

# **On Meshless Methods: A Novel Interpolatory Method and a GPU-Accelerated Implementation**

Maïen Hamed

Submitted in partial fulfillment of the requirements for the degree of Master of  
Science in Applied Mathematics in the Faculty of Science at the Nelson  
Mandela Metropolitan University

December, 2012

Supervisor: Prof J.W. Gonsalves

Co-Supervisor: C.J. Pretorius

# Contents

|                                                              |             |
|--------------------------------------------------------------|-------------|
| <b>Abstract</b>                                              | <b>vi</b>   |
| <b>Acknowledgements</b>                                      | <b>viii</b> |
| <b>Declaration</b>                                           | <b>ix</b>   |
| <b>1 Introduction</b>                                        | <b>1</b>    |
| 1.1 Model Problems . . . . .                                 | 3           |
| 1.1.1 Axially Loaded Elastic Bar . . . . .                   | 3           |
| 1.1.2 Thermal Equilibrium in a Plate . . . . .               | 7           |
| 1.1.3 A Plate Under Plane-Stress . . . . .                   | 10          |
| 1.2 Interpolation and Approximation Functions . . . . .      | 15          |
| <b>2 Literature Survey</b>                                   | <b>19</b>   |
| 2.1 Meshless and Extended Shape Functions . . . . .          | 20          |
| 2.1.1 Window Functions . . . . .                             | 20          |
| 2.1.2 Weighted Integral Approximations . . . . .             | 23          |
| 2.1.3 Moving Least-Squares Approximations . . . . .          | 24          |
| 2.1.4 Extrinsic Basis and Extended Finite Elements . . . . . | 29          |
| 2.2 Meshless Interpolants . . . . .                          | 31          |
| 2.3 GPU Computation . . . . .                                | 39          |
| <b>3 Moving Regularized Interpolation</b>                    | <b>41</b>   |
| 3.1 Parallel Distance Function . . . . .                     | 48          |

|          |                                                                               |           |
|----------|-------------------------------------------------------------------------------|-----------|
| 3.2      | Differentiation of Regularized Interpolation Method Shape Functions . . . . . | 50        |
| 3.3      | An Interpolation Example . . . . .                                            | 51        |
| 3.4      | Consistency and Multiple Stage Interpolation . . . . .                        | 54        |
| 3.5      | Solutions of Model Problems . . . . .                                         | 56        |
| 3.5.1    | Axially Loaded Elastic Bar . . . . .                                          | 56        |
| 3.5.2    | Plate in Thermal Equilibrium . . . . .                                        | 57        |
| 3.5.3    | Cantilever Beam in Plane Stress . . . . .                                     | 58        |
| <b>4</b> | <b>GPU-accelerated EFG Implementation</b>                                     | <b>63</b> |
| 4.1      | Data Structures . . . . .                                                     | 64        |
| 4.1.1    | Node Points . . . . .                                                         | 64        |
| 4.1.2    | Quadrature Points . . . . .                                                   | 64        |
| 4.1.3    | Contributions . . . . .                                                       | 67        |
| 4.1.4    | Interactions . . . . .                                                        | 70        |
| 4.2      | Computation Stages . . . . .                                                  | 71        |
| 4.2.1    | Preprocessing . . . . .                                                       | 72        |
| 4.2.2    | Computation of Moment Matrices . . . . .                                      | 74        |
| 4.2.3    | Solution of Moment Matrix Linear Systems . . . . .                            | 75        |
| 4.2.4    | Population of the Load Vector . . . . .                                       | 76        |
| 4.2.5    | Population of the Stiffness Matrix . . . . .                                  | 76        |
| 4.3      | Performance Results . . . . .                                                 | 77        |
| <b>5</b> | <b>Discussion and Future Work</b>                                             | <b>88</b> |
| 5.1      | Future Work . . . . .                                                         | 89        |
|          | <b>Bibliography</b>                                                           | <b>91</b> |
|          | <b>Appendices</b>                                                             | <b>96</b> |

|          |                                                                                   |            |
|----------|-----------------------------------------------------------------------------------|------------|
| <b>A</b> | <b>Some Concepts from Linear Algebra</b>                                          | <b>96</b>  |
| A.1      | Vector Space . . . . .                                                            | 96         |
| A.1.1    | Inner Product Spaces . . . . .                                                    | 97         |
| A.1.2    | Basis Vectors and Dimension of a Vector Space . . . . .                           | 97         |
| A.2      | Quadratic Forms . . . . .                                                         | 98         |
| A.2.1    | Positive Definiteness . . . . .                                                   | 98         |
| <b>B</b> | <b>Code</b>                                                                       | <b>100</b> |
| B.1      | Interpolation in 1D with the Moving Regularized Interpolation<br>Method . . . . . | 100        |
| B.2      | Model Problem: Axially Loaded Elastic Bar . . . . .                               | 105        |
| B.3      | Model Problem: Plate Under Thermal Equilibrium . . . . .                          | 109        |
| B.4      | Model Problem: Cantilever Beam Under Plane Stress . . . . .                       | 114        |
| B.5      | GPU-accelerated EFG Implementation . . . . .                                      | 121        |
| B.5.1    | Data Structures . . . . .                                                         | 121        |
| B.5.2    | Host Code . . . . .                                                               | 123        |

## List of Figures

|      |                                                              |    |
|------|--------------------------------------------------------------|----|
| 1.1  | Force Balance on a Cross Section . . . . .                   | 4  |
| 2.1  | Discontinuity in the Point Interpolation Method . . . . .    | 38 |
| 3.1  | Parallel Distance Function . . . . .                         | 50 |
| 3.2  | Moving Regularized Interpolation 1D Example . . . . .        | 54 |
| 3.3  | Window Function as Evaluated at $x = 0.42$ . . . . .         | 55 |
| 3.4  | Parallel Distance Function . . . . .                         | 55 |
| 3.5  | Displacement of Elastic Bar Under a Tensile Load . . . . .   | 58 |
| 3.6  | A Plate in Thermal Equilibrium . . . . .                     | 59 |
| 3.7  | Node Distribution for Thermal Model Problem . . . . .        | 59 |
| 3.8  | $y$ -component of the Displacement (Not to Scale) . . . . .  | 60 |
| 3.9  | $x$ -component of the Displacement (Not to Scale) . . . . .  | 61 |
| 3.10 | $y$ -component of the Displacement at Two Sections . . . . . | 61 |
| 3.11 | $x$ -component of the Displacement at Two Sections . . . . . | 62 |
| 3.12 | Node Distribution for Plane-Stress Model Problem . . . . .   | 62 |
| 4.1  | Total Computation Time Comparison . . . . .                  | 78 |
| 4.2  | Memory Management Time Comparison . . . . .                  | 79 |
| 4.3  | Data Transfer Time Comparison . . . . .                      | 80 |
| 4.4  | Node Processing Time Comparison . . . . .                    | 81 |
| 4.5  | Quadrature Structure Processing Time Comparison . . . . .    | 82 |
| 4.6  | Contribution Data Processing Time Comparison . . . . .       | 83 |
| 4.7  | Moment Matrices Processing Time Comparison . . . . .         | 84 |

|      |                                                         |    |
|------|---------------------------------------------------------|----|
| 4.8  | Correction Vectors Processing Time Comparison . . . . . | 85 |
| 4.9  | Load Vector Population Time Comparison . . . . .        | 86 |
| 4.10 | Stiffness Matrix Population Time Comparison . . . . .   | 87 |

## **Abstract**

Meshless methods have been developed to avoid the numerical burden imposed by meshing in the Finite Element Method. Such methods are especially attractive in problems that require repeated updates to the mesh, such as problems with discontinuities or large geometrical deformations. Although meshing is not required for solving problems with meshless methods, the use of meshless methods gives rise to different challenges. One of the main challenges associated with meshless methods is imposition of essential boundary conditions. If exact interpolants are used as shape functions in a meshless method, imposing essential boundary conditions can be done in the same way as the Finite Element Method.

Another attractive feature of meshless methods is that their use involves computations that are largely independent from one another. This makes them suitable for implementation to run on highly parallel computing systems. Highly parallel computing has become widely available with the introduction of software development tools that enable developing general-purpose programs that run on Graphics Processing Units.

In the current work, the Moving Regularized Interpolation method has been developed, which is a novel method of constructing meshless shape functions that achieve exact interpolation. The method is demonstrated in data interpolation and in partial differential equations.

In addition, an implementation of the Element-Free Galerkin method has been written to run on a Graphics Processing Unit. The implementation is described and its performance is compared to that of a similar implementation that does not make use of the Graphics Processing Unit.

## **Acknowledgements**

I am grateful to many individuals, without whose help and support this treatise would not have been possible.

I am deeply grateful to my supervisor, Prof John Gonsalves, for his guidance and support. Since my undergraduate days, he has taught me more than mathematics, its history, and its pleasures. I am thankful to him for always having been available for discussing my research, for reading this manuscript and providing valuable feedback, and for the valuable advices that he has given me whenever I needed guidance.

I am grateful to my co-supervisor, Christiaan Pretorius. I am thankful to him for being available to discuss the research, and for his diligent reading of this manuscript.

I am thankful to my parents and to my brothers and sisters for their love, support, and constant encouragement.

I thank my colleagues at eNtsa for their support on my work towards finishing this work, and I am thankful for the financial support that eNtsa has provided during my studies.

### **Declaration**

I, Maien Hamed, hereby declare that the work reported in this treatise is my own, and that the treatise has not previously been submitted in full or partial fulfillment of the requirements of another qualification.

.....

Author's Signature

Date: .....

# Chapter 1

## Introduction

The Finite Element Method (FEM) is an invaluable tool in solving partial differential equations of scientific and industrial interest. A major part of Finite Element Analysis, which is the practice of applying the Finite Element Method, is meshing of the geometrical domain of the partial differential equation under consideration. *Meshing* is the process of generating a mesh, a collection of subdomains that satisfy a number of geometrical requirements. This reliance on a mesh has long been a limitation of the FEM. Traditionally, meshing has been labour intensive; and though the process can be largely automated now, in many analyses it still requires laborious human intervention. These limitations are exacerbated in problems characterised by large geometrical deformations and problems with discontinuities.

Meshless methods, also called meshfree methods, have been devised to avoid these limitations imposed by reliance on a mesh. A distribution of points scattered in the problem domain is sufficient for applying these methods; meshing is not required. Consequently, these methods are attractive in problems with large deformations or discontinuities.

In most meshless methods, as in the FEM, it is necessary to evaluate and integrate a set of functions, which are called shape functions. The mesh in the FEM aids in these procedures. In many meshless methods, although the computational effort required for meshing is largely obviated, evaluation and integration of shape functions are computationally heavier than they are in the FEM. This added computational burden may be justified in problems that require remeshing and ones with evolving geometries, where projection of intermediate solutions onto the mesh may introduce numerical errors. Nevertheless, it is desirable in meshless method implementations that these computations be optimized. Fortunately, in most meshless methods these procedures involve a large collection of decoupled calculations, which makes them amenable to parallel computation.

Parallel implementations of meshless methods may be a suitable application for modern Graphics Processing Units (GPU's), which have been endowed with the capability to perform general purpose computations. These devices are readily available in systems ranging from personal computers to high performance computing clusters.

In addition to their evaluation having higher computational requirements, many meshless shape functions are not interpolants. This poses difficulties when imposing essential boundary conditions. As a result, interpolatory meshless shape functions are desirable.

This treatise presents the two main outcomes of the current work. First, it introduces the Moving Regularized Interpolation method, which is a novel method of constructing meshless shape functions that achieve exact interpolation. Second, it presents a GPU-accelerated meshless method implementation. In the next section, some model problems are presented. A survey of the literature of meshless methods follows in Chapter 2. Next, in Chapter 3, the Moving Regularized

Interpolation method is described and demonstrated in solutions to the model problems. A detailed explanation of a GPU-accelerated meshless method implementation is then provided in Chapter 4. Chapter 5 concludes this treatise with a discussion and suggestions for future work.

## 1.1 Model Problems

In the chapters to follow, model problems serve two purposes. First, they serve as a context in which to discuss the meshless methods in the literature. Second, they are used in testing the meshless method software implementations constructed in the current work.

### 1.1.1 Axially Loaded Elastic Bar

The model of an axially loaded elastic bar is one of the simplest representatives of an important class of problems studied in applied mathematics: problems of systems in equilibrium [1]. Characteristics of this class of problems are distilled in this one-dimensional model so that they can be studied in isolation from the complications specific to higher dimensional problems.

What is being modeled is an elastic bar undergoing deformation in reaction to a load to which it is subjected. By studying the problem in one dimension, the assumption is made that each cross section remains undeformed. Deformation of the bar is constituted by motion of cross sections, each of which is an infinitesimally thin slice of the bar centered at a given distance from its base. The bar is modeled as a continuum; a one-to-one mapping is made linking each

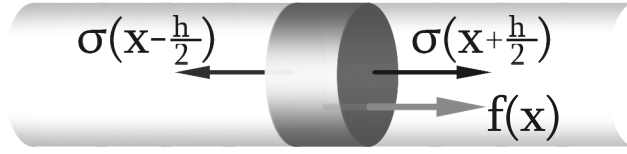


**Figure 1.1:** Force Balance on a Cross Section

cross section of the bar to a point in a one-dimensional continuum. Hence physical attributes of the bar, such as displacement, can be represented as functions whose domain is the subset of the real line representing the continuum.

An expression relating the deformation of the bar to the applied load is desired. Let  $u(x)$  denote the final displacement of the bar cross section linked to point  $x$ , and let  $f(x)$  be the external axial force density applied at the same point. At equilibrium, the externally applied force density is balanced by the internal forces in the bar. Figure 1.1 shows a free body diagram of an infinitesimally thin cross section represented by point  $x$ . As the figure shows, the force balance gives the equation:

$$f(x) + \frac{\sigma(x + \frac{h}{2}) - \sigma(x - \frac{h}{2})}{h} = 0$$

In the limit as  $h \rightarrow 0$ , this force balance equation reduces to:

$$-\frac{d\sigma}{dx} = f(x) \tag{1.1}$$

At this point, an expression for the internal forces  $\sigma$  is required. Internal forces arise in reaction to deformation. Therefore, a constitutive law, which is a model

relating deformations to the internal forces that they induce, is required. One of the simplest elastic constitutive laws is that of linear elasticity, in which the internal forces acting on a cross section of the bar are proportional to its elongation. Elongation is defined as the ratio between the change in length and the original length of a section of the bar. Elongation of a section of the bar whose thickness  $h$  approaches zero is given by:

$$\epsilon(x) = \lim_{h \rightarrow 0} \frac{u(x+h) - u(x)}{h} = \frac{du}{dx} \quad (1.2)$$

Substituting the elongation in the expression for a linear elastic constitutive law gives:

$$\sigma(x) = k(x)\epsilon(x) = k(x)\frac{du}{dx} \quad (1.3)$$

where  $k$  is governed by material properties and cross section geometry of the bar. When this constitutive law is substituted in the expression for the force balance, the result is the governing equation:

$$-\frac{d}{dx} \left( k(x) \frac{du}{dx} \right) = f(x) \quad (1.4)$$

Along with the boundary conditions imposed on the elastic bar, this governing equation describes the behaviour of the bar in reaction to any applied loads.

**Weighted Residual and Weak Formulations**

Multiplying both sides of the governing equation (1.4), by a test function  $v(x)$  and integrating over the domain  $\Omega$  of the problem yields:

$$-\int_{\Omega} v(x) \frac{d}{dx} \left( k(x) \frac{du}{dx} \right) dx = \int_{\Omega} v(x) f(x) dx \quad (1.5)$$

This equality must hold for any admissible weight function.

The weak formulation is obtained by distributing the derivative evenly between  $u$  and  $v$ . In this one-dimensional case, this is achieved by integration by parts, which gives:

$$\int_{\Omega} \frac{dv}{dx} k(x) \frac{du}{dx} dx - \left( v(x_1) k(x_1) \frac{du}{dx} \Big|_{x_1} - v(x_0) k(x_0) \frac{du}{dx} \Big|_{x_0} \right) = \int_{\Omega} v(x) f(x) dx \quad (1.6)$$

where  $x_0$  and  $x_1$  are respectively the left and right boundaries of the domain  $\Omega$ .

**Galerkin Formulation**

Given a set of global shape functions  $\{\Phi_i\}_{i=1}^n$ , a discretized approximation of the system described above may be obtained using Galerkin's method. To do that, the solution is assumed to be an element of a finite dimensional subspace spanned by the shape functions, and the test functions  $v$  are restricted to the

same subspace. Using these assumptions and the weak formulation in equation (1.6), a system of  $n$  equations in  $n$  variables can be constructed. Each of the  $n$  equations corresponds to using one of the shape functions as a test function, and the  $n$  variables correspond to the coefficients in the representation of the approximate solution  $u^h$  as a combination of the shape functions. This system of equations is represented by constructing the stiffness matrix  $K = [k_{ij}]$  and the load vector  $\mathbf{f}$  by the formulae:

$$k_{ij} = \int_{\Omega} \frac{d\Phi_i}{dx} k(x) \frac{d\Phi_j}{dx} dx \quad (1.7)$$

$$f_i = \int_{\Omega} \Phi_i(x) f(x) dx + \left( \Phi_i(x_1) k(x_1) \frac{du}{dx} \Big|_{x_1} - \Phi_i(x_0) k(x_0) \frac{du}{dx} \Big|_{x_0} \right) \quad (1.8)$$

### 1.1.2 Thermal Equilibrium in a Plate

In this model problem, a plate in thermal equilibrium is studied. It is modeled as a continuum in two-dimensional space. Each point in a closed region  $\bar{\Omega}$  of the  $xy$ -plane is mapped to a point on the plate. At equilibrium, conservation of energy requires, at any point on the plate, that the sum of internal and external heat influx add to zero, resulting in:

$$\nabla \cdot \phi(x, y) = q(x, y) \quad (1.9)$$

In this expression,  $\nabla \cdot \phi(x, y)$  is the divergence of  $\phi(x, y)$ ; and  $q(x, y)$  is the heat influx from external sources at point  $(x, y)$ .

The rate of internal heat flow  $\phi(x, y)$  is a function of the temperature gradient at point  $(x, y)$ . It is given by Fourier's law [2]:

$$\phi(x, y) = -k \nabla T(x, y) \quad (1.10)$$

where  $k$  is the thermal conductivity, and  $T(x, y)$  is the temperature. Substituting this constitutive relation in the conservation of energy equation (1.9) gives the governing equation:

$$-\nabla \cdot (k \nabla T(x, y)) = q(x, y) \quad (1.11)$$

Along with the boundary conditions, this governing equation specifies the thermal distribution on the plate at equilibrium.

### Weighted Residual and Weak Formulations

The steps of deriving weighted residual and weak formulations for two- and three-dimensional problems resemble those used in one-dimensional problems. Multiplying the governing equation by a test function  $v(x, y)$  and integrating over the domain  $\Omega$  gives the weighted residual formulation:

$$-\int_{\Omega} v(x, y) \nabla \cdot (k \nabla T(x, y)) \, d\Omega = \int_{\Omega} v(x, y) q(x, y) \, d\Omega \quad (1.12)$$

The differentiation is distributed evenly between  $v$  and  $T$  using the divergence

theorem to give the weak formulation:

$$\int_{\Omega} \nabla v(x, y) \cdot k \nabla T(x, y) d\Omega - \int_{\Gamma_N} v(x, y) k \nabla T(x, y) \cdot \boldsymbol{\nu} d\Gamma = \int_{\Omega} v(x, y) q(x, y) d\Omega \quad (1.13)$$

where  $\Gamma_N$  is the part of the boundary of  $\Omega$  where a natural boundary condition is specified, and  $\boldsymbol{\nu}$  is the outward unit normal at the boundary. A natural boundary condition at a point is one that specifies the value that the secondary variable, which here is denoted by  $\phi$ , must have at that point.

### Galerkin Formulation

Similar to the one-dimensional case described in the previous section, the weak formulation can be used along with a set of global shape functions  $\{\Phi_i\}_{i=1}^n$  to form a discretized approximation of the system. The stiffness matrix  $K = [k_{ij}]$  and the load vector  $\mathbf{f}$  are given by:

$$k_{ij} = \int_{\Omega} \nabla \Phi_i(x, y) \cdot k \nabla \Phi_j(x, y) d\Omega \quad (1.14)$$

$$f_i = \int_{\Omega} \Phi_i(x, y) q(x, y) d\Omega + \int_{\Gamma_N} \Phi_i(x, y) k \nabla T(x, y) \cdot \boldsymbol{\nu} d\Gamma \quad (1.15)$$

### 1.1.3 A Plate Under Plane-Stress

In this model problem, a plate in a state of plane stress is studied. At each point, the sum of the external forces and internal reaction forces due to stress is zero. This applies to the  $x$ - and  $y$ -components of these forces. So at each point  $(x, y)$  [3]:

$$\begin{aligned}\frac{\partial}{\partial x}\sigma_x + \frac{\partial}{\partial y}\tau_{yx} &= -b_x \\ \frac{\partial}{\partial y}\sigma_y + \frac{\partial}{\partial x}\tau_{xy} &= -b_y\end{aligned}\tag{1.16}$$

where  $\sigma_x$  and  $\sigma_y$  are respectively the normal stresses in the  $x$ - and  $y$ -directions;  $\tau_{xy}$  is the shear stress with  $\tau_{xy} = \tau_{yx}$ ; and  $b_x$  and  $b_y$  are respectively the  $x$ - and  $y$ -components of the body force  $\mathbf{b}$ . Thus equation (1.16) can be re-written in matrix form as:

$$\begin{bmatrix} \frac{\partial}{\partial x} & 0 & \frac{\partial}{\partial y} \\ 0 & \frac{\partial}{\partial y} & \frac{\partial}{\partial x} \end{bmatrix} \begin{bmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{bmatrix} = \begin{bmatrix} -b_x \\ -b_y \end{bmatrix}\tag{1.17}$$

In the same notation, the strain is given by:

$$\begin{bmatrix} \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} \end{bmatrix} \begin{bmatrix} u_x \\ u_y \end{bmatrix} = \begin{bmatrix} \epsilon_x \\ \epsilon_y \\ \epsilon_{xy} \end{bmatrix}\tag{1.18}$$

where  $\epsilon_x$  and  $\epsilon_y$  are respectively the normal strains in the  $x$ - and  $y$ -directions,  $\epsilon_{xy}$  is the shear strain with  $\epsilon_{xy} = \epsilon_{yx}$ ; and  $u_x$  and  $u_y$  are respectively the  $x$ - and  $y$ -components of the displacement vector  $\mathbf{u}$ . So by using  $\mathbf{B}$  to denote the differential operator,

$$\mathbf{B} = \begin{bmatrix} \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} \end{bmatrix}$$

equations (1.17) and (1.18) can be written as:

$$\mathbf{B}^T \boldsymbol{\sigma} = -\mathbf{b} \quad (1.19)$$

$$\mathbf{B}\mathbf{u} = \boldsymbol{\epsilon} \quad (1.20)$$

where:

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{bmatrix} \quad ; \quad \boldsymbol{\epsilon} = \begin{bmatrix} \epsilon_x \\ \epsilon_y \\ \epsilon_{xy} \end{bmatrix} \quad (1.21)$$

To obtain the governing equation, a constitutive model is needed. A commonly used constitutive model is the linear elastic constitutive relation, given by [4]:

$$\boldsymbol{\sigma} = \mathbf{D}\boldsymbol{\epsilon} \quad (1.22)$$

where  $\mathbf{D}$  is the stiffness, which in a plane stress model is given by:

$$\mathbf{D} = \frac{E}{1 - \nu^2} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix} \quad (1.23)$$

where  $E$  and  $\nu$  are respectively Young's modulus and Poisson's ratio. They are physical properties of the material from which the plate is made.

The governing equation for a plate in a condition of plane-stress is then given by:

$$-\mathbf{B}^\top (\mathbf{D}\mathbf{B}\mathbf{u}) = \mathbf{b} \quad (1.24)$$

This governing equation, along with the boundary conditions, describes the deformation of the plate.

### Weighted Residual and Weak Formulations

The governing equation (1.24) is a matrix representation of two scalar equations in two variables. Each of the two equations corresponds to one component of the force vector  $\mathbf{b}$ . The weighted residual formulation is derived by multiplying both sides of both the scalar equations by a test function  $v(x, y)$  and integrating over the domain  $\Omega$ , which gives:

$$-\int_{\Omega} \mathbf{V}\mathbf{B}^\top (\mathbf{D}\mathbf{B}\mathbf{u}) \, d\Omega = \int_{\Omega} \mathbf{V}\mathbf{b} \, d\Omega \quad (1.25)$$

where

$$\mathbf{V} = \begin{bmatrix} v(x, y) & 0 \\ 0 & v(x, y) \end{bmatrix} \quad (1.26)$$

The weak formulation is obtained by distributing the differentiation evenly between  $v(x, y)$  and  $\mathbf{u}$  by applying the divergence theorem to each of the two scalar equations, which gives the matrix equation:

$$\int_{\Omega} (\mathbf{B}\mathbf{V})^T \mathbf{D} (\mathbf{B}\mathbf{u}) \, d\Omega - \int_{\Gamma_N} \mathbf{V}\mathbf{N}^T \mathbf{D} (\mathbf{B}\mathbf{u}) \, d\Gamma = \int_{\Omega} \mathbf{V}\mathbf{b} \, d\Omega \quad (1.27)$$

where  $\Gamma_N$  is the part of the boundary of  $\Omega$  where a natural boundary condition is specified, and

$$\mathbf{N} = \begin{bmatrix} n_x & 0 \\ 0 & n_y \\ n_y & n_x \end{bmatrix}, \quad (1.28)$$

where  $n_x$  and  $n_y$  are respectively the  $x$ - and  $y$ -components of the outward unit normal to the boundary.

