SMITH, ELIZABETH CHATHAM. Application of the Meshless Local Petrov-Galerkin Method to Unsteady, Multi-Dimensional Fluid Dynamics with Interfaces. (Under the direction of Dr. Tarek Echekki.)

The Meshless Local Petrov-Galerkin (MLPG) method is a numerical framework for solving partial differential equations. This method is unique in that it uses the governing equations in the local symmetric weak form and does not rely on a prescribed node connectivity. MLPG does not specify the numerical tools used for accomplishing the required tasks. In this work, Gaussian quadrature was used for integration over one-dimensional and two-dimensional domains. The Moving Least Squares (MLS) method was used for approximation of data and spatial derivatives of the data. The primitive variable form of the Navier-Stokes equations was used. MLPG results for benchmark fluids applications as well as the Rayleigh Taylor Instability (RTI) are presented.

The Shape Function Interpolation Method (SFIM), a novel treatment for gradient type boundary conditions, is introduced. The SFIM method uses the MLS shape functions to identify the nodal boundary value that satisfies the prescribed gradient condition. A capability for computing a inward facing normal vector for a boundary node, based upon the other boundary nodes in the identified region, was also created. Using the standard method for the treatment of Neumann boundary conditions, the quadrature and test function require modification for new geometry types. This feature eliminates the need for modified quadrature and allows the weight function and MLS approximation to use their native Cartesian representations. These two additions minimize the phenomena or geometry specific coding required to apply MLPG to wide range of applications on complex domains.

SFIM is demonstrated using two-dimensional heat transfer problems on both a square and a square with a circular void. The lid-driven flow in a square cavity was used to validate the performance of SFIM. The Rayleigh-Taylor Instability (RTI) was modeled with MLPG and the SFIM boundary condition. The MLPG results show smooth movement of both the dense spike and less dense bubble. The solution showed no numerical artifacts and has good qualitative agreement with previous literature. The modeling of the Rayleigh-Taylor Instability adds immiscible fluid interfaces to the fluids applications that can be modeled using MLPG.

The MLPG implementation was done in object-oriented C++ using the Template Method Design pattern. This yielded a product that is maintainable as well as easily extended for new methodology features or applications. Recommendations for future work in the areas of optimization and parallelization and methodology improvements to support moving nodes are proposed. This research provides a new methodology for the enforcement of Neumann boundary conditions, a tool for the identification of local boundary normal vectors for any domain shape,
and extends the MLPG body of work to include immiscible fluid interface problems.
Application of the Meshless Local Petrov-Galerkin Method to Unsteady, Multi-Dimensional Fluid Dynamics with Interfaces

by

Elizabeth Chatham Smith

A dissertation submitted to the Graduate Faculty of North Carolina State University in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

Mechanical Engineering

Raleigh, North Carolina

2016

APPROVED BY:

____________________________  ______________________________
Dr. Jack Edwards               Dr. Kevin Lyons

____________________________  ______________________________
Dr. Xiao-Biao Lin              Dr. Erik Lowndes

____________________________
Dr. Tarek Echekki
Chair of Advisory Committee
BIOGRAPHY

Elizabeth C. Smith received her bachelor’s degree in Mechanical Engineering from North Carolina State University in 2003. She completed a master’s degree in Mechanical Engineering from Pennsylvania State University in 2005. Her masters thesis work involved the modeling of pulsatile blood flow in partially blocked arteries. Ms. Smith earned her Professional Engineering License in 2009 and has ten years of experience as a defense contractor doing work in the areas of computational modeling and software development.
ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Tarek Echekki, for providing guidance during this process. Also my committee members for the time they devoted to the preliminary and final examinations, and document draft reviews. Applied Research Associates, Inc provided funding for the a portion of the preliminary research. Dr. Bryan Susi and Dr. Erik Lowndes of ARA provided technical input, creative inspiration, and endless encouragement. Finally, I would like to thank my family for their patience and support.
TABLE OF CONTENTS

LIST OF TABLES .................................................. vii

LIST OF FIGURES ................................................ viii

Chapter 1 Introduction ............................................. 1
  1.1 MLPG Methodology ......................................... 2
    1.1.1 Local Symmetric Weak Form (LSWF) ................. 2
    1.1.2 Test Functions ....................................... 3
    1.1.3 Trial Function Approximation ....................... 3
    1.1.4 Numerical Integration ............................... 5
    1.1.5 Domain Discretization ............................... 6
    1.1.6 Boundary Conditions ............................... 7
    1.1.7 Parallelization .................................. 8
  1.2 MLPG Applications ....................................... 9
    1.2.1 Fluid Flows ....................................... 10
    1.2.2 Other Applications ............................... 12
  1.3 Summary ................................................ 13
  1.4 Objectives ............................................. 14
  1.5 Outline ............................................... 14

Chapter 2 MLPG Methodology .................................... 15
  2.1 Domain Discretization and Subdomain Creation ......... 15
    2.1.1 Weight Functions ................................. 16
  2.2 Treatment of the Governing Equations ................... 18
    2.2.1 Order of Derivatives ............................. 19
  2.3 Trial Function Approximation ........................... 19
    2.3.1 Moving Least Squares (MLS) ....................... 20
  2.4 Numerical Integration .................................. 25
    2.4.1 Gaussian Quadrature .............................. 25
  2.5 Linear System Solution ................................ 27

Chapter 3 Boundary Conditions Implementation ............... 28
  3.1 Dirichlet Conditions ................................... 28
    3.1.1 Example ........................................... 29
  3.2 Neumann Conditions .................................... 29
    3.2.1 Example .......................................... 30
  3.3 Periodic Conditions .................................... 33
    3.3.1 Example .......................................... 34

Chapter 4 Validation Cases ..................................... 36
  4.1 Stokes Flow ............................................ 36
    4.1.1 Implementation .................................... 37
    4.1.2 Results ........................................... 38
4.2 Laminar Channel Flow ........................................... 40
  4.2.1 Implementation ........................................... 40
  4.2.2 Pouiselle Flow ........................................... 40
  4.2.3 Couette Flow ........................................... 42
4.3 Conclusion ................................................... 43

Chapter 5 Shape Function Interpolation Method - Boundary Condition . . . 44
  5.1 Methodology ................................................ 44
  5.2 Computing Normals ........................................ 46
  5.3 Validation ................................................ 51
  5.4 Demonstration on Immersed Boundary Geometry ................. 53
  5.5 Conclusion ................................................. 55

Chapter 6 Lid-Driven Flow in a Square Cavity .......................... 57
  6.1 Governing Equations ......................................... 57
  6.2 Domain Characterization ..................................... 59
  6.3 Implementation ............................................ 60
    6.3.1 Matrix Construction ................................. 61
    6.3.2 Boundary Conditions ............................... 62
  6.4 Results ................................................... 63
    6.4.1 Reynolds Number 100 .............................. 63
    6.4.2 Additional Reynolds Number ....................... 65
  6.5 Conclusion ............................................... 66

Chapter 7 Rayleigh-Taylor Instability Implementation .................. 69
  7.1 Governing Equations ........................................ 69
    7.1.1 Boundary Conditions ............................... 72
    7.1.2 Interface ............................................ 73
  7.2 Implementation ........................................... 73
  7.3 Results ................................................... 75
    7.3.1 Growth Rate ....................................... 78
  7.4 Conclusions ................................................. 79

Chapter 8 Software Design ........................................... 80
  8.1 MLPGProblem Class (Template Method Design) .................. 81
  8.2 Execution Path ............................................ 82
  8.3 Data Classes ................................................ 84
    8.3.1 Node Class .......................................... 84
    8.3.2 Quadrature Class ................................... 84
    8.3.3 Shape Function Data Class ......................... 86
    8.3.4 Domain Class ....................................... 88
  8.4 Run Time .................................................. 89
  8.5 Conclusions ................................................. 92

Chapter 9 Conclusions and Recommendations for Future Work ............. 93
**LIST OF TABLES**

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Basis functions for MLS Approximation</td>
<td>21</td>
</tr>
<tr>
<td>3.1</td>
<td>Sample K Matrix for Periodic Case</td>
<td>34</td>
</tr>
<tr>
<td>4.1</td>
<td>Terms to be integrated to create the K matrix for Stokes Flow.</td>
<td>37</td>
</tr>
<tr>
<td>5.1</td>
<td>Finite Difference Operators</td>
<td>45</td>
</tr>
<tr>
<td>8.1</td>
<td>Average iteration runtime in seconds for each of three domain discretizations of the square lid-driven cavity problem.</td>
<td>90</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 2.1 Example of an arbitrarily shaped domain with non-uniform node placement with subdomains for interior node \((x)\) and boundary node \((y)\). ........................................ 16
Figure 2.2 Examples of the quartic spline in one dimension \((a)\) and in two dimensions with a rectangular region of influence \((b)\). ................................................................. 17
Figure 2.3 Examples of the first derivative of the quartic spline in the \(x\)-direction in one dimension \((a)\) and in a two dimensions with a rectangular region of influence \((b)\). ................................................................. 18
Figure 2.4 Input data for MLS example. Positions of the 1000 Halton points \((a)\) and the Franke’s function surfaces at the Halton Points \((b)\). ......................................................... 19
Figure 2.5 MLS Approximation of Franke’s function on a uniform grid. ............................. 23
Figure 2.6 MLS approximation using both representations of the derivative and different radius factors. ................................................................. 26
Figure 2.7 Quadrature point configurations in one and two dimensions. ................................. 29

Figure 3.1 MLPG results for two-dimensional Poisson’s with Dirichlet boundary conditions. ................................................................................................. 30
Figure 3.2 Domain and boundary condition diagram for two-dimensional Poisson’s with mixed boundary conditions. ......................................................... 31
Figure 3.3 MLPG results contour for two-dimensional Poisson’s with mixed boundary conditions. ................................................................................................. 33
Figure 3.4 MLPG results for the mixed boundary condition two-dimensional Poisson’s problem. Results on the boundaries \((a)\) and centerlines in the \(x\) and \(y\) directions \((b)\). ................................................................. 32
Figure 3.5 Diagram of nodes in periodic domain. ................................................................. 33
Figure 3.6 MLPG results for one-dimensional heat conduction with periodic boundary conditions. ................................................................................................. 35

Figure 4.1 MLPG results for Stokes Flow with Dirichlet conditions. ........................................ 38
Figure 4.2 Error contour for MLPG computation Stokes Flow with Dirichlet conditions. ........ 39
Figure 4.3 Contour plots of individual velocity component results for Stokes Flow. ................. 39
Figure 4.4 Comparison of transient MLPG results \((\text{symbols})\) and the analytical solution \((\text{solid line})\) of laminar Poisuielle flow. ................................................................. 40
Figure 4.5 Comparison of transient MLPG results \((\text{symbols})\) and the analytical solution \((\text{solid line})\) of laminar Couette flow. ................................................................. 41

Figure 5.1 Sample section of a closed domain with boundary nodes marked. ............................ 47
Figure 5.2 Sample subdomain for node of interest, in yellow, with boundary nodes shown in blue. ................................................................................................. 48
Figure 5.3 Depiction of local normal vector computation test points. ........................................ 48
Figure 5.4 Local normals for all boundary points \((\text{black})\) and the results local normal for the point of interest \((\text{blue})\). ................................................................. 49
Figure 5.5 Set of subdomain nodes rotated based on the local normal and the desired normal in the \(y\) direction. ................................................................. 50
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.6</td>
<td>Depiction of the final normals for each boundary node in the sample domain.</td>
</tr>
<tr>
<td>5.7</td>
<td>Boundary nodes and local normals.</td>
</tr>
<tr>
<td>5.8</td>
<td>Temperature contour at $t=1.0$.</td>
</tr>
<tr>
<td>5.9</td>
<td>Temperature contour at $t=2.5$.</td>
</tr>
<tr>
<td>5.10</td>
<td>Temperature contour at $t=15$.</td>
</tr>
<tr>
<td>5.11</td>
<td>Setup for heat conduction on an insulated square plate.</td>
</tr>
<tr>
<td>5.12</td>
<td>Plot of temperature profiles along the $y=0$ at different times. Lines are Finite Difference, Squares are MLPG, and Diamonds are MLPG with SFIM.</td>
</tr>
<tr>
<td>5.13</td>
<td>Temperature contour at $t=0.5$.</td>
</tr>
<tr>
<td>5.14</td>
<td>Temperature contour at $t=1.0$.</td>
</tr>
<tr>
<td>5.15</td>
<td>Temperature contour at $t=2.5$.</td>
</tr>
<tr>
<td>5.16</td>
<td>Temperature contour at $t=15$.</td>
</tr>
<tr>
<td>5.17</td>
<td>Diagram of immersed boundary heat transfer problem.</td>
</tr>
<tr>
<td>5.18</td>
<td>Temperature contour at $t=0.03$.</td>
</tr>
<tr>
<td>5.19</td>
<td>Temperature contour at $t=0.5$.</td>
</tr>
<tr>
<td>5.20</td>
<td>Temperature contour at $t=1$.</td>
</tr>
<tr>
<td>5.21</td>
<td>Steady state temperature contour.</td>
</tr>
<tr>
<td>5.22</td>
<td>Temperature profiles at three $x$ positions through the plate with the circular hole at $t=2$.</td>
</tr>
<tr>
<td>6.1</td>
<td>Diagram of lid-driven cavity in a square domain.</td>
</tr>
<tr>
<td>6.2</td>
<td>Lid-driven cavity nodal arrangement for node concentration factor of 1.7.</td>
</tr>
<tr>
<td>6.3</td>
<td>Velocity component profiles for the steady-state solution of the lid-driven flow in a square cavity with a Reynolds number of 100.</td>
</tr>
<tr>
<td>6.4</td>
<td>Velocity component profile development for the lid-driven flow in a square cavity with a Reynolds number of 100 using MLPG with SFIM.</td>
</tr>
<tr>
<td>6.5</td>
<td>Contour plots of velocity components for the lid-driven cavity at $t=10$ for the Reynolds number 100 case.</td>
</tr>
<tr>
<td>6.6</td>
<td>Contour plots of velocity and pressure for the lid driven cavity at $t=1$ for the Reynolds number 100 case.</td>
</tr>
<tr>
<td>6.7</td>
<td>Velocity component profiles for the steady-state solution of the lid-driven flow in a square cavity with a Reynolds number of 400.</td>
</tr>
<tr>
<td>6.8</td>
<td>Velocity component profiles for the steady-state solution of the lid-driven square cavity with a Reynolds number of 1000.</td>
</tr>
<tr>
<td>6.9</td>
<td>Velocity component profiles for the steady state-solution of the lid-driven flow in a square cavity with a Reynolds number of 3200.</td>
</tr>
<tr>
<td>7.1</td>
<td>Depiction of the RTI initial density distributions.</td>
</tr>
<tr>
<td>7.2</td>
<td>Diagram of RTI domain dimensions and boundary conditions.</td>
</tr>
<tr>
<td>7.3</td>
<td>Comparison of prescribed interface and nodal locations.</td>
</tr>
<tr>
<td>7.4</td>
<td>Density contours for the MLPG solution of RTI various time steps.</td>
</tr>
<tr>
<td>7.5</td>
<td>RTI $u$ velocity component (a), $v$ velocity component (b), and density contours (c) at a time of 1.0.</td>
</tr>
<tr>
<td>7.6</td>
<td>With Streamlines.</td>
</tr>
</tbody>
</table>
Figure 7.7  RTI $u$ velocity component (a), $v$ velocity component (b), and density contours (c) at a time of 2.0. .................................................. 77

Figure 7.8  Time evolution of the density interface position on the vertical surfaces in MLPG results of RTI. .................................................. 78

Figure 8.1  MLPGProblem class overview. .................................................. 81

Figure 8.2  Percentage of average iteration runtime in each CBS solution step. ........ 91

Figure 8.3  Percentage of average iteration runtime in the primary computational functions. 91
Chapter 1

Introduction

The conventional classes of computational methods, primarily Finite Element and Finite Difference, are well known to most researchers. There also exists a lesser known class of methods called meshless methods. Meshless methods began with the Smoothed-Particle Hydrodynamics (SPH) method in 1977 and evolved to include dozens of methods in the decades that followed. Meshless methods were created to address shortcomings of conventional methods, namely the tedious manual process of meshing or gridding the domains. This process is often performed multiple times (remeshing) during simulations. Remeshing is typically required when elements are distorted due to large deformations. Other benefits of meshless methods include a reduction in complexity and bookkeeping since pre-pecified or static relationships do not exist between nodes. Meshless methods also allow for insertion and removal of nodes during simulations without impact to the global system, thus avoiding the significant computational cost incurred by other methods.

In the class of meshless methods very few are truly meshless. Most use a meshless system of nodes to describe the domain but rely on a support or background mesh for integration. It can be said that the Meshless Local Petrov-Galerkin (MLPG) method is the only truly meshless method. The MLPG method was initially defined by Atluri and Zhu [1] as using the local weak form of the governing equations and having a test and trial function that do not necessarily share the same solution space or solution method. Thakur et al. [2] phrased it as “the MLPG method provides a rational basis for constructing meshfree methods with a greater degree of flexibility”. This definition encompasses a large number of different methodology formulations which were developed in the years after Atluri and Zhu’s initial publication. MLPG methods include Direct MLPG [3], Local Boundary Integral Equation (LBIE), and many more unnamed methods. The approaches are unique in the selection of test and trial functions, function approximation method, numerical integration method, etc.

The remainder of this work is divided into a methodology section, an applications section,
and a conclusion section. The methodology section covers (1) the relevant details of each MLPG component, (2) the introduction of a novel approach to modeling Neumann boundary conditions, (3) demonstration of the MLPG method using benchmark fluid dynamics problems, (4) the application of MLPG to fluid interfaces, and (5) a discussion of the software design and implementation.

1.1 MLPG Methodology

The MLPG approach creates a system of equations via a node-by-node analysis. The equations are then solved via the common stiffness matrix \( K \), unknowns vector \( \vec{x} \) and forces vector \( \vec{f} \) equation:

\[
K \vec{x} = \vec{f}
\]  

(1.1)

Each node gathers information from a subdomain, a region around the node. Subdomains are usually selected as simple shapes, circles or squares in two dimensions and spheres or cubes in three dimensions. Overlapping of subdomains is allowed, and is very common. For an accurate solution the union of the subdomains must cover the entire global domain. Intersections between subdomains and global boundaries are one of the more difficult parts of the method implementation. The following sections provide a theoretical discussion of the MLPG solution procedure applied at each node for the construction of the matrix system. These include: Local Weak Form, Test Functions, Trial Function Approximation, Numerical Integration, Domain Discretization, Boundary Conditions, and Parallelization. An numerical discussion of the methodology is presented in terms of the Possion’s Equation in Chapter 2.

1.1.1 Local Symmetric Weak Form (LSWF)

One unique feature of MLPG is the use of the Local Symmetric Weak Form (LSWF) of the governing equations. The LSWF is obtained by applying the method of weighted residuals, including the Galerkin Method, to the standard partial differential equations (PDE) governing the problem. A test function \( w \) is selected to specify the influence, based on spatial proximity, of the node being evaluated and the other nodes in within the subdomain. The system is integrated over each of the subdomains \( \Omega_s \).

Often the divergence theorem is applied to reduce the order of derivatives within the equations and integration by parts is used for product terms in the original PDEs. These techniques can result in additional line integral terms which must be evaluated if the subdomains intersect the global boundary. The derivatives of the test function are also introduced into the system of equations. The equations are complicated further by terms added for the enforcement of boundary conditions, stabilization methodologies, etc.
1.1.2 Test Functions

The selection of the test function, \( w \), is one of the identifying features of various MLPG methods. In one of Atluri’s early publications on this method [4] six unique MLPG variations (referred to as MLPG1-MLPG6) were created with the sole differentiating factor being the selection of the test function. Of the six methods considered, only MLPG1, which uses the MLS weight function, and MLPG5, which uses the Heaviside unit step function, are frequently used in the literature. MLPG5 is favored by many researchers because the derivative of the function is zero, removing terms from the governing equations that include test function derivatives. This makes the implementation more computationally efficient than MLPG1. Others say that by using the MLPG5 implementation “global matrices may be diagonally less dominant that those of MLPG1 and therefore more difficult to resolve” [5]. Applications with axi-symmetric properties have also had poor results when using MLPG5 yet satisfactory results using other formulations [6].

The selection of test function not only influences the nodes included within the subdomain, but also the weight of influence they have on the final solution. Function selection resulting in very small weights can result in a poorly conditioned matrix as well as an unnecessary increase in computational load. Functions that result in large numbers of contributing nodes can also result in matrices that are not sparse and prevent the use of many common solution techniques.

