Surface and Volumetric Parametrisation using Harmonic Functions in Non-Convex Domains

Richard Klein
0707074G

Supervised by
Prof. Michael Sears and Dr. Hari Voruganti

A Dissertation submitted to the Faculty of Science, University of the Witwatersrand, in fulfillment of the requirements for the degree of Master of Science.

Johannesburg, 2013

Abstract

Many of the problems in mathematics have very elegant solutions. As complex, real–world geometries come into play, however, this elegance is often lost. This is particularly the case with meshes of physical, real–world problems. Domain mapping helps to move problems from some geometrically complex domain to a regular, easy to use domain. Shape transformation, specifically, allows one to do this in 2D domains where mesh construction can be difficult. Numerical methods usually work over some mesh on the target domain. The structure and detail of these meshes affect the overall computation and accuracy immensely. Unfortunately, building a good mesh is not always a straight forward task. Finite Element Analysis, for example, typically requires 4–10 times the number of tetrahedral elements to achieve the same accuracy as the corresponding hexahedral mesh. Constructing this hexahedral mesh, however, is a difficult task; so in practice many people use tetrahedral meshes instead. By mapping the geometrically complex domain to a regular domain, one can easily construct elegant meshes that bear useful properties. Once a domain has been mapped to a regular domain, the mesh can be constructed and calculations can be performed in the new domain. Later, results from these calculations can be transferred back to the original domain. Using harmonic functions, source domains can be parametrised to spaces with many different desired properties. This allows one to perform calculations that would be otherwise expensive or inaccurate.

This research implements and extends the methods developed in Voruganti *et al.* [2006 2008] for domain mapping using harmonic functions. The method was extended to handle cases where there are voids in the source domain, allowing the user to map domains that are not topologically equivalent to the equivalent dimension hypersphere. This is accomplished through the use of various boundary conditions as the void is mapped to the target domains which allow the user to reshape and shrink the void in the target domain. The voids can now be reduced to arcs, radial lines and even shrunk to single points. The algorithms were implemented in two and three dimensions and ultimately parallelised to run on the *Centre for High Performance Computing* clusters. The parallel code also allows for arbitrary dimension genus-0 source domains. Finally, applications, such as remeshing and robot path planning were investigated and illustrated.

Declaration

I, Richard Klein, hereby declare the contents of this dissertation to be my own work. This report is submitted for the degree of Master of Science at the University of the Witwatersrand. This work has not been submitted to any other university, or for any other degree.

[illegible]

Richard Klein
March 18, 2013

Acknowledgements

I would like to express my sincere gratitude to those who have contributed to this research and my studies. I would like to specifically thank:

Prof. Michael Sears and Dr. Hari Voruganti, my supervisors, for all the guidance and motivation throughout this research. I gratefully acknowledge their heavy investment in my academic endeavours, including arrangements for conference papers, presentations and the associated travel. All feedback has been supportive and constructive and is greatly appreciated.

My family – my parents, Margie and David, and my brother, Peter – I thank them for their advice, love, unwavering support and hours of proof-reading.

I thank all of those within the School of Computer Science who have assisted in so many ways. For particular advice regarding parallel computing, large datasets and visualisation, thanks to Prof. Ekow Otoo. I thank all the others who offered assistance debugging, destressing and generally getting this dissertation through to the finish line.

The research has been performed under the Centre for High Performance Computing Flagship Project Computational Imaging and Remote Sensing. Their training, equipment and funding has helped immensely. This research was also funded by the National Research Foundation and their contribution is valued.

Finally, I thank the Mathematical Science IT laboritories, specifically, Mr. Shunmunga Pillay, for the use of the Hydra cluster and the other high performance hardware used for this research.

Contents

List of Figures

List of Tables

List of Algorithms

Publications

Portions of this research have been presented at:

The 28th European Workshop on Computational Geometry and appeared in Klein *et al.* [2012].

The Fourth Cross-Faculty Postgraduate Symposium at the University of the Witwatersrand (2012).

The CHPC National Meetings in Durban (2012) and Pretoria (2011).

Chapter 1

Introduction

Many of the problems faced in mathematics are ambitious, not because of some inherent difficulty in the problems, but rather because of the space in which they reside. Many mathematical and numerical techniques are efficient, accurate and elegant until they are applied to the arbitrary shapes and spaces in the physical world. Domain mapping helps maintain these desirable properties while allowing one to work with arbitrary shapes and spaces.

Domain mapping is the process of mapping from a source domain to some target domain that exhibits certain desirable properties. Usually this means mapping some geometrically complicated source domain to a regular, convex domain where problems are simplified. This is achieved by finding some parametrisation of the original domain that is diffeomorphic. This means that there is a continuous and smooth map from the source domain onto the target domain and that the inverse map exists and is also continuous and smooth. Hence, one can map points forward, perform calculations in the target domain and map the results back to the source domain. When the source domain is non-convex, domain mapping is a highly useful tool in performing operations accurately and efficiently. Some applications of domain mapping include robot path planning and re-meshing, which are discussed later in Chapter 8.

With robot path planning, a path between two points in a non-convex space is required. In general, this is a non-trivial task. After mapping the domain to a convex shape, however, the points can be joined by a straight line. This line can then be mapped back to the original domain, providing the required path. A similar approach can be used for remeshing and other applications.

One severe disadvantage of domain mapping is that the initial calculation of the map may be computationally expensive. Worse still, the number of calculations increase exponentially in the number of dimensions of the source domain. Once the map is calculated however, points can be mapped forwards and backwards easily and in constant time. An efficient parallel implementation is required to leverage hardware and architectural improvements in computers.

This work extends the methods developed in Voruganti *et al.* [2006 2008] for domain mapping using an Artificial Potential Field approach based on harmonic functions. The original methods, however, can only deal with genus-0 domains – domains without voids. A number of applications exist which cannot be translated to genus-0 domains. The methods are thus extended to handle parametrisation of compact source domains that are not of genus-0; in other words, source domains that contain voids and thus are no longer simply connected. These methods for 2D, as well as the methods for n-dimensional genus-0 spaces were then implemented in serial in Matlab and then in C++. Finally the C++ code was parallelised and run on both the *University of the Witwatersrand* and the *Centre for High Performance Computing (CHPC)* clusters. This is the first computationally efficient implementation of these methods.

There are three principal models of parallel computing: distributed memory, shared memory and a hybrid of the two models. In the distributed memory model, the memory used by each process is local to that process. This is often the case when the processes are run on separate machines connected by some network. In this model, processes communicate by passing messages and data explicitly using a library such as the *Message Passing Interface* (MPI). In the shared memory model, processes are able to read and write to portions of shared memory independently. Such models are usually used when the program is running on a single multicore machine. This model of computing can be achieved using technologies such as *OpenMP*, *Unified Parallel C* (UPC) and *OpenSHMEM*. Finally, a hybrid approach uses a mixture of both shared memory and message passing. A distributed memory approach using MPI was used as this allows greater scaling with regards to the number of nodes. A hybrid approach, using shared memory on each node but message passing between nodes, may be appropriate for optimising memory use when multiple processes are run on each node.

The main contributions of this work include the extension of the above method of domain mapping to handle genus-$\emptyset$ domains as well as a parallel implementation of these methods. The parallel implementation is tested on the CHPC clusters and the results indicate that the problem scales well to run on parallel cores. This allows one to map larger, more complicated domains.

The remainder of this thesis is organised as follows. Chapter 2 considers background and related work. This includes basic explanations of domain mapping and its applications as well as the methods proposed in Voruganti *et al.* [2006 2008]. Harmonic functions and the necessary numerical methods that are used in the research are discussed. The chapter concludes with some notes on High Performance Computing (HPC) and the technologies used in the development of parallel applications. Chapter 3 sketches the aims of the research, a research hypothesis and various questions upon which the work focused.

Chapter 4 then provides a detailed description of the domain mapping algorithm developed in Voruganti *et al.* [2006 2008] for a 2D context. Once the basic algorithm has been described, the chapter develops two extensions to the method so that it can handle domains not of genus-0. The methods from this chapter were implemented in MATLAB.

Following on from the development of serial C++ code, Chapter 5 describes the parallelisation of the code from the previous chapter. Both changes to the algorithm and changes in the implementation are discussed. Chapter 6 compares the performance of the serial and parallel implementations. The parallel implementation was then extended to work in an arbitrary number of dimensions in Chapter 7.

This code is then run on applications in robot path planning and meshing in two and three dimensions in Chapter 8. Finally Chapter 9 provides a summary and conclusion to the dissertation along with the final contributions of the work and a number of topics for future research.

After the logical conclusion of the dissertation, there are two appendices. Appendix A lists pseudo code for the algorithms described within the body of the dissertation. Appendix B provides stereoscopic images from the 3D simulations that should aid in the reader's 3D interpretation of some visualisations.

Chapter 2

Background

2.1 Notation

This section provides the notation used throughout the rest of this dissertation.

Of primary importance is the concept domains of genus-0 versus domains not of genus-0. To assist in notation, any domain not of genus-0 will be denoted as genus-$\emptyset$.

Symbol	Definition
genus-$\emptyset$	Not of genus-0
2D Domain	Two Dimensional Domain
3D Domain	Three Dimensional Domain
∂B	Boundary around domain B

Symbol	Definition
$\phi()$	Potential distribution
$X(t)$	Vector of points along a streamline
x	A point along the streamline
$\vec{x}$	A vector quantity

2.2 Domain Mapping

Domain mapping allows one to move a problem from a geometrically complicated space into another domain where calculations are easier, more accurate or more efficient. This commonly involves the transformation of a geometrically complicated domain to a well-shaped, usually convex domain [Voruganti *et al.* 2008]. In general, this means finding some bijective map between the spaces.

Definition 2.1 *For a map $f : A \to B$, to be a bijection it must be both* one-to-one *and* onto *as shown in Equations 2.1 and 2.2.*

$$\textit{1-1} \quad \Leftrightarrow \quad \forall a, b \in A : f(a) = f(b) \Rightarrow a = b \tag{2.1}$$

$$\textit{Onto} \quad \Leftrightarrow \quad \forall b \in B : \exists a \in A | f(a) = b \tag{2.2}$$

A bijective map ensures that an inverse map exists. This is required so that points in the target domain can be mapped back to the source domain. One also requires that the map be continuous. A continuous and bijective map on a compact domain is called a *Homeomorphism*.

Ideally, the map should also be smooth so that the results of differential equations solved in the target domain are valid in the source domain. This means that the solution of some partial differential equation can be transformed back to the source domain. A map that is both homeomorphic and smooth is called a *Diffeomorphism*. If the map is at least C^r-diffeomorphic, then a differential equation of order r can be mapped to the target domain, solved and the solution mapped back to the source domain.

Definition 2.2 *A map, $f : A \to B$, is a C^r-diffeomorphism if both f and f^{-1} are r times continuously differentiable bijections between the two sets.*

Domain mapping, or volumetric parametrisation, has many applications in a variety of fields. These are found in computer graphics and animation, robotics, computational biology, image registration and recognition, as well as many medical and engineering contexts [Voruganti *et al.* 2006 2008; Martin *et al.* 2009; Xia *et al.* 2010]. For example, given an arbitrary non-convex, 2D domain of genus-0, robotic path planning can be a complicated task. Using domain mapping however, the domain can be mapped to a convex disk where one simply joins two points representing the desired configurations – by the definition of a convex space this is easy to do. The line joining the points is then inverted or mapped back to the original space and it provides a path between the configuration points in the original domain. While the gains in 2D may seem limited, it quickly becomes apparent that in higher dimensions domain mapping becomes increasingly important for this task. Section 2.3 below describes a number of applications.

Various approaches to domain mapping have been investigated. Many of these approaches have been developed in the context of robotic path planning due to the difficulties faced when planning paths in high dimensional spaces. Surayawamshi *et al.* [2003] used a Finite Element approach where the source domain is decomposed into a number of elastic triangles. Other approaches for path planning apply an Artificial Potential Field (APF) to the domain where streamlines then form the planned paths. While many potential functions have been proposed to help solve the robot path planning problem [Khatib 1986; Barraquand and Latombe 1991], there are criticisms for their local minima problems which cause inaccurate or incorrect results [Voruganti *et al.* 2006].

Solutions to the local minima problem have been proposed by Wang and Chirikjian [2000], who exploit similarities with the heat transfer problem with variable thermal conductivity. In Sunder and Shiller [1997], Hamilton-Jacobi-Bellman theory is used to establish a potential field. Another approach to the path planning problem is the use of a combination of simulated annealing algorithms with the APF.

Harmonic functions satisfy the maximum principle, which means that they completely eliminate the local minima problem. They have many elegant properties and this has attracted many researchers in their application to path planning [Voruganti *et al.* 2006]. Some of these properties are investigated later in Section 2.5. Voruganti *et al.* [2006] note that "no one exploited the strength of the harmonic functions completely." They have developed a novel potential field approach using harmonic functions to solve the path planning problem in arbitrary dimension, non-convex, genus-0 domains. This research implements and extends their method, which is discussed later in Section 2.4 and Chapter 4.

2.3 Applications of Domain Mapping

2.3.1 Meshing

Numerical methods usually require some discrete mesh on which to work. The construction of this mesh is not always straight forward [Xia *et al.* 2010; Han *et al.* 2010]. Finite Element Analysis is an example of a tool used in solving many scientific and engineering problems. The system works by decomposing some 3D model into a number of smaller, discrete elements. These are usually tetrahedral or hexahedral elements [Han *et al.* 2010].

Tetrahedral (pyramid) meshes typically require 4-10 times the number of elements needed to achieve the same accuracy as the corresponding hexahedral mesh [Han *et al.* 2010]. The problem however, is that obtaining a hexahedral mesh in a geometrically complicated domain can be extremely difficult. So in practice tetrahedral meshes are significantly more popular [Han *et al.* 2010].

Domain mapping can be applied to this situation to achieve the desired hexahedral mesh. The geometrically complicated domain is mapped to a regular convex domain where a hexahedral mesh is constructed. Calculations can then be performed on the regular, hexahedral mesh and are later mapped back to the original domain.

2.3.2 Remeshing

In the same vein as Section 2.3.1 above, domain mapping can also be used for re-meshing where the quality and/or smoothness of a mesh needs to be improved. Figure 2.1 from Fuhrmann *et al.* [2010] shows how re-meshing can be used to reduce the size of a mesh while maintaining its quality. It shows a red horse model made of about 50000 vertices (left) and a corresponding re-meshed blue horse made up of only 6000 vertices (right). Figure 2.2, also from Fuhrmann *et al.* [2010] shows the Hygieia model with 8000 vertices (left) up-sampled and smoothed to 10000 vertices (right).

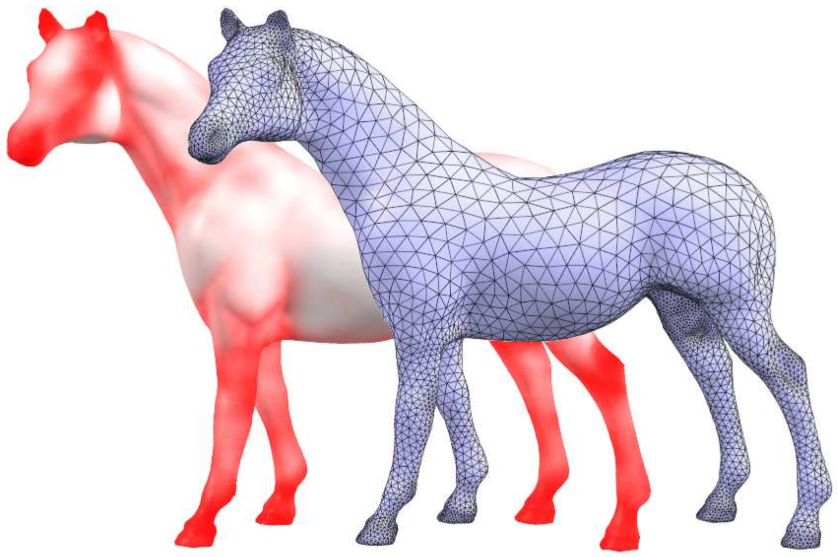


Figure 2.1: Down Sampling from 50000 vertices to 6000 vertices from [Fuhrmann *et al.* 2010]

This is not always feasible or practical when the domain is geometrically complex. By calculating the map between the source domain and a regular target domain the new map can be constructed on the convex domain and the map transforms the mesh in such a way that it has a higher density in non-convex areas of the source domain. This is illustrated further in Chapter 8.

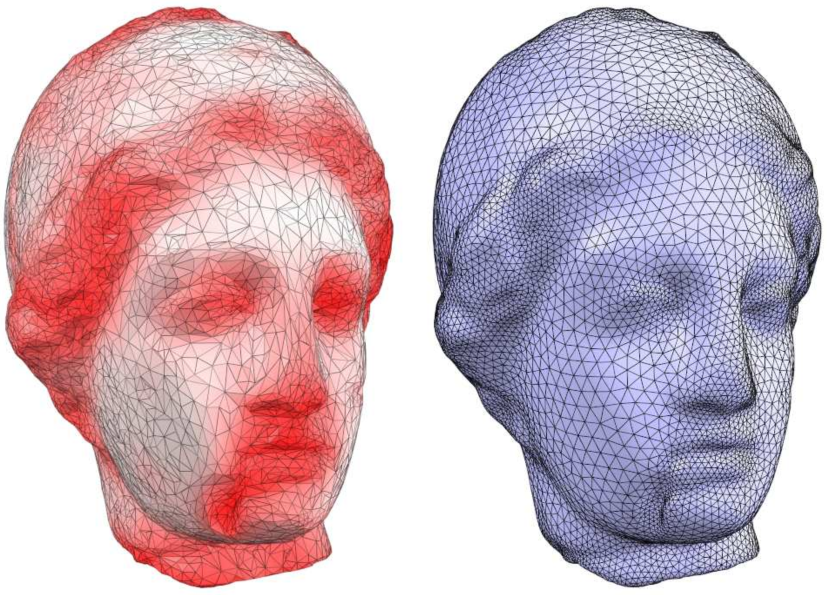


Figure 2.2: Up-sampling from 8000 vertices to 10000 vertices from [Fuhrmann *et al.* 2010]

2.3.3 Robot Path Planning

Another application of domain mapping lies in the area of robotics. If C is the configuration space of some robot, then every point $\vec{x} \in C$ represents some configuration of the joints of that robot. The dimensionality of C is usually equal to either the *number of joints* or the *degrees of freedom* of the robot in question. Therefore, moving the robot from one configuration to another, $\vec{x}_1 \rightarrow \vec{x}_2$, is equivalent to finding a path between these points such that the path remains inside $C - free$ i.e. the obstacle free space in C. [Latombe 1991]

This is a trivial problem in a convex space as one can simply draw a straight line between $\vec{x}_1$ and $\vec{x}_2$ to calculate the intermediate configurations of the robot. In a non-convex space, however, this becomes a much more difficult task as paths need to avoid non-convex boundaries and possibly even voids.

By applying domain mapping, a genus-0, geometrically complicated source domain can be transformed to a convex domain. The two relevant configurations of the robot are mapped forward and joined with a straight line in the convex domain. By the definition of a convex domain, this line remains entirely inside the domain. The points along this straight line are then mapped back to the source domain to give a valid path from $\vec{x}_1$ to $\vec{x}_2$ [Voruganti *et al.* 2006]. For domains of genus-$\emptyset$, Section 4.4 develops ways to manipulate the shape of the voids in the target domain so that path planning in the convex domain remains trivial.

2.3.4 Other Applications

Another application is found in computer animation. By mapping two different shapes to the same target domain, a continuous, bijective mapping is generated between them. This is useful for morphing/interpolating between the shapes in a process called *tweening*. More on the application of volumetric parametrisation to computer animation can be found in Chang *et al.* [2006].

Using a similar idea, domain mapping can be applied to computational biology's protein docking problem, which requires the registration of geometrically complex protein shapes. Domain mapping also finds uses in general registration and recognition problems. [Voruganti and Dasgupta 2009]

2.4 Domain Mapping on Genus-0 Domains

Voruganti *et al.* [2006 2008] present a new method for domain mapping on non-convex spaces of genus-0. The method parametrises the original domain by making use of both the values and the flows of an artificial potential field. Boundary values of high potential are applied to the exterior boundary of the source domain and a *shape centre* is chosen which is assigned a low potential value. Using these boundary conditions, Laplace's equation is solved over the source domain to provide potential values for each point.

Streamlines orthogonal to the contours of the field are then calculated based on the gradients of the resulting harmonic function. As the streamlines flow from high to low potential they all approach the selected shape centre. The streamlines are made up of a number of co-ordinate vectors, $X(t)$, originating from each boundary point at $t = 0$ and travelling towards the shape centre as t increases. If μ is a normalising constant, the streamlines can be calculated according to Equation 2.3 as the direction of $\bigtriangledown\phi$ is always orthogonal to the contour lines. The direction of $-\bigtriangledown\phi$ is also orthogonal, but in the opposite direction. The negative gradient is always the direction orthogonal to the contour facing the descent direction.

$$\dot{X}(t) = -\mu \bigtriangledown \phi[X(t)] \tag{2.3}$$

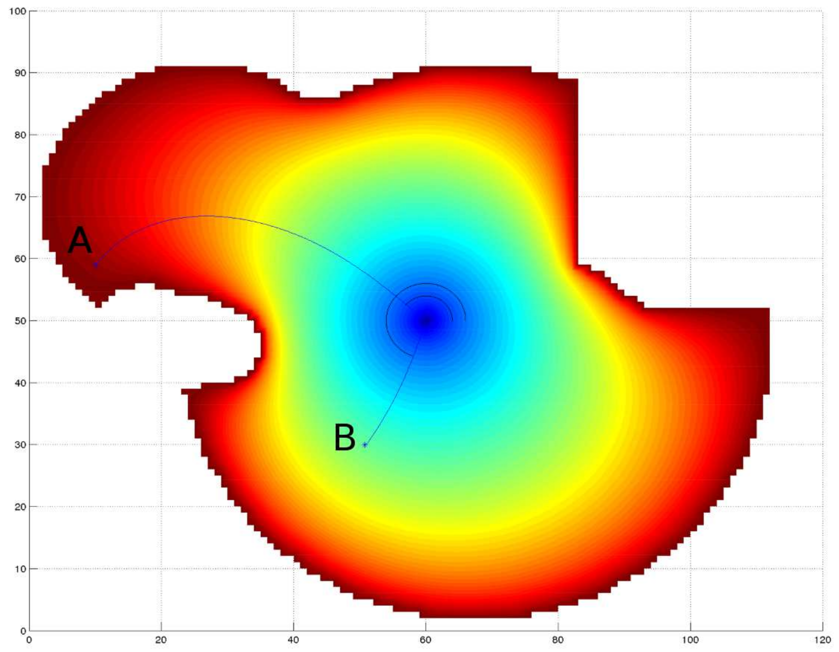
The domain is then parametrised to the spherical co-ordinates $(\vec{\theta}, \rho)$ using each point's potential value and the unique streamline that passes through it heading towards the shape centre. A point's potential value is used as the distance from the centre, $\rho = \phi(\vec{x})$, and the angle(s) at which the relevant streamline approaches the centre as $\vec{\theta}$. This is demonstrated in Figures 2.3(a) and 2.3(b) on the following page.

Figure 2.3(a) shows an example of this method on a 2D domain. The boundary of the domain is set to a high potential, while a centre point is chosen to have a low potential. The boundary of the source domain then maps to the boundary of the disk in Figure 2.3(b). The streamlines originating at the two points are then tracked to the centre point. The angles at which the streamlines approach the centre point are used as the θ values in the new domain, while the potential at the point is its ρ value. At this stage it is crucial that there are no local minima in the domain as streamlines may flow to those instead of the selected centre point. The Maximum Principle discussed in Section 2.5.3 guarantees that there are no local extrema.

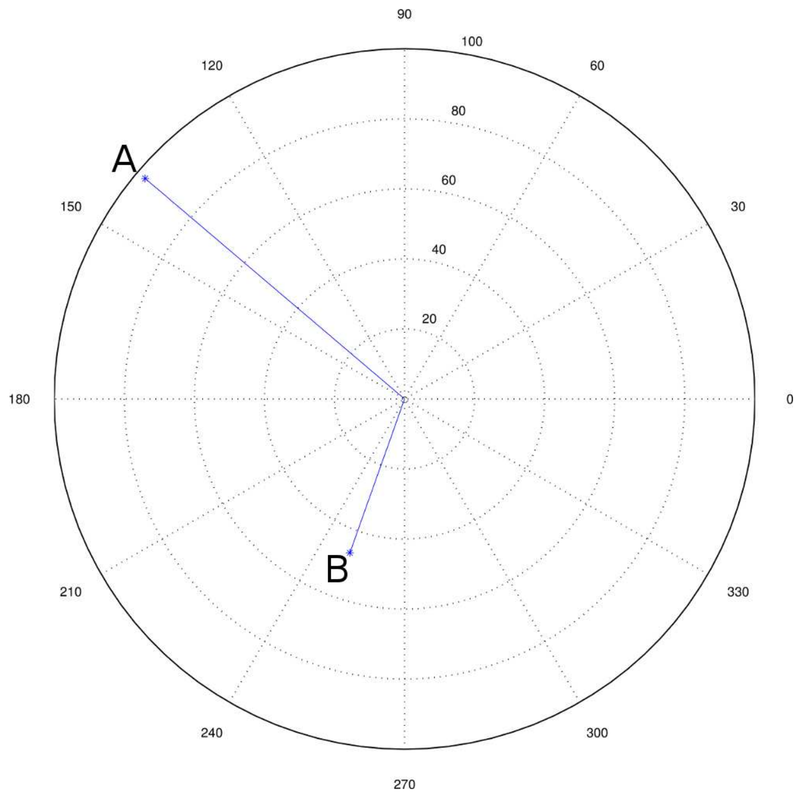
2.5 Harmonic Functions

2.5.1 Definition

The methods developed in this research rely heavily on Harmonic Functions as they possess some elegant properties that until recently were not fully utilised [Voruganti *et al.* 2008]. The two properties of primary importance to this research are the Mean Value Property and the Maximum Principle.



(a) Potential Value Distribution with 2 mapped points.



(b) Points in the mapped domain.

Figure 2.3: Domain Mapping Example

Definition 2.3 (Harmonic Functions) *A Harmonic function is any real valued function which solves Laplace's equation:*

$$\bigtriangledown^2 \phi = 0 \tag{2.4}$$

In other words any function $\phi : U \to \mathbb{R}$, where $U \subseteq \mathbb{R}^n$, is harmonic iff its Laplacian vanishes on U. So,

$$\bigtriangledown^2 \phi = \sum_{i=1}^{n} \frac{\partial^2 \phi}{\partial x_i^2} = 0 \tag{2.5}$$

where x_i is the i-th Cartesian co-ordinate and n is dimensionality of U.

2.5.2 The Mean Value Property

Harmonic functions satisfy the mean value property and, conversely, if a function satisfies the mean value property it is Harmonic [Weisstein 2011a]. The following theorem is based on Axler *et al.* [2001]; Voruganti *et al.* [2006 2008] and Weisstein [2011a].

Theorem 2.5.1 (The Mean Value Theorem) *If $B(\vec{x}, r)$ is a closed ball with centre $\vec{x}$ and radius r, and $B \subseteq U$, then the value of the harmonic function at the centre of the ball, $\phi(\vec{x})$, is the average value of ϕ at the surface of the ball, $\partial B(\vec{x}, r)$. Formally, where V_n is the volume of the unit ball in n dimensions:*

$$\begin{aligned} \phi(\vec{x}) &= \frac{1}{\mathit{SurfaceAreaB}} \int_{\partial B(\vec{x},r)} \phi(\theta) d\theta \\ &= \frac{1}{nV_n r^{n-1}} \int_{\partial B(\vec{x},r)} \phi(\theta) d\theta \end{aligned} \tag{2.6}$$

A useful numerical result follows from this property. If we discretise U with some uniform mesh, then the potential value of any point is just the average of the neighbouring potential values. Hence, the 2D and 3D cases result in Equations 2.7 and 2.8 respectively. These are exactly the finite difference equations that are derived later in Section 2.6.1. For this reason, the finite difference method is extremely well suited to calculate the potential field.

$$\phi(x_i, y_j) \approx \frac{1}{4} \left[\phi(x_{i+1}, y_j) + \phi(x_{i-1}, y_j) + \phi(x_i, y_{j+1}) + \phi(x_i, y_{j-1})\right] \tag{2.7}$$

$$\begin{aligned} \phi(x_i, y_j, z_k) \approx \frac{1}{6}[&\phi(x_{i+1}, y_j, z_k) + \phi(x_{i-1}, y_j, z_k) + \phi(x_i, y_{j+1}, z_k) + \\ &\phi(x_i, y_{j-1}, z_k) + \phi(x_i, y_j, z_{k+1}) + \phi(x_i, y_j, z_{k-1})] \end{aligned} \tag{2.8}$$

2.5.3 The Maximum Principle

As noted in Section 2.2, one of the primary problems with Artificial Potential Field (APF) methods is the issue of local extrema in the induced field. In general, domain mapping, as well as the method set out in Voruganti *et al.* [2006], fails if local extrema occur in the potential field as streamlines may be attracted to local extrema rather than the desired point. As a result the bijectivity of the map is lost. Conversely, a potential field with no local extrema guarantees a bijective mapping [Voruganti *et al.* 2006].