### Galerkin Formulation

The weak formulation given in equation (1.27) can be used with a set of global shape functions,  $\{\Phi_i\}_{i=1}^n$ , to form a discrete approximation of the problem using the Galerkin method. The components,  $u_x$  and  $u_y$  of the solution vector  $\mathbf{u}$  are assumed to be in a finite dimensional subspace  $\mathbf{V}^h$  spanned by the shape functions. The test functions are restricted to the same subspace  $\mathbf{V}^h$ . Then a stiffness matrix  $\mathbf{K} \in \mathbb{R}^{2n \times 2n}$  and a load vector  $\mathbf{f} \in \mathbb{R}^{2n}$  can be formed. A

convenient way of populating the stiffness matrix is to represent it as a block matrix that has  $n \times n$  blocks, each block,  $\boldsymbol{\kappa}_{ij}$ , being a  $2 \times 2$  submatrix given by:

$$\kappa_{ij} = \int_{\Omega} [\mathbf{B}(\Phi_i \otimes I_2)]^T \mathbf{D} [\mathbf{B}(\Phi_j \otimes I_2)] d\Omega \quad (1.29)$$

where  $I_2$  is the identity tensor and

$$\Phi_i \otimes I_2 = \begin{bmatrix} \Phi_i & 0 \\ 0 & \Phi_i \end{bmatrix} \quad (1.30)$$

The load vector can be given by a similar block representation:

$$\mathbf{F}_i = \int_{\Omega} (\Phi_i \otimes I_2) \mathbf{b} d\Omega + \int_{\Gamma_N} (\Phi_i \otimes I_2) \mathbf{N}^T \mathbf{D} (\mathbf{B} \mathbf{u}) d\Gamma \quad (1.31)$$

where

$$\mathbf{F}_i = \begin{bmatrix} f_{2i-1} \\ f_{2i} \end{bmatrix} \quad (1.32)$$

The solution vector  $\mathbf{u}$  is obtained by solving the linear system given by the stiffness matrix  $\mathbf{K}$  and the load vector  $\mathbf{f}$ . The components of the sought solution  $\mathbf{u}$  are then given by:

$$u_x = \sum_{i=1}^n \Phi_i w_{2i-1} \quad (1.33)$$

$$u_y = \sum_{i=1}^n \Phi_i w_{2i} \quad (1.34)$$

## Cantilever Beam

In addition to the governing equation, description of a structure requires knowledge of the loads and boundary conditions to which it is subjected. An arrangement of loads and boundary conditions that is often encountered in practice is that of a cantilever beam, which is a beam that is fully constrained at one end with the other end unsupported.

## 1.2 Interpolation and Approximation Functions

Meshless methods, as does the Finite Element Method, produce discretized approximations of problems that they are employed to solve. Construction of such approximations relies on restricting the test and trial spaces to finite dimensional subspaces. These finite dimensional subspaces are produced by constructing basis sets of functions. In the Finite Element Method and several meshless methods, elements of these basis sets are interpolation functions. In other meshless methods, elements of the basis sets are approximation functions. The problems of interpolation and approximation are described here.

For a problem domain  $\Omega \subset \mathbb{R}^n$  and a finite set,  $\mathbf{X} = \{(\mathbf{x}_i, u_i)\}_{i=1}^N \subset \Omega \times \mathbb{R}$ , of ordered pairs, the interpolation problem is to find a function  $u^h : \Omega \rightarrow \mathbb{R}$  such that for any  $(\mathbf{x}_i, u_i) \in \mathbf{X}$ ,  $u^h(\mathbf{x}_i) = u_i$ . An interpolation method can be thought of as a mapping, its domain being the set of all finite subsets of  $\Omega \times \mathbb{R}$ , and its range a subset of the set of all functions from  $\Omega$  to  $\mathbb{R}$ . In each of the ordered pairs  $(\mathbf{x}_i, u_i)$ , the first entry  $\mathbf{x}_i$  is called a *node*, and the second entry  $u_i$  is called a *nodal value* or *nodal parameter*.

Most interpolation methods used in practice are linear with respect to the nodal values. Linearity of an interpolation method with respect to the nodal values is defined in:

**Definition 1.1 (Linearity with respect to nodal values)** *An interpolation method is linear with respect to the nodal values if and only if:*

- for any set of nodes  $\{\mathbf{x}_i\}_{i=1}^n \subset \Omega$  and any pair of sets of nodal values  $\{u_i\}_{i=1}^n, \{v_i\}_{i=1}^n \subset \mathbb{R}$
- for any scalars  $\alpha, \beta \in \mathbb{R}$

*the interpolation functions  $u^h, v^h$  and  $w^h$ , produced respectively for the sets of ordered pairs  $\{(\mathbf{x}_i, u_i)\}_{i=1}^n, \{(\mathbf{x}_i, v_i)\}_{i=1}^n$  and  $\{(\mathbf{x}_i, \alpha u_i + \beta v_i)\}_{i=1}^n$ , satisfy  $w^h(\mathbf{x}) = \alpha u^h(\mathbf{x}) + \beta v^h(\mathbf{x})$  for all  $\mathbf{x} \in \Omega$ .*

One example of an interpolation method that is linear with respect to nodal values is the Lagrange interpolation method. If it is linear with respect to the nodal values, an interpolation method readily produces a basis for a finite dimensional subspace of the functions from the domain  $\Omega$  to  $\mathbb{R}$ . The reason is the following theorem:

**Theorem 1.1** *If an interpolation method  $I$  is linear with respect to the nodal values, then for any set of nodes  $\{\mathbf{x}_i\}_{i=1}^n \subset \Omega$  there is a set of functions  $\{\Phi_i\}_{i=1}^n$  such that the interpolation function  $u^h : \Omega \rightarrow \mathbb{R}$  produced by  $I$  for a set of ordered pairs  $\{(\mathbf{x}_i, u_i)\}_{i=1}^n$  is given by:*

$$u^h(\mathbf{x}) = \sum_{i=1}^n \Phi_i(\mathbf{x}) u_i \quad (1.35)$$

Furthermore, the functions  $\Phi_i$  satisfy the Kronecker delta property. That is to say, for all  $i = 1, \dots, n$ ,

$$\Phi_i(\mathbf{x}_j) = \delta_{ij} \quad (1.36)$$

where

$$\delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases} \quad (1.37)$$

**Proof.** For any  $i \in \{1, \dots, n\}$ , let  $\Phi_i$  be the function that the interpolation method produces for the set of ordered pairs  $\{(\mathbf{x}_j, \delta_{ij})\}_{j=1}^n$ . Let  $u^h$  be the interpolation function produced for the set of ordered pairs  $\{(\mathbf{x}_i, u_i)\}_{i=1}^n$ . Furthermore, for any  $m \in \{1, \dots, n\}$ , let  $w_m^h$  be the interpolation function produced for the set of ordered pairs  $\{(\mathbf{x}_i, \sum_{j=1}^m u_i \delta_{ij})\}_{i=1}^n$ . It is easy to verify that  $u^h = w_n^h$ .

By applying Definition 1.1, taking  $\{\mathbf{x}_i\}_{i=1}^n$  as the set of nodes, taking  $\{\delta_{1i}\}_{i=1}^n$  as both the sets of nodal values, and taking  $\alpha = 0$  and  $\beta = u_1$ , it holds that  $w_1^h = u_1 \Phi_1$ .

Similarly, taking  $\{\mathbf{x}_i\}_{i=1}^n$  as the set of nodes, taking  $\{\sum_{j=1}^1 u_j \delta_{ij}\}_{i=1}^n$  as the first set of nodal values, taking  $\{\delta_{2i}\}_{i=1}^n$  as the second set of nodal values, and taking  $\alpha = 1$  and  $\beta = u_2$ , and applying Definition 1.1 gives  $w_2^h = \sum_{i=1}^2 \Phi_i u_i$ .

Repeating this step, for  $m \in \{3, \dots, n\}$ , taking  $\{\mathbf{x}_i\}_{i=1}^n$  as the set of nodes, taking  $\{\sum_{j=1}^{m-1} u_j \delta_{ij}\}_{i=1}^n$  as the first set of nodal values, taking  $\{\delta_{mi}\}_{i=1}^n$  as the second set of nodal values, and taking  $\alpha = 1$  and  $\beta = u_m$ , and applying Definition 1.1 gives  $w_m^h = \sum_{i=1}^m \Phi_i u_i$ . Therefore, since  $u^h = w_n^h$ , equation (1.35) holds. Furthermore, since  $\Phi_i$  is the interpolation function for the set of ordered pairs  $\{(\mathbf{x}_j, \delta_{ij})\}_{j=1}^n$ , it satisfies the Kronecker delta property defined in equation (1.36).

■

For a problem domain  $\Omega \subset \mathbb{R}^n$ , and a finite set of ordered pairs  $\mathbf{X} = \{(\mathbf{x}_i, u_i)\}_{i=1}^n \subset \Omega \times \mathbb{R}$ , the approximation problem is to find a function  $u^h : \Omega \rightarrow \mathbb{R}$  such that for any  $(\mathbf{x}_i, u_i) \in \mathbf{X}$ ,  $u^h(\mathbf{x}_i) = u_i + \epsilon_i$ , and the errors  $\epsilon_i$  are minimal according to criteria specific to the problem. An approximation method can also be thought of as a mapping from the set of all finite subsets of  $\Omega \times \mathbb{R}$ , to a subset of the set of all functions from  $\Omega$  to  $\mathbb{R}$ . The definition of linearity of an approximation method with respect to the nodal parameters has the same form as Definition 1.1. Also, approximation functions that are linear with respect to the nodal parameters satisfy a theorem similar to Theorem 1.1:

**Theorem 1.2** *If an approximation method  $I$  is linear with respect to the nodal parameters, then for any set of nodes  $\{\mathbf{x}_i\}_{i=1}^n \subset \Omega$  there is a set of functions  $\{\Phi_i\}_{i=1}^n$  such that the approximation function  $u^h : \Omega \rightarrow \mathbb{R}$  produced by  $I$  for a set of ordered pairs  $\{(\mathbf{x}_i, u_i)\}_{i=1}^n$  is given by:*

$$u^h(\mathbf{x}) = \sum_{i=1}^n \Phi_i(\mathbf{x}) u_i \quad (1.38)$$

**Proof.** For any  $i = 1, \dots, n$ , let  $\Phi_i$  be the approximation function produced for the set of ordered pairs  $\{(\mathbf{x}_j, \delta_{ij})\}_{j=1}^n$ . It is easy to verify that for any  $i = 1, \dots, n$ ,

$$(\mathbf{x}_i, u_i) = (\mathbf{x}_i, \sum_{j=1}^n u_i \delta_{ij}) \quad (1.39)$$

So, by recursively applying Definition 1.1  $n$  times, the approximation function  $u^h$  for the set of ordered pairs  $\{(\mathbf{x}_i, u_i)\}_{i=1}^n$  satisfies equation (1.38). ■

It should be noted that in the case of approximation methods the shape functions  $\Phi_i$  do not necessarily satisfy the Kronecker delta property.

## Chapter 2

### Literature Survey

Meshless methods, like the Finite Element Method, produce discrete problems by restricting the trial and test functions to finite dimensional subspaces. These finite dimensional subspaces are created by constructing finite sets of shape functions. The key to understanding meshless methods and differentiating them from one another is to study the methods employed in constructing their shape functions. In most meshless methods, shape functions are constructed using an interpolation or approximation method, as described in Theorems 1.1 and 1.2. An exception to this is the construction of enrichment shape functions using a partition of unity, as will be described in Subsection 2.1.4. The following section is a brief survey of the literature of meshless shape functions. Section 2.3 looks at the use of Graphics Processing Units for general purpose computation, and Section 2.2 looks at meshless interpolants and prepares for the introduction of the Moving Regularized Interpolation method, which is the interpolatory meshless method developed in the current study.

## 2.1 Meshless and Extended Shape Functions

Many meshless methods have been proposed in the literature. These methods can be classified into three broad categories, according to the shape functions that form the bases for their trial and test spaces [5, 6]. The first category contains weighted integral approximation methods, such as the Smoothed Particle Hydrodynamics method. The second category contains Moving Least-Squares approximation methods, such as the Element-Free Galerkin method. The third category contains extrinsic basis approximation methods, such as the Extended Finite Element method.

In various meshless methods, the shape functions are formed using weight functions called window functions. Window functions are discussed in the next subsection, and the subsections following that review the broad categories of meshless shape functions: Weighted Integral approximations, Moving Least-Squares approximations, and Extrinsic Basis and Extended approximations.

### 2.1.1 Window Functions

In many meshless methods, construction of shape functions involves the use of window functions, which are also called smoothing functions or kernel functions. Typically, a window function is constructed from a prototypical mapping  $\tilde{w} : \mathbb{R}^+ \rightarrow \mathbb{R}$ , where  $\mathbb{R}^+$  denotes the set of non-negative real numbers. Meshless methods impose various requirements on the window functions, but all the methods reviewed here require that  $\tilde{w}(r)$ :

### Conditions 2.1

- *be continuous on  $\Omega$*
- *satisfy  $\tilde{w}(r) = 0$  whenever  $r \geq 1$*
- *satisfy  $\tilde{w}(r) > 0$  whenever  $r < 1$*

Commonly used window functions [5, 6, 7] include:

- cubic spline:

$$\tilde{w}(r) = \begin{cases} \frac{2}{3} - 4r^2 + 4r^3 & r \leq \frac{1}{2} \\ \frac{4}{3} - 4r + 4r^2 - \frac{4}{3}r^3 & \frac{1}{2} < r \leq 1 \\ 0 & r > 1 \end{cases} \quad (2.1)$$

- quartic spline:

$$\tilde{w}(r) = \begin{cases} 1 - 6r^2 + 8r^3 - 3r^4 & r \leq 1 \\ 0 & r > 1 \end{cases} \quad (2.2)$$

- exponential:

$$\tilde{w}(r) = \begin{cases} e^{-(r/\alpha)^2} & r \leq 1 \\ 0 & r > 1 \end{cases} \quad (2.3)$$

Although the exponential window function has a discontinuity and is nonzero at  $r = 1$ , its value  $\tilde{w}(1)$  at that point is so small that the discontinuity has no significance in practical implementations.

Given a centre point  $\mathbf{x}_0$  and a support radius  $h$ , a window function  $w : \Omega \rightarrow \mathbb{R}$  can be constructed from the prototype  $\tilde{w}(r)$  by taking:

$$w(\mathbf{x}_0, \mathbf{x}, h) = \tilde{w}(d(\mathbf{x}_0, \mathbf{x}, h)) \quad (2.4)$$

Commonly  $d(\mathbf{x}_0, \mathbf{x}, h)$  is the Euclidian distance  $\|\mathbf{x}_0 - \mathbf{x}\|$  normalized by the support radius  $h$ :

$$d(\mathbf{x}_0, \mathbf{x}, h) = \frac{\|\mathbf{x}_0 - \mathbf{x}\|}{h} \quad (2.5)$$

though different mappings can be used for constructing the distance function  $d$  or the window function  $w$ . For example, often in two-dimensional problems, window functions with rectangular support are constructed using two distance functions ( $d_x$  and  $d_y$ ), and two support dimensions ( $h_x$  and  $h_y$ ). These functions are then given by:

$$d_x(\mathbf{x}_0, \mathbf{x}, h) = \frac{|x_0 - x|}{h_x} \quad (2.6)$$

$$d_y(\mathbf{x}_0, \mathbf{x}, h) = \frac{|y_0 - y|}{h_y} \quad (2.7)$$

$$w(\mathbf{x}_0, \mathbf{x}, h_x, h_y) = \tilde{w}(d_x(\mathbf{x}_0, \mathbf{x}, h)) \tilde{w}(d_y(\mathbf{x}_0, \mathbf{x}, h)) \quad (2.8)$$

where  $x_0$  and  $y_0$  are the coordinates of  $\mathbf{x}_0$ , and  $x$  and  $y$  are the coordinates of  $\mathbf{x}$ .

## Domain of Influence

A term that is used often in the context of meshless methods is the *domain of influence* of a point. If a meshless method uses a window function  $w(\mathbf{x}_0, \mathbf{x}, h)$ , then the domain of influence of a point  $\mathbf{x}_0 \in \Omega$  includes all the points  $\mathbf{x} \in \Omega$  for which  $w(\mathbf{x}_0, \mathbf{x}, h) \neq 0$ . The commonly used window functions are symmetric around  $\mathbf{x}_0$ , so if a point  $\mathbf{x}$  is in the domain of influence of another point  $\mathbf{y}$ , then  $\mathbf{y}$  is in the domain of influence of  $\mathbf{x}$ , provided the support radius  $h$  is the same for the two points.

### 2.1.2 Weighted Integral Approximations

Weighted integral approximation methods use the window function directly to form the shape functions. The earliest instance of this class of methods is the Smoothed Particle Hydrodynamics method. It was introduced by Lucy [8] and Gingold and Monaghan [9] and is the earliest of meshless methods in the literature [5, 6].

A weighted integral approximation can be thought of as a discrete representation of a kernel approximation. A kernel approximation of a function  $u : \Omega \rightarrow \mathbb{R}$  is given by:

$$\tilde{u}^h(\mathbf{x}) = \int_{\Omega} w(\mathbf{y}, \mathbf{x}, h) u(\mathbf{y}) \, d\Omega_{\mathbf{y}} \quad (2.9)$$

where  $w(\mathbf{y}, \mathbf{x}, h)$  is the value at  $\mathbf{x}$  of the window function constructed with centre  $\mathbf{y}$  and support radius  $h$ . In addition to Conditions 2.1, window functions for weighted integral approximation methods must satisfy [8, 9]:

- $\int_{\Omega} w(\mathbf{y}, \mathbf{x}, h) d\Omega_{\mathbf{y}} = 1$
- $w$  is constructed from a monotonically decreasing prototype  $\tilde{w} : \mathbb{R}^+ \rightarrow \mathbb{R}$
- $\lim_{h \rightarrow 0} w(\mathbf{y}, \mathbf{x}, h) = \delta(\mathbf{x} - \mathbf{y})$ , where  $\delta(\mathbf{x})$  is the Dirac delta function

Weighted integral meshless shape functions for a set of scattered nodes  $\{\mathbf{x}_i\}_{i=1}^N$  are then obtained by a discrete approximation of the integral in equation (2.9) obtained using a quadrature rule. The nodes  $\mathbf{x}_i$  are used as quadrature points in this integration, and by using  $\Delta V_i$  to denote the quadrature weight for point

$\mathbf{x}_i$ , the weighted integral approximation is given by:

$$u^h(\mathbf{x}) = \sum_{i=1}^N w(\mathbf{x}_i, \mathbf{x}, h) u_i \Delta V_i \quad (2.10)$$

A difficulty in generating meshless shape functions in this way arises from locations of the nodes being determined before a quadrature rule is selected. Meshless methods using weighted integral formulations assign methods for the selection of the quadrature weights  $\Delta V_i$ . For instance, in Smoothed Particle Hydrodynamics for one-dimensional problems,  $\Delta V_i$  is taken as the internode spacing at node  $\mathbf{x}_i$ .

### 2.1.3 Moving Least-Squares Approximations

In this second category of meshless shape functions, the strategy used involves generating a mapping between points in the domain  $\Omega$  and problems, each of which has a unique solution that is a function on  $\Omega$ . Often, the problem mapped to each point  $\mathbf{x}$  is a least-squares problem of fitting a set of functions, typically polynomials, to the nodes in the vicinity of  $\mathbf{x}$ . The use of this strategy with a single basis function, a constant function, for approximation of scattered data was reported as early as 1968 [10]. Several works that used more basis functions or introduced singularities to give exact interpolation were published in the 1970's [11, 12, 13]. Lancaster and Salkauskas [14] gave an overview of the Moving Least-Squares interpolation method, and presented an analysis of the smoothness of the resulting interpolants.

The concept of a uniquely solvable problem promises to be useful in characterising the methods often classified in the category of Moving Least-Squares approximations. It can be formalized by defining it as a mapping:

**Definition 2.1 (Problem)** A problem  $\Pi$  on a set  $X$  is a mapping  $\Pi : X \rightarrow \{0, 1\}$ .

An element  $x \in X$  is said to be a solution to  $\Pi$  if  $\Pi x = 1$ .

**Definition 2.2 (Uniquely solvable problem)** A problem  $\Pi$  on a set  $X$  is a uniquely solvable problem if both the following conditions hold:

- there exists  $x \in X$  such that  $\Pi x = 1$
- for all  $x, y \in X$ , if both  $\Pi x = 1$  and  $\Pi y = 1$ , then  $x = y$

The Moving Least-Squares approximation maps to each point  $\mathbf{x} \in \Omega$  a uniquely solvable problem  $\Pi_{\mathbf{x}}$  on the set of all functions defined on  $\Omega$ . The value of the approximation function at that point  $u^h(\mathbf{x})$  is then obtained by evaluating the solution  $f_{\mathbf{x}}$  of  $\Pi_{\mathbf{x}}$  at  $\mathbf{x}$  to get:

$$u^h(\mathbf{x}) = f_{\mathbf{x}}(\mathbf{x}) \tag{2.11}$$

where

$$\Pi_{\mathbf{x}} f_{\mathbf{x}} = 1 \tag{2.12}$$

In the Moving Least-Squares approximation method, the uniquely solvable problem mapped to a point  $\mathbf{x}$  is a weighted least-squares fitting problem whose weights come from a window function  $w$  centered at  $\mathbf{x}$  with support radius  $h$ . The uniquely solvable problem mapped to  $\mathbf{x}$  then has the form:

**Uniquely Solvable Problem 2.1 (Moving Least Squares)** For a set of ordered pairs of nodes and nodal parameters  $\{(\mathbf{x}_i, u_i)\}_{i=1}^N \subset \Omega \times \mathbb{R}$ , and a set of lin-

**Table 2.1:** Pascal Triangle of Bivariate Monomials

|       |        |        |          |        |        |
|-------|--------|--------|----------|--------|--------|
|       |        |        | 1        |        |        |
|       |        | $x$    |          | $y$    |        |
|       | $x^2$  |        | $xy$     |        | $y^2$  |
| $x^3$ |        | $x^2y$ |          | $xy^2$ | $y^3$  |
| $x^4$ | $x^3y$ |        | $x^2y^2$ |        | $xy^3$ |
|       |        |        |          | $y^4$  |        |

early independent functions  $\{p_i\}_{i=1}^m$ , find a function of the form  $f_{\mathbf{x}} = \sum_{i=1}^m p_i(\mathbf{x})a_i$  that minimizes the quantity:

$$\sum_{i=1}^N w(\mathbf{x}, \mathbf{x}_i, h) (f_{\mathbf{x}}(\mathbf{x}_i) - u_i)^2 \quad (2.13)$$

In this formulation,  $w(\mathbf{x}, \mathbf{x}_i, h)$  is the value of a specific window function centered at  $\mathbf{x}$  and evaluated at node  $\mathbf{x}_i$ . The functions  $\{p_i\}_{i=1}^m$  are typically a set of monomials that is complete to a specific order [5, 7, 14]. For instance, in one-dimensional problems, the monomials  $p_i(x) = x^{(i-1)}$  are often used, and in two-dimensional problems, the basis functions are commonly the functions above a certain level in a Pascal's triangle of monomials, a few levels of which are shown in Table 2.1.

Since  $f_{\mathbf{x}}(\mathbf{x}_i) = \sum_{j=1}^m p_j(\mathbf{x}_i)a_j$ , the minimization objective function in equation (2.13) becomes:

$$\sum_{i=1}^N w(\mathbf{x}, \mathbf{x}_i, h) \left( \sum_{j=1}^m p_j(\mathbf{x}_i)a_j - u_i \right)^2 \quad (2.14)$$

This expression can be written in a concise form by defining the matrices  $\mathbf{P} \in \mathbb{R}^{N \times m}$  and  $\mathbf{W} \in \mathbb{R}^{N \times N}$  and the vectors  $\mathbf{a} \in \mathbb{R}^m$  and  $\mathbf{u} \in \mathbb{R}^N$ . The matrix  $\mathbf{P} = [p_{ij}]$  is given by  $p_{ij} = p_j(\mathbf{x}_i)$ . The moving weight matrix  $\mathbf{W} = [w_{ij}]$  is a diagonal matrix whose diagonal entries are given by  $w_{ii} = w(\mathbf{x}, \mathbf{x}_i, h)$ . Furthermore, the entries of  $\mathbf{a}$  and  $\mathbf{u}$  are given by  $a_i$  and  $u_i$  as they appear in equation (2.14). The minimization objective in equation (2.14) can then be written as:

$$(\mathbf{P}\mathbf{a} - \mathbf{u})^\top \mathbf{W} (\mathbf{P}\mathbf{a} - \mathbf{u}) \quad (2.15)$$

This minimization problem has a unique solution if and only if the matrix  $\mathbf{P}^\top \mathbf{W} \mathbf{P}$ , which is known as the moment matrix, is invertible. This solution is the point  $\hat{\mathbf{a}} \in \mathbb{R}^m$  where the variation of the minimization objective function with respect to  $\mathbf{a}$  vanishes:

$$\begin{aligned} \frac{\partial}{\partial \mathbf{a}} [(\mathbf{P}\mathbf{a} - \mathbf{u})^\top \mathbf{W} (\mathbf{P}\mathbf{a} - \mathbf{u})] \Big|_{\hat{\mathbf{a}}} &= 2 \cdot \mathbf{P}^\top \mathbf{W} \mathbf{P} \hat{\mathbf{a}} - 2 \cdot \mathbf{P}^\top \mathbf{W} \mathbf{u} \\ &= 0 \end{aligned}$$

which reduces to:

$$\hat{\mathbf{a}} = (\mathbf{P}^\top \mathbf{W} \mathbf{P})^{-1} \mathbf{P}^\top \mathbf{W} \mathbf{u} \quad (2.16)$$

This gives the unique solution to Problem 2.1 as:

$$\begin{aligned} f_{\mathbf{x}}(\mathbf{y}) &= \mathbf{p}(\mathbf{y})^\top \hat{\mathbf{a}} \\ &= \mathbf{p}(\mathbf{y})^\top (\mathbf{P}^\top \mathbf{W} \mathbf{P})^{-1} \mathbf{P}^\top \mathbf{W} \mathbf{u} \end{aligned} \quad (2.17)$$

where  $\mathbf{p}(\mathbf{y})$  is a column vector whose entries are given by  $p_i = p_i(\mathbf{y})$  so that  $\mathbf{p}(\mathbf{y})^\top \hat{\mathbf{a}} = \sum_{i=1}^m p_i(\mathbf{y}) a_i$ . The value of the Moving Least-Squares approximation function at  $\mathbf{x}$  is then given by evaluating  $f_{\mathbf{x}}$  at  $\mathbf{x}$ :

$$\begin{aligned} u^h(\mathbf{x}) &= f_{\mathbf{x}}(\mathbf{x}) \\ &= \mathbf{p}(\mathbf{x})^\top \hat{\mathbf{a}} \\ &= \mathbf{p}(\mathbf{x})^\top (\mathbf{P}^\top \mathbf{W} \mathbf{P})^{-1} \mathbf{P}^\top \mathbf{W} \mathbf{u} \end{aligned} \quad (2.18)$$

In computing this expression, it is not necessary to explicitly compute the inverse  $(\mathbf{P}^\top \mathbf{W} \mathbf{P})^{-1}$  of the moment matrix. A more efficient alternative was described by Belytschko *et al* [15]. In this procedure, equation (2.18) is written as:

$$u^h(\mathbf{x}) = \mathbf{c}(\mathbf{x})^\top \mathbf{P}^\top \mathbf{W} \mathbf{u} \quad (2.19)$$

where

$$\mathbf{P}^\top \mathbf{W} \mathbf{P} \mathbf{c}(\mathbf{x}) = \mathbf{p}(\mathbf{x}) \quad (2.20)$$

So the inverse of the moment matrix is only required for solving the linear system in equation (2.20). In the procedure described by Belytschko *et al* [15],

the linear system is solved by computing the LU decomposition of the moment matrix and then solving the system by back substitution followed by forward substitution. This procedure is more efficient than explicit inversion of the moment matrix. However, an even more efficient procedure is available because of the symmetry and positive definiteness of the moment matrix. The required computational time and resources can be approximately halved by solving the linear system using LDL decomposition followed by back substitution and then forward substitution [16].

