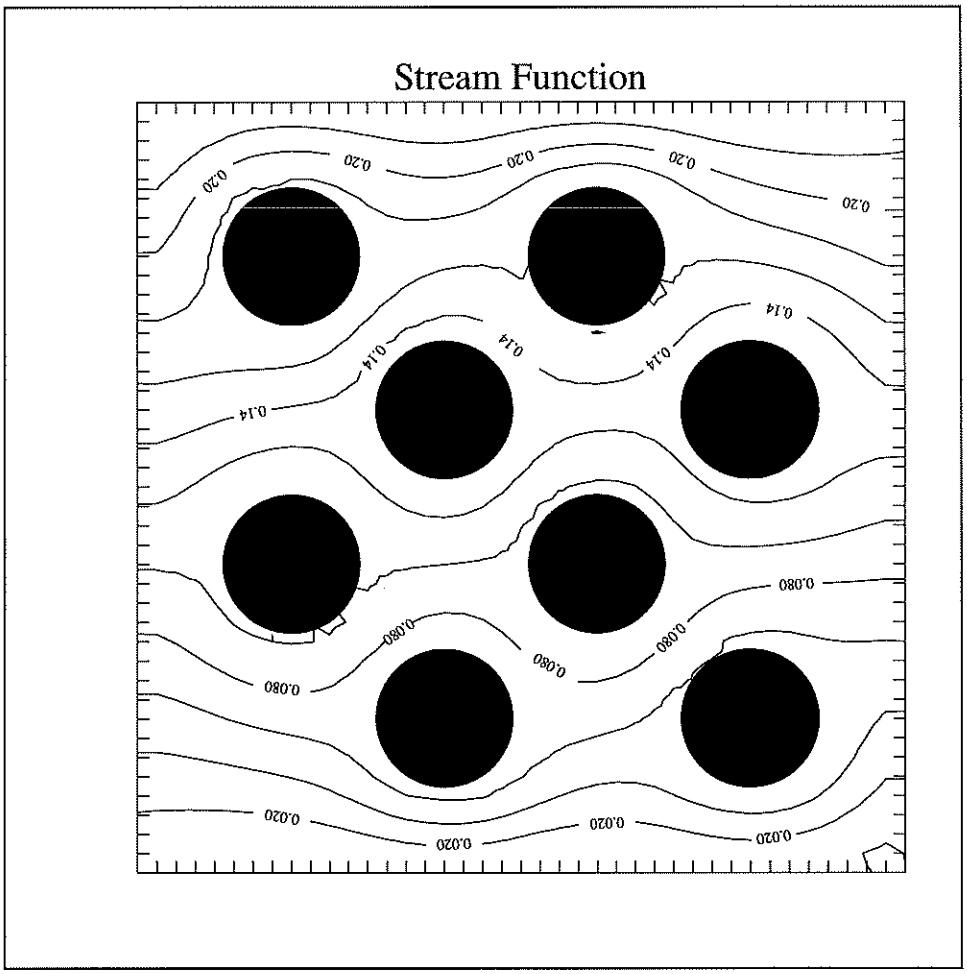

Figure 3.12: Flow past a staggered array of cylinders.

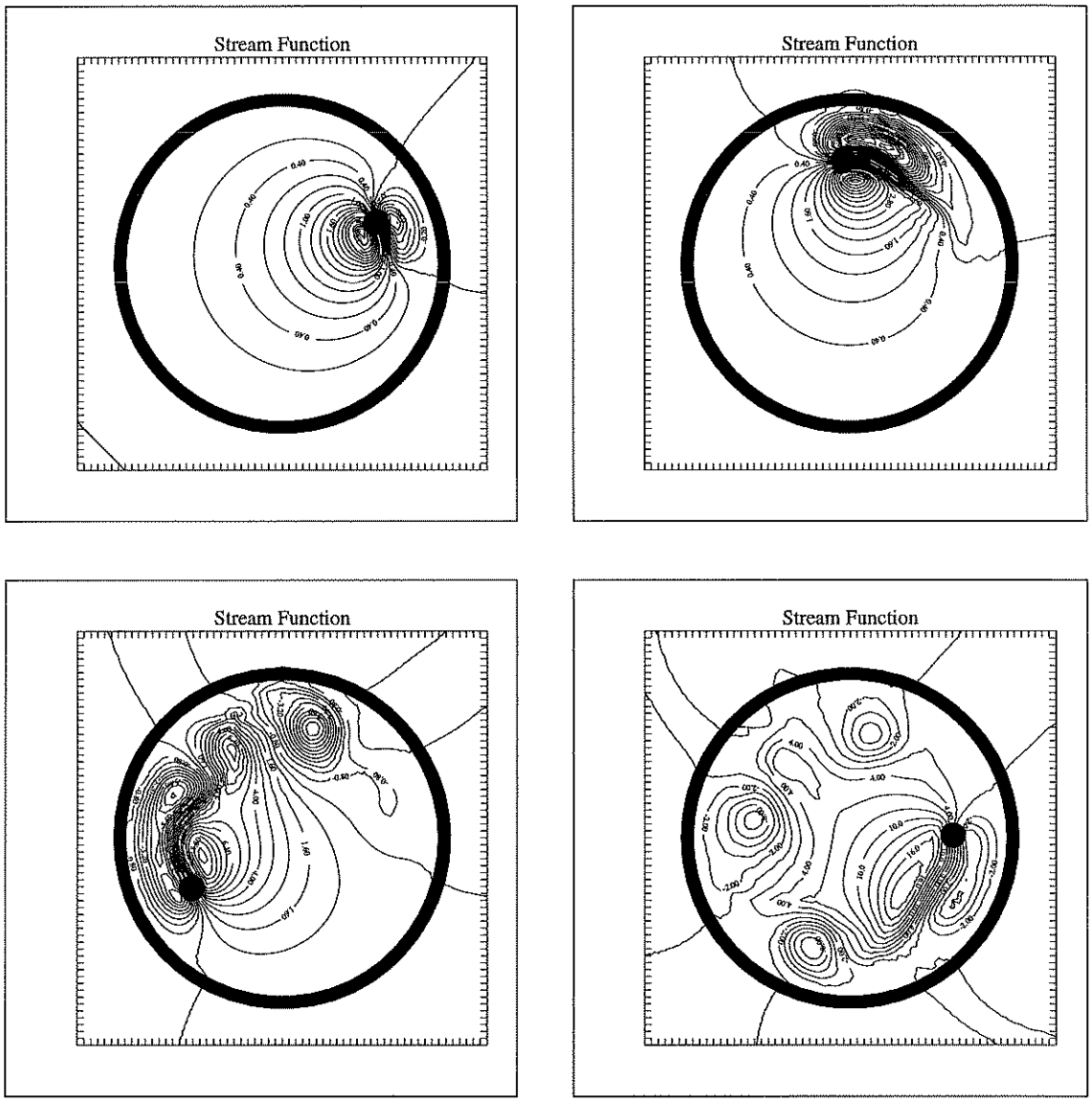

Figure 3.13: Stream function of a coffee cup being stirred.

Chapter 4

Integral Equations:

As we have seen in Chapters 2 and 3, we are interested in integral equations because they can help us deal with complicated geometry. In our fluid solver we apply a projection step that makes use of the stream function. This choice leads us to a uniquely solvable Poisson problem with Dirichlet data determined by the given velocity boundary conditions. In a simple domain it is easy to determine the stream function's boundary values and to solve the resulting Dirichlet problem; however, things become more complicated when more general domains are allowed.

For example, in a simply connected domain, the boundary values of the stream function can be obtained simply by integrating the velocity's no-flow condition (Eq. 3.24). However, in multiply connected domains the no-flow boundary condition only determines the Dirichlet data up to a constant per boundary component, and these constants must be chosen to produce the correct local fluid circulations. In Chapter 3, we showed that this problem can be be reduced to choosing boundary data constants for Laplace's equation subject to constraints on the flux (Eq. 3.27, and in this chapter we show how integral equations can be used to solve this problem efficiently. This problem differs from the typical Laplace problem (where the Dirichlet data is prescribed), and solving it leads to integral equations that are subtly different than the standard integral formulations.

Likewise when solving a Poisson problem in a rectangular domain we can make use of standard fast Poisson solvers, but for our general, multiply connected domains these solvers cannot be directly applied. In the previous chapter (Section 2.3), we created a Poisson solver for complicated domains that combines integral equations with other approaches, and in this chapter (Section 4.2) we review the standard integral techniques that this solver uses.

4.1 Overview:

We begin by representing the solution as a modification of a double layer potential, and this transforms the PDE into a second kind integral equation. Numerically this approach is attractive because it only requires the discretization of the boundary, whereas the entire domain would have to be gridded if a finite difference or finite element method was used. This property makes it easier to solve problems with complex domains, plus it reduces the number of unknowns that must be solved for. The resulting discrete equations reduce to a dense linear system, but this system is well-conditioned and fast techniques [32, 8] exist that enable the system to be solved iteratively in an asymptotically optimal time. Furthermore, fast techniques also exist, [8, 23, 22, 25, 3] for the evaluation of the integral solution at a collection of points. A second feature of the integral equation approach is that useful additional information is automatically obtained in the solution process: specifically the total flux of the solution about each boundary contour can be obtained with no additional calculations. In a fluids context (when solving for the stream function), these fluxes correspond to the circulation about the immersed bodies in the domain. So, integral equations can be an effective way of dealing with problems characterized by complex geometry, especially for incompressible flow about multiple bodies.