Theorem 2.5.2 (The Maximum Principle) *Unless a harmonic function, ϕ, is constant, it achieves extrema on its boundaries only.*

This means that if there are at least two different boundary values in the system, there are no local extrema on the interior points. As the method described in Section 2.4 has boundary values of 0 and 1, this is guaranteed. Therefore streamlines will always flow from the domain boundary to the desired shape centre without interference from local minima or maxima.

2.5.4 Applications

Laplace's equation arises in many physical problems as both initial and boundary value problems. For example, the heat equation in n dimensions is,

$$\alpha \bigtriangledown^2 \phi = \frac{\partial \phi}{\partial t} \tag{2.9}$$

where $\phi(\vec{x}, t) : U \times \mathbb{R} \to \mathbb{R}$ is the temperature at any point $\vec{x} \in U \subseteq \mathbb{R}^n$ at time $t \geq 0$. When the temperature of the domain has reached an equilibrium, however, it is said to have reached a steady state and the change in temperature over time becomes and remains zero. So if t_e is the time at which the system reaches equilibrium,

$$\forall t \geq t_e; \text{ the appropriate model becomes } \frac{\partial \phi}{\partial t} = 0. \tag{2.10}$$

Thus Equation 2.9 reduces to Laplace's equation (Equation 2.4). Similarly, many problems in the areas of fluid dynamics, gravitational potential and electrostatics reduce to Laplace's equation.

2.5.5 Intuitive understanding of boundary conditions

For an intuitive understanding of the potential field, Laplace's equation can be interpreted as the steady state heat equation with the relevant boundary conditions. This intuition helps one understand how potential (heat) is distributed over the domain of interest. Areas of high potential are synonymous with high temperatures while low potentials relate to lower temperatures. As Laplace's equation corresponds to the steady state of the system, no boundary conditions can rely on time. This interpretation of the potential values is helpful in understanding the following methods' application to source domains of genus-$\emptyset$.

Following the discussion of genus-0 domains, Section 4.4 focuses on developing methods to handle domains that contain voids. Interpretations of different physical boundary conditions will be useful in developing an intuitive understanding of the behaviour of the potential field. Relating Laplace's equation back to the steady state heat equations, the following conditions on the boundary, ∂U, can be given physical interpretations:

- Dirichlet Boundary Conditions ($\phi(\partial U) = c$) correspond to some constant heat sink or source.
- Neumann Boundary Conditions ($\vec{n} \cdot \bigtriangledown \phi(\partial U) = c$) can now be interpreted as forcing some area to maintain a constant heat flux.
- Robin Boundary Conditions ($\alpha \phi(\partial U) + \beta \vec{n} \cdot \bigtriangledown \phi(\partial U) = c$) equate to convection conditions.

The Neumann boundary condition with $c = 0$ (zero heat flux) means that potential (heat) cannot pass over a certain area. In physical terms this equates to insulation. This special case is useful later when dealing with voids.

2.6 Numerical Methods

This research makes use of a number of numerical methods to calculate both the artificial potential field and the streamlines. The primary numerical method used for the potential field calculation is finite difference. Following the potential field calculation, the streamlines are calculated using the Runge-Kutte method with adaptive step-size control. These are explained in detail in the following two sections.

2.6.1 Finite Difference

As mentioned in Section 2.5.2, Laplace's equation lends itself to a finite difference approximation. The method can be derived from the definition of the first derivative or from a truncated Taylor series. In one dimension the derivative is,

$$\frac{d\phi}{dx} = \lim_{h\to 0} \frac{\phi(x+h) - \phi(x)}{h} \tag{2.11}$$

In a discritised domain, let h be some small step size. The equation thus becomes,

$$\begin{aligned} \frac{d\phi}{dx}(x) &\approx \frac{\phi(x+h) - \phi(x)}{h} \\ \frac{d\phi}{dx}(x_i) &\approx \frac{\phi(x_{i+1}) - \phi(x_i)}{h} \end{aligned} \tag{2.12}$$

where $x_{i+1} = x_i + h$. Using the same method, the second derivative becomes,

$$\frac{d^2\phi}{dx^2}(x_i) \approx \frac{\phi(x_{i+1}) - 2\phi(x_i) + \phi(x_{i-1})}{h^2} \tag{2.13}$$

In two dimensions the results are similar,

$$\frac{\partial^2\phi}{\partial x^2}(x_i, y_j) \approx \frac{\phi(x_{i+1}, y_j) - 2\phi(x_i, y_j) + \phi(x_{i-1}, y_j)}{h^2} \tag{2.14}$$

$$\frac{\partial^2\phi}{\partial y^2}(x_i, y_j) \approx \frac{\phi(x_i, y_{j+1}) - 2\phi(x_i, y_j) + \phi(x_i, y_{j-1})}{h^2} \tag{2.15}$$

Hence, Laplace's equation in 2D and 3D becomes,

$$0 \approx \phi(x_{i+1}, y_j) + \phi(x_{i-1}, y_j) - 4\phi(x_i, y_j) + \phi(x_i, y_{j+1}) + \phi(x_i, y_{j-1}) \tag{2.16}$$

$$\begin{aligned} 0 \approx\ & \phi(x_{i+1}, y_j, z_k) + \phi(x_{i-1}, y_j, z_k) + \phi(x_i, y_{j+1}, z_k) + \\ & \phi(x_i, y_{j-1}, z_k) + \phi(x_i, y_j, z_{k+1}) + \phi(x_i, y_j, z_{k-1}) - 6\phi(x_i, y_j, z_k) \end{aligned} \tag{2.17}$$

which still obeys the mean value property outlined in Section 2.5.2. Because of its natural correspondence with the mean value property, as well as its ease of use, finite difference is used to solve Laplace's equation throughout this research.

Another benefit of the Finite Difference Method, is that the matrices produced are highly sparse. This can be leveraged to minimize memory usage and increase performance when solving the set of linear equations. This is discussed further in Section 2.7.

2.6.2 Streamlines: Runge-Kutte, Dormand-Prince & Adaptive Step-size Control

Runge-Kutte

Once the potential field has been found, streamlines orthogonal to the contours must be calculated. Each streamline consists of a number of points $\vec{x}_i$ and is the solution of an initial value problem with $t = 0$ and starting at each boundary point. At each point $- \bigtriangledown \phi(x_i)$ is the direction of steepest descent and is orthogonal to the contour passing through that point. This results in Equation 2.18 where $\mu > 0$ is some normalisation value that adjusts the step-size.

$$\vec{x}(t_{i+1}) = \vec{x}(t_i) - \mu \bigtriangledown \phi[\vec{x}(t_i)] \tag{2.18}$$

This is the discritised form of Equation 2.3 and is the well known formula for Euler's method of solving initial value problems for Ordinary Differential Equations. In terms of stability and efficiency, however, "Euler's method is not recommended for practical use" [Press *et al.* 2007, pg 907]. The primary problem with this formula is that it is nonsymmetrical, it advances using derivative information at only the beginning of the interval. By calculating points within the interval, one can increase the accuracy of each step. Runge-Kutte methods make use of this principle. A fourth order Runge-Kutte method uses information from 4 points when calculating each step.

By using the derivative information at the beginning of the interval, the midpoint is calculated. A second midpoint is then calculated using the derivative from the first midpoint, but starting at the beginning of the interval again. A third point is then calculated by using the derivative at the second midpoint, but again starting at the beginning of the interval. There are now four points from which to calculate the end of the interval; the starting point, the two midpoints and the last point. These are used in a weighted average to calculate the next step. Equation 2.19 shows the relevant equations, note that all quantities but h and t_i are vectors. Figure 2.4, taken from Press *et al.* [2007], illustrates the process in 2D.

$$\begin{aligned}
f(t_i, x_i) &= - \bigtriangledown \phi(x_i) \\
k_1 &= hf(t_i, x_i) \\
k_2 &= hf(t_i + \frac{1}{2}h, x_i + \frac{1}{2}k_1) \\
k_3 &= hf(t_i + \frac{1}{2}h, x_i + \frac{1}{2}k_2) \\
k_4 &= hf(t_i + h, x_i + k_3) \\
x_{i+1} &= x_i + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4 + O(h^5)
\end{aligned} \tag{2.19}$$

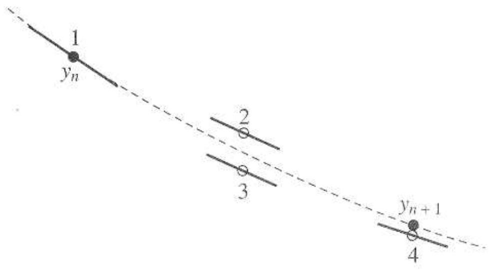


Figure 2.4: "Fourth-order Runge-Kutte method. In each step the derivative is evaluated four times: once at the initial point, twice at trial midpoints, and once at a trial endpoint. From these derivatives the final function value (shown as a filled dot) is calculated." [Press *et al.* 2007, Figure 17.1.3]

Error Estimates

It is useful for the ODE integrator to monitor its progress, making changes to the step-size so that the process achieves at least some specified accuracy, but with the smallest possible computational effort. As the streamlines pass through "smooth uninteresting countryside" the steps should be large, otherwise the system should use small steps so that it "tiptoes through treacherous terrain" [Press *et al.* 2007]. In order to implement step-size control, the process needs to keep track of some estimate of the truncation error.

According to Press *et al.* [2007] the most straightforward technique for estimating the truncation error is through step doubling. This approach takes each step twice; once as a full step and once as two half-steps. This ultimately provides the accuracy of the smaller half-steps with an additional overhead cost of 1.375 [Press *et al.* 2007]. For a fourth-order Runge-Kutte method outlined in Equation 2.19 we have:

$$y(x+h) = y^* + O(h^5) + \cdots \tag{2.20}$$

where y^* is the approximate solution at $x + h$.

Let h be the half-step-size and let $y(x + 2h)$ be the exact solution at the end of the step.

Then where y_1 and y_2 are the approximate solutions at $x + 2h$ using the full and half steps respectively:

$$y(x+2h) = y_1 + (2h)^5\phi + O(h^6) + \cdots \tag{2.21}$$
$$y(x+2h) = y_2 + 2(h^5)\phi + O(h^6) + \cdots \tag{2.22}$$

where ϕ is constant, up to $O(h^5)$, over the step [Press *et al.* 2007]. Equation 2.21 contains $(2h)^5$ as $2h$ is the step-size. On the other hand Equation 2.22 contains $2(h^5)$, as the error for each step is $h^5\phi$. This follows from Equation 2.20 and the Taylor series expansion of y. The difference between the two numerical estimates can be used as an indicator of the truncation error mentioned earlier. [Press *et al.* 2007]

$$\Delta \equiv y_2 - y_1 \tag{2.23}$$

Dormand-Prince Error Estimates

Step doubling is superseded by a "more efficient step-size adjustment algorithm based on embedded Runge-Kutte formulas" [Press *et al.* 2007]. Runge-Kutte formulae of orders M higher than 4 require more than M function evaluations. Fehlberg discovered a fifth-order method, requiring 6 function evaluations. By combining these same 6 evaluations in a different manner one can construct a fourth-order method – the embedded formula. These two estimates can then be used as an estimate of the error. The general form of the fifth-order Runge-Kutte follow in Equation 2.24. [Press *et al.* 2007]

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + c_2 h, y_n + a_{21} k_1) \\ &\cdots \\ k_6 &= hf(x_n + c_6 h, y_n + a_{61} k_1 + \cdots + a_{65} k_5) \\ y_{n+1} &= y_n + b_1 k_1 + b_2 k_2 + b_3 k_3 + b_4 k_4 + b_5 k_5 + b_6 k_6 + O(h^6) \end{aligned} \tag{2.24}$$

The embedded fourth-order formula is

$$y^*_{n+1} = y_n + b^*_1 k_1 + b^*_2 k_2 + b^*_3 k_3 + b^*_4 k_4 + b^*_5 k_5 + b^*_6 k_6 + O(h^5) \tag{2.25}$$

The coefficients for these equations were calculated by Dormand and Prince and are provided in Table 2.1 taken from Press *et al.* [2007, pg 913]. This results in the error estimate in Equation 2.26.

$$\Delta \equiv y_{n+1} - y^*_{n+1} = \sum_{i=1}^{6} (b_i - b^*_i) k_i \tag{2.26}$$

Table 2.1: Dormand-Prince 5(4) Parameters for Embedded Runge-Kutta Method [Press *et al.* 2007]

i	c_i	a_{ij}						b_i	b^*_i
1								$\frac{35}{384}$	$\frac{5179}{57600}$
2	$\frac{1}{5}$	$\frac{1}{5}$						0	0
3	$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$					$\frac{500}{1113}$	$\frac{7571}{16695}$
4	$\frac{4}{5}$	$\frac{44}{45}$	$-\frac{56}{15}$	$\frac{32}{9}$				$\frac{125}{192}$	$\frac{393}{640}$
5	$\frac{8}{9}$	$\frac{19372}{6561}$	$-\frac{25360}{2187}$	$\frac{64448}{6561}$	$-\frac{212}{729}$			$-\frac{2187}{6784}$	$-\frac{92097}{339200}$
6	1	$\frac{9017}{3168}$	$-\frac{355}{33}$	$\frac{46732}{5247}$	$\frac{49}{176}$	$-\frac{5103}{18656}$		$\frac{11}{84}$	$\frac{187}{2100}$
7	1	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$-\frac{2187}{6784}$	$\frac{11}{84}$	0	$\frac{1}{40}$
$j =$		1	2	3	4	5	6		

Adaptive Step-size

By adjusting h, the estimated truncation error Δ can be kept within some predetermined range. The upper bound of Δ ensures that the error does not grow too large, while the lower bound ensures that the process achieves the required accuracy without unnecessarily small step-sizes.

If Δ is too large, then the current step is thrown away and h is decreased according to the size of the error. Alternately, if Δ is too small, the current step can be included, but the h can be increased for the next step. By increasing the step-size whenever possible, the algorithm proceeds faster in smooth areas.

Remembering that both y and Δ are in fact vectors, there are a number of ways to measure the 'size' of the error. A few methods include focusing on summary data such as the norm of Δ, while others look at correcting individual components one at a time. The implementations developed in this research use the ratio of the current error over the desired error to adjust the step-size. A more detailed look at adaptive step-size algorithms can be found in Press *et al.* [2007].

2.7 High Performance Computing

High Performance Computing (HPC) focuses on the modification of the standard computing model to allow for the exploitation of modern advances in hardware. Algorithms should be designed to be fast, efficient and scalable; allowing for larger and larger simulations as resources are increased.

While clock speeds have increased dramatically in the past, there are physical limitations to the speed at which a circuit can operate. Many factors are limited by the speed of light, a barrier that is already affecting many components. Transistor sizes continue to shrink allowing hardware designers to increase the density of their chips, but ultimately they will reach a size where quantum effects may become problematic. On the other hand, power consumption and cooling issues will continue to become more difficult to handle as chip density increases [Tanenbaum 2006].

Due to these physical limitations, architectural changes are being introduced to increase the effective computation speed of machines. Parallel computation, rather than just serial execution of instructions has and is becoming more prevalent and important. To a large degree, computer manufacturers have been making use of parallel technology on a sub-instruction level for quite some time in the form of instruction pre-fetching, pipelining and the use of co-processors [Stallings 2013]. In HPC the focus has largely shifted from micro-parallelism within a single processor to parallelism involving multiple processing units or *cores* working together to solve problems with requirements significantly more extensive then those completed on single cores. HPC finds its primary computing gains through the use of parallel architectures.

There are four types of systems, namely:

1. Single instruction, single data (SISD)
2. Single instruction, multiple data (SIMD)
3. Multiple instruction, single data (MISD)
4. Multiple instruction, multiple data (MIMD)

The SISD approach is the conventional, sequential computing approach. There is a single processor that executes its instruction on some single data stream. SIMD approaches have some single instruction that is executed simultaneously across a number of processing elements. These instructions are executed in lockstep and each processing element is allocated a specific block of memory upon which to operate. Vector and array processors are examples of SIMD systems. In an MISD system, some data is streamed to a number of processors, each executing its own instruction. Finally, an MIMD system has a number of processors that can simultaneously execute a number of different instructions on a number of different data streams. Examples of such systems include multicore computers (SMP) and clusters. [Stallings 2013]

This research falls in the realm of *Multiple Instruction, Multiple Data (MIMD)* computing as the relevant algorithms can be broken down into separate tasks operating on different data synchronously.

This format lends itself to a highly flexible style of parallel computing through the use of multicore (SMP) and multicomputer systems (clusters).

Symmetric multiprocessing (SMP) machines provide a single computer where there are multiple processors that share the same main memory. In this architecture the processors can communicate quickly as there is the benefit of small distances and extremely fast interconnections among them. A multicore machine is similar in that the system reports more processors, but these extra processing elements can be located on the same processor rather than on separate chips. This allows a single processor to handle more than one thread at the same time. It is common to find multiple, multicore processors in a single machine as well.

Another approach for MIMD processing is through the use of clusters. A cluster is a collection of computers – each with its own memory, processing and network capabilities – made in a way that allows a program to run across all of the machines simultaneously. Stallings [2013, pg 633] note that by using commodity computers as the building blocks of the cluster "it is possible to put together a cluster with equal or greater computing power than a single large machine, at much lower cost." The processors communicate by passing messages across the cluster's network.

The benefit of a multicore approach is that communication between processors is extremely fast, especially when compared to the slow network speeds for communication over a cluster. Multicore computers, however, do not scale as well as clusters, becoming more complex and more difficult to build and cool as the number of cores grow. Clusters are usually built as a collection of multicore computers. The systems used to conduct this research are clusters built from multicore nodes.

Clusters lend themselves to an inherently distributed model of memory and there are physical boundaries between different sections of memory. As a result, the distributed memory model with message passing (MPI) was used in the development of this system. As the nodes of the cluster are multicore machines, a shared memory model could be implemented on each node while they communicated with message passing. This hybrid approach could provide a more efficient use of the memory in the cluster. This is beyond the scope of this research and an implementation using the hybrid memory approach is considered for future work.

2.8 Developing Parallel Applications

2.8.1 Amdahl's Law

Not all computational problems can be parallelised effectively. There is a vast difference in how effectively a problem can be solved when dividing it into smaller pieces. There are problems that are inherently serial, where each step relies on the result from the previous step. These problems cannot be divided up among a number of processors. On the other hand, there are so called *embarrassingly parallel* problems. These problems can be heavily divided up into a large number of small parts which can all be solved at the same time on separate processors, without the need to communicate.

In practice, most applications fall somewhere between these two extremes with some sections that need to be run in serial and others that can run freely on as many processors as are available. In this case, all the processors must wait for the serial section of the code to finish before they continue. This concept is captured in Definition 2.4 – Amdahl's Law, cited from Stallings [2013].

Definition 2.4 (Amdahl's Law) *Consider a program that runs on a single processor such that it spends some fraction $(1-f)$ of the total time in inherently serial code. On the other hand a fraction f of the total time is spent in* infinitely parallelisable *code that introduces no overhead cost.*

Let T be the execution time of the program run on a single processor.

$$T = T \cdot (1-f) + T \cdot f \tag{2.27}$$

Then the amount of time taken to run on N parallel processors is S.

$$S = T \cdot (1-f) + \frac{T \cdot f}{N} \tag{2.28}$$

The speed-up is measured as shown in Equation 2.29.

$$\begin{aligned} \textit{Speedup} &= \frac{\textit{Time to execute on a single processor}}{\textit{Time to execute on multiple processors}} = \frac{T}{S} \\ &= \frac{T \cdot (1-f) + T \cdot f}{T \cdot (1-f) + \frac{T \cdot f}{N}} \\ &= \frac{1}{(1-f) + \frac{f}{N}} \end{aligned} \tag{2.29}$$

Two conclusions follow:

1. *When f is small, the use of parallel processors has little effect.*
2. *As N approaches ∞, the speed-up is bound by $1/(1-f)$ due to the time spent in the serial portions of code.*

Amdahl's Law for multiprocessors is illustrated in Figure 2.5. This shows how the number of processors affects the overall speed-up based on how big a fraction of the code is parallelisable. An analysis of the mapping algorithm in terms of Amdahl's Law is not provided as measurements of f are generally artificial. An analysis of which parts of the method are parallelisable is provided in Chapter 5, while the effect of the number of processors is analysed in Chapter 6.

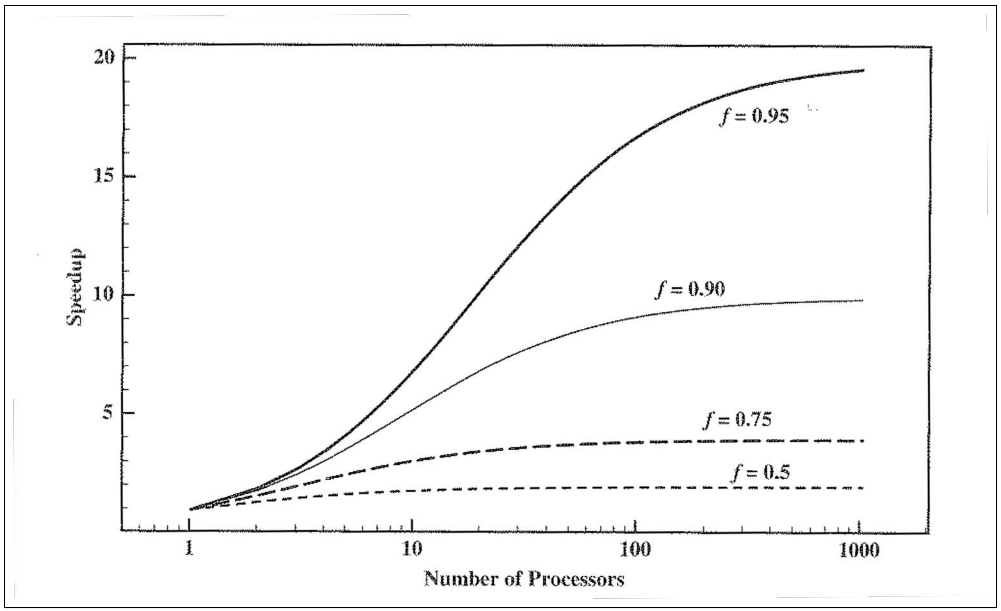


Figure 2.5: Amdahl's Law [Stallings 2013, Figure 2.14]

2.8.2 Message Passing Interface (MPI)

In a cluster environment the code running on each processor needs to be able to communicate. There is a standard Message Passing Interface (MPI) that handles many of the problems involved in this communication. The MPI system is setup beforehand and knows what computers are part of the cluster as well as the number of cores on each of these machines. When compiled, the user's program must link to the MPI libraries. MPI is then used to launch the program on the master node. MPI then connects to each slave node in the cluster and launches the user program.

Each instance of the program is assigned a unique rank. Processes can communicate with one another by calling the relevant MPI library routines. These routines work within some *MPI communicator*. These communicators keep track of the topology of the cluster and simulate a linear array of machines, a grid, a ring etc. The communicator used the most is `MPI_COMM_WORLD`, which contains every process labelled linearly.

The basic MPI routines allow the programmer to query the MPI communicator and send and receive data. There are blocking and non-blocking versions of all of the send and receive routines. The MPI function prototypes for the most common commands are outlined below, every function returns 0 or an integer error code.

- `int MPI_Init(int *argc, char ***argv) / MPI_Finalize()`
  - Called at the beginning and end of the program respectively. This initialises the MPI environment, creating all the relevant variables and finally shutting down and freeing all the MPI variables.
- `int MPI_Comm_size(MPI_Comm comm, int *size)`
  - Requests the number of processes running in the given communicator.
- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
  - Requests the unique rank of the current processes in the given communicator.

-
  ```
  int MPI_Send(void* buffer, int count, MPI_Type type, int dest
               int tag, MPI_Comm comm)
  ```
  - Sends the data in the buffer to the process with the relevant rank. This is a blocking send and the function does not return until the other process has received the data.

-
  ```
  int MPI_Isend(void* buffer, int count, MPI_Type type, int dest
                int tag, MPI_Comm comm, MPI_Request *request)
  ```
  - Sends the data in the buffer to the process with the relevant rank. This is a non-blocking send and the function returns immediately.

-
  ```
  int MPI_Recv(void* buffer, int count, MPI_Type type, int source,
               int tag, MPI_Comm comm, MPI_Status *status)
  ```
  - Blocking receive. The function returns when the relevant data has been received from the source process.

-
  ```
  int MPI_Irecv(void* buffer, int count, MPI_Type type, int source,
                int tag, MPI_Comm comm, MPI_Request *request)
  ```
  - Non-blocking receive. The requests MPI to receive a message from the source process in the background. The `request` variable can be queried to check the status of the operation.

As seen in the functions above, MPI uses its own datatypes so that functions remain portable across the network. These functions are the basic routines required to use MPI effectively. The full standard is, however, advanced and allows the programmer a great deal of flexibility.

2.8.3 The Portable Extensible Toolkit for Scientific Computation (PETSc)

PETSc is a library built upon the Message Passing Interface (MPI) and Basic Linear Algebra Subprograms (BLAS) [Balay *et al.* 1997 2011 2012]. BLAS is an API providing an interface for linear algebra implementations. There are number of different implementations of this interface, all of which should work with the PETSc library.

The PETSc library provides parallel methods that use MPI for communication. It provides parallel data structures for sparse and dense matrices as well as vectors. These structures are used extensively in the calculation of the potential field required in the mapping algorithm.

As mentioned in Section 2.6.1, the matrices produced by the finite difference method are highly sparse. This is beneficial as the $n \times n$ matrices can be represented in a way that uses significantly less space than the standard $O(n^2)$ array representation. PETSc provides a number of matrix formats, some of which include compressed sparse row (AIJ), block compressed sparse row (BAIJ), symmetric block compressed row (SBAIJ) and dense. Sequential implementations of these formats are also provided by the library. [Balay *et al.* 2011]

The following descriptions of both the sequential and parallel formats are adapted from Balay *et al.* [2011]. In a sequential AIJ sparse matrix, the non-zero elements are stored by each row along with an array containing the corresponding column numbers. When two rows have the same non-zero structure, these 'column arrays' are reused. Thus matrix A can be represented as shown in Figure 2.6, where arrows represent pointers.

$$A = \begin{pmatrix} 0 & 5 & 2 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 \end{pmatrix} \tag{2.30}$$

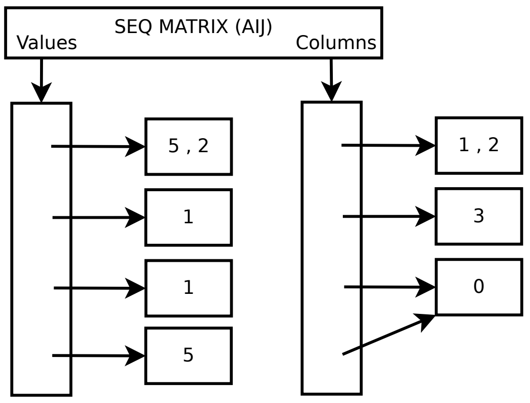


Figure 2.6: Sparse Sequential Matrix Memory Organisation

This representation has a space complexity of $O(n + nz)$, where nz is the number of non-zero elements in the matrix and n is the number of rows. When the matrix is sparse ($nz << n^2$) this representation is significantly more space efficient.

When working with parallel matrices each process is assigned a number of rows to be stored locally. All columns are stored locally. The developer has control over which rows are allocated to each process. If this is not specified, PETSc automatically divides the n rows evenly between the m processes with the process of MPI rank 0 receiving the first n/m rows. While the basic representation is the same as that shown in Figure 2.6, the library keeps track of on- and off-diagonal blocks separately. A more detailed explanation of the parallel representation is described in Balay *et al.* [2011, pg 57].

$$A\vec{x} = \vec{b} \tag{2.31}$$

PETSc provides both sequential and parallel, sparse and dense vector objects that work in similar ways to the sparse matrices above. Once the matrix A and vector $\vec{b}$ have been constructed, Equation 2.31 can be solved using various parallel linear solvers. The primary linear solvers provided with PETSc are Krylov Subspace (KSP) methods with various pre-conditioners. This is considered to be "at the centre of most modern numerical codes for the iterative solution of linear systems." [Balay *et al.* 2011]

PETSc provides, among others, Richardson, Chebychev, Conjugate Gradient, BiConjugate Gradient, GMRES and many other KSP methods. More information on the KSP methods in the PETSc library can be found in Balay *et al.* [2011, pg 71]. Saad [1989] notes how KSP methods are general purpose iterative methods that became popular following the "discovery and popularization of preconditioning techniques."

$$M^{-1}Ax = M^{-1}b \tag{2.32}$$

Preconditioning involves replacing the original linear system, A with some equivalent system as shown in Equation 2.32. Often M is of the form $M = LU$ where L and U are lower and upper triangular matrices with the same structure as the lower and upper triangular parts of A respectively [Saad 1989]. In this research the incomplete LU factorization of A is used as the pre-conditioner.