1.1.3 Trial Function Approximation

The most critical step in the solution procedure is the approximation the trial function. The trial function is built from weighted contributions of nodes in the support domain using a meshless approximation method, most commonly Moving Least Squares (MLS). In most MLPG methods the trial function \( (u) \) is replaced by the sums of the products of the MLS shape function \( (\phi) \) vector and the values at the sample points. This provides an approximation of the function at any point of interest, and is not limited to node locations. Likewise derivatives of the trial function are replaced by a summation of the product of the derivatives of the shape function multiplied by the nodal values. The integration points are not always node locations, necessitating these approximations. There are many sources for details on MLS (Levin [7] and Tanojo [8]) as well as its implementation in MLPG (Atluri and Zhu [1] and Pan et al. [9]). Therefore, only details relevant to its performance in MLPG are included here.

Within the MLS framework exists a basis function. The basis controls the type of information that can be approximated with the data. For the approximation of two-dimensional data either a linear basis \((1,x,y)\) or a quadratic basis \((1, x, y, xy, x^2, y^2)\) can be used. The quadratic basis can capture more behavior within the data set, but is also computationally more expensive. It has been shown that the selection of basis is a large factor in solution accuracy [5].
Like the previously discussed test function, a weight function is also part of the MLS procedure. Unlike the test function, the selection of a heaviside function for this weight function is not acceptable as it will create matrices that can not be inverted, causing the method to fail. Common selections are Gaussian functions and splines (usually cubic or quartic). Often if a Gaussian or spline function is selected for the test function, the same function will be used within MLS for the trial function approximation.

There is much criticism over the use of MLS within MLPG. MLS is primarily a methodology for approximating the value of a polynomial function from a discrete data set through the minimization of the L2 error norm. However MLPG uses MLS for approximating values within a discrete data set representing rational data not polynomial functions. Additionally, the shape functions are computed by a series of matrix operations, including inverses, and are computationally intensive procedures. Many of the shape function derivatives are equal to zero at the center of the domain, usually this is the location of the point of interest. These zeros result in unstable solutions as some of the constructed matrices can not be inverted [10].

Alternative methodologies have been created to address the shortcomings of MLS within MLPG. In their review paper of MLPG, Nguyen et al. [10] present an implementation of MLS that eliminates the need to invert the matrix used in the calculation of the shape functions. Nguyen et al. cited multiple Element-Free Galerkin articles which also use this methodology. Also, the utilization of a shifted basis function can prevent the zero point of the shape function derivatives from coinciding with the node of interest, resulting in a more stable solution method [10].

The shape function derivatives appear in two forms in the literature. Full derivatives, and diffuse derivatives, which are an approximation to the derivative and use fewer terms. It has not been shown that the error introduced by the diffuse derivatives negatively impacts the MLPG solution [5]. It has been shown that there is a runtime decrease when the simpler derivative equations are used [3].

Authors have proposed alternatives to MLS including two different approaches known as Generalized Moving Least Squares (GMLS). The first published GMLS by Atluri et. al [11] in 1999 was developed to allow the solution of fourth order boundary value problems. It should be noted that this research group reverts to the standard MLS formulation in subsequent publications for boundary value problems. The premise of the work was to allow the approximation of trial function derivatives by using the product of the nodal derivatives and the shape function \( \phi u_x \) instead of the product of the nodal values and the derivative of the shape function \( \phi_x u \). The researchers chose to minimize the H1 error norm instead of the L2 norm for their approach. Their final trial function approximation included linear combinations of shape function approximations on the function and its derivatives, similar to Finite Element method formulations. The actual MLPG formulation also included fundamental changes to subdomain definitions and
usage. Although the results for the thin beam application were very good it is difficult to assess
the viability of this generalized MLS formulation [11].

The second approach has been named Direct MLPG (DMLPG). This is a promising vari-
ation on the traditional MLS based MLPG. The first published application is Mirzaei and
Schaback [3] with companion publications devoted to the derivation of their GMLS [3] and
notes on its implementation [12]. Their methodology computes trial function approximations
via GMLS methodology [3] which avoids the computationally intensive calculation of shape
functions altogether. Instead this method produces polynomial functionals to describe points
and their derivatives. Boundary condition enforcement can be done directly in some cases rather
than by approximation [3]. GMLS replaces integration over shape functions with integration
over polynomials which requires fewer quadrature points as well [13]. Dehghan and Schaback
considered the standard six MLPG formulations replacing MLS with GMLS and performed
comparisons using Poisson’s equations with Franke’s functions for scattered data. MLPG 3
and 6 were not analyzed because they are not compatible with the GMLS approach. For the
remaining MLPG methods (1, 2, 4, and 5), DMLPG was able to lower maximum errors and
achieve faster run times [3].

Since their initial publication Mirzaei and Schaback [14] have applied DMLPG to benchmark
heat conduction problems. They achieved better accuracy than the MLPG MLS formulation and
at reduced run times. Taleei and Dehghan [15] used DMLPG to solve the Linearized Poisson-
Boltzmann equation and linear elasticity problems with two-phases. Interface problems such
as these present many computational challenges. In this case the researchers used an irregu-
lar shaped interface with both regular and random nodal arrangements. The results achieved
accuracy similar to other computational methods at reduced computational cost [15].

Other enhancements related to MLS within MLPG include the use of two-valued trigonomet-
ic functions as weighting functions to provide better treatment of discontinuities such as cracks
[16]. Dehghan and Salehi [17] used MLPG with a Moving Least Squares Reproducing Kernal
(MLSRK) method for approximating the trial function to the time-dependent Maxwell equa-
tions. This equation set describes electromagnetic wave propagation. Three two-dimensional
applications having analytical solutions were studied. The MLSRK method is similar to MLS,
having comparable computational complexity, but MLSRK may provide smoother approxima-
tions. Reproducing Kernal methods are common in numerical methods and many references
are provided by Dehghan and Salehi [17]. The implementation within MLPG is novel.

1.1.4 Numerical Integration

The numerical integration procedures are one of the most studied parts of the MLPG method-
ology. The MLPG solution procedure requires the numerical integration of the integrands which
contain combinations of trial function approximations and their derivatives, weight functions and their derivatives, and constants. Beginning with Atluri’s initial MLPG publication [1], Gaussian, also called Gauss-Legendre, quadrature has been the most commonly implemented method. This quadrature rule can be applied to a unit circle in polar coordinates, usually requiring a coordinate transformation within the MLPG implementation, or using the Gauss-Legendre Product Rule on a unit square.

The numerical integration plays an important role in the convergence of numerical solutions of meshless methods, especially when domain integrals are involved to generate the stiffness matrix as happens in MLPG [18]. Conventional numerical quadrature schemes are more applicable with regard to speed and accuracy to polynomials than rational functions, as documented by numerous authors. MLPG requires the integration of rational data in the form of MLS shape functions. Many publications investigating the accuracy of various known quadrature and cubature formulations exist. Some of these publications examine the methodologies with general meshfree constructs and a few focus specifically on implementation of the formulas within MLPG methods.

Mazzia published two articles with different collaborators (Mazzia and Pini [18] and Mazzia et al. [19]) which focused on the implementation of cubature formulas on unit disks and circular sectors within MLPG1. In standalone experiments numerous cubature methodologies are proven to be more accurate and more efficient than the commonly used Gauss quadrature in polar coordinates and the Gauss-Legendre Product Rule in Cartesian coordinate. Within MLPG however, the results were contradictory. In the second paper [18], the actual values in the stiffness matrix were compared so that convergence of each method and accuracy could be discussed independently of the complete matrix solution procedure. They concluded that “the generally complicated non-polynomial behavior of the integrand functions” creates error that dominates the system, obscuring the effects of the various of quadrature methods [18].

Pecher [5] conducted similar studies within MLPG applying Lobatto quadrature. Lobatto requires less quadratures points than Gaussian quadrature in spherical cases; this is relevant because three-dimensional applications are often viewed as the definition of success for numerical methods. Like Mazzia’s works, Pecher achieved superior results with his new quadrature implementation in a standalone study but inconclusive results within an MLPG implementation. Pecher concluded that “MLS-approximation error is the controlling factor of the integration accuracy” [5].

1.1.5 Domain Discretization

Domain discretization is a sensitive aspect of all numerical methods. In conventional methods many rules of thumb and automated tools have been created to help reduce the complexity and
tedium of the mesh or grid creation step. Most problems utilize a non-uniform arrangement of nodes, with increased nodal density near regions of interest such as boundary layers in fluid dynamics, or cracks in structural mechanics. There are known pitfalls when the nodal density in one direction is significantly higher than in another direction. In Finite Element methods this is known as the aspect ratio of an element. Poor aspect ratios generate errors and instability in the solution. Element aspect ratios are frequently monitored and used to signal the need to remesh the domain.

Although many researchers do initial studies of methodology changes on regular and irregular nodal arrangements for simple governing equations, the level of sensitivity to the nodal arrangement is not consistent between authors and MLPG methods. Some show independence and others show large errors on irregular grids that are not always removed by increasing the number of nodes. To date MLPG results for fluids applications are always computed on regular grids with conventional cluster and spacing.

Kim and Atluri [20] have published an attempt to dynamically control error by the addition of nodes during a simulation. Their approach however also introduced a method to construct weight functions on polygonal subdomains using linear finite element shape functions. This method appears very computationally intensive and the results were very similar in accuracy to the traditional formulation for the benchmark computational mechanics applications presented [20]. Shibahara and Atluri [21] used a novel multigrid like approach for the MLPG solution of moving heat source. A high density region of nodes was created that moved through the larger domain with a coarse node density. The high density region was used to capture finer details near the moving source without creating unnecessarily high computational burden in the large domain. This methods worked well in this application [21].

### 1.1.6 Boundary Conditions

Treatment of boundary conditions is one of the most challenging aspects of all numerical solution techniques. MLPG is no different. Beginning with its earliest publications numerous methodologies for the handling of boundary conditions were suggested. In 2006 Gu and Liu [22] assessed the performance of MLPG for the two main types of boundary conditions as follows: MLPG methods are “computationally efficient, and the solution is accurate when there are only Dirichlet boundary conditions. However the are major shortcoming of these methods is that the derivative (Neumann) boundary conditions may lead to large computational error”.

There are two common methods for essential boundary condition enforcement in MLPG: Direct Interpolation, also call collocation, and the penalty method. Direct Interpolation is used more commonly because a tuning parameter is not added to the solution procedure. This method replaces the governing equation for the boundary node with an MLS application of the
boundary condition value applied at the boundary and in a weighted sense to the surrounding nodes. It is worth noting that the direct approach “destroys the symmetry of the stiffness matrix” [23]. MLPG often does not produce a symmetric matrix but direct interpolation can effect the conditioning and results of preconditioning methods applied to the stiffness matrix. The DMLPG method for applying Dirichlet boundary conditions is similar to collocation above but does not requires the MLS approximation.

The penalty method was proposed in the initial MLPG formulation [1] and the difficulty in determining the correct value for the penalty parameter was discussed. The approach can be summarized by applying a penalty parameter to all line integrals in the governing equations. Since these terms are only evaluated when the subdomain intersects the boundary, the penalty term is used to balance the weight of domain integrals and boundary integrals while preserving a uniform equation set for all nodes.

Neumann boundary conditions are conventionally enforced by adding additional terms to the solution set for each boundary node. These additional terms are integrals one dimension lower than the subdomain. This boundary condition treatment is computationally intensive and difficult to apply to new domain shapes. In this work a novel Neumann boundary condition, the Shape Function Interpolation Method (SFIM), replaces the traditional line integral approach with one that more closely resembles Direct Interpolation. The new method is shown to be more computational efficient and easily applied to boundaries of various shapes.

1.1.7 Parallelization

As computational methods mature the changes to the actual method formulation usually slow or stop. Research in the areas of optimization and extension then increase. Extension is accomplished via coupling with other models, such as material damage models, turbulence models, etc. Trobec et al. [24] summarizes it as follows “like all computationally intensive numerical methods meshless methods should be efficiently parallelized in order to be competitive.” They also remind readers that successful parallelization of any numerical method requires efficient load balancing and minimal interprocessor communication. Standard techniques for matrix solutions and storage of sparse matrix data have been utilized by many researchers but are not novel or produced standalone publication. However two articles detailing parallelization of MLPG have been published Trobec et al. [24] and Bergamashci et al. [25] both in 2009.

Since many known methods exist for optimizing and parallelizing the matrix solution process, Trobec et al. [24] focused on the parallelization of the construction of the linear system, via domain decomposition. Unlike other MLPG articles this paper detailed the storage method used for the nodes. In this case a k-D tree. This provided an initial runtime improvement during subdomain construction where a search for nearest neighbors must be performed. During
parallelization the tree organization was used to defined the decomposition of the domain. This is more extensible than using human intuition for the creation of decomposition rules. The researchers performed speed-up tests on a two-dimensional domain with an off-center holes and solved diffusion equations with conditions that create steep gradients and rapid changes with time. The parallelization was successful and achieved speed-ups that scaled as expected with the number of nodes. The maximum reported speed-up was approximately 25 times faster than single processor runtime. The group suggests evaluation of the parallelization approach other types of equation sets and for three-dimensional domains, which result in more complex tree storage and may result in increased inter-processor communication [24].

1.2 MLPG Applications

The MLPG methods have been used to solve problems in many fields including computational mechanics, heat transfer, fluid flow, and others. Many examples can be found in the review papers by Sladeck et al. [26] and Nguyen [10]. Sladeck et al.’s review is focused only on MLPG and has a section devoted to fluid flows. Nguyen covers many classes of meshless methods and is focused only on solid mechanics. Nguyen does include a focus on the implementation of the methods with pseudo-code for many common aspects of the methods including stiffness matrix construction and boundary condition enforcement.

The bulk of the MLPG literature focuses on solid mechanics applications rather than fluid flows. Solid mechanics applications are not only more numerous, but also demonstrate more advancements in size and complexity of domains, boundary conditions, and phenomena captured. This is expected to some degree as MLPG is closely related to the Finite Element method which is more commonly used for solid mechanics than for fluids flows. Also advanced capabilities such as damage models are more easily transitioned from Finite Element codes to MLPG than some fluids tools which are native to Finite Difference implementations. Highlighted below are some of the newer and more complex applications in the entire body of literature followed by a more in-depth review for publications related to the application of MLPG to fluids applications.

Three-dimensional potential problems solutions have been presented for both MLPG [27] and DirectMLPG [28]. This allows for a direct comparison of the two MLPG formulations as well as demonstrating their performance in three-dimensional problems. One of the most complex and industry relevant simulations is presented in a pair of articles by Liu, Han Rajendran, and Atluri [29] [30]. This work include multiple three-dimensional impact problems using a coupled ceramic micro-crack and void collapse damage model. The final application of this work is ballistic penetration of ceramic plates such as those used in armor elements. Results are compared to both experiments and Finite Element code results.

Heat transfer is another research area with a large number of MLPG publications. Appli-
cations include benchmark problems such as one-dimensional and two-dimensional conduction and two-dimensional convection and diffusion over a block [31], laminar convection in a square cavity, and forced convection by fluid flow over non-isothermal tube bundles [32]. Additionally more complex phenomena such as three-dimensional heat conduction through a plate with continuously varying properties was successfully simulated with the common MLPG formulation [33]. One industrially relevant application related to welding was published by Shibahara and Atluri [21]. They simulated a moving heat source in a bead-on-plate welding scenarios. The problem was modeled in two dimensions and used the MLPG5 formulation with a higher nodal density near the welding path. The method showed good agreement with Finite Element calculations [21].

1.2.1 Fluid Flows

The numerical solution of relevant problems of interest in fluid dynamics usually requires the solution of the three-dimensional Navier-Stokes equations at a minimum. Often turbulent Reynolds numbers, complex domains, or multi-material aspects are introduced when solving industry relevant applications. To achieve such simulations the Navier-Stokes equations are solved in primitive variable form with schemes for pressure splitting or pressure correction for the coupled equations. Solving in vorticity-stream function form can be an appealing option for one-dimensional and two-dimensional problems, but is not extensible to three dimensions. Although a few advanced fluid flow applications have been published using MLPG, they are not extendable to three dimensions. A free surface wave problem was solved in two dimensions by Ma et al. [34] using the vorticity-stream function form and a similar problem was solved in three dimensions using a modified MLPG formulation that utilizes the Rankine source solution [35].

The following sections provide more in depth discussion and comparison of the MLPG publications using the primitive form of the Navier-Stokes equations. These publications focus on the application of the method to a fairly small set of benchmark problems including the lid-driven cavity, Stokes Flow, and the backward facing step. The results and methodology differences of the publications will be discussed.

Lid-Driven Cavity

The lid-driven cavity results of Ghia et al. [36] have been used as a comparison point for many MLPG incompressible flow formulations and implementations. The lid-driven cavity problem is a two-dimensional, laminar, incompressible flow most often in a square domain. The top wall is assigned a velocity while the other walls are stationary and have no-slip boundary conditions. This results in a vortical flow with singularities in two corners. Ghia et al. present results
obtained from a coupled strongly implicit multi-grid solver.

The three known MLPG publications analyzing this benchmark problem all solved the incompressible Navier-Stokes equations in primitive variable form. They all used MLS to generate the trial solution. The first of these articles is Lin and Atluri [37] who first published lid-driven cavity and Stokes Flow using MLPG. They were able to achieve a satisfactory comparison with Ghia’s results for a flow with a Reynolds number of 100. The Reynolds number of 400 simulations were stable but did not show good agreement. Their implementation included both a penalty parameter for the enforcement of boundary conditions and a stability parameter within their up-winding scheme. They cited the need for improved boundary condition enforcement and a more accurate iterative scheme, Newtonian iteration was used in this work [37].

Wu and his collaborators [38] demonstrated the use of the Streamline Upwind Petrov-Galerkin (SUPG) method, which is widely used in Finite Element methods, within MLPG. They applied this method to the lid-driven cavity, the backward facing step, and natural convection in a square. The MLPG implementation neglected the second order terms to reduce computational cost, a common approach in other methodologies. Wu et al. showed better agreement with Ghia et al. for the Reynolds number of 100 case but poorer agreement for a Reynolds number of 400. However, Wu et al. did achieve stable solutions for the Reynolds number cases of 1000 and 5000. However the results deviate significantly from the accepted solution. The error is attributed to the low order upwinding scheme used. Wu et al. extended the range of Reynolds numbers for which a stable solution could be obtained with MLPG and showed the merit of taking well known tools from conventional methods and applying them within the framework of MLPG [38].

The most recent work is Najafi et al. [39]. This group introduced another well-known algorithm into their MLPG framework. The Characteristic-Based Split (CBS) algorithm is used to discretize the primitive form of the governing equations and provide stability at higher Reynolds numbers. The method has been used in conventional methods with much success. Najafi et al. simulated lid-driven cavity, backward facing step, and flow past a circular cylinder. This work demonstrated good agreement at Reynolds numbers of 100, 400, 1000, 3200, 5000, 7500, and 10000 with the Ghia et al.’s lid-driven cavity results. Although these are all laminar simulations they show that with the correct methodology MLPG methods can achieve incompressible flow results at high Reynolds numbers. Additional differences in Najafi et al.’s implementation are use of the unity test function, a cubic spline function within MLS, and a lumped matrix solution method [39]. The other publications in this section used the quartic spline for both the test function and within MLS.
Stokes Flow

Another common benchmark problem is Stokes Flow. This problem is commonly studied because it is a simplification of the Navier-Stokes equations and eliminates the need for upwinding or pressure splitting. In these flows the viscous forces dominate the solution and the nonlinear convection terms are not present. There are also sets of boundary conditions with known analytical solutions. Both Lin and Atluri [37] and Sataphrahm and Luadsong [40] published MLPG solutions for this problem. Lin and Atluri used the same formulation discussed for the lid-driven cavity and showed qualitative results. They were able to identify sets of parameters that resulted in stable and characteristically correct solutions. Sataphrahm and Luadsong used a primitive variable formulation of the governing equations but applied a unique solution method. In their work MLPG was applied only to Poisson’s equation to solve for pressure, and the velocity components were computed from the differencing scheme. Both regular and irregular node distributions were examined. Both contour plots as well as tables of computed error values were presented.

Backward Facing Step

The backward facing step introduces many challenges when compared to the previously discussed fluids problems. Not only is it a deviation from the simple square domain, but the solution boundary conditions are also more complex. The problem results in an eddy or vortex at the step face and flow reattachment to the bottom surface downstream of the step. Wu et al. [38] presented results with a uniform inlet velocity and Reynolds numbers of 50 and 100. They achieved good agreement with the velocity field and reattachment length when compared to a Finite Volume code. They did observe pressure oscillations at the exit.

Instead of allowing the velocity to develop in an entrance region Najafi et al. [39] included a parabolic horizontal velocity component at the top of the inlet edge and allowed the bottom of the inlet domain edge to be the step face. This resulted in a more simple domain geometrically than Wu et al. Najafi used a Reynolds number of 800 and compared to previously published Finite Element and Finite Difference results. Both qualitative results as well as upper wall and lower wall recirculation lengths were presented. Najafi et al.’s numbers were bracketed by the other simulations results for both recirculation lengths. No pressure oscillations are discussed for this solution method.