A great deal of work has been done on the application of integral equations to Laplace's equation with prescribed boundary data. Theory for simply connected domains can be found in many sources (such as [13, 20, 28, 27]). Multiply connected domains are considered in [13, 28, 27], and a lucid recent presentation of the key points (along with a numerical approach that we adopt) appears in [14]. Although the requisite material is covered in the above references, a summary of needed results is presented for completeness. First we consider the case of a simply connected bounded domain. Then we examine the multiply connected unbounded case, followed by the multiply connected bounded domain. (Note that only two dimensional domains are presented. The three dimensional situation is discussed in some of the above references and is highly analogous to the two dimensional case.) Issues regarding the numerical solution and evaluation are discussed. In the final section, an integral formulation of a different Laplace problem is considered where boundary constants are chosen to satisfy flux constraints. This section plays a key role in efficiently applying the projection step in our fluid solver.

4.2 Integral equation theory for the Dirichlet problem:

4.2.1 Statement of the problem:

Let Ω be a bounded domain in the plane with a C^2 boundary consisting of one bounding contour $\partial\Omega_{M+1}$ and M inner contours $\partial\Omega_1 \cdots \partial\Omega_M$ ($\partial\Omega = \bigcup_{k=1}^{M+1} \partial\Omega_k$, see Fig. 4.1). Note that corners can be allowed (see [20] for some theory and adaptive computations) but we restrict this discussion to smooth contours. Given continuous boundary data g (in [27] only integrability is assumed), solve the following equation:

$$\begin{aligned} \Delta u(\boldsymbol{x}) &= 0, \quad \boldsymbol{x} \in \Omega \\ \lim_{\substack{\boldsymbol{x}\to\boldsymbol{x}_o \\ \boldsymbol{x}\in\Omega}} u(\boldsymbol{x}) &= g(\boldsymbol{x}_o), \quad \boldsymbol{x}_o \in \partial\Omega \end{aligned} \tag{4.1}$$

In the unbounded case, Ω is the unbounded domain that lies exterior to M contours $\partial\Omega_1 \cdots \partial\Omega_M$ ($\partial\Omega = \bigcup_{k=1}^{M} \partial\Omega_k$, see Fig. 4.2), and we seek a solution to

$$\begin{aligned} \Delta u(\boldsymbol{x}) &= 0, \quad \boldsymbol{x} \in \Omega \\ \lim_{\substack{\boldsymbol{x}\to\boldsymbol{x}_o \\ \boldsymbol{x}\in\Omega}} u(\boldsymbol{x}) &= g(\boldsymbol{x}_o), \quad \boldsymbol{x}_o \in \partial\Omega \end{aligned} \tag{4.2}$$

$$u(\boldsymbol{x}) = O(1), \quad (\boldsymbol{x} \to \infty). \tag{4.3}$$

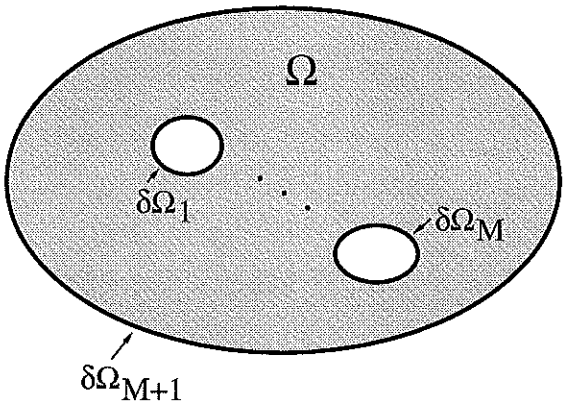

Figure 4.1: A bounded domain.

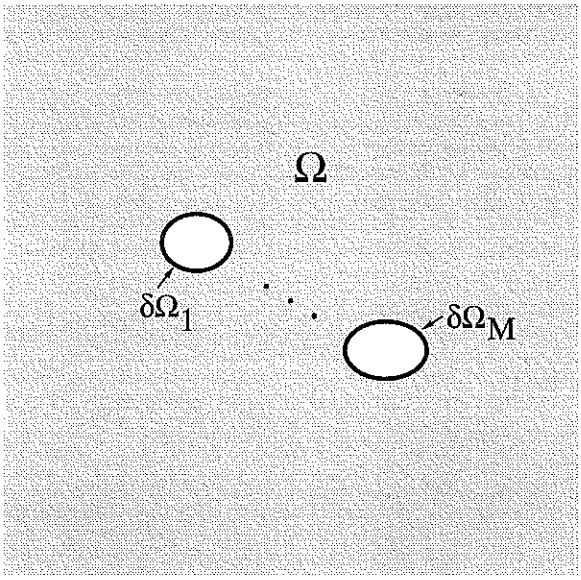

Figure 4.2: An unbounded domain.

4.2.2 Preliminaries:

Any contour $\partial\Omega_k$, considered individually, divides the plane into a bounded and unbounded region. (see Fig. 4.3) On the contour we can define two normal directions at each point: η_{bdd} which points into the bounded region, and η_{unbdd} which points into the unbounded part. Likewise at any boundary point, there is a choice of unit tangents available, and we denote them as the unit tangent pointing in the direction of clockwise traversal τ_{clock}, and the tangent pointing towards counter-clockwise traversal $\tau_{counterclock}$. (see Fig. 4.4) Finally, relative to $\partial\Omega_k$, a point in the plane is denoted as x_{bdd} if it lies in the bounded region contained within $\partial\Omega_k$, x_o if it lies on $\partial\Omega_k$, and x_{unbdd} if it lies in the unbounded region.

A **double layer potential** is defined by placing dipole sources on the boundary:

$$u(\boldsymbol{x}) = \int_{\partial\Omega_k} \phi(\boldsymbol{y}) \frac{\partial}{\partial\eta(\boldsymbol{y})} (\frac{1}{2\pi} \log \mid \boldsymbol{x} - \boldsymbol{y} \mid) \, ds(\boldsymbol{y}) \tag{4.4}$$

Here, η could be either normal. Note that a double layer potential satisfies Laplace's equation away from the boundary (we may differentiate under the integral sign when away from the boundary), so it is reasonable to seek a solution to the Dirichlet problem in the form of a double layer potential. To be able to satisfy the boundary conditions, we ask what the double layer does as we

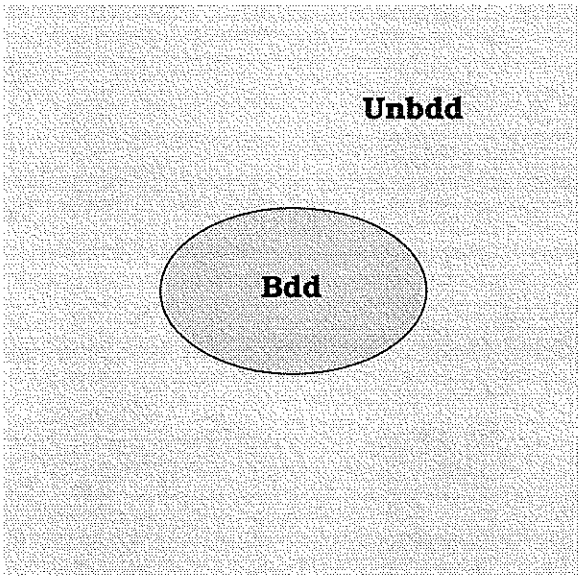

Figure 4.3: Regions relative to a contour.

approach the contour from the bounded and unbounded domains. If $\eta = \eta_{bdd}$, we see the following limiting behavior (see Fig. 4.5):

$$\lim_{\boldsymbol{x}_{bdd} \to \boldsymbol{x}_o} u(\boldsymbol{x_{bdd}}) = -\frac{1}{2}\phi(\boldsymbol{x_o}) + \int_{\partial\Omega} \phi(\boldsymbol{y}) \frac{\partial}{\partial \eta(\boldsymbol{y})} (\frac{1}{2\pi} \log \mid \boldsymbol{x}_o - \boldsymbol{y} \mid) \quad (4.5)$$

$$\lim_{\boldsymbol{x}_{unbdd} \to \boldsymbol{x}_o} u(\boldsymbol{x_{unbdd}}) = \frac{1}{2}\phi(\boldsymbol{x_o}) + \int_{\partial\Omega} \phi(\boldsymbol{y}) \frac{\partial}{\partial \eta(\boldsymbol{y})} (\frac{1}{2\pi} \log \mid \boldsymbol{x}_o - \boldsymbol{y} \mid) \quad (4.6)$$

If $\eta = \eta_{unbdd}$ (Fig. 4.6),

$$\lim_{\boldsymbol{x}_{bdd} \to \boldsymbol{x}_o} u(\boldsymbol{x_{bdd}}) = \frac{1}{2}\phi(\boldsymbol{x_o}) + \int_{\partial\Omega} \phi(\boldsymbol{y}) \frac{\partial}{\partial \eta(\boldsymbol{y})} (\frac{1}{2\pi} \log \mid \boldsymbol{x}_o - \boldsymbol{y} \mid) \quad (4.7)$$

$$\lim_{\boldsymbol{x}_{unbdd} \to \boldsymbol{x}_o} u(\boldsymbol{x_{unbdd}}) = -\frac{1}{2}\phi(\boldsymbol{x_o}) + \int_{\partial\Omega} \phi(\boldsymbol{y}) \frac{\partial}{\partial \eta(\boldsymbol{y})} (\frac{1}{2\pi} \log \mid \boldsymbol{x}_o - \boldsymbol{y} \mid) \quad (4.8)$$

4.2.3 Simply connected domains:

First consider the bounded case where the boundary consists of a single contour. Represent the solution to Eq. 4.1 as a double layer potential (with $\eta = \eta_{unbdd}$).