### 2.1.4 Extrinsic Basis and Extended Finite Elements

Using a partition of unity of the problem domain  $\Omega \subset \mathbb{R}^n$ , a function defined globally on  $\Omega$  can be represented as a sum of locally defined functions. This is the strategy for construction of shape functions discussed in this subsection.

A partition of unity of the problem domain  $\Omega \subset \mathbb{R}^n$  is a set of functions  $\{\psi_i\}_{i=1}^m$  that satisfy a number of conditions [17]. Each of the partition of unity functions  $\psi_i$  is locally defined on a subdomain  $\Omega_i \subset \Omega$ . The subdomains  $\{\Omega_i\}_{i=1}^m$  may overlap with one another, but they must entirely cover the domain  $\Omega$ . Furthermore, for any  $\mathbf{x} \in \Omega$ , the partition of unity functions  $\{\psi_i\}$  satisfy:

$$\sum_{i=1}^m \psi_i(\mathbf{x}) = 1 \tag{2.21}$$

Hence, any function  $p : \Omega \rightarrow \mathbb{R}$  can be represented as:

$$p(\mathbf{x}) = \sum_{i=1}^m \psi_i(\mathbf{x}) p(\mathbf{x}) \quad (2.22)$$

In this representation, the global function  $p(\mathbf{x})$  is given by the sum of  $m$  local functions,  $\psi_i(\mathbf{x})p(\mathbf{x})$ . Methods based on the partition of unity representation use the terms in equation (2.22) as shape functions. That is to say, the shape functions in methods based on partition of unity representation are of the form:

$$\Phi_{I,J} = \psi_I(\mathbf{x}) p_J(\mathbf{x}) \quad (2.23)$$

where  $\{\psi_I\}_{I=1}^m$  is a partition of unity of the problem domain, and  $\{p_J\}_{J=1}^n$  is a set of functions defined globally on  $\Omega$ . The functions  $\{p_J\}_{J=1}^n$  are called enrichment functions. Each one of them produces  $m$  functions to be included in the bases of the trial and test spaces. A basis for the approximations space can then be composed of functions of the form in equation (2.23) in addition to the partition of unity functions  $\{\psi_I\}_{I=1}^m$ . So an approximation function has the form:

$$\begin{aligned} u^h(\mathbf{x}) &= \sum_{I=1}^m \sum_{J=1}^n \Phi_{I,J}(\mathbf{x}) \tilde{a}_{I,J} + \sum_{I=1}^m \psi_I(\mathbf{x}) \hat{a}_I \\ &= \sum_{I=1}^m \sum_{J=1}^n \psi_I(\mathbf{x}) p_J(\mathbf{x}) \tilde{a}_{I,J} + \sum_{I=1}^m \psi_I(\mathbf{x}) \hat{a}_I \\ &= \sum_{I=1}^m \psi_I(\mathbf{x}) \left( \sum_{J=1}^n p_J(\mathbf{x}) \tilde{a}_{I,J} + \hat{a}_I \right) \end{aligned} \quad (2.24)$$

for some coefficients  $\tilde{a}_{I,J}$  and  $\hat{a}_I$ .

## 2.2 Meshless Interpolants

Moving Least-Squares shape functions are not exact interpolants, so when they are used to form trial and test spaces for meshless methods, essential boundary conditions cannot be imposed, as in the Finite Element Method, by simple substitution of nodal values. Several methods have been formulated for imposing essential boundary conditions in meshless methods with shape functions that are not exact interpolants [18, 19, 20]. Use of these methods, however, comes at an added numerical effort. Using Lagrange multipliers as proposed by Belytschko [18] gives rise to a larger system that is not positive definite. Using collocation methods [19, 21] requires partition of the node displacements into dependent and independent displacement vectors, and preprocessing of the stiffness matrix to eliminate the dependent displacement vector variables. Imposing essential boundary conditions by the penalty method [19] results, in general, in an inexact fulfillment of the boundary conditions.

Because of the difficulties in enforcing essential boundary conditions in meshless methods with shape functions that are not interpolatory, it is advantageous to formulate meshless shape functions that are interpolants. Several formulations based on the Moving Least-Squares concept have been proposed to produce meshless interpolants; they include the use of window functions that produce interpolants when used in the Moving Least-Squares formulation [14, 22, 23], and the Point Interpolation Method [24], in which no window function is used. To investigate the strategy adopted in these methods, it is helpful to recall the description of the Moving Least-Squares formulation given in Section 2.1.3; it is a mapping from the domain  $\Omega$  to a set of uniquely solvable problems (Definition 2.2) on the set of functions on  $\Omega$ . In this description, the value of the Moving Least-Squares approximation function  $u^h$  at a point  $\mathbf{x} \in \Omega$  is computed in three implicit steps. In the first step, the uniquely solvable problem  $\Pi_{\mathbf{x}}$  that

is mapped to  $\mathbf{x}$  is formed. In the second step, the function  $f_{\mathbf{x}} : \Omega \rightarrow \mathbb{R}$  that uniquely solves  $\Pi_{\mathbf{x}}$  is obtained. In the third step,  $f_{\mathbf{x}}$  is evaluated at  $\mathbf{x}$ , yielding the desired image  $u^h(\mathbf{x})$ . Following this three-step procedure, the value of  $u^h$  at any of the nodes  $\mathbf{x}_i$  is given by evaluating  $f_{\mathbf{x}_i}$  at  $\mathbf{x}_i$ :

$$u^h(\mathbf{x}_i) = f_{\mathbf{x}_i}(\mathbf{x}_i) \quad (2.25)$$

where  $f_{\mathbf{x}_i}$  is the solution to the uniquely solvable problem  $\Pi_{\mathbf{x}_i}$  that is mapped to the node  $\mathbf{x}_i$ . Exact interpolation requires that the value of  $u^h$  at the nodes be equal to their nodal parameters. In other words, exact interpolation means that  $u^h(\mathbf{x}_i) = u_i$  for all the nodes. This can be formulated as:

**Condition 2.2** *Let  $u^h : \Omega \rightarrow \mathbb{R}$  be a function that is constructed by first mapping to each  $\mathbf{x} \in \Omega$  a uniquely solvable problem  $\Pi_{\mathbf{x}}$  on the set of functions on  $\Omega$ , and then assigning to  $\mathbf{x}$  the image  $u^h(\mathbf{x})$  obtained by evaluating, at  $\mathbf{x}$ , the function  $f_{\mathbf{x}}$  that solves  $\Pi_{\mathbf{x}}$ . Let each  $\mathbf{x}_i$  in a set of nodes  $\{\mathbf{x}_i\}_{i=1}^N \subset \Omega$  have a nodal parameter  $u_i \in \mathbb{R}$ , and let  $f_{\mathbf{x}_i}$  be the solution to the uniquely solvable problem  $\Pi_{\mathbf{x}_i}$  mapped to  $\mathbf{x}_i$  in the first step of constructing  $u^h$ . Then for  $u^h$  to achieve exact interpolation, it must hold that:*

$$f_{\mathbf{x}_i}(\mathbf{x}_i) = u_i \quad (2.26)$$

In the Moving Least-Squares formulation and the other formulations based on this family, the uniquely solvable problem mapped to each point specifies a set of basis functions  $\{p_j\}_{j=1}^m$  and stipulates a solution of the form:

$$f_{\mathbf{x}} = \sum_{j=1}^m p_j a_j \quad (2.27)$$

Any point  $\mathbf{a} \in \mathbb{R}^m$  can be used to construct a function of this form by substituting for the coefficients  $a_j$  the coordinates of  $\mathbf{a}$ . Hence, there is a one-to-one correspondence between functions of this form and the  $m$ -dimensional vector space  $\mathbb{R}^m$ ; the coordinates of any vector  $\mathbf{a} \in \mathbb{R}^m$  can be used as the coefficients  $a_j$  of a function  $f$ , and vice-versa. This mapping gives a geometrical sense to Condition 2.2, because substituting equation (2.27) into equation (2.26) gives:

$$f_{\mathbf{x}_i}(\mathbf{x}_i) = \sum_{j=1}^m p_j(\mathbf{x}_i) a_j = u_i \quad (2.28)$$

This equation describes a hyperplane in  $\mathbb{R}^m$ . The coordinates  $a_j$  of any point  $\mathbf{a}$  on this hyperplane correspond to a function that exactly interpolates  $u_i$ .

The function form in equation (2.27) is used in the Moving Least-Squares formulation, in the interpolatory methods discussed in this section, as well as in the Moving Regularized Interpolation method developed in this work and described in the next chapter. In all these methods, at each point  $\mathbf{x} \in \Omega$ , a number of nodes in a neighbourhood around  $\mathbf{x}$  are used along with their nodal parameters to form the uniquely solvable problem to be mapped to  $\mathbf{x}$ , each of these nodes producing an equation of the form in equation (2.28). In the Moving Least-Squares formulation, the number of nodes used is greater than the number of basis functions  $m$ . Geometrically, this means that the number of hyperplanes is greater than the dimension of the vector space. In general, this means that there is no point  $\mathbf{a} \in \mathbb{R}^m$  that is on all the hyperplanes. This can easily be visualized by considering three lines in  $\mathbb{R}^2$ . The object of the Moving Least-Squares uniquely solvable problem is then to find the point  $\mathbf{a} \in \mathbb{R}^m$  such that the weighted sum of the squared distances between  $\mathbf{a}$  and the hyperplanes is as small as possible.

Before investigating the modifications to the Moving Least Squares, Condition 2.2 may be illustrated by considering the consistency of a Moving Least-Squares approximation function. An approximation method is said to have order- $k$  consistency if it represents exactly any function whose  $k^{th}$  derivative is a constant [5]. Then recalling Uniquely Solvable Problem 2.1 (on page 25) used to construct the Moving Least-Squares approximation, if the nodal parameters  $u_i$  of the nodes  $\{\mathbf{x}_i\}_{i=1}^N$  being approximated are given by any of the basis functions ( $u_i = p_j(\mathbf{x}_i)$  for all  $\mathbf{x}_i \in \{\mathbf{x}_i\}_{i=1}^N$ ), then quantity (2.13) evaluates to 0 for the function  $f(\mathbf{x}) = p_j(\mathbf{x})$ . Consequently, since quantity (2.13) is non-negative,  $p_j$  is a solution to Uniquely Solvable Problem 2.1 (and is therefore, by definition of a uniquely solvable problem, the only solution). This holds regardless of the weight function used. In this case,  $p_j$  is the solution to any uniquely solvable problem  $\Pi_{\mathbf{x}}$  mapped to any point  $\mathbf{x} \in \Omega$ , including the nodes  $\mathbf{x}_i$ . Therefore  $u^h(\mathbf{x}) = p_j(\mathbf{x})$ , because the image  $u^h(\mathbf{x})$  of  $\mathbf{x}$  is obtained by evaluating, at  $\mathbf{x}$ , the solution  $f_{\mathbf{x}}$  to the uniquely solvable problem  $\Pi_{\mathbf{x}}$ . Similarly, if the nodal parameters  $u_i$  are given by any linear combination of the basis functions ( $u_i = \sum_{j=1}^m p_j(\mathbf{x}_i)a_j$  for all  $\mathbf{x}_i \in \{\mathbf{x}_i\}_{i=1}^N$ ), then  $u^h(\mathbf{x}) = \sum_{j=1}^m p_j(\mathbf{x})a_j$  for all  $\mathbf{x} \in \Omega$ . Hence, if the basis functions used for construction of the Moving Least-Squares functions include all the monomials of orders 0 to  $k$ , then the resulting approximation is order- $k$  consistent. It should be noted that in this case, a condition that is stronger than condition 2.2 is satisfied. The solution to the uniquely solvable problem corresponding to a node  $\mathbf{x}_i$  passes exactly not only through  $(\mathbf{x}_i, u_i)$ , but through  $(\mathbf{x}_j, u_j)$  for all the nodes  $\mathbf{x}_j$  in the support domain of  $\mathbf{x}_i$ . This stronger condition is satisfied throughout the domain  $\Omega$ , not only at the nodes  $\mathbf{x}_i$ .

The Point Interpolation Method proposed by Liu and Gu [24] is based on shape functions produced from a form of uniquely solvable problems that enforces a similar condition. At each point  $\mathbf{x} \in \Omega$ , the uniquely solvable problem  $\Pi_{\mathbf{x}}$  to be mapped to  $\mathbf{x}$  is constructed such that its solution  $f_{\mathbf{x}}$  exactly interpolates

all the nodes in a neighbourhood around  $\mathbf{x}$ . As in the Moving Least-Squares formulation, the solution sought has the form  $f_{\mathbf{x}}(\mathbf{y}) = \sum_{j=1}^m p_j(\mathbf{y})a_j$ . For each point  $\mathbf{x} \in \Omega$ , the corresponding uniquely solvable problem is constructed using the  $m$  nodes  $\{\tilde{\mathbf{x}}_i\}_{i=1}^m$  that are closest to the point  $\mathbf{x}$ . If for each node  $\tilde{\mathbf{x}}_i$  in  $\{\tilde{\mathbf{x}}_i\}_{i=1}^m$  the associated nodal value is given by  $u_i$ , the uniquely solvable problem corresponding to the Point Interpolation Method is given by:

**Uniquely Solvable Problem 2.2 (Point Interpolation Method)** *Find the function  $f_{\mathbf{x}}(\mathbf{y}) = \sum_{j=1}^m p_j(\mathbf{y})a_j$  such that  $f_{\mathbf{x}}(\tilde{\mathbf{x}}_i) = u_i$  for all  $i = 1, \dots, m$ .*

This uniquely solvable problem gives rise to a linear system of  $m$  equations in  $m$  variables. This system can be expressed as a matrix-vector product by defining, as was done in the Moving Least-Squares formulation, the matrix  $\mathbf{P} \in \mathbb{R}^{m \times m}$  and the vectors  $\mathbf{a}, \mathbf{u} \in \mathbb{R}^m$ . Again, the entries of  $\mathbf{P}$  are  $p_{ij} = p_j(\tilde{\mathbf{x}}_i)$  and the entries of  $\mathbf{u}$  and  $\mathbf{a}$  are given respectively by the nodal parameters  $u_i$  and the coefficients  $a_i$  of  $f_{\mathbf{x}}$ . The solution to Uniquely Solvable Problem 2.2 is then given by the function  $f_{\mathbf{x}}$  constructed from the coefficients of the vector  $\mathbf{a}$  given by:

$$\mathbf{a} = \mathbf{P}^{-1}\mathbf{u} \quad (2.29)$$

So the solution to Uniquely Solvable Problem 2.2 is  $f_{\mathbf{x}}(\mathbf{y}) = \mathbf{p}(\mathbf{y})^T \mathbf{a}$ , where  $\mathbf{p}(\mathbf{y})$  is a column vector whose  $i^{\text{th}}$  entry is given by  $p_i(\mathbf{y})$ . The image of  $\mathbf{x}$  under the Point Interpolation Method shape function is then obtained by evaluating  $f_{\mathbf{x}}$  at  $\mathbf{x}$  to yield:

$$u^h(\mathbf{x}) = f_{\mathbf{x}}(\mathbf{x}) = \mathbf{p}(\mathbf{x})^T \mathbf{a} \quad (2.30)$$

which, by substituting equation (2.29), gives:

$$u^h(\mathbf{x}) = \mathbf{p}(\mathbf{x})^\top \mathbf{P}^{-1} \mathbf{u} \quad (2.31)$$

It has been assumed in this discussion of the Point Interpolation Method that the matrix  $\mathbf{P}$  is invertible. This assumption is implicit in the definition of Uniquely Solvable Problem 2.2. Indeed, if  $\mathbf{P}$  is singular then the requirements given in the definition of Uniquely Solvable Problem 2.2 do not specify a uniquely solvable problem. If  $\mathbf{P}$  is singular, then its rows do not span  $\mathbb{R}^m$ , and so there is a non-zero vector  $\mathbf{n} \in \mathbb{R}^m$  in the nullspace of  $\mathbf{P}$ , that is,  $\mathbf{P}\mathbf{n} = \mathbf{0}$ . So if there is a function  $f_{\mathbf{x}} = \sum_{j=1}^m p_j a_j$  that satisfies the requirements set in the definition of Uniquely Solvable Problem 2.2, and whose coefficients  $a_j$  are given by a vector  $\mathbf{a}$ , then any function given by  $\tilde{f}(\mathbf{y}) = f_{\mathbf{x}}(\mathbf{y}) + \mathbf{p}(\mathbf{y})^\top \mathbf{n}$  also satisfies the requirements. Also, if  $\mathbf{P}$  is singular, then its column space does not span  $\mathbb{R}^m$ , and there is a non-zero vector  $\mathbf{u}_0 \in \mathbb{R}^m$  in the orthogonal complement of the range of  $\mathbf{P}$ . Furthermore, if the nodal parameters are given by the entries of a vector  $\mathbf{u} = \alpha \mathbf{u}_0$  where  $\alpha \in \mathbb{R}$  is nonzero, then there is no values  $a_j$  that satisfy the requirements. In other words, if  $\mathbf{P}$  is singular, then the problem either has no solutions, or has infinitely many.

Using the geometrical interpretation given above, the Point Interpolation Method is based on taking  $m$  hyperplanes in  $\mathbb{R}^m$ . Invertibility of the matrix  $\mathbf{P}$  is geometrically interpreted as linear independence of the normal vectors of the  $m$  hyperplanes. In other words, if  $\mathbf{P}$  is invertible, then the hyperplanes intersect in a single point  $\mathbf{a} \in \mathbb{R}^m$ . This point is the solution to the uniquely solvable problem produced by the Point Interpolation Method. This interpretation can be visualized by considering two lines in  $\mathbb{R}^2$ . The lines intersect in a single point, unless their normal vectors are parallel. The normal vectors are parallel if and

only if the matrix  $\mathbf{P}$ , whose rows are given by their coefficients, is singular. If the normal vectors are parallel, then the two lines are either coincident or they do not intersect.

Another aspect of the Point Interpolation Method that merits consideration is compatibility. A meshless shape function is said to be compatible if it is continuous throughout the problem domain. Compatibility of shape functions is necessary for convergence of the Galerkin method [7]. Shape functions generated by the Point Interpolation Method are, in general, discontinuous and therefore incompatible. This can be illustrated by considering the following interpolation problem. A set of four evenly spaced nodes in the interval  $\Omega = [0, 1]$  are chosen. For each point  $x \in \Omega$ , a Point Interpolation Method uniquely solvable problem is constructed using the three nearest nodes  $\xi_j$  ( $j = 1, 2, 3$ ) to  $x$ . Furthermore, the basis functions used are  $p_j = x^{(j-1)}$ , ( $j = 1, 2, 3$ ). Thus, if for each one of the nodes  $\xi_i$  the corresponding nodal value is  $u_i$ , then the uniquely solvable problem is to find the function  $f_x = \sum_{j=1}^3 x^{(j-1)} a_j$  such that  $f_x(\xi_i) = u_i$ , ( $i = 1, 2, 3$ ). If each of two nodes has the same distance to  $x$ , the one with a smaller  $x$ -coordinate is considered nearer to  $x$ . In this setting, all the points on the interval  $[0, \frac{1}{2}]$  are mapped to one uniquely solvable problem  $\Pi_1$ , and all the points on the interval  $(\frac{1}{2}, 1]$  are mapped to another uniquely solvable problem  $\Pi_2$ . Since the interpolant assumes at each point  $x \in \Omega$  the value of the solution  $f_x$  to the uniquely solvable problem mapped to  $x$ , the interpolant on the interval  $[0, \frac{1}{2}]$  assumes the value of the solution to  $\Pi_1$ , and in the interval  $(\frac{1}{2}, 1]$  it is given by the solution to  $\Pi_2$ . Since these two functions do not necessarily have the same value at  $x = \frac{1}{2}$ , the interpolant is in general discontinuous at that point. This scenario is illustrated in Figure 2.1, where the Point Interpolation Method is used to interpolate the points  $\{(0, 2), (\frac{1}{3}, 1), (\frac{2}{3}, 2), (1, 1)\}$ .

Several methods that achieve interpolation by using special window functions with the Moving Least-Squares method have been proposed [14, 22, 23]. Like the

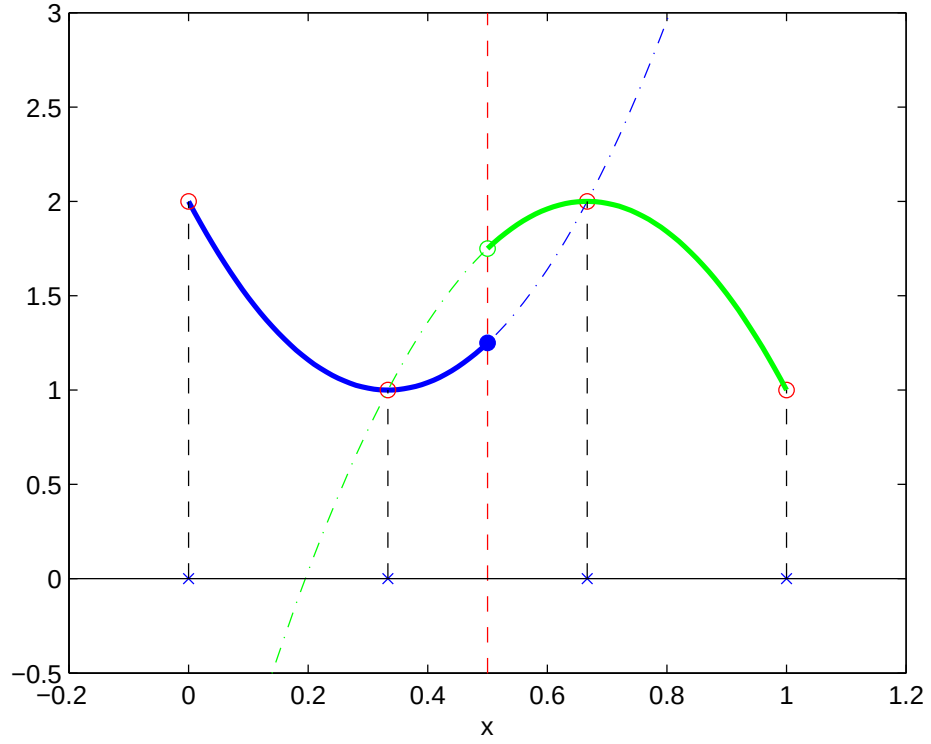


**Figure 2.1:** Discontinuity in the Point Interpolation Method

Moving Least-Squares method with standard window functions, these methods map to each point  $\mathbf{x} \in \Omega$  a uniquely solvable problem whose solution is of the form in equation (2.27). The solution minimizes a weighted sum of the squared errors in interpolating the nodes in the influence domain of  $\mathbf{x}$ . Also like the standard Moving Least-Squares method, the window function is chosen such that there are more nodes to interpolate than there are basis functions in the form of the solution. The difference is that as the point  $\mathbf{x}$  approaches a node point  $\mathbf{x}_i$ , the weight corresponding to the error in interpolating  $\mathbf{x}_i$  becomes much larger than the weights corresponding to the other nodes. In the limit as  $\mathbf{x}$  approaches  $\mathbf{x}_i$ , the ratio between the weight corresponding to  $\mathbf{x}_i$  and the weight corresponding to any other node becomes infinite, so that the solution to the uniquely solvable problem associated with  $\mathbf{x}_i$  exactly interpolates it. In the interpolation method reported by Lancaster and Salkauskas [14], the weight

corresponding to  $\mathbf{x}_i$  becomes infinite in the limit as  $\mathbf{x}$  approaches  $\mathbf{x}_i$ . In the interpolating window function proposed by Most and Bucher [23], in the limit as  $\mathbf{x}$  approaches  $\mathbf{x}_i$ , the weight corresponding to  $\mathbf{x}_i$  approaches unity while the weights corresponding to all the other nodes approach zero.