1.2.2 Other Applications

Additional applications were published by the authors discussed above but were not common between the publications and therefore are listed here for completeness. Wu et al. [38] studied natural convection in a square cavity. Comparisons of minimum, average, and maximum
Nusslet numbers as well as maximum velocity components are presented against previously published work with good agreement. Najafi et al. [39] presented flow past a circular cylinder in a rectangular domain. Three Reynolds numbers were considered: 100, 150, and 200. The average drag coefficient and the maximum lift coefficient were compared with multiple other publications. The MLPG results were within the bounds of the published values. Wu et al. [41] presented results for concentric annulus between two circular cylinders and concentric annulus between an inner circular cylinder and an outer square cylinder. The maximum stream function and the average Nusslet number were compared to previous numerical results and are in good agreement for both cases. This research used the vorticity-stream function approach and has not been published using MLPG method with a primitive variable formulation.

1.3 Summary

MLPG methods exist in many unique formulations. The basic method has matured from a novel concept to a framework that is being adopted by a growing number of researchers. The framework is now being used to test meshless approximation methods, numerical integration rules, and other known numerical techniques within the MLPG solution procedure. One of the most notable variations is DMLPG, which reduces the complexity of the trial space approximation terms and may present the opportunity make a large improvement in the integration scheme. Optimizations such as multigrid and parallelization attempts are being applied to MLPG as well.

While the MLPG methods are evolving into a more robust and capable solution methodology, proven abilities in the fluid dynamics realm are still lacking. Even a cursory investigation into MLPG publications reveals the domination of computational mechanics applications followed by heat transfer problems. Within the fluids applications a void exists for publications using the extensible primitive variable form of the incompressible Navier-Stokes equations to anything more complex than two-dimensional benchmark problems (most commonly lid-driven cavity and backward facing step). More complex domains and phenomena must be successfully simulated by MLPG methods to prove this method a viable alternative to conventional methods for fluid flows.

The MLPG methods were created to ease the burden of meshing and remeshing through the elimination of static relationships between nodes. This same property presents the opportunity to investigate the application of MLPG to simulations with both fixed and moving nodes, additional adaptive meshing options such as interface tracking, and multi-material problems. These possibilities have not yet been fully explored.
1.4 Objectives

This research was undertaken to further explore various MLPG solution procedures and their applicability to fluid dynamics problems. An MLPG code was written and validated through benchmark fluids applications. Two notable contributions of this work are a new process of modeling Neumann boundary conditions and the application of the methods to the Rayleigh-Taylor Instability (RTI). Additionally the work was programmed using C++ with an intentional design process. This was done to create an extendable, efficient code that could be leveraged by future researchers.

This work will extend the body of research to include the Rayleigh-Taylor Instability, a density driven fluid interface problem. It was modeled using the two-dimensional primitive variable form of the Navier-Stokes equations.

1.5 Outline

This work presents both discussion of the MLPG solution procedure and the available numerical modeling techniques as well as MLPG results for various fluid dynamics applications. The research is organized as follows:

- MLPG Methodology
- MLPG Boundary Condition Treatment
- Validation Problems
- Introduction of the Shape Function Interpolation Method (SFIM) Boundary Condition
- Lid-Driven Cavity in a Square Domain (Implementation and Results)
- Rayleigh-Taylor Instability (Implementation and Results)
- Software Design
- Recommendations for Future Work
Chapter 2

MLPG Methodology

The MLPG method consists of the following components: domain discretization and subdomain creation, treatment of the governing equations, trial function approximation, numerical integration, and solving the linear system. There are many options for performing each of these tasks. The discussion below focuses on the approaches implemented in this research. Understanding these techniques and their limitations is critical to successful implementation and application of the MLPG method as well as for interpreting results.

2.1 Domain Discretization and Subdomain Creation

One of the most touted benefits of the MLPG method is the ease of domain discretization. Domains can be of any shape and size. The nodal arrangement is not constrained. Nodes can be uniformly spaced, randomly spaced, clustered near features of interest, or in any other arrangement. Additionally the nodes can be moved, added, or removed during multi-step or iterative computations.

Figure 2.1 shows an example of an MLPG domain with non-uniformly placed nodes. During the computation, the governing equations are evaluated at each node based on the node's subdomain (\( \Omega \)). The subdomain for both an interior node (\( x \)) and a boundary node (\( y \)) are shown in the figure. The subdomain can be a line, circle, rectangle, cube, or other shape having the same number of dimensions as the problem being solved.

The subdomains are defined by applying the test function (\( w \)) and a radius of influence parameter (\( \alpha \)). The radius of influence parameter is usually a multiplier for the distance to the nearest neighbor. This radius is used to provide a relative distance between the nodes within the subdomain and the node of interest. This is one way that the arbitrary node placement is supported. The test function can be one of many weight functions discussed in the following section. The boundaries of the global domain and subdomains are denoted \( \Gamma \). The boundaries
are not treated as entities. Nodes on the boundary and nodes with subdomains that intersect the global domain boundaries may have modified equations in the linear system. The boundaries are one dimension lower than the overall problem dimension.

### 2.1.1 Weight Functions

Many mathematical functions can be used as weight functions to provide spatial data in the absence of a grid or elements. For some uses the functions are required to be continuous, or piece-wise continuous, and smooth through a specified number of derivatives. Heaviside functions and splines are most common. The heaviside and quartic splines were chosen for this implementation. Heaviside functions were chosen for speed when minimal information was needed, and the quartic spline was chosen when spatial relationships are required. The quartic spline is piece-wise continuous and smooth through the second derivative.

The one-dimensional quartic spline is shown in Eq. 2.1. In this equation $r$ is the absolute value of distance between the point being evaluated and the point of interest, divided by the specified radius of influence ($r_\alpha$). In this work the one dimensional equation was applied in each required dimension and then the product is used to represent the overall weight. Examples of the spline in one dimension and a two-dimensional rectangle are shown Figure 2.2. In both examples the node of interest is the center point of the domain. Notice that the weight function is always positive and the magnitude approaches one as the point approaches the node of interest.

$$w(r) = \begin{cases} 1 - 6r^2 + 8r^3 - 3r^4, & \text{for } r \leq 1 \\ 0, & \text{otherwise} \end{cases}$$ (2.1)
The one-dimensional example was created on a line from 0 to 1 containing 101 equally spaced points. A node of interest of 0.5 and a radius of influence of 0.4 were selected. As expected the value of the weight function is zero outside the radius. The two-dimensional example uses a box of length 2 on each side. A grid of 201 uniformly spaced nodes in each direction was evaluated using a node of interest at the center of the box. In this case the $x$ and $y$ components of the radius of interest were not equal. The $x$ component was assigned a value of 0.4 while the $y$ component was 0.8. This allows the creation of a rectangular region of influence.

![Figure 2.2: Examples of the quartic spline in one dimension (a) and in two dimensions with a rectangular region of influence (b).](image)

The first derivatives of the weight functions were needed as well. These were made more complex by the absolute value in the definition of $r_\alpha$. The first derivative for the quartic spline in one dimension are shown in Eq. 2.2. Here $d$ is the distance from evaluation point to the source point. The process for computing the derivative in multiple dimensions follows in Eq. 2.3.

\[
dw(r) = \begin{cases} 
\frac{12d}{r_\alpha} (-1 + 2r - r^2), & \text{for } r \leq 1 \\
0, & \text{otherwise}
\end{cases} \tag{2.2}
\]

\[
\frac{dw(x, y, z)}{dx} = \frac{dw(x)}{dx} w(y)w(z) + w(x) \frac{dw(y)}{dx} w(z) + w(x)w(y) \frac{dw(z)}{dx} \tag{2.3}
\]

Using the one-dimensional and two-dimensional configurations, the first derivative in the $x$-
direction is shown in Figure 2.3. The derivative yields both positive and negative values. There is no set peak magnitude. The peak magnitude is related to the radius of influence. The region outside of radius of influence has a magnitude of zero.

![Graphs showing derivative](image)

Figure 2.3: Examples of the first derivative of the quartic spline in the $x$-direction in one dimension (a) and in a two dimensions with a rectangular region of influence (b).

### 2.2 Treatment of the Governing Equations

The MLPG methodology uses the governing equations in the Local Symmetric Weak Form (LSWF). This is an integral form of the equations consistent with the format used in Finite Element methods. The two-dimensional Poisson’s equation, Eq. 2.4, is used to demonstrate the process for obtaining the LSWF.

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = F \quad (2.4)$$

To obtain the LWSF each term in the governing equation is first multiplied by the test function ($w$) and then integrated over the subdomain ($\Omega$). The simplest version of the equation is shown in Eq. 2.5. In this case boundary conditions or simplifications have not been applied.

$$\int_{\Omega} \left( \frac{\partial^2 u}{\partial x^2} w + \frac{\partial^2 u}{\partial y^2} w \right) d\Omega = \int_{\Omega} F w d\Omega \quad (2.5)$$

The governing equations are solved as a linear system in the form $K\vec{x} = \vec{f}$ where $K$ is the coefficient matrix, $\vec{x}$ is the vector of unknowns, and $\vec{f}$ is the forcing vector. The terms are
grouped into this linear system, Eq. 2.6.

\[
\left[ \int_{\Omega} \left( \frac{\partial^2}{\partial x^2} w + \frac{\partial^2}{\partial y^2} w \right) d\Omega \right] [u] = \left[ \int_{\Omega} F w d\Omega \right] \tag{2.6}
\]

2.2.1 Order of Derivatives

In many cases the order of the derivatives is lowered through the application of the divergence theorem to products of shape functions and test functions. An example of this procedure using the first term of the Poisson’s equation is shown below:

\[
\int_{\Omega} \frac{\partial^2 u}{\partial x^2} w d\Omega = \int_{\Gamma} \frac{\partial^2 u}{\partial x^2} w d\Gamma - \int_{\Omega} \frac{\partial u}{\partial x} \frac{\partial w}{\partial x} d\Omega \tag{2.7}
\]

The application of the divergence theorem to a domain integral yields two integral terms. The first term is one dimensional order lower, a line integral in this case. The line integral can be dropped if the test function goes to 0 along the boundary of the subdomain, which is the case with splines. The second term remains a domain integral over the subdomain, but the shape function is now a first derivative and the derivative of the test function is present. The exception is when the test function is heaviside. In this case the derivative for the test function is 0 and so only the line integral remains.

This type of manipulation of the governing equation is very common in MLPG formulations for various governing equation sets. In some cases this is done for computational efficiency to prevent the necessity of computing the derivatives of the shape functions. In other cases the motivation is to reduce non-linearities that can arise when products of shape function derivatives are present. The final form without boundary conditions is shown in Eq. 2.8. This formulation is independent of test function selection.

\[
\int_{\Gamma} \frac{\partial^2 u}{\partial x^2} w d\Gamma - \int_{\Omega} \frac{\partial u}{\partial x} \frac{\partial w}{\partial x} d\Omega + \int_{\Gamma} \frac{\partial^2 u}{\partial y^2} w d\Gamma - \int_{\Omega} \frac{\partial u}{\partial y} \frac{\partial w}{\partial y} d\Omega = \int_{\Omega} F w d\Omega \tag{2.8}
\]

2.3 Trial Function Approximation

The trial function is used to approximate the value of the functions and their derivatives. This is applied to the unknowns on the left hand side of the equation set and known functions on the right hand side of the equation set. There are many options for performing this approximation including MLS, Radial Basis Functions (RBF), and others. In this work, as in the initial MLPG formulation, MLS was used.
### 2.3.1 Moving Least Squares (MLS)

The MLS method approximates the value of a function and its derivatives using a set of discrete data. The discrete data does not have to be ordered, uniformly spaced, or correspond to the locations being approximated. The MLPG method utilizes shape functions, an intermediate value produced by the MLS method. The MLS methodology, notes on implementation, and an example are presented here.

MLS is a common mathematical tool and the derivation is presented in standalone MLS papers such as Tanojo [8] and Levin [7]. The derivation is replicated in the introduction of most MLPG articles as well. The methodology is presented here from an implementation perspective not a theoretical mathematical perspective.

Following the notation of Pan et al. [9] the final MLS approximation is shown in Eq. 2.9. Notice that a single value (\(\hat{u}\)) is being approximated based on a vector (indicated by the bold font) of known data (\(u\)) and the vector of shape functions (\(\phi^T\)). When a series of data points is being approximated, the equation Eq. 2.10 is used. The summation notation is shown as well. In this case \(n\) is the size of the two vectors. Notice that the same vector of data can be used to generate different shape function vectors.

\[
\hat{u} = \phi^T u = \sum_{i=1}^{n} \phi^T(i)u(i) \tag{2.9}
\]

\[
\hat{u}(x) = \phi^T(x)u = \sum_{i=1}^{n} \phi^T(i)(x)u(i) \tag{2.10}
\]

The shape function computations have three components that the user controls. The first component is the vector of known discrete data, this is the input. The size of the known data vector is \(n\). In this MLPG implementation the vector contains the nodes in the subdomain of the point being approximated. The second component is a trial function. This is a weight function that is smooth and continuous to the necessary number of derivatives. The trial function and the test function can be the same, but it is not required. Usually a spline such as those discussed in Section 2.1.1 is used. The third component is the basis function. The basis function is constructed of elements that will be used to describe the relationships between the known data points. It is a function of the number of dimensions being used in the computation and the desired order of the approximation (linear, quadratic, etc). A list of standard basis functions is shown in Table 2.1.

The order of the basis function, and therefore the approximation, is usually denoted \(k\). For a quadratic basis function \(k\) is 2. The length of the standard basis function vector \((m)\) for a
Table 2.1: Basis functions for MLS Approximation

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Order</th>
<th>Basis Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Linear</td>
<td>1, x</td>
</tr>
<tr>
<td>1</td>
<td>Quadratic</td>
<td>1, x, x^2</td>
</tr>
<tr>
<td>1</td>
<td>Cubic</td>
<td>1, x, x^2, x^3</td>
</tr>
<tr>
<td>2</td>
<td>Quadratic</td>
<td>1, x, y</td>
</tr>
<tr>
<td>2</td>
<td>Quadratic</td>
<td>1, x, x^2, xy, y^2</td>
</tr>
<tr>
<td>2</td>
<td>Cubic</td>
<td>1, x, x, x^2, xy, y^2, x^3, x^2 y, y^2 x, y^3</td>
</tr>
</tbody>
</table>

given order and dimension can be computed as follows:

\[ m = \prod_{i=1}^{Dimensions} \frac{k + i}{i} \]  

(2.11)

For a successful computation, the number of points being used in the MLS computation must be greater than than the number of terms in the selected basis function \((n > m)\). To reduce error and bias, the implementation of MLS should utilize a local coordinate system. The unknown point or node of interest should be positioned at the origin and the support domain should be shifted to the new coordinate system as well. The test function does not need modification as it uses the distance between points.

The known data vector and the trial function are used to create the \( W \) matrix. \( W \) contains the weight as computed by applying the selected trial function to the known node locations using the unknown location as the reference point. The matrix is a \( n \) by \( n \) matrix with values only along the diagonal.

The \( P \) matrix is size \( n \) by \( m \) and is the evaluation of the basis function at each point in the known data vector. Both the matrix and its transpose are used in computations of the shape function. The \( B \) matrix is the product of the transpose of the \( P \) and the weight matrix. It is of size \( m \) by \( n \).

\[ B = P^T W \]  

(2.12)

The \( A \) matrix is the product of the \( B \) and the \( P \) matrix and is size \( m \) by \( m \).

\[ A = BP \]  

(2.13)
The inverse of $A$ and the $B$ matrices along with the point vector ($p_f$) are used to create the shape functions as shown in Eq. 2.14. The point vector ($p_f$) is the evaluation of the basis function for the unknown point, also referred to as the point of interest. These are the intermediates used within MLPG.

$$\phi^T = p_f^T A^{-1}B$$

The full first and second derivatives [42] are shown in Eq. 2.15 and Eq. 2.16 respectively. In these equations the first derivative is in the $q$ dimension. The $r$ is the dimension for the second derivative. The equations require the derivatives of $A$, $B$, $P$, and $P_f$ as well as the inverses of some of these components. The full derivative is very computationally intensive.

$$\phi_{rq}^T = p_{fq}^T A^{-1}B + p_{f}; A^{-1}B + p_{f}^T A^{-1}B$$

$$\phi_{qr}^T = p_{qfr}^T A^{-1}B + p_{f}^T A^{-1}B + p_{f}^T A^{-1}B + p_{qfr}^T A^{-1}B$$

The diffuse derivative approach was used in this research. The diffuse derivative is presented in Eq. 2.17. The approximation uses only the terms that have the derivative of the point vector $p_f$. This is much quicker computationally and has a sufficient level of accuracy.

$$\phi_{rq}^T = p_{fq}^T A^{-1}B$$

$$\phi_{qr}^T = p_{qfr}^T A^{-1}B$$

The shape functions generated by the MLS procedure have two properties that can be used to check the accuracy of the approximation. The Partition of Unity applies to the shape functions and states that the sum of the shape functions must equal 1 to machine precision in an accurate approximation. Similarly the Partition of Nullity states that the sum of the shape function derivatives must equal zero in an accurate approximation [42].

It is also useful to note that derivatives can be computed two ways, as shown in Eq. 2.18. This is useful when the derivatives of known data, like that on the right hand side of many equations, are required.

$$\hat{u}_{r,x} = \phi_{r,x}^T u$$

Returning to the LSWF Poisson’s equation, Eq. 2.6, the shape functions are substituted to
produce:
\[
\int_{\Omega} \left( \phi_{xx} w + \phi_{yy} w \right) d\Omega \quad [u] = \int_{\Omega} Fwd\Omega
\] (2.19)

Additionally the divergence theorem an be applied to yield:
\[
\left( \int_{\Gamma} \phi_{xx} wd\Gamma + \int_{\Gamma} \phi_{yy} wd\Gamma - \int_{\Omega} \phi_{x} w_{x} d\Omega - \int_{\Omega} \phi_{y} w_{y} d\Omega + \right) [u] = \int_{\Omega} Fwd\Omega
\] (2.20)

The forcing vector contributions may take many forms. Examples are constant expressions, functions of position, or functions of the other node properties, such as previous timestep information. The evaluation of the force vector will not contain shape functions when only constants and spatial functions are selected. However in more complex cases, such as the transient cases, the shape functions will be used to approximate the values or derivatives of the values at the required locations.

**Example**

The MLS implementation is demonstrated using a set of 1000 Halton points generated by Matlab. These points are not uniformly spaced and are representative of a node layout that can be used to discretize an MLPG domain. These points are assigned the value of Franke’s function, this demonstration was also performed by Mirzaei, et al. [3]. The Franke’s function at the 1000 Halton points, shown in Figure 2.4, is used as the known data set.

![Figure 2.4](image)

**Figure 2.4**: Input data for MLS example. Positions of the 1000 Halton points (a) and the Franke’s function surfaces at the Halton Points (b).
Franke’s function and its first derivatives are shown in Eq. 2.21, Eq. 2.22, and Eq. 2.23. Note that Franke’s function presented in Mirzaei et al. is not correct and so the equation from Franke’s original report [43] is used here. The derivatives were obtained analytically.

\[
f(x, y) = 0.75 \exp\left(-\frac{(9x - 2)^2 - (9y - 2)^2}{4}\right) + 0.75 \exp\left(-\frac{(9x + 1)^2}{49} - \frac{(9y + 1)}{10}\right) + 0.5 \exp\left(-\frac{(9x - 7)^2 - (9y - 3)^2}{4}\right) - 0.2 \exp\left(-\frac{(9x - 4)^2 - (9y - 7)^2}{4}\right)
\]  

(2.21)

\[
f_{dx}(x, y) = -3.375 \exp\left(-\frac{(9x - 2)^2 - (9y - 2)^2}{4}\right) (9x - 2) - \frac{27}{98} \exp\left(-\frac{(9x + 1)^2}{49} - \frac{(9y + 1)}{10}\right) (9x + 1) - 2.25 \exp\left(-\frac{(9x - 7)^2 - (9y - 3)^2}{4}\right) (9x - 7) + 3.6 \exp\left(-\frac{(9x - 4)^2 - (9y - 7)^2}{4}\right) (9x - 4)
\]  

(2.22)

\[
f_{dy}(x, y) = -3.375 \exp\left(-\frac{(9x - 2)^2 - (9y - 2)^2}{4}\right) (9y - 2) - 0.675 \exp\left(-\frac{(9x + 1)^2}{49} - \frac{(9y + 1)}{10}\right) (9y - 3) - 2.25 \exp\left(-\frac{(9x - 7)^2 - (9y - 3)^2}{4}\right) (9y - 3) + 3.6 \exp\left(-\frac{(9x - 4)^2 - (9y - 7)^2}{4}\right) (9y - 7)
\]  

(2.23)

The MLS process was used to approximate the Franke’s function values on a uniform unit grid with 11 nodes in each direction. The results of both methods for computing the derivative, Eq. 2.18, are shown. Method 1 includes the derivative of the shape function, and Method 2 uses the known derivative of the input data. Figure 2.5 shows the MLS results against the true function computed on the same grid. The $R$ indicates the size of radius used to identify the subdomain. Points in the subdomain make up the known data vector. For $R=0.1$ the average subdomain size was 32 nodes. The corner nodes have smaller subdomains, consisting of only 8 nodes. Edges nodes had subdomains containing an average of 20 nodes. The interior nodes have the largest subdomains, approaching a size of 40.