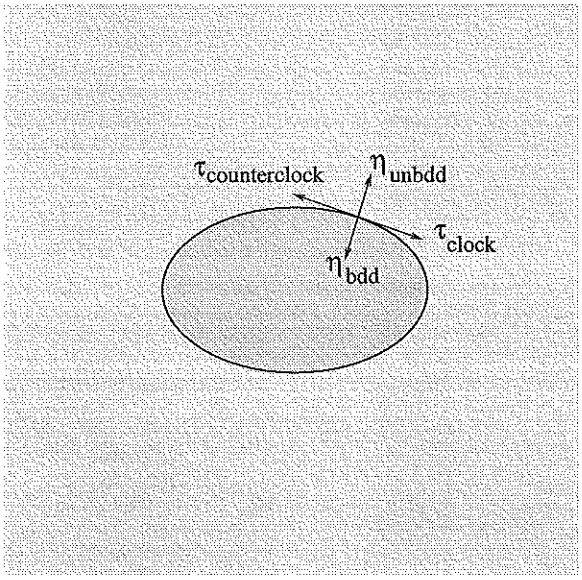

Figure 4.4: Normals and tangents relative to a contour.

As mentioned, this representation satisfies the PDE, and from Eq. 4.7, satisfying the given boundary data is equivalent to solving an integral equation.

$$\frac{1}{2}\phi(\boldsymbol{x_o}) + \int_{\partial\Omega} \phi(\boldsymbol{y}) \frac{\partial}{\partial\eta(\boldsymbol{y})}\left(\frac{1}{2\pi}\log \mid \boldsymbol{x}_o - \boldsymbol{y} \mid\right) ds(\boldsymbol{y}) = g(\boldsymbol{x}_o) \tag{4.9}$$

This equation is uniquely solvable [13], and once the density ϕ is known, then Eq. 4.4 will be the solution to the Dirichlet problem.

4.2.4 Multiply connected domains:

Next, consider the unbounded domain exterior to a single contour. If we again attempt to represent the solution as a plain double layer potential, then we find that the resulting integral equation does not have a unique solution (we can add any constant to the charge density and still satisfy the integral equation). This can make it more difficult to solve the integral equation, but this difficulty can be avoided by modifying the kernel of the double layer. Therefore, we choose to represent the solution to the unbounded Dirichlet problem as a **modified double layer potential** [20].

$$u(\boldsymbol{x}) = \int_{\partial\Omega} \phi(\boldsymbol{y})[1 + \frac{\partial}{\partial\eta(\boldsymbol{y})}\left(\frac{1}{2\pi}\log \mid \boldsymbol{x} - \boldsymbol{y} \mid\right)] ds(\boldsymbol{y}) \tag{4.10}$$

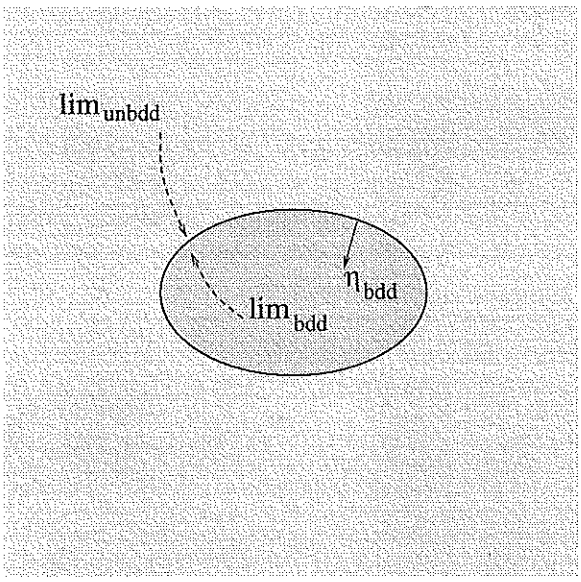

Figure 4.5: Limits approaching the contour using the bdd normal.

Note that the modified double layer potential still satisfies Laplace's equation away from the boundary, and when we attempt to satisfy the boundary condition (set $\eta = \eta_{unbdd}$, and use Eq. 4.8), we arrive at the following integral equation:

$$-\frac{1}{2}\phi(\boldsymbol{x_o}) + \int_{\partial\Omega} \phi(\boldsymbol{y})[1 + \frac{\partial}{\partial\eta(\boldsymbol{y})}(\frac{1}{2\pi}\log \mid \boldsymbol{x}_o - \boldsymbol{y} \mid)]\, ds(\boldsymbol{y}) = g(\boldsymbol{x}_o) \quad (4.11)$$

This modified equation can again be solved uniquely [20].

If we now look at the unbounded domain exterior to M contours (where M>1), and we seek a solution as in Eq. 4.10, then enforcing the boundary condition leads to an integral equation with a M-1 dimensional nullspace (spanned by M-1 constants on the M contours). This does not mean that the integral equation is unsolvable (by Fredholm's theorem there are an infinite number of solutions if g is orthogonal to the nullspace of the adjoint equation), but instead of worrying about the nullspaces we choose to follow the procedure described by Mikhlin [27] (and later used in [14]) and modify the representation of the solution. Mikhlin showed that the following integral equation is uniquely solvable (here $\eta = \eta_{unbdd}$, and g_2 can be any continuous boundary data):

$$-\frac{1}{2}\phi(\boldsymbol{x_o}) + \int_{\partial\Omega} \phi(\boldsymbol{y})[1 + R(\boldsymbol{x}_o, \boldsymbol{y}) + \frac{\partial}{\partial\eta(\boldsymbol{y})}(\frac{1}{2\pi}\log \mid \boldsymbol{x}_o - \boldsymbol{y} \mid)]\, ds(\boldsymbol{y}) = g_2(\boldsymbol{x}_o) \quad (4.12)$$

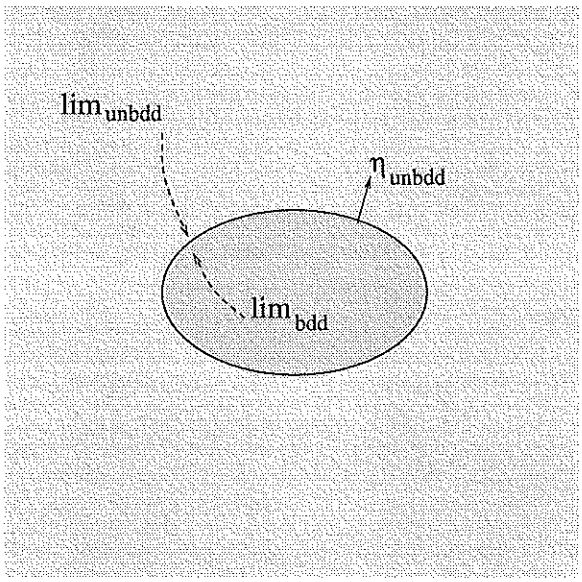

Figure 4.6: Limits approaching the contour using the unbdd normal.

$$R(\boldsymbol{x}_o, \boldsymbol{y}) = \begin{cases} 1 & \text{if} \quad \boldsymbol{x}_o, \boldsymbol{y} \in \partial\Omega_k, \quad k = 1, \ldots, M-1 \\ 0 & \text{else} \end{cases}$$

Armed with this result, we seek a solution with the following form,

$$u(\boldsymbol{x}) = \int_{\partial\Omega} \phi(\boldsymbol{y})[1 + \frac{\partial}{\partial\eta(\boldsymbol{y})}(\frac{1}{2\pi}\log \mid \boldsymbol{x} - \boldsymbol{y} \mid)]\, ds(\boldsymbol{y}) + \sum_{k=1}^{M} A_k \log \mid \boldsymbol{x} - \boldsymbol{z_k} \mid \quad (4.13)$$

where $\boldsymbol{z_k}$ is an interior point of the kth object. This representation satisfies Laplace's equation, and to specify the M constants $\{A_k\}$ we add M constraints.

$$\int_{\partial\Omega_k} \phi(\boldsymbol{y})\, ds(\boldsymbol{y}) = 0, \quad k = 1, \ldots, (M-1) \quad (4.14)$$

$$\sum_{k=1}^{M} A_k = 0 \quad (4.15)$$

Enforcing the boundary conditions leads to the equation,

$$-\frac{1}{2}\phi(\boldsymbol{x_o}) + \int_{\partial\Omega} \phi(\boldsymbol{y})[1 + \tfrac{\partial}{\partial\eta(\boldsymbol{y})}(\tfrac{1}{2\pi}\log \mid \boldsymbol{x}_o - \boldsymbol{y} \mid)]\, ds(\boldsymbol{y}) \quad (4.16)$$

$$= g(\boldsymbol{x}_o) - \textstyle\sum_{k=1}^{M} A_k \log \mid \boldsymbol{x}_o - \boldsymbol{z_k} \mid \quad (4.17)$$

$$\int_{\partial\Omega_k} \phi(\boldsymbol{y})\, ds(\boldsymbol{y}) = 0, \qquad k = 1, \ldots, (M-1) \quad (4.18)$$

$$\textstyle\sum_{k=1}^{M} A_k = 0$$

Note that the final constraint (Eq. 4.15) ensures that the solution satisfies the condition at infinity (Eq. 4.3), and the M-1 constraints (Eq. 4.14) make the resulting integral equation equivalent to the uniquely solvable problem (where $g_2(\boldsymbol{x}_o) \equiv g(\boldsymbol{x}_o) - \sum_{k=1}^{M} A_k \log \mid \boldsymbol{x}_o - \boldsymbol{z_k} \mid$). Therefore our choice of representation converts the Dirichlet problem into an integral equation that we can solve uniquely.