Once the system is preconditioned, the method uses some initial guess x_0. This initial guess is then updated by a projection into an affine subspace, $x_0 + L_m$. In KSP methods L_m must be a Krylov subspace. Further details of this method are beyond the scope of this research but can be found in Saad [1989]. Saad [1989] notes how KSP methods are particularly effective for problems arising from elliptic partial differential equations. As Laplace's equation is an elliptic PDE, this makes KSP methods particularly well suited for use in the mapping process.

Chapter 3

Research Questions and Methods

3.1 Aims

This research aimed to understand how harmonic functions apply to volumetric parametrisation as well as domain mapping in general. The work of Voruganti *et al.* [2006 2008] was to be implemented in parallel for an arbitrary number of dimensions. Implementations should run on clusters at the *University of the Witwatersrand* and the *Centre for High Performance Computing* and should be parallel and scalable. The 2D methods were extended to work on domains that contain voids. The effect of various boundary conditions for voids was to be investigated in an attempt to allow the method to handle compact source domains of genus-$\emptyset$. When handling these genus-$\emptyset$ domains the aim was to find both boundary conditions that maintain bijectivity, mapping the void to the target domain, as well as conditions that shrink the voids to arcs and/or points in the target domain.

3.2 Research Hypothesis

By applying the correct boundary conditions, the methods set out in Voruganti *et al.* [2006 2008] can be extended to work on source domains that contain voids. This can be done in a manner that is continuous and bijective over the source domain so that the topology is preserved. Alternatively, by applying different boundary conditions to the voids and relaxing bijectivity constraints of the map, the topology of the target domain can be altered by shrinking or eliminating voids.

The methods can be implemented for dimensions higher than those tested in Voruganti *et al.* [2006 2008] and can be parallelised to run on multi-processor systems. This parallelism should enable the system to process larger domains, with higher dimensionality as more hardware becomes available.

3.3 Research Questions

1. How do different boundary conditions, such as Dirichlet and Neumann conditions affect the mapping and can they be used to distort the shape of the map?
2. What effects do various boundary conditions have on the 2D method when the target domain is not topologically equivalent to a disk?
3. Which parts of the system should be parallelised to increase efficiency?

3.4 Methodology

The research was completed in a number of phases. The initial phase focused on the development of a 2D, serial prototype in MATLAB. Correct results from this program ensured that the method was properly understood and gave a first estimate as to the computational requirements of the method.

When this implementation was completed, the mathematical methods required to handle voids became the focus. These methods were tested by adapting the MATLAB implementation. The implementation and results of these methods are presented in Section 4.4.

After the development and testing of these methods in MATLAB, the 2D system was implemented in serial in C++ and PETSc. Visualisation was done using gnuplot. This code was then adapted to use MPI and the parallel capabilities of PETSc to allow it to run across a number of cluster nodes. The code was finally extended to work in 3D and ultimately in n-dimensions. The description of the code as well as the parallelisation of the algorithm is discussed in Chapters 5.

Finally, the code was run for both two and three dimensional applications – robotic path planning as well as meshing. These results and the resources used are reported in Chapter 8.

Chapter 4

2D Domains

4.1 Introduction

This chapter first describes the implementation of the 2D method from Voruganti *et al.* [2006] for genus-0 domains. Following that, genus-$\emptyset$ domains are discussed and the extensions to the original method are presented. Parts of this chapter appeared in Klein *et al.* [2012].

4.2 2D Genus-0 Domains

The method from Voruganti *et al.* [2006] is now presented. It was implemented in serial code in MATLAB and formed the basic prototype upon which the extensions to the method could be tested. MATLAB provides an advanced and highly flexible *Just-In-Time* (JIT) compiled language that allows rapid prototyping in mathematical domains. Visualisation was initially coded alongside the calculations, this however slowed down each simulation significantly – primarily due to memory constraints and continuous updates of the output images as new data became available. For this reason the most efficient code separated these two phases for each simulation. At logical breaks in the code, the relevant data was saved to the hard drive. When MATLAB had finished all of the calculations, this data was rendered by gnuplot.

4.3 Algorithm and Implementation

This section is a more detailed explanation of the algorithm described in Section 2.4 for 2D domains, along with notes concerning its implementation in MATLAB. Full pseudo code is provided in Algorithm 1 of Appendix A.

4.3.1 Step 1: Read the Domain Geometry

The method is not tied to any specific representation of the source domain. In the 2D case however, it is intuitive to use a digital image to represent grid points that are either part of the domain, on the boundary or outside of the domain. An example of such an image is shown in Figure 4.1

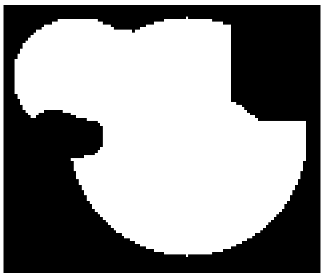


Figure 4.1: Non-convex Source Domain of Genus-0

The image can be read in using the `imread` function in MATLAB. This function reads the image into a 3D matrix where there are three layers each representing the RGB intensity values respectively. The image matrix is then converted to a binary image matrix using the `im2bw` function. There is now a 2D logical matrix of ones and zeros, where ones represent a point inside the source domain. In Figure 4.1 the source domain is the white shape.

Once this logical matrix has been constructed the `bwboundaries` routine searches for edges in the binary image. This boundary is shown in red in Figure 4.2. Some *shape centre* must be chosen. It is noted that the centre of mass of the domain cannot be used as it is not necessarily inside the domain. In this code the centre is chosen by selecting the point furthest from any boundary point. A good choice of shape centre affects the numerical accuracy of the map. A shape centre that lies too close to the boundary causes the gradients of the potential field to be extremely steep on the side close to the boundary and extremely flat on the other side. This distorts the map disproportionately causing one side to expand and the other to contract significantly. This causes a number of numerical issues. By choosing a shape centre as far from the domain boundary as possible, these numerical problems can be avoided.

Figure 4.3 shows the distance of each interior point from the boundary. The distances from the boundary of the domain are calculated by using an adaptation of the Floyd-Warshall Shortest Path Algorithm. The point furthest from the boundary has proven to be a good heuristic when selecting the shape centre.

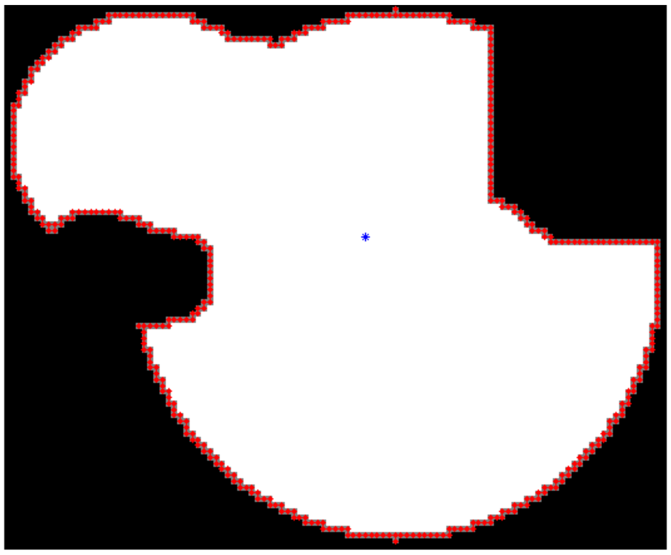


Figure 4.2: Non-convex Source Domain of Genus-0 with boundary and centre highlighted.

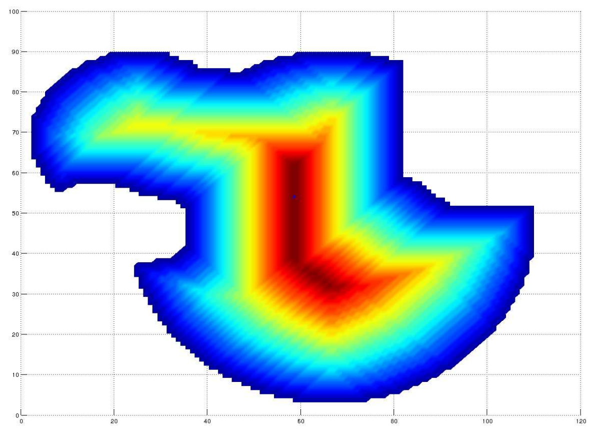


Figure 4.3: Non-convex Source Domain – Distance from Boundary

4.3.2 Step 2: Solve for Potential Field

At this stage the system needs to solve the Laplace equation boundary value problem. Equations 2.7 and 2.16 result in a number of linear equations. These equations are used to calculate the potential value at each grid point inside the domain. The boundary and centre points are set to 1 and 0 respectively. For example, where the grid points are arranged as in Figure 4.4, the matrix in Equation 4.1 shows the relevant linear equations for potential values ϕ_0 to ϕ_4 where all but ϕ_2 are boundary points. The centre point value is set in the same way as the boundary points.

	ϕ_0	
ϕ_1	ϕ_2	ϕ_3
	ϕ_4	

Figure 4.4: Pixel Arrangement for Finite Difference System in Equation 4.1.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & \\ 0 & 1 & 0 & 0 & 0 & \\ 1 & 1 & -4 & 1 & 1 & \cdots \\ 0 & 0 & 0 & 1 & 0 & \\ 0 & 0 & 0 & 0 & 1 & \\ & & \vdots & & & \ddots \\ & & & & & \end{bmatrix} \begin{bmatrix} \phi_0 \\ \phi_1 \\ \phi_2 \\ \phi_3 \\ \phi_4 \\ \vdots \\ \\ \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ \vdots \\ \\ \end{bmatrix} \tag{4.1}$$

The system is then solved using MATLAB's build-in matrix solvers. The solution of this system provides a potential value for each point on the grid. The resulting distribution is usually flat close to the boundaries; this means that points close to the boundary are disproportionately close to the boundary in the target domain. To mitigate these effects the potential is scaled using a formula such as Equation 4.2, where n is some scaling parameter. Note that the potential is scaled in a way that ensures the range of $[0; 1]$. The potential distribution with contour lines for the domain in Figure 4.2 is shown in Figure 4.5.

$$\phi_{ij} \leftarrow \frac{\phi_{ij}^n}{\max(\phi)^n} \tag{4.2}$$

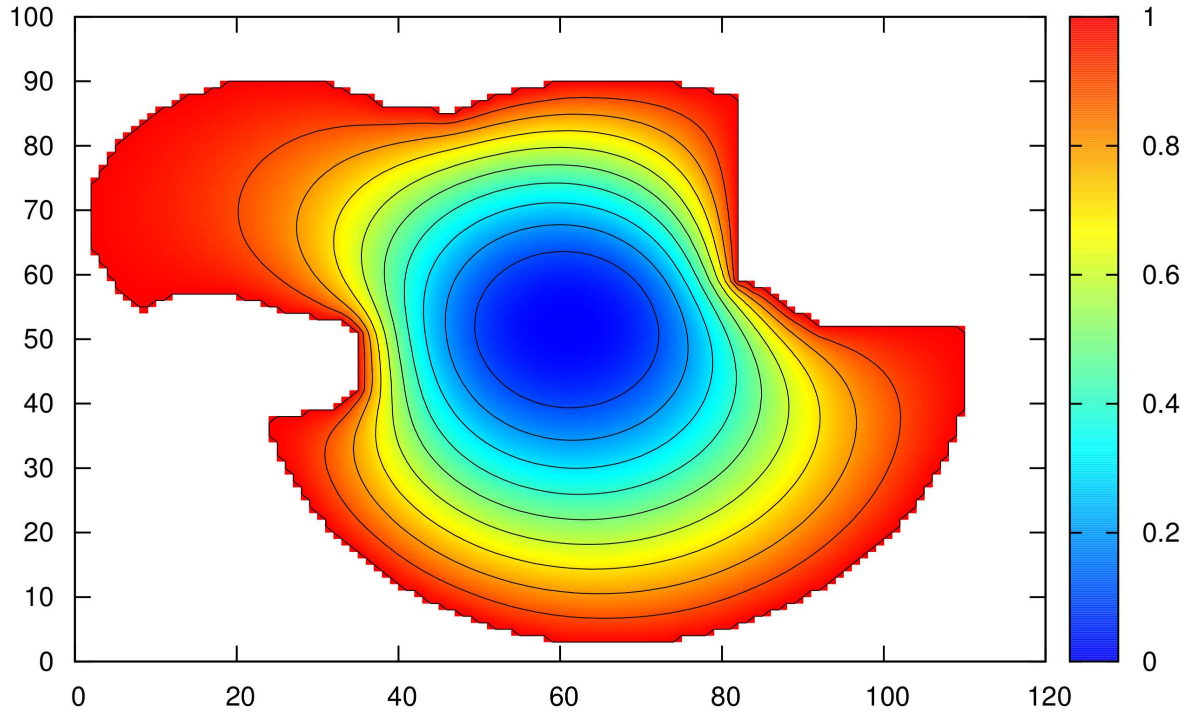


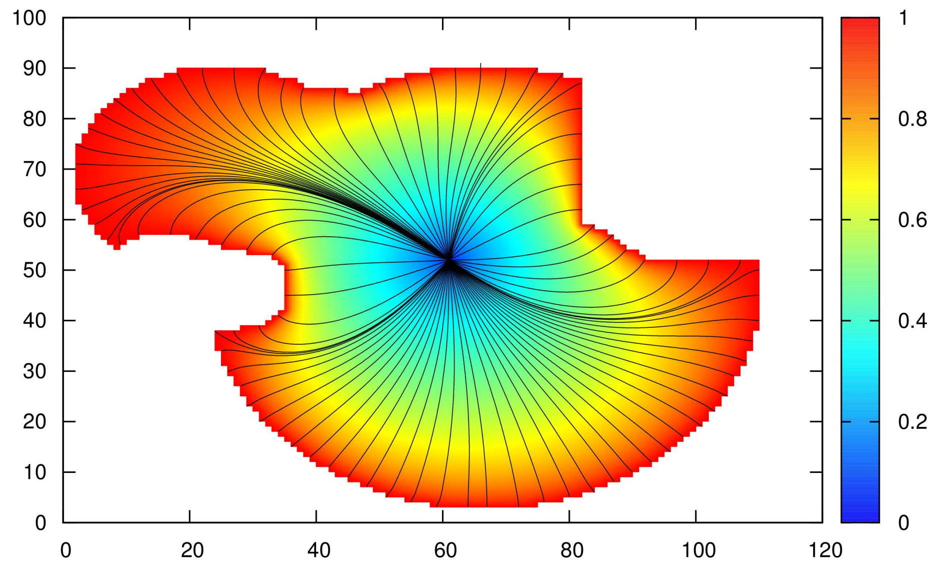
Figure 4.5: Scaled Potential Distribution for the Source Domain in Figure 4.2

4.3.3 Step 3: Calculate Streamlines and Parametrisation

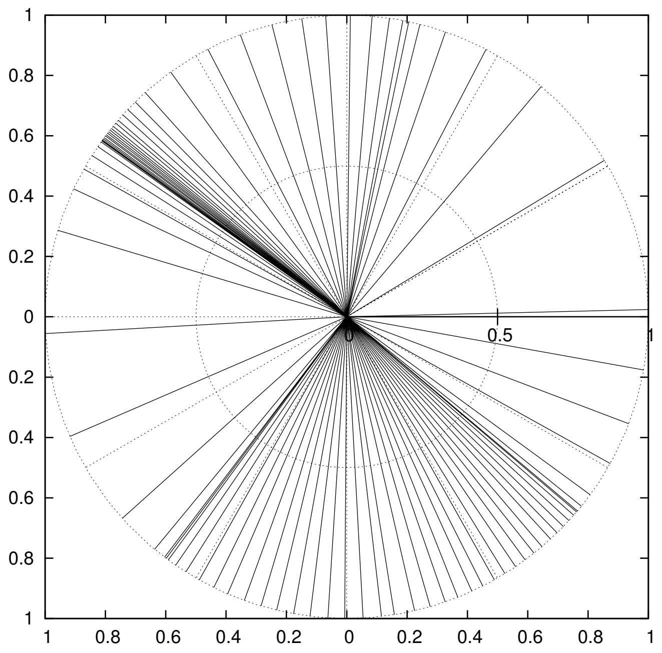
Once the potential distribution has been calculated, streamlines beginning at each of the boundary points are calculated. In MATLAB this can be calculated by the `stream2` function. The streamlines run from the domain boundary towards the shape centre. The streamlines never insect. All points along a single streamline are parametrised with the same polar angle – the angle at which the streamline approaches the shape centre. The potential value serves as the distance from the centre in the polar co-ordinate system. Figure 4.6 on the next page shows this mapping process. The streamlines are traced from the boundary towards the centre in Figure 4.6(a). The points along these streamlines are illustrated in Figure 4.6(b) where all points along a streamline are parametrised with the same polar angle and the distance of a point along the radial lines from the centre is set according to its potential.

At some set distance away from the shape centre the streamlines are stopped. This allows the angle of approach to be calculated more accurately. As long as the area where streamlines stop is convex and wholly inside the source domain the properties of the map are not affected. For example, they could be stopped at the innermost contour in Figure 4.5.

The points inside this area also need to be associated with some angle. The streamline cannot be used as the streamline has been stopped and does not continue over this area. Any method that connects the end of the streamline to the shape centre in a continuous and smooth manner that preserves the ordering without overlapping is appropriate. In the current implementation, the points in this area are parametrised using a straight line. Future work should focus on an interpolant that connects the centre to the streamline smoothly.



(a) Potential Value Distribution with every tenth streamline.



(b) Streamlines shown in the target domain.

Figure 4.6: Domain Mapping Example

4.4 2D Genus-$\emptyset$ Domains

The remainder of this chapter focuses on extending the above method to handle domains that contain voids in 2D. The source domain must remain connected, although it need not be simply connected. This relaxation allows the domain to contain any number of *holes* or *voids*.

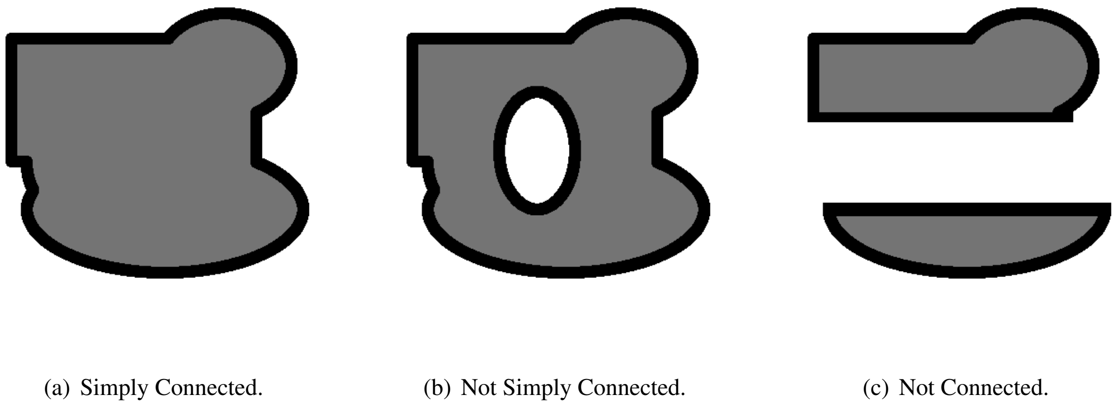

(a) Simply Connected. (b) Not Simply Connected. (c) Not Connected.

Figure 4.7: Domain Connectedness

A naïve approach is to simply ignore the voids and treat the domain as simply connected. Once the map has been completed, the boundaries of the voids are mapped to the target domain and those areas are removed from the target domain. This method does not exert any constraints on the voids and the shape of each void in the target domain is arbitrary. It is preferable to control the shape of the voids in the target domain in a way that assists the user with the final problem. Another approach when there is a single, convex void is to treat the entire void as the shape centre. In other cases this method fails. The purpose of this chapter is to construct an algorithm that forces the void to maintain certain properties in the target domain, while ensuring that the map retains the necessary properties mentioned in earlier chapters.

The overall method remains the same with the addition that boundary conditions are applied to the voids when calculating the potential field. Specifically, the aim is to change the flow of the potential field so that the void becomes manageable in the target domain. To aid understanding of these additional conditions, the potential field can also be interpreted as a solution to the steady state heat equation. Thus, the potential value at a point is comparable to its temperature. In this framework, the centre acts as a heatsink while the domain boundary acts as a heat source. The streamlines now represent the flow of heat in the domain. There are two boundary conditions found to be useful on the boundaries of the voids– a Dirichlet boundary condition where the potential is kept constant and the Neumann boundary condition where a constant potential flux is forced over the boundary. These two methods, along with the relevant implications are discussed below. Both methods are demonstrated using the source domain shown below in Figure 4.8.

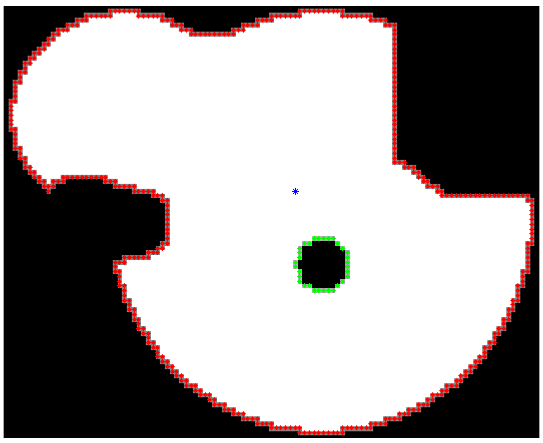


Figure 4.8: Source Domain with 1 Void.

4.5 Void as an Infinite Conductor

By applying a Dirichlet boundary condition to the void one can specify the potential – or "temperature" – along the void boundary. If the void is treated as an infinite conductor, the temperature values on the void boundaries are forced to be equal as shown in Figure 4.9.

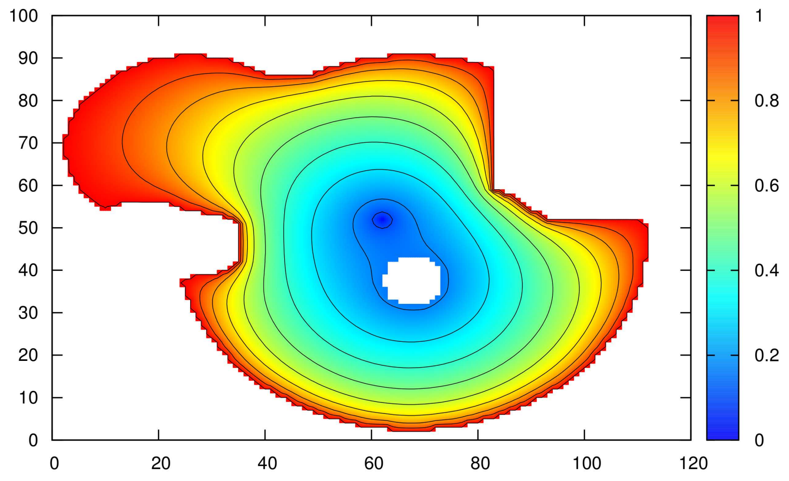


Figure 4.9: Potential Distribution Constant over the Void Boundary.

In the finite difference matrix, this is accomplished by setting all of the points on a void boundary equal to one point on the boundary. The points adjacent to the boundary are handled as normal interior points. Suppose that one encounters the pixel layout in Figure 4.10, where ϕ_i is a point on the interior and α_i is a point on the void boundary.

ϕ_0	ϕ_1	ϕ_2
ϕ_3	α_4	α_5
ϕ_6	α_7	

Figure 4.10: Pixel Arrangement for Finite Difference System in Equation 4.3.

$$\begin{bmatrix} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & \\ 1 & 0 & 0 & -4 & 1 & 1 & 0 & 1 & \cdots \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{-1} & \mathbf{0} & \mathbf{0} & \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{-1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \\ 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 & \\ & & & \vdots & & & & & \ddots \\ & & & & & & & & \end{bmatrix} \begin{bmatrix} \phi_0 \\ \phi_1 \\ \phi_2 \\ \phi_3 \\ \alpha_4 \\ \alpha_5 \\ \phi_7 \\ \alpha_7 \\ \vdots \\ \\ \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ \\ \end{bmatrix} \tag{4.3}$$

The system in Equation 4.3 however is singular because of rows corresponding to α_4 and α_5 marked in bold. While the system specifies that the void boundary potentials are equal, it does not specify what that potential should be, thus there is no unique solution to the system and the matrix becomes singular. The potential value can be selected by the user, as long as it does not introduce any local extrema. This is not straight forward if there are multiple voids that eclipse one another, potentially introducing some extrema between them. It is better that the potential value is set dynamically by the system.

It is sufficient to set one point on the void boundary equal to a neighbouring interior point to fix this issue. For example, α_4 could be set to ϕ_1, changing the system to the one in Equation 4.4. This system is non-singular and can be solved using any standard technique. This forces the entire void boundary to the same potential value, that of ϕ_1, thus moving all these points to the same radial distance from the centre in the target domain. This system results in the potential distribution shown in Figure 4.11.

$$\begin{bmatrix} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & \\ 1 & 0 & 0 & -4 & 1 & 1 & 0 & 1 & \cdots \\ \mathbf{0} & \mathbf{-1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & \\ 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 & \\ & & & \vdots & & & & & \ddots \\ & & & & & & & & \end{bmatrix} \begin{bmatrix} \phi_0 \\ \phi_1 \\ \phi_2 \\ \phi_3 \\ \alpha_4 \\ \alpha_5 \\ \phi_7 \\ \alpha_7 \\ \vdots \\ \\ \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ \\ \end{bmatrix} \tag{4.4}$$

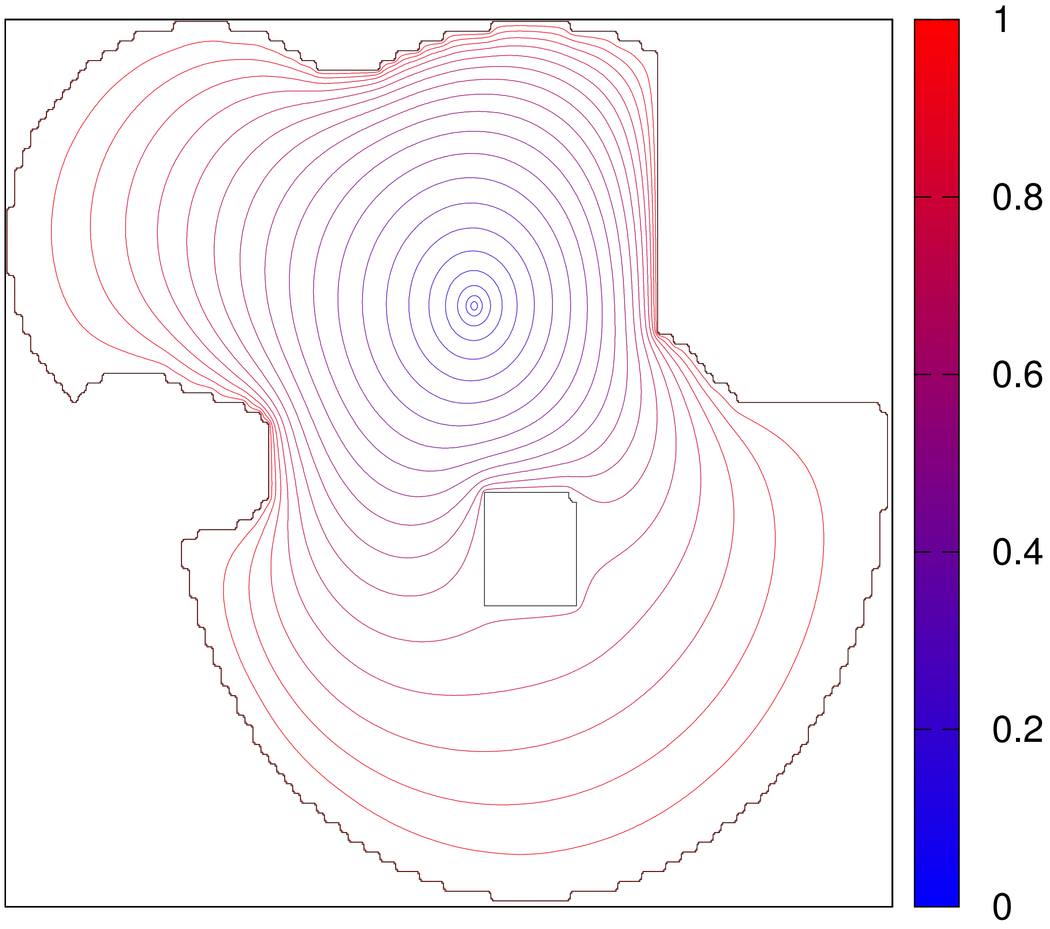


Figure 4.11: Contour plot of the potential field containing a void treated as a *perfect conductor*.