The increase of $R$ results in less accuracy as the area of usable data is too large and features harder for the method to capture. As $R$ decreases the results degrade as the quantity of known data points is too small. For $R=0.05$ the average subdomain size was only 8 nodes. Corner
nodes had subdomains of only 2 nodes. These are too small to actually make an approximation. The center nodes have subdomains containing approximately 12 nodes.

Figure 2.5: MLS Approximation of Franke’s function on a uniform grid.

Figure 2.6 shows the MLS approximation obtained by each of the derivative methods for Franke’s function along the line \( x = 0.3 \). Methods 1 and 2 again yield comparable results. The approximation behaves as expected with changes to \( R \).

2.4 Numerical Integration

As with trial function approximation there are many options for numerical integration. In recent years papers comparing integration methods within the MLPG framework have been published. These include studies on cubature formulas on unit disk and circular sectors [18] [19] and Lobatto quadrature [5]. These papers conclude that more accurate integration in isolation is possible. However within the MLPG framework, the integration method is not the primary source of error or accuracy. In this work Gaussian quadrature was chosen because it is well understood and efficient.

2.4.1 Gaussian Quadrature

Gaussian quadrature replaces the integral with a sum over a specifically distributed set of points within the integration domain. The points, referred to as Gauss points, are paired with Gauss
weights. A leading coefficient, or Jacobian, is used to transform the actual integration domain into a normalized integration domain. The basic formulation is shown in Eq. 2.24.

$$\int_{\Omega} u \, d\Omega = J_q \sum_{i=1}^{n_q} w_q(i) \, u(\vec{x}_q(i))$$

(2.24)

The Gaussian locations and weights can be found in numerous math texts and reference websites. Point-weight pairs are provided for one dimension but can be applied in two or more dimensions for integration over those domains. Point-weight pairs are provided for as few as 2 points and into the hundreds. In this work 4 point, 6 point, and 16 point quadrature were used. Figure 2.7 shows a one-dimensional configuration of 16 quadrature points and a two-dimensional layout with 6 quadrature points in each direction.

The Gaussian quadrature is substituted for the integrals in Eq. 2.19. The result is the MLPG formulation of the two-dimensional Poisson’s equation, Eq. 2.25. This notation for the integral domain is incorporated into the limits of the summations.

$$\begin{bmatrix} \Omega_{n_q} \\ J_q \sum_{i=1}^{n_q} w_q(i) \, (\phi_{,xx} w + \phi_{,yy} w) \end{bmatrix} [u] = \begin{bmatrix} \Omega_{n_q} \\ J_q \sum_{i=1}^{n_q} w_q(i) \, Fw \end{bmatrix}$$

(2.25)
Figure 2.7: Quadrature point configurations in one and two dimensions.

2.5 Linear System Solution

There are many options for solving linear systems including commercial packages, such as Matlab, open source libraries in various languages, such as C++ and FORTRAN, and all the documentation needed to write the methods in custom software. In this work the open source package Eigen version 3.2.2 provided by Tux Family [44] was selected. This package supports both dense and sparse matrices. For this work dense matrices were used for ease of implementation and debugging. There is also uncertainty regarding the true sparseness of the matrices for large support domains and relatively small global domains. The memory footprint and runtimes of this MLPG implementation can be improved by moving to the sparse matrix support.

The Eigen package provides multiple options for decomposing matrices and solving linear systems. In this work the \texttt{PartialPivLU}, \texttt{FullPivLU}, \texttt{HouseholderQR}, and \texttt{FullPivHouseholderQR} solution options were explored. The method of inverting the $K$ matrix and then performing the multiplication is used only within MLS calculations where the matrices are very small, never for the MLPG linear system.
Chapter 3

Boundary Conditions
Implementation

Most problems of interest can be modeled using Dirichlet conditions, Neumann conditions, or a combination of the two. Periodic conditions can also be useful in simplifying the representation of large problems. As with conventional methods boundary condition implementation is one of the biggest challenges in numerical methods. The following sections discuss the standard approaches to boundary condition types within the MLPG method.

3.1 Dirichlet Conditions

Dirichlet conditions specify the solution value along the boundary. This can be used to represent scenarios such as a sink or source term, no slip conditions, or fixed positions. There are two methods frequently used for Dirichlet condition enforcement within MLPG. The first method was the penalty method, included in the initial MLPG publications [1]. The second, more popular method, is Direct Interpolation or collocation [23]. The main drawback to the penalty method is that it introduces another tuning parameter. The penalty parameter provides a way to mask other errors and increases the level of experience and intuition the user must have to successfully apply MLPG to a problem. In this work the Direct Interpolation method was used. This method replaces the governing equations for the nodes on the boundary with Eq. 3.1. In this equation \( x \) is the value in the solution vector and \( F \) is the value which is placed in the force vector. The summation is over the nodes in the subdomain.

\[
\sum_{i=1}^{n} \phi(i) [x] = [F] \quad (3.1)
\]

The corresponding row in the \( K \) matrix consists of only the shape function values. Integra-
tion is not required. The shape functions will sum to one. The highest value will be at the node of interest (the boundary node) and the values will decrease with distance. The $\vec{f}$ contribution is the value being imposed at the evaluation point.

### 3.1.1 Example

The steady two-dimensional Poisson’s problem with Dirichlet conditions on all four edges was used to demonstrate the implementation of the boundary condition within MLPG. The domain is a unit square with the lower left corner at the origin. Thirty-three nodes were placed uniformly along each direction resulting in 1089 total nodes. A radius of influence factor of 2.1 and a two-dimensional quadratic basis function were used. The spline weight function was used as both the test and trial functions. Gaussian quadrature with 6 nodes in each direction was used for integration. The governing equations in LWSF were used without application of the divergence theorem.

The analytical solution is shown in Eq. 3.2. The exact solution is only non-zero on the right edge. The corresponding forcing function is shown in Eq. 3.3. Since the forcing function consists of only spatial terms it can be evaluated using the quadrature point locations, eliminating the need for shape functions.

$$U_{ex} = y(1 - y)x^3$$

$$F = 6xy(1 - y) - 2x^3$$

The problem is steady and the linear system is solved in a single step. An iterative procedure is not required. Figure 3.1 shows the MLPG solution for this problem. This result agrees well with the exact solution and is sufficiently smooth.

### 3.2 Neumann Conditions

Neumann conditions are used to enforce the value of a derivative at the boundary. These are often used to model processes like thermal insulation. Unlike Dirichlet conditions, Neumann conditions are implemented by adding terms to the governing equations for the boundary node. The additional terms represent the boundary and are therefore one order lower than the overall problem dimension. For a two-dimensional problem with Neumann boundary conditions the new terms will be line integrals. Continuing with the two dimensional Poisson’s equation set, Eq. 3.4 shows the addition of the line integral to both the sides of the equation. In this equation
\( \bar{q} \) is the specified value along the Neumann boundary.

\[
\left[ \int_\Omega (\phi_{xx} w + \phi_{yy}) \, d\Omega - \int_\Gamma \phi_n \bar{n} w d\Gamma \right] [u] = \left[ \int_\Omega F w d\Omega + \int_\Gamma \bar{q} \bar{n} w d\Gamma \right]
\] (3.4)

Unlike the line integrals that result from the divergence theorem, these line integrals are along the global boundary and the spline weight functions are non-zero. The line integrals are gradients in the direction normal to the boundary and are multiplied by the value of the normal. When calculating the value of these integrals, care must be taken to use the correct sign and magnitude, especially at corners.

The force vector line integral enforces the boundary condition value through integration. If the value is zero, which is common, the force vector line integral term is zero as well. If the boundary condition is provided analytically then the MLS approximation is not needed, as the boundary value at the quadrature point is easily computed. If the boundary condition is only known at node points, then the MLS approximation is used to obtain the value at the quadrature points.

### 3.2.1 Example

The two-dimensional Poisson’s equation will again be used for this demonstration. In this case both Dirichlet and Neumann conditions were applied. Neumann conditions with zero and non-
zero values were included. The boundary conditions are provided in Eq. 3.5

\[ u = u_{exact}, \quad \text{on } \Gamma_a \]  
\[ \frac{\partial u}{\partial y} = 0, \quad \text{on } \Gamma_b \text{ and } \Gamma_d \]  
\[ \frac{\partial u}{\partial x} = \frac{\partial u_{exact}}{\partial x}, \quad \text{on } \Gamma_c \]  

\[ u_{exact} = (7x + x^7) \cos(\pi y) \] (3.6)
\[ F = \pi^2 (7x + x^7) - 42x^5 \cos(\pi y) \] (3.7)

The domain is shown in Figure 3.2. This scenario is similar to that presented in Bathe [45]. The MLPG simulation used a 31 by 31 uniform node arrangement.

The integration was performed using 16 Gauss points for the line integrals and a 16 by 16 point configuration for domain integrals. The quartic spline with a radius of influence parameter (\(\alpha\)) of 0.5 and a quadrature radius parameter (\(\alpha_q\)) of 2.1 was used. The results are shown in Figure 3.3. The results on all edges and centerlines are shown in Figure 3.4. These results are consistent with the exact solution. The solution is smooth and shows no numerical artifacts near the boundaries or in regions of high gradients.
Figure 3.3: MLPG results contour for two-dimensional Poisson’s with mixed boundary conditions.

Figure 3.4: MLPG results for the mixed boundary condition two-dimensional Poisson’s problem. Results on the boundaries (a) and centerlines in the $x$ and $y$ directions (b).
3.3 Periodic Conditions

Periodic boundary conditions, like symmetry conditions, are often used to reduce domain size. Instead of creating a long channel with specified inlet and outlet conditions, a much smaller channel with a periodic condition can be used for cases such as development of flow profiles. The periodic boundary conditions were implemented by creating additional nodes that extended outside the physical domain (ghost nodes) and were linked with nodes that were inside the domain (parent nodes). This is shown in Figure 3.5. Only the parent nodes receive index numbers and corresponding space within the linear system. During subdomain creation and shape function calculation the ghost nodes are treated as regular nodes. Ghost cells use the index of their parent node when contributing to the linear system matrices. An additional bookkeeping step is required to ensure that the ghost node always has the same state variable values ($u$, $v$, $p$, etc) as its parent node. This can be accomplished through a helper function within the node class or through an explicit loop at the beginning or end of each iteration.

![Figure 3.5: Diagram of nodes in periodic domain.](image)

Most governing equations with only periodic boundary conditions will result in a $K$ matrix with identical rows. Table 3.1 shows an example for the heat equation discussed below on a 9 by 9 node domain. This is in contrast to Dirichlet and Neumann boundary conditions which create additional terms in the system or replace rows entirely. Without periodic domains, any subdomain that intersects with the boundary will be different from interior node rows. This is due to the variance of shape function values associated with a change in the number of and spacing of the available data points.
Table 3.1: Sample K Matrix for Periodic Case

<table>
<thead>
<tr>
<th></th>
<th>0.6076</th>
<th>0.3342</th>
<th>-0.0168</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>-0.0168</th>
<th>0.3342</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.6076</td>
<td>0.3342</td>
<td>-0.0168</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-0.0168</td>
<td>0.3342</td>
<td>0</td>
</tr>
<tr>
<td>0.3342</td>
<td>0.6076</td>
<td>0.3342</td>
<td>-0.0168</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-0.0168</td>
<td>0.3342</td>
<td>0</td>
</tr>
<tr>
<td>-0.0168</td>
<td>0.3342</td>
<td>0.6076</td>
<td>0.3342</td>
<td>-0.0168</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>-0.0168</td>
<td>0.3342</td>
<td>0.6076</td>
<td>0.3342</td>
<td>-0.0168</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-0.0168</td>
<td>0.3342</td>
<td>0.6076</td>
<td>0.3342</td>
<td>-0.0168</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-0.0168</td>
<td>0.3342</td>
<td>0.6076</td>
<td>0.3342</td>
<td>-0.0168</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-0.0168</td>
<td>0.3342</td>
<td>0.6076</td>
<td>0.3342</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-0.0168</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-0.0168</td>
<td>0.3342</td>
<td>0.6076</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-0.0168</td>
<td>0.3342</td>
<td>0.6076</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.3.1 Example

The periodic boundary condition was applied to the heat equation for this example, Eq. 3.8. The Crank-Nicolson integration scheme was applied to the time derivative resulting in Eq. 3.9. The superscript $n$ indicates evaluation at the previous timestep and $n + 1$ indicates evaluation at the current timestep.

$$\frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial x^2}$$ (3.8)

$$T^{n+1} = T^n + \frac{\Delta t}{2} \left( \frac{\partial^2 T}{\partial x^2} |^n + \frac{\partial^2 T}{\partial x^2} |^{n+1} \right)$$ (3.9)

The MLPG formulation of the equation is shown in Eq. 3.10. In this case the only boundary condition is the periodic condition and so this equation is applied at all nodes without modification.

$$\left[ J_q \sum_{i=1}^{\Omega_{mq}} w_q(i) \left( \phi - \frac{\Delta t}{2} \phi_{xx} \right) \right] [T^{n+1}] = \left[ J_q \sum_{i=1}^{\Omega_{mq}} w_q(i) wT^n \left( \phi + \frac{\Delta t}{2} \phi_{xx} \right) \right]$$ (3.10)

A one-dimensional domain extending from -1 to 1 was used. An initial condition of $\sin(\pi x)$ was selected. It is important to consider the consistency of the boundaries when using periodic boundary conditions. Figure 3.6 shows results from a 21 node simulation and a radius factor of 2.1. A quadratic basis function and a quartic spline weight function were used. Six Gaussian points were applied in each linear subdomain for integration. The timestep was 0.001.
Figure 3.6: MLPG results for one-dimensional heat conduction with periodic boundary conditions.
Chapter 4

Validation Cases

Two common fluid dynamics problems were chosen for validation of this MLPG implementation. Stokes Flow is a steady, very low Reynolds number flow. The problem is governed by the two-dimensional Navier-Stokes equations. Due to the low Reynolds numbers it can be modeled without introducing any additional methodology such as a splitting scheme. The second problem is laminar channel flow. This problem was undertaken to demonstrate the behaviour of the method for transient fluid flows. The treatment of an external force is also tested by the channel problem. In this chapter all problems were solved in primitive variable form.

4.1 Stokes Flow

Stokes Flow is governed by the Navier-Stokes equations. In these flows the viscous forces dominate the solution and the non-linear convection terms are not present, eliminating the need for upwinding schemes or pressure splitting methods. The set of governing equations for this problem in primitive variable form is presented below:

\[
\begin{align*}
-\frac{\partial p}{\partial x} + \mu \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) &= F_x \quad (4.1) \\
-\frac{\partial p}{\partial y} + \mu \left( \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) &= F_y \quad (4.2) \\
\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} &= 0 \quad (4.3)
\end{align*}
\]

In this work \( \mu = 1 \) and the forcing functions are 0. Dirichlet boundary conditions are applied to all four edges for both velocity components and for pressure. The analytical solution, shown in Eq. 4.4, is enforced at the boundaries. The domain for this work is a square domain of length 2 in each dimension. The lower left corner is positioned -1 in the \( x \)-direction and -1 in the
\[ u(x, y) = x \]
\[ v(x, y) = -y \]
\[ p(x, y) = 2 \]

4.1.1 Implementation

This problem is a steady problem that can be solved as one linear system. In this case there are three state variables and three equations being solved. Therefore the length \( N \) of the system vectors is calculated by multiplying the number of nodes by the number of state variables. The \( K \) matrix has a size of \( N \times N \). In this work the data is organized such that the unknown vector elements are ordered as \( u_1, v_1, p_1, u_2, v_2, p_2 \) through \( u_N, v_N, p_N \). Another organization would be \( u_1, v_2 \) through \( u_N \) followed by \( v_1, v_2 \) through \( v_N \) then \( p_1, p_2 \) through \( p_N \).

For each node being evaluated three governing equations are being solved, the population of the \( K \) matrix is shown in Table 4.1. This table demonstrates how the equation set is deconstructed into the MLPG contributions for each state variable. These equations will be integrated over each subdomain resulting in contributions from each quadrature point in the subdomain of each interior node.

<table>
<thead>
<tr>
<th></th>
<th>( U )</th>
<th>( V )</th>
<th>( P )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eq. 4.1</td>
<td>(-\mu (\phi_{,xx} + \phi_{,yy})w)</td>
<td>(-\phi_x w)</td>
<td></td>
</tr>
<tr>
<td>Eq. 4.2</td>
<td>(-\mu (\phi_{,xx} + \phi_{,yy})w)</td>
<td>(-\phi_y w)</td>
<td></td>
</tr>
<tr>
<td>Eq. 4.3</td>
<td>(\phi_x w)</td>
<td>(\phi_y w)</td>
<td></td>
</tr>
</tbody>
</table>

For this problem the \( \vec{f} \) vector contributions are 0 for interior nodes. For boundary nodes the Direct Interpolation method is used. This results in \( K \) matrix contributions that are simply \( \phi \) along the diagonal of the block presented in the table. The Direct Interpolation method requires neither approximation or integration. The resulting contribution is the boundary value, the analytical solution, being assigned to \( \vec{f} \).
4.1.2 Results

The MLPG calculation used 31 nodes in each the $x$- and $y$-direction. These nodes were uniformly spaced. The radius of influence parameter ($\alpha$) was 2.1. Integration was accomplished using Gaussian quadrature with 6 Gauss points in each direction.

![MLPG results for Stokes Flow with Dirichlet conditions.](image)

Figure 4.1: MLPG results for Stokes Flow with Dirichlet conditions.

A comparison with the analytical solution was performed using Eq. 4.5. The results for the velocity magnitude error are presented in Figure 4.2. When the analytical solution is less than $10^{-6}$ only the square root of the difference is used as dividing by such a small number results in unreasonable large error values. The error plot shows that very good agreement with the analytical solution was achieved by MLPG. The error is highest when the velocity magnitude is near zero.

$$ Error = \sqrt{\frac{(x-x_{\text{Analytic}})^2}{x_{\text{Analytic}}^2}} $$ (4.5)

Plots of the individual velocity components are shown in Figure 4.3. These plots confirm that the solution is smooth and shows no directional bias. Additionally there is no observable error associated with the boundary condition treatment.
Figure 4.2: Error contour for MLPG computation Stokes Flow with Dirichlet conditions.

Figure 4.3: Contour plots of individual velocity component results for Stokes Flow.
4.2 Laminar Channel Flow

Both Poiseuille flow and Couette flow were used to demonstrate the ability of this MLPG implementation to achieve the steady-state profiles in a time accurate manner. For this work the flows were solved in one dimension as there are no flow features in the second direction. These two flows are governed by the Navier-Stokes equations. Applying simplifications for the one dimensionality and constant pressure, the governing equation is:

\[
\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + F_x
\]

The body force \(F_x\) is zero in the Couette flow and is the driving force in the Pousiille flow. The time derivative is treated using a difference operator while the spatial derivatives are approximated using MLS. Equation 4.7 shows the equation in LWSF. The superscript \(n\) and \(n+1\) indicate previous and current timestep data respectively.

\[
\int_{\Omega} w \phi u^{n+1} d\Omega = \int_{\Omega} w \phi u^n d\Omega + \Delta t \int_{\Omega} w F_x d\Omega
\]

For both flows only Dirchlet boundary conditions are required and so no further manipulation of the equations is required.

4.2.1 Implementation

Both problems were solved in the non-dimensional form with the floor of the channel at \(x = 0\) and the channel height \(L\) equal to 1. These surfaces are treated as no-slip. Thirty-one uniformly spaced nodes were used for the simulation results presented here. A successful simulation was also performed with only 11 uniformly spaced nodes but smoother plots result for the higher nodal density. A radius of influence multiplier (\(\alpha\)) of 0.5 and a quadrature radius (\(\alpha_q\)) of 2.01. The quartic spline was used for both the test and trial function. MLS with a quadratic basis function was used for trial function approximation. Six Gaussian integration points were used for the integration of each subdomain. Dirchlet boundary conditions were satisfied using Direct Interpolation. A non-dimensional timestep of \(10^{-5}\) was selected.