We follow a similar procedure when dealing with bounded multiply connected domains. Assume that there is one bounding contour and M inner contours (where M>0), and seek a solution in the following form (here $\eta = \eta_{unbdd}$ on the bounding contour and $\eta = \eta_{bdd}$ on the inner contours):

$$u(\boldsymbol{x}) = \int_{\partial\Omega} \phi(\boldsymbol{y}) \frac{\partial}{\partial\eta(\boldsymbol{y})} (\frac{1}{2\pi} \log \mid \boldsymbol{x} - \boldsymbol{y} \mid) \, ds(\boldsymbol{y}) + \sum_{k=1}^{M} A_k \log \mid \boldsymbol{x} - \boldsymbol{z_k} \mid \quad (4.19)$$

We add M constraints to specify the M log coefficients:

$$\int_{\partial\Omega_k} \phi(\boldsymbol{y}) \, ds(\boldsymbol{y}) \quad = 0, \quad k = 1, \ldots, M \quad (4.20)$$

Applying the boundary conditions leads to a uniquely solvable integral equation [27].

$$\frac{1}{2}\phi(\boldsymbol{x_o}) + \quad \int_{\partial\Omega} \phi(\boldsymbol{y}) \frac{\partial}{\partial\eta(\boldsymbol{y})} (\tfrac{1}{2\pi} \log \mid \boldsymbol{x}_o - \boldsymbol{y} \mid) \, ds(\boldsymbol{y}) \quad (4.21)$$

$$= g(\boldsymbol{x}_o) - \textstyle\sum_{k=1}^{M} A_k \log \mid \boldsymbol{x}_o - \boldsymbol{z_k} \mid \quad (4.22)$$

$$\textstyle\int_{\partial\Omega_k} \phi(\boldsymbol{y}) \, ds(\boldsymbol{y}) = 0, \quad k = 1, \ldots, M$$

Now that we have derived an integral formulation of Laplace's equation for both bounded and unbounded multiply connected domains, we must consider how to solve the resulting integral equations. In the next section a numerical method is considered and efficiency issues are explored.

4.3 Numerical Issues:

The integral equations we have derived can be solved analytically only for a few special cases. If general geometries and data are considered, some numerical procedure must generally be used. There are a variety of numerical approaches available to us, including the Nyström, collocation, and Galerkin methods. We will compute approximate solutions by applying a Nyström [29] method, which is an efficient method for the two dimensional domains we will consider. This method will now be presented, along with a discussion on how one can make it fast.

4.3.1 Discretization of integral equations:

First, a numerical quadrature method is used to replace the integrals with a finite sum. We currently use the trapezoid rule because of its simplicity and because it is spectrally accurate when the contours and charge densities are smooth (as can be seen from the Euler-MacLaurin formula [1]). If we sample n_k boundary points on the kth contour ($\{\boldsymbol{x}_i^k\}$, $i = 1, \ldots, n_k$), the discretized integral can be written as a simple sum.

$$\begin{aligned} &\int_{\partial\Omega_k} \phi(\boldsymbol{y}) \frac{\partial}{\partial\eta(\boldsymbol{y})}\left(\frac{1}{2\pi}\log \mid \boldsymbol{x}_o - \boldsymbol{y} \mid\right) ds(\boldsymbol{y}) \\ &\approx \sum_{i=1}^{n_k} \phi(\boldsymbol{x}_i^k) \frac{\partial}{\partial\eta(\boldsymbol{x}_i^k)}\left(\frac{1}{2\pi}\log \mid \boldsymbol{x}_o - \boldsymbol{x}_i^k \mid\right) h_i \end{aligned} \tag{4.23}$$

Here h_i represents the average arclength of the two boundary intervals that have $\boldsymbol{x}_i^k$ as an endpoint.

Next, we enforce the integral equation at each of the sample boundary points and apply the quadrature rule. Note that when the integration point coincides with the evaluation point ($\boldsymbol{x}_o = \boldsymbol{x}_i^k$), the kernel appears to be undefined; however, a closer examination reveals a well defined limit.

$$\lim_{\substack{\boldsymbol{x}\to\boldsymbol{x}_o \\ \boldsymbol{x}\in\partial\Omega_k}} \frac{\partial}{\partial\eta(\boldsymbol{x})}\left(\frac{1}{2\pi}\log \mid \boldsymbol{x}_o - \boldsymbol{x} \mid\right) = \frac{1}{4\pi}\kappa(\boldsymbol{x}_o) \tag{4.24}$$

Here $\kappa(\boldsymbol{x}_o)$ is the curvature of the contour at $\boldsymbol{x}_o$.

These approximations reduce the integral equation (and constraints) to a finite dimensional matrix equation which can be solved for the log coefficients and the charge densities at the sampled boundary points.

$$\begin{pmatrix} \frac{1}{2}\boldsymbol{I} + \boldsymbol{D_{cntr}} & \boldsymbol{L_{cntr}} \\ \boldsymbol{D_{con}} & \boldsymbol{L_{con}} \end{pmatrix} \begin{bmatrix} \vec{\phi} \\ \vec{A} \end{bmatrix} = \begin{pmatrix} \vec{g} \\ \vec{0} \end{pmatrix} \tag{4.25}$$

where

$$\vec{\phi} = \begin{bmatrix} \phi(\boldsymbol{x}_1^1) \\ \vdots \\ \phi(\boldsymbol{x}_{n_M}^M) \end{bmatrix}, \vec{A} = \begin{bmatrix} A_1 \\ \vdots \\ A_M \end{bmatrix}, \vec{g} = \begin{bmatrix} g(\boldsymbol{x}_1^1) \\ \vdots \\ g(\boldsymbol{x}_{n_M}^M) \end{bmatrix} \tag{4.26}$$

$\boldsymbol{D_{cntr}}$ represents the discrete contribution of the double layer potentials, $\boldsymbol{L_{cntr}}$ the effects of the log terms, $\boldsymbol{D_{con}}$ holds the discrete density constraints, $\boldsymbol{L_{con}}$ has the constraints on the log terms (a zero matrix for the case of a bounded domain), and $\boldsymbol{I}$ is the identity matrix.

4.3.2 Solution of discrete equations:

In this thesis, the linear system is solved using Gaussian elimination. We use this direct matrix solver for simplicity of development, but since its cost increases as the cube of the number of unknowns, we expect that it would become computationally expensive when the number of unknowns is large. On the other hand, although the matrix is dense it is well conditioned, and the condition number does not increase as we refine. This feature makes iterative solvers attractive when the number of unknowns is large because the cost of a matrix vector multiplication only grows as the square of the number of unknowns. Better still, if we forego an exact matrix multiply and use an intelligent approximation based on a hierarchical partitioning of space, we obtain an $O(n)$ procedure known as the fast multipole method or FMM [32, 15, 8]. The complexity of the FMM algorithm is optimal, but the asymptotic constant can be large (depending on the level of accuracy specified), and a FMM enhanced iterative method will be faster than a direct solver when the number of unknowns is roughly a few hundred. Optimally our matrix solver would simply check the number of unknowns and then select the direct or iterative solver as appropriate.

4.3.3 Evaluation of integral representation:

After we solve for the unknowns, another numerical issue is how to evaluate the solution at a set of points. The simplest approach is to apply the quadrature method and compute the resulting finite sum. There are two possible drawbacks to this procedure: First, if the solution must be evaluated at many points, then this process incurs a significant computational cost. Second, the discretized integral yields a poor approximation if the evaluation point lies near the boundary. This poor resolution is not surprising since the derivatives of the integrand increase as the evaluation point nears the boundary, hence the quadrature errors (for a fixed discretization level) increase as well. However, this particular failing does not bother us because we only use the evaluated integral solution in a preconditioner (see Section 2.3), and so we do not require great accuracy throughout the domain. If one is interested in other applications, the loss of resolution can be avoided by using adaptive integration or by using a method of local corrections [22] which we will presently discuss.

One way of accelerating the evaluation process is to apply the previously mentioned FMM. The FMM can be used to evaluate our integral representation at a collection of points in an asymptotically optimal way; however, because of the large asymptotic constant involved, it turns out to be faster to use a fast Poisson solver in a method of local corrections.

The method of local corrections:

The method of local corrections is a general procedure for approximately evaluating a function, u, on a given grid Ω_h, and particular instances have appeared in [22, 23, 24, 3, 38]. The method requires three conditions to be satisfied: First, there must be an invertible discrete operator $\mathcal{L}_h$, where our function's response to that operator is approximately known for most grid points ($f(x_{i,j}) \approx \mathcal{L}_h(u)(x_{i,j})$ is known at most $x_{i,j} \in \Omega_h$). Second, there must be some way of calculating the function's response at the few remaining grid points. Finally, a rapid method must exist for inverting the discrete operator ($\mathcal{L}_h^{-1}$ can be applied rapidly).

If these three conditions are met, the method of local corrections proceeds by first specifying the known responses for most of the grid points, filling in (or correcting) the responses at the few (local) remaining points, and finally calling the rapid inverse operator to obtain the function values on the grid ($u \approx \mathcal{L}_h^{-1}(f)$). The details of the method depend on the particular function being evaluated and on the grid we are given. For example, if we want to evaluate a very costly function on an equally spaced grid where most of the function's Fourier frequencies are known, and if the remaining frequencies can be determined, then a FFT can be used to evaluate the function rapidly on all grid points. Or, if we need to evaluate a complicated function on an unstructured grid, and if we know that the function satisfies a certain elliptic equation, then we can apply a multigrid scheme to determine the function values on the grid.