## 2.3 GPU Computation

In a large class of modern computers, a comparatively large portion of the demand for computing power has come from graphics processing requirements, for generation and manipulation of images to be displayed to the users. This made it desirable to improve the efficiency of modern computers in performing this form of computation. This, along with the highly parallel nature of graphics computation functions, has given rise to the development of Graphics Processing Units. Graphics Processing Units are computing subsystems designed to accelerate the processing of images to be displayed to the user.

Both the availability and demand for computing power have been growing steadily since the invention of the electronic computer. The availability of computing power has been fueled by improvements in the technology, which have faithfully followed the trend commonly known as Moore's law [25]. Moore's law is the name given to the exponential trend of growth in the number of transistors that can be incorporated into an integrated circuit. The performance attainable from processors has grown with the number of transistors that they can be manufactured to contain. Historically, increased transistor density in processors has been transmuted into improved performance by increasing the clock speed, the rate at which computation cycles are completed. However, this method of increasing performance reached a barrier when clock speeds reached a level which it was unfeasible to exceed because of heat dissipation require-

ments. At that point, harnessing transistor density for improved performance was sought in parallelism, and processors with multiple cores became commonplace [26, 27].

In general, for increasing computing speed, it is important to both increase throughput and reduce latency. For graphics computation, however, latency is less critical than it is for general computing tasks. As a result, it is not surprising that modern Graphics Processing Units (GPU's) outperform modern Central Processing Units in the number of floating point operations per second (FLOPs) that they can complete [28]. This has created an opportunity to apply GPU hardware in computation tasks other than graphics processing, if the tasks are highly parallel. To harness the power of Graphics Processing Units in general computation, several programming protocols have been developed to make GPU resources accessible to programmers. These protocols include Common Uniform Device Architecture (CUDA), openCL and Direct-Compute [29]. CUDA is used in the implementation developed in the current work.

## Chapter 3

### Moving Regularized Interpolation

Moving Regularized Interpolation is a novel method which produces a function that interpolates a set of points scattered in a domain  $\Omega$ . Like the Moving Least-Squares method, the Moving Regularized Interpolation method can be interpreted as a mapping from the domain  $\Omega$  to a set of uniquely solvable problems on the set of all functions from  $\Omega$  to  $\mathbb{R}$ . Thus, given a point  $\mathbf{x} \in \Omega$ , the Moving Regularized Interpolation method produces a uniquely solvable problem  $\Pi_{\mathbf{x}} : X \rightarrow \{0, 1\}$ , where  $X$  is the set of functions from  $\Omega$  to  $\mathbb{R}$ . The image  $u^h(\mathbf{x})$  of any point  $\mathbf{x} \in \Omega$  under the interpolant  $u^h : \Omega \rightarrow \mathbb{R}$  produced by the Moving Regularized Interpolation method is then given by evaluating, at  $\mathbf{x}$ , the function  $f_{\mathbf{x}}$  that solves that uniquely solvable problem  $\Pi_{\mathbf{x}}$ . Like the Moving Least-Squares method, the function  $f_{\mathbf{x}}$  is a linear combination of linearly independent functions  $\{p_i\}_{i=1}^m$ , and, for any  $\mathbf{y} \in \Omega$ , it has the form:

$$f_{\mathbf{x}}(\mathbf{y}) = \sum_{i=1}^m p_i(\mathbf{y}) a_i$$

Also like the Moving Least-Squares method, the Moving Regularized Interpolation method generates a uniquely solvable problem for every point  $\mathbf{x}$  from the nodes in the neighbourhood around  $\mathbf{x}$ . Unlike the Moving Least-Squares

method, however, in the Moving Regularized Interpolation method, the nodes that enter into the uniquely solvable problem are fewer than the basis functions in the form of the solution. Recalling the geometric interpretation given to equation (2.28), having fewer nodes than there are functions translates to having fewer hyperplanes than the dimension of the solution space. An example is the problem of interpolating two nodes with a general quadratic function, which incorporates three basis functions:  $\{1, x, x^2\}$ . There are infinitely many solutions to this problem. The coefficient vectors of these solutions correspond to points that lie on a straight line in  $\mathbb{R}^3$ . It is the line of intersection of two planes, namely the two planes that are given by the two interpolation equations of the two points being interpolated. Thus, the problem of minimizing the interpolation errors is not a uniquely solvable problem. In the Moving Regularized Interpolation method, uniqueness is obtained by introducing regularization to the problem. Regularization in this context means adding the norm of the solution to the objective function. As an example, a regularized version of the problem given above is the problem of approximating the two points with a quadratic function

$$f(x) = \sum_{i=1}^3 x^{(i-1)} a_i \quad (3.1)$$

such that the regularized objective function

$$\sum_{i=1}^2 (f(x_i) - u_i)^2 + \rho \sum_{i=1}^3 a_i^2 \quad (3.2)$$

is as small as possible. This regularized problem, where  $\rho > 0$ , has exactly one solution. In the limit as  $\rho \rightarrow 0$ , the solution approaches an exact interpolant. Such use of regularization is the idea behind the Moving Regularized Interpolation method.

In the Moving Regularized Interpolation method, the problem  $\Pi_{\mathbf{x}}$  mapped to each point  $\mathbf{x}$  is a regularized least-squares problem. In a regularized least-

squares problem, the minimization objective function is a norm involving the solution as well as the error. So for  $\mathbf{P} \in \mathbb{R}^{N \times m}$  and  $\mathbf{u} \in \mathbb{R}^N$ , a regularized least-squares minimization objective function has the form [30]:

$$J(\mathbf{a}) = \|\mathbf{P}\mathbf{a} - \mathbf{u}\|_{\mathbf{W}}^2 + \|\mathbf{a}\|_{\mathbf{R}}^2 \quad (3.3)$$

In this equation, the seminorm  $\|\cdot\|_{\mathbf{W}}$  and the norm  $\|\cdot\|_{\mathbf{R}}$  are given by a symmetric positive semidefinite matrix  $\mathbf{W} \in \mathbb{R}^{N \times N}$  and a symmetric positive definite matrix  $\mathbf{R} \in \mathbb{R}^{m \times m}$  so that for any  $\mathbf{u} \in \mathbb{R}^N$ ,  $\|\mathbf{u}\|_{\mathbf{W}}^2 = \mathbf{u}^\top \mathbf{W} \mathbf{u}$  and for any  $\mathbf{a} \in \mathbb{R}^m$ ,  $\|\mathbf{a}\|_{\mathbf{R}}^2 = \mathbf{a}^\top \mathbf{R} \mathbf{a}$ .

One way to solve the regularized least-squares problem is to recognize that for a vector  $\hat{\mathbf{a}} \in \mathbb{R}^m$  to minimize the objective function  $J(\mathbf{a})$ , the variation of  $J$  must be zero at  $\hat{\mathbf{a}}$ , that is,

$$\left. \frac{\partial}{\partial \mathbf{a}} ((\mathbf{a}^\top \mathbf{P}^\top - \mathbf{u}^\top) \mathbf{W} (\mathbf{P} \mathbf{a} - \mathbf{u}) + \mathbf{a}^\top \mathbf{R} \mathbf{a}) \right|_{\hat{\mathbf{a}}} = 0 \quad (3.4)$$

which evaluates to:

$$(\mathbf{P}^\top \mathbf{W} \mathbf{P} + \mathbf{R}) \hat{\mathbf{a}} = \mathbf{P}^\top \mathbf{W} \mathbf{u} \quad (3.5)$$

Since  $\mathbf{P}^\top \mathbf{W} \mathbf{P}$  is symmetric positive semidefinite for any  $\mathbf{P} \in \mathbb{R}^{N \times m}$  and  $\mathbf{R}$  is positive definite,  $(\mathbf{P}^\top \mathbf{W} \mathbf{P} + \mathbf{R})$  is positive definite and therefore invertible, so equation (3.5) has the unique solution given by:

$$\hat{\mathbf{a}} = (\mathbf{P}^\top \mathbf{W} \mathbf{P} + \mathbf{R})^{-1} \mathbf{P}^\top \mathbf{W} \mathbf{u} \quad (3.6)$$

To define the Moving Regularized Interpolation uniquely solvable problem for a point  $\mathbf{x} \in \Omega$ ,

- Let  $\{(\mathbf{x}_i, u_i)\}_{i=1}^N \subset \Omega \times \mathbb{R}$  be the set of ordered pairs of nodes to be interpolated and their corresponding nodal values.

- Let  $\{p_i\}_{i=1}^m$  be a linearly independent set of functions on  $\Omega$ .
- For some influence domain radius  $h \in \mathbb{R}$ , let  $w(\mathbf{x}, \mathbf{y}, h)$  be a window function that satisfies Conditions 2.1.
- Let  $\rho : \Omega \rightarrow \mathbb{R}^+$  be a continuous positive distance measure such that  $\rho(\mathbf{x}_i) = 0$  for all  $i \in \{1, \dots, N\}$ .
- Define the matrix  $\mathbf{P} \in \mathbb{R}^{N \times m}$  by  $\mathbf{P} = [p_{ij}]$  where  $p_{ij} = p_j(\mathbf{x}_i)$ .
- Form the vector  $\mathbf{u} \in \mathbb{R}^N$  by taking the nodal value  $u_i$  as its  $i^{\text{th}}$  entry.
- Define the diagonal matrix  $\mathbf{W} \in \mathbb{R}^{N \times N}$  by  $\mathbf{W} = [w_{ij}]$  where

$$w_{ij} = \begin{cases} w(\mathbf{x}, \mathbf{x}_i, h) & : i = j \\ 0 & : i \neq j \end{cases}$$

- For some positive definite matrix  $\mathbf{R}_0 \in \mathbb{R}^{m \times m}$ , define  $\mathbf{R} = \rho \mathbf{R}_0$ .
- For any  $\mathbf{y} \in \Omega$ , define the vector  $\mathbf{p}(\mathbf{y}) \in \mathbb{R}^m$  by taking  $p_i(\mathbf{y})$  as its  $i^{\text{th}}$  entry.

**Uniquely Solvable Problem 3.1 (Moving Regularized Interpolation)** *Find the function  $f_{\mathbf{x}} : \Omega \rightarrow \mathbb{R}$  of the form  $f_{\mathbf{x}}(\mathbf{y}) = \mathbf{p}(\mathbf{y})^T \mathbf{a}$  such that if  $\mathbf{x} \notin \{\mathbf{x}_i\}_{i=1}^N$ , the quantity:*

$$J_{\mathbf{x}}(\mathbf{a}) = (\mathbf{P}\mathbf{a} - \mathbf{u})^T \mathbf{W}(\mathbf{P}\mathbf{a} - \mathbf{u}) + \mathbf{a}^T \mathbf{R}\mathbf{a} \quad (3.7)$$

*is minimized over all  $\mathbb{R}^m$ . Otherwise if  $\mathbf{x} \in \{\mathbf{x}_i\}_{i=1}^N$ ,  $\mathbf{W}\mathbf{P}\mathbf{a} = \mathbf{W}\mathbf{u}$  and the quantity:*

$$J_{\mathbf{x}_i}(\mathbf{a}) = \mathbf{a}^T \mathbf{R}_0^{-1} \mathbf{a} \quad (3.8)$$

*is minimized over all  $\mathbb{R}^m$ .*

If  $\mathbf{x} \notin \{\mathbf{x}_i\}_{i=1}^N$ , the solution to this uniquely solvable problem is given by the coefficient vector  $\hat{\mathbf{a}}$  specified by equation (3.6). The function that solves the regularized least-squares problem corresponding to point  $\mathbf{x} \in \Omega \setminus \{\mathbf{x}_i\}_{i=1}^N$  is then given by:

$$f_{\mathbf{x}}(\mathbf{y}) = \mathbf{p}(\mathbf{y})^T (\mathbf{P}^T \mathbf{W} \mathbf{P} + \mathbf{R})^{-1} \mathbf{P}^T \mathbf{W} \mathbf{u} \quad (3.9)$$

where  $\mathbf{W}$  and  $\mathbf{R}$  depend on the point  $\mathbf{x}$ . The value, at  $\mathbf{x}$ , of the interpolant produced by the Moving Regularized Interpolation method is then given by evaluating  $f_{\mathbf{x}}$  at  $\mathbf{x}$ , which yields:

$$u^h(\mathbf{x}) = f_{\mathbf{x}}(\mathbf{x}) = \mathbf{p}(\mathbf{x})^\top (\mathbf{P}^\top \mathbf{W} \mathbf{P} + \mathbf{R})^{-1} \mathbf{P}^\top \mathbf{W} \mathbf{u} \quad (3.10)$$

If  $\mathbf{x} \in \{\mathbf{x}_i\}_{i=1}^N$ , the solution to Uniquely Solvable Problem 3.1 is given by a coefficient vector  $\mathbf{a}$  that exactly solves  $\mathbf{W} \mathbf{P} \mathbf{a} = \mathbf{W} \mathbf{u}$ . Therefore,  $f_{\mathbf{x}_i}(\mathbf{x}_j) = u_j$  for any  $i, j$  such that  $w_{ii} > 0$  and  $w_{jj} > 0$ . In particular, if  $\mathbf{x} = \mathbf{x}_i$ , then  $f_{\mathbf{x}_i}(\mathbf{x}_i) = u_i$  and therefore  $\mathbf{x}_i$  is interpolated exactly.

The reason for the alternative definition of the uniquely solvable problem and its solution at the node points is that the distance measure  $\rho$  vanishes at the node points, and as a consequence the objective function given in equation (3.7) ceases to specify a unique solution. However, specifying that the solution  $\mathbf{a}$  at the node points satisfy  $\mathbf{W} \mathbf{P} \mathbf{a} = \mathbf{W} \mathbf{u}$  ensures that the resulting function is not only an exact interpolant, but also a continuous function.

**Proposition 3.1** *The Moving Regularized Interpolation method produces an exact interpolant that is continuous.*

**Proof.** It is clear that the resulting function is an exact interpolant, as the uniquely solvable problem specifies, at any node  $\mathbf{x}_i$ , that  $\mathbf{W} \mathbf{P} \mathbf{a} = \mathbf{W} \mathbf{u}$ . The  $i^{\text{th}}$  row of this system of equations specifies that  $w_{ii} \mathbf{p}_{(i,:)}^\top \mathbf{a} = w_{ii} u_i$ , where  $\mathbf{p}_{(i,:)} \in \mathbb{R}^m$  is a column vector whose  $j^{\text{th}}$  entry is given by  $p_j(\mathbf{x}_i)$ . Dividing both sides of this equation by  $w_{ii}$  gives  $f_{\mathbf{x}_i}(\mathbf{x}_i) = u_i$ , which is the requirement for exact interpolation.

The resulting interpolant is clearly continuous at any point that is not a node

point, since it is given by equation (3.10), which is continuous whenever the regularized moment matrix  $(\mathbf{P}^\top \mathbf{W} \mathbf{P} + \mathbf{R})$  is invertible.

To show that the method is continuous at the nodes, let  $\mathbf{x}_i$  be any one of the nodes. The objective is to show that in the limit as  $\mathbf{x}$  approaches  $\mathbf{x}_i$ , the coefficients vector  $\mathbf{a}$  that minimizes the objective function in equation (3.7) satisfies  $\mathbf{W} \mathbf{P} \mathbf{a} = \mathbf{W} \mathbf{u}$ . Let  $\Xi = (\xi_j) \subset \Omega \setminus \{\mathbf{x}_i\}_{i=1}^N$  be a sequence that converges in  $\Omega$  to one of the nodes  $\mathbf{x}_i$ . For each  $\xi_j$ , let  $\mathbf{W}_j$  and  $\mathbf{R}_j = \rho(\xi_j) \mathbf{R}_0$  be the matrices formed for  $\xi_j$  as described in the definition of the Moving Regularized Interpolation uniquely solvable problem. Then for each  $\xi_j$ , the uniquely solvable problem is to find a function  $f_{\xi_j}(\mathbf{y}) = \mathbf{p}(\mathbf{y})^\top \mathbf{a}_j$  such that the quantity:

$$J_{\xi_j}(\mathbf{a}_j) = (\mathbf{P} \mathbf{a}_j - \mathbf{u})^\top \mathbf{W}_j (\mathbf{P} \mathbf{a}_j - \mathbf{u}) + \mathbf{a}_j^\top \mathbf{R}_j \mathbf{a}_j \quad (3.11)$$

is minimum over all of  $\mathbb{R}^m$ . The vector of coefficients  $\mathbf{a}_j$  is given by equation (3.5) as:

$$(\mathbf{P}^\top \mathbf{W}_j \mathbf{P} + \mathbf{R}_j) \mathbf{a}_j = \mathbf{P}^\top \mathbf{W}_j \mathbf{u} \quad (3.12)$$

By construction of the Moving Regularized Interpolation method, there are more functions  $p_i$  than there are nonzero entries in the diagonal matrix  $\mathbf{W}_j$ ; let the number of nonzero entries in  $\mathbf{W}_j$  be  $n$ . Then, a matrix  $\tilde{\mathbf{P}} \in \mathbb{R}^{n \times m}$  can be formed by removing from  $\mathbf{P}$  all the rows that correspond to nodes other than those in the domain of influence of  $\xi_j$ . Similarly, a diagonal matrix  $\tilde{\mathbf{W}}_j \in \mathbb{R}^{n \times n}$  can be formed by removing from  $\mathbf{W}_j$  all the rows and columns that correspond to nodes outside the domain of influence of  $\xi_j$ . Equation (3.12) then reduces to:

$$(\tilde{\mathbf{P}}^\top \tilde{\mathbf{W}}_j \tilde{\mathbf{P}} + \mathbf{R}_j) \mathbf{a}_j = \tilde{\mathbf{P}}^\top \tilde{\mathbf{W}}_j \mathbf{u} \quad (3.13)$$

The right-hand side of this equation is in  $\mathcal{R}(\tilde{\mathbf{P}}^\top) \subset \mathbb{R}^m$ : the column space of  $\tilde{\mathbf{P}}^\top$ . Therefore, the left-hand side must also be in  $\mathcal{R}(\tilde{\mathbf{P}}^\top)$ . Hence, the second

term on the left-hand side does not change by taking its projection onto  $\mathcal{R}(\tilde{\mathbf{P}}^\top)$ , resulting in the equation:

$$\left( \tilde{\mathbf{P}}^\top \tilde{\mathbf{W}}_j \tilde{\mathbf{P}} + \tilde{\mathbf{P}}^\top (\tilde{\mathbf{P}} \tilde{\mathbf{P}}^\top)^{-1} \tilde{\mathbf{P}} \mathbf{R}_j \right) \mathbf{a}_j = \tilde{\mathbf{P}}^\top \tilde{\mathbf{W}}_j \mathbf{u} \quad (3.14)$$

Multiplying both sides by  $(\tilde{\mathbf{P}} \tilde{\mathbf{P}}^\top)^{-1} \tilde{\mathbf{P}}$  gives:

$$\left( \tilde{\mathbf{W}}_j \tilde{\mathbf{P}} + (\tilde{\mathbf{P}} \tilde{\mathbf{P}}^\top)^{-1} \tilde{\mathbf{P}} \mathbf{R}_j \right) \mathbf{a}_j = \tilde{\mathbf{W}}_j \mathbf{u} \quad (3.15)$$

By assumption,  $\lim_{j \rightarrow \infty} \boldsymbol{\xi}_j = \mathbf{x}_i$ . Thus, since  $\rho$  is continuous,  $\lim_{j \rightarrow \infty} \mathbf{R}_j = \mathbf{0}$ . Therefore, taking the limit as  $j \rightarrow \infty$  on both sides of equation (3.15) gives:

$$\left( \tilde{\mathbf{W}} \tilde{\mathbf{P}} \right) \mathbf{a} = \tilde{\mathbf{W}} \mathbf{u} \quad (3.16)$$

Restoring the rows of  $\mathbf{P}$  and the rows and columns of  $\mathbf{W}$  that were removed to form  $\tilde{\mathbf{P}}$  and  $\tilde{\mathbf{W}}$  leaves the equation unchanged. Therefore:

$$(\mathbf{W} \mathbf{P}) \mathbf{a} = \mathbf{W} \mathbf{u} \quad (3.17)$$

which completes the proof. ■

Constructing a regularized least-squares problem to be mapped to a point  $\mathbf{x}$  requires mapping, to  $\mathbf{x}$ , two matrices to parametrize the regularized least-squares problem: the diagonal weight matrix  $\mathbf{W}$  and the regularization matrix  $\mathbf{R}$ . The matrix  $\mathbf{W}$  is generated in the same way as in the Moving Least-Squares method: using a window function. The mapping for the matrix  $\mathbf{R}$ , however, has different requirements. It must be continuous and positive, and it must evaluate to zero at the nodes. Furthermore, to ensure that the shape functions produced by the method have local support, only the nodes in the vicinity of a point  $\mathbf{x}$  should

enter in the formulation of the uniquely solvable problem mapped to  $\mathbf{x}$ . In constructing the matrix  $\mathbf{W}$ , this is guaranteed by selecting a window function with a local support. But this locality requirement must be taken into account in constructing a regularization matrix. In the next section, a method of constructing the regularization matrix that meets these requirements is presented.

### 3.1 Parallel Distance Function

To achieve exact interpolation in the Moving Regularized Interpolation method, the penalty on the norm of the solution vector  $\mathbf{a}$  must vanish at the points where interpolation is required. For continuity, the regularization value must not change abruptly at the node point; smoothness of the regularization matrix is required for smoothness of the resulting interpolation functions. This section describes the novel procedure employed to achieve this end, the Parallel Distance Function.

What is required is a function  $\rho : \Omega \rightarrow \mathbb{R}$  whose value approaches zero in the vicinity of any of the nodes where exact interpolation is required. This function should exhibit local behaviour, thus computing its value at a point  $\mathbf{x}$  must not involve nodes that are far from  $\mathbf{x}$ . Also, it must be continuous on  $\Omega$ . A function that meets these requirements was formulated and named the *Parallel Distance Function*, because its form resembles the formula for the total resistance of electrical resistors connected in parallel. It is defined by:

$$p_d(\mathbf{x}) = \begin{cases} \left( \sum_{i=1}^N \frac{w(\mathbf{x}, \mathbf{x}_i, h)}{\|\mathbf{x} - \mathbf{x}_i\|^2} \right)^{-1} & \mathbf{x} \notin \{\mathbf{x}_i\}_{i=1}^N \\ 0 & \mathbf{x} \in \{\mathbf{x}_i\}_{i=1}^N \end{cases} \quad (3.18)$$

**Proposition 3.2** *If the window function  $w(\mathbf{x}, \mathbf{x}_i, h)$  is continuous on  $\Omega$ , then*

the Parallel Distance Function defined by equation (3.18) is also continuous on  $\Omega$ .

**Proof.** If the window function  $w(\mathbf{x}, \mathbf{x}_i, h)$  is continuous on  $\Omega$  then for any  $\mathbf{y} \in \Omega \setminus \{\mathbf{x}_i\}_{i=1}^N$ :

$$\begin{aligned} \lim_{\mathbf{x} \rightarrow \mathbf{y}} p_d(\mathbf{x}) &= \lim_{\mathbf{x} \rightarrow \mathbf{y}} \left( \sum_{i=1}^N \frac{w(\mathbf{x}, \mathbf{x}_i, h)}{\|\mathbf{x} - \mathbf{x}_i\|^2} \right)^{-1} \\ &= \left( \sum_{i=1}^N \frac{\lim_{\mathbf{x} \rightarrow \mathbf{y}} w(\mathbf{x}, \mathbf{x}_i, h)}{\lim_{\mathbf{x} \rightarrow \mathbf{y}} \|\mathbf{x} - \mathbf{x}_i\|^2} \right)^{-1} \\ &= \left( \sum_{i=1}^N \frac{w(\mathbf{y}, \mathbf{x}_i, h)}{\|\mathbf{y} - \mathbf{x}_i\|^2} \right)^{-1} \\ &= p_d(\mathbf{y}) \end{aligned}$$

Therefore,  $p_d$  is continuous on  $\Omega \setminus \{\mathbf{x}_i\}_{i=1}^N$ .

In addition, for any node  $\mathbf{x}_i$ ,  $i = 1 \dots N$ :

$$\begin{aligned} \lim_{\mathbf{x} \rightarrow \mathbf{x}_i} \frac{w(\mathbf{x}, \mathbf{x}_i, h)}{\|\mathbf{x} - \mathbf{x}_i\|^2} &= \frac{w(\mathbf{x}_i, \mathbf{x}_i, h)}{\lim_{\mathbf{x} \rightarrow \mathbf{x}_i} \|\mathbf{x} - \mathbf{x}_i\|^2} \\ &= \infty \end{aligned}$$

Thus, since  $\frac{w(\mathbf{x}, \mathbf{x}_j, h)}{\|\mathbf{x} - \mathbf{x}_j\|^2} \geq 0$  for any node  $\mathbf{x}_j$ ,  $j = 1 \dots N$ :

$$\lim_{\mathbf{x} \rightarrow \mathbf{x}_i} \sum_{i=1}^N \frac{w(\mathbf{x}, \mathbf{x}_i, h)}{\|\mathbf{x} - \mathbf{x}_i\|^2} = \infty$$

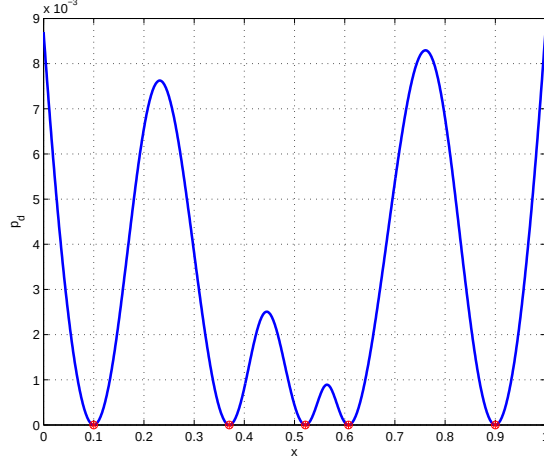


**Figure 3.1:** Parallel Distance Function

and as a result:

$$\begin{aligned} \lim_{\mathbf{x} \rightarrow \mathbf{x}_i} p_d(\mathbf{x}) &= \lim_{\mathbf{x} \rightarrow \mathbf{x}_i} \left( \sum_{i=1}^N \frac{w(\mathbf{x}, \mathbf{x}_i, h)}{\|\mathbf{x} - \mathbf{x}_i\|^2} \right)^{-1} \\ &= 0 \\ &= p_d(\mathbf{x}_i) \end{aligned}$$

Therefore,  $p_d$  is continuous on  $\{\mathbf{x}_i\}_{i=1}^N$  as well, which completes the proof. ■

Figure 3.1 shows a plot of the Parallel Distance Function computed for a set of five nodes.