4.2.2 Pouiselle Flow

The Pouiselle flow was modeled using the parameters provided by Morris and Fox [46]. This work provided analytical transient solutions that are ideal for validation. A non-dimensional body force of 640 is applied to mimic pressure gradient driving the flow. The analytical solution
is shown in Eq. 4.8. This equation is modified from the original work [46] to achieve the desired physical behavior. In the original form as \( t \) approached infinity the velocity profile was negative. In the presented formula the final profile is positive. Additionally the summation portion of the equation is subtracted not added from the leading term to ensure that the profile never grows beyond the specified maximum velocity.

\[
u(x,t) = \frac{F_x}{2\nu} x(L - x) - \sum_{n=0}^{\infty} \left[ \frac{4F_xL^2}{\nu \pi^3 (2n + 1)^3} \right] \\
\sin \left( \frac{\pi x}{L} (2n + 1) \right) \exp \left( -\frac{(2n + 1)^2 \pi^2 \nu}{L^2 t} \right) \tag{4.8}
\]

The simulation was run for 3,500 iterations which is equivalent to 0.0175 in non-dimensional time. The MLPG results compared well with the analytical solution at every time step. Figure 4.4 shows both solutions. The symbols are the MLPG solution and the solid lines are the analytical solution. The colors indicate the time as shown by the legend.

![Figure 4.4: Comparison of transient MLPG results (symbols) and the analytical solution (solid line) of laminar Poisuielle flow.](image)

The MLPG solution maintains symmetry at each point in time. No error is observed at the boundary points. This case includes a body force that is constant. However the implementation was done generically so that non-constant body functions, but not functions of \( u \) could be easily
substituted. The body force must be integrated but does not need to approximated via shape functions. Therefore its contribution to the $\vec{f}$ should be made per quadrature point, but outside the loop over the quadrature subdomain nodes.

### 4.2.3 Couette Flow

Couette flow is driven by an applied velocity at one boundary point instead of by a body force or pressure gradient. This problem is essentially a one dimensional version of the lid-driven cavity application that will be studied in the Chapter 6. The flow parameters and analytical solution were found in Morris and Fox [46] as well. The specified ceiling velocity, $U_0$, is 1. The analytical solution is shown in Eq. 4.9.

$$u(x,t) = \frac{U_0}{L} x + \sum_{n=1}^{\infty} \left[ \frac{2U_0}{n\pi} \left( -1 \right)^n \sin \left( \frac{n\pi x}{L} \right) \exp \left( -\nu \frac{n^2 \pi^2}{L^2} t \right) \right]$$  \hspace{1cm} (4.9)

The solution was run to 3,500 iterations or 0.0175 time units. Figure 4.5 shows the comparison of MLPG results and the analytical solution at various points in time. The symbols are the MLPG solution and the solid lines are the analytical solution. The colors indicate the time as shown by the legend.

![Figure 4.5: Comparison of transient MLPG results (symbols) and the analytical solution (solid line) of laminar Couette flow.](image-url)
The agreement is excellent in late time. However the initial steps show some error in the MLPG solution near the stationary (lower) boundary. Near the plate a boundary layer forms and higher nodal density is required to fully resolve the details.

4.3 Conclusion

In this chapter various benchmark fluids applications are presented. MLPG successfully models these problems without the any modifications, tricks, or tuning parameters. This indicates that an accurate implementation of MLPG has been created for use in more complex problems or as a basis for further development. The only negative observation is that the runtime of MLPG is much slow in comparison to Finite Difference codes of similar accuracy.
Chapter 5

Shape Function Interpolation Method - Boundary Condition

The standard treatment of Neumann boundary conditions is the line integral approach discussed in Section 3.2. This methodology is common in Finite Element and solid mechanics modeling. In this work a new boundary condition methodology for Neumann boundary conditions enforcement in MLPG has been created. Named the Shape Function Interpolation Method (SFIM), this methodology is more similar to Finite Difference boundary conditions and to the Direct Interpolation method. This novel method leverages the shape function information available through MLS. This boundary condition approach is less computationally intensive, is easily implemented in any number of dimensions, and is more easily applied to different domain geometries than the traditional approach. In this chapter the details of the methodology will be discussed in depth. Results for two-dimensional heat transfer problems on various domains will be presented and runtime comparisons with the standard method will be discussed.

5.1 Methodology

In the Finite Difference method the forward or backward difference operator is applied to enforce the Neumann, also called gradient or derivative, boundary condition. These operators are shown in Table 5.1, where $h_x$ is the cell dimension and $x_0$ is the node of interest.

The most simple application of the Finite Difference approach is the first order approximation when the Neumann condition is prescribed to be zero. In this case the implementation of the condition along the left boundary simplifies to an assignment statement, Eq. 5.1. Depending on the solution method these boundary conditions may use previous timestep data or current
### Table 5.1: Finite Difference Operators

<table>
<thead>
<tr>
<th>Type</th>
<th>Order of Accuracy</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>1</td>
<td>( \frac{f(x + h x) - f(x_0)}{h x} )</td>
</tr>
<tr>
<td>Forward</td>
<td>2</td>
<td>( -\frac{3f(x_0) + 4f(x + h x) - f(x + 2h x)}{2h x} )</td>
</tr>
<tr>
<td>Backward</td>
<td>1</td>
<td>( \frac{f(x_0) - f(x - h x)}{h x} )</td>
</tr>
<tr>
<td>Backward</td>
<td>2</td>
<td>( \frac{3f(x_0) - 4f(x - h x) + f(x - 2h x)}{2h x} )</td>
</tr>
</tbody>
</table>

timestep data for evaluation.

\[
\begin{align*}
\frac{\partial u}{\partial x} &= 0 \\
\frac{u(x + h) - u(h)}{h} &= 0 \\
u(h) &= u(x + h)
\end{align*}
\] (5.1)

The SFIM approach solves for the unknown boundary value using the MLS representation of the prescribed condition, Eq. 5.2. In this case \( n \) is the size of the subdomain around the boundary node.

\[
\begin{align*}
\frac{\partial \mathbf{u}}{\partial x} &= 0 \\
\phi_x \mathbf{u} &= 0 \\
\sum_{i=1}^{n} \phi_x(i) \mathbf{u}(i) &= 0
\end{align*}
\] (5.2)

The summation of shape functions is then decomposed to isolate the unknown boundary node product:

\[
\phi_x(j) \mathbf{u}(j) + \sum_{i \neq j}^{n} \phi_x(i) \mathbf{u}(i) = 0
\] (5.3)

The final step is to isolate the unknown value. In this case the prescribed boundary condition value is zero. For more complex cases non-zero expressions would be present on the right hand side (RHS) of the equation. Equation 5.4 shows the final form with an arbitrary RHS. The
system is more complicated when both $x$ and $y$ normal components are present.

$$u(j) = \frac{1}{\phi_x(j)} \left[ RHS - \sum_{i \neq j}^n \phi_x(i) u(i) \right]$$ \hspace{1cm} (5.4)

Equation 5.4 is implemented in a fashion similar to the Direct Interpolation method. The $K$ matrix contribution is 1 for the node of interest and 0 everywhere else. The $\vec{f}$ contribution is the right hand side of Eq. 5.4 at the node of interest only.

For applications where the $RHS$ contains nodal state information the system is using data from the previous timestep. Therefore it is useful to evaluate the equation again independently of the linear system as a post-processing step. During the second evaluation the current timestep data is available and the solution is more accurate.

Note that although the procedure is being performed twice per iteration for each boundary node it is still less computationally intensive than the traditional integral method. The integral method does the MLS computation at each of the quadrature points in the domain and then again for the line integral. Additionally if the $RHS$ is non-zero the line integral may be computed twice, once for $K$ and once for $\vec{f}$.

### 5.2 Computing Normals

A methodology for obtaining the local normal from the domain nodes was created to simplify the process of computing the Neumann boundary condition value for arbitrary or shape changing domains. The only requirements for this process are (1) the domain is closed, (2) the boundary nodes are indicated as such, and (3) a function for returning whether a point is contained in the domain is available. Many external packages exist for creating domains, marking boundary nodes, and determining inclusion. For complex systems, direct integration with these packages may be necessary. The methodology for computing normals and their use to solve for a single gradient value is discussed in this section. Results of the computation of the normals for various geometries are included as well.

This process is performed node by node. Figure 5.1 shows a section of a closed domain that will be used for this discussion. Note that the boundary nodes are marked and that each of these boundary nodes has an inward pointing normal with non-zero magnitudes for both the $x$ and $y$ components.

For the node of interest a standard subdomain is obtained using the test function and the quadrature radius factor. Using the standard support size will not yield enough data points. Figure 5.2 shows the subdomain for the node of interest (marked in yellow). The methodology requires that at least two boundary nodes in addition to the node of interest are contained
within the subdomain. The subdomain will attempt to enlarge itself to encompass the required number of nodes if needed.

A set of line segments connecting the point of interest to each boundary node in the subdomain is generated. A pair of normal vectors originating at the midpoint of each line segment is created. The endpoints of these vectors are checked for containment within the global domain. Only one of the two points should be contained. The results of this for rightmost boundary node and the point of interest is shown in Figure 5.3. In this case the upper endpoint would be retained for future use as it is the one contained with the domain.

If the number of test points from each point of interest-boundary point pair contained in the domain is not equal to 1, additional logic must be used. If both points are contained then the assumed length of the vectors is increased, and the containment test is performed again. If neither point is contained the distance is decreased and the containment process is performed again.

Once each point of interest-boundary node pair has been evaluated the resulting endpoints are averaged to yield a new endpoint that is representative of point of interest. A unit vector in the direction of the point of interest from the new point is created. Figure 5.4 shows these vectors. The lengths were modified for clarity of discussion.

Now that a local normal has been established, an angle of rotation is needed. This angle represents the rotation needed to yield a normal vector for the node of interest that is in the positive y direction. The angle is computed by taking the arccosine of the dot product of the computed and desired normal vector.
Figure 5.2: Sample subdomain for node of interest, in yellow, with boundary nodes shown in blue.

Figure 5.3: Depiction of local normal vector computation test points.
All points in the subdomain are then rotated by this quantity. This includes interior nodes, boundary nodes, and the node of interest. The rotated subdomain is then passed into the MLS methodology to compute the shape functions required for the first derivative in the $y$ direction. Figure 5.5 shows the sample subdomain after rotation.

During the course of a simulation all boundary points with assigned Neumann boundary conditions will undergo the discussed process. Figure 5.6 shows the normals for each boundary point in the sample domain.

The computation of the local normals uses MLPG tools such as the identification of a subdomain via the weight function and the calculation of MLS shape functions, but is independent of the MLPG calculation. The boundary nodes and local normals for four geometries are plotted. In these plots the red symbols are the boundary nodes and the green symbols are the normal points. The actual length of the normal vector is irrelevant and so the overlap of normal nodes seen in the figures is not a concern. The angle of rotation is the true goal of the process.

- **Right Triangle** (Figure 5.7) - demonstrates the performance in an domain that is not centered at $(0,0)$ and has acute angles.

- **Square Plate** (Figure 5.8) - demonstrates performance of right angles and shows that the normals are symmetric and independent of quadrant.

- **Isosceles Triangle** (Figure 5.9) - demonstrates a domain with no right angles and shows symmetry is maintained.

- **Plate with Cutout** (Figure 5.10) - demonstrates the ability to calculate inward facing normal vectors.
Figure 5.5: Set of subdomain nodes rotated based on the local normal and the desired normal in the $y$ direction.

Figure 5.6: Depiction of the final normals for each boundary node in the sample domain.
for exterior and immersed boundaries.

**Figure 5.7:** Boundary nodes and local

**Figure 5.8:** Temperature contour at t=1.0

**Figure 5.9:** Temperature contour at t=2.5

**Figure 5.10:** Temperature contour at t=15

### 5.3 Validation

The new boundary condition and local normal calculation process were validated in MLPG using a two-dimensional heat transfer application. A plate with insulated edges was used as the domain. This yields Neumann boundary conditions on all edges. The center point of the plate was assigned a constant temperature and the heat dissipates from that point to the entirety of
the plate. The problem is complete when the entire plate has reached a steady state temperature equal to the specified center temperature.

This problem is governed by the standard heat equation, Eq. 5.5. The Crank-Nicolson scheme was applied to the time derivative yielding Equation 5.6, where \( n \) is the previous timestep and \( n + 1 \) is the current timestep.

\[
\frac{\partial T}{\partial t} = \alpha \left( \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right) \tag{5.5}
\]

\[
T^{n+1} - \frac{\alpha \Delta t}{2} \left( \frac{\partial^2 T^{n+1}}{\partial x^2} + \frac{\partial^2 T^{n+1}}{\partial y^2} \right) = T^n + \frac{\alpha \Delta t}{2} \left( \frac{\partial^2 T^n}{\partial x^2} + \frac{\partial^2 T^n}{\partial y^2} \right) \tag{5.6}
\]

The equation is then transformed into the LSWF and the MLS shape functions are substituted for the spatial derivatives. In these computations a quadratic basis function was used. The quartic spline was selected as both the test and trial function. Integration was performed using 6 Gaussian quadrature points in each direction.

\[
\int_\Omega \left( \phi - \frac{\alpha \Delta t}{2} (\phi_{xx} + \phi_{yy}) \right) [T^{n+1}] \, d\Omega = \int_\Omega \phi T^n + \frac{\alpha \Delta t}{2} (\phi_{xx} + \phi_{yy}) T^n \, d\Omega \tag{5.7}
\]

A square domain, as shown in Figure 5.11, was selected to validate the implementation of the SFIM boundary condition. Finite Difference, MLPG with line integral Neumann conditions, and MLPG with SFIM boundary conditions are compared. All three solution methods used 21 nodes in each the \( x \) and \( y \) direction, a time step of 0.001, and a fixed non-dimensional temperature of 1 at the center of the plate. The heat transfer coefficient \( \alpha \) is 1 as well.

All three methods achieved the desired steady state temperature to 8 significant digits and had symmetric temperature profiles across the plate. Figure 5.12 shows a comparison of the profiles from the three solution methods at various points in time.
Figure 5.12: Plot of temperature profiles along the $y=0$ at different times. Lines are Finite Difference, Squares are MLPG, and Diamonds are MLPG with SFIM

Temperature contours from the MLPG solution with the SFIM boundary condition are shown in Figures 5.13 - 5.16. Notice that there are no artifacts at the boundaries or corners, and that solution is symmetric in all directions.

For this test case SFIM achieved runtimes were 30% faster than the traditional boundary condition implementation. This runtime improvement will only be increased when a non-zero prescribed condition is applied.

### 5.4 Demonstration on Immersed Boundary Geometry

The validation problem above was repeated on additional geometries to demonstrate the ability of the new methodology to be easily applied to various closed geometries. The computation of the normals into the domain in the local region of each boundary node is discussed in Section 5.1 above. Very few code modifications required to transition between the square plate above and new geometry. The modifications are functions to (1) create the grid, (2) identify the boundary nodes, which is often done during the grid creation process, and (3) compute containment of a location with the domain.

The immersed boundary problem selected for this work is a square plate with a circular hole in the center. This plate is offset from the axis to demonstrate the accuracy of the normals for positive and negative coordinates. Figure 5.17 shows the set up. The interior of the circular cutout and the exterior of the plate are insulated. The constant point source temperature is 1 and is applied at the point (-0.15, 1.0). The heat transfer coefficient $\alpha$ is 1 and the time step is 0.001. This problem uses the same MLPG configuration discussed in Section 5.3. The only
A modification to the code to perform this application was the customization of the grid creation function and the containment function.

The solution successfully reached steady state and contours of temperature at four points in time are shown in Figures 5.18 - 5.21. The pixilated appearance is due to the import of the nodal arrangement into the plotting software. Attempts to show a smooth contour via triangulation created visual artifacts that obscured the symmetry of the solution.

The temperature profiles for three vertical slices of the domain at time of 2 are shown in Figure 5.22. These show the smooth interaction of the profile with the circular hole as well as the symmetric propagation of the heat. It is also observed that the zero gradient boundary condition is correctly enforced.

This test case demonstrated the ease of use of the SFIM boundary condition within MLPG for a complex domain. To use the standard integral method additional information would be
required for the boundary nodes to indicate which surface they belong to and their inward facing normal. The quadrature domains and weight functions would have to be modified to handle the circular geometry. Finally additional logic would be required to select the correct quadrature and weight function logic for each boundary shape.

5.5 Conclusion

The SFIM provides an efficient system for enforcing Neumann boundary conditions on a variety of domain shapes. This methodology is consistent with the fundamental idea of MLPG that node connectivity is not needed and nodes need not be stationary. This chapter has demonstrated the local normal computation procedure as well as the performance of the new boundary condition for two-dimensional heat transfer problems on square and immersed boundary domains. The SFIM method brings MLPG closer to successful modeling of real world fluids applications.
Figure 5.18: Temperature contour at t=0.03. Figure 5.19: Temperature contour at t=0.5

Figure 5.20: Temperature contour at t=1  Figure 5.21: Steady state temperature contour

Figure 5.22: Temperature profiles at three $x$ positions through the plate with the circular hole at $t=2$.  

56
Chapter 6

Lid-Driven Flow in a Square Cavity

The lid-driven cavity is a two-dimensional, laminar, incompressible flow in a square domain. The top wall is assigned a horizontal velocity while the other walls are stationary and have no-slip boundary conditions. This results in a vortical flow with singularities in two corners. In this chapter the lid-driven square cavity problem is modeled using MLPG. The lid-driven cavity results of Ghia et al. [36] are used for comparison of the velocity profiles and refereed to as exact results. Ghia et al. present results obtained from a coupled strongly implicit multi-grid solver and are used as the reference point for most lid-driven cavity publications. This chapter presents the development of the governing equations, discusses the problem set-up and boundary conditions, outlines the MLPG specific implementation, and presents results using both the standard integral and SFIM Neumann boundary condition treatments.

6.1 Governing Equations

This problem is governed by the two-dimensional, incompressible Navier-Stokes equations for constant density. The equations in primitive variable form are shown in Eq. 6.1 and Eq. 6.2.

\[
\frac{\partial u_i}{\partial t} + \frac{\partial u_i u_j}{\partial x_j} = -\frac{\partial P}{\partial x_i} + \frac{1}{Re} \frac{\partial^2 u_i}{\partial x_j \partial x_j} \tag{6.1}
\]

\[
\frac{\partial u_j}{\partial x_j} = 0 \tag{6.2}
\]

Following Najafi et al. [39] the Characteristic-Based Split (CBS) approach was used to convert the equations into a numerically solvable form. The solution process is a three step methodology:
1. Intermediate velocity components ($\tilde{u}, \tilde{v}$):

\[
\tilde{u}^{n+1} = u^n + \Delta t \left[ -\left( \frac{\partial (uu)}{\partial x} + \frac{\partial (uv)}{\partial y} \right) + \frac{1}{Re} \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \right] 
+ \frac{\Delta t}{2} \left( u \frac{\partial}{\partial x} \left( \frac{\partial (uu)}{\partial x} + \frac{\partial (uv)}{\partial y} \right) + v \frac{\partial}{\partial y} \left( \frac{\partial (uu)}{\partial x} + \frac{\partial (uv)}{\partial y} \right) \right) \right]^{n+1}
\]

\[
\tilde{v}^{n+1} = v^n + \Delta t \left[ -\left( \frac{\partial (uv)}{\partial x} + \frac{\partial (vv)}{\partial y} \right) + \frac{1}{Re} \left( \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) \right] 
+ \frac{\Delta t}{2} \left( u \frac{\partial}{\partial x} \left( \frac{\partial (uv)}{\partial x} + \frac{\partial (vv)}{\partial y} \right) + v \frac{\partial}{\partial y} \left( \frac{\partial (uv)}{\partial x} + \frac{\partial (vv)}{\partial y} \right) \right) \right]^{n+1}
\]

2. Pressure via Poisson’s equation ($P$):

\[
\left( \frac{\partial^2 P}{\partial x^2} + \frac{\partial^2 P}{\partial y^2} \right)^{n+1} = \frac{1}{\Delta t} \left( \frac{\partial \tilde{u}}{\partial x} + \frac{\partial \tilde{v}}{\partial y} \right)^{n+1}
\]

3. Velocity Correction ($u, v$):

\[
u_i^{n+1} = \tilde{u}_i^{n+1} - \Delta t \left( \frac{\partial P}{\partial x} \right)^{n+1}
\]

\[
v_i^{n+1} = \tilde{v}_i^{n+1} - \Delta t \left( \frac{\partial P}{\partial y} \right)^{n+1}
\]

Next the equations are transformed into the LWSF by integrating over the subdomain and multiplying by the test function ($w$). The final step is to convert all state variables and state variable derivatives to approximations using the shape function ($\phi$) and shape function derivatives. The state variable on the right hand side is separated to represent the $K\vec{x} = \vec{f}$ form.
1. Intermediate velocity components (MLPG Form) \( (\tilde{u}, \tilde{v}) \):

\[
\begin{align*}
\int_{\Omega} \phi w \left[ \tilde{u}^{n+1} \right] d\Omega &= \int_{\Omega} w \phi u^n d\Omega - \Delta t \int_{\Omega} w (\phi_x uu + \phi_y uv)^n d\Omega \\
&+ \int_{\Omega} \frac{\Delta t}{Re} w (\phi_x x u + \phi_y y u)^n d\Omega \\
&+ \frac{\Delta t^2}{2} \int_{\Omega} w \phi u^n (\phi_x x u + \phi_y y u)^n d\Omega \\
&+ \frac{\Delta t^2}{2} \int_{\Omega} w \phi v^n (\phi_x x v + \phi_y y v)^n d\Omega 
\end{align*}
\]

\[
\begin{align*}
\int_{\Omega} \phi w \left[ \tilde{v}^{n+1} \right] d\Omega &= \int_{\Omega} w \phi v^n d\Omega - \Delta t \int_{\Omega} w (\phi_x vv + \phi_y vv)^n d\Omega \\
&+ \int_{\Omega} \frac{\Delta t}{Re} w (\phi_x x v + \phi_y y v)^n d\Omega \\
&+ \frac{\Delta t^2}{2} \int_{\Omega} w \phi u^n (\phi_x x v + \phi_y y v)^n d\Omega \\
&+ \frac{\Delta t^2}{2} \int_{\Omega} w \phi v^n (\phi_x x v + \phi_y y v)^n d\Omega 
\end{align*}
\]

2. Pressure via Poisson’s equation (MLPG Form) \( P \):

\[
(\phi_{xx} + \phi_{yy}) \left[ P \right]^{n+1} = \frac{1}{\Delta t} (\phi_x \tilde{u} + \phi_y \tilde{v})^{n+1}
\]

3. Velocity Correction (MLPG form) \( u, v \):

\[
\begin{align*}
\int_{\Omega} w \phi \left[ u^{n+1} \right] d\Omega &= \int_{\Omega} w \phi u^{n+1} d\Omega - \Delta t \int_{\Omega} w \phi_x P^{n+1} d\Omega \\
\int_{\Omega} w \phi \left[ v^{n+1} \right] d\Omega &= \int_{\Omega} w \phi v^{n+1} d\Omega - \Delta t \int_{\Omega} w \phi_y P^{n+1} d\Omega 
\end{align*}
\]

These equations are now solved during each iteration of the transient calculation.