For our particular problem, we need to evaluate an integral representation (such as Eq. 4.19) at all the points of a Cartesian grid. We know that the analytic Laplacian of our solution is zero in our domain, so it seems reasonable that the discrete Laplacian should be approximately zero. If the standard five point discrete Laplacian is used, then zero is a second-order accurate approximation to the discrete Laplacian. (Zero is a fourth-order approximation if the nine point discrete Laplacian is used.) Furthermore, we note that we can rapidly invert the discrete Laplacian on a Cartesian grid by using a standard fast Poisson solver. Asymptotically, these solvers require $O(n \log(n))$ operations to solve for n unknowns, however their asymptotic constants are very small and in practice they are extremely fast. Therefore, we should be able to apply the method of local corrections to rapidly evaluate our solution. The one remaining issue is that the discrete Laplacian values are not known at all of the Cartesian grid points. Our integral representation is harmonic in the domain but it is not harmonic through the boundary, and we cannot assume that the discrete Laplacian is zero when some of the stencil points lie on opposite sides of the boundary. We will refer to these points as irregular grid points (as in Section 2.1), while points whose standard five point stencils lie completely on one side of the boundary will be known as regular points (Note: the standard five point stencil of a given grid point is the portion of the mesh that connects that point to its nearest neighbors as illustrated in Figs. 4.7 and 4.8). In order to im-

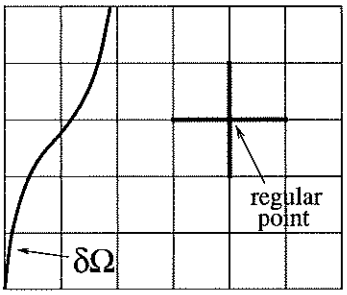

Figure 4.7: A standard five point stencil at a regular point.

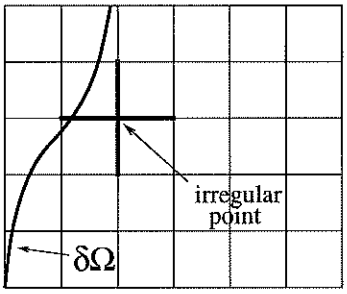

Figure 4.8: A standard five point stencil at an irregular point.

plement our fast evaluation procedure, we must consider how to form the local corrections to the discrete Laplacian at these additional points.

A direct method for determining local corrections:

A simple way of determining the discrete forcing terms for irregular points is to evaluate the solution at grid points near the boundary and then to compute the discrete Laplacian using these values. The evaluations can be done using a direct summation of a quadrature rule (Eq. 4.23) [23] or by applying the FMM[25], and this approach achieves a speed up because we evaluate the solution at fewer points. There are two issues that should be considered when using this approach. First, since the forcing terms come from a numerical evaluation of the solution, the accuracy of the method of local corrections is limited to the accuracy of this initial evaluation algorithm. Therefore, near the boundary we can still see the same loss of accuracy that occurs when we do a direct evaluation at all grid points. In our application (Section 2.3), we only use our evaluated solution as a preconditioner, so this possible loss of accuracy does not bother us (however, if one needs full accuracy throughout the domain then one should be more

careful). Another issue is that if we replace our double layer potential with an approximate sum, then we cannot assume that the discrete Laplacian values are close to zero at all regular grid points. Analytically, the Laplacian of both the double layer and the approximate sum are zero, but the discrete Laplacian only approximates the analytic Laplacian up to an error term that depends on the derivatives. For example, when we use the standard five point discrete Laplacian and assume a uniform grid mesh width, h, we see the following behavior:

$$\begin{aligned} \Delta_h u &= \Delta u - \frac{h^2}{12}(u_{xxxx} + u_{yyyy}) \\ &= 0 - \frac{h^2}{12}(u_{xxxx} + u_{yyyy}) \end{aligned} \tag{4.27}$$

The double layer potential has bounded fourth derivatives, so its discrete Laplacian will be near zero at all regular grid points. On the other hand, the fourth derivatives of the discrete sum behave like $\frac{h_{IE}}{d^5}$, where d is the distance between the boundary and evaluation point and h_{IE} is the boundary interval length used to discretize the integral (assume for now that the boundary has been discretized into intervals of equal arclength). Therefore, the discrete Laplacian of the direct sum is larger at points near the boundary, and we cannot always assume that it is zero.

For example, consider the annular domain with inner radius of 0.12, outer radius of 0.34, and center at (0.5,0.5). We use the integral equation techniques of Section 4.2 to solve a Laplace problem on the domain, where boundary values are defined by the function $x^2 - y^2$ (the boundary is discretized using 40 points per contour). We embed the domain in a Cartesian grid (Fig. 4.9), and evaluate the integral solution on grid points in the domain (using both direct summation and the method of local corrections). If we use a direct summation to calculate the discrete forcing terms, and if we only calculate forcing terms for grid points that lie a fixed number (three) of mesh widths from the boundary, then we see a degradation of our rapidly evaluated solution as we refine the Cartesian mesh. Specifically, if we denote the solution obtained using direct summation by u^d, and the rapidly obtained solution by u^r, then the difference between the two solutions increases as we evaluate them on finer grids (see Table. 4.1).

In our implementation of this approach, we compensate for this effect by calculating discrete forcing terms for additional points up to a distance $\tilde{d}$ from the boundary, where $\tilde{d} = h + h_{IE}$. Computing forcing values out to this distance ensures that the resulting method of local corrections produces solution values that remain close to the directly calculated values for any level of grid refinement (see Table 4.2). For the problems in this thesis, this distance is computationally reasonable because we generally use enough boundary points to produce a fairly resolved integral solution, and so $\frac{h_{IE}}{h}$ is usually less than 4. However, asymptotically, the number of correction terms needed for this fixed distance is of the same order as the total number of Cartesian gridpoints! Therefore (although it is not necessary for our particular problem), one can choose a less

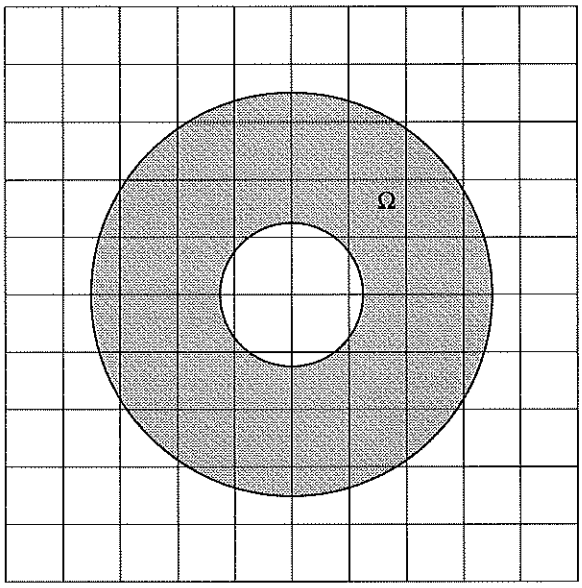

Figure 4.9: Test geometry embedded in a Cartesian grid.

restrictive distance criterion to decrease the computational cost when $h \ll h_{IE}$. For instance, if $\tilde{d} \propto h^{2/5}$, then we observe a roughly constant difference between the directly evaluated and rapidly evaluated solution, as shown in Table 4.3.

In the multiply connected case, we also have log sources to evaluate (see Eq. 4.13), and we similarly determine their contributions to the discrete forcing terms by evaluating the log terms at grid points near the source locations (we shift the source location slightly if it happens to lie directly on a grid point), and then calculating the discrete Laplacian of these values (this was previously done in [3]). Again we can choose to calculate the forcing terms out to a simple fixed distance, or a more involved distance that varies with the mesh width.

Table 4.1: Results of using direct computation to form local corrections at points a fixed number of mesh widths from the boundary.

Grid	$\|u^d - u^r\|_\infty$	$\|u^d(.5,.25) - u^r(.5,.25)\|$
40x40	2.6e-05	1.5e-05
80x80	0.00128	0.00016
160x160	0.00330	0.00077
320x320	0.01095	0.00140

Table 4.2: Results of using direct computation to form local corrections at points a fixed distance from the boundary.

Grid	$\|u^d - u^r\|_\infty$	$\|u^d(.5,.25) - u^r(.5,.25)\|$
80x80	0.00128	0.00016
160x160	0.00016	4.0e-05
320x320	6.2e-05	1.7e-05

Table 4.3: Results of using direct computation to form local corrections at points a distance proportional to $h^{2/5}$ from the boundary.

Grid	$\|u^d - u^r\|_\infty$	$\|u^d(.5,.25) - u^r(.5,.25)\|$
80x80	0.00017	6.8e-05
160x160	0.00016	4.0e-05
320x320	0.00036	0.00011

It should be noted that since we use our evaluated integral only as a preconditioner (Sec. 2.3), we may not need a great deal of accuracy; therefore, the distance $\tilde{d}$ can be adjusted to provide a faster evaluation process that is still an effective preconditioner.