As shown in Figure 4.12, streamlines now flow either from the boundary directly to the centre or from the boundary to one side of the void. The streamlines flowing to the void have then to be restarted on the other side of the void and allowed to reach the centre. These are illustrated in green in the figure. This *streamline restarting* is easily calculated, even when the obstacle is non-convex. First the boundary streamlines are calculated, some of these will flow to the void. The two *extremal streamlines* are the last streamlines that hit the void before they start travelling to the centre. The endpoints of all other streamlines stopping at the void fall between these two points on the void boundary. Streamlines need to be started on the part of the boundary that doesn't fall between these two points. Each streamline that stops at the void should be associated with a streamline beginning on the other side of the void. This can be done in any manner provided that the order of streamlines is preserved. A linear map is used in the implementation – mapping the leftmost streamline to the leftmost restarting point.

There are now multiple streamlines approaching the centre from the void – the green streamlines in Figure 4.12. This results in multiple angles but a single potential value describing the points of the void. Therefore, the void is reduced to a single arc in the target domain. This is illustrated in Figure 4.13, where the blue dots represent boundary points and the green line represent the void boundary. Note that this map no longer preserves the topology as the void is collapsed into a single arc. This is acceptable as the interior points of the void are not relevant. The map remains bijective and smooth in all areas except for the actual void boundary.

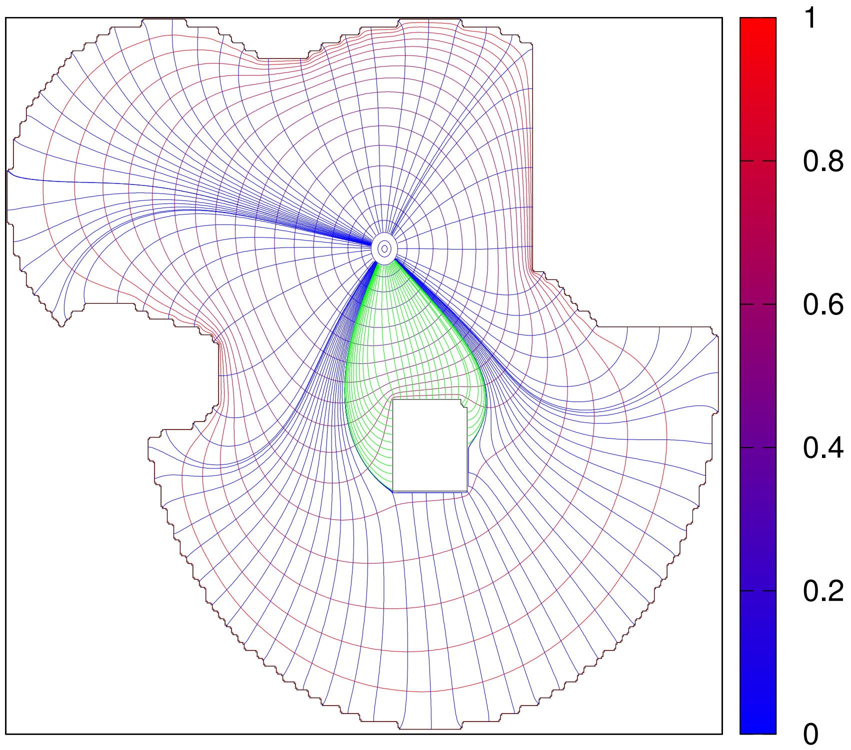


Figure 4.12: Streamline flows when encountering a *perfect conductor*.

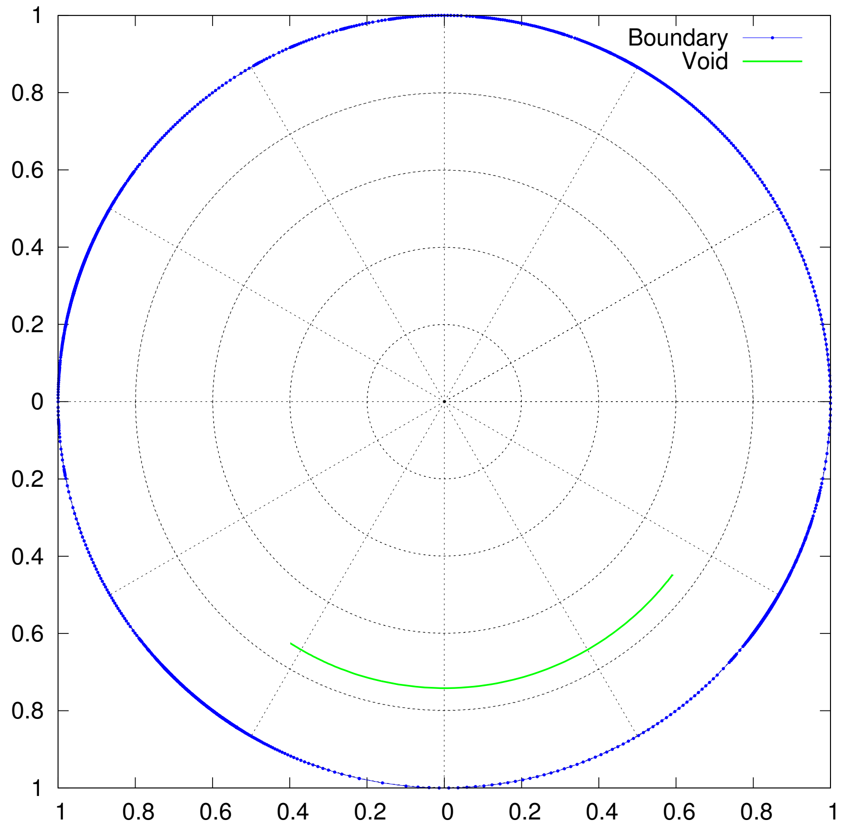


Figure 4.13: Target domain with the void reduced to an arc.

4.6 Void as an Insulator

In the previous section the boundary of the void was forced to maintain a specific potential value. In the approach in this section, a Neumann boundary condition is applied, which forces the system to maintain some constant potential flux across the void boundary. In the analogy of heat transfer, this is equivalent to forcing the system to maintain some constant heat flux across the boundary. When the flux is positive it means that heat is flowing across the boundary, out of the void into the domain interior at some constant rate. On the other hand, a negative value means that heat is flowing from the domain interior into the void at some constant rate. In these cases the void acts as a heat source or heat sink, radiating or absorbing heat respectively. Both conditions introduce local extrema into the potential field and cause problems when calculating the streamlines. When the flux is set to zero, however, no heat is allowed to transfer across the void boundary. In other words, the void becomes an insulator. In this case the Neumann boundary condition results in Equation 4.5 where $\vec{n}$ is a vector normal to the boundary,

$$\vec{n} \cdot \bigtriangledown\phi = 0 \tag{4.5}$$

The potential field differs from the previous method as the contours now intersect the void boundary orthogonally. This is visible in the potential field shown in Figure 4.14. Because the contours are orthogonal to the void boundary, the streamlines must be parallel to the void boundary when in close proximity. This means that the streamlines will pass around the side of the void and continue to the centre. This is discussed in detail after a description of the numerical approach used to calculate this field.

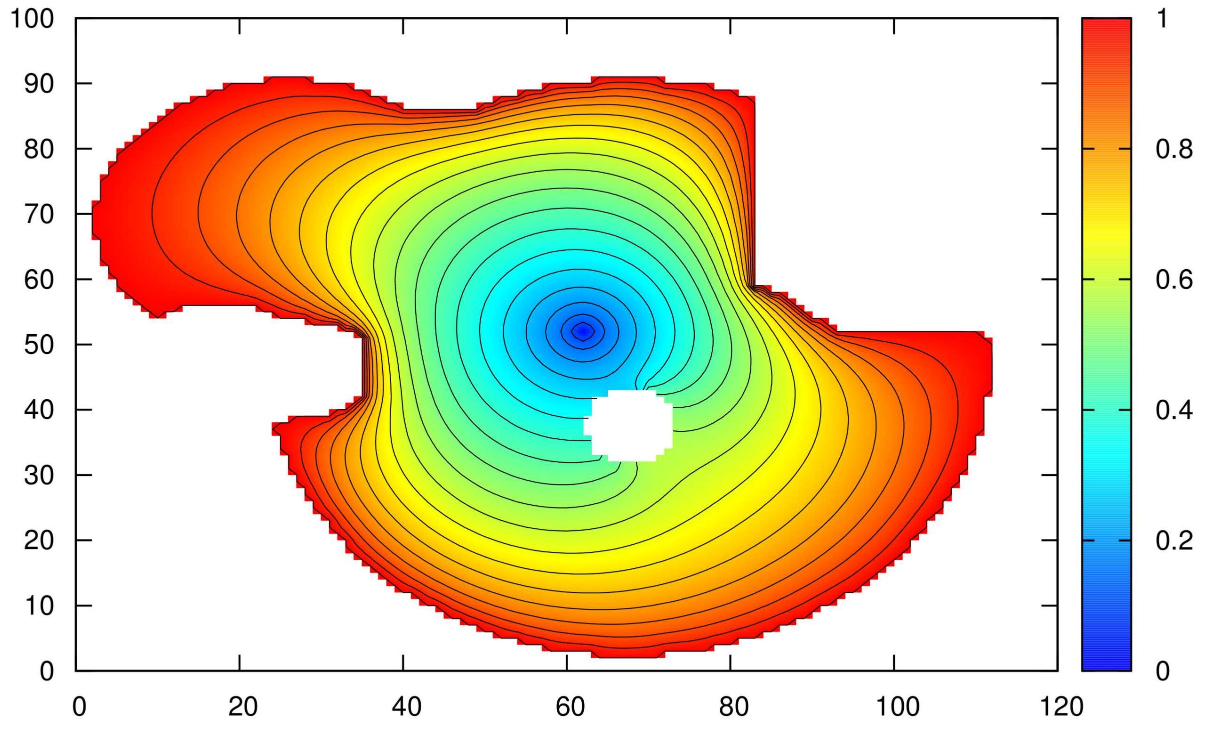


Figure 4.14: Potential Insulated over the Void Boundary

Using finite difference, Equation 4.5 results in the formula in Equation 4.6.

$$\vec{n} \cdot \begin{bmatrix} \phi_{i,j} - \phi_{i-1,j} \\ \phi_{i,j} - \phi_{i,j-1} \end{bmatrix} = 0 \tag{4.6}$$

In the case of a vertical boundary wall, as shown in Figure 4.15 with $\phi_{i,j}$ representing an interior point and $\alpha_{i,j}$ representing a void boundary point, the finite difference equation results in the expression in Equation 4.7 and finally the system in Equation 4.8.

$\phi_{0,0}$	$\alpha_{0,1}$
$\phi_{1,0}$	$\alpha_{1,1}$
$\phi_{2,0}$	$\alpha_{2,1}$

Figure 4.15: Pixel Arrangement for Finite Difference System in Equation 4.8.

$$\begin{aligned} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \cdot \begin{bmatrix} \phi_{i,j} - \phi_{i-1,j} \\ \phi_{i,j} - \phi_{i,j-1} \end{bmatrix} &= 0 \\ \Rightarrow \qquad \phi_{i,j} - \phi_{i-1,j} &= 0 \\ \Rightarrow \qquad \phi_{i,j} &= \phi_{i-1,j} \end{aligned} \tag{4.7}$$

$$\begin{bmatrix} -4 & 1 & 0 & 1 & 0 & 0 & & \\ 1 & -4 & 1 & 0 & 1 & 0 & & \\ 0 & 1 & -4 & 0 & 0 & 1 & \cdots & \\ -1 & 0 & 0 & 1 & 0 & 0 & & \\ 0 & -1 & 0 & 0 & 1 & 0 & & \\ 0 & 0 & -1 & 0 & 0 & 1 & & \\ & & \vdots & & & & \ddots & \\ & & & & & & & \end{bmatrix} \begin{bmatrix} \phi_{0,0} \\ \phi_{1,0} \\ \phi_{2,0} \\ \alpha_{0,1} \\ \alpha_{1,1} \\ \alpha_{2,1} \\ \vdots \\ \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ \end{bmatrix} \tag{4.8}$$

As heat flow across the void is prohibited, the flow is forced around the boundary of the void; thus the streamlines avoid it. This is reinforced by the theoretical result that as the streamlines get closer to the void boundary, they become parallel to it. This is true in all cases except two. The outer contours form a ring between the void and the source domain's boundary. As the potential values get closer to those of the insulated void, the contours begin to intersect the void boundary orthogonally. The same thing happens with the inner contours which form a ring between the void and the shape centre. As the contours approach the void boundary, one inner and one outer contour will form a cusp that is ultimately the first of contact with the void boundary. This is illustrated in Figure 4.16, where a cusp on either side of the void can be seen. The formation of these cusps creates a fixed point on each side of the void. These are the *inner* and *outer* fixed points.

The shape of the potential field forces a single streamline to travel from the boundary of the domain to the outer fixed point. This streamline should then be started as close as possible to the inner fixed point. In practical terms it is easier to trace a streamline backwards from the shape centre to the inner fixed point. All other streamlines in the domain will travel around the void as shown in Figure 4.17.

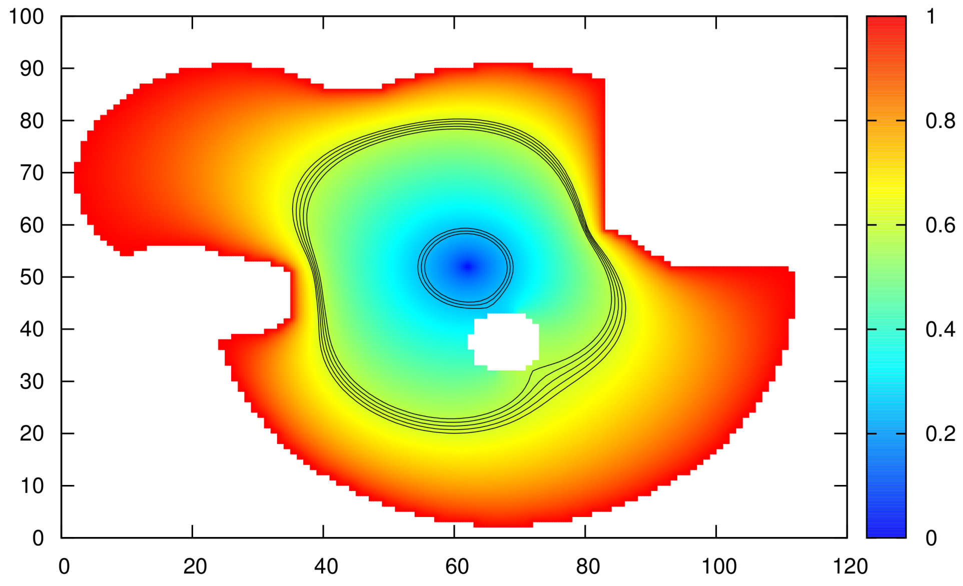


Figure 4.16: Contours indicating the fixed points near the insulated void.

The boundary of the void now has a number of associated potential values, but there is only a single streamline intersecting/originating at the void. Hence, the void is associated with a single angle and multiple potential values. It is therefore reduced to a single radial line in the target domain as shown in Figure 4.18 on the next page. As before, this method has not preserved the topology of the domain as the boundary of the void is collapsed into a single line. Again, this map is considered acceptable as it maintains all the necessary properties of bijectivity, continuity and smoothness except on the void boundary.

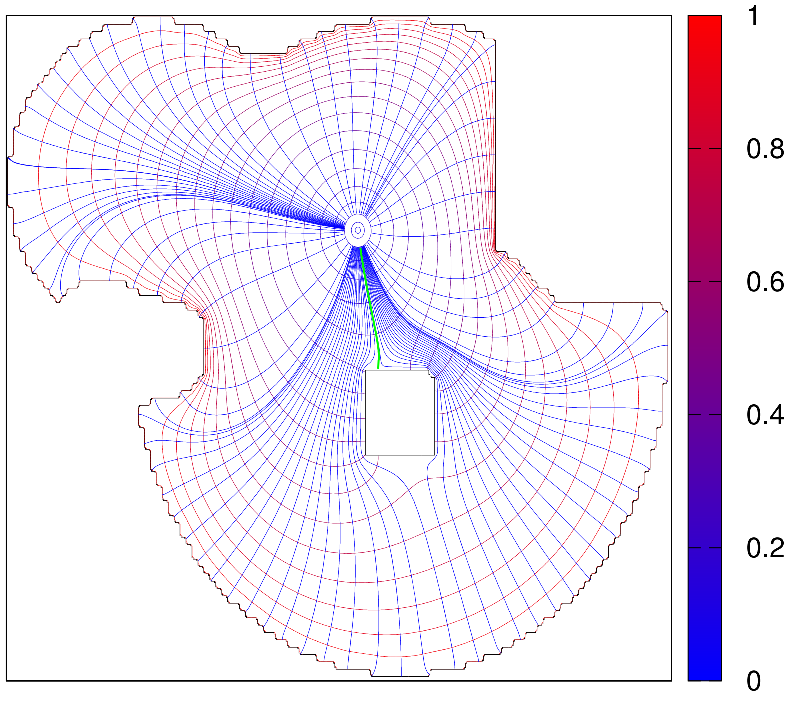


Figure 4.17: Streamline flows around an *insulator*.

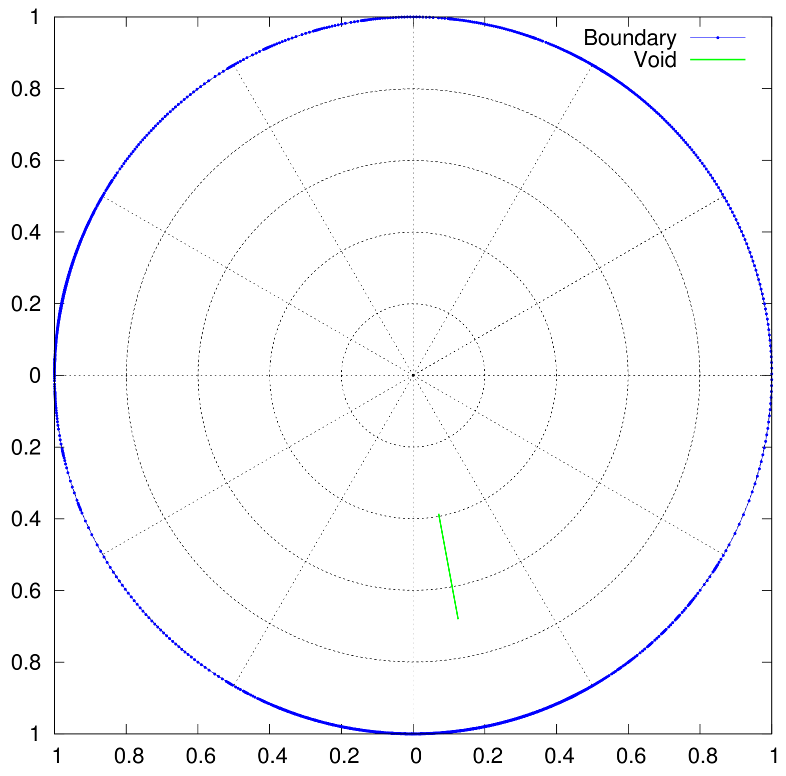


Figure 4.18: Target domain with the void reduced to a radial line.

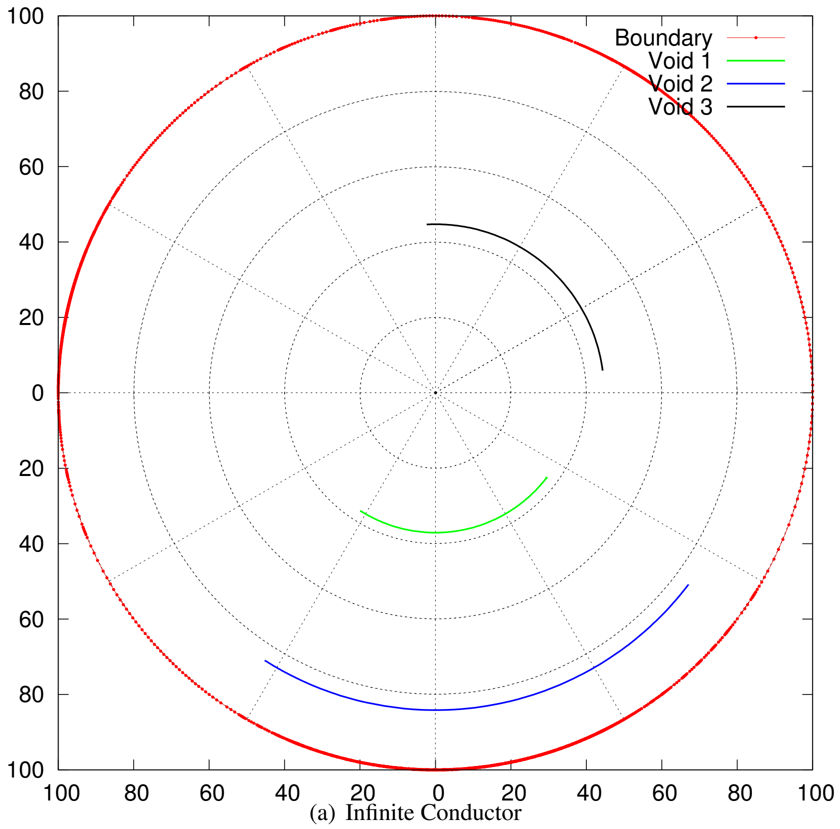
4.7 Multiple Voids

If there are multiple voids in the source domain, they can be handled in the same way as the case of a single void. A void is said to eclipse another if there is a streamline that passes through both voids. When there is no eclipsing, the methods work as described in the previous sections. When the source domain involves the partial or total eclipse of one void by another the streamline should be continually restarted until it reached the shape centre. Regardless of the method used, the standard steps should be followed when restarting streamlines.

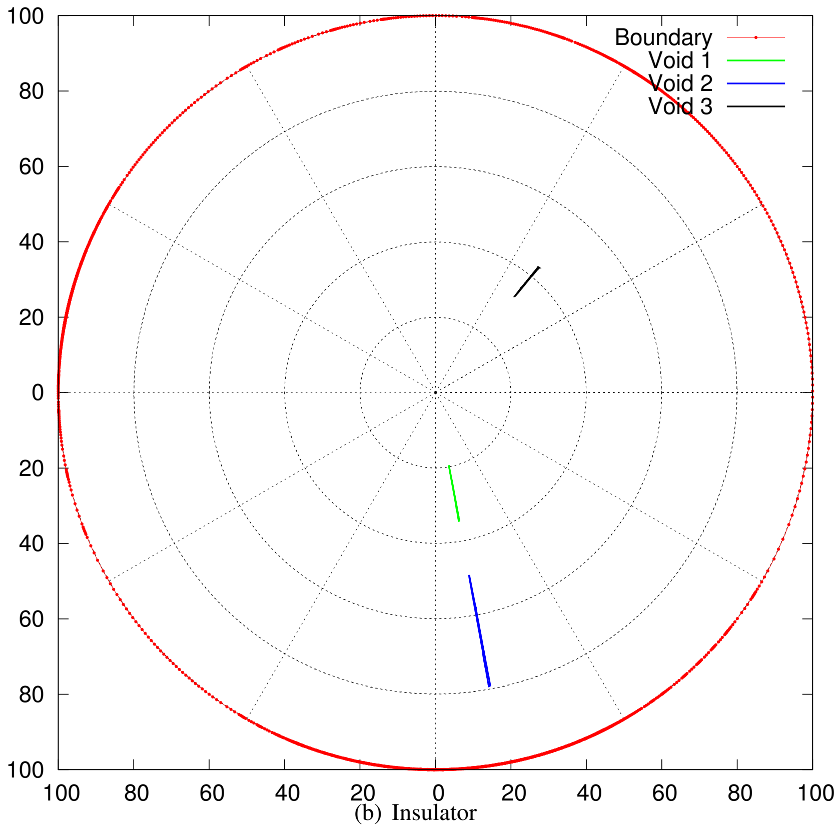
Figure 4.19 shows how eclipsing affects each of the aforementioned methods. The first method reduces each void into separate arcs. The second method is unlikely to encounter the eclipsing effect as it would require that the single streamline that reaches the first void also then reaches the second. If this does happen, however, there will be two separate radial lines in the target domain that have the same angular position but different potential values.

4.8 Reducing Voids to Single Points

The methods can be used in conjunction with one another. Firstly mapping the domain to a disk where all voids have been transformed into arcs and then mapping this target domain again transforming the arcs to radial lines. By mapping the disk domain using the other method, the voids now shrink to zero length lines – single points. This method compounds numerical errors from both steps so requires a much finer grid as small numerical inaccuracies around the voids can be problematic.



(a) Infinite Conductor



(b) Insulator

Figure 4.19: Target Domain with Multiple Voids

4.9 Conclusion

This chapter gave an overview of the implementation of the algorithm from Voruganti *et al.* [2006] as well as a number of extensions to the method. Section 4.3 focuses primarily on the major elements of the code:

- preprocessing the source domain and detecting boundaries;
- constructing the relevant set of linear equations and solving the resulting system;
- calculating the streamlines orthogonal to the potential contours and
- calculating the parametrisation based on the streamline angles and potentials.

Pseudo code for the standard 2D approach is given in Algorithm 1 of Appendix A.

Following that, Section 4.4 discusses extensions to this approach so that source domains containing voids can be mapped in a similar way. The extensions allow the user to map the voids to arcs, radial lines or through a combination of both methods, to a single point. Multiple voids can be handled by restarting any streamlines that reach a fixed point, on the other side of the void. The following chapter discusses the implementation of the methods in parallel.

Chapter 5

Parallel Domain Mapping

5.1 Introduction

There were large performance gains when the method was reimplemented in C++ and PETSc. Both the MATLAB and C++ implementations were run with a single process for computation forcing serial execution of the code. In this chapter, the method is adapted to find a solution using multiple processors. The changes to the implementation that allow for distributed computation are discussed. The chapter following this one provides a comparison of the optimized serial code against the optimized parallel code both, written with C++ and PETSc.

As described in Chapter 4, there are three primary phases to the computation of the map, namely constructing/reading the domain geometry (Section 4.3.1), solving the potential field (Section 4.3.2) and finally solving for the streamlines and calculating the parametrisation (Section 4.3.3). Each phase is dependent on the previous and they need to be processed in order; there are however, significant performance gains to be found through parallelisation within each phase. This chapter shows in detail how the entire process can be optimized by distributing sections of work among multiple processes in a system where shared memory is not guaranteed. As processors do not share memory, they communicate using the Message Passing Interface (MPI) discussed in Section 2.8.2. It is assumed that message passing is slow due to latency caused by network communication. As such, it is used sparingly. The parallel algorithm is listed as Algorithm 4 in Appendix A.

5.2 Parallel Domain Representation and Construction

This initial phase of the system involves reading in the source domain and translating the stored file into the data structures needed to solve Laplace's equation. It is assumed that the description of the domain is initially on the master node – the MPI process of rank 0. This node should either perform the task serially, or distribute the data to the nodes for parallel computation. The primary task in this phase is to detect the boundaries and centre point of the domain supplied by the user. A grid point is considered a boundary point if it is part of the domain and adjacent to at least one point outside of the domain. Voids can be calculated in the same way – any point inside the domain adjacent to at least one void point is considered part of the void boundary. If the initial source representation does not differentiate between points outside the domain and points within a void, a flood fill can be performed until all points outside the domain have been visited. The largest flooded area is considered to be outside the domain and all smaller floods are considered to be the interior of the void regions. This can be performed in parallel, although the process is fast in comparison to the rest of the mapping and so was not parallelised.

It is thus the master process's job to read the domain, find the boundaries and then distribute a copy of this data to each process. This can be done by an MPI broadcast, where all processes in the communicator receive a message from the same node at once. Once the domain has been broadcast to the nodes the next phase can begin. In short there is no need for parallelisation in the first phase as the work done is minimal.

5.3 A Parallel Solution to Laplace's Equation

5.3.1 Construction of the Linear Finite Difference System

Once each process has a copy of the domain, the construction of the finite difference system can begin. As noted in Section 2.8.3, a parallel matrix is represented in row compressed format where each process owns some selection of the rows in the matrix. The rows of the matrix are allocated evenly to the processes in monotonically increasing order – i.e. the first process has the first 10 rows, the second process has the second 10 and so on. The PETSc manual recommends that, when possible, processes only act upon the matrix in elements of local rows [Balay *et al.* 2011].

Within the finite difference matrix, each row/column corresponds to an interior, boundary, centre or void boundary point. As long as there is some consistent map from the grid points into the matrix, the processes can continue independently of one another. Unfortunately, the number of these points is not constant per row on the domain grid. It is thus necessary to assign each interior/boundary/centre point some row in the finite difference matrix. This can be done as each of these points contribute one equation to the linear system. This is inherently a sequential operation as the second process can only assign row numbers when it knows how many rows are used up by the first. Where the domain is $m \times m$ grid points, this is an $O(m^2)$ operation.