### 6.2 Domain Characterization

The square cavity was selected for this validation work. The cavity is a unit square with the lower left corner located at \((0,0)\). All boundaries are treated as no-slip. The top boundary has an assigned \( u \) velocity component of 1 that drives the development of the flow field. The geometry is shown in Figure 6.1.

The no-slip conditions require a high nodal density near the boundaries to properly resolve the evolving gradients. Najafi et al. proposed a hyperbolic tangent based node clustering func-
Equation 6.12 shows the equation for computing the modified node locations. In this equation \( C_{\text{spacing}} \) is a constant used to control the nodal spacing. \( x' \) is the modified node location while \( x \) is the uniform spacing location. This equation is applied to both the \( x \) and \( y \) coordinate of each point.

\[
x' = \frac{1 + \tanh(C_{\text{spacing}} x)}{2 \tanh(C_{\text{spacing}})}
\]  

(6.12)

For this work 31 nodes in each direction with a spacing factor of 1.7 were used for most cases. For higher Reynolds numbers 41 nodes in each direction were used but the spacing factor remained 1.7.

### 6.3 Implementation

This section discusses the MLPG specific aspects of modeling the lid-driven cavity case. A brief listing of MLPG feature settings is presented, followed by additional discussion for matrix construction and boundary conditions. Nodal subdomains were constructed using a radius factor of 0.75. For uniform grids a radius multiplier of 0.5 was used to minimize overlap. Recall that the radius factor is applied to the distance to the nearest neighbor in each Cartesian direction. For non-uniform nodal layouts such as the one used here, a factor of 0.5 fails to provide complete coverage of the global domain. The spline weight function was used as the test function to
Numerical integration was performed using Gaussian quadrature using 6 nodes per direction. This resulted in a 36 point grid of integration points within the subdomains. For line integrals only 6 points were needed. A quadrature radius factor (α_q) of 2.25 was used. When using a non-uniform nodal arrangement, a larger quadrature radius factor is needed to capture the correct number of support points for the MLS computation. A quadratic function was used for the MLS basis. A quartic spline was selected as the trial function. The linear system was solved using the FullPivLU solver within the Eigen Tux library. A time step of 0.001 was selected.

6.3.1 Matrix Construction

In steps one and three of the solution process two state variables are being solved for. In this case the equations allow for the linear system to be treated as either two separate systems of size N or one system of size N^2. The approach of using two separate linear systems was used for this work. Additionally the K matrices for each velocity component in the first and third CBS steps is identical. Therefore a single matrix was calculated and used in solution of the 4 linear systems. An additional optimization was leveraged with regard to the matrices. The nodal arrangement in this problem is static and all data in the K matrices are independent of state variable information. Therefore the K matrices were constructed as a preprocessing step and not within the iteration loop.

An initial difficulty with applying MLPG to the CBS form of the Navier-Stokes equations was the proper representation of the products and other complex terms. For example, in the cal-
culation of the intermediate velocities, non-linear terms and products of derivatives are present. Equation 6.13 shows the derivative of a product which appears to have multiple MLS representations (a) through (c), Eq. 6.14.

\[ \frac{\partial (uu)}{\partial x} \]  \hspace{1cm} (6.13)

\begin{align*}
(a) & \quad \phi_x (uu) \\
(b) & \quad 2u \phi_x u \\
(c) & \quad 2\phi u \phi_x u
\end{align*}

In this case the (a) is accurate due to the way the equations are evaluated in the MLPG implementation. Option (c) should definitely be avoided as the product of the shape functions can yield unexpected results. Another term with multiple possible implementations is:

\[ v \frac{\partial}{\partial y} \left( \frac{\partial (uu)}{\partial x} \right) \]  \hspace{1cm} (6.15)

In this expression there are two derivatives and three state variable meaning that an expression can be written with as many as 5 shape functions and shape function derivatives. By distributing the \( y \) derivative into the parenthesis MLS can be used to compute the \( xy \) derivative, preventing one of the shape function products. In this work the term was modeled as:

\[ \phi v \phi_{xy} (uu) \]  \hspace{1cm} (6.16)

### 6.3.2 Boundary Conditions

The no-slip boundary conditions for the \( u \) and \( v \) velocity components in the first and third step of the solution process are enforced using the Direct Interpolation method. The lid velocity is set to 1 from the initial time step forward. The approach of slowly increasing the lid velocity from rest to the desired velocity creates a less sharp gradient and improve solution stability for some methods. It was not needed in this work.

The pressure computation requires more complex boundary conditions. A Neumann conditions is applied to all the walls in combination with a reference pressure. Although many papers state the condition of \( \frac{\partial P}{\partial n} \), this is not consistent with the physical system. The applied velocity of the lid creates a pressure gradient in the upper corners that converts the initial horizontal velocity into a vertical one. Therefore the specified pressure gradient in this work was derived from the momentum equation. Equation 6.17 shows the boundary condition where \( n \) indicates
the normal direction. At the corners the normal from each direction must be applied.

\[
\frac{\partial P}{\partial n} = \frac{1}{Re} \frac{\partial^2 u_n}{\partial n^2}
\]  

(6.17)

The pressure reference point was located in the lower left quadrant, at the node closest to \((0.75, 0.25)\). The exact location varied slightly for nodal densities and distributions. The reference pressure was not set to zero but to the previous pressure of a neighboring node. The system pressure increases during the simulation as energy is introduced into the system by the continuously moving lid.

6.4 Results

The results section is composed of two focus areas. The first is the performance of the SFIM boundary condition for the Reynolds number of 100 case. The second is the performance of MLPG for lid-driven cavities with different Reynolds numbers. Overall the MLPG method performed well for low Reynolds numbers cases of lid-driven flow in a square cavity.

6.4.1 Reynolds Number 100

The Reynolds number 100 case was used to validate the two configurations of the MLPG method. In this section results labeled MLPG using the traditional line integral method for Neumann boundary condition enforcement. While MLPG-SFIM results are MLPG with the novel boundary condition treatment introduced in this research. Finite Difference results and the published multi-grid results of Ghia et al. are used in the comparisons as well. The Finite Difference code used a higher density but uniform grid. With first order differencing of the Eqs 6.3 – 6.6.

Figure 6.3 shows that all the solution methods achieve the expected final velocity profile. The left figure shows the \(u\) velocity component along the vertical line at \(x = 0.5\). The right portion of Figure 6.3 shows the \(v\) component along the horizontal line at \(y = 0.5\).

Since the agreement between methods was favorable, MLPG with SFIM is used for the examination of the development of the solution at this Reynolds number. Figure 6.4 shows the evolution of the velocity components through time. At early time the lid velocity is enforced along the top boundary but the energy has only been transferred to a small region of the domain. This results in the sharp angle in the \(u\) component profile. This feature is what many researchers try to avoid by accelerating the lid gradually instead of assigning the maximum lid velocity during the first time step. The \(v\) component along the centerline remains zero.

During the first time step the boundary condition on the lid provides a non-zero \(u\) velocity component for nodes close to that boundary. In the corners of the domain a pressure gradient
Figure 6.3: Velocity component profiles for the steady-state solution of the lid-driven flow in a square cavity with a Reynolds number of 100.

Figure 6.4: Velocity component profile development for the lid-driven flow in a square cavity with a Reynolds number of 100 using MLPG with SFIM.
develops as the horizontally moving fluid encounters the walls. During the correction step the pressure transfers energy to the $v$ velocity component.

The final velocity components are shown in Figure 6.5. The $u$ component velocity remains highest at the lid and the viscous and convective forces dictate the final profiles. The $v$ velocity component shows the vortical flow pattern induced by the lid motion. Figure 6.5 shows contour plots of the velocity components at $t=1$.

![Figure 6.5: Contour plots of velocity components for the lid-driven cavity at $t=10$ for the Reynolds number 100 case.](image)

Figure 6.6 contains contour plots of velocity magnitude and pressure for the same point in time. As expected the velocity plot shows the vortical flow pattern and the pressure contour shows the interaction at the corners of the lid.

### 6.4.2 Additional Reynolds Number

The MLPG configuration using the line integral Neumann boundary condition treatment was used to model the lid-driven square cavity at additional Reynolds numbers. Again Finite Difference results and results published by Ghia et al. are used for comparison. For these case the MLPG code used a 41 by 41 node density and a 1.7 nodal concentration factor. The node density was selected as a balance between extremely long run times and desired accuracy.

The velocity component profiles for the Reynolds number of 400 case are shown in Figure 6.7. The Finite Difference code shows good agreement, especially for the $v$ component. The MLPG implementation falls short of the expected profile. Both methods capture the behavior near the
Figure 6.6: Contour plots of velocity and pressure for the lid driven cavity at t=1 for the Reynolds number 100 case.

lid but the final interior features are not as accurate. The case was allowed to run thousands of iterations past steady-state and no changes were observed. This indicates that energy is not being lost from the system each iteration. Further increasing the nodal density may resolve improve the results. However the computation was stable and shows the expected features.

Next the Reynolds number of 1000 case were modeled. Velocity component profiles are shown in Figure 6.8. Again the Finite Difference results show good agreement with Ghia. MLPG however shows lack of convergence and signs of instability especial near the sharp gradients in both profiles. The behaviour near the lid is resolved well. As shown in figure the flow near the lower and left surfaces begins to reverse direction as the solution becomes unstable.

Finally the Reynolds number 3200 cases was attempted. MLPG failed to reach a steady-state results and again shows error at the lower and left walls. The velocity component profiles are shown in Figure 6.9. The higher Reynolds number cases could most likely be modeled more successfully with higher density nodal arrangements than were used in this work. The runtime is prohibitive at this time.

6.5 Conclusion

In this chapter MLPG was used to successfully model lid-driven flow in a square cavity for various Reynolds numbers. The MLPG implementation showed good agreement at lower Reynolds numbers but was unable to accurately model the steady-state at Reynolds numbers 1000 and 3200. The SFIM boundary condition was successfully validated on this problem. Although the geometry is simple, the enforced Neumann condition is a function of the state variables at the
Figure 6.7: Velocity component profiles for the steady-state solution of the lid-driven flow in a square cavity with a Reynolds number of 400.

Figure 6.8: Velocity component profiles for the steady-state solution of the lid-driven square cavity with a Reynolds number of 1000.
Figure 6.9: Velocity component profiles for the steady state-solution of the lid-driven flow in a square cavity with a Reynolds number of 3200.

previous time step. The lid-driven cavity is a stepping stone to more complex fluid flows that can be modeled with the CBS Navier-Stokes equations.
Chapter 7

Rayleigh-Taylor Instability
Implementation

The Rayleigh-Taylor Instability (RTI) is a density driven fluid interface problem. In this case the fluids are taken to be immiscible. A more dense fluid is suspended above a lower density fluid. The interface between the fluids is perturbed initiating the downward motion of the heavy fluid. Gravity is applied and sustains the movement. The more dense fluid displaces the lighter fluid which moves upward. Eventually the system arrives at a steady-state with the heavy fluid below the lighter fluid. In numerical modeling however the goal is to capture the characteristic shape of column of heavy fluid as it moves through the less dense fluid, and the rate at which it moves. To date even the more complex MLPG fluids applications have focused on the constant density Navier-Stokes applications. This application introduces the variable density component as well as the ability to model the evolving interface. The RTI case provides a foundation for more complex mixing applications. In this chapter the RTI governing equations are presented followed by a discussion of boundary conditions, initial conditions, and MLPG implementation. The results are presented with a focus on the density field and the growth rate of the interface.

7.1 Governing Equations

RTI is modeled using the variable density two-dimensional Navier-Stokes equations in primitive variable form, Equations 7.1, 7.2, and 7.3. Note the addition of gravity force, $g$. In this application the gravity acts in only the $y$ direction. Equation 7.3 is added to the equation set used for the lid-driven cavity case. This equation computes the change in the density field due
to the velocity and enforces conservation.

\[ \frac{\partial \rho u_i}{\partial t} + \frac{\partial (\rho u_i u_j)}{\partial x_j} = -\frac{\partial P}{\partial x_i} + \mu \frac{\partial^2 u_i}{\partial x_j \partial x_j} + g \delta \rho \]  

(7.1)

\[ \frac{\partial u_j}{\partial x_j} = 0 \]  

(7.2)

\[ \frac{\partial \delta \rho}{\partial t} + u_j \frac{\partial \rho}{\partial x_j} = 0 \]  

(7.3)

In this problem there are three representations of density. The first density is a reference density. This is the density of each node for the initial state before the interface is perturbed, Figure 7.1(a). In these figures the red indicates the more dense fluid and the blue indicates the less dense fluid.

Figure 7.1: Depiction of the RTI initial density distributions.

In this work the total density is computed as \( \rho = \rho_{\text{ref}} + \delta \rho \). The final density is \( \delta \rho \) is the difference between the reference density and the total density. Figure 7.1(c) shows \( \delta \rho \) for the initial condition. Although the total density, \( \rho \), is used in the governing equation set to compute momentum, \( \delta \rho \) is the quantity that the gravity force acts on. During the computation the \( \delta \rho \) is solved for and then the total density is updated.

The changing density is also used to compute \( \mu \) which appears in the governing equations. In this work the kinematic viscosity \( \nu \) is held constant and so \( \mu \) is easily updated using the expression in Eq. 7.4.

\[ \mu = \rho \nu \]  

(7.4)
The CBS procedure is applied to put the equations set in a form that can be numerically solved. For this problem the second order series expansion terms ($\Delta t^2$) were not included. Although this reduces the small scale mixing seen at later times in the calculation, the solution still have the characteristic elements of RTI. Also note that $\mu$ is not constant, but included within the derivative. This is because $\mu$ is a function of the spatially and temporally varying density. This equation set is consistent with both Chandrasekhar [47] and Daly [48]. Equations 7.5 through 7.8 are the set of governing equations after CBS has been applied. They appear here in the order that they are solved within the MLPG solution process.

$$\delta \rho^{n+1}_i = \delta \rho^n_i - \Delta t \left( \frac{\partial (u_i \delta \rho)}{\partial x_i} \right)$$  \hspace{1cm} (7.5)

$$\tilde{u}^{n+1}_i = u^n_i + \frac{\Delta t}{\rho} \left[ \frac{\partial (\mu u_j u_i)}{\partial x_j} + \mu \frac{\partial^2 u_i}{\partial x_j \partial x_j} \right]$$  \hspace{1cm} (7.6)

$$\left( \frac{\partial^2 P}{\partial x_i \partial x_j} \right)^{n+1} = \rho \frac{\Delta t}{\partial x_i} \left( \frac{\partial \tilde{u}_i}{\partial x_i} \right)^{n+1}$$  \hspace{1cm} (7.7)

$$u^{n+1}_i = \tilde{u}^n_i - \frac{\Delta t}{\rho} \left( \frac{\partial P}{\partial x_i} \right)^{n+1}$$  \hspace{1cm} (7.8)

Next the equations are converted to LSWF with the MLS shape functions applied to approximate the spatial derivatives.

$$\int_{\Omega} \int_{\Omega} w \phi [\delta \rho]^{n+1} d\Omega = \int_{\Omega} w \phi \delta \rho^n d\Omega - \Delta t \int_{\Omega} w (u^n \phi_x \delta \rho + v^n \phi_y \delta \rho)^n d\Omega$$  \hspace{1cm} (7.9)

$$\int_{\Omega} \int_{\Omega} \phi w [\tilde{u}^{n+1}] d\Omega = \int_{\Omega} w \phi u^n d\Omega + \frac{\Delta t}{\rho} \int_{\Omega} w \phi y (\phi_x v + \phi_y u)^n d\Omega$$  \hspace{1cm} (7.10)

$$\int_{\Omega} \int_{\Omega} \phi w [\tilde{v}^{n+1}] d\Omega = \int_{\Omega} w \phi v^n d\Omega + \frac{\Delta t}{\rho} \int_{\Omega} w 2\phi_y \mu (\phi_y v)^n d\Omega$$  \hspace{1cm} (7.11)

$$\int_{\Omega} \int_{\Omega} (\phi_{xx} + \phi_{yy}) [P]^{n+1} d\Omega = \int_{\Omega} \frac{\rho}{\Delta t} (\phi_x \tilde{u} + \phi_y \tilde{v})^{n+1} d\Omega$$  \hspace{1cm} (7.13)
\begin{align*}
\int_{\Omega} w \phi [u^{n+1}] d\Omega &= \int_{\Omega} w \phi \tilde{u}^{n+1} d\Omega - \frac{\Delta t}{\rho} \int_{\Omega} w \phi_x P^{n+1} d\Omega \quad (7.14) \\
\int_{\Omega} w \phi [v^{n+1}] d\Omega &= \int_{\Omega} w \phi \tilde{v}^{n+1} d\Omega - \frac{\Delta t}{\rho} \int_{\Omega} w \phi_y P^{n+1} d\Omega \quad (7.15)
\end{align*}

### 7.1.1 Boundary Conditions

Physically, RTI consists of many fingers or spikes of high density fluid moving through an interface. Numerically, the domain model is usually one or one-half of a finger. In this work the half width domain was used. The $x$ domain extends from 0.0 to 0.5 while the $y$ domain extends from -0.5 to 0.5. This domain is truncated in the $y$ direction to allow a quicker, higher fidelity solution while the application is being solved by MLPG for the first time. Figure 7.2 shows the domain dimensions and the associated boundary conditions for each surface. The vertical surfaces in both of these cases are modeled as zero gradient. In actuality, the $u$ velocity component can be treated as a fixed value of zero since fluid cannot flow through the symmetry surface. The top and bottom surfaces are treated as no slip, so both velocity components are zero on these surfaces. The pressure and density conditions for the horizontal walls is a zero gradient condition as well. A reference pressure and density are also required. These were selected near the upper right corner since little activity is expected in this region for the time period of interest.

![Diagram of RTI domain dimensions and boundary conditions.](image)

Figure 7.2: Diagram of RTI domain dimensions and boundary conditions.
The Dirchlet conditions were enforced using the Direct Interpolation method. The Neumann conditions were enforced using the new SFIM boundary condition. The integral method was attempted but yielded inaccuracy where the interface contacts the wall and the solution became unstable. SFIM was applied as a two step process. First as a part of the linear system creation and again as a post-processing step before the values were used by the next CBS computational step.

7.1.2 Interface

Following the work of He et al. [49] the initial interface between the two densities is defined by Eq. 7.16. In this equation the disturbance amplitude is \( D \). An amplitude of 0.1 is used here. Figure 7.1(b) shows the interface implemented in a half space domain.

\[ y_{\text{disturbance}} = -D \sin(2\pi(x - 0.25)) \] (7.16)

Frequently the process of applying an analytical equation to a gridded system introduces error, however in this case the error was much more severe than expected. The first implementation approach was to use the interface as a cutoff value. That is to assign any node that was classified as under the interface by the evaluation of Eq. 7.16 with the lower density value. Nodes with coordinates that indicated they were above or on the interface were initially assigned the higher density. Figure 7.3 shows a comparisons of the analytical interface expression and the generated interface on the MLPG nodes.