Mayo's method for determining local corrections:

Another way of determining the forcing for the irregular points was given by Mayo[22]. To describe this alternate approach, we re-examine the five point discrete Laplacian of a double layer potential: If we apply Taylor's theorem to the discrete Laplacian, we find that it is a second-order approximation to the analytic Laplacian (Eq. 4.27). This analysis assumes that the solution can be expanded in a Taylor series that is valid along all the stencil arms of the discrete Laplacian, and this assumption is valid for regular grid points. For irregular points, the solution is represented by a different Taylor series on each side of the boundary, and when we apply our analysis, we obtain jump terms where the two series meet. For example, consider an irregular point $\boldsymbol{x}_{i,j}$ whose stencil intersects the boundary along all four of its arms. As shown in Fig. 4.10 we denote the right, left, up, and down intersection points by $\boldsymbol{x}_R, \boldsymbol{x}_L, \boldsymbol{x}_U$, and $\boldsymbol{x}_D$ while the distances from the center of the stencil to these intersection points are labeled d_R, d_L, d_U, and d_D. Furthermore, subscripted brackets will be used to denote the jump values from the side containing the

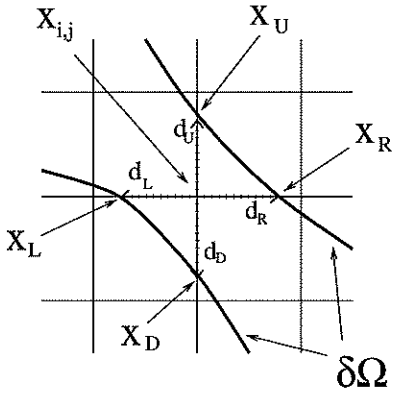

Figure 4.10: Notation for an irregular point.

extremal stencil point to the side containing the center stencil point: $[u]_o(\boldsymbol{x}_R) \equiv u(x_R^+, y_R) - u(x_R^-, y_R)$, $[u]_o(\boldsymbol{x}_L) \equiv u(x_L^-, y_L) - u(x_L^+, y_L)$, $[u]_o(\boldsymbol{x}_U) \equiv u(x_U, y_U^+) - u(x_U, y_U^-)$, and $[u]_o(\boldsymbol{x}_D) \equiv u(x_D, y_D^-) - u(x_D, y_D^+)$. Using this notation, a Taylor series analysis of the discrete Laplacian reveals the following relationship (for convenience we assume that the grid has an equal mesh width h in both the x and y directions):

$$\begin{aligned}
\Delta_h u = \Delta u \\
&+ \frac{[u]_o(\boldsymbol{x}_R) + (h-d_R)[u_x]_o(\boldsymbol{x}_R) + \frac{(h-d_R)^2}{2}[u_{xx}]_o(\boldsymbol{x}_R) + \frac{(h-d_R)^3}{6}[u_{xxx}]_o(\boldsymbol{x}_R)}{h^2} \\
&+ \frac{[u]_o(\boldsymbol{x}_L) - (h-d_L)[u_x]_o(\boldsymbol{x}_L) + \frac{(h-d_L)^2}{2}[u_{xx}]_o(\boldsymbol{x}_L) - \frac{(h-d_L)^3}{6}[u_{xxx}]_o(\boldsymbol{x}_L)}{h^2} \\
&+ \frac{[u]_o(\boldsymbol{x}_U) + (h-d_U)[u_y]_o(\boldsymbol{x}_U) + \frac{(h-d_U)^2}{2}[u_{yy}]_o(\boldsymbol{x}_U) + \frac{(h-d_U)^3}{6}[u_{yyy}]_o(\boldsymbol{x}_U)}{h^2} \\
&+ \frac{[u]_o(\boldsymbol{x}_D) - (h-d_D)[u_y]_o(\boldsymbol{x}_D) + \frac{(h-d_D)^2}{2}[u_{yy}]_o(\boldsymbol{x}_D) - \frac{(h-d_D)^3}{6}[u_{yyy}]_o(\boldsymbol{x}_D)}{h^2} \\
&+ O(h^2)
\end{aligned} \tag{4.28}$$

From this equation we see that we can accurately approximate the discrete Laplacian at an irregular point if we know the analytic Laplacïan (zero for a double layer potential) and the jump values of the solution and its derivatives. As Mayo showed in her paper [22], when dealing with a double layer potential these jump terms can be computed directly from the charge density and its derivatives. If we denote the boundary intersection point as $\boldsymbol{x}_o = \boldsymbol{x}_o(s) = (x_o(s), y_o(s))$ where s denotes some parameterization of the boundary, and we introduce a jump notation [ ] to represent the jump from the exterior side to the interior side (i.e. $[u(\boldsymbol{x_o})] \equiv u(\boldsymbol{x_o}(s) + \epsilon\hat{\boldsymbol{\eta}}(s)) - u(\boldsymbol{x_o}(s) - \epsilon\hat{\boldsymbol{\eta}}(s))$, where $\epsilon > 0, \epsilon \ll 1$, and $\hat{\eta}$ points out of the domain), then the parameterized jump

terms are given by the following formulas

$$\begin{aligned}
[u] &= -\phi \\
[u_x] &= -\frac{\dot{x}_o\dot{\phi}}{\dot{x}_o^2+\dot{y}_o^2} \\
[u_y] &= -\frac{\dot{y}_o\dot{\phi}}{\dot{x}_o^2+\dot{y}_o^2} \\
[u_{xx}] &= -\frac{(\dot{x}_o^4-\dot{y}_o^4)\ddot{\phi}+(\ddot{x}_o(-\dot{x}_o^3+3\dot{x}_o\dot{y}_o^2)+\ddot{y}_o(\dot{y}_o^3-3\dot{x}_o^2\dot{y}_o))\dot{\phi}}{(\dot{x}_o^2+\dot{y}_o^2)^3} \\
[u_{yy}] &= -[u_{xx}]
\end{aligned} \tag{4.29}$$

By using these equations, we can approximately compute the discrete forcing terms at the irregular points without having to do any solution evaluations at all. Furthermore, since we calculate the forcing directly from the charge densities, the method of local corrections does not lose accuracy at points near the boundary. As for the log terms, we cannot apply the above formulas since these point sources do not have the the same type of jump behavior. Therefore, their contribution to the discrete forcing terms will be determined by differencing the local evaluations of the log values as previously described.

The Mayo procedure possesses two advantages: First, it provides an approximate solution which does not lose resolution near the boundary. Second, it can be calculated rapidly because correction terms only need to be found at irregular points, and because these corrections only involve local information about the charge densities. The disadvantage of the Mayo procedure is that it is complicated and can be difficult to implement. It requires the charge density and its derivatives to be known at arbitrary boundary points, and one must identify all irregular points and points where the grid lines intersect the boundary. This detailed information can quickly become cumbersome, but the proper use of computational tools can make these details more manageable. For instance, the COG class library discussed in Chapter 2 encapsulates the details of representing the boundary, grid, and data; thereby freeing the user to focus on how to use that information without worrying about how to obtain and store it.

4.4 Choosing Dirichlet constants subject to flux constraints:

So far we have discussed how to formulate Laplace's equation as an integral equation and how to solve the resulting equation efficiently. Most of the discussion has appeared in previous works since integral equations have been used to solve Laplace's equation for many years. In exploring the fluid simulations of this thesis, it was found that the classical integral approach used for Laplace's

equation could (with modifications) be fruitfully applied to areas in fluid dynamics. In particular, our Navier-Stokes solver requires the solution of a Laplace equation which differs from the standard Dirichlet problem. In this section we will state what this Laplace problem is and describe how we can use a modified integral approach to efficiently solve it.

4.4.1 Statement of the problem:

In our Navier-Stokes solver (Chapter 3), the nature of the boundary conditions associated with viscous flow requires us to solve the following problem: Let Ω be a bounded domain in the plane with a C^2 boundary consisting of one bounding contour $\partial\Omega_{M+1}$ and M inner contours $\partial\Omega_1 \cdots \partial\Omega_M$ (Fig. 4.1). Given M fluxes, $l_1, \ldots, l_M$, determine M Dirichlet data constants $c_1, \ldots, c_M$ that will produce a solution to Laplace's equation with the desired fluxes. In other words the constants are chosen so that if the following equation is solved,

$$\begin{aligned} \Delta u(\boldsymbol{x}) &= 0, \quad \boldsymbol{x} \in \Omega \\ \lim_{\substack{\boldsymbol{x}\to\boldsymbol{x}_o \\ \boldsymbol{x}\in\Omega}} u(\boldsymbol{x}) &= \begin{cases} c_k & \text{if } \boldsymbol{x}_o \in \partial\Omega_k, \ k = 1, \ldots, M \\ 0 & \text{if } \boldsymbol{x}_o \in \partial\Omega_{M+1} \end{cases} \end{aligned} \tag{4.30}$$

then

$$\int_{\partial\Omega_k} \frac{\partial}{\partial\eta(\boldsymbol{x})} u(\boldsymbol{x})\, ds(\boldsymbol{x}) = l_k, \quad k = 1, \ldots, M. \tag{4.31}$$

(Note: in our Navier-Stokes solver, the Laplace solution only has to be determined up to an additive constant. For this reason we can arbitrarily fix the Dirichlet data constant for any one contour, and for consistency we always set the constant for the bounding contour to zero. The unbounded situation exterior to M contours is similar, where M-1 Dirichlet constants are chosen to satisfy M-1 fluxes, and zero data is used for the Mth contour). Two methods will be given for the solution of this problem: the first has been previously used in [35] works, and requires the solution of M elliptic problems, while the second is an approach which (through the use of an integral formulation) only requires a single elliptic solve.