Once this is complete, each process can begin construction of the linear equations within its range according to Equations 4.1, 4.3 and 4.8. Each process calls `MatSetValues` to set a value in the matrix. This function generally caches the values in a structure called the *stash* and the values are only written to the parallel data structure when the matrix is assembled. Once each process has set its values it must block until all processes are ready to assemble. Each process now calls `MatAssemblyBegin` and `MatAssemblyEnd`, which assembles the matrix and copies any non-local values to the relevant process – by constructing the rows on the correct process to start with, there is minimal communication. Where p is the number of processes and n is the size of the stash/number of cached values, this routine does work of $O(pn + n) \leftarrow O(pn)$ [Balay *et al.* 2012]. As the processes only construct values in local rows, the pn component of work is removed. The assembly routine only performs the work associated with copying values from the local stash into the local section of the parallel structure. This takes $O(n)$ operations.

The right hand side vector for the system should be constructed at the same time as the matrix. This vector is handled in a similar fashion, where a process owns the same items in the vector as rows in the matrix. As with the matrix, PETSc needs to assemble the vector when the entries have been added by calling `VecAssemblyBegin/MatAssemblyEnd` – $O(pn + n) \leftarrow O(pn)$ where n is the stash size for the vector.

At this stage the matrix is ready for preconditioning and the linear solver in the following phase. In summary, this phase performs $O(m^2)$ serial operations to map domain points to rows in the matrix. In parallel the processes all perform $n \times O(1) \in O(n)$ operations to prepare the local portion of the matrix and vector. Finally the Assembly routine runs in $O(pn + n) \leftarrow O(pn)$ operations. Overall, the longest running process takes $O(m^2 + pn + n) \leftarrow O(m^2 + pn)$ operations.

5.3.2 Solving the Linear Equation

Now that the finite difference matrix has been constructed using the parallel data structure, the preconditioning and solving methods are applied. The KSP solver and preconditioning requirements are beyond the scope of this research. The interested reader is directed to Prasad *et al.* [1995], Cunha and Hopkins [1994] and Bahi *et al.* [2011] for in depth evaluations of the GMRES methods with LU preconditioning on parallel and sequential systems.

5.4 Streamline Tracking and Parametrisation

5.4.1 Streamline Tracking

Once the potential system has been solved, relevant parts must be distributed to all processes so that streamline tracking can begin. In this implementation it was found that the time taken to distribute the entire potential field was acceptable. If another level of memory and/or speed optimization is necessary, one could transmit only the parts of the potential field through which a streamline passes to the relevant process. Arguments against this are strong.

If each process only has the potential field for some small area it can only calculate streamlines in that limited part of the domain. Whenever a streamline reaches the edge of the local area, the process must either request more potential values from another process or it must allow another process to continue with the streamline calculation. The overhead to achieve either of these protocols is heavy. On top of this overhead, the approach means that all streamlines for one process must originate in the same area. Some areas are closer to the centre than others, which means that keeping the distribution of work even among the processes requires still more overhead. All of this overhead also comes with the cost of network communication. If each process rather broadcasts its potential values one by one, then in total only $(m^2 - n)$ numbers are broadcast onto the network. In practice this has performed well.

Once each process has a copy of the potential field, streamline tracking is considered an *embarrassingly parallel* task. Each streamline can be calculated completely independently from other streamline calculations. As mentioned above, different parts of the boundary are different distances from the shape centre. This means that some streamlines are longer and take more time to calculate. It is important that the work is distributed among the processes such that each process does approximately $1/p$ of the total streamline work.

One obvious approach is to have a master/slave approach where the master node dishes out more work to any slave node as it finishes. This method, however, wastes a processor and requires more network communication. It would be better to divide up all of the work at the start in a way that fairly allocates work without the extra overhead.

Here the continuity and smoothness of the map can be exploited. If two boundary points are adjacent then the streamlines originating at these two points will be approximately the same length and thus approximately the same amount of work to calculate. This means that if two processes are calculating streamlines originating from adjacent boundary points, they will perform approximately the same amount of work. Therefore, if there are two processes calculating all of the streamlines; rather than giving the first process the first half of the points and the second process the remaining points, the work is more evenly spread if the processes alternate points. In other words the first process does boundary points $0, 2, 4, 6, 8 \ldots$ while the second process does boundary points $1, 3, 5, 7, 9 \ldots$ By dividing the work in this manner the processes do approximately the same amount of work over the whole source domain. This keeps all processes busy for the entire time without introducing any network overhead.

5.4.2 Parametrisation

Once all of the streamlines have been traced to the shape centre, the parametrisation can begin. Again this task is embarrassingly parallel. As long as each process has a copy of the potential field and the relevant streamlines, it can calculate the parametrised values for any point along those streamlines. The parametrisation should be calculated for a number of grid points to give a map with enough points to interpolate the parametrised values to the required accuracy. As this forward map is calculated and stored, the reverse map should be stored as well. Again, it is important the workload is divided fairly among the processes. As each process already has approximately the same length streamlines, the number of points that each process will encounter are approximately the same. Thus, by dividing the streamlines as mentioned in the previous section, the workload is divided evenly for both parts – in fact, a process that has finished its own streamline calculation need not wait for other processes to finish before it begins parametrising the points along its own streamlines.

Finally, once the partial maps have been calculated on each process, the processes should return these maps to the master node for reconstruction. This can be done in a linear fashion where each process waits to transmit to the master resulting in $O(xp)$ operations, where x is the work required to transmit a local parametrisation to the master node and for the master node to merge the new information into the global representation. Alternately it can be done in a binary tree type fashion where there are $O(\log(p))$ cycles. In this approach the p processors are divided into pairs. During a cycle a pair merges its local data - $p/2$ merges. Once the first cycle is complete, only one process from each pair continues to the next cycle. The $p/4$ processes that now continue, repeat the process performing $p/8$ merges. This continues until there is only one process left. This forms a binary tree with a maximum height of $O(\log(p))$. This results in an execution time of $O(x \cdot \log(p))$ where x and p represent the same as before. Once this merge has taken place, points can be mapped forwards and backwards in constant time.

5.5 Conclusion

This chapter provided insight to the decisions made in order to parallelise the system. The serial, parallel and embarrassingly parallel portions of the algorithm were identified and trade-offs between extra processing and network communication were considered. The following chapter compares the parallel and serial 2D implementations in terms of execution time.

Chapter 6

Parallel Implementations and Performance Analyses

6.1 Introduction

In Chapter 5 the parallel computing algorithm was discussed. In this chapter the algorithm is tested on a varying number of cores and the runtimes are compared. The algorithm is implemented as described in the previous chapter using C++, PETSc and MPI. When timing the parallel implementation, the *wall clock time* of the longest running process is used rather than the *user time* used over all the processors.

The analysis is performed on two different parallel architectures. In Section 6.2 one large machine is used to run the parallel system. In Section 6.3 the University of the Witwatersrand's *hydra* cluster is used to perform the same tests. These comparisons provide insight into the way the algorithm scales when the network communication is used for a cluster. These results investigate the *strong scalability* of the program, measuring the performance of the system as the number of processors increase and the problem size remains constant.

In each case, the domain in Figure 6.1 is processed 10 times for each number of cores. The average and standard deviation are reported in each case. The domain undergoes scaling as well so that the performance is measured on different sized domains. The scales are described in Table 6.1.

Finally, the same tests as the ones done on Hydra are run on the CHPC Sun Cluster. These encouraging results show how effectively the system runs on a cluster with low network latency.

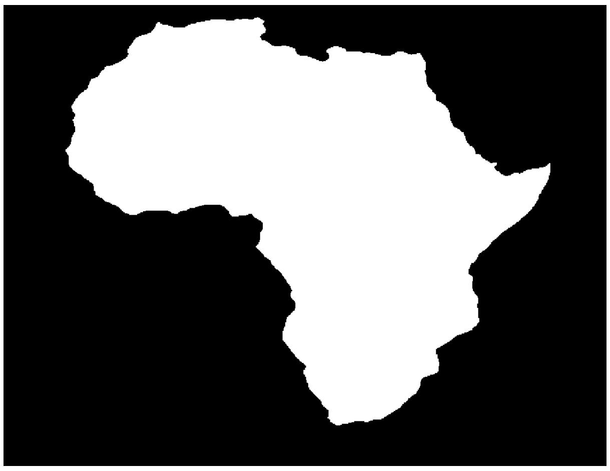


Figure 6.1: Parallel Performance Test Domain

Table 6.1: Domain Dimensions for Performance Tests

Dimensions			Points		
Scale	**Width**	**Height**	**Total**	**Interior**	**Interior Scale**
0.5	384	455	304380	88190	0.25
1	766	909	696294	354116	1.00
2	1528	1814	2771792	1408836	3.98
3	2292	2721	6236532	3175582	8.97

6.2 A Single Machine

6.2.1 Setup

The timing runs below were obtained from a single multicore machine. The hardware configuration was as follows:

- Intel Xeon(R) CPU E5620 2.4 GHz. There were 2 physical processors each with 4 physical cores. With hyperthreading this resulted in 16 virtual cores.
- 50 GB Ram on the whole machine
- 8.5 TB hard drive storage

The software configuration was as follows:

- Ubuntu 12.04 64-bit
- Linux Kernel 3.2.0-32-generic x86_64
- GNU C Compiler (`gcc/g++`) version 4.6.3 with level three optimization
- OpenMPI 1.6.1 (custom compiled)
- PETSc 3.2-p7 (custom compiled)

6.2.2 Multicore Scaling: Number of Cores vs Time

Table 6.2 shows the running times in seconds for the different sized domains on a different number of cores. Figure 6.2 graphs these results, note that time is shown on a log scale. It is immediately evident that the parallel processing has increased the performance of the system substantially. The best times in all cases occurred when using all of the physical hardware present on the machine, before any hyperthreading was activated. Using all 8 physical cores gave, on average, a speed-up of 6 times over the 4 different sized domains.

All of the simulations, however, show a similar pattern. The running time decreases while there are physical cores available – up to 8. As soon as more than 8 threads are running, hyperthreading uses context switching to run up to two threads on each of the physical cores. It is evident that this context switching slows down the overall program with the fastest runtime occurring when all of the physical cores are used without hyperthreading. This result is general and occurred on all test machines.

Table 6.2: Parallel Run Times on a Multicore Machine

Cores	**Average Running Time (s)**				**Cores**	**Standard Deviation (s)**			
	0.5	1	2	3		0.5	1	2	3
1	4.547	46.187	506.550	2115.879	1	0.035	0.014	0.312	1.886
2	2.808	28.630	315.550	1290.589	2	0.065	0.072	3.560	1.981
3	1.964	19.739	208.428	865.412	3	0.007	0.091	1.079	13.676
4	1.570	15.192	133.861	657.709	4	0.010	0.091	2.150	4.111
5	1.402	12.783	129.703	554.905	5	0.038	0.034	0.542	14.994
6	1.288	11.135	110.459	427.666	6	0.092	0.111	0.225	6.412
7	1.153	10.001	98.184	419.612	7	0.106	0.272	1.035	7.440
8	1.104	9.104	70.430	345.827	8	0.082	0.180	0.811	6.372
9	1.549	13.807	128.911	572.752	9	0.120	0.362	1.876	2.491
10	1.478	12.649	100.270	489.163	10	0.108	0.429	1.413	2.559
11	1.477	11.958	102.762	462.167	11	0.095	0.264	1.603	3.693
12	1.382	11.173	107.737	330.477	12	0.109	0.270	1.688	5.606
13	1.364	10.601	99.760	402.326	13	0.062	0.058	1.544	5.467
14	1.332	10.206	99.982	347.018	14	0.005	0.155	0.571	5.141
15	1.275	9.856	98.092	375.777	15	0.047	0.269	0.426	2.745
16	1.244	9.677	85.797	314.034	16	0.004	0.026	0.481	2.228

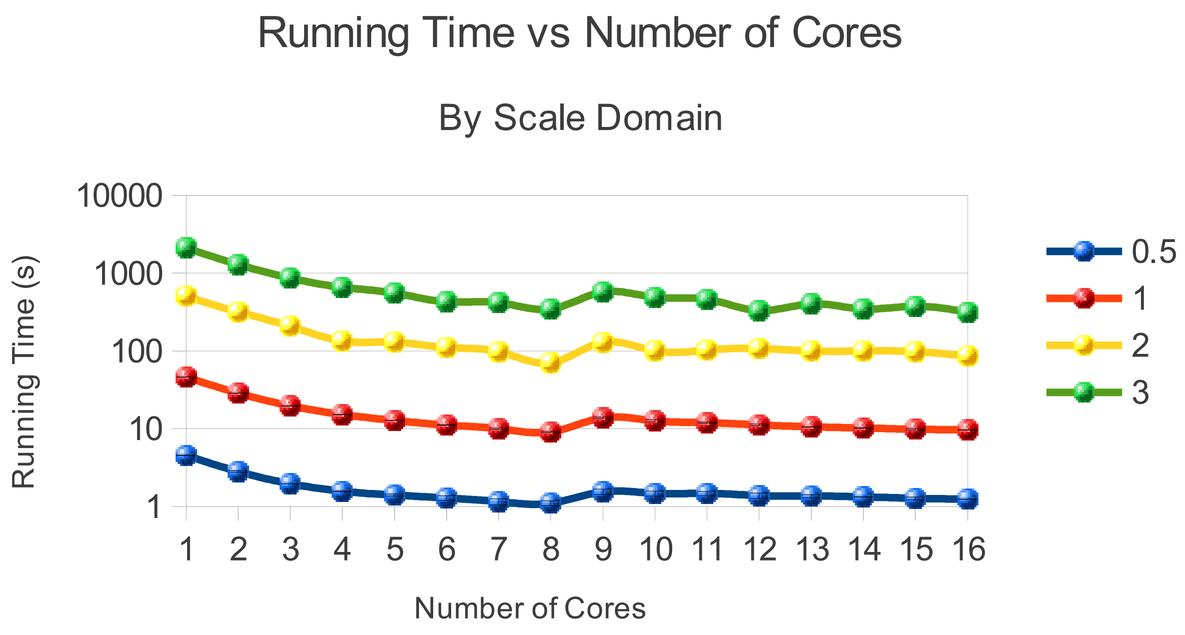


Figure 6.2: Multicore Architecture: Running Time vs Number of Cores (Log Scale)

6.3 The Hydra Cluster

6.3.1 Setup

- Intel Core i7 CPU 3.0 GHz. There was 1 physical processor with 4 physical cores per node. There were 7 nodes.
- 6 GB Ram per node
- 10/100 Mbps Ethernet Adapters connected by a switch
- Unknown storage capacity, mounted by NFS

The software configuration was as follows:

- Ubuntu 10.10 64-bit
- Linux Kernel 2.6.35-28-generic x86_64
- GNU C Compiler (`gcc/g++`) version 4.4.5 with level three optimization
- OpenMPI 1.6.1 (custom compiled)
- PETSc 3.2-p7 (custom compiled)

6.3.2 Multi-computer Scaling: Number of Cores vs Time

Table 6.3 and Figure 6.3 shows the running times for the different sized domains when run over the Hydra cluster. These run times make use of a maximum of 4 cores per node to prevent the context switching encountered in the previous experiment. It is again evident that the parallel processing has increased the overall performance.

The performance gains over the cluster are only evident on large domains. When run on the same size domains as those in the previous test, the running times increased as the cluster size increased. On larger domains, however, the benefits were obvious. This indicates that the network overhead is significant in comparison to the work done in the first tests, but as the test size grows this overhead becomes less important. Due to the increased memory capacity and cores, domains significantly larger than before can be processed. The tests in this section were thus run on domains significantly larger than those used previously. For Figure 6.3, the domain from Figure 7.1 with grid sizes of 40^3 and 80^3 were each run 4 times.

Table 6.3: Parallel Run Times on a Multi-Computer System

Cores	**Average Running Time (s)**	
	40^3	80^3
4	1242	81890
8	0328	09808
12	0219	05348
16	0165	03634
20	0148	02374
24	0162	03104
28	0181	03819

Cores	**Standard Deviation (s)**	
	$40 \times 40 \times 40$	$80 \times 80 \times 80$
4	1.4	95
8	1.9	71
12	1.4	41
16	2.7	38
20	0.7	35
24	1.8	30
28	0.3	36

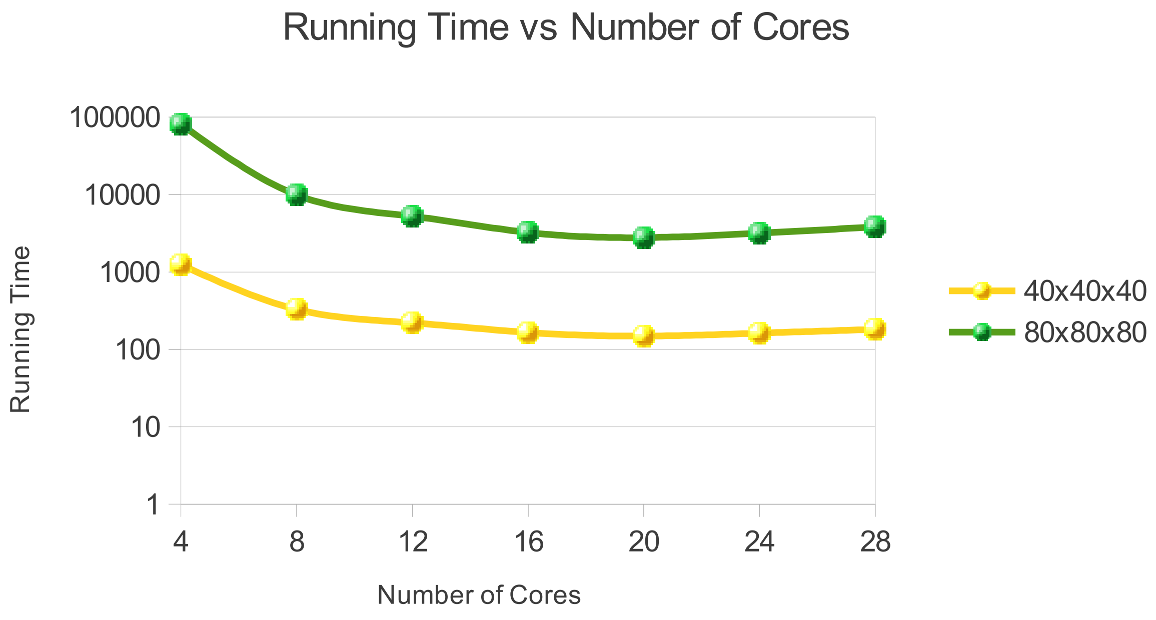


Figure 6.3: Multi-Computer Architecture: Running Time vs Number of Cores (Log Scale)

6.4 The CHPC Sun Cluster

As of writing, this cluster is partitioned into 4 sections each used for different sized jobs. The specific details of the cluster can be found in CHPC [2012]. The partitions are shown below in Table 6.4. Of primary importance is the infiniband network connections among the compute nodes. Because of this low latency network, the overhead seen on Hydra with an ethernet network, is avoided. The experiments run on the sun clusters were run on the Harpertown and Nehalem clusters with 2 processes per node.

System Name	**Sun Hapertown**	**Sun Nehalem**	**Sun Westmere**	**Dell Westmere**
CPU	Intel Xeon	Intel Nehalem	Intel Westmere	Intel Westmere
CPU Clock	3.0 GHz	2.93 GHz	2.93 GHz	2.93 GHz
CPU Cores	384	2304	1152	2880
Memory	768 GB	3456 GB	2304 GB	8640 GB

Table 6.4: SUN Hybrid Cluster [CHPC 2012]

Running the same 3D domain as before, i.e. Figure 7.1, the same tests are run on the CHPC cluster. Each simulation was run three times and the average of these runtimes is reported in Table 6.5 and Figure 6.4. It is immediately evident that the implementation performs significantly better on the CHPC system than on the previous systems.

Table 6.5: Parallel Run Times on the CHPC Sun Cluster

Cores	**Average Running Time (s)**	
	40^3	80^3
28	2.641	45.459
32	2.582	36.175
36	2.359	28.366
40	2.241	23.222
44	2.169	18.948
48	2.254	15.922
52	2.200	13.154
56	2.193	11.771

Cores	**Standard Deviation (s)**	
	40^3	80^3
28	0.280	1.158
32	0.321	2.147
36	0.158	1.980
40	0.280	1.256
44	0.209	0.645
48	0.047	0.584
52	0.344	0.202
56	0.206	1.056

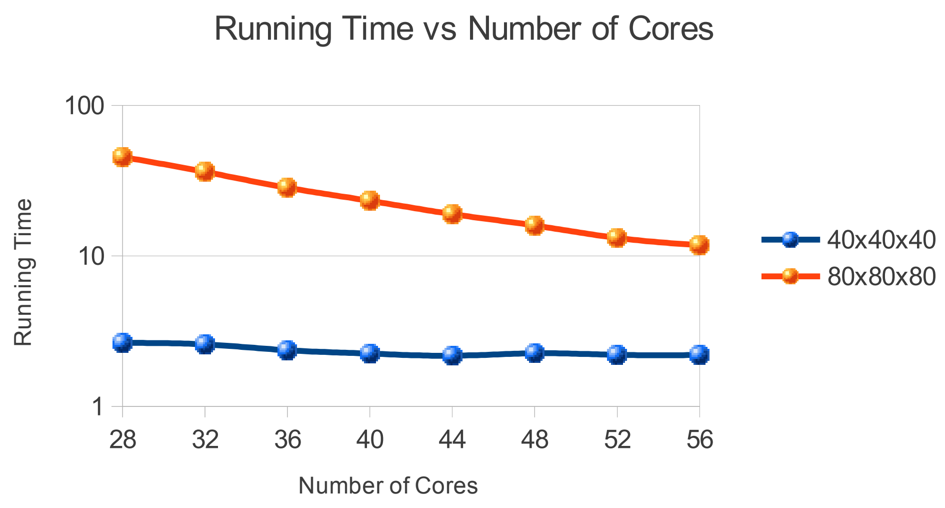


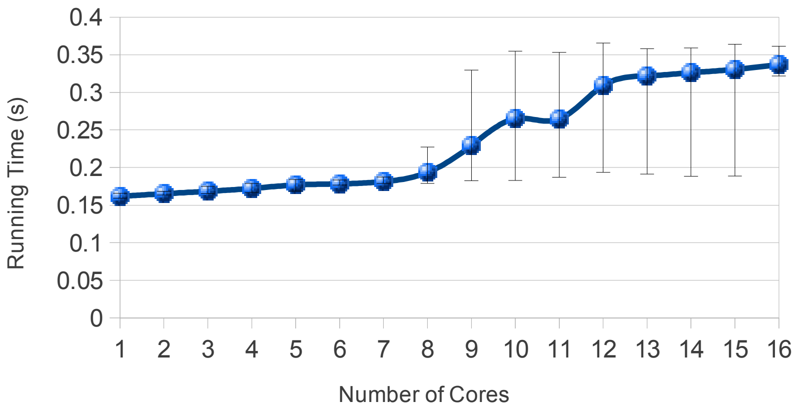
Figure 6.4: CHPC Sun Cluster: Running Time vs Number of Cores (Log Scale)

6.5 Performance of Parallel Routines

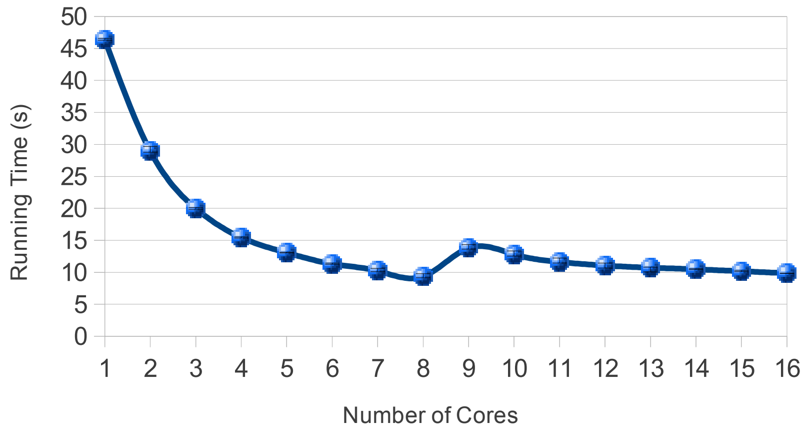
Figures 6.5 and 6.6 show how the time spent in the different phases changes based on the number of threads spawned. These timing runs were performed on the machine from Section 6.2 and the same data, again the effects of hyperthreading are seen when the number of threads exceed the number of physical cores. Constructing the finite difference matrix, the method appeared slower when more CPU's were added. This indicates that the current approach does not scale well to a cluster. However, the total time spent in this section of the process is insignificant in comparison to the time spent in the other phases. There may be some more parallel optimizations that could be used in this phase, but it requires significant effort for very little performance benefit.

On the other hand, when solving the large linear systems adding more CPU's as the number of points increased had a positive impact overall. There is, however, a balance as eventually the extra overhead diminishes the performance gains in this phase. Solving the linear equations requires interactions among the processors – the more processors, the more overhead.

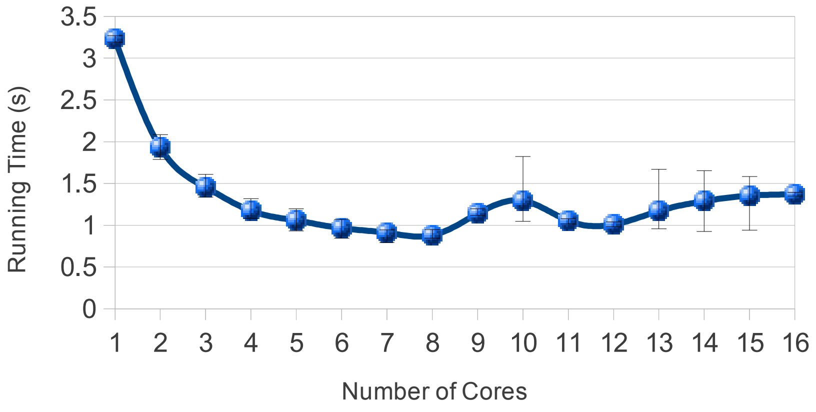
Finally, when calculating the streamlines and the parametrisation, the more CPU's the better. This phase is embarrassingly parallel. As such if the domain of interest is large enough, a linear change in the number of processors has a linear impact on this section of code. This relationship breaks down when transferring the potential field to all of the processes takes more time than the gains offered by calculating streamlines concurrently. Even in the unscaled domain, the number of streamlines exceeded 3800. It is unlikely that a cluster would be working on such a small domain, for practical purposes the linear relationship between the number of cores and the runtime can be expected to hold.



(a) Setup Phase



(b) Solve Phase



(c) Streamline Phase

Figure 6.5: Running Times by Phase and Number of Cores

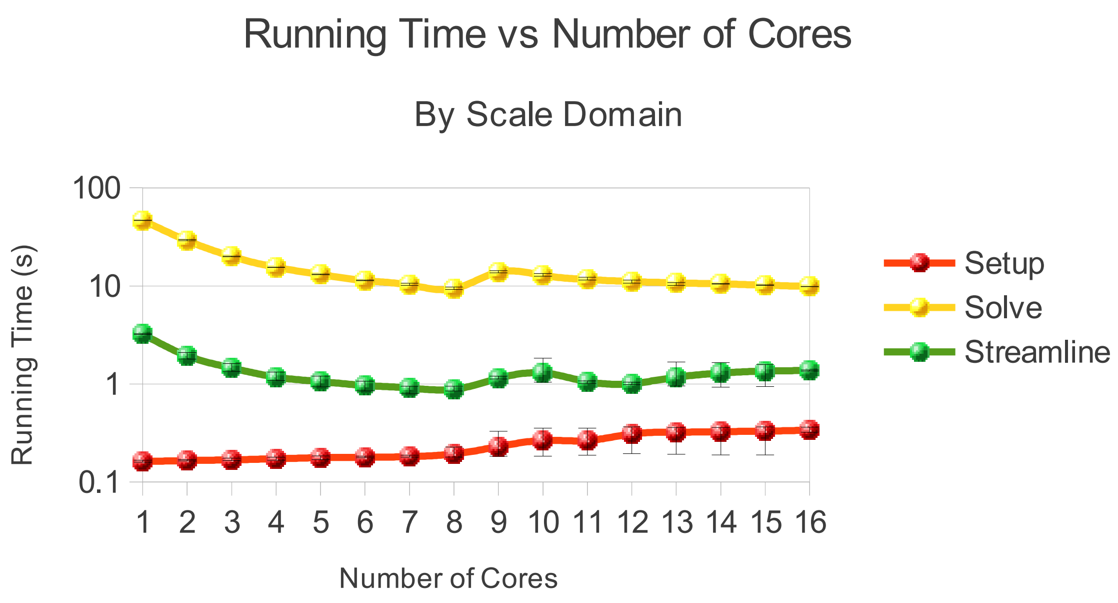


Figure 6.6: Running Times by Phase and Number of Cores

6.6 Conclusion

This chapter focused on the performance of the parallel C++ implementation. The results clearly show that parallel computing gives the algorithm an enormous boost in terms of speed as well as the size of domains that can be processed. The tests reveal that the use of virtual cores through hyperthreading causes the overall running time to increase as the hardware begins context switching. This result can be generalised; one should not use more threads than the physical hardware can handle. This is true both for hardware context switches through hyperthreading and for software context switches through the operating system. This proves to be especially important when domains of a higher dimensionality are considered. The implementation was then run on the Hydra cluster at the University of the Witwatersrand, which uses an ethernet network. Finally, the implementation was run on the CHPC Sun cluster and its use on large domains is illustrated.