The interface as assigned to the nodes is not smooth. This is expected as it is being mapped to a grid. As the node spacing is decreased the interface becomes smoother. However, the runtimes would be negatively impacted. The critical issue is that the interface as implemented is inconsistent with the boundary conditions on the vertical surfaces. The specified gradient for density using the cutoff approach yields a sharp gradient at the surface. The solution is to add additional logic that applies the interface to the nearer of the nodes that bracket the specified position. This yields a zero density gradient at both boundaries, as seen in the contour plots in Figure 7.1(b), which is consistent with the boundary condition.

7.2 Implementation

For this work a uniform nodal arrangement was used. Unlike the lid-driven cavity example the areas of highest interest occur in the interior of the domain and are moving with time. Therefore increasing nodal density near the walls is not a viable way to minimize the total number of nodes needed for the computation. There are 31 nodes in the \( x \) direction and 61 nodes in the \( y \) direction. This yields equal spacing in each Cartesian directions.
The MLPG implementation used a quadratic basis function within MLS and a quartic spline function for both the test and trial function. The radius factor for the support domain was 0.5 and the radius factor for the quadrature domain was 2.01. These factors are smaller than those used in the lid driven cavity case because the nodes are uniformly spaced. Domain integrals were performed using Gaussian quadrature with 6 points in each Cartesian direction. A time step of 0.0025 was applied.

The solution procedure is a four step process. The first step computes the updated $\delta \rho$ via the linear system and then updates the derived quantities, $\rho$ and $\mu$. The remaining steps mimic those in the lid-driven cavity case: intermediate velocities, pressure, and velocity corrections. Since the SFIM boundary condition is applied in each of the four steps, the update or post-processing function is used after each computational step.

The reuse of matrices was again leveraged to minimize runtime. One linear system was used for each state variable instead of creating a system for each computational step. The matrices for the velocity components and density in each step are identical and do not change with time. Therefore a single $K$ matrix was created before the integration loop and used for the linear systems. The second $K$ matrix was required for the pressure step as the second derivative in both $x$ and $y$ appear on the left hand side of the equation. This matrix does not change with time and was created outside the integration loop as well. A total of 6 unique $\vec{f}$ vectors exist and required population during each iteration.

The problem was scaled following He et al. [49] using the channel width ($W$) as the charac-
teristic length. An Atwood number of 0.5 with a density ratio of 3 (non-dimensional densities are \( \rho_H = 3 \) and \( \rho_L = 1 \)). The Reynolds number is 256 and is imposed through a constant kinematic viscosity (\( \nu \)) of 0.0015625. Gravity acts in the negative \( y \) direction and has a non-dimensional magnitude of 0.0016. The time results are scaled by the quantity \( \sqrt{W/g} \).

\[
At = \frac{(\rho_H - \rho_L)}{(\rho_H + \rho_L)} \quad (7.17)
\]

\[
Re = \frac{g \sqrt{W g}}{\nu} \quad (7.18)
\]

### 7.3 Results

The RTI results are characterized by the features in the density field. The solution begins with vertical velocities initiated by the gravity force acting on \( \delta \rho \). Through the pressure equation the momentum is converted to horizontal velocity. Figure 7.4 shows the evolution of the density field.

![Figure 7.4: Density contours for the MLPG solution of RTI various time steps.](image)

The characteristic shape of the falling fluid column is captured beginning at a time of 1.5.
The lower surface of the column begins spreading, as would be expected even if the lower no-slip surface was farther away. The edge of the rounded portion begins to show recirculation at time 2.0. If the higher order terms are included and the nodal density is fine enough, very small scale features are observed in this rolled up region. In our results the wall is encountered and the column of fluid transfers momentum from vertical velocity to horizontal velocity. The rounded edge shape is lost but the vortical flow field is initiated.

As mentioned above the evolution of the density field is driven by the velocity. Figure 7.5 shows the velocity components and density at a time of 1.0, early in the simulation. The $v$ component shows the pocket for falling fluid on the right boundary and the rising pocket along the left boundary. These result from the gravity force acting on the $\delta \rho$ disturbance amplitude specified as the initial condition. Notice that the center of the field is stationary as the prescribed $\delta \rho$ was zero in this region. These $u$ velocity components are reacting to the $v$ components and resulting in the creation of a circular flow pattern around the interface. This pattern is shown in Figure 7.6 where select velocity streamlines are overlaid on the density contour. At this time the spike and bubble are moving along the boundaries but there is no observable change to the interior density field.

![Figure 7.5: RTI $u$ velocity component (a), $v$ velocity component (b), and density contours (c) at a time of 1.0.](image)

Figure 7.7 shows the system at a time of 2.0. For the shortened domain used in this work this state most closely resembles the standard RTI solution. The rounded shape of the downward moving column is visible and some small scale features are indicated in the rolled up edge. The velocity components have maintained the circular pattern discussed for the previous time. However the features encompass larger regions of the domain as the interface has developed.
Figure 7.6: With Streamlines.

Figure 7.7: RTI $u$ velocity component (a), $v$ velocity component (b), and density contours (c) at a time of 2.0.
7.3.1 Growth Rate

The growth rate refers to the vertical movement of the peak of each fluid. In the half width version being used in this work that corresponds to the boundaries. The fluid interface along each vertical wall is plotted in Figure 7.8. The bubble refers to the rising lower density fluid along the left boundary. The spike refers to the descending more dense fluid long the right boundary. The plot was gathered manually by identifying the location of the the interface, defined as 2.99 or greater. The data collection interval was every 1000 iterations which corresponds to 0.1 seconds.

![Figure 7.8: Time evolution of the density interface position on the vertical surfaces in MLPG results of RTI.](image)

Similar growth rate plots appear in other RTI publications including He et al [49]. Other growth plot include the bubble and spike velocities as a function of time. This work is not directly comparable to the published plots since the domain was of shorter length than what is standardly used. Due to the modified length, the interaction with the no-slip surface occurs much earlier in the calculation and slows the displacement of the fluids. It is worth noting that the general behavior is correct. At early times the interface is not changing shown by initial horizontal slope of the lines. Starting near a time of 0.5 the bubble rises (positive growth rate) and the spike falls (negative growth rate). The of a constant initial time followed by smooth slopes of opposite sign but similar magnitude is observed. The slopes are similar in magnitude.
Note that there is no reversal in flow direction when the no-slip surfaces are encountered. Instead as observed in Figure 7.4, the vertical fluid motion is translated into horizontal motion and the flow becomes vortical.

7.4 Conclusions

The RTI problem is more complex than the lid-driven cavity due to the introduction of variable density and the gravity body force. The computational process consists of four linear systems. Node clustering at the boundaries is not feasible due to the need to resolve the features in the center of the domain. In this work RTI of two immiscible fluids was successfully modeled. A standard MLPG with addition of SFIM results in a smooth solution with good qualitative agreement. The characteristic shape of the interface is observed as the more dense fluid displaces the less dense fluid. The growth of the rounded nose of the dense fluid column with some smaller scale mixing at its edges is captured. The growth rate is smooth and symmetric. This MLPG calculation successfully illustrates the ability of MLPG with SFIM to model fluid interfaces, providing a foundation for more complex phenomena such as mixing.
Chapter 8

Software Design

Computer codes created during the development of a numerical method or the application of a method to a new problem or domain are often called research or prototype codes. This term frequently refers to a code that has evolved following the path of the research and is often very customized to the specific problem and numerical method. When a new problem needs to be solved with the same computational approach, the code is often copied in its entirety and customized again for the new problem. If a bug is found in one version of the code, the researcher must remember to make the change in all other versions of the code. Functionality that is intended to be temporary as a problem is attempted or debugged often remain indefinitely. This decreases efficiency as unnecessary computations are performed in future iterations of the code. Frequently there are sections of code that are commented out, or never executed. Code developed in this manner is often hard for other researchers to learn and reuse.

One goal of this research effort was to design an MLPG implementation using modern, accepted computer science practices. This research effort aimed to create readable, reusable, object-oriented code that can be used by other researchers for various problems of interest. The design pattern selected for the algorithm execution was the Template Method, a description with examples in multiple programming languages can be found on the SourceMaking website [50]. This design approach allows the components of the algorithm to be specified once as an abstract class. The details of each step are then specified by subclasses or concrete classes. Polymorphism allows multiple versions of the subclasses to solve different physical problems or try different computational approaches. This is discussed in greater detail in Section 8.1 and Section 8.2 below.

The basic concepts of inheritance and polymorphism are used to make objects that are reusable and minimize duplicate coding. Most data in this implementation is contained within instances of the node class which is made up of other objects including Point and Dimension objects. Data objects for quadrature and MLS data are also discussed (Section 8.3).
The final sections focus on runtime profiling of this MLPG implementation. Analysis of runtime in each computational function at different problem sizes is also included. A discussion of possible optimizations and parallelization approaches to improve efficiency and support larger sized problems concludes the section.

8.1 MLPGProblem Class (Template Method Design)

The MLPGProblem class is the template class for the execution of a MLPG algorithm. This class contains the functions and data structures needed for the execution. The execution path is discussed in Section 8.2. The class contains the critical computational functions for populating the $K$ matrix and $F$ vector, as well as helper functions, such as one for finding the node closest to a location. Debugging capabilities can be activated in this class as well to print intermediate information. The class also specifies the needed data objects including the vector of nodes ($DomainNodes$), the $K$ matrix, $F$ vector, and $X$ vector. Problem specific variables, such as Reynolds number, and computational options, such as the MLS basis function order, are also contained within the class. Figure 8.1 provides an overview of the class, it does not include the exhaustive list of data objects and functions. In this implementation, a single instance of the

![Figure 8.1: MLPGProblem class overview.](image-url)
MLPGProblem class, called ProblemObj, is used during the execution of the algorithm. The declaration and construction of the ProblemObj is defined using the desired subclass. Recall that each subclass corresponds to a specific problem and a set of computational functions. ProblemObj is then used generically to execute the necessary commands. Switching from one problem to another is as simple as changing this single line. An example of this is system is:

```cpp
LidDriven* ProblemObj = new LidDriven();
ProblemObj->CreateGrid();
ProblemObj->time += ProblemObj->dt;
```

Functions declared as virtual in the template class must be implemented in the subclasses. Functions without this modifier can be overridden within the subclass but generic versions still are available within the template class. Implementation of virtual functions from the template class must exist within the subclass. However the implementation may be blank (i.e. `{}`). A simplified declaration of the MLPGProblem class is shown in Appendix A.1. A sample subclass declaration and constructor for the square lid-driven cavity problem is provided as well (Appendix A.2.1 and Appendix A.2.2).

### 8.2 Execution Path

The execution path for this MLPG implementation is divided into four sections. The first, referred to as Step 0, is executed outside the iteration loop. The others, labeled Step 1, Step 2, and Step 3, are executed inside the iteration loop. This concept is shown in the psuedocode below. The boolean variables bStep0, bStep1, bStep2, and bStep3 are declared in MLPGProblem and are set by the subclass. The booleans control the problem specific computations performed in each part of the execution path.

```cpp
if(bStep0)
    for each node in the domain
        ConstructMatrices();

for each iteration{
    if(bStep1)
        for each node in the domain
            ConstructMatrices();
    if(bStep2)
        for each node in the domain
            ConstructMatrices();
    if(bStep3)
        for each node in the domain
            ConstructMatrices();
```
In each step the `ConstructMatrices` function is called. These are virtual functions and must have implementation within the subclasses. The `ConstructMatrices` function is shown in pseudocode:

```c
ConstructMatrices (){
    if (Step0)
        Step0K ();
        Step0F ();
    if (Step1)
        Step1K ();
        Step1F ();
    if (Step2)
        Step2K ();
        Step2F ();
    if (Step3)
        Step3K ();
        Step3F ();
}
```

Often one or more of the functions (`Step3K()`, etc.) is empty. Computational efficiency is reduced if the step booleans are used correctly to skip the node loop when implementations for both the K and F functions for the step are empty. In the case of the square lid driven cavity problem with stationary nodes, `Step0K()` is used to specify the needed K matrices. The content of these matrices does not change during the execution of the problem and are calculated outside the iteration loop. Likewise `Step1K()`, `Step2K()`, and `Step3K()` do not require any code. The F vector requires calculations using data from the previous steps in the iteration so `Step0F()` is empty but `Step1F()`, `Step2F()`, and `Step3F()` contain computations.

The node loops are kept in the execution loop instead of in the problem specific functions. This reduces code duplication and localizes modifications for new functionality, such as the introduction of ghost cells for periodic boundary conditions. This also eases implementation and ensures consistency between the execution of different problem types. In addition to populating the matrices for each step, the system of equations is solved, and the state variables for each node are updated. Optional functions like printing the matrices can be invoked as well. The entire basic execution loop is provided in Appendix A.3. This execution path is customized somewhat to the three step CBS system used in this MLPG implementation.
8.3 Data Classes

In research code, the data evolves as the code is developed. There is no way to fully predict all the data that will be needed and specify the optimal organization. Therefore examining the data at regular intervals for inefficiencies, duplications, and opportunities for object creation is a necessary step. Although it takes time and effort to create new objects or extend existing objects, it keeps the code readable and reusable and can improve efficiency. The objects presented here are in no way the only options for MLPG data management, and are the results of the evolution of this specific implementation.

8.3.1 Node Class

The first Node class of this implementation included the positional information, state variables, boundary condition flags, and helper functions. During development, this dataset grew to include versions of the state variables and positions at three times. Additional boundary condition information to support periodic conditions were also incorporated. The node class was used not only for true MLPG nodes, but also for quadrature point locations, intermediate locations within the MLS process, and other purposes. In many cases these additional usages needed only variables for \(x, y,\) and \(z\). Therefore unnecessary work was performed to create and destroy these large objects when only a fraction of their functionality was being leveraged.

To reduce runtime and memory footprint a Point class was created to hold only the positional data. Node was modified to contain multiple instances of Point. These member objects hold the locations at different points in time, etc. The three components of the Point class can also be used to hold velocity data which has \(x, y,\) and \(z\) components as well. The introduction of this class reduced the number of variables, greatly improving readability, and reduced runtime.

8.3.2 Quadrature Class

The Quadrature class was written so that the number of quadrature points and the methodology to place them on new geometries could be extended easily. The base class is called Quadrature and is similar to the ProblemObj discussed above. The specifics of the quadrature object can be modified at construction and then the QuadObj can be used generically.

Quadrature contains the constants and weights specified for Gaussian quadrature, these values are assigned within the code (WeightData and LocationsData). The locations and weights for the specified geometry is computed and stored in the vectors Weights and Locations. The Quadrature class is shown below.
Multiple numbers of quadrature points were required during this work. The code below shows the declaration of the different classes and their inheritance from the Quadrature. In this design the classes specify the number of points only.

```cpp
class Quadrature
{
  public:
    Quadrature();
    ~Quadrature();

    double LeadingCoefficient;
    std::vector<double> Weights;
    std::vector<Point*> Locations;

  protected:
    std::vector<double> WeightsData;
    std::vector<double> LocationsData;

  private:
};
```

Each of the quadrature classes above have multiple constructors that allow the specification of the area over which the quadrature will be performed. The code below shows the three constructors in the 6 point quadrature class. The first is the most basic, it places the points along a straight line in either the x or y direction.

```cpp
class Quadrature_6Pt: public Quadrature
{
  public:
    // creates a 1D along either x or y layout
    Quadrature_6Pt(double SecMin, double SecMax,
                   bool XDir, double ConstPosition);

    // creates either a 2D layout or a 1D along the
Once the object is declared and the constructor is called, the object is used most generically by utilizing the vector lengths to control looping. This ensures that the constructor can be replaced with an alternate, using more or less quadrature points, without making any other modifications to the code (see example below). These classes can be modified for other geometries or extended to three dimensions without disrupting existing code.

```cpp
Quadrature* QuadObj = new Quadrature_6Pt(Point1, Point2, false);

for (unsigned int iQPt=0; iQPt < QuadObj->Locations.size(); iQPt++) {
    Point* QDPoint=QuadObj->Locations[iQPt];
    Value= QuadObj->LeadingCoefficient * QuadObj->Weights[iQPt]
        * Func(QDPoint->x, QDPoint->y);
}
delete QuadObj;
```

### 8.3.3 Shape Function Data Class

The MLS calculation is performed in a single standalone function but uses an object to exchange data with the calling function. The `ShapeFunction` data class declaration is:

```cpp
class ShapeFunction
{
    public:
        ShapeFunctions();
        virtual ~ShapeFunctions();
        void Reset();

        int iOrder;
        Dimensions* FirstDerivative;
        Dimensions* SecondDerivative;
}```
The inputs are the dimensional order of the calculation and specification of which derivatives are required. The shape functions and derivatives are stored in vectors which are order dependent containers. In the MLPG implementation the shape function values are used with the associated node. This code relies on the order of objects in the vector. So the the same vector must be passed into the MLS calculation and used when applying the resulting shape functions, i.e. $\phi(i)u(i)$. This is not the most generic and reusable way to code the MLS algorithm. However, the approach achieves the smallest memory footprint as the node data, locations, index etc, are not stored again with the shape functions. An example of the MLS data usage follows.

```cpp
ShapeFunctions* ShapeFunctionData = new ShapeFunctions();
ShapeFunctionData->FirstDerivative->x = true;
ShapeFunctionData->FirstDerivative->y = false;
ShapeFunctionData->SecondDerivative->x = true;
ShapeFunctionData->SecondDerivative->y = false;
ShapeFunctionData->iOrder = 2;
CalcMLSShapeFunctions_Centered(ProbDimObj, Domain, PointOfInterest,
    Radius, ShapeFunctionData, DebugFile);

for(int iDomPt=0; iDomPt < Domain.size(); iDomPt++){
    Point* Pt = Domain[iDomPt];
    Value += ShapeFunctionData->Phi[iDomPt] * Pt->x();
}
delete ShapeFunctionData;
```
8.3.4 Domain Class

The creation of the SFIM boundary condition necessitated the creation of a new class. This class holds data about the domain such as shape, number of nodes, extents, etc. It also has functions for determining containment of a point within the domain and creation of the grid given the domain attributes.

The current Domain data class declaration is:

```cpp
class Domain
{
    public:
        Domain();
        virtual ~Domain();

        int DomainShape; // flag used to direct computation
        enum Shape{RECTANGLE=0, CIRCLE=1, TRIANGLE=2, , PLATE_WITH_HOLE=4};

        bool InteriorToDomainCheck(Point* TrialPoint);
        void CreateGrid(Point* TrialPoint);

        double XMinAtGivenY(double Y);
        double XMaxAtGivenY(double Y);
        double YMinGivenX(double X);
        double XMinGivenY(double X);

        int iNumX;
        int iNumY;
        double LengthY;
        double LengthX;

        int NumNodes;

        double dXMin;
        double dXMax;
        double dYMin;
        double dYMax;

        // Triangle Specific
        double TriangleDepth;
        double Triangle_Slope;
        double Triangle_B;
        bool RightTriangle;
};
```
This is a good example of a class in development. When only square or rectangular domains were considered, dimensions and containment were easily described. With each added domain shape additional parameters are added to the class that are not needed or used by other shapes. This makes it difficult for a new user of the code to understand what they must specify to correctly characterize their domain. The next step for this class would be to create classes for each shape that inherit from the domain class. These individual classes would contain the parameters specific to their shape. This means that adding a new domain shape entails only defining a new class with parameters, containment functions, and grid creation functions. The alternative would require modifying every instance that uses the shape flag is used in the entire code base. The new class could also include interfaces to external tools to perform the containment functions.

### 8.4 Run Time

The run time of this MLPG implementation was analyzed using the lid-driven square cavity problem. This problem was selected because it is transient and utilizes the three step CBS solution process. Although the nodes are stationary, the execution was performed as though they were moving. This means that neighboring nodes must be identified and the $K$ matrices must be recalculated within the iteration loop. All debugging and output functions were disabled for these tests.

The standard C++ function `clock` from the ctime header was used for the timing. An example of the timing implementation is shown below. Timers were inserted around various functions in the code and written to file after the iteration time was calculated.

```cpp
double Timer1=0;
for(int i=0; i < NumIterations; i++){
    clock_t StartTime = clock();
    ConstructKMatrix();
    Timer1 += (clock() - StartTime) / (double)CLOCKS_PER_SEC;
}
```
Three domain discretizations were considered: 961 nodes (labeled 31 as this is the number along a direction), 3721 nodes (labeled 61), 8281 nodes (labeled 91). Fifty iterations were performed at each discretization level and the results were averaged. The table below shows the averaged iteration time for each discretization level.

Table 8.1: Average iteration runtime in seconds for each of three domain discretizations of the square lid-driven cavity problem.