4.4.2 An existing approach:

We introduce the notation $u(\boldsymbol{x}|c_1 \ldots c_M)$ to represent the solution to Laplace's equation where c_j is the Dirichlet data on the jth contour (and zero data is specified on the bounding contour). Note that the linearity of the Laplace

operator implies that both the solution and the total fluxes about each contour are linear functions of the Dirichlet data $\{c_j\}$. That is, $u(\boldsymbol{x}|c_1 \ldots c_M) = \sum_{j=1}^{M} c_j u(\boldsymbol{x}|0 \ldots 1 \ldots 0)$ and

$$\int_{\partial\Omega_i} \frac{\partial}{\partial\eta(\boldsymbol{x})} u(\boldsymbol{x}|c_1 \ldots c_M)\, ds(\boldsymbol{x}) = \sum_{j=1}^{M} c_j \int_{\partial\Omega_i} \frac{\partial}{\partial\eta(\boldsymbol{x})} u(\boldsymbol{x}|0 \ldots 1 \ldots 0)\, ds(\boldsymbol{x}),$$

where the unit data is specified on the jth contour. We need to choose the Dirichlet data to satisfy the given fluxes:

$$\sum_{j=1}^{M} c_j \int_{\partial\Omega_k} \frac{\partial}{\partial\eta(\boldsymbol{x})} u(\boldsymbol{x}|0 \ldots 1 \ldots 0)\, ds(\boldsymbol{x}) \quad = \quad l_k, \quad k = 1, \ldots, M \quad (4.32)$$

If we denote $K_{ij} = \int_{\partial\Omega_i} \frac{\partial}{\partial\eta(\boldsymbol{x})} u(\boldsymbol{x}|0 \ldots 1 \ldots 0)\, ds(\boldsymbol{x})$ (again the unit data appears on the jth contour), then we can write our problem as a matrix equation

$$\begin{pmatrix} K_{1,1} & \cdots & K_{1,M} \\ \vdots & & \vdots \\ K_{M,1} & \cdots & K_{M,M} \end{pmatrix} \begin{pmatrix} c_1 \\ \vdots \\ c_M \end{pmatrix} = \begin{pmatrix} l_1 \\ \vdots \\ l_M \end{pmatrix} \quad (4.33)$$

which can be solved to obtain the constants $c_1, \ldots, c_M$. To obtain the matrix values $\{K_{ij}\}$, we solve M Laplace equations with Dirichlet data and then compute the flux about each contour. If finite differences or finite elements are used, additional calculations are needed to calculate the fluxes (this approach was used in [35]), whereas if the integral representation is used (Eq. 4.21) then the fluxes are determined automatically ($\int_{\partial\Omega_k} \frac{\partial}{\partial\eta(\boldsymbol{x})} u(\boldsymbol{x})\, ds(\boldsymbol{x}) = 2\pi A_k$). The main drawback of this approach is that an elliptic solve is needed to generate each matrix column in Eq. 4.33, therefore, it will involve a large amount of computation if the boundary consists of many contours. Furthermore, when the boundary is allowed to move, the matrix itself can change, thus this approach requires M elliptic problems to be solved at every time step. We would like to be able to consider flows about multiple moving bodies, so we are interested in finding better ways of solving Eqs. 4.30-4.31.

4.4.3 An integral approach:

A more efficient method involves a modification of the integral equation approach of Section 4.2. First, the log coefficients are chosen to ensure that the integral representation satisfies the prescribed fluxes ($A_k = \frac{l_k}{2\pi}$, $k = 1, \ldots, M$). Next the data constants $c_1, \ldots, c_M$ are treated as additional unknowns that are determined by the M solvability constraints. This results in the following

integral equation (where $c_{M+1} \equiv 0$) :

$$\frac{1}{2}\phi(\boldsymbol{x_o}) \quad + \int_{\partial\Omega} \phi(\boldsymbol{y}) \frac{\partial}{\partial\eta(\boldsymbol{y})}\left(\frac{1}{2\pi}\log \mid \boldsymbol{x}_o - \boldsymbol{y} \mid\right) ds(\boldsymbol{y}) - c_j \tag{4.34}$$

$$= \sum_{k=1}^{M} \frac{l_k}{2\pi}\log \mid \boldsymbol{x}_o - \boldsymbol{z_k} \mid \tag{4.35}$$

$$\boldsymbol{x}_o \in \partial\Omega_j, \quad j = 1, \ldots, M+1 \tag{4.36}$$

$$\int_{\partial\Omega_k} \phi(\boldsymbol{y})\, ds(\boldsymbol{y}) = 0, \qquad k = 1, \ldots, M$$

This integral equation is similar to the ones we have previously considered (such as Eq. 4.21), and we even have the same constraints that ensure its solvability. The difference is that now our unknowns are the charge densities and the Dirichlet data constants, and this difference allows us to obtain all of the data constants simultaneously by solving a single integral equation. We solve this equation by using the methods discussed in Section 4.3, where we first apply a quadrature method which converts the integral equation into a single linear system

$$\begin{pmatrix} \frac{1}{2}\boldsymbol{I} + \boldsymbol{D_{cntr}} & \boldsymbol{I2} \\ \boldsymbol{D_{con}} & \boldsymbol{0} \end{pmatrix} \begin{bmatrix} \vec{\phi} \\ \vec{c} \end{bmatrix} = \begin{pmatrix} \vec{g} \\ \vec{0} \end{pmatrix} \tag{4.37}$$

where

$$\vec{\phi} = \begin{bmatrix} \phi(\boldsymbol{x_1}) \\ \vdots \\ \phi(\boldsymbol{x}_{\boldsymbol{n_M}}) \end{bmatrix}, \vec{c} = \begin{bmatrix} c_1 \\ \vdots \\ c_M \end{bmatrix}, \vec{g} = \begin{bmatrix} \sum_{k=1}^{M} \frac{l_k}{2\pi}\log \mid \boldsymbol{x}_1^1 - \boldsymbol{z_k} \mid \\ \vdots \\ \sum_{k=1}^{M} \frac{l_k}{2\pi}\log \mid \boldsymbol{x}_{n_M}^M - \boldsymbol{z_k} \mid \end{bmatrix} \tag{4.38}$$

Here, $\boldsymbol{D_{cntr}}$ represents the discrete contribution of the double layer potentials, and $\boldsymbol{D_{con}}$ holds the discrete charge density constraints. $\boldsymbol{I2}$ is a matrix that incorporates the Dirichlet constants into the solution, and assuming that there are n_j points on the jth contour, its jth column ($j = 1, \ldots, M$) consists of zeroes and n_j entries of -1 starting at row $1 + \sum_{k=1}^{j-1} n_k$. This matrix equation can be solved directly or iteratively (as discussed in the previous section), and will yield the desired Dirichlet constants $c_1, \ldots, c_M$ with no additional computations.

This integral approach is an attractive method for dealing with Eqs. 4.30-4.31 when the domain contains several contours. The main benefits are that it only requires a single elliptic solve no matter how many contours there are, it does not require the evaluation of the solution on a grid, and fast methods (like the FMM) can be applied to efficiently solve the resulting integral equation when the number of unknowns is large.

4.4.4 Summary:

We have examined how integral equations can be used to efficiently solve two types of Laplace problems. The first is Laplace's equation in a multiply connected domain with Dirichlet data. The second problem involves choosing

Dirichlet data subject to constraints on the local fluxes of the solution. Both problems are solved in the projection step of our fluid solver, where integral equations are used to form correction terms that convert a procedure that works in simple domains into a method for complex domains. These correction terms serve to take the presence of the boundaries into account, and therefore increase the flexibility of methods that rely on simple geometry.

Appendix A

A Data-Structure-Neutral Iterative Solver Class:

To solve the linear systems that appear in our Navier-Stokes algorithm, we used an iterative method. Our linear operators are represented as functions, and are not stored in matrix form. Likewise, the quantities we solve for are given on a flagged cut-out Cartesian grid, and are not sequentially stored in vectors. The iterative solvers contained in software libraries typically require that the linear system be represented by vectors and matrices, but this format is inconvenient for our applications. The iterative algorithms themselves are expressed in terms of mathematical operators which do not have to be expressed in terms of vectors and matrices, and by using a particular programming approach (which emphasizes functions and pointers), one can implement iterative methods without assuming a particular matrix-vector representation. Some library routines now let users provide functions to apply the linear operators, instead of passing in actual matrices. We decided to extend this approach (following the example set in [18, 17, 4]), and let users provide both their own operators and their own data representations. This frees the user from having to use an unnecessary (and often unnatural) matrix-vector representation, and can make it easier to use iterative methods. Additionally, not having to explicitly form vectors and matrices can reduce both the number of computations performed and the amount of memory used. This data-structure-neutral approach was very useful in implementing our Navier-Stokes algorithm, and some documentation for the software is included in this appendix.

class Iterative_soln

Introduction:

The Iterative_soln class is a data type for the representation and solution of linear systems. When a class object is created, the user supplies parameters that specify the particular linear system being considered. The parameters corresponding to "vectors" (the right hand side and solution) are represented by void pointers which can hold the address of any class or data structure, while key operators which act on these generalized "vectors" (such as the matrix application) are represented using function pointers. Once the class object has been created, member functions can be used to operate on the linear system, and a choice of iterative solvers can be applied (CG, PCG, GMRES, and FGMRES).

The advantage of our use of pointers is that it results in a flexible, data-structure-neutral approach [18, 17, 4], where users are free to use their own data representations. For example, if the linear system corresponds to solving a discretized partial differential equation, then users can continue to work with their grids and loops instead of having to convert them into vectors and matrices. This flexibility makes it easier to solve linear systems with non-trivial representations.

public member functions:

`Iterative_soln(...)`

Description:

The constructor for the class. It creates a class object that represents the particular linear system described by the user-supplied argument list. These arguments are as follows:

max_iter : the maximum number of iterations to be attempted.

tol : the relative residual tolerance level to be used as the stopping criterion in the iterative solvers.

soln : a pointer to a "vector" that will be filled with the iterative solution after a solver has been called. If the initial guess flag has been set (see init_guess_on()), the "vector" will be used as an initial guess by the iterative solver.

rhs : a pointer to a "vector" that represents the right hand side for the linear system.

a_mult : a pointer to a function that applies the linear operator("matrix").

pre_cond : a pointer to a function that applies the preconditioner.

create : a pointer to a function that creates new "vectors".

destroy : a pointer to a function that frees up(deletes) "vectors".

copy : a pointer to a function that copies one "vector" to another.

inner_prod : a pointer to a function that computes the inner product of two "vectors".

add : a pointer to a function that finds the sum of two "vectors".

scale : a pointer to a function that scales a "vector" by a constant.

obj : a void pointer that is associated with the linear system. This is an optional parameter that can be used when additional information needs to be passed around.