Chapter 7

Beyond 2 Dimensions

7.1 Introduction

The previous chapters all focused on 2D domains. This chapter focuses on the extension of the parallel implementation to three dimensions and higher. Although visualisations are only provided for 3D domains, the extensions of the method to higher domains is exactly the same. The extension of the methods from Section 4.4 for handling voids is beyond the scope of this research as the topological implications of voids in higher dimensions becomes problematic - this is briefly noted in Section 7.3.

7.2 Mathematical Extensions

The method outlined in Chapter 4 extends to higher dimensions in a straightforward manner. The necessary properties of Laplace's equation still hold in higher dimensions and the only substantial changes are found in the actual implementation. It can be shown that the mean value theorem in Section 2.5.2 always results in the correct finite difference equations for a regular grid. Thus the relevant discritised formula for Laplace's equation in n dimensions is just the average of the neighbours in each direction throughout all the dimensions. In general, where $\vec{b}_i$ is the i^{th} basis vector, the discrete formula becomes:

$$\phi(\vec{x}) \approx \frac{1}{2n}\sum_{i=1}^{n}\left[\phi(\vec{x}+\vec{b}_i)+\phi(\vec{x}-\vec{b}_i)\right] \tag{7.1}$$

In 3D, ϕ_{ijk} is then the average of its 6 neighbours as shown in Equation 7.2,

$$\begin{aligned}\phi(x_i,y_j,z_k) \approx\ & \frac{1}{6}[\phi(x_{i+1},y_j,z_k)+\phi(x_{i-1},y_j,z_k)+\phi(x_i,y_{j+1},z_k)+ \\ & \phi(x_i,y_{j-1},z_k)+\phi(x_i,y_j,z_{k+1})+\phi(x_i,y_j,z_{k-1})]\end{aligned} \tag{7.2}$$

Equation 7.1 leads to a linear system similar to those shown previously. The size of this linear system, however, is exponential in the number of dimensions. For example, a square domain of width m in 2D has $O(m^2)$ grid points. In 3D however, this becomes $O(m^3)$ and in general an n-dimensional system has $O(m^n)$ grid points. The linear system is still created as a 2D matrix and solved as usual, but the number of points is significantly larger. This highlights the importance of scalability and the need for parallel implementations.

7.3 Topological Issues in Higher Dimensions

In 2D the topology of the source domain is wholly described by the number of voids in the connected domain. As such the topology of the target domain was obvious – a disk with the correct number of arcs/radial lines/points in the disk interior. In 3D, however, this becomes more difficult as a void could be entirely contained inside the domain or it could make contact with the source domain boundary multiple times forming a torus or other topology. In higher dimensions this becomes even more problematic as the combinations of different types of voids becomes greater and the system would have to handle many more cases.

In the case where the void is entirely contained within the source domain, the methods described in Section 4.4 will work correctly. In more complicated topologies these methods fail. This could be the subject of future work.

7.4 Domain Mapping in n-Dimensional Domains

The algorithm remains the same as in the 2D case, except that the target domain is now a hypersphere in n-dimensions. The potential field is still constant over the entire domain boundary while a single shape centre is chosen. Streamlines are calculated in the same manner, except each point along a streamline is now a point in n-dimensional space. As such there are now $n-1$ angles describing each streamline's approach to the shape centre. This provides a valid map to the n-sphere as each point is parametrised by $n-1$ angles and 1 potential value. For example, in 3D the target domain is a sphere. In spherical co-ordinates, each point is described by the distance from the centre as well as two angles known as the azimuth and zenith.

7.5 Programming Extensions

Arbitrary dimension domain mapping was only implemented in the parallel version of the code. The code was implemented in a way that all mathematical methods worked in n-dimensions. The primary changes involved the generalisation of the methods constructing the finite difference matrix as each row potentially holds up to $2n+1$ values. As the finite difference matrix is still 2D, the system can be easily solved using the same methods as before.

Once the potential field has been calculated, the helper functions that interpolated the field to approximate off grid potential values and gradients were generalised to work in n-dimensions. The streamline tracking methods needed to be able to work in an arbitrary number of dimensions as well, although there remains no change in the mathematics used to calculate these flows. Finally, the parametrisation at the end of the method now calculates $n-1$ angles to describe the streamline as it approaches the shape centre. Visualisation for 3D simulations was done through the use of the VTK library and ParaView.

7.6 A 3D Example

The following is an example where the 3D source domain from Figure 7.1 is mapped to the relevant sphere. Images from different viewpoints are shown to assist understanding of the 3D system.

Appendix B contains copies of the same domains using stereoscopic imaging to help visualise the images as 3D objects.

7.6.1 3D Source Domain

For these examples a non-convex 3D domain is created by intersecting a coarse sphere and a rectangle. This results in the image shown in Figure 7.1

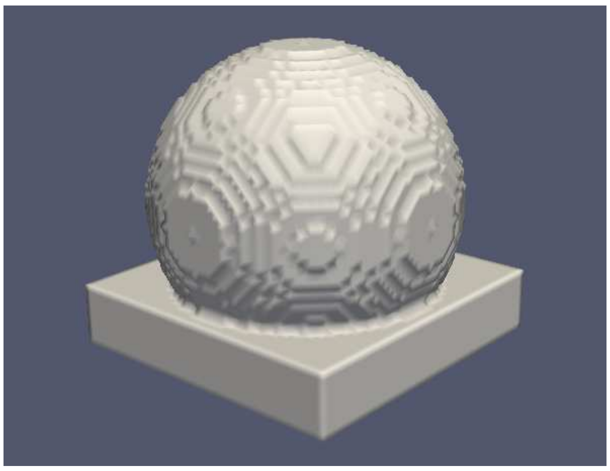


Figure 7.1: Example 3D Source Domain

7.6.2 Potential Field

A potential of 1 is applied to the boundary and the centre of the sphere is chosen as the shape centre. Solving Laplace's equation over this domain results in the potential distribution shown in Figures 7.2 and 7.3.

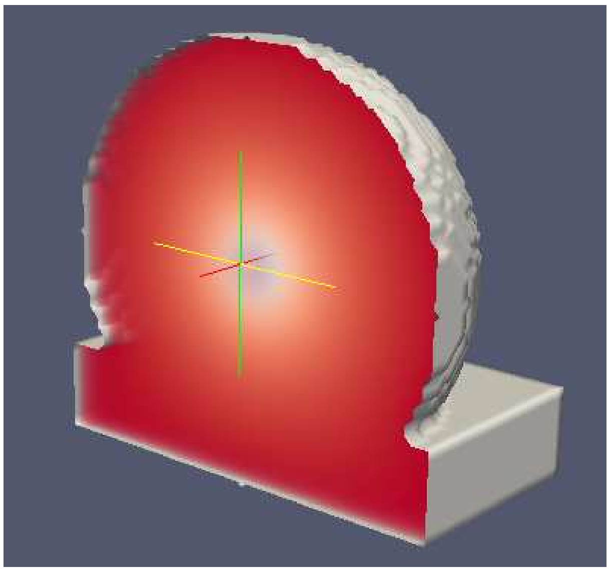


Figure 7.2: 3D Potential Distribution

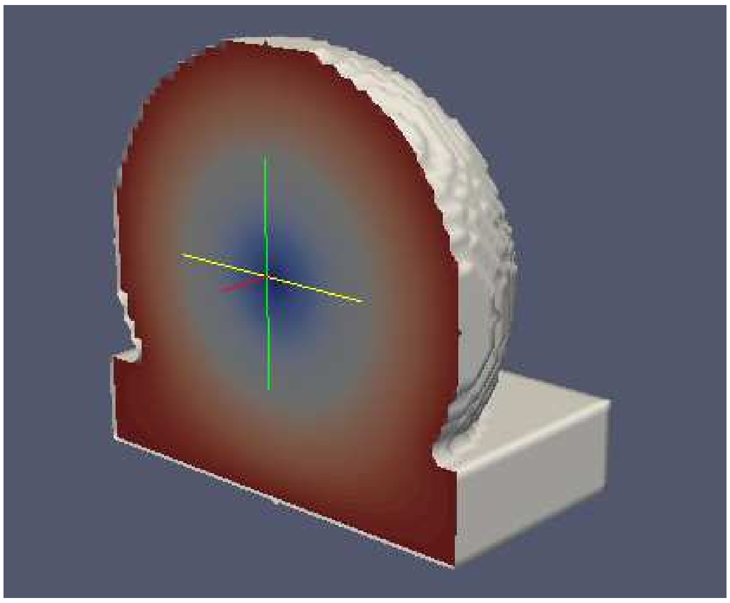


Figure 7.3: 3D Scaled Potential Distribution

7.6.3 Streamline Tracking

As in the 2D domain, the streamlines are tracked from each boundary point towards the centre. For numerical accuracy when calculating the angles, the streamlines are stopped at some distance away from the centre. This is illustrated in Figures 7.4, 7.5 and 7.6 below. The ball around the shape centre shows the surface where the streamlines are stopped.

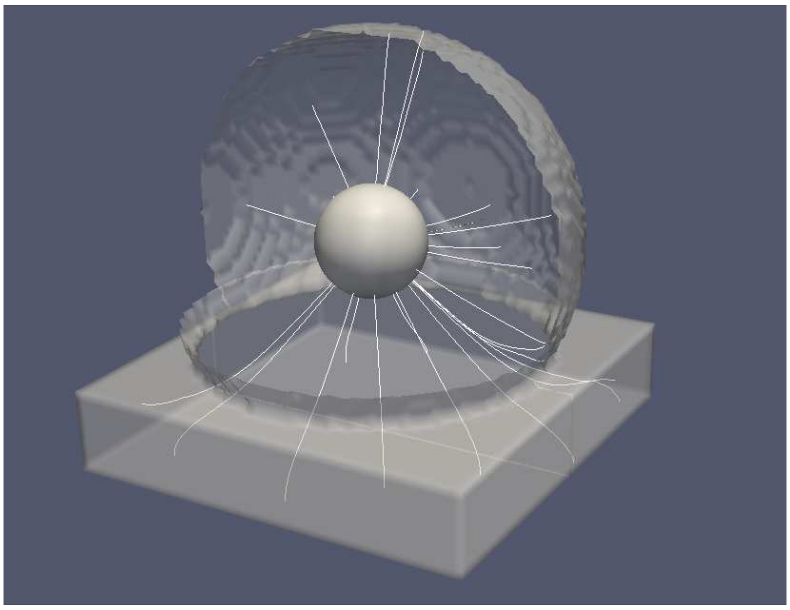


Figure 7.4: 3D Streamline Traces, Angle 1

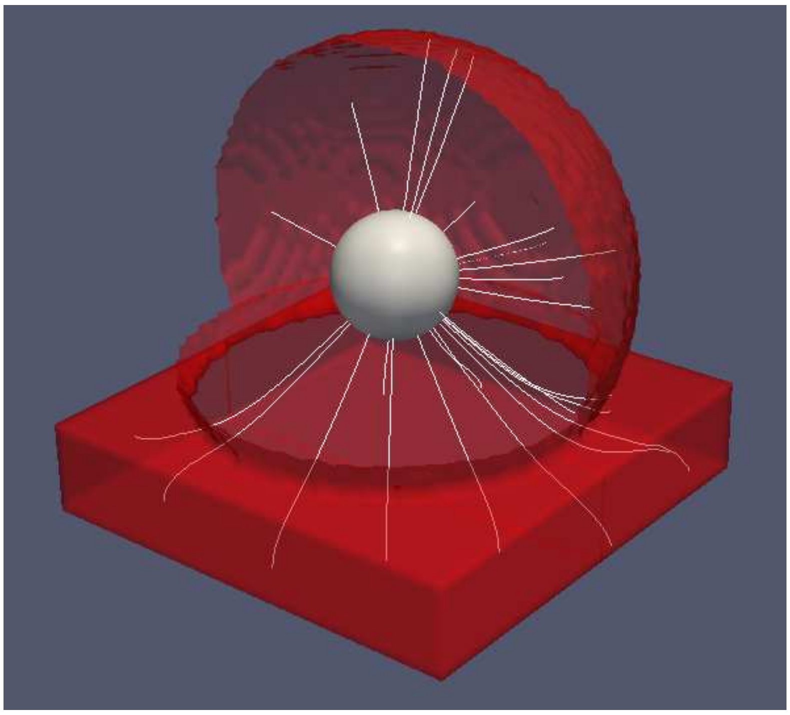


Figure 7.5: 3D Streamline Traces, Angle 2

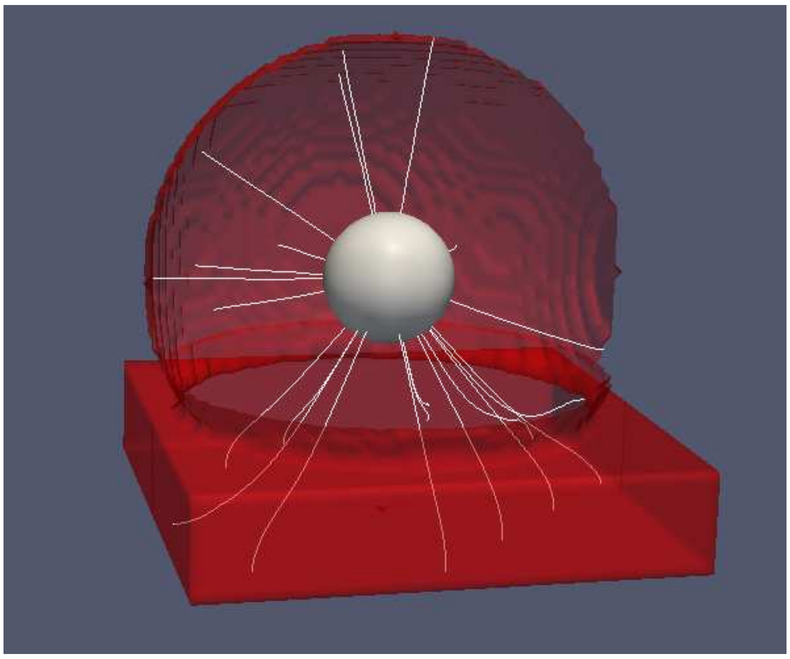


Figure 7.6: 3D Streamline Traces, Angle 3

7.6.4 Target Domain

Finally, the angle at which the streamline approaches the centre, along with the potential value at each point is used to calculate the parametrisation. Figure 7.7 shows the radial lines for one fifth of the boundary points.

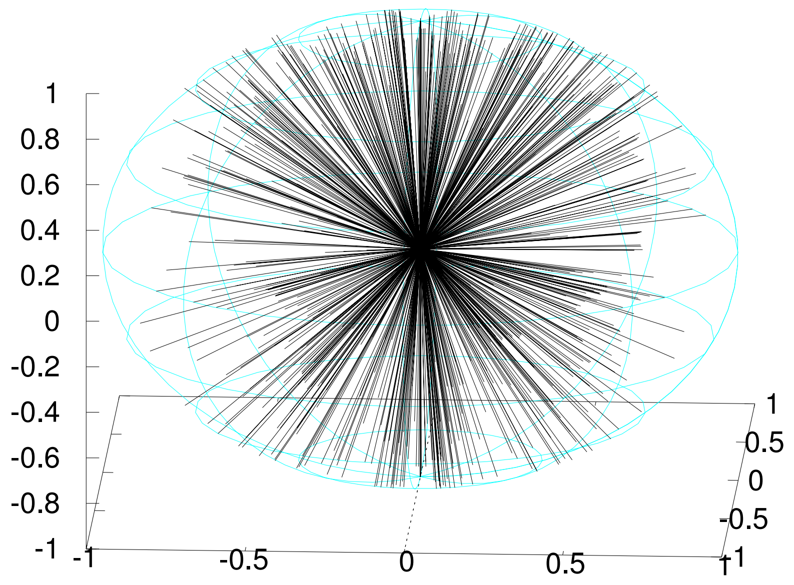


Figure 7.7: 3D Spherical Target Domain.

7.7 Conclusion

This chapter discussed how increasing the dimensionality affects domain mapping. In genus-0 domains, there are no mathematical changes. In genus-$\emptyset$ domains, however, the methods for reducing voids only work in special cases as the choice of topology for the target domain is no longer straight forward. Finally implementation extensions and illustrations of the method show the method applied to a 3D source domain.

Chapter 8

Applications

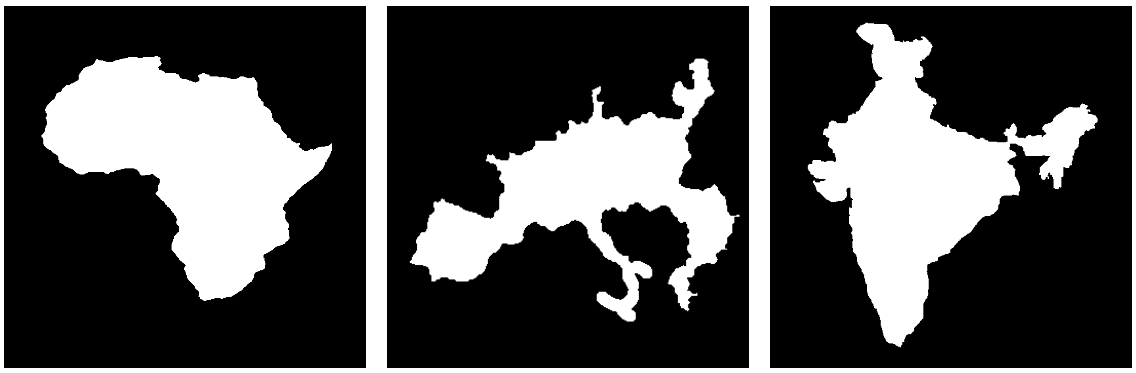
8.1 Introduction

The previous chapters have explained the methods, extensions and implementations. They also provided analysis of the scalability of the parallel algortihm. This chapter focuses on applications of the developed systems. These applications can be generalised to higher dimensions, but visualisation is no longer straight forward. Methods to explore this higher dimensional data include the open source GGobi tool, parallel coordinates and polar charts. These methods provide insight into the structure of high dimensional data, but are beyond the scope of this dissertation.

First, Section 8.2 focuses on robot path planning in 2D and 3D. Second, Section 8.5 illustrates remeshing in a 2D genus-0 domain.

8.2 Applications of Domain Mapping in 2D

Applications of 2D domain mapping are shown in this chapter. The domains of interest are shown in Figure 8.1 below. While these domains have shapes that may be recognisable to the reader, these shapes should be considered arbitrary and do not necessarily have to maintain any physical properties. Note that Figures 8.1(b) and 8.1(c) are found in Voruganti *et al.* [2006] so that the system can be validated according to the results in their original paper.



(a) Africa Domain (b) Europe Domain (c) India Domain

Figure 8.1: 2D Application Domain Shape

8.2.1 Selection of the Shape Centre

The shape centres are selected by taking the interior point furthest from the domain boundary. Figure 8.2 shows this selection.

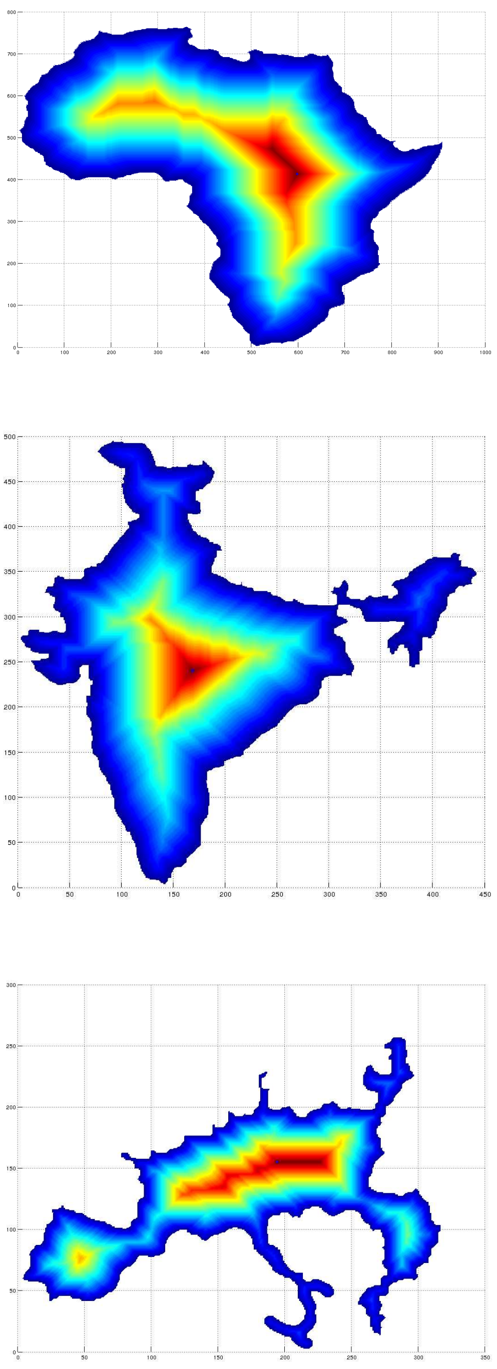
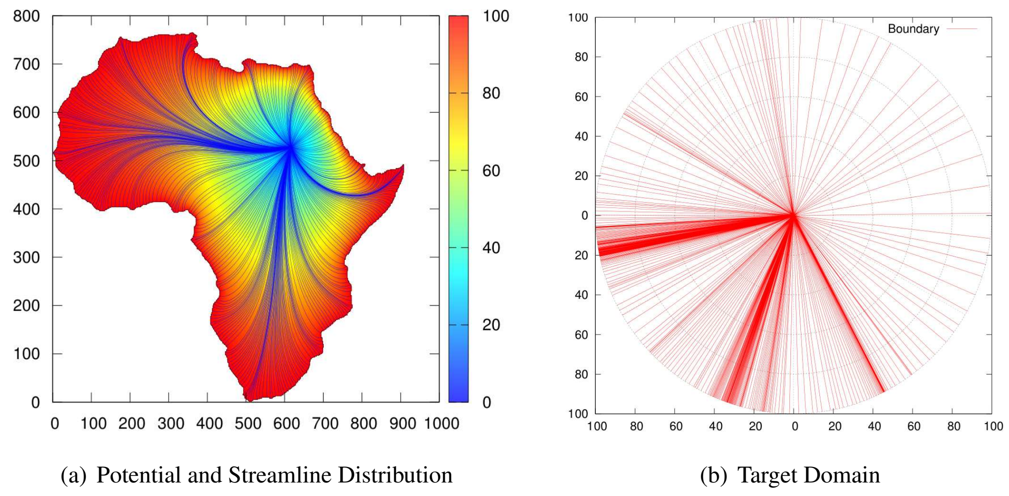


Figure 8.2: Selecting the Shape Centre

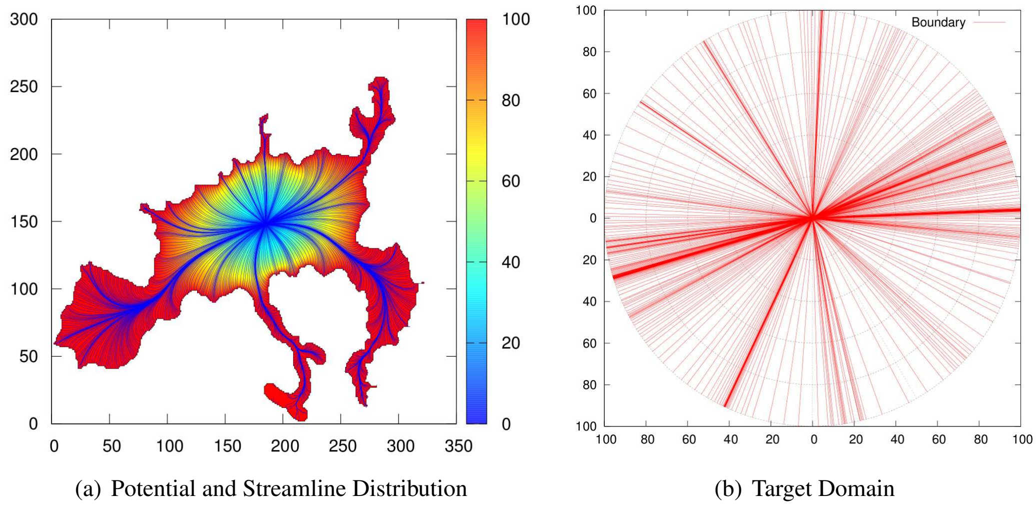
8.2.2 Mapping

Figures 8.3, 8.4 and 8.5 show the potential distributions and target domains for the domains shown above.



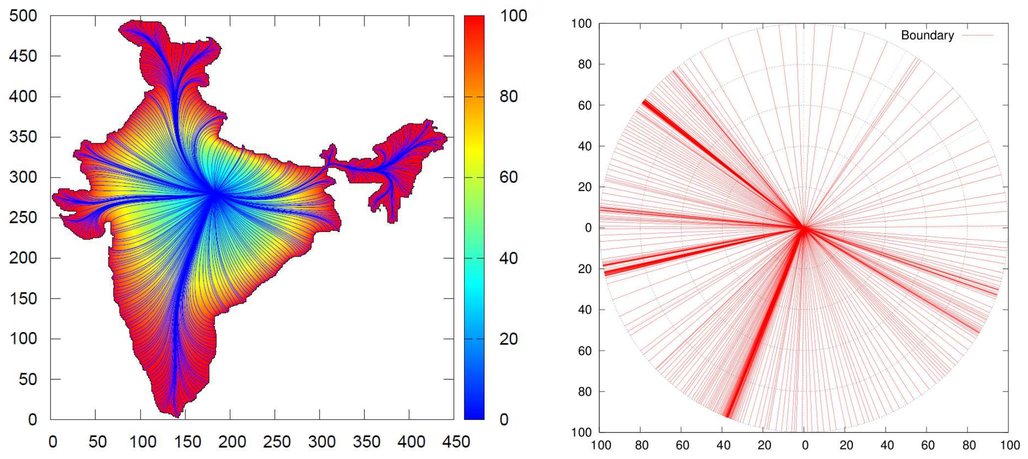
(a) Potential and Streamline Distribution (b) Target Domain

Figure 8.3: Africa Domain, Potential Distribution, Flows and Target Domain



(a) Potential and Streamline Distribution (b) Target Domain

Figure 8.4: Europe Domain, Potential Distribution, Flows and Target Domain



(a) Potential and Streamline Distribution (b) Target Domain

Figure 8.5: India Domain, Potential Distribution, Flows and Target Domain

8.3 Robot Path Planning in 2D

As discussed previously, domain mapping effectively solves the problem of robot path planning in genus-0 2D domains. As an example, suppose there is a robotic arm with two degrees of freedom. Let the angles of the two joints of the arm be represented as θ and ϕ, measured in radians. Because of physical constraints on the robot, the following constraints are necessary: $0 \leq \theta \leq \pi$ and $0 \leq \phi \leq \pi$. So that the robot does not lose balance, there is a final constraint:

$$\theta \leq \left(\phi - \frac{\pi}{2}\right)^2 + \frac{\pi}{3} \tag{8.1}$$

These constraints result in the configuration space shown in in Figure 8.6.

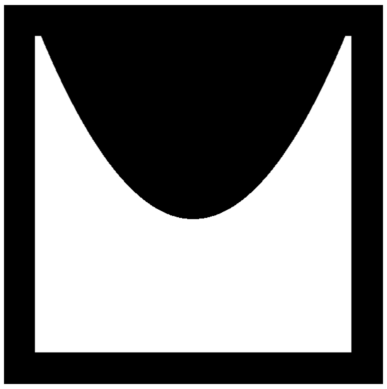


Figure 8.6: Configuration Space of Robotic Arm

The potential distribution in Figure 8.7 is calculated as shown previously.

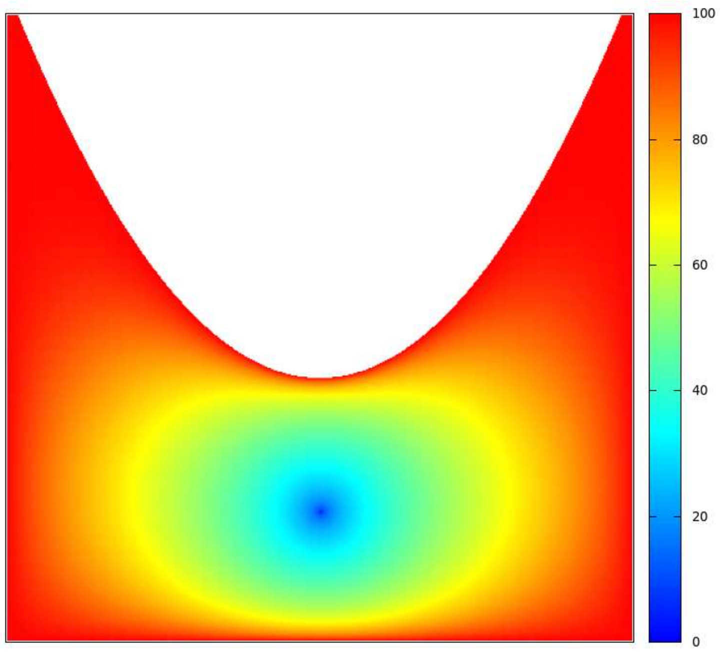


Figure 8.7: Potential Distribution for the Robotic Arm

The streamlines are then traced to the centre and the mapping is calculated as shown in Figures 8.8 and 8.9.