## 3.2 Differentiation of Regularized Interpolation

### Method Shape Functions

For use of shape functions produced by the Moving Regularized Interpolation method in a Galerkin formulation, the spatial derivatives of these functions

are required. The form of the Moving Regularized Interpolation method shape functions closely resembles that of the Moving Least-Squares shape functions. So it is not surprising that the spatial derivatives of the functions produced by the two methods have similar structures. Differentiation of the Moving Regularized Interpolation method shape functions requires differentiation of an additional term: the regularization matrix  $\mathbf{R}$ . Differentiation of the regularization matrix, in turn, requires differentiation of the distance measure which, in the current work, is given by the Parallel Distance Function, which is defined in equation (3.18).

The shape functions, whose form is shown in equation (3.10), have spatial derivatives of the form given by:

$$\begin{aligned} f_{,i}(\mathbf{x}) = & \mathbf{p}_{,i}^T (\mathbf{P}^T \mathbf{W} \mathbf{P} + \mathbf{R})^{-1} \mathbf{P}^T \mathbf{W} \mathbf{u} \\ & - \mathbf{p}^T (\mathbf{P}^T \mathbf{W} \mathbf{P} + \mathbf{R})^{-1} (\mathbf{P}^T \mathbf{W}_{,i} \mathbf{P} + \mathbf{R}_{,i}) (\mathbf{P}^T \mathbf{W} \mathbf{P} + \mathbf{R})^{-1} \mathbf{P}^T \mathbf{W} \mathbf{u} \\ & + \mathbf{p}^T (\mathbf{P}^T \mathbf{W} \mathbf{P} + \mathbf{R})^{-1} \mathbf{P}^T \mathbf{W}_{,i} \mathbf{u} \end{aligned} \quad (3.19)$$

where the subscript  $i$  denotes differentiation with respect to the  $i^{\text{th}}$  spatial coordinate. This expression is obtained by differentiating equation (3.10) using the product rule, and the derivative of a matrix inverse given by [31]:

$$(\mathbf{A}^{-1})_{,i} = -\mathbf{A}^{-1} \mathbf{A}_{,i} \mathbf{A}^{-1} \quad (3.20)$$

### 3.3 An Interpolation Example

In this section, an example of using the Moving Regularized Interpolation method for interpolating a set of data points in a one-dimensional domain is presented.

The Matlab script that performs this demonstration is included in Appendix B.1. A set of nodes,  $\{x_i\}_{i=1}^N, N = 21$ , is obtained by starting with a set of evenly spaced nodes in the interval  $[0, 1]$  and then adding a small perturbation to the coordinate of each inner node. The nodal values are obtained by sampling a uniform random distribution. For the sake of this description, the values of the node coordinates and nodal values are given in Table 3.1. As an example, the procedure for computing the Moving Regularized Interpolant value at  $x = 0.42$  is outlined. The plot of the interpolant is shown in Figure 3.2.

The basis functions used in this example are  $\{1, x, x^2, \dots, x^9\}$ . The cubic spline window function described in equation (2.1) on page 21 is used, with an influence domain radius  $h = 0.15$ . This window function as it is evaluated for  $x = 0.42$  is shown in Figure 3.3. The values of this window function at the nodes, which are highlighted on the figure with rings, are used as the diagonal entries of the weight matrix  $\mathbf{W}$ . There are only 6 nodes whose corresponding weights are nonzero, namely  $\{x_7, x_8, x_9, x_{10}, x_{11}, x_{12}\}$ . The regularization matrix is formed by multiplying the Parallel Distance Function value at  $x = 0.42$  by a constant diagonal matrix  $\mathbf{R}_0$ . The  $i^{\text{th}}$  diagonal entry of  $\mathbf{R}_0$  is given by  $r_{ii} = i!$ , which results in imposing a higher regularization penalty weight to higher order functions than lower order ones. Figure 3.4 shows the Parallel Distance Function corresponding to this node distribution, and highlights its value at  $x = 0.42$ , which is  $\rho(0.42) = 1.964 \times 10^{-4}$ . Using these matrices along with the matrix  $\mathbf{P}$  of the nodal function values, the resulting interpolation function value at 0.42 is  $u^h(0.42) = 0.6234$ .

**Table 3.1:** Nodes and Nodal Values

| Node | x-Coordinate | Nodal Value |
|------|--------------|-------------|
| 1    | 0.000000     | 0.879014    |
| 2    | 0.051587     | 0.988912    |
| 3    | 0.102068     | 0.000522    |
| 4    | 0.147115     | 0.865439    |
| 5    | 0.200720     | 0.612566    |
| 6    | 0.241807     | 0.989950    |
| 7    | 0.296200     | 0.527680    |
| 8    | 0.345906     | 0.479523    |
| 9    | 0.402599     | 0.801348    |
| 10   | 0.449929     | 0.227843    |
| 11   | 0.494222     | 0.498094    |
| 12   | 0.549952     | 0.900852    |
| 13   | 0.596551     | 0.574661    |
| 14   | 0.646667     | 0.845178    |
| 15   | 0.704321     | 0.738640    |
| 16   | 0.750567     | 0.585987    |
| 17   | 0.801992     | 0.246735    |
| 18   | 0.854420     | 0.666416    |
| 19   | 0.900901     | 0.083483    |
| 20   | 0.952754     | 0.625960    |
| 21   | 1.000000     | 0.660945    |

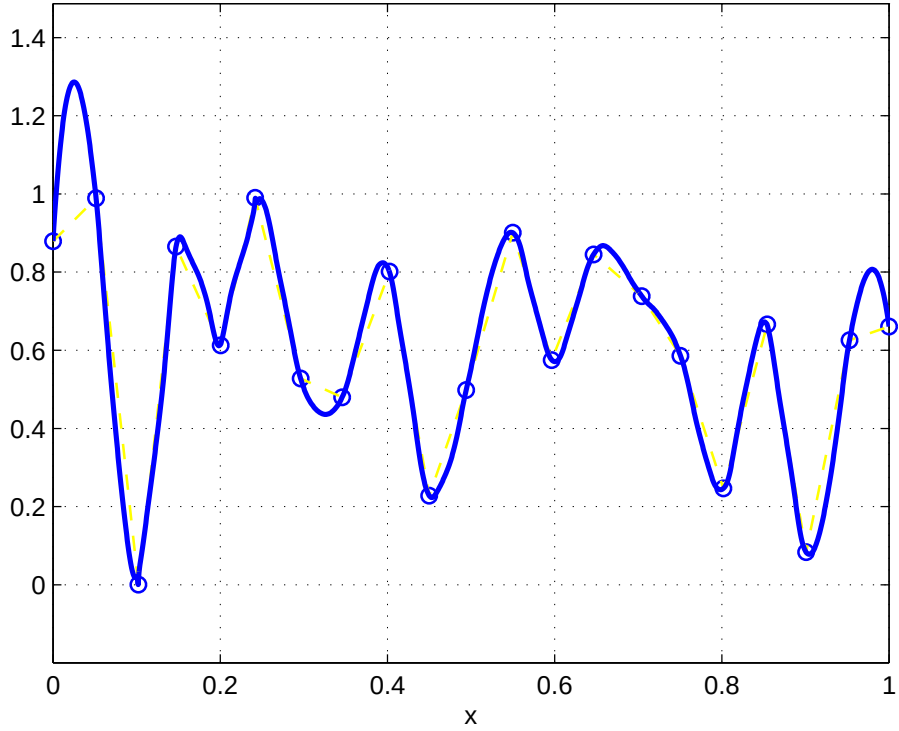


**Figure 3.2:** Moving Regularized Interpolation 1D Example

### 3.4 Consistency and Multiple Stage Interpolation

An interpolation or approximation method is said to have order- $k$  consistency if it represents exactly any function whose  $k^{th}$  derivative is a constant [5]. In Section 2.2, it was shown that the Moving Least-Squares method produces an approximation function that is order- $k$  consistent if all the monomials of orders  $0, \dots, k$  are included in the basis functions used. In the Moving Regularized Interpolation method, if a polynomial  $p_0(\mathbf{x})$  of order smaller than or equal to  $k$  is being interpolated, even if all the monomials of orders  $0, \dots, k$  are included in the basis functions used, the value of the produced interpolation function away from the nodes may deviate from the value of  $p_0(\mathbf{x})$ . This possible deviation is due to inclusion of a penalty on the solution size in the objective function being minimized for evaluating the interpolant at each point.

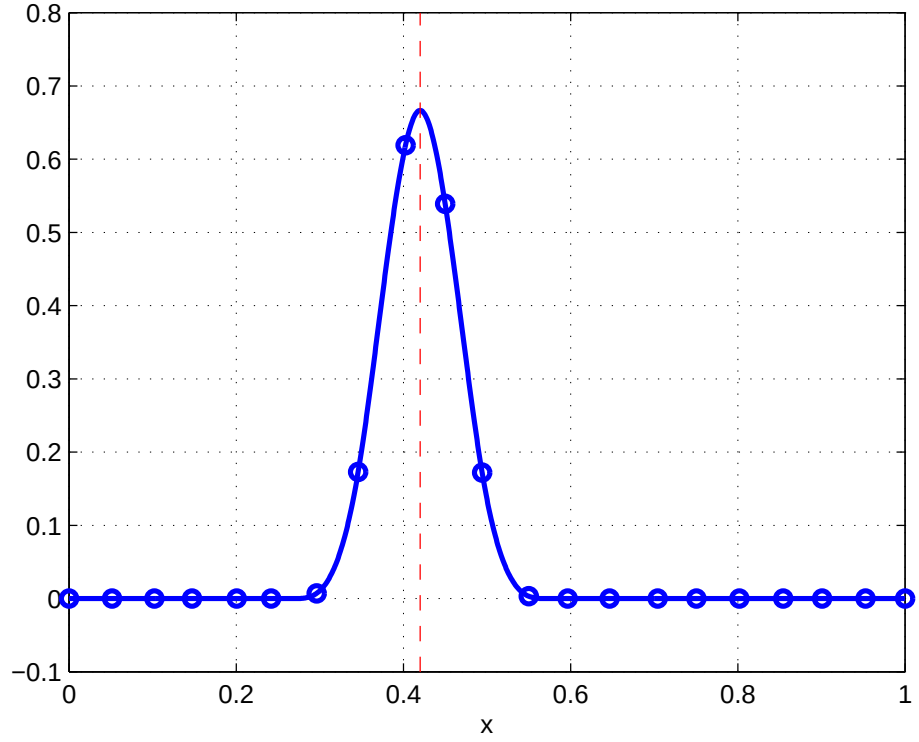


**Figure 3.3:** Window Function as Evaluated at  $x = 0.42$

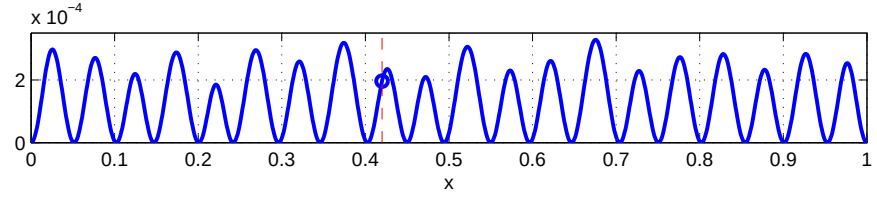


**Figure 3.4:** Parallel Distance Function

The Moving Regularized Interpolation method may be used in a multiple stage interpolation procedure. For instance, let  $\{(\mathbf{x}_i, u_i)\}_{i=1}^N \subset \Omega \times \mathbb{R}$  be a set of ordered pairs of nodes and nodal values for which an interpolant is sought in  $\Omega$ . The interpolation may be carried out in two steps. In the first step, the set is approximated in  $\Omega$  with a Moving Least-Squares function  $u_{MLS}^h$ . In the second step, the errors are interpolated with the Moving Regularized Interpolation method; an interpolant  $u_{MRI}^h$  is produced for the set of ordered pairs  $\{(\mathbf{x}_i, u_i - u_{MLS}^h(\mathbf{x}_i))\}_{i=1}^N$ . The overall interpolant is then obtained by adding the

results of the two stages:  $u^h = u_{MLS}^h + u_{MRI}^h$ . In this case, if the nodal values  $u_i$  are given by a function that can be represented exactly by the Moving Least-Squares method, no errors will remain to be interpolated in the second stage, and therefore the result of the second stage will be identically zero ( $u_{MRI}^h = 0$ ). So in this two-stage procedure, the Moving Least-Squares method guarantees consistency, while the Moving Regularized Interpolation method guarantees exact interpolation.

## 3.5 Solutions of Model Problems

This section presents the use of the Moving Regularized Interpolation method in constructing shape functions for use in a Galerkin method for solving the model problems presented in Section 1.1. The Matlab code listings that carry out these solutions are included in Appendix B.

### 3.5.1 Axially Loaded Elastic Bar

This model problem helps demonstrate the use of one-dimensional Moving Regularized Interpolation shape functions in a Galerkin weak formulation. The governing equation (1.4), gives a mathematical model of the elongation in the bar. For the purpose of this test, the external force applied is specified as:

$$f(x) = x$$

and the boundary conditions imposed on the bar are:

$$u(0) = 0$$

$$u'(1) = 0$$

Furthermore, the material stiffness used is  $k(x) = 1$  throughout the bar. The exact solution to this problem is easily obtained by integration of the governing equation; the exact expression for the displacement of the bar is given by:

$$u(x) = \frac{x}{2} - \frac{x^3}{6} \quad (3.21)$$

The Matlab code listing for the Moving Regularized Interpolation solution to this problem is included in Appendix B.2. Figure 3.5 shows a comparison between the analytical solution and the numerical solution.

### 3.5.2 Plate in Thermal Equilibrium

In this model problem, the Moving Regularized Interpolation method is used to produce two-dimensional shape functions to construct a Galerkin approximation for a thermal equilibrium problem as described in Section 1.1.2.

In this example, the problem domain is:

$$\Omega = \{(x, y) \in \mathbb{R}^2 : 0 < x < 1, \quad 0 < y < 1\}$$

The thermal conductivity is taken to be  $k = 1$ . The boundary conditions modeled are  $T(0, y) = 0$ ;  $k \frac{\partial T}{\partial x} \Big|_{x=1} = 1$ ;  $k \frac{\partial T}{\partial y} \Big|_{y=1} = 1$ ; and the flux is zero on the remaining two boundaries. The external heat influx is zero throughout  $\Omega$ . The exact solution to this problem is a first order polynomial in  $x$ .

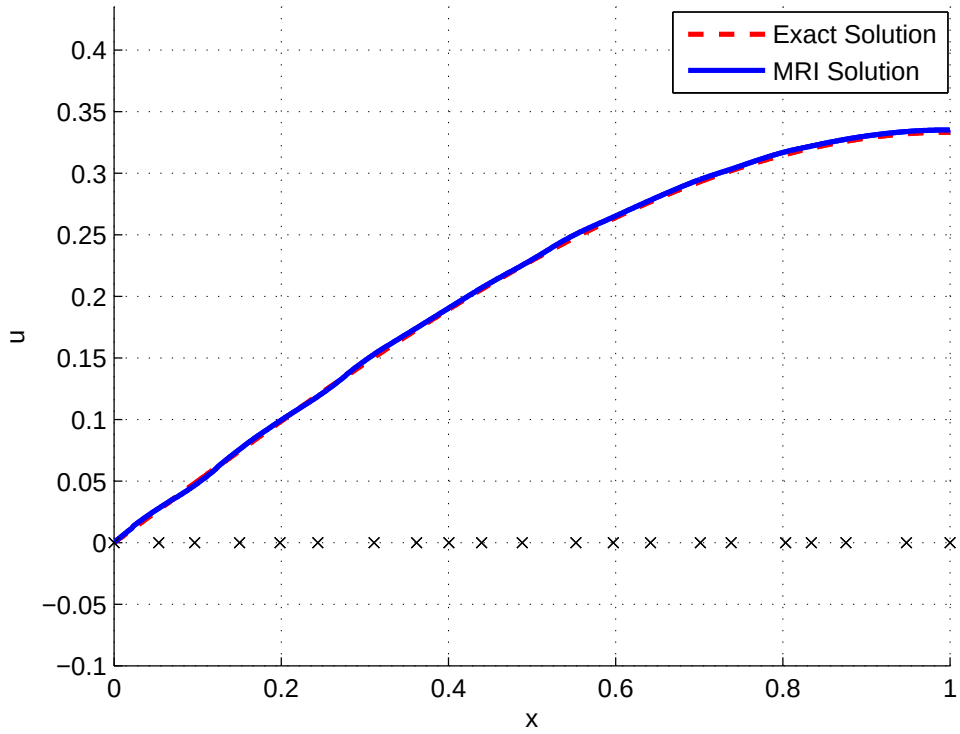


**Figure 3.5:** Displacement of Elastic Bar Under a Tensile Load

The numerical solution, which is shown in Figure 3.6, was computed using the node distribution shown in Figure 3.7. Appendix B.3 lists the Matlab code for this sample solution. The following model problem is a more complex test of the method for solving two-dimensional systems.

### 3.5.3 Cantilever Beam in Plane Stress

In this model problem, the Moving Regularized Interpolation method is used to produce two-dimensional shape functions to solve a plane stress problem. This specific model problem, the cantilever beam, was chosen because it is one of the plane stress problems with known analytical solutions. The exact expressions for the  $x$ - and  $y$ -displacements are given by [32]:

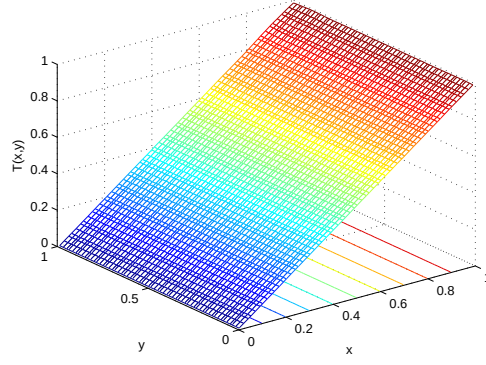


**Figure 3.6:** A Plate in Thermal Equilibrium

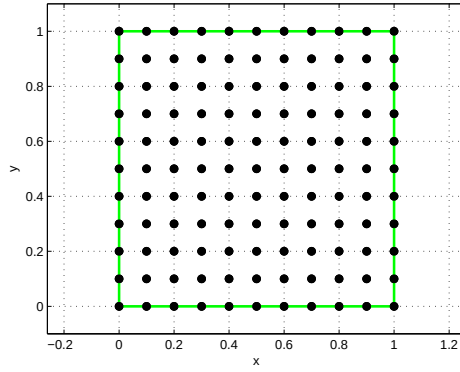


**Figure 3.7:** Node Distribution for Thermal Model Problem

$$u_x = -\frac{P}{6EI} \left( (6L - 3x)x + (2 + \nu) \left( y^2 - \frac{D^2}{4} \right) \right) \quad (3.22)$$

$$u_y = -\frac{P}{6EI} \left( 3\nu y^2(L - x) + (4 + 5\nu) \frac{D^2 x}{4} + (3L - x)x^2 \right) \quad (3.23)$$

where  $P$  is the applied load,  $E$  is the Young's modulus of the material from which the beam is made,  $\nu$  is its Poisson's ratio,  $L$  is the length of the beam and  $D$  is its width. The moment of inertia of the beam,  $I$ , is given by:

$$I = \frac{D^3}{12}$$

In this example, the beam has a Young's modulus  $E = 200\text{GPa}$ , and a Poisson's

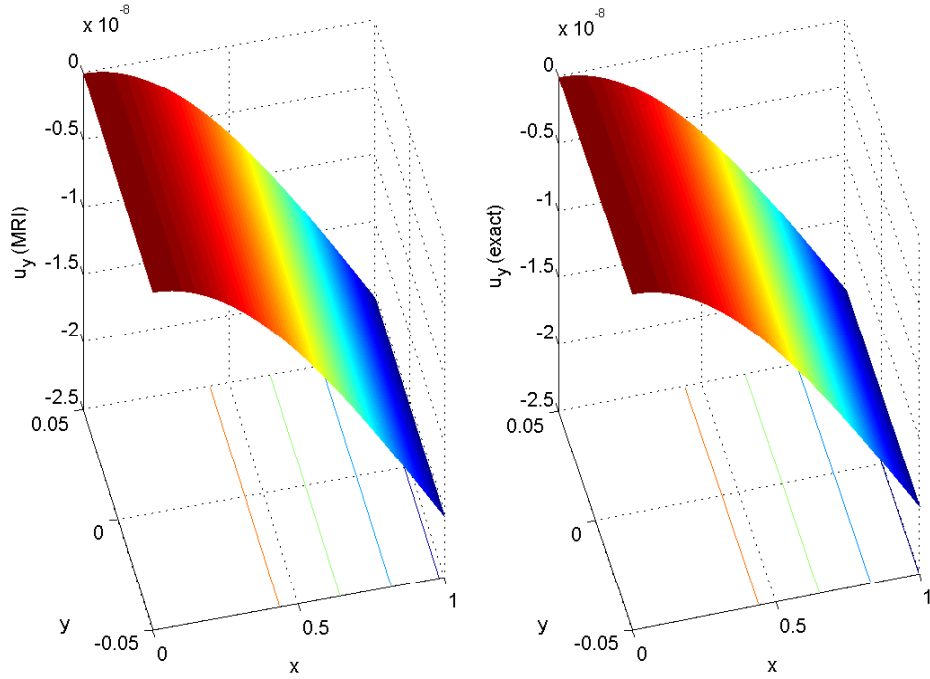


**Figure 3.8:**  $y$ -component of the Displacement (Not to Scale)

ratio  $\nu = 0.3$ . Its length is  $L = 1\text{m}$  and its width is  $D = 0.1\text{m}$ . Figure 3.8 shows the  $y$ -component of the displacement obtained from the exact analytical solution and that computed numerically using the Moving Regularized Interpolation method, and Figure 3.9 shows their  $x$ -components. Figures 3.10 and 3.11 show the displacements at sections with fixed  $y$ -coordinates. The node distribution used in computing the numerical solution is shown in Figure 3.12.

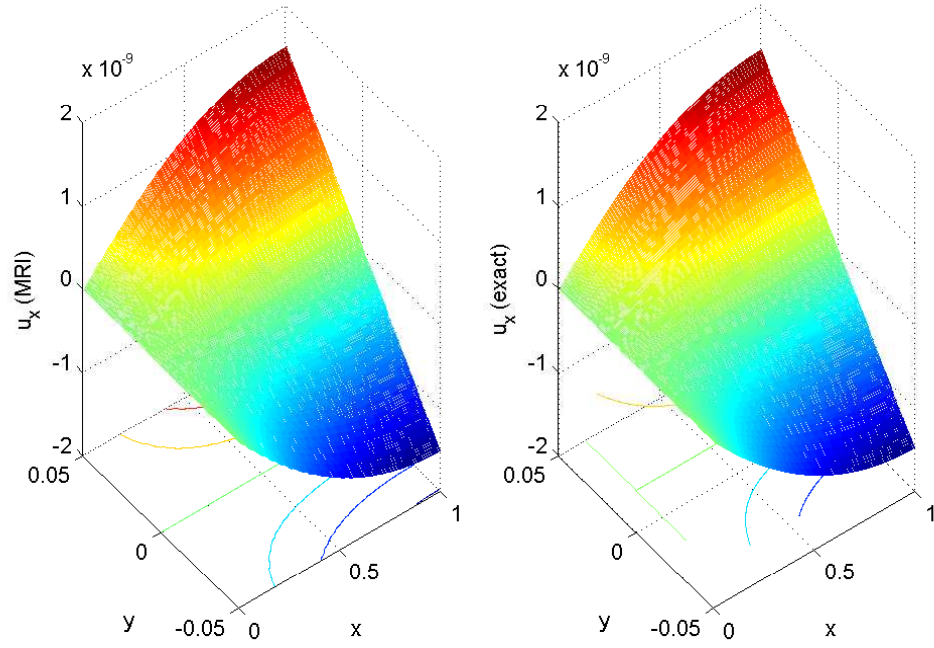


**Figure 3.9:**  $x$ -component of the Displacement (Not to Scale)

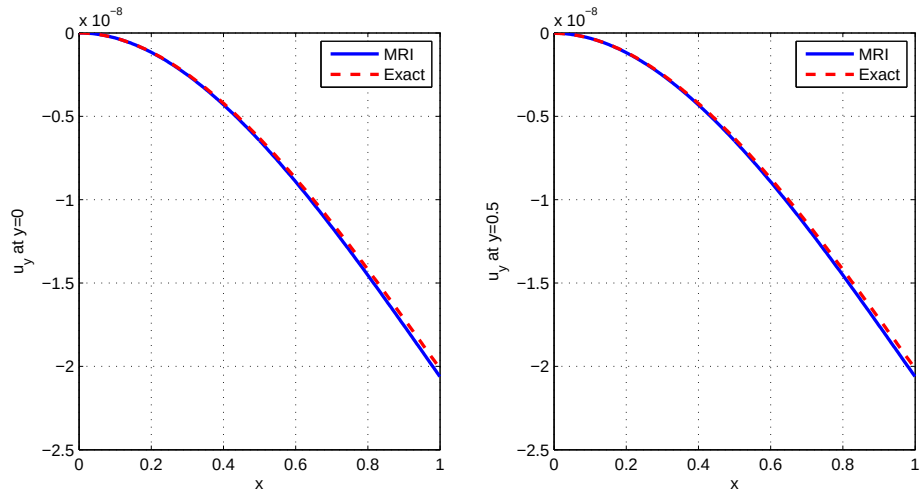


**Figure 3.10:**  $y$ -component of the Displacement at Two Sections

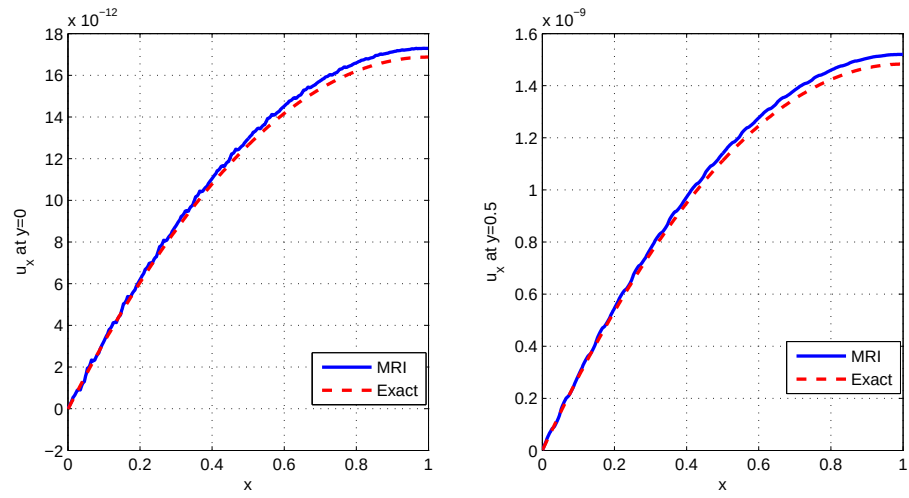


**Figure 3.11:**  $x$ -component of the Displacement at Two Sections

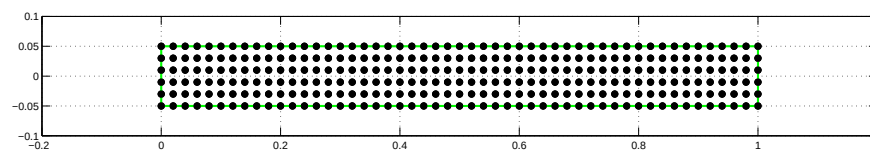


**Figure 3.12:** Node Distribution for Plane-Stress Model Problem

## Chapter 4

### GPU-accelerated EFG Implementation

The use of a numerical method to solve most but the smallest of problems requires a software implementation of that method. In addition to the mathematical description of the numerical method, effective software implementation is governed by the programming language and the hardware platform as well as the nature of the problems that the implementation is intended to solve.