`void cg()`
Description:
Solve the linear system represented by this Iterative_soln object using the conjugate gradient method. After completion, the user specified solution "vector" holds the final iterate, and both the relative residual error achieved and the number of iterations taken are stored internally. (see get_tol(double) and get_max_iter(long))

`void pcg()`
Description:
Solve the linear system represented by this Iterative_soln object using the preconditioned conjugate gradient method. After completion, the user specified solution "vector" holds the final iterate, and both the relative residual error achieved and the number of iterations taken are stored internally. (see get_tol(double) and get_max_iter(long))

`void gmres(long restart_iter=0)`
Description:
Take an int, and solve the linear system represented by this Iterative_soln object using the GMRES algorithm. The int is used as the restart parameter. After completion, the user specified solution "vector" holds the final iterate, and both the relative residual error achieved and the number of iterations taken are stored internally. (see get_tol(double) and get_max_iter(long))

`void fgmres(long restart_iter=0)`
Description:
Take an int, and solve the linear system represented by this Iterative_soln object using the FGMRES algorithm. The int is used as the restart parameter This algorithm differs from regular GMRES in that it stores the preconditioned

residual "vectors". This results in more storage but half as many preconditioner calls. After completion, the user specified solution "vector" holds the final iterate, and both the relative residual error achieved and the number of iterations taken are stored internally. (see get_tol(double) and get_max_iter(long))

`void set_max_iter(long max)`
Description:
Take a long as a parameter, and store it as the maximum number of iterations that an iterative method will try before stopping.

`void get_max_iter(long& max)`
Description:
Take a long by reference, and change it to hold the current value of the maximum iteration parameter. Before an iterative method is called, this parameter holds the maximum number of iterations that will be attempted. After a method is called, this parameter holds the actual number of iterations taken.

`void set_tol(double tolerance)`
Description:
Take a double and store it as the relative residual tolerance level (the stopping criterion for the iterative methods).

`void get_tol(double& tolerance)`
Description:
Take a double by reference and change it to hold the current value of the tolerance parameter. Before an iterative method is called, this parameter holds the stopping criterion to be used. After a method is called, this parameter holds the relative residual of the iterative solution.

`void init_guess_on()`
Description:
Set the initial guess flag to indicate that the given solution "vector" contains an initial guess for the iterative solvers.

`void init_guess_off()`
Description:
Set the initial guess flag to indicate that the given solution "vector" does not contain an initial guess for the iterative solvers.

`void set_pre_cond(void(*p_c)(void*,void*,void*))`
Description:
Take a pointer to a function and store it as the preconditioner.

`void set_rhs(void* b)`
Description:
Take a pointer to a "vector", and store it as the right hand side of the linear system.

`void initialize(...)`
Description:
An initialization function. It takes the same arguments as the main constructor, and initializes the current object to represent the linear system described by the argument list.

`void initialize(const Iterative_soln& B)`
Description:
An initialization function. It takes an Iterative_soln object as a reference argument, and initializes the current object as a true copy of the argument.

`Iterative_soln(const Iterative_soln& B)`
Description:
The copy constructor. It takes an Iterative_soln object as a reference argument, and creates the current object as a true copy of the argument.

Bibliography

[1] M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions.* Dover Publications, Inc., New York, 1965.

[2] A. S. Almgren, J. B. Bell, P. Colella, and T. Marthaler. A cartesian grid projection method for the incompressible Euler equations in complex geometries. *SIAM J. Sci. Comput.*, (to appear).

[3] C. R. Anderson. A method of local corrections for computing the velocity field due to a distribution of vortex blobs. *J. Comput. Phys.*, 62:111–123, 1986.

[4] S. Balay, L. Curfman McInnes, W. Gropp, and B. Smith. PETSc 2.0 users manual. Technical Report ANL-95/11, Argonne National Laboratory, 1995.

[5] S. A. Bayyuk, K. G. Powell, and B. van Leer. An algorithm for the simulation of 2-D unsteady inviscid flows around arbitrarily moving and deforming bodies of arbitrary geometry. Technical Report 93-3391, AIAA, July 1993.

[6] J. Benek, J. Steger, and F. Dougherty. A flexible grid embedding technique with application to the Euler equations. In *Proceedings of the 6th AIAA Computational Fluid Dynamics Conference*, 1983.

[7] M. J. Berger and R. J. LeVeque. An adaptive Cartesian mesh algorithm for the Euler equations in arbitrary geometries. Technical Report 89-1930, AIAA, 1989.

[8] J. Carrier, L. Greengard, and V. Rokhlin. A fast adaptive multipole algorithm for particle simulations. *SIAM. J. Sci. Stat. Comput.*, 9:669–686, 1988.

[9] G. Chessire and W. D. Henshaw. Composite overlaping meshes for the solutoin of partial differential equations. *J. Comput. Phys.*, 90:1–64, 1990.

[10] A. J. Chorin. Numerical solution of the Navier-Stokes equations. *Math. Comp.*, 22:745–762, 1968.

[11] A. J. Chorin and J. A. Marsden. *A Mathematical Introduction to Fluid Mechanics.* Springer-Verlag, New York, 1990.

[12] W. J. Coirier and K. G. Powell. Solution-adaptive Cartesian cell approach for visous and invisid flows. *AIAA J.*, 34:938–945, 1996.

[13] G. B. Folland. *Introduction to Partial Differential Equations.* Princeton University Press, Princeton, 1976.

[14] A. Greenbaum, L. Greengard, and G. B. McFadden. Laplace's equation and the Dirichlet-Neumann map in multiply connected domains. *J. Comput. Phys.*, 105:267–278, 1993.

[15] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comput. Phys.*, 73:325–348, 1987.

[16] P. M. Gresho and R. L. Sani. On pressure boundary conditions for the incompressible Navier-Stokes equations. *Int. J. Numer. Meth. Fluids*, 7:1111–1145, 1987.

[17] W. Gropp and B. Smith. The design of data-structure-neutral libraries for the iterative solution of sparse linear systems. Technical Report Preprint MCS-P356-0393, (to appear in Scientific Programming), Argonne National Laboratory.

[18] W. Gropp and B. Smith. Users manual for KSP: Data-structure-neutral codes implementing Krylov space methods. Technical Report ANL-93/30, Argonne National Laboratory, August 1993.

[19] W. D. Henshaw. A fourth-order accurate method for the incompressible Navier-Stokes equations on overlapping grids. *J. Comput. Phys.*, 113:12–25, 1994.

[20] R. Kress. *Linear Integral Equations.* Springer-Verlag, Berlin, 1989.

[21] J. D. Lambert. *Computational Methods in Ordinary Differential Equations.* John Wiley & Sons, Ltd., Chichester, 1973.

[22] A. Mayo. The fast solution of Poisson's and the biharmonic equations on irregular regions. *SIAM J. Numer. Anal.*, 21:285–299, 1984.

[23] A. Mayo. Fast high order accurate solution of Laplace's equation on irregular regions. *SIAM J. Sci. Stat. Comput.*, 6:144–156, 1985.

[24] A. Mayo. The rapid evaluation of volume integrals of potential theory on general regions. *J. Comput. Phys.*, 100:236–245, 1992.

[25] A. McKenney, L. Greengard, and A. Mayo. A fast Poisson solver for complex geometries. *J. Comput. Phys.*, 118:348–355, 1995.

[26] J. E. Melton, M. J. Berger, M. J. Aftosmis, and M. D. Wong. 3D applications of a Cartesian grid Euler method. Technical Report 95-0853, AIAA, 1995.

[27] S. G. Mikhlin. *Integral Equations.* Pergammon Press, London, 1957.

[28] N. I. Muskhelishvili. *Singular Integral Equations.* Dover Publications Inc, New York, 1992.

[29] E. J. Nyström. Über die praktische auflösung von integralgleichungen mit anwendungen auf randwertaufgaben. *Acta. Math.*, 54:185–204, 1930.

[30] R. Peyret and T. D. Taylor. *Computational Methods for Fluid Flow.* Springer-Verlag, New York, 1983.

[31] S. E. Rodgers, D. Kwak, and C. Kiris. Steady and unsteady solutions of the incompressible Navier-Stokes equations. *AIAA J.*, 29:603–610, 1991.

[32] V. Rokhlin. Rapid solution of integral equations of classical potential theory. *J. Comput. Phys.*, 60:187–207, 1985.

[33] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM. J. Sci. Comput.*, 14:461–469, 1993.

[34] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM. J. Sci. Stat. Comput.*, 7:856–869, 1986.

[35] C. Shakarji. *Object Oriented, Numerical methods for Two Dimensional, Incompressible Flows, Using Overlapping Grids.* PhD thesis, University of California, Los Angeles, 1995.

[36] E. Y. Tau. A second-order projection method for the incompressible Navier-Stokes equations in arbitrary domains. *J. Comput. Phys.*, 115:147–152, 1994.

[37] J. Y. Tu and L. Fuchs. Overlapping grids and multigrid methods for 3-dimensional unsteady flow calculations in IC engines. *Int. J. Numer. Meth. Fluids*, 15:693–714, 1992.

[38] S. O. Unverdi and G. Tryggvason. A front-tracking method for viscous, incompressible, multi-fluid flows. *J. Comput. Phys.*, 100:25–37, 1992.

[39] D. M. Young and R. T. Gregory. *A Survey of Numerical Mathematics, Vol 2.* Addison-Wesley Publishing Company, Inc., Philippines, 1973.

[40] D. Zeeuw and K. G. Powell. An adaptively refined Cartesian mesh solver for the Euler equations. *J. Comput. Phys.*, 104:56–68, 1993.