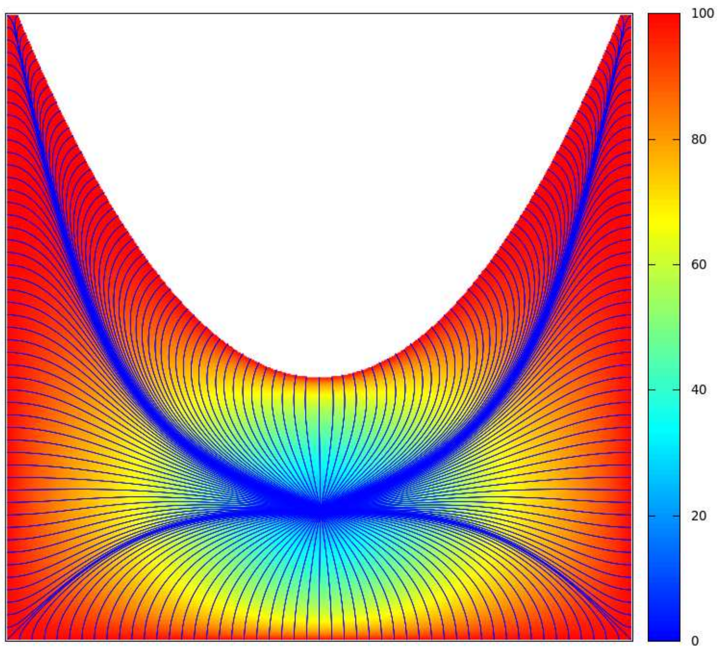


Figure 8.8: Streamline Tracking for the Robotic Arm

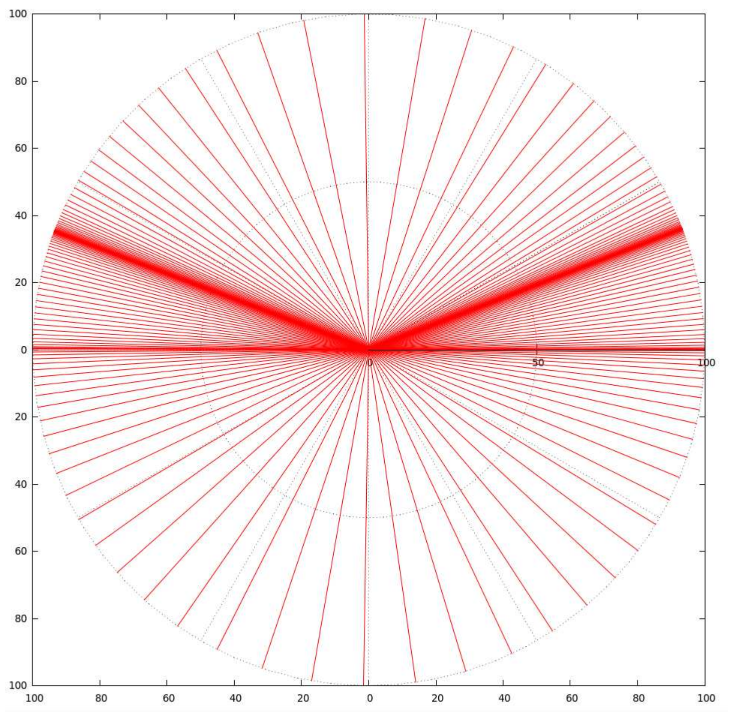


Figure 8.9: Target Domain for Robotic Arm

Finally, to map configuration A to configuration B, a path is required that joins two points without passing outside of the domain. The two points are mapped forward into the convex target domain. These points are then joined by a straight line in the target domain. This line is mapped back to the source domain. As discussed in Section 2.3.3, this line remains inside the domain and provides a valid path between the two points. This is illustrated in Figures 8.10 and 8.11.

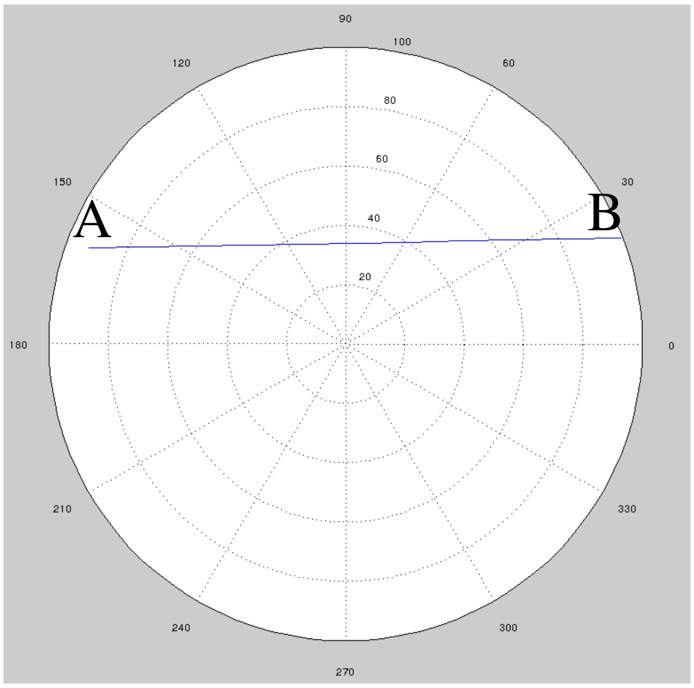


Figure 8.10: Planned Path in the Target Domain

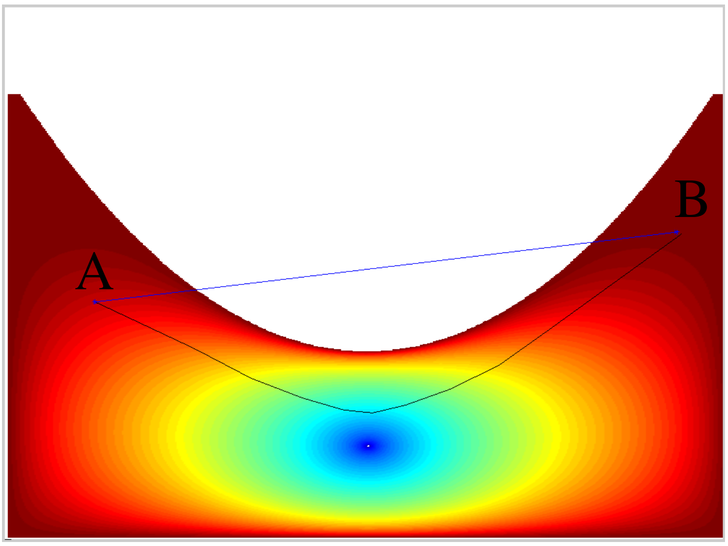
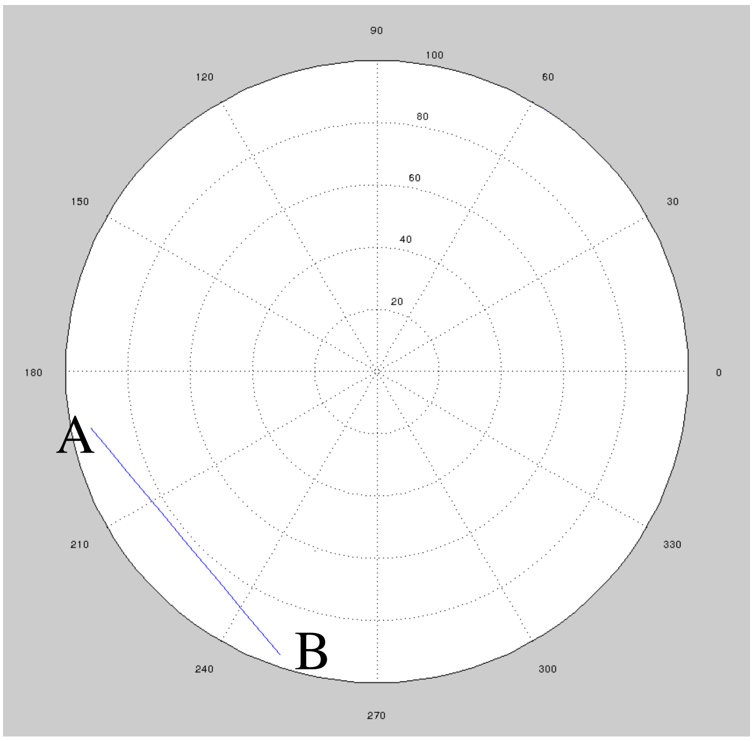
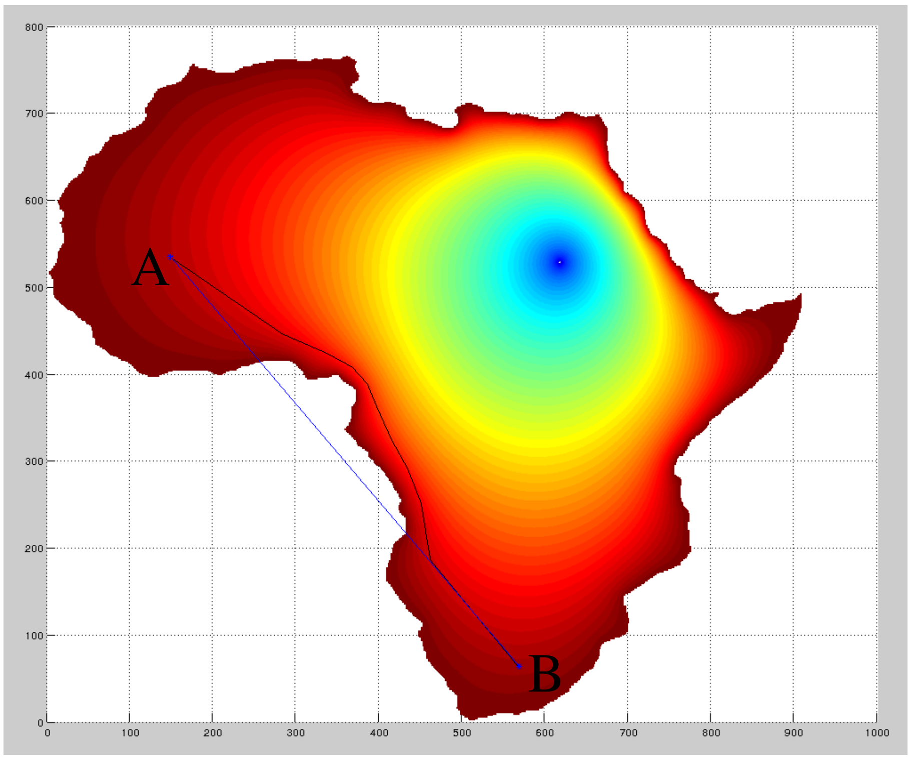


Figure 8.11: Planned Path in the Source Domain

As further examples of the approach, the method is applied to the domains shown in Figures 8.3, 8.4 and 8.5. Again it is noted that these domains are *configuration spaces* and do not necessarily correspond to physical domains.

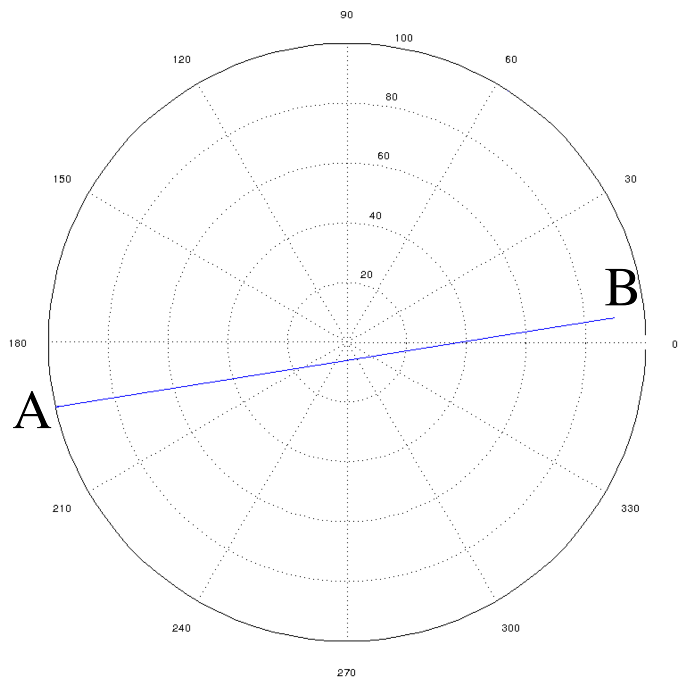


(a) Target Domain

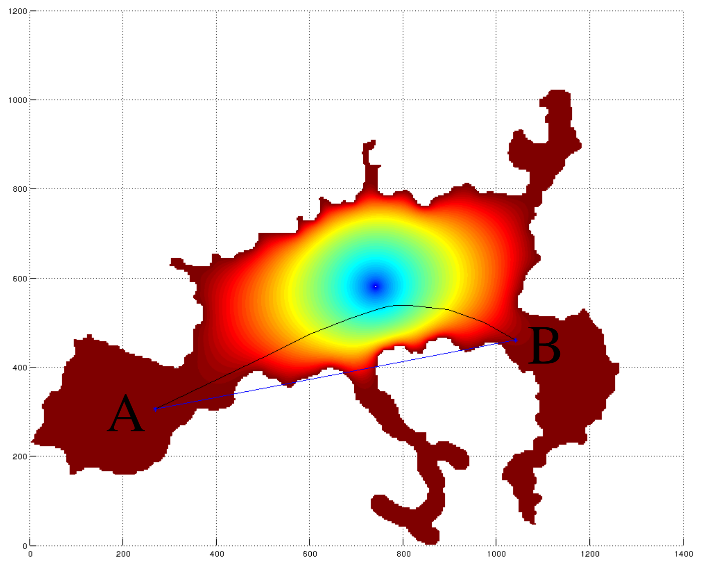


(b) Source Domain

Figure 8.12: Path Planning over the Africa Shaped Configuration Space

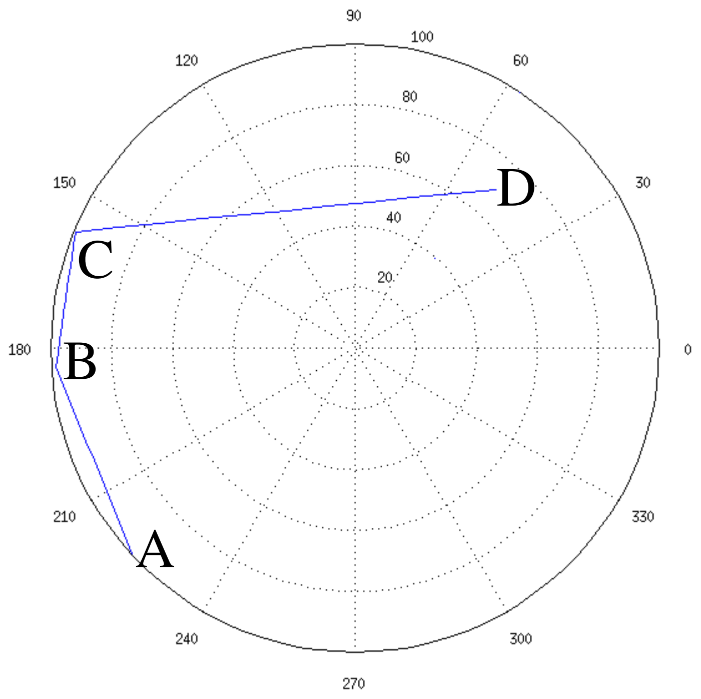


(a) Target Domain

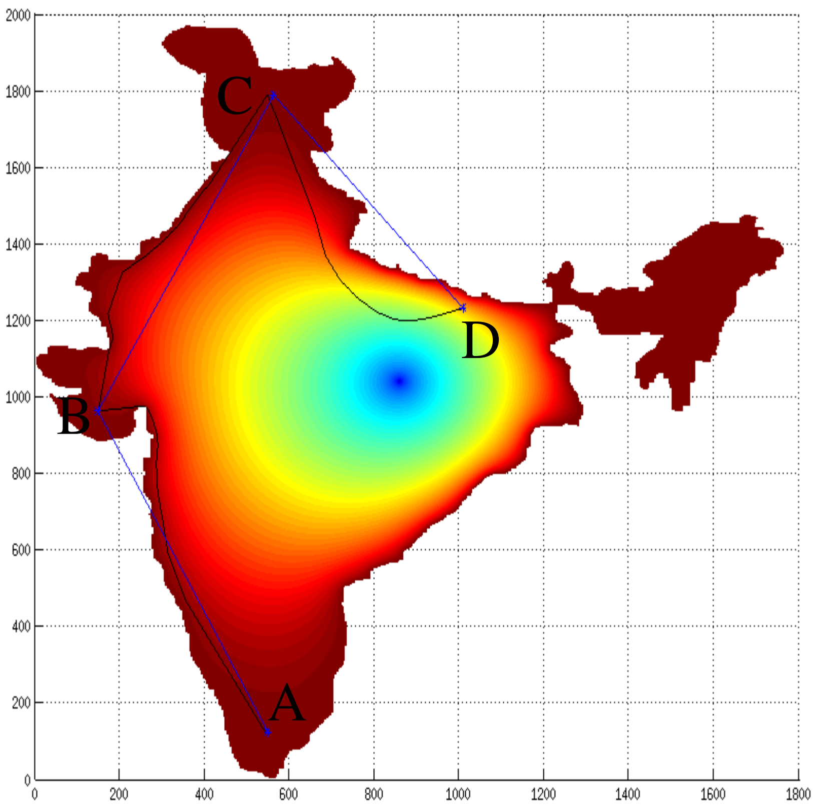


(b) Source Domain

Figure 8.13: Path Planning over the Europe Shaped Configuration Space



(a) Target Domain

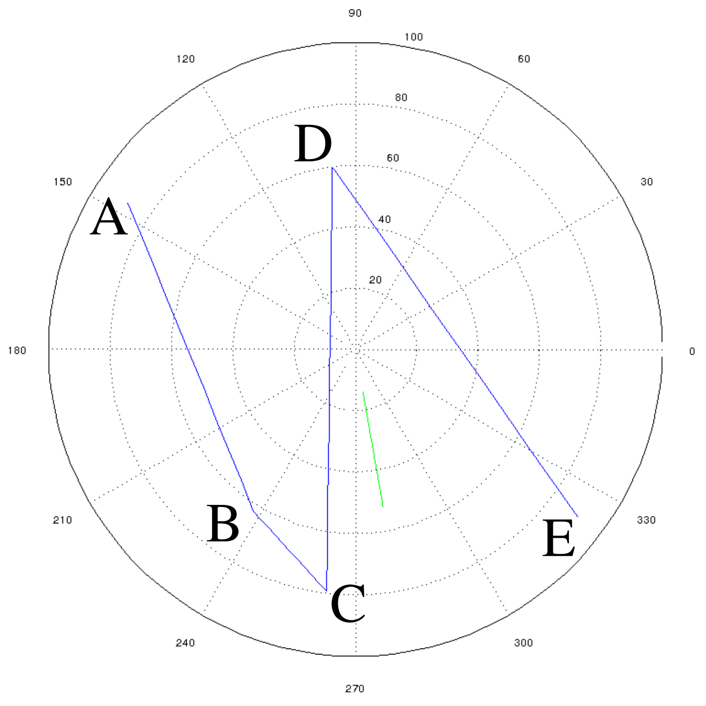


(b) Source Domain

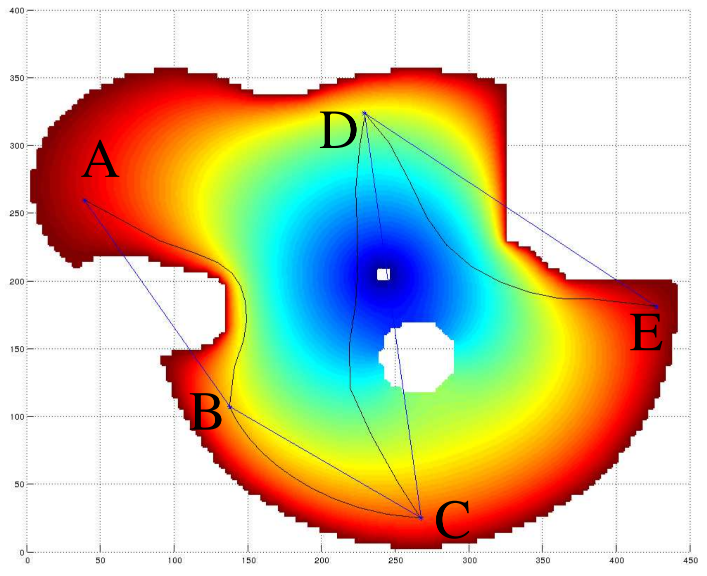
Figure 8.14: Path Planning over the India Shaped Configuration Space

8.3.1 Genus-∅

When performing path planning in genus-∅ domains, the process is the same as above. It is important that the path in the target domain does not intersect the radial line or arc representing the void. Figures 8.15 and 8.16 show path planning for an insulator and perfect conductor respectively.

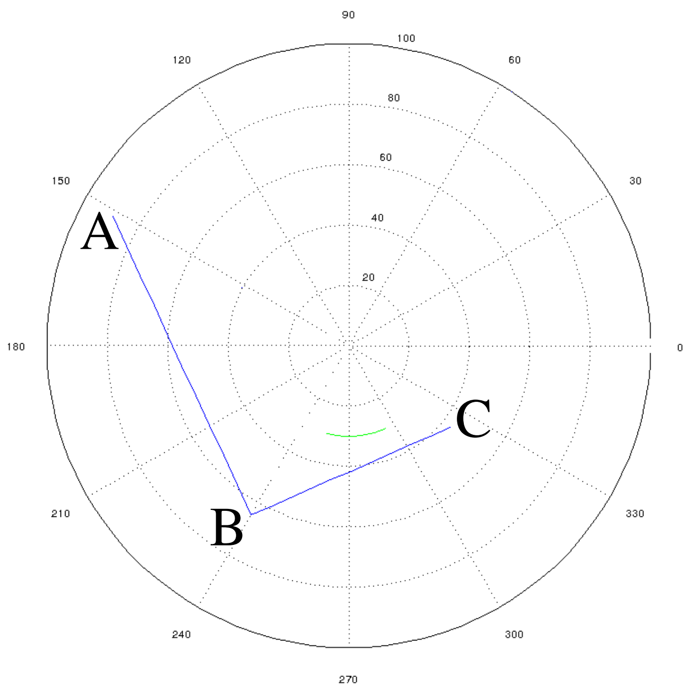


(a) Path in the Target Domain

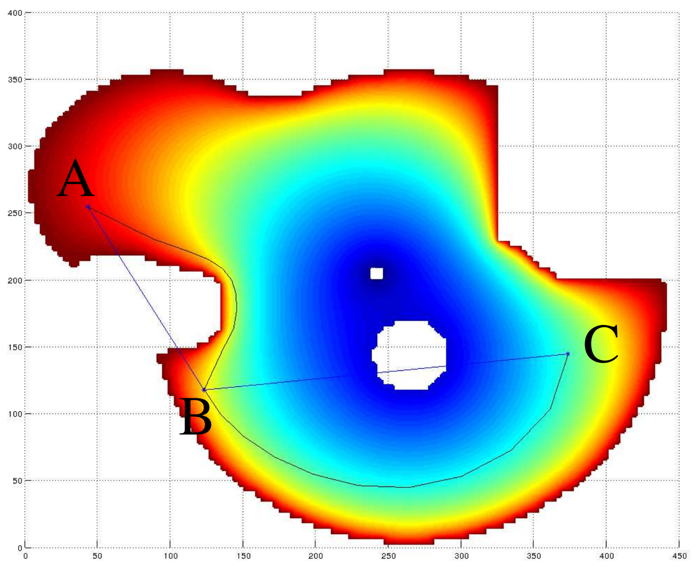


(b) Source Domain

Figure 8.15: Path Planning with an Insulating Void



(a) Path in the Target Domain

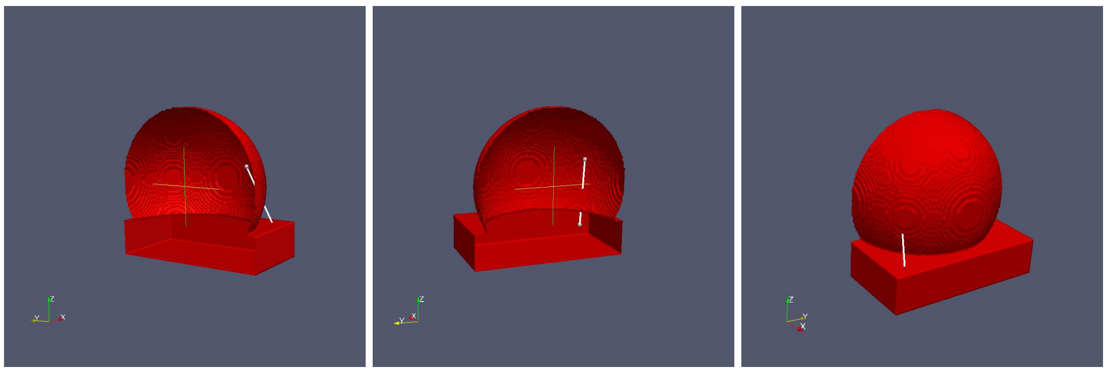


(b) Source Domain

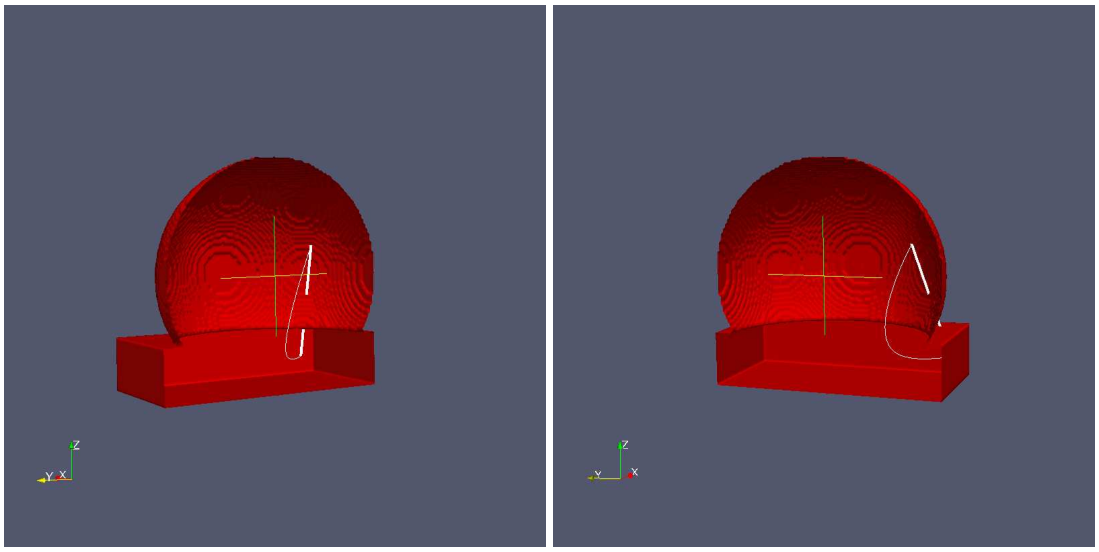
Figure 8.16: Path Planning with an Infinite Conductor

8.4 Robot Path Planning in 3D

Path planning in 3D is the same as in 2D. Figure 8.17 below shows the computed path for the straight line inside the spherical domain.



(a) Invalid Path in Non-Convex Domain



(b) Mapped Path

Figure 8.17: 3D Path Planning

8.5 Meshing in 2D

To mesh or remesh a domain, it should first be mapped to the regular convex domain. In 2D this means mapping the source domain to the target polar domain. One can then construct any mesh over the convex domain and transfer it back to the source domain.

This is illustrated in Figure 8.18 below. This mesh is constructed by placing a 40×40 rectangular grid over the polar domain. Points that lie outside the polar domain are deleted. These points are then mapped back to the source domain forming a mesh with a total of 1600 points, compared to 696294 points originally – only 0.23% of the original number of points. Figure 8.19 is constructed in the same manner, but with a 100×100 rectangular grid on the polar domain. This results in 10000 points – 1.44% of the original mesh size. Both Figures illustrate how the mesh density increases around non-convex regions, which is desirable.