<table>
<thead>
<tr>
<th>Discretization</th>
<th>Avg. Iteration Time (sec)</th>
<th>Avg. Iteration Time per node (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>9.26</td>
<td>0.01</td>
</tr>
<tr>
<td>61</td>
<td>312.72</td>
<td>0.084</td>
</tr>
<tr>
<td>91</td>
<td>3176.21</td>
<td>0.384</td>
</tr>
</tbody>
</table>

As expected the more nodes included in the analysis the more time required to complete an iteration. The runtimes were post-processed so that the time in each CBS step can be compared. Figure 8.2 presents the percentage of iteration time spent in each step. The percentage of time per iteration is similar. Step 2 is consistently quicker. This can be explained by the matrix system being smaller, as only one state variable is being solved for instead of the two used in steps 1 and 3. Step 3 may be slightly shorter than step 1 as the $K$ matrix is reused and so a few lines of computations to populate the matrix are omitted. However all quadrature, MLS, and linear system solving procedures are still required since the $\vec{f}$ vector is dependent on Step 2 data.

The runtimes were then processed to show the percentage of iteration time spent in the main computational functions: MLS, Subdomain Identification, and Linear System Solution. The time in the three computational areas presented accounts for more than 99.8% of the iteration time. Time spent performing the creation and destruction of objects, MLS computations, quadrature point assignment, and mathematical operations the $K$ matrix and $\vec{f}$ vector population are insignificant. This observation is true for each CBS step. The MLS time is the summation of the the time spent in all MLS computations in the iteration. These results are shown in Figure 8.3.

This data shows that the time spent solving the linear system is the largest percentage. Additionally as the problem size increases the system solution time percentage increases. This is a positive result as it shows that the solution process, data flow, looping, etc, which are unique to MLPG are not using the majority of the time. A common operation, linear system solution, with many available libraries and parallelization options is the culprit. This functionality is used only three times per iteration so this is not a case of calling a function many many times.
Figure 8.2: Percentage of average iteration runtime in each CBS solution step.

Figure 8.3: Percentage of average iteration runtime in the primary computational functions.
resulting in long run times. The MLPG linear system does not lend itself to many of the efficient approaches that can be applied to systems with a sparse or banded matrix but options do exist. The second most time consuming functional area is the identification of the subdomain. This is an example of a function needing to be efficient due to the large number of times it is called. Subdomains are identified for each quadrature point (usually 36 quadrature point per node) during each CBS step. Currently this function is a loop over the vector of domain nodes. Within the loop the distance between the quadrature point and the loop node is computed and compared to a cut off value. A vector of nodes passing the distance check is returned to the calling function. No objects are created or destroyed and a single loop with limited branching is executed. Future attempts to optimize this functionality would include the move from a vector to a tree structure, such as K-D tree or R-Tree, for the storage of nodes. The caveat is that for moving nodes the tree, which is based on spatial data, would have to be regenerated for each iteration which will reduce the realized runtime improvements. For stationary nodes a large improvement in runtime is expected.

8.5 Conclusions

The MLPG code created in this research to demonstrate the viability of the MLPG methodology for fluid dynamics interface problems was written in object oriented C++ following the Template Method design pattern. The implementation presented here uses modern computer science approaches to code organization and reuse. The code easily extends to new applications as needed during the course of the research.

The most published criticism of the MLPG methodology is the long runtimes for large, complex problems. MLPG is characterized by freedom from predefined connectivity between nodes, the use of Gaussian quadrature and MLS shape functions to solve the weak form of the governing equations. The results of this analysis show that the MLPG methodology is not inherently inefficient. The bulk of runtime is spent in the linear system solver. Linear system solution is a well understood process with many existing computational options for optimization and parallelization. By leveraging hardware advances such the use of graphics processing units (GPUs) for scientific computing and ever larger and more efficient traditional computer clusters, MLPG runtime should not be an obstacle to future success of the methodology.
Chapter 9
Conclusions and Recommendations for Future Work

The MLPG method provides an alternative to conventional classes of methods such as Finite Element and Finite Difference. The most promising aspect of this method is absence of node connectivity or prescribed spacings required during the discretization of the domain. This flexibility reduces the time spent creating and refining grids before calculations, or during calculation, as required due to large deflections, etc. Although the core methodology supports this idea, many numerical tools within MLPG must be customized to handle the complex or changing domains that MLPG was created to support. These include free surface, immersed boundaries, nodes that follow features or gradient. In this work, the Shape Function Interpolation Method, a new treatment for Neumann boundary conditions, was introduced. The SFIM method uses the MLS shape functions to identify the nodal boundary value that satisfies the prescribed gradient boundary condition. Previous methodologies required integration along the surface via line integrals for two dimensional problems or surface integrals for three dimensional problems. The process of computing quadrature point locations along the boundaries was customized for the different geometries. Additionally, the test function and trial function approximation (MLS) relies upon Cartesian distance components and relationships. A second capability for computing a normal vector for a node based upon the other boundary nodes in the identified region was created to compliment the boundary condition. This feature allows the weight function and MLS approximation to consider a Cartesian environment using a group of rotated nodes. These two additions minimize the phenomena or geometry specific coding required to apply MLPG to wide range of complex applications.

MLPG’s first publications feature solid mechanic benchmark applications. Solid mechanics application still dominate the literature for this relatively new method. This is most likely due to the similarities between Finite Element methods and MLPG. The use of the weak form
of the governing equations is much less common in fluids research. For this and other reason
the fluids community has been slower to explore the method. In this work, common fluids
benchmark problems are presented to validate the implementation of the MLPG methodology
and to validate the novel SFIM boundary condition. The body of work was expanded through
the successful modeling of the Rayleigh-Taylor Instability. This application requires a four-step
solution process of the variable density two-dimensional Navier-Stokes equations in primitive
variable form. RTI is a immiscible fluid interface problem. A more dense fluid is suspended above
a less dense fluid and gravity drives the resulting fluid motion. RTI is characterized the shape
that results as the heavy fluid moves downward. In this work RTI was modeled successfully
with MLPG using the new SFIM boundary condition. The fluid motion was monitored via a
growth rate for both the spike of more dense fluid and the rising bubble of less dense fluid. The
growth rate was smooth and compares well qualitatively with existing literature. The contours
of density and velocity are smooth with no boundary or numerical artifacts. The modeling of
the Rayleigh-Taylor Instability introduces a new class of fluids applications that can be modeled
using MLPG.

Research software frequently has a short life as reusability for other purposes and by
other researchers is not adequately plan for. The implementation of MLPG presented here was
written in object oriented C++ with a Template Method Design pattern. The data is stored
in objects that can be modified or used for inheritance to new objects. This is useful for both
new methodology that requires new data or for new applications requiring custom data. The
execution path is generalized so that modeling a new application does not prevent the execution
of previous problem types. Following design patterns and data modeling in this way reduces
the amount of copied or replicated code. For example the MLS implementation exists only
once. When the new rotated system was needed a function was created to do the necessary
rotation logic and then the original MLS function is called to perform the remainder of the
work. Additionally when the RTI case required more density definitions they were easily added
to the problem with default values. The other problem types did not have to be modified to
include these values nor did logic have to be to explicit ignore or utilize them.

9.1 Recommendations for Future Work

The research and development of the MLPG method specifically in the area of fluid dynamics
is on going. Areas for continued development exist within the base methodology and utilized
numerical tool. Efforts to reduce runtime through optimizations, parallelization, and the utili-
zation of more efficient linear system solvers are needed before MLPG will transition to tool
for real-world applications. Some specific research areas are discussed below.

All though the lack of sensitivity to nodal arrangement is a much touted attribute of MLPG,
algorithms for identifying subdomains more accurately are needed to fully support this capability. The nearest node computation is commonly done one of two ways. The first is to find the nearest node via distance formula. The distance to this node is then used as both the $x$ distance and $y$ distance for test functions. The alternate approach is to find the two nearest neighbors, one based on the distance in $y$ and the second based on the distance in $x$. This results in two distance components that are not necessarily equation for the later calculations. The radius of influence and quadrature radius multipliers rely on these distances to identify the needed subdomains. For cases with moving nodes, randomly spaced nodes, or complex domain shape spacing the nearest neighbor node methodology fails. In such cases nodes may be extremely close, or have a very small distance components even if the over all nearest node distance is reasonable. A variety of errors can result from this situation. One is that the some number of subdomains are of very small size. This causes incomplete coverage of the global domain. Another result is domains that have poor aspect ratios. These skewed domains result in inaccurate characterization of the local environment. This leads to poor solution accuracy and system instability.

The long runtimes for most MLPG implementations is the most limiting aspect of the technology. Optimization and parallelization of a numerical method is rarely a focus early in development. As the new methodology stabilizes and the focus moves toward new, larger, and more complex applications the need for efficiency is usually addressed. MLPG has multiple target areas for such research. The first is the the identification or creation of efficient, and possibly parallelized, linear system solvers. Solvers that allow piecewise execution to leverage constant or infrequently changed $K$ matrices is desirable. The second area is the node by node construction of the linear system. For applications that require the solution of the Navier-Stokes equations the linear system is constructed 3 or more times. For each computational step in each iteration every node goes through the processes of subdomain identification, quadrature via MLS at each quadrature point, and test function evaluation. For simple problems it may appear that much of this data could be computed and reused. However this is short sighted as the longer term vision for MLPG includes large numbers of nodes and moving nodes. The storage space required and memory time would negate the anticipated improvements. Since these computations use previous time data and are assigning values to the linear system data objects, not to the nodes themselves, concurrent execution is a more promising approach. The ability to execute these processes in parallel via multi-core CPUs, Graphical Processing Units (GPU), or other hardware configurations would improve the efficiency of the system.

The application of MLPG to a larger number of fluids problems as well as problems with more complex phenomena is required for this method to become a more accepted modeling tool in the research community. The successful simulation of RTI and lid driven cavity problems was shown in this work. Some logical next steps are the addition of mixing to interface or
multi-fluid type problems. The addition of turbulence models will allow a much larger class of fluids problems to be modeled as well.
REFERENCES


petrov-galerkin method with generalized moving least squares interpolations. *Computa-


interface problems with applications in electrostatic and elastostatic. *Computer Methods
in Applied Mechanics and Engineering*, 278:479–498, ???


Appendix A

MLPG Sample Implementation

This appendix provides the basic implementation approach used in this research for the execution of the MLPG solution procedure. The Template Method design pattern was applied with \texttt{MLPGProblem} being the abstract or template. Subclasses contain different implementations of the template. These implementations may reflect different physical problems or different computational approaches. In this Appendix the example subclass is the lid-driven cavity (\textit{LidDriven}).

A.1 \ MLPGProblem Class Definition

This class defines the functions and data objects needed to execute the MLPG algorithm. The virtual functions must be implemented in the subclasses even if the implementation does not contain any commands (i.e. \{\}).

\begin{verbatim}
  class MLPGProblem
  {
    public:
      MLPGProblem();
    virtual ~MLPGProblem();

    virtual void CreateGrid();
    virtual void ConstructMatrices();
    virtual void OutputSolution();
    virtual void UpdateStateVariables();
    virtual void SolveLinearSystem();
    virtual void Restart();
    virtual void Step0K();
    virtual void Step1K();
    virtual void Step2K();
  }
\end{verbatim}

103
virtual void Step3K();
virtual void Step0F();
virtual void Step1F();
virtual void Step2F();
virtual void Step3F();

Dimensions* ProbDimObj;
Point* UpperRightCorner;
Point* LowerLeftCorner;
double NodeConcentrationCoeff;

Domain * DomainObj;

int NumStateVars;
int NumStateVarsAlt;
int N;  // used to determine matrix size (NumNodes*NumStateVars)
int NAlt;

std::vector<Node*> DomainNodes;
double RadiusFactor;
double QuadRadiusFac;

// flags to describe problem execution
bool PeriodicInX;
bool PeriodicInY;
bool CBS_iterative;
bool Transient;
bool bStep0;
bool bStep1;
bool bStep2;
bool bStep3;

int iWeightFunction;
int iQuadPtsNum;
int iMLSOrder_Domain;
int iMLSOrder_Line;

int MaxIterations;
int IterationNumber;
int PrintFrequency;
bool bRestart;
int RestartIteration;
virtual void Restart() {};

double time;
double dt;

ofstream SimplePlotFile;
ofstream DebugFile;
ifstream RestartFile;

Matrix<double, Dynamic, Dynamic> K;
Matrix<double, Dynamic, Dynamic> KInv;
Matrix<double, Dynamic, Dynamic> F;
Matrix<double, Dynamic, Dynamic> X;

Matrix<double, Dynamic, Dynamic> K2;
Matrix<double, Dynamic, Dynamic> K2Inv;
Matrix<double, Dynamic, Dynamic> F2;
Matrix<double, Dynamic, Dynamic> X2;

Matrix<double, Dynamic, Dynamic> KAalt;
Matrix<double, Dynamic, Dynamic> KAaltInv;
Matrix<double, Dynamic, Dynamic> FAalt;
Matrix<double, Dynamic, Dynamic> XAalt;

void CreateSubdomain(std::vector<Node> InitialDomain,
                      std::vector<Node>& Subdomain,
                      Point* SolutionPt, Point* Radius,
                      Dimensions* DimObj);

Node* FindClosestPt(double x, double y, double z);
void ComputeNodeRadii(Node* NodeI);
void ComputeModifiedNodeLocation(Point* Original, Point* Modified);
void IdentifyNearestNeighbors(std::vector<Node> InitialDomain);
bool InteriorToDomainCheck(Point* TrialPoint);
void InitializeArrays(bool InitK, bool bAlt, int MatrixNumber);

void PrintK(bool bAlt, int MatrixNumber);
void PrintF(bool bAlt, int MatrixNumber);
void PrintX(bool bAlt, int MatrixNumber);

// values needed by functions during an iteration
Node* NodeI;
Point* Radius;
Point* QuadRadius;
WeightFunction* pWeights;

protected:
private:
}
A.2 MLPG Problem: Lid-Driven Cavity Problem

A.2.1 Lid-Driven Cavity Problem Declaration

This section contains the declaration of the LidDriven subclass. The LidDriven class includes virtual functions from the template class as well as problem specific functions and data objects.

class LidDriven : public MLPGProblem
{
    public:
        LidDriven();
        virtual ~LidDriven();

        void CreateGrid();
        void ConstructMatrices();
        void OutputSolution();
        void SimpleFormatNodeSol();
        void UpdateStateVariables();
        void Restart();

        void Step0K();
        void Step1K();
        void Step2K();
        void Step3K();

        void Step0F();
        void Step1F();
        void Step2F();
        void Step3F();

        bool debug;
        double Uavg();
        double Re;
        double Uinit;
        double Vinit;
        double Pinit;

        Quadrature* FQuadObj;
        Quadrature* KQuadObj;

        void OutputLineFiles();
        void InitExactData();
        bool bWriteExactLinesFile;

};
std::vector<Point> ExactPositions;
std::map<int, std::vector<double>> ExactUvelocity;
std::map<int, std::vector<double>> ExactVvelocity;
ofstream GnuLinePlotExact;

ofstream GnuPlot;
ofstream GnuLinePlot;
ofstream GnuPLinePlot;
ofstream GnuDerivLinePlot;

protected:
private:
};
A.2.2 Lid-Driven Cavity Problem Constructor

The *LidDriven* constructor assigns the specific values for the physical problem being modeled and the necessary computational options.

```cpp
LidDriven::LidDriven(){
    ProbDimObj->x = true;
    ProbDimObj->y = true;

    DomainObj->iNumX = 31;
    DomainObj->iNumY = 31;
    UpperRightCorner = new Point(1.0,1.0);
    LowerLeftCorner = new Point(0.0,0.0);

    dt = 0.0005;
    Re = 100.0;
    Uinit = 0.0;
    Vinit = 0.0;
    Pinit = 0.0;
    debug = false;
    bRestart = false;

    CBS_iterative = true;
    bStep0 = true;
    bStep1 = true;
    bStep2 = true;
    bStep3 = true;

    MaxIterations = 5000;
    RestartIteration = 0;
    IterationNumber = 0;
    PrintFrequency = 1;
    bWriteExactLinesFile = true;

    RadiusFactor = 0.75;
    QuadRadiusFac = 2.25;
    NodeConcentrationCoeff = 1.7;

    iWeightFunction = 4;
    iMLSOOrder_Domain = 2;
    iMLSOOrder_Line = 2;
}
DomainObj->NumNodes = DomainObj->iNumX*DomainObj->iNumY;
NumStateVars = 1;
NumStateVarsAlt = 1;

N = DomainObj->NumNodes*NumStateVars;
NAlt = DomainObj->NumNodes*NumStateVarsAlt;

K. resize (N,N);
F. resize (N,1);
X. resize (N,1);
F2. resize (N,1);
X2. resize (N,1);
K_Alt. resize (NAlt,NAlt);
F_Alt. resize (NAlt,1);
X_Alt. resize (NAlt,1);

for(int iNode=0; iNode < N; iNode++){
  for(int jNode=0; jNode < N; jNode++){
    K(iNode,jNode)=0.0;
  }
  X(iNode,0)=0.0;
  F(iNode,0)=0.0;
  X2(iNode,0)=0.0;
  F2(iNode,0)=0.0;
}

for(int iNode = 0; iNode < NAlt; iNode++){
  for(int jNode=0; jNode < NAlt; jNode++){
    K_Alt(iNode,jNode)=0.0;
  }
  X_Alt(iNode,0)=0.0;
  F_Alt(iNode,0)=0.0;
}
SimplePlotFile.open("LidDrivenCavitySimpleNode.txt", ios ::out);
InitExactData();
}
A.3 MLPG Driver Function

The following code execution process for the MLPGProblem class. This driver is used to execute the functions implemented by a subclasses of MLPGProblem. The subclass is specified at the declaration and construction of the ProblemObj.

```cpp
void MLPGDriver::MLPGMain()
{
    LidDriven* ProblemObj = new LidDriven();
    ProblemObj->CreateGrid();
    if(ProblemObj->bRestart) ProblemObj->Restart();
    ProblemObj->time += ProblemObj->dt;
    ProblemObj->IdentifyNearestNeighbors(ProblemObj->DomainNodes);

    for(int iPt=0; iPt < ProblemObj->DomainNodes.size(); iPt++)
    {
        ProblemObj->NodeI = ProblemObj->DomainNodes[iPt];
        if(ProblemObj->NodeI->GhostPt) continue;
        ProblemObj->ComputeNodeRadii(ProblemObj->NodeI);
        ProblemObj->ConstructMatrices();
    }

    for(ProblemObj->IterationNumber = ProblemObj->RestartIteration;
        ProblemObj->IterationNumber <= ProblemObj->MaxIterations;
        ProblemObj->IterationNumber++)
    {
        if(ProblemObj->bStep1)
        {
            ProblemObj->iStep = 1;
            for(int iPt=0; iPt < ProblemObj->DomainNodes.size(); iPt++)
            {
                ProblemObj->NodeI = ProblemObj->DomainNodes[iPt];
                if(ProblemObj->NodeI->GhostPt) continue;
                ProblemObj->ComputeNodeRadii(ProblemObj->NodeI);
                ProblemObj->ConstructMatrices();
            }
            ProblemObj->SolveLinearSystem();
            if(debug){
                ProblemObj->PrintK(false, 1);
                ProblemObj->PrintF(false, 1);
                ProblemObj->PrintX(false, 1);
            }
            ProblemObj->UpdateStateVariables();
        }
        ProblemObj->InitializeArrays(false, false, 1);
    }
```

111
} 
if (ProblemObj->bStep2) {
    ProblemObj->iStep = 2;
    for (int iPt = 0; iPt < ProblemObj->DomainNodes.size(); iPt++) {
        ProblemObj->NodeI = ProblemObj->DomainNodes[iPt];
        if (ProblemObj->NodeI->GhostPt) continue;
        ProblemObj->ComputeNodeRadii(ProblemObj->NodeI);
        ProblemObj->ConstructMatrices();
    }
    ProblemObj->SolveLinearSystem();
    if (debug) {
        ProblemObj->PrintK(true, 0);
        ProblemObj->PrintF(true, 0);
        ProblemObj->PrintX(true, 0);
    }
    ProblemObj->UpdateStateVariables();
    ProblemObj->InitializeArrays(false, true, 0);
}
if (ProblemObj->bStep3) {
    ProblemObj->iStep = 3;
    for (int iPt = 0; iPt < ProblemObj->DomainNodes.size(); iPt++) {
        ProblemObj->NodeI = ProblemObj->DomainNodes[iPt];
        if (ProblemObj->NodeI->GhostPt) continue;
        ProblemObj->ComputeNodeRadii(ProblemObj->NodeI);
        ProblemObj->ConstructMatrices();
    }
    ProblemObj->SolveLinearSystem();
    if (debug) {
        ProblemObj->PrintK(false, 2);
        ProblemObj->PrintF(false, 2);
        ProblemObj->PrintX(false, 2);
    }
    ProblemObj->UpdateStateVariables();
    ProblemObj->InitializeArrays(false, false, 2);
}
    ProblemObj->OutputSolution();
    ProblemObj->time += ProblemObj->dt;
}
delete ProblemObj;