This chapter describes the parallelized implementation of the Element-Free Galerkin method developed in the current work to run on a Graphics Processing Unit. The C++ Code listing of this implementation is included in Appendix B. The following section describes the data structures used, and section 4.2 describes the computation stages for solving the axially loaded elastic bar problem described in subsection 1.1.1.

## 4.1 Data Structures

The problem data was organized into four variable types. These are described in this section.

### 4.1.1 Node Points

The structure of type `MeshNode` describes the nodes distributed in the problem domain to construct the Moving Least-Squares shape functions. The structure contains a pointer to a variable that stores the number of nodes in the domain, a pointer to a variable that stores the radius of the influence domain, which is taken to be the same for all the nodes, and a pointer to an array that stores the coordinates of the nodes.

### 4.1.2 Quadrature Points

The structure of type `QuadraturePoint` describes the quadrature points used for numerical integration. It includes pointers to several variables, whose descriptions follow.

#### Number of Quadrature Points

The variable `*d_num_quad_points` holds a pointer to a single unsigned integer variable that holds the number of quadrature points in the domain.

## **Coordinates**

The variable `*d_coordinates` holds a pointer to an array, each of whose elements holds the coordinates of one of the quadrature points.

## **Quadrature Weights**

The variable `*d_weights` holds a pointer to an array that holds the quadrature weights corresponding to quadrature points in the domain.

## **Contribution Counts**

The variable `*d_contributionCounts` holds a pointer to an array that stores the number of contributions for the quadrature points in the domain. Contributions are described in Subsection 4.1.3. A contribution is a relation between a node point and a quadrature point. A node point contributes to a quadrature point if the shape function corresponding to the node has a nonzero value at the quadrature point. So in the contribution counts array, each element holds the number of nodes in the influence domain of the quadrature point whose coordinates are stored in the corresponding position of the coordinates array.

## **Contribution Indices**

The variable `*d_contributionIndices` holds a pointer to an array that stores the contribution indices for the quadrature points in the domain. These indices are used to access the contributions corresponding to a quadrature point. For

instance, if the contribution index corresponding to quadrature point  $q$  is  $i$ , and the contribution count corresponding to  $q$  is  $n$ , then the elements with indices  $i, (i + 1), \dots, (i + n - 1)$  in the arrays of contribution information correspond to quadrature point  $q$ .

### **Total Count of Contributions**

The variable `*d_total_contributions` holds a pointer to a variable that holds the total number of contributions to all the quadrature points.

### **Interaction Indices**

Interactions are described in Subsection 4.1.4. An interaction is a relation between a quadrature point and a pair of node points. Interactions are used in populating the stiffness matrix. The variable `*d_interactionSetIndices` holds a pointer to an array that stores the interaction set indices for the quadrature points in the domain. Similar to the contribution indices, these variables are used to access the interactions corresponding to a quadrature point.

### **Total Count of Interactions**

The variable `*d_total_interactions` stores a pointer to a variable that holds the total number of interactions corresponding to all the quadrature points.

### Moment Matrix and Its Derivative

The variable `*d_momentMatrix` stores a pointer to an array that holds the elements of the moment matrices corresponding to the quadrature points in the domain. The variable `*d_derivMomentMatrix` stores a pointer to an array that holds the elements of the moment matrix derivatives. Since these matrices are symmetric, only the elements whose column indices are equal to or greater than their row indices are stored. So, if the matrix is  $n \times n$ , only  $\frac{n(n+1)}{2}$  elements are stored, not  $n^2$ .

### Moment Vector and Its Derivative

The variable `*d_momentVector` stores a pointer to an array that holds the elements of the moment vectors corresponding to the quadrature points in the domain. The variable `*d_derivMomentVector` stores a pointer to an array that holds the elements of the moment vector derivatives.

#### 4.1.3 Contributions

A contribution is a relation between a node point and a quadrature point. If the shape function  $\Phi_n$  corresponding to a node point  $\mathbf{x}_n$  has a nonzero value at a quadrature point  $\mathbf{x}_Q$ , then the node point  $\mathbf{x}_n$  contributes to the quadrature point  $\mathbf{x}_Q$ . This term is justified by considering the numerical integration of functions that involve the shape functions, such as approximation functions. For instance, the integral of the meshless approximation function  $\int_{\Omega} u^h d\Omega$  can be approximated for numerical evaluation with a quadrature rule of the form:

$$I = \sum_{q=1}^{n_Q} w_q u^h(\mathbf{x}_q) \quad (4.1)$$

where  $n_Q$  is the number of quadrature points,  $\mathbf{x}_q$  is the  $q^{\text{th}}$  quadrature point, and  $w_q$  is its quadrature weight. If there are only  $m_q$  node points for which the corresponding shape functions  $\{\Phi_{j(k)}\}_{k=1}^{m_q}$  are nonzero at  $\mathbf{x}_q$ , then

$$u^h(\mathbf{x}_q) = \sum_{k=1}^{m_q} \Phi_{j(k)}(\mathbf{x}_q) u_{j(k)} \quad (4.2)$$

where  $u_{j(k)}$  is the nodal parameter corresponding to the node point whose shape function is  $\Phi_{j(k)}$ . So the integral in equation (4.1) is given by:

$$I = \sum_{q=1}^{n_Q} \sum_{k=1}^{m_q} w_q \Phi_{j(k)}(\mathbf{x}_q) u_{j(k)} \quad (4.3)$$

The double summation can be performed in a single summation over all the contributions in the domain.

$$I = \sum_{c=1}^{n_C} w_{q(c)} \Phi_{j(k_c)}(\mathbf{x}_{q(c)}) u_{j(k_c)} \quad (4.4)$$

where

- $n_C$  is the number of contributions
- $\mathbf{x}_{q(c)}$  is the quadrature point corresponding to contribution  $c$
- $w_{q(c)}$  is the quadrature weight of  $\mathbf{x}_{q(c)}$
- $\Phi_{j(k_c)}$  is the shape function corresponding to the node point linked to contribution  $c$
- $u_{j(k_c)}$  is the nodal parameter of the node point corresponding to contribution  $c$

The structure of type `ContributionData` describes the contributions in the problem. It contains pointers to several variables. Descriptions of these variables follow.

### **Mesh Node Index**

The variable `*d_meshNodeAddress` holds a pointer to the index of the mesh node that the current contribution involves.

### **Quadrature Point Index**

The variable `*d_quadPointAddress` holds a pointer to the index of the quadrature point involved in the current contribution.

### **Local Contribution Index**

In addition to its global index, each contribution has a local index which identifies it among the other contributions involving the same quadrature point. The variable `*d_localContributionAddress` holds a pointer to the local index of the current contribution.

### **Distance to Node**

The variable `*d_distanceToNode` holds a pointer to the distance between the quadrature point and the node point involved in the current contribution.

### Window Function Weight Value and Derivative

The variable `*d_weight` holds a pointer to the window function value corresponding to the contribution. The window function is centered at the quadrature point of the contribution, and evaluated at the node point of the contribution. The variable `*d_deriv_weight` holds a pointer to the derivative of the window function corresponding to the contribution.

#### 4.1.4 Interactions

An interaction is a relation between a quadrature point and a pair of, not necessarily distinct, node points. An interaction relates a quadrature point to a pair of node points if the quadrature point is in the supports of both the shape functions  $\Phi_i$  and  $\Phi_j$  corresponding to the node point pair. Interactions are useful in constructing the stiffness matrix of the meshless method. Let the stiffness matrix elements be given by:

$$k_{ij} = \int_{\Omega} \nabla \Phi_i \cdot \nabla \Phi_j d\Omega \quad (4.5)$$

If numerical integration is used to compute  $k_{ij}$ , its form becomes:

$$k_{ij} = \sum_{q=1}^{n_Q} w_q \nabla \Phi_i|_{\mathbf{x}_q} \cdot \nabla \Phi_j|_{\mathbf{x}_q} \quad (4.6)$$

where  $n_Q$  is the number of quadrature points,  $\mathbf{x}_q$  is the  $q^{\text{th}}$  quadrature point and  $w_q$  is its quadrature weight. For meshless shape functions with local support,

it is not necessary to include in this sum any quadrature points that lie outside the support of  $\Phi_i$  or outside the support of  $\Phi_j$ . Similarly, it is not necessary to compute  $k_{ij}$  if the supports of  $\Phi_i$  and  $\Phi_j$  do not intersect. As a consequence, in computing the stiffness matrix, it is only necessary to add terms of the form in equation (4.6) where the quadrature point  $\mathbf{x}_q$  is in the supports of both the shape functions  $\Phi_i$  and  $\Phi_j$ . It is hence possible to construct the stiffness matrix by adding up all the terms corresponding to such combinations of quadrature points and pairs of shape functions. In this procedure, each term must be added to the partial sum pertaining to the element  $k_{ij}$  whose indices  $i$  and  $j$  correspond to those of the shape function pair.

In the current implementation, interactions are used to index a collection of computations each of which evaluates a term of the form in equation (4.6), and adds it to the corresponding element of the stiffness matrix. The interactions are not explicitly stored in memory. Instead, they are computed from the quadrature points and their contribution sets. If a quadrature point has  $n$  contributions, then it is involved in  $\frac{n(n+1)}{2}$  interactions. Of these,  $\frac{n(n-1)}{2}$  interactions relate the quadrature point to pairs of distinct node points, and  $n$  interactions relate it to pairs of node points in each of which one node point accounts for both the elements.

## 4.2 Computation Stages

In this GPU implementation of the Element-Free Galerkin method, computation is divided into several stages. Each computation stage uses outputs from previous stages and processes it for use in later stages. The stages are: pre-processing, initialization of moment matrices, solution of moment matrix linear systems, and population of the load vector and stiffness matrix to compute

the displacements in the bar. These computation stages are described in the following subsections.

### 4.2.1 Preprocessing

In the preprocessing stage, memory is allocated for the data structures involved in the remaining stages, the problem nodes and quadrature points are initialized, and the contributions data structure is populated. Following are descriptions of these procedures.

#### Memory Allocation

The computation starts with the allocation of memory for the nodes, the quadrature points, the stiffness matrix and the load vector. The nodes memory is allocated by the function call ‘allocateNodeMemory(&mNode, num\_nodes)’, and the function call ‘allocateQuadPointMemory(&qPoint, num\_quad\_points)’ is used for allocating the quadrature points memory. The stiffness matrix and load vector memory is allocated using the directive ‘cudaMalloc’. Memory is also allocated for the contribution memory, but only when the number of contributions becomes available, after initialization of the quadrature structure is complete. Allocation of contribution memory is done by the function call ‘allocateContributionMemory(&contributionInfo, p\_num\_contributions[0])’.

### Initialization of Nodes and Quadrature Structure

After allocation of memory, the data structures of the nodes and quadrature points are populated. The main part of the node initialization is population of the coordinates array. This is performed on the GPU by running the function ‘computeNodeLocations’. Each thread that runs this function computes the coordinate of one node and saves it to the coordinates array. The thread with index 0 also populates the scalars corresponding to the influence domain radius and the number of nodes.

Initialization of the quadrature structure is performed by calling the GPU function ‘computeQuadratureStructure’. It is done in three stages: computing the quadrature point coordinates, counting the contributions for each quadrature point, and assigning the contribution and interaction indices of each quadrature point. Also part of initializing the quadrature structure is computing the total numbers of contributions and interactions. These numbers are used to determine the size of the memory required for storing the contributions data structure, and the numbers of threads required for processing contributions and interactions. The number of contributions for each quadrature point is computed by iterating over all the nodes in the domain; if a node contributes to the quadrature point at hand, the contribution count is incremented by one. The number of interactions involving each quadrature point is computed from its number of contributions.

### Computation of Contribution Data

After the nodes and quadrature structure have been initialized, the following step is computation of contribution data. This computation is performed by

calling the GPU function ‘populateContributionData’. One thread corresponding to each quadrature point executes this function. It iterates over the node points in the domain, and for each node that contributes to the current quadrature point, a contribution linking it to the quadrature point is populated.

### 4.2.2 Computation of Moment Matrices

In the expression for the Moving Least-Squares approximation function given in equation (2.18), the matrix  $(\mathbf{P}^\top \mathbf{W} \mathbf{P})$  is called the moment matrix. For computation purposes, it is useful to represent the matrix as a weighted sum of diads. Let  $\mathbf{p}_i^\top$  represent the  $i^{\text{th}}$  row of  $\mathbf{P}$ , so that

$$\mathbf{P}^\top \mathbf{W} \mathbf{P} = \begin{bmatrix} \mathbf{p}_1 & \mathbf{p}_2 & \cdots & \mathbf{p}_n \end{bmatrix} \begin{bmatrix} w_1 & 0 & \cdots & 0 \\ 0 & w_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & w_n \end{bmatrix} \begin{bmatrix} \mathbf{p}_1^\top \\ \mathbf{p}_2^\top \\ \vdots \\ \mathbf{p}_n^\top \end{bmatrix} \quad (4.7)$$

which is equivalent to

$$\mathbf{P}^\top \mathbf{W} \mathbf{P} = \sum_{i=1}^n w_i \mathbf{p}_i \mathbf{p}_i^\top \quad (4.8)$$

Each term in this sum corresponds to one of the problem nodes. Normally, the window function has a small support and, consequently, only a small number of the diagonal entries  $w_i$  are nonzero. In computing the moment matrix corresponding to a quadrature point  $\mathbf{x}_q$ , the nonzero entries of  $\mathbf{W}$  are associated with

the nodes linked to  $\mathbf{x}_q$  with contributions. So if a zero matrix of the same size as the moment matrix is created for each quadrature point, and then the term  $w_i \mathbf{p}_i \mathbf{p}_i^\top$  is computed for each contribution and added to the running sum of the quadrature point involved in the contribution, the moment matrices of all the quadrature points will result. This procedure is used in computing the moment matrices in the current implementation. The derivatives of the moment matrices are computed in the same way; the only exception being that the entries of the diagonal matrix are the derivatives of the window function.

The computation is performed by calling the GPU function ‘processMomentMatrices’. Each GPU thread corresponds to a contribution. Because the moment matrix is symmetric, only one half of the off-diagonal entries are computed and stored.

### 4.2.3 Solution of Moment Matrix Linear Systems

After the moment matrices and their derivatives are computed, the linear system described in equation (2.20) is solved as an intermediate step in evaluating the expression of the Moving Least-Squares functions given in equation (2.19). This is done by launching the GPU function ‘processMomentVectors’ in threads each of which computes the vector  $\mathbf{c}$  and its derivative for one quadrature point. The function ‘completeLdlSolution’ is used, which calls functions that perform LDL decomposition, back substitution, and then forward substitution. Implementations of these functions are also included in the code included in Appendix B.5.

### 4.2.4 Population of the Load Vector

The  $i^{\text{th}}$  entry of the load vector is given by:

$$f_i = \int_{\Omega} \Phi_i(\mathbf{x}) f(\mathbf{x}) d\Omega \quad (4.9)$$

Approximating this integral by numerical quadrature gives:

$$f_i = \sum_{q=1}^{n_Q} w_q \Phi_i(\mathbf{x}_q) f(\mathbf{x}_q) \quad (4.10)$$

where  $n_Q$  is the number of quadrature points,  $\mathbf{x}_q$  is the  $q^{\text{th}}$  quadrature point, and  $w_q$  is its quadrature weight. These values can be computed more efficiently by iterating over the contributions, computing, one at each iteration, the terms of the form in equation (4.10), adding each term to the load vector entry that corresponds to the shape function of the node linked to the contribution, substituting for  $\mathbf{x}_q$  the quadrature point linked to the contribution. This procedure is similar to that described in Subsection 4.1.3 on page 67. It is performed by calling the GPU function ‘populateLoadVector’. Each thread running this function processes one contribution, adding one of the terms to the corresponding entry of the load vector.

### 4.2.5 Population of the Stiffness Matrix

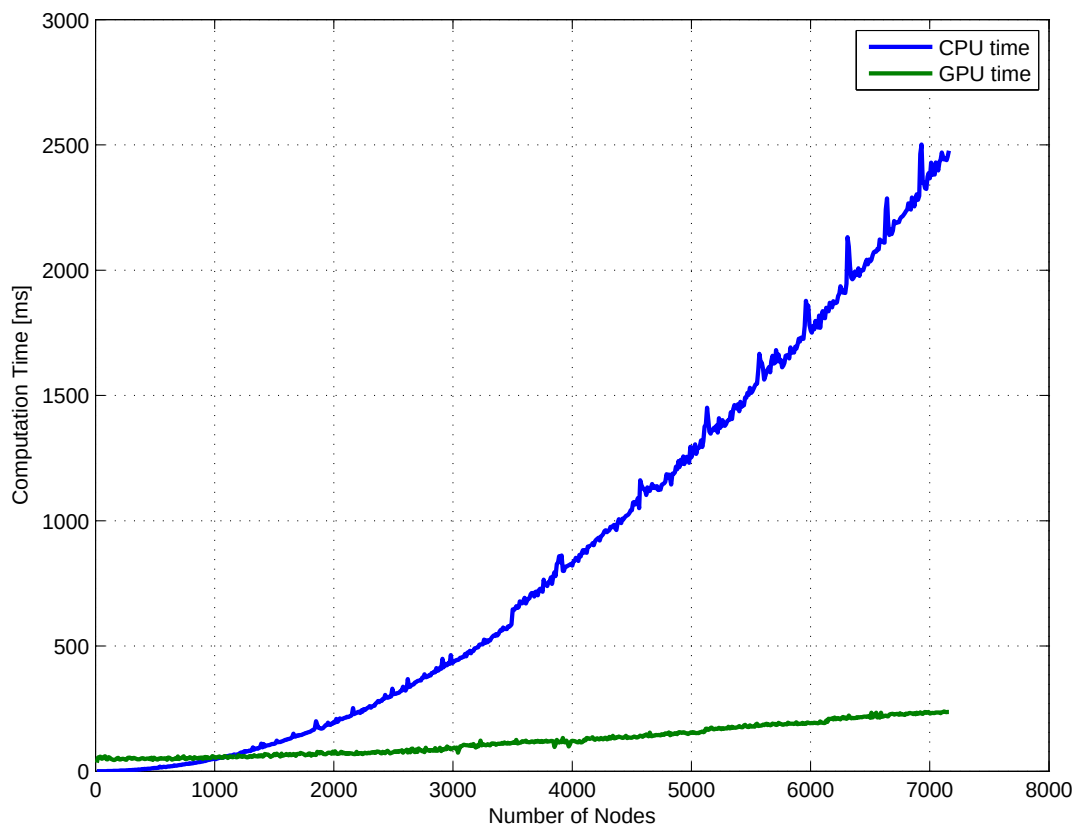
The stiffness matrix is populated using the procedure described to demonstrate the use of interactions in Subsection 4.1.4 on page 70. It is executed by calling

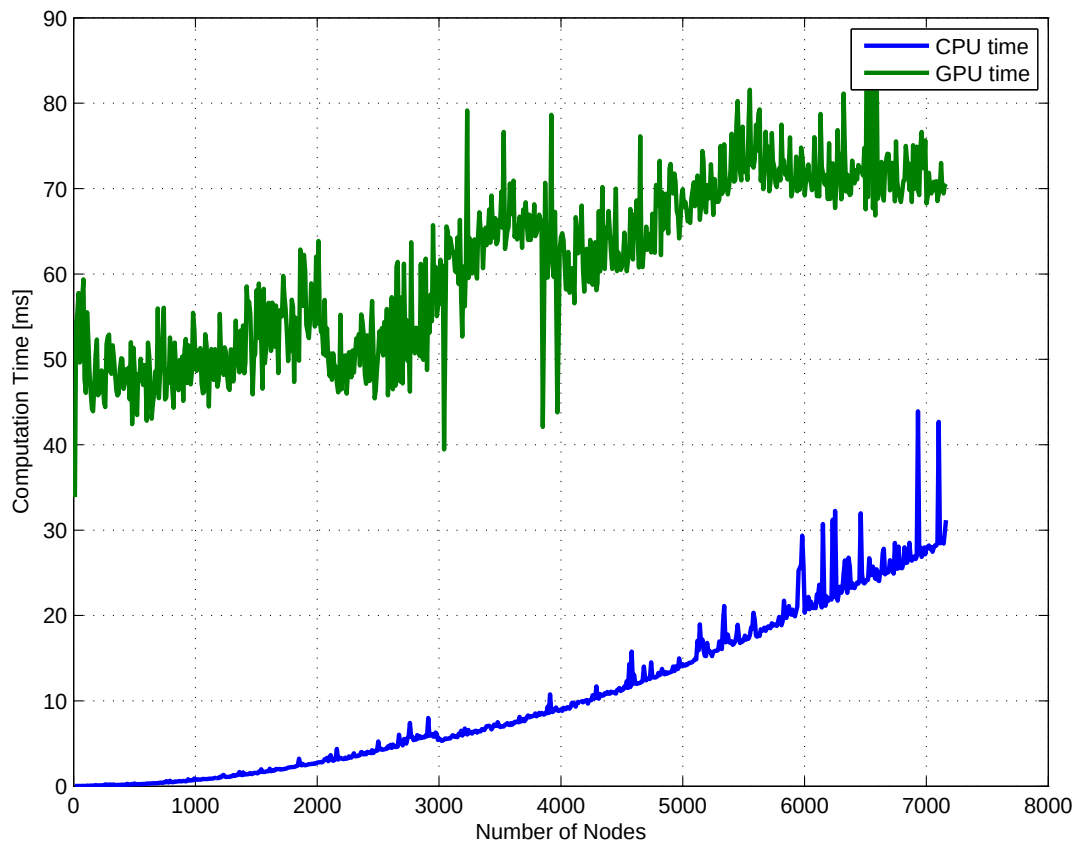
the GPU function ‘populateStiffnessMatrix’, launching one thread for each interaction, each thread adding one term to the corresponding entry of the stiffness matrix.