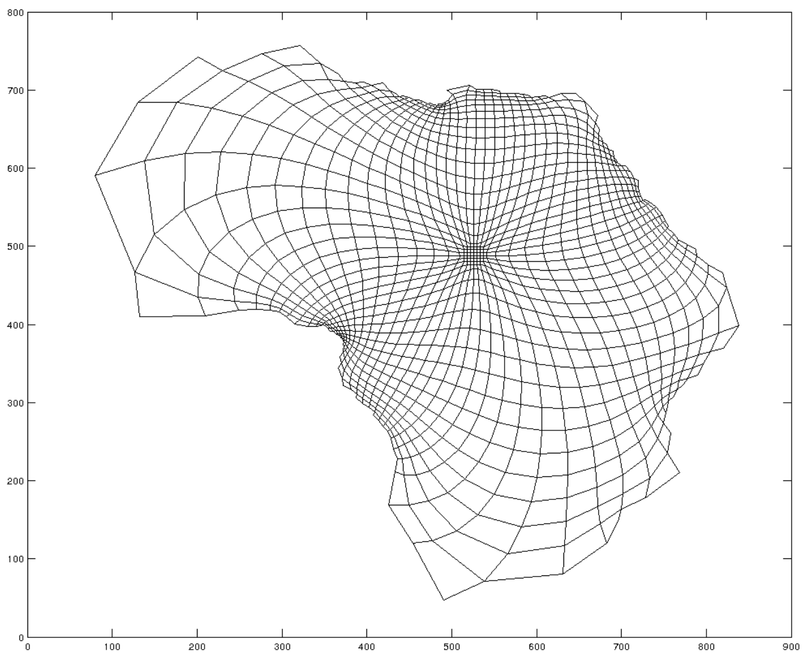


Figure 8.18: Very Low Resolution Mesh: 1600 Points, 0.23% of the original mesh

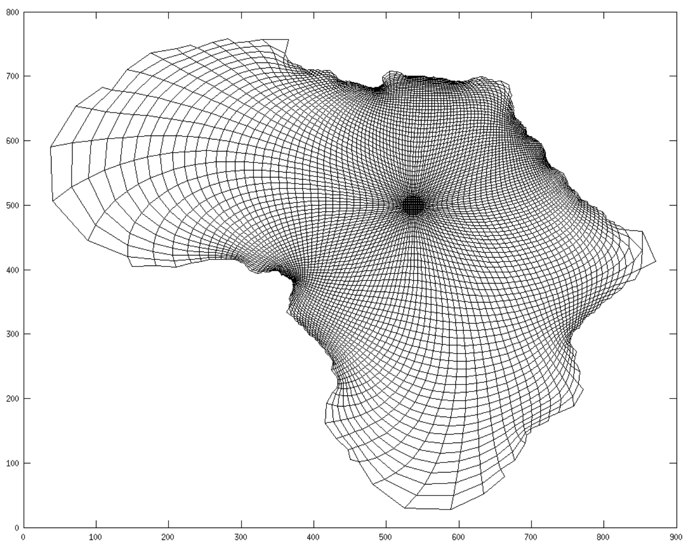


Figure 8.19: Low Resolution Mesh: 10000 Points, 1.44% of the original mesh

8.6 Conclusion

This chapter showed applications of domain mapping in 2D and 3D. Path planning in 2D in both genus-0 and genus-$\emptyset$ domains was illustrated on a number of different domains. Meshing was also shown over a 2D genus-0 domain. Finally, path planning in 3D genus-0 was shown. These examples illustrate the usefulness of the system with regards to these two problems.

Chapter 9

Conclusions, Contribution and Future Work

9.1 Conclusion

This research focused on the implementation and extension of the method for domain mapping using harmonic functions proposed in Voruganti *et al.* [2006 2008]. Domain mapping is desirable as it allows one to map some geometrically complex domain to a regular one where many mathematical operations are more efficient. It is important that the map maintains a number of properties so that the existence of a smooth inverse map is guaranteed. The method upon which this research is based ensures that the map is smooth, continuous and bijective; hence diffeomorphic. There are a number of applications ranging, from robot path planning to engineering and video animation tweening.

Domain mapping suffers from the fact that the calculation of the map is usually an expensive task. However, this map only needs to be calculated once for a given domain and lookups that follow can be done in constant time.

The methods outlined in Voruganti *et al.* [2006 2008] focused on constructing a map from a genus-0 domain to a topologically equivalent sphere. This was accomplished through the use of an artificial potential field approach with a novel method of parametrising the domain once the field was generated. This research developed an extension to the original approach to allow the mapping of 2D source domains that contain voids i.e. the domain is no longer simply connected. The original method, as well as the extensions were implemented and successfully tested in MATLAB.

The extension worked by applying various potential-insulating and potential-conducting boundary conditions to the voids. By using these boundary conditions, the system could transform the voids in the source domain into radial lines or arcs in the target domain respectively. By using a combination of these methods, the voids can be transformed into a single point in the target domain. It is noted that the boundary of each void is condensed into fewer overlapping points, thus the map is no longer diffeomorphic in that local area. This is considered acceptable in most situations as one can still get as close to the boundary as the numerical accuracy allows. The methods proposed to handle voids were only developed to work in 2D. These methods should work in higher dimensions as long as the void is wholly encased in the source domain, such as a bubble inside a 3D domain that does not touch the domain boundary. Formally, the complement of the domain should be disconnected.

These methods were then translated from serial MATLAB code into serial C++ code using the PETSc mathematics library. This serial C++ code was updated to run in parallel using MPI and then extended to run in an arbitrary number of dimensions. There are portions of the serial code that are embarrassingly parallel and the system thus lends itself to a parallel architecture. This was confirmed when the runtime of the parallel implementation was compared for different numbers of cores. The parallel system ran significantly faster and scaled much better than the serial version as the problem size and number of cores were scaled up.

Investigations revealed how the parallel system scaled with the number of CPU's and it is noted that some parts of the process are inherently serial, others are embarrassingly parallel and other sections fall somewhere in between. The final class of methods sees improvement as the hardware scales, but only up to a point where the overhead associated with communication and synchronisation surpasses the gains. One such example is the solution to the linear system describing Laplace's equation. More CPU's does not necessarily mean that the system will be solved faster. This relationship was investigated and discussed.

Applications of domain mapping were successfully run on both the University of the Witwatersrand's *Hydra* cluster as well as the Centre for High Performance Computing's *Sun Hybrid* cluster. These simulations investigated the applications of the developed system to meshing and robot path planning in two and three dimensions.

In conclusion, the research hypothesis outlined in Chapter 3 is accepted as the methods discussed were all implemented and gave correct results. The implementations were able to handle large domains and were able to perform work efficiently in parallel when speed and memory constraints of a single machine were insufficient to solve a large problem.

Similarly the research questions from Chapter 3 were investigated. Different Neumann and Dirichlet boundary conditions were investigated. The primary issue with different boundary conditions was the introduction of local extrema. These local extrema stopped the streamlines from reaching the shape centre, which invalidates the map. It was found, however, that the two specific cases explained in Section 4.4 could be used to control the shape of voids in the target domains without adversely affecting the quality of the map. The choice of centre was not directly investigated, although it was found that it should be as far away from boundaries as possible to avoid numerical errors as the potential was no longer spread evenly throughout the domain. Finally the parts of the algorithm appropriate for parallelisation were identified and tested in a parallel implementation. This parallel implementation was successful as a significant performance increase was measured in comparison to the serial versions of the code.

9.2 Contributions and Future Work

This work contributed a new extension to domain mapping that allows for 2D source domains of genus-$\emptyset$ and lays the foundation for research in higher dimensions. This part of the research was presented and published in Klein *et al.* [2012]. It also contributes both a serial and parallel implementation of the 2D extension for non-simply-connected domains as well as an implementation that handles arbitrary dimension genus-0 domains. Finally, a general framework for meshing and robot path planning was developed that could be used to solve these problems in arbitrary genus-0 domains.

The parallel implementation is a significant practical contribution that allows users to map much larger domains at much higher resolutions than previously possible. It allows the domain size to scale with hardware making use of the performance gains given by parallel architectures.

There are multiple avenues open for future work. From a theoretical point of view research should be done on methods of handling high dimension genus-$\emptyset$ domains – particularly three dimensions as there are many applications of this result. An in-depth investigation on the choice of the shape centre should be performed and a method developed to automatically choose this point from the optimization of some metric based either numerical accuracy or the computational resources required.

From a practical point of view, there is an opening for research into both the efficiency and numerical accuracy of the methods. By solving Laplace's equation over the source domain using some multi-grid or spectral method, the amount of work necessary for some required numerical accuracy may be improved. More detailed investigations of the parallel methods of solving the system could also be performed as well as a comparison of optimized MATLAB code against the optimized C++ code in both serial and parallel environments. Finally, the current implementations should be re-factored and organised into some parallel library that makes the relevant methods available to other software.

Appendix A

Pseudo Code

Algorithm 1 2D Genus-0 Source Domains

```
 1: // Read & Construct Domain
 2: Read the domain from the file
 3: Crop Domain
 4: Find Boundaries                                         } $O(n^2)$
 5: Set Centre
 6: $n, m \leftarrow$ dimensions of the grid
 7: $b \leftarrow$ number of boundary points
 8: // Construct & Solve Finite Difference Matrix
 9: $a \leftarrow 0$          ▷ Assign internal/boundary points cols in the FD matrix
10: for all points $(i, j)$ in Domain do
11:     if $(i, j)$ is not an external pixel then
12:         $coords(i, j) \leftarrow a$                     } $O(n^2)$
13:         $a$++
14:     end if
15: end for
16: $A$ = Sparse $a \times a$ matrix of 0's    ▷ Construct Finite Difference Matrix
17: $b$ = Vector of length $a$
18: $currentrow \leftarrow 0$
19: for all points $(i, j)$ in Domain do
20:     if $(i, j) \in$ Interior then
21:         $FD[currentRow, coords(i, j)] = -4$
22:         $FD[currentRow, coords(i - 1, j)] = 1$
23:         $FD[currentRow, coords(i + 1, j)] = 1$
24:         $FD[currentRow, coords(i, j - 1)] = 1$
25:         $FD[currentRow, coords(i, j + 1)] = 1$
26:         $b[currentRow] = 0$
27:         $currentRow$++
28:     else if $(i, j)$ is a boundary point then          } $O(n^2)$
29:         $FD[currentRow, coords(i, j)] = 1$
30:         $b[currentRow] = 1$
31:         $currentRow$++
32:     else if $(i, j)$ is the centre point then
33:         $FD[currentRow, coords(i, j)] = 1$
34:         $b[currentRow] = 0$
35:         $currentRow$++
36:     end if
37: end for
38: Solve $Ax = b$
39: Place the corresponding values from $x$ into $\phi$.   } $O(n^3 + n^2)$
40: // Phase 3: Solve Streamlines and Parametrisation
41: $l \leftarrow$ limit for the length of the longest streamline
42: for all points $(i, j)$ on the domain boundary do
43:     currentStream = ODE$(i, j, \phi)$
44:     $(i_{end}, j_{end}) \leftarrow$ Last point in the streamline   } $O(b + l) \in O(n^2)$
45:     $\theta = \arctan(\frac{i_{end} - i_{centre}}{j_{end} - j_{centre}})$
46: end for
```

Algorithm 2 2D Genus-$\emptyset$ Source Domains (Perfect Conductor)

```
 1: Perform Lines $1 - 18$ from Algorithm 1
                          ⋮
18: $currentrow \leftarrow 0$
19: for all points $(i, j)$ in Domain do
20:     if $(i, j) \in$ Interior then
                          ⋮
28:     else if $(i, j)$ is a boundary point then
                          ⋮
32:     else if $(i, j)$ is the centre point then
                          ⋮
36:     else if $(i, j)$ is a void boundary point then
37:         if $(i, j)$ is the first boundary point for this void then
38:             $(i_{first}, j_{first}) = (i, j)$
39:             $FD[currentRow, coords(i, j)] = 1$
40:             $FD[currentRow, coords($neighbouring interior point$)] = -1$
41:         else
42:             $FD[currentRow, coords(i, j)] = 1$
43:             $FD[currentRow, coords(i_{first}, j_{first})] = -1$
44:         end if
45:         $b[currentRow] = 0$
46:         $currentRow$++
47:     end if
48: end for
49: Solve $Ax = b$
50: Place the corresponding values from $x$ into $\phi$.
51: // Phase 3: Solve Streamlines and Parametrisation
52: $l \leftarrow$ limit for the length of the longest streamline
53: for all points $(i, j)$ on the domain boundary do
54:     currentStream = ODE$(i, j, \phi)$
55:     if currentStream stops at a void then
56:         Restart streamline on other side of void
57:     end if
58:     $(i_{end}, j_{end}) \leftarrow$ Last point in the streamline
59:     $\theta = \arctan(\frac{i_{end} - i_{centre}}{j_{end} - j_{centre}})$
60: end for
```

Algorithm 3 2D Genus-$\emptyset$ Source Domains (Insulator)

```
1:  Perform Lines $1 - 18$ from Algorithm 1
                              ⋮
18: $currentrow \leftarrow 0$
19: for all points $(i, j)$ in Domain do
20:     if $(i, j) \in$ Interior then
                              ⋮
28:     else if $(i, j)$ is a boundary point then
                              ⋮
32:     else if $(i, j)$ is the centre point then
                              ⋮
36:     else if $(i, j)$ is a void boundary point then
37:         Find neighbouring interior pixel
38:         $FD[currentRow, coords(i, j)] = 1$
39:         $FD[currentRow, coords(interiorNeighbour)] = -1$
40:         $b[currentRow] = 0$
41:         $currentRow$++
42:     end if
43: end for
44: Solve $Ax = b$
45: Continue as in Algorithm 2
                              ⋮
```

Algorithm 4 2D Parallel Mapping

```
// Read & Construct Domain
Rank = MPI Rank
$p$ ←MPI Communicator Size (Number of Processes)
**if** MPI Rank = 0 **then**
    Read the domain from the file
    Crop Domain
    Find Boundaries
    Set Centre
    $n, m \leftarrow$ dimensions of the grid
    $b \leftarrow$ number of boundary points
    Broadcast information to other processes
**else**
    Wait for broadcast from Process 0
**end if**
// Construct & Solve Finite Difference Matrix
$numRows \leftarrow n/p$
$RowStart \leftarrow$ Rank $* \, numRows$
$RowStop \leftarrow$ (Rank $+$ 1) $* \, numRows$
**if** MPI Rank = 0 **then**
    $a \leftarrow 0$
**else**
    Wait for $a$ value from MPI Rank$-1$
**end if**
**for all** points $(i, j)$ in Domain where $RowStart \leq i < RowStop$ **do**
    **if** $(i, j)$ is not an external pixel **then**
        $coords(i, j) \leftarrow a$
        $a$++
    **end if**
**end for**
**if** MPI Rank $\neq p - 1$ **then**
    Send $a$ to (Rank$+1$)
    $totalNonExternal \leftarrow$ Wait for broadcast from Rank $p - 1$
**else**
    Broadcast final $a$ to all processes
    $totalNonExternal \leftarrow a$
**end if**
$A =$ Sparse $totalNonExternal \times totalNonExternal$ matrix of 0's
$b =$ Vector of length $totalNonExternal$
```

Algorithm 5 2D Parallel Mapping (continued)

```
39: $currentrow \leftarrow RowStart$
40: **for all** points $(i, j)$ in Domain where $RowStart \leq i < RowStop$ **do**
41:     As in Algorithms 1, 2 or 3
                              ⋮
42: **end for**
43: MPI Block – wait for all processes for finish local matrix construction
44: Assemble Matrix
45: Solve $Ax = b$
46: **for** $rowStart \leq row < rowStop$ **do**
47:     Place the corresponding values in $row$ from $x$ into $\phi$.
48: **end for**
49: Synchronise all rows of $\phi$
50: // Phase 3: Solve Streamlines and Parametrisation
51: $l \leftarrow$ limit for the length of the longest streamline
52: Let $b_i \in B$ be the domain boundary points, such that $b_i$ is adjacent to $b_{i-1}$ and $b_{i+1}$
53: **for** $(i = \text{Rank}; i < |B|; i \leftarrow i + p)$ **do**
54:     As in Algorithms 1, 2 or 3
                              ⋮
55: **end for**
56: Save Boundary Parametrisation back to Rank 0
```

Algorithm 6 3D Mapping

```
 1: // Read & Construct Domain
 2: As in Algorithm 1, Lines 1-5
                                    ⋮
 6: $n, m, p \leftarrow$ dimensions of the grid
 7: $b \leftarrow$ number of boundary points
 8: // Construct & Solve Finite Difference Matrix
 9: $a \leftarrow 0$                    ▷ Assign internal/boundary points cols in the FD matrix
10: for all points $(i, j, k)$ in Domain do
11:     if $(i, j, k)$ is not an external pixel then
12:         $coords(i, j, k) \leftarrow a$
13:         $a$++
14:     end if
15: end for
16: $A =$ Sparse $a \times a$ matrix of 0's          ▷ Construct Finite Difference Matrix
17: $b =$ Vector of length $a$
18: $currentrow \leftarrow 0$
19: for all points $(i, j, k)$ in Domain do
20:     if $(i, j) \in$ Interior then
21:         $FD[currentRow, coords(i, j, k)] = -6$
22:         $FD[currentRow, coords(i-1, j, k)] = 1$
23:         $FD[currentRow, coords(i+1, j, k)] = 1$
24:         $FD[currentRow, coords(i, j-1, k)] = 1$
25:         $FD[currentRow, coords(i, j+1, k)] = 1$
26:         $FD[currentRow, coords(i, j, k-1)] = 1$
27:         $FD[currentRow, coords(i, j, k+1)] = 1$
28:         $b[currentRow] = 0$
29:         $currentRow$++
30:     else if $(i, j)$ is a boundary point then
31:         $FD[currentRow, coords(i, j, k)] = 1$
32:         $b[currentRow] = 1$
33:         $currentRow$++
34:     else if $(i, j)$ is the centre point then
35:         $FD[currentRow, coords(i, j, k)] = 1$
36:         $b[currentRow] = 0$
37:         $currentRow$++
38:     end if
39: end for
40: Solve $Ax = b$
41: Place the corresponding values from $x$ into $\phi$.
42: // Phase 3: Solve Streamlines and Parametrisation
43: $l \leftarrow$ limit for the length of the longest streamline
44: for all points $(i, j, k)$ on the domain boundary do
45:     currentStream = ODE$(i, j, k, \phi)$
46:     $(i_{end}, j_{end}, k_{end}) \leftarrow$ Last point in the streamline
47:     $\theta_1, \theta_2 =$Approaching angle of streamline
48: end for
```

Appendix B

3D Stereoscopic Images

The images in this appendix are all stereoscopic.

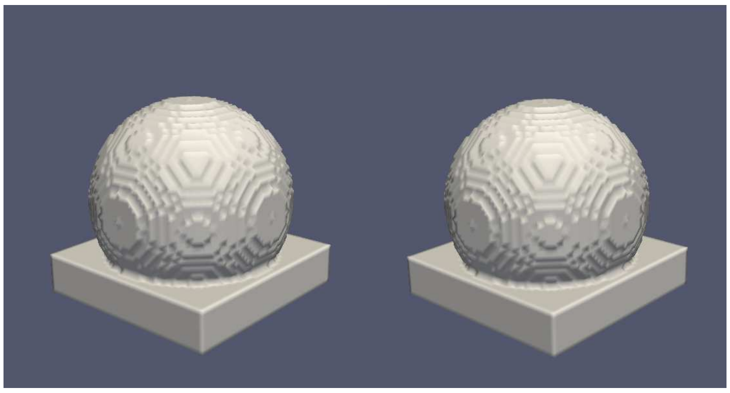


Figure B.1: Stereoscopic 3D Domain

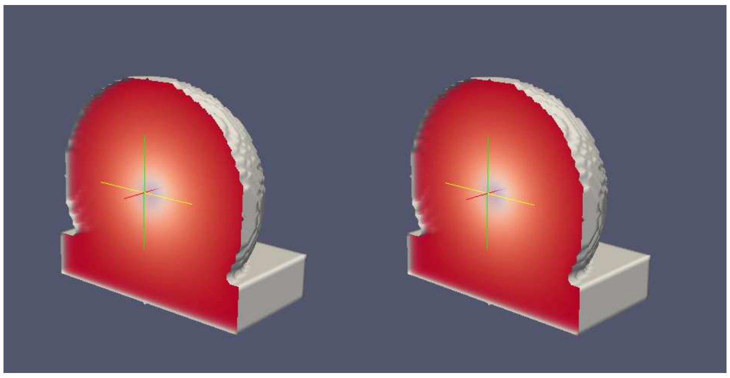


Figure B.2: Stereoscopic 3D Potential Field

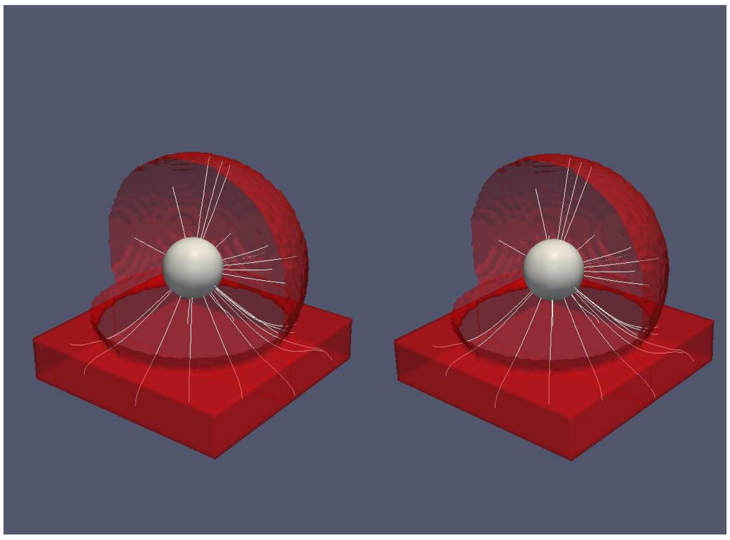


Figure B.3: Stereoscopic 3D Streamlines

References

[Aliaga *et al.* 2004] J Aliaga, F Almeida, JM Badia, S Barrachina, V Blanco, M Castillo, U Dorta, R Mayo, ES Quintana, G Quintana, C Rodriguez, and F de Sande. Parallelization of the GNU Scientific Library on heterogeneous systems. In *Parallel and Distributed Computing, 2004. Third International Symposium on/Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks.*, pages 338–345, December 2004.

[Axler *et al.* 2001] Sheldon Axler, Paul Bourdon, and Wade Ramey. *Graduate Texts in Mathematics, Harmonic Function Theory*. Springer, Second edition, 2001.

[Bahi *et al.* 2011] Jacques M. Bahi, Raphaël Couturier, and Lilia Ziane Khodja. Parallel gmres implementation for solving sparse linear systems on gpu clusters. In *Proceedings of the 19th High Performance Computing Symposia*, HPC '11, pages 12–19, San Diego, CA, USA, 2011. Society for Computer Simulation International.

[Balay *et al.* 1997] Satish Balay, William D. Gropp, Lois C. McInnes, and Barry F. Smith. Effecient Management of Parallelism in Object Oriented Numerical Software Libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[Balay *et al.* 2011] Satish Balay, William D. Gropp, Lois C. McInnes, and Barry F. Smith. *PETSc Users Manual*. Technical Report ANL-95/11 - Revision 3.2, Argonne National Laboratory, 2011.

[Balay *et al.* 2012] Satish Balay, William D. Gropp, Lois C. McInnes, and Barry F. Smith. PETSc Home Page. http://www.mcs.anl.gov/petsc, 2012.

[Barraquand and Latombe 1991] J Barraquand and JC Latombe. Robot Motion Planning: A Distributed Representation Approach. *International Journal of Robotics Research*, 10:628–649, 1991.

[Chang *et al.* 2006] Yen-Tuo Chang, Bing-Yu Chen, Wan-Chi Luo, and Jian-Bin Huang. Skeleton-driven animation transfer based on consistent volume parameterization. In *Proceedings of the 24th international conference on Advances in Computer Graphics*, CGI'06, pages 78–89, Berlin, Heidelberg, 2006. Springer-Verlag.

[CHPC 2012] *Sun Hybrid System*, 2012. Retrieved 13 December 2012, from `http://www.chpc.ac.za/sun/`

[Cunha and Hopkins 1994] RudneiDias Cunha and Tim Hopkins. A parallel implementation of the restarted gmres iterative algorithm for nonsymmetric systems of linear equations. *Advances in Computational Mathematics*, 2:261–277, 1994.

[Fuhrmann *et al.* 2010] Simon Fuhrmann, Jens Ackermann, Thomas Kalbe, and Michael Goesele. Direct Resampling for Isotropic Surface Remeshing. In *Proceedings of Vision, Modeling and Visualization 2010*, Siegen, Germany, 2010.

[Galassi *et al.*] M Galassi, J Davies, J Theiler, B Gough, G Jungman, P Alken, M Booth, and F Rossi. *GNU Scientific Library Reference Manual*. ISBN 0954612078, http://www.gnu.org/software/gsl/, 3rd edition.

[Han *et al.* 2010] Shuchu Han, Jiazhi Xia, and Ying He. Hexahedral Shell Mesh Construction via Volumetric Polycube Map. In *Proceedings of the 14th ACM Symposium on Solid and Physical Modeling*, SPM '10, pages 127–136, New York, NY, USA, 2010. ACM.

[Khatib 1986] O Khatib. Real-Time Obstacle Avoidance for Manipulators and Mobile Robots. *International Journal of Robotics Research*, 5(1):90–98, 1986.

[Klein *et al.* 2012] Richard Klein, Hari K. Voruganti, and Michael Sears. Domain Mapping using Harmonic Functions in non-convex domains of genus non-zero. In *28th European Workshop on Computational Geometry*, 2012.

[Latombe 1991] J-C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers Group, Eighth edition, 1991.

[Martin *et al.* 2009] T Martin, E Cohen, and RM Kirby. Volumetric parameterization and trivariate B-spline fitting using harmonic functions. *Computer Aided Geometric Design*, 26:648–664, 2009.

[MathWorks 2003] MathWorks. *Picking up the Pace with the MATLAB Profiler*. Matlab Newsletters - MATLAB News & Notes, 2003. Retrieved May 2003, from `http://www.mathworks.com/company/newsletters/news_notes/may03/profiler.html`

[MathWorks 2012] MathWorks. *Vectorization*. Matlab R2012b Documentation, 2012. Retrieved November 2012, from `http://www.mathworks.com/help/matlab/matlab_prog/vectorization.html`

[Prasad *et al.* 1995] K.G. Prasad, DE Keyes, and JH Kane. *GMRES for sequentially multiple nearby systems*. Technical report, Technical Report , Old Dominium University, 1995.

[Press *et al.* 2007] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes, The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 2007.

[Saad 1989] Youcef Saad. Krylov Subspace Methods on Supercomputers. *Society for Industrial and Applied Mathematics, Journal on Scientific and Statistical Computing*, 10(6):1200 – 1232, November 1989.

[Stallings 2013] William Stallings. *Computer Organization and Architecture, Designing for Performance*. Pearson Education Inc., Upper Saddle River, NJ, USA, Nineth edition, 2013.

[Sunder and Shiller 1997] S Sunder and Z Shiller. Optimal Obstacle Avoidance based on Hamilton-Jacobi-Bellman Equation. *IEEE Transactions on Robotics and Automaton.*, 13(2):305–310, 1997.

[Surayawamshi *et al.* 2003] AB Surayawamshi, MB Joshi, B Dasgupta, and A Biswas. Domain Mapping as an Expeditionary Strategy for Fast Path Planning. *Mechanism and Machine Theory*, 38(11):1237–1256, 2003.

[Tanenbaum 2006] Andrew S. Tanenbaum. *Structured Computer Organization*. Pearson Education Inc., Upper Saddle River, NJ, USA, Fifth edition, 2006.

[Voruganti and Dasgupta 2009] Hari K. Voruganti and Bhaskar Dasgupta. A geometric approach for segmentation and parameterization of molecular surfaces for docking. *International Journal of Computational Science*, 3(2), 2009.

[Voruganti *et al.* 2006] Hari K Voruganti, Bhaskar Dasgupta, and Günter Hommel. Harmonic Function based Domain Mapping Method for General Domains. *WSEAS Transactions on Computers*, 5(10):2495–2502, October 2006.

[Voruganti *et al.* 2008] Hari K Voruganti, Vikash Gupta, and Bhaskar Dasgupta. Domain Mapping for Spherical Parameterization of 3-D Domains using Harmonic Functions. In *IISc Centenary - International Conference on Advances in Mechanical Engineering (IC-ICAME), Bangalore, India.* July 2008.

[Wang and Chirikjian 2000] Y Wang and GS Chirikjian. A New Potential Field Method for Robot Path Planning. In *IEEE International Conference on Robotics and Automation.*, 2000.

[Weisstein 2011a] Eric W. Weisstein. *Mean-Value Property*. MathWorld–A Wolfram Web Resource, 2011. Retrieved 3 October 2011, from `http://mathworld.wolfram.com/Mean-ValueProperty.html`

[Weisstein 2011b] Eric W. Weisstein. *Simply Connected.* MathWorld–A Wolfram Web Resource, 2011. Retrieved 3 October 2011, from `http://mathworld.wolfram.com/SimplyConnected.html`

[Xia *et al.* 2010] Jiazhi Xia, Ying He, Xiaotian Yin, Shuchu Han, and Xianfeng Gu. Direct-Product Volumetric Parameterization of Handlebodies via Harmonic Fields. In *Shape Modeling International*, 2010.