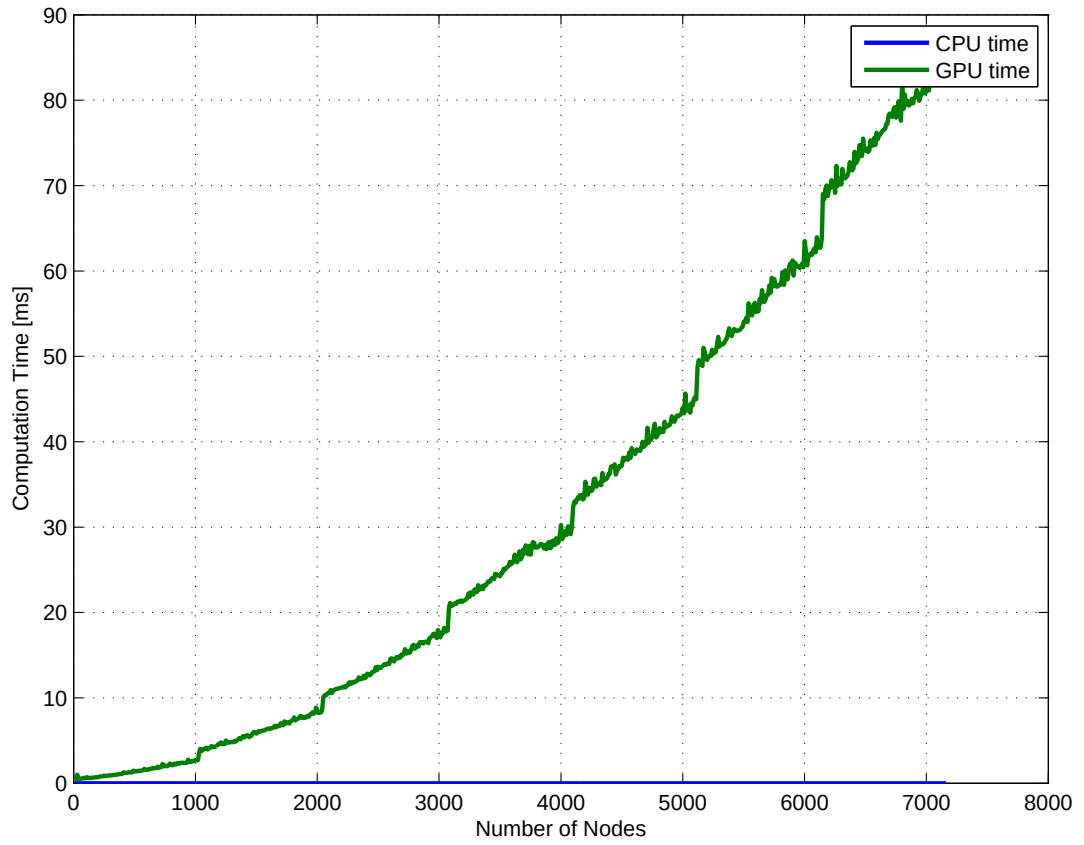
## 4.3 Performance Results

To evaluate the performance benefit of using the Graphics Processing Unit in this implementation, a CPU version of the code was created by modifying the GPU implementation to run on the CPU alone (without GPU acceleration). The run-times that the two implementations required for performing the various computation stages were measured for various problem sizes. The problem size was varied by varying the number of nodes and quadrature cells used to solve the problem. The GPU-accelerated implementation of the final step of the solution, solving the linear system represented by the stiffness matrix and the load vector, was not investigated in this work, because solution of dense and sparse linear systems has been well documented [33, 34, 35, 36].

Figure 4.1 shows a comparison between the time taken by GPU and CPU codes to perform the entire computation for problems with various sizes. For a problem with around 7000 nodes, the GPU-accelerated implementation performs the computation approximately 10 times faster than the CPU code. The time comparisons for the individual computation stages are shown in Figures 4.2 to 4.10. Memory management, which involves memory allocation and release, required more time in the GPU implementation than it did in the serial implementation, as shown in Figure 4.2. All the remaining computation tasks were completed faster in the GPU implementation, with the exception of data transfer between the GPU memory and the CPU memory, which is not required in the CPU implementation.

**Figure 4.1:** Total Computation Time Comparison

**Figure 4.2:** Memory Management Time Comparison

**Figure 4.3:** Data Transfer Time Comparison

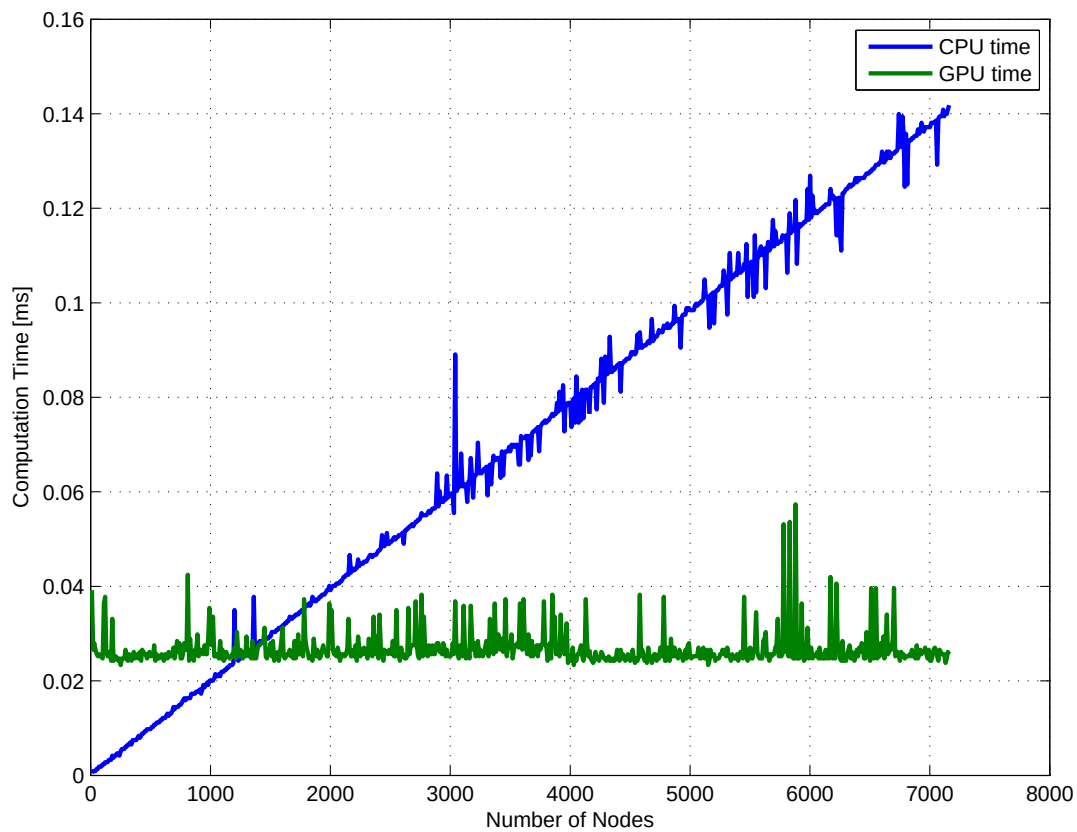


**Figure 4.4:** Node Processing Time Comparison

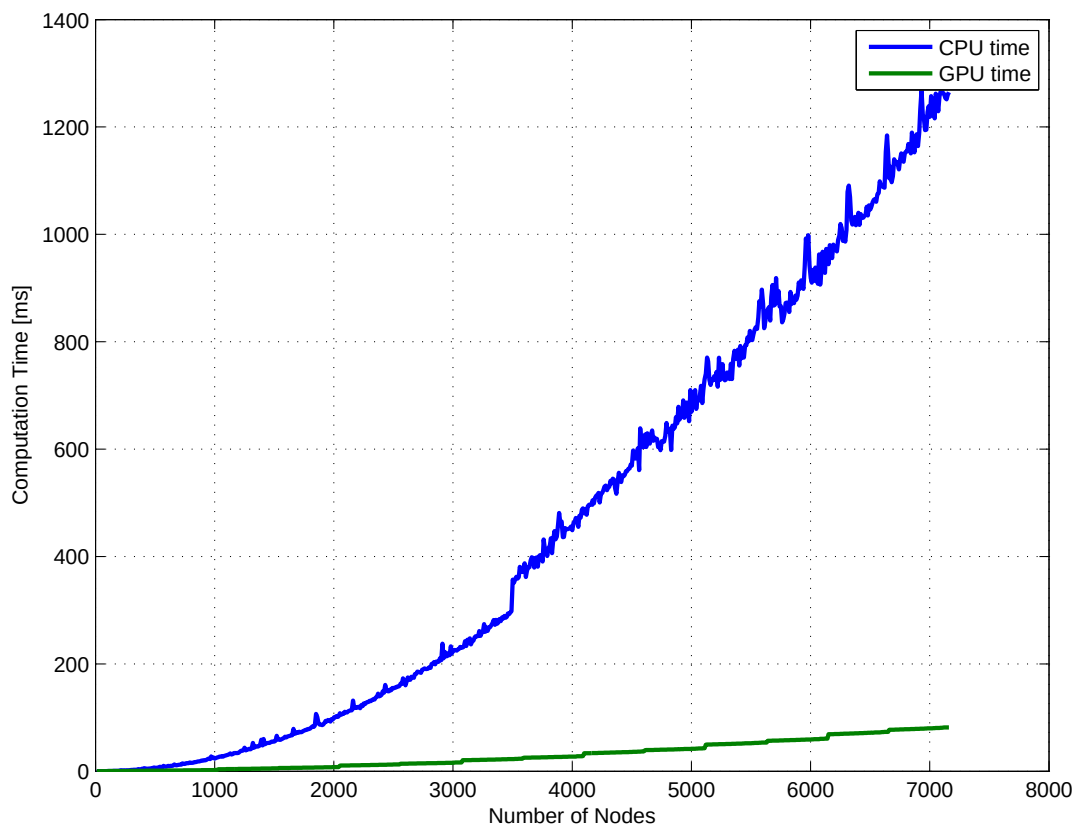


**Figure 4.5:** Quadrature Structure Processing Time Comparison

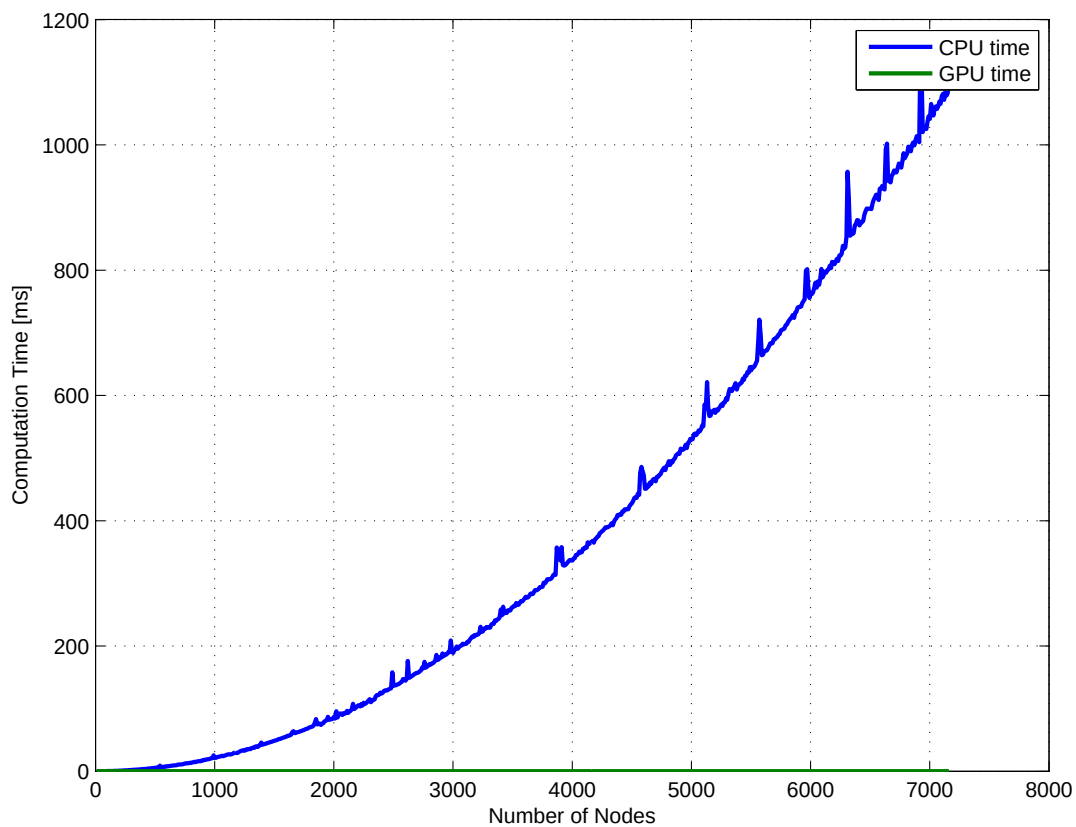


**Figure 4.6:** Contribution Data Processing Time Comparison

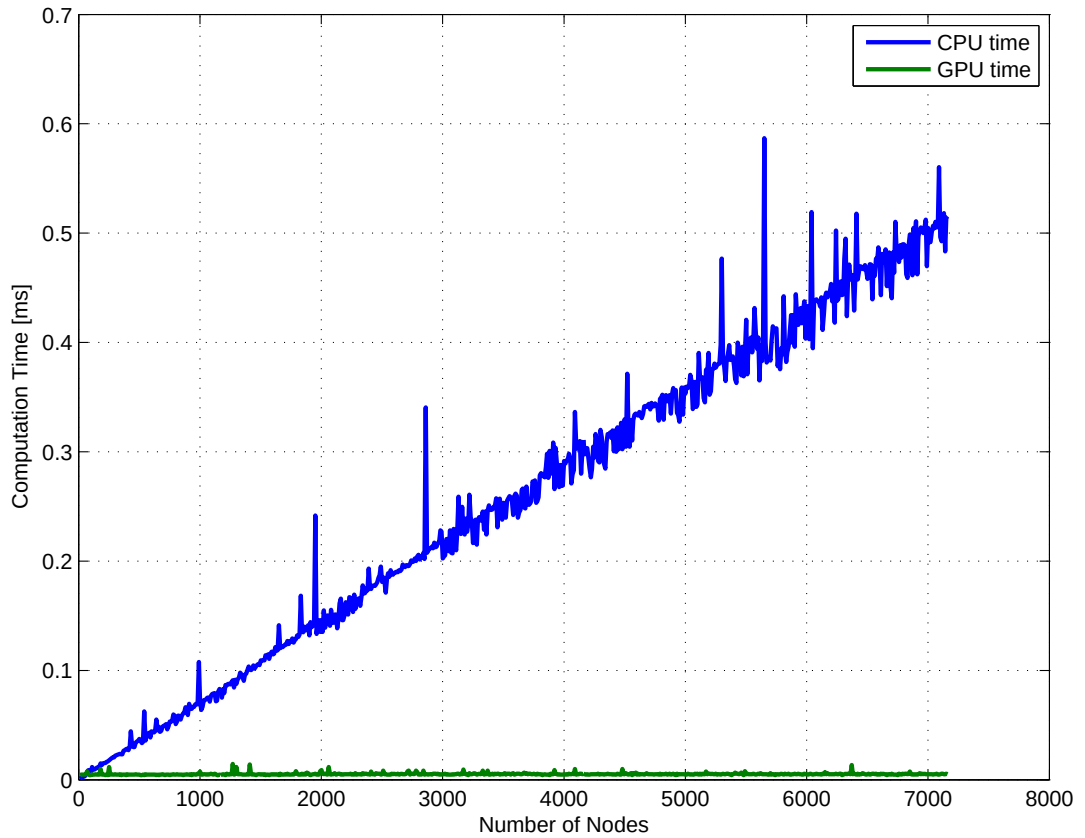


**Figure 4.7:** Moment Matrices Processing Time Comparison

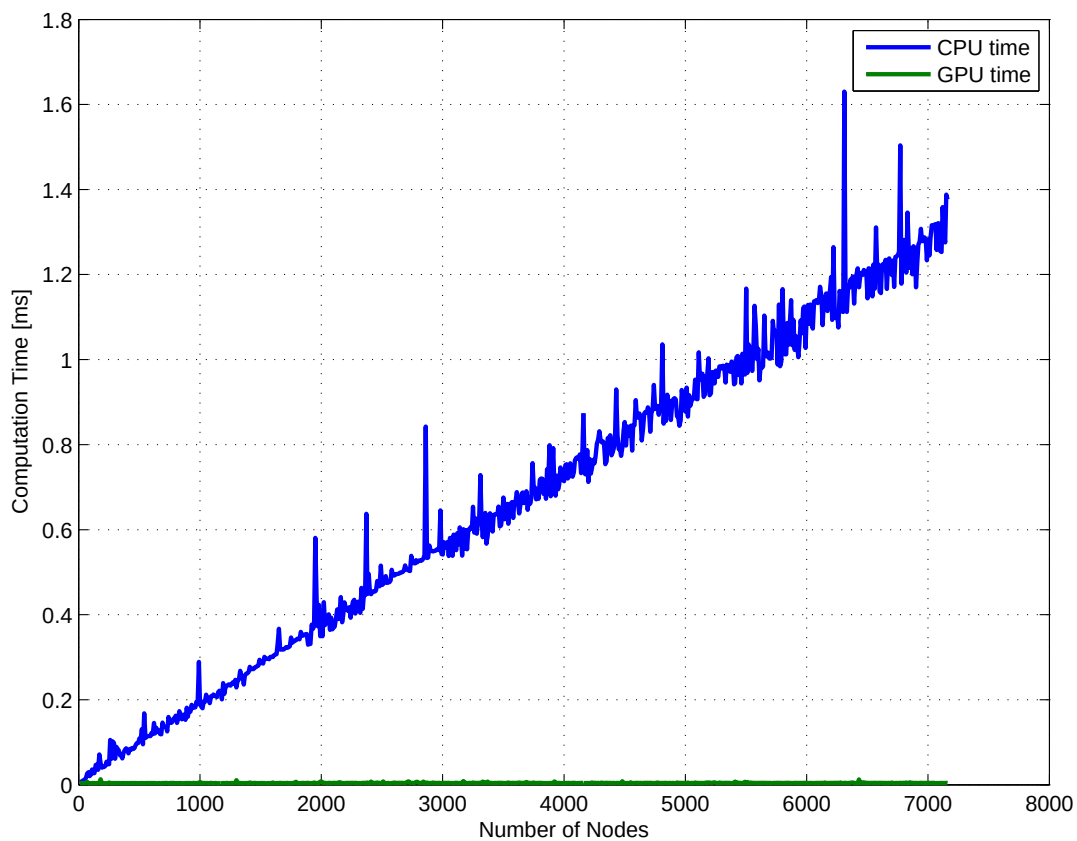


**Figure 4.8:** Correction Vectors Processing Time Comparison

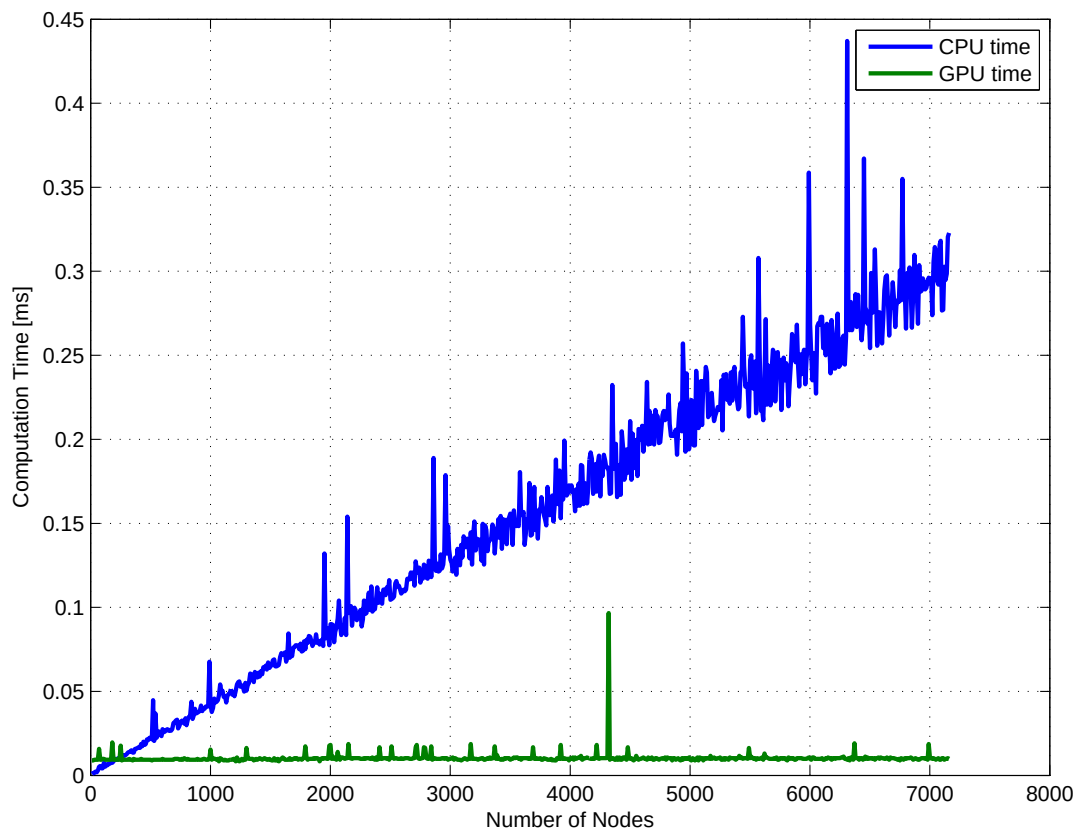


**Figure 4.9:** Load Vector Population Time Comparison

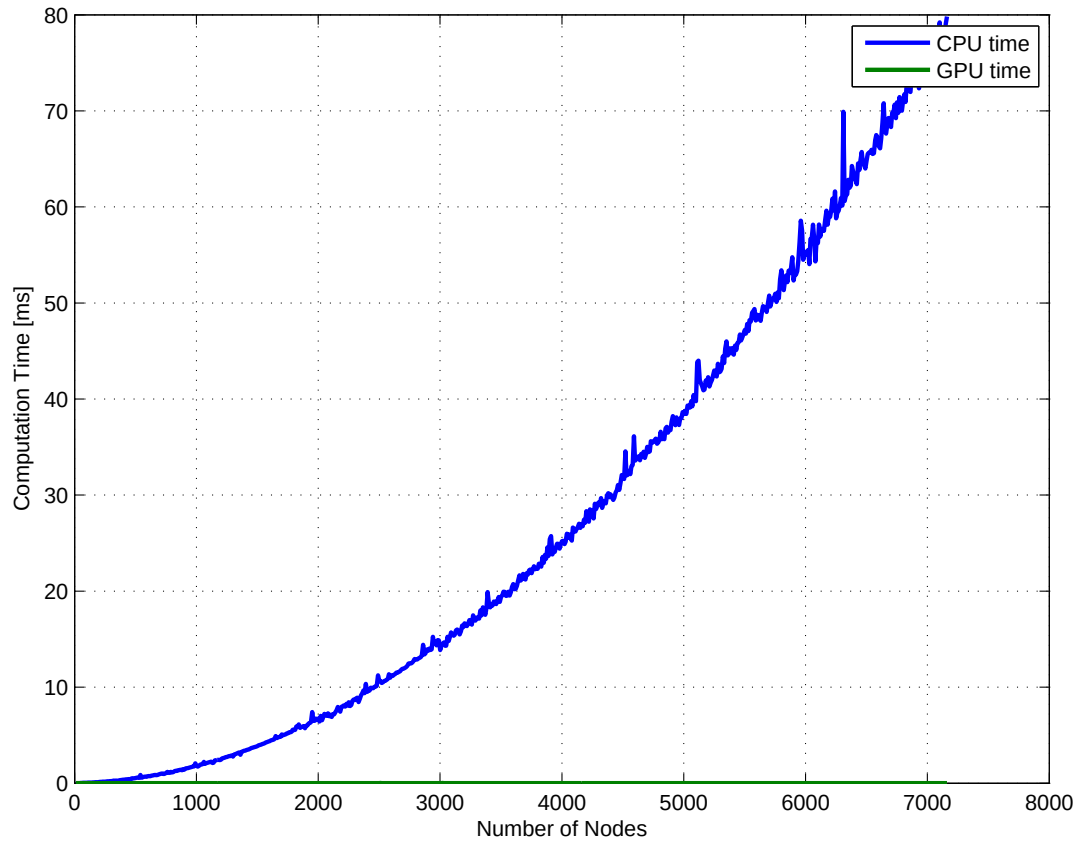


**Figure 4.10:** Stiffness Matrix Population Time Comparison

## Chapter 5

### Discussion and Future Work

The Moving Regularized Interpolation method, a meshless method that achieves exact interpolation, was developed and demonstrated on an interpolation problem and in a Galerkin method for solving some one- and two-dimensional test problems. The method relies on mapping to each point a uniquely solvable problem. The value of the interpolant at that point is then computed by first obtaining the function that solves that problem, and then evaluating that function at that point. The uniquely solvable problems mapped to the nodes are constructed such that their solutions interpolate all the nodes in the domain of influence. Away from the nodes, regularization ensures continuity of the resulting interpolants. Numerical results produced by the method for one- and two-dimensional problems accurately reproduced the analytically computed results, showing that the method is effective in solving partial differential equations.

In addition, an implementation of the Element-Free Galerkin method that makes use of a Graphics Processing Unit was presented. It was compared to a similar implementation that runs exclusively on the Central Processing Unit. This comparison showed that GPU-acceleration was effective in enhancing the speed of computation in using the meshless method. Nevertheless, it may be possible

to improve the performance of both implementations with additional optimization.

Although the GPU-accelerated implementation was of a one-dimensional Element-Free Galerkin method, it can readily be modified to solve two- and three-dimensional problems and to employ other meshless methods, such as the Moving Regularized Interpolation method. This is because the use of contributions and interactions to simplify and speed up computation of the stiffness matrix and load vector is applicable to all meshless methods that use a background integration grid, and it is applicable to two- and three-dimensional problems as well as one-dimensional ones. For the same reason, the architecture of the current implementation makes it possible to employ more than one meshless method in the same implementation. One example of such combination of meshless methods is using two or more methods in multiple stage interpolation, which was briefly described in Section 3.4. This area of intersection of the mathematical theory and computer implementation of meshless methods is worthy of further investigation in future research.

## 5.1 Future Work

Some important aspects of the introduced Moving Regularized Interpolation method may be investigated in future research projects. Foremost is the convergence behaviour of the method as the node density is increased. Another closely related question is that of the convergence of the Galerkin method when Moving Regularized Interpolants are employed, and its robustness against node distribution irregularities. Another aspect for future research is the use of the Moving Regularized Interpolation in adaptive Galerkin schemes.

The developed GPU-accelerated Element-Free Galerkin method implementation is worthy of further development and use in future projects. Expansion and testing of the implementation on problems in two- and three-dimensions may be followed by introduction of additional parallelization strategies, such as using multiple computing nodes that communicate through the Message Passing Interface. Most important for its use in future projects, however, is that the GPU-accelerated implementation be restructured as a solver module that can be integrated with pre- and post-processors and possibly a graphical user interface to form a complete package for solving Partial Differential Equations.

## Bibliography

- [1] G. Strang. A framework for equilibrium equations. *SIAM Review*, 30(2):283–297, June 1988.
- [2] Y.A. Çengel and R.H. Turner. *Fundamentals of Thermal-Fluid Sciences*. McGraw Hill, New York, NY, 2001.
- [3] R.D. Cook. *Finite Element Modeling for Stress Analysis*. John Wiley & Sons, New York, NY, 1995.
- [4] J. Dolbow and T. Belytschko. An introduction to programming the meshless Element-Free Galerkin method. *Archives of Computational Methods in Engineering*, 5(3):208–241, 1998.
- [5] T. Belytschko, Y. Krongauz, D. Organ, M. Fleming, and P. Krysl. Meshless methods: an overview and recent developments. *Computer Methods in Applied Mechanics and Engineering*, 139(1):3–47, 1996.
- [6] V.P. Nguyen, T. Rabczuk, S. Bordas, and M. Duflot. Meshless methods: a review and computer implementation aspects. *Mathematics and Computers in Simulation*, 79(3):763–813, 2008.

- [7] G.R. Liu. *Mesh Free Methods: Moving Beyond the Finite Element Method*. CRC Press LLC, 2003.
- [8] L.B. Lucy. A numerical approach to the testing of the fission hypothesis. *The Astronomical Journal*, 82:1013–1024, 1977.
- [9] R.A. Gingold and J.J. Monaghan. Smoothed Particle Hydrodynamics - theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society*, 181:375–389, 1977.
- [10] D. Shepard. A two-dimensional interpolation function for irregularly-spaced data. In *Proceedings of the 1968 23rd ACM National Conference*, pages 517–524. ACM, 1968.
- [11] D.H. McLain. Drawing contours from arbitrary data points. *The Computer Journal*, 17(4):318–324, 1974.
- [12] W.J. Gordon and J.A. Wixom. Shepard’s method of “metric interpolation” to bivariate and multivariate interpolation. *Mathematics of Computation*, 32(141):253–264, 1978.
- [13] R.E. Barnhill. Representation and approximation of surfaces. *Mathematical Software*, 3:69–120, 1977.
- [14] P. Lancaster and K. Salkauskas. Surfaces generated by Moving Least-Squares methods. *Mathematics of Computation*, 37(155):141–158, July 1981.
- [15] T. Belytschko, Y. Krongauz, M. Fleming, D. Organ, and W.K. Liu. Smoothing and accelerated computations in the Element-Free Galerkin

- method. *Journal of Computational and Applied Mathematics*, 74(1):111–126, 1996.
- [16] G.H. Golub and C.F. Van Loan. *Matrix computations*, volume 3. Johns Hopkins University Press, 1996.
- [17] I. Babuska and J.M. Melenk. The Partition of Unity Finite Element Method. Technical report, DTIC Document, 1995.
- [18] T. Belytschko, Y.Y. Lu, and L. Gu. Element-Free Galerkin Methods. *International Journal for Numerical Methods in Engineering*, 37(2):229–256, 1994.
- [19] T. Zhu and S.N. Atluri. A modified collocation method and a penalty formulation for enforcing the essential boundary conditions in the Element-Free Galerkin method. *Computational Mechanics*, 21(3):211–222, 1998.
- [20] Y.Y. Lu, T. Belytschko, and L. Gu. A new implementation of the Element-Free Galerkin method. *Computer Methods in Applied Mechanics and Engineering*, 113(3):397–414, 1994.
- [21] T. Belytschko and M. Tabbara. Dynamic fracture using Element-Free Galerkin methods. *International Journal for Numerical Methods in Engineering*, 39(6):923–938, 1996.
- [22] T. Most and C. Bucher. A Moving Least-Squares weighting function for the Element-Free Galerkin method which almost fulfills essential boundary conditions. *Structural Engineering and Mechanics*, 21(3):315–332, 2005.
- [23] T. Most and C. Bucher. New concepts for Moving Least Squares: An inter-

- polating non-singular weighting function and weighted nodal least squares. *Engineering Analysis with Boundary Elements*, 32(6):461–470, 2008.
- [24] G.R. Liu and Y.T. Gu. A point interpolation method for two-dimensional solids. *International Journal for Numerical Methods in Engineering*, 50(4):937–951, 2001.
- [25] R.R. Schaller. Moore’s law: past, present and future. *Spectrum, IEEE*, 34(6):52–59, 1997.
- [26] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to parallel computing*, volume 110. Benjamin/Cummings USA, 1994.
- [27] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov. The impact of multicore on math software. *Applied Parallel Computing. State of the Art in Scientific Computing*, pages 1–10, 2007.
- [28] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.
- [29] J. Nickolls and W.J. Dally. The GPU computing era. *Micro, IEEE*, 30(2):56–69, 2010.
- [30] G.H. Golub, P.C. Hansen, and D.P. O’Leary. Tikhonov regularization and total least squares. *SIAM Journal on Matrix Analysis and Applications*, 21(1):185–194, 1999.
- [31] J.R. Magnus and H. Neudecker. *Matrix Differential Calculus with Applications in Statistics and Econometrics*. Wiley, New York, 1988.

- [32] S.P. Timoshenko. *Theory of Elasticity*. McGraw-Hill, 1951.
- [33] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Parallel & Distributed Processing, 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010.
- [34] V. Volkov and J. Demmel. LU, QR and Cholesky factorizations using vector capabilities of GPUs. *EECS Department, University of California, Berkeley, Technical Report, UCB/EECS-2008-49, May*, pages 2008–49, 2008.
- [35] P. Stpiczynski. Solving linear recurrences on hybrid GPU-accelerated many-core systems. In *Computer Science and Information Systems (FedCSIS), 2011 Federated Conference on*, pages 465–470. IEEE, 2011.
- [36] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU-accelerated manycore systems. *Parallel Computing*, 36(5):232–240, 2010.
- [37] H. Anton and C. Rorres. *Elementary Linear Algebra with Applications*. John Wiley & Sons, New York, NY, 2005.

# Appendix A

## Some Concepts from Linear Algebra

### A.1 Vector Space

A vector space over a scalar field  $\mathbf{K}$  is a set  $\mathbf{V}$  on which vector addition and scalar multiplication are defined such that the following conditions are satisfied for all vectors  $\mathbf{u}, \mathbf{v}, \mathbf{w} \in \mathbf{V}$  and all scalars  $s, t \in \mathbf{K}$ :

- $(\mathbf{u} + \mathbf{v}) \in \mathbf{V}$
- $\mathbf{u} + \mathbf{v} = \mathbf{v} + \mathbf{u}$
- $\mathbf{u} + (\mathbf{v} + \mathbf{w}) = (\mathbf{u} + \mathbf{v}) + \mathbf{w}$
- there is an element  $\mathbf{0} \in \mathbf{V}$  such that  $\mathbf{v} + \mathbf{0} = \mathbf{v}$  for any  $\mathbf{v} \in \mathbf{V}$
- for any element  $\mathbf{v} \in \mathbf{V}$  there is an element  $-\mathbf{v} \in \mathbf{V}$  such that  $\mathbf{v} + (-\mathbf{v}) = \mathbf{0}$
- for any scalar  $s \in \mathbf{K}$  and any vector  $\mathbf{v} \in \mathbf{V}$ ,  $s \cdot \mathbf{v} \in \mathbf{V}$
- $s \cdot (\mathbf{u} + \mathbf{v}) = s \cdot \mathbf{u} + s \cdot \mathbf{v}$
- $(s + t) \cdot \mathbf{v} = s \cdot \mathbf{v} + t \cdot \mathbf{v}$  and  $s \cdot (t \cdot \mathbf{v}) = (st) \cdot \mathbf{v}$
- there is a scalar 1 such that  $1 \cdot \mathbf{v} = \mathbf{v}$  for any  $\mathbf{v} \in \mathbf{V}$

### A.1.1 Inner Product Spaces

An inner product space is a vector space  $\mathbf{V}$  on which an inner product operation is defined, which is an operation that maps any two vectors  $\mathbf{u}, \mathbf{v} \in \mathbf{V}$  into a scalar,  $\langle \mathbf{u}, \mathbf{v} \rangle \in \mathbf{K}$ , such that for any  $s, t \in \mathbf{K}$  and  $\mathbf{u}, \mathbf{v}, \mathbf{w} \in \mathbf{V}$ :

- $\langle \mathbf{u}, \mathbf{v} \rangle = \langle \mathbf{v}, \mathbf{u} \rangle$
- $\langle \mathbf{u}, s\mathbf{v} + t\mathbf{w} \rangle = s\langle \mathbf{u}, \mathbf{v} \rangle + t\langle \mathbf{u}, \mathbf{w} \rangle$
- $\langle \mathbf{u}, \mathbf{u} \rangle \geq 0$  and  $\langle \mathbf{u}, \mathbf{u} \rangle = 0$  if and only if  $\mathbf{u} = \mathbf{0}$

If  $\mathbf{V}$  is an inner product space over the real line, such that  $\mathbf{K} = \mathbb{R}$ , then a norm on  $\mathbf{V}$  can be defined by:

$$\|\mathbf{v}\| = \sqrt{\langle \mathbf{v}, \mathbf{v} \rangle} \quad (\text{A.1})$$

for any  $\mathbf{v} \in \mathbf{V}$ .

### A.1.2 Basis Vectors and Dimension of a Vector Space

A set of vectors  $\{\mathbf{v}_i\}_{i=1}^n$  spans a subspace  $\mathbf{V}$  if any vector  $\mathbf{v} \in \mathbf{V}$  has a representation of the form:

$$\mathbf{v} = \sum_{i=1}^n \mathbf{v}_i a_i \quad (\text{A.2})$$

for some  $a_1, \dots, a_n \in \mathbf{K}$ . A linearly independent set that spans  $\mathbf{V}$  is called a basis for  $\mathbf{V}$ . The number of vectors in any basis of  $\mathbf{V}$  is the minimum number of elements in a set that spans  $\mathbf{V}$ , and is the maximum number of vectors that a linearly independent subset of  $\mathbf{V}$  can have. This number is called the dimension of  $\mathbf{V}$ .

## A.2 Quadratic Forms

A quadratic form is a sum of second degree monomials. Given a set of variables  $x_1, \dots, x_n$ , the associated quadratic form is given by:

$$q = a_{11}x_1^2 + a_{12}x_1x_2 + \dots + a_{1n}x_1x_n + a_{22}x_2^2 + a_{23}x_2x_3 + \dots + a_{nn}x_n^2 \quad (\text{A.3})$$

which can also be written in matrix form as:

$$q = [x_1 \ x_2 \ \dots \ x_n] \begin{bmatrix} a_{11} & \frac{1}{2}a_{12} & \dots & \frac{1}{2}a_{1n} \\ \frac{1}{2}a_{12} & a_{22} & \dots & \frac{1}{2}a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{1}{2}a_{1n} & \frac{1}{2}a_{2n} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad (\text{A.4})$$

By defining  $\mathbf{x} \in \mathbb{R}^n$  as the vector with coordinates  $x_1, x_2, \dots, x_n$ , and  $\mathbf{A} \in \mathbb{R}^{n \times n}$  as the matrix of coefficients, the quadratic form (A.4) is given by:

$$q = \mathbf{x}^\top \mathbf{A} \mathbf{x} \quad (\text{A.5})$$

Although the matrix  $\mathbf{A}$  is not necessarily symmetric, any quadratic form can be represented by a symmetric matrix [37].

### A.2.1 Positive Definiteness

An important class of quadratic forms is that of positive definite quadratic forms. A positive definite quadratic form  $\mathbf{A}$  is one which satisfies:

$$\mathbf{x}^\top \mathbf{A} \mathbf{x} > 0 \quad \text{for all non-zero vectors } \mathbf{x} \quad (\text{A.6})$$

A positive semi-definite quadratic form  $\mathbf{A}$  is one for which:

$$\mathbf{x}^\top \mathbf{A} \mathbf{x} \geq 0 \quad \text{for all vectors } \mathbf{x} \quad (\text{A.7})$$

It is clear that any positive definite quadratic form is also positive semi-definite.

Furthermore, a matrix is said to be positive definite (respectively positive semi-definite) if its corresponding quadratic form is positive definite (respectively positive semi-definite).

The sum of a positive definite matrix and a positive semi-definite one is positive definite. This is easy to verify, since if  $\mathbf{A}$  is positive definite and  $\mathbf{B}$  is positive semi-definite, then for any non-zero vector  $\mathbf{x}$ :

$$\mathbf{x}^\top (\mathbf{A} + \mathbf{B}) \mathbf{x} = \mathbf{x}^\top \mathbf{A} \mathbf{x} + \mathbf{x}^\top \mathbf{B} \mathbf{x} > 0 \quad (\text{A.8})$$

Negative definiteness and negative semi-definiteness are defined similarly. A quadratic form  $\mathbf{A}$  is negative definite if for all nonzero vectors  $\mathbf{x}$  it holds that  $\mathbf{x}^\top \mathbf{A} \mathbf{x} < 0$ .  $\mathbf{A}$  is a negative semi-definite quadratic form if for any vector  $\mathbf{x}$ ,  $\mathbf{x}^\top \mathbf{A} \mathbf{x} \leq 0$ . A matrix  $\mathbf{A}$  is negative definite (respectively negative semi-definite) if its corresponding quadratic form is negative definite (respectively negative semi-definite).

A matrix that is neither positive semi-definite nor negative semi-definite is said to be indefinite.

## Appendix B

### Code

#### B.1 Interpolation in 1D with the Moving Regularized Interpolation Method

This section contains the implementation of the Moving Regularized Interpolation method to perform interpolation in 1D. The code is in the file *MRI\_Interp.m* in the folder *MRI\_Interp* in the accompanying disk.

```
% Initialization
clear, clc
% Say we have a function u, we know its values at a few points (xh)_i
xh = (0:0.05:1)';
xh(2:end-1) = xh(2:end-1) + random('norm',0,0.005,size(xh(2:end-1)));
ui = rand(size(xh));
%ui = ones(size(xh));
%ui = xh;
%ui = 2*xh.^3-xh+0.3*sin(2*pi*xh);
```

```

xg = (min(xh):1e-3:max(xh))'; % for plotting
h = 0.15;
[Ph dPh] = ApproximationFunctions(xh,h);
%hreg = exp(1.2*(1:size(Ph,2))-10);
%hreg = 1e-7*[ones(1,3) factorial(1:(size(Ph,2)-3))];
hreg = 1e-9*[factorial(1:(size(Ph,2)))];

figure(1), subplot(1,1,1), plot(xh,ui,'o')

%% moving least squares with fixed regularization

uh = zeros(size(xg));
phi = zeros(size(xg,1),length(xh));

for i = 1:length(xg)
    x = xg(i);
    [P dP] = ApproximationFunctions(x,h);
    [wh dwh] = CubicSplineAtPoint(xh-x,h);
    wnonzero = wh > 0;
    wh = wh(wnonzero);
    Q = diag(hreg);
    dwh = dwh(wnonzero);
    Ph_c = Ph(wnonzero,:);
    uh(i) = P*(((Ph_c'*diag(wh)*Ph_c+Q)\Ph_c'*diag(wh)*ui(wnonzero)));
    phi(i,wnonzero) = P*(((Ph_c'*diag(wh)*Ph_c+Q)\Ph_c'*diag(wh)));
end

uhls = uh;

```

```

%% Plotting

figure(1), subplot(2,2,1), plot(xh, ui, 'o', xh, ui, '--y')
title('uh'), xlim([min(xh) max(xh)]), ylim([min(uh)-0.2 max(uh)+0.2])
figure(1), hold on, subplot(2,2,1), plot(xg, uh, '-')
grid on, title('uh')
xlim([min(xh) max(xh)]), ylim([min(uh)-0.2 max(uh)+0.2]), hold off
figure(1), subplot(2,2,2), plot(xg, phi*diag(ui), '-', xh, ui, 'o')
grid, title('scaled basis'), xlim([min(xh) max(xh)])
%figure(2), plot(xg, dphi), grid

%% moving regularized least squares
[Ph dPh] = ApproximationFunctions(xh,h);
pd = parallelDistance(xh, xg);
rf = 1;
uh = zeros(size(xg));
phi = zeros(size(xg,1),length(xh));

for i = 1:length(xg)
    x = xg(i);
    [P dP] = ApproximationFunctions(x,h);
    [wh dwh] = CubicSplineAtPoint(xh-x,h);
    wnonzero = wh > 0;
    wh = wh(wnonzero);
    dwh = dwh(wnonzero);
    Q = rf*pd(i)*diag(hreg);
    Ph_c = Ph(wnonzero,:);
    if(pd(i) > 1e-7)
        uh(i) = P*(((Ph_c'*diag(wh)*Ph_c+Q) ...
            \Ph_c'*diag(wh)*ui(wnonzero)));
    end
end

```

```

        phi(i,wnonzero) = P*(((Ph_c'*diag(wh)*Ph_c+Q) ...
            \Ph_c'*diag(wh)));
    else
        uh(i) = P*diag(1./hreg)*Ph_c'* ...
            ((Ph_c*diag(1./hreg)*Ph_c')\ui(wnonzero));
        phi(i,wnonzero) = P*diag(1./hreg)* ...
            Ph_c'*((Ph_c*diag(1./hreg)*Ph_c')\eye(sum(wnonzero)));
    end
end

% Plotting

figure(1), subplot(2,2,3), plot(xh, ui, 'o', xh, ui, '--y')
title('uh-wreg')
xlim([min(xh) max(xh)]), ylim([min(uh)-0.2 max(uh)+0.2])
figure(1), hold on, subplot(2,2,3), plot(xg, uh, '-')
grid on, title('uh-wreg')
xlim([min(xh) max(xh)]), ylim([min(uh)-0.2 max(uh)+0.2])
hold off
figure(1), subplot(2,2,4), plot(xg, phi*diag(ui), '- ', xh, ui, 'o')
grid, title('scaled basis-wreg'), xlim([min(xh) max(xh)])

%%
sqrt(sum((uhls-uh).^2)/length(uh))

return
%%
for i = 1:size(phi,2)
    figure(3), plot(xg, phi(:,i), '- ', xh(i), 1, 'o')
    grid, title(sprintf('Node %d shape fcn',i))
end

```

```

pause(5)
end

%%
x = 0.42;
[P dP] = ApproximationFunctions(x,h);
[wh dwh] = CubicSplineAtPoint(xh-x,h);
[wg dwg] = CubicSplineAtPoint(xg-x,h);
figure, plot(xg,wg,'-','LineWidth',2), grid
hold on, plot(xh,wh,'o','LineWidth',2)
hold on, plot([0.42 0.42],[-0.2 1.2],'--r')
ylim([-0.1 0.8])
xlabel('x'), ylabel('w')
wnonzero = wh > 0;
wh = wh(wnonzero);
dwh = dwh(wnonzero);
Q = rf*pd(i)*diag(hreg);
Ph_c = Ph(wnonzero,:);
if(pd(i) > 1e-7)
    %sprintf('%g\n',cond(Ph_c'*diag(wh)*Ph_c+Q))
    uh(i) = P*(((Ph_c'*diag(wh)*Ph_c+Q)\Ph_c'*diag(wh)*ui(wnonzero)));
    phi(i,wnonzero) = P*(((Ph_c'*diag(wh)*Ph_c+Q)\Ph_c'*diag(wh)));
else
    uh(i) = P*diag(1./hreg)*Ph_c'* ...
        ((Ph_c*diag(1./hreg)*Ph_c')\ui(wnonzero));
    phi(i,wnonzero) = P*diag(1./hreg)* ...
        Ph_c'*((Ph_c*diag(1./hreg)*Ph_c')\eye(sum(wnonzero)));
end

figure, plot(xg,pd,'-','LineWidth',2), grid

```

```
hold on, plot(0.42,pd(xg==0.42),'o','LineWidth',2)
hold on, plot([0.42 0.42],[0 3.5]*1e-4,'--r')
ylim([0 3.5]*1e-4)
xlabel('x'), ylabel('pd')
```

## B.2 Model Problem: Axially Loaded Elastic Bar

This section contains the implementation of the Moving Regularized Interpolation method for constructing shape functions to be used with a Galerkin method to solve the one-dimensional model problem of an axially loaded elastic bar. The code is in the file *elasticBar.m* in the folder *Elastic Bar*, which is in the *ModelProblems* folder of the accompanying disk.

```
%% Initialization
clear, clc
xh = (0:0.05:1)';
xh(2:end-1) = xh(2:end-1) + random('norm',0,0.01,size(xh(2:end-1)));

xg = (min(xh):1e-3:max(xh))'; % for plotting
h = 0.13;
Ph = ApproximationFunctions(xh,h);
%hreg = exp(0.05*(1:size(Ph,2))-15);
%hreg = 1e-8*[ones(1,3) factorial(1:(size(Ph,2)-3))];
%hreg = 1e-9*[factorial(1:(size(Ph,2)))];
hreg = 1e-9*(1:size(Ph,2));

%% moving regularized least squares
```

```

[Ph dPh] = ApproximationFunctions(xh,h);
[pd dpd] = parallelDistance(xh, xg);
rf = 1;
uh = zeros(size(xg));
duh = zeros(size(xg));
phi = zeros(size(xg,1),length(xh));
dphi = zeros(size(xg,1),length(xh));

for i = 1:length(xg)
    x = xg(i);
    [P dP] = ApproximationFunctions(x,h);
    [wh dwh] = CubicSplineAtPoint(xh-x,h);
    wnonzero = wh > 0;
    wh = wh(wnonzero);
    dwh = dwh(wnonzero);
    Q = rf*pd(i)*diag(hreg);
    dQ = rf*dpd(i)*diag(hreg);
    Ph_c = Ph(wnonzero,:);
    ei = eye(sum(wnonzero));
    dei = zeros(sum(wnonzero));
    a = zeros(size(Ph_c,2),sum(wnonzero));
    da = zeros(size(Ph_c,2),sum(wnonzero));
    minm = 3;
    for j = [minm 5]
        pj = Ph_c(:,1:j);
        if(pd(i) > 1e-7 || size(pj,2)<size(pj,1))
            A = (pj'*diag(wh)*pj+sqrt(j-minm)*Q(1:j,1:j));
            dA = (pj'*diag(dwh)*pj+sqrt(j-minm)*dQ(1:j,1:j));
            B = (pj'*diag(wh));
            dB = (pj'*diag(dwh));
        end
    end
end

```

```

        aj = A\B*ei;
        daj = (-A\((dA*(A\B))+A\dB)*ei+A\B*dei;
    else
        aj = diag(1./hreg(1:j))*pj'*((pj*diag(1./hreg(1:j))*pj')\ei);
        daj = diag(1./hreg(1:j))*pj'*((pj*diag(1./hreg(1:j))*pj')\dei);
    end
    ei = ei - pj*aj;
    dei = dei - pj*daj;
    a(1:j,:) = a(1:j,:) + aj;
    da(1:j,:) = da(1:j,:) + daj;
end
phi(i,wnonzero) = P*a;
dphi(i,wnonzero) = dP*a + P*da;
end

%% Stiffness matrix and load vector
K = dphi(:,2:end)'*dphi(:,2:end);
F = phi(:,2:end)'*xg;
ui = [0; K\F];
uexact = -xg.^3/6+xg/2;

% Plotting the solution
%figure(1), subplot(1,2,1), plot(xh, ui, 'o', xh, ui, '--y')
title('uh-wreg')
xlim([min(xh) max(xh)]), ylim([min(uh)-0.2 max(uh)+0.2])
figure(1), hold on, subplot(1,2,1), plot(xg,uexact,'--r',xg, phi*ui, '-')
grid on, title('uh-wreg')
xlim([min(xh) max(xh)]), ylim([min(ui)-0.2 max(ui)+0.2]), hold off
figure(1), hold on
subplot(1,2,1), plot(xh,zeros(size(xh)),'xk')
```

```

hold off
figure(1), subplot(1,2,2), hold off
plot(xg, phi*diag(ui), '- ', xh, ui, 'o')
grid, title('scaled basis-wreg'), xlim([min(xh) max(xh)])

return
%%
figure(2), hold on, p = plot(xg,uexact,'--r',xg, phi*ui, '- '), grid on,
xlim([min(xh) max(xh)]), ylim([min(ui)-0.1 max(ui)+0.1]), hold off
xlabel('x'), ylabel('u'), legend('Exact Solution','MRI Solution')
set(p,'LineWidth',2)
figure(2), hold on, plot(xh,zeros(size(xh)),'xk'), hold off

%%
for i = 1:size(phi,2)
figure(3), plot(xg, phi(:,i), '- ', xh(i), 1, 'o')
grid, title(sprintf('Node %d shape fcn',i))
pause(0.6)
end

%%
for i = 1:size(phi,2)
figure(3), plot(xg, dphi(:,i), '- ', xh(i), 1, 'o')
grid, title(sprintf('Node %d shape fcn derivative',i))
pause(0.8)
end

```

## B.3 Model Problem: Plate Under Thermal Equilibrium

This section contains the implementation of the Moving Regularized Interpolation method to generate two-dimensional shape functions for solving the model problem of a plate under thermal equilibrium. The code is included in the file *thermalPlate.m* in the folder *Thermal Plate*, which is in the *Model Problems* folder.

```
clear, clc

tic

%% Initialization
% Nodes
xh = (0:0.1:1)';
yh = (0:0.1:1)';
[xh yh] = meshgrid(xh,yh);
%xh(12:end) = xh(12:end) + random('norm',0,1.2e-2,size(xh(12:end)));
%yh(12:end) = yh(12:end) + random('norm',0,1.2e-2,size(yh(12:end)));
%xh(xh>1)=1;xh(xh<0)=0;
%yh(yh>1)=1;yh(yh<0)=0;

% Constitutive
k = 1;

%%
cellWidth = 1/20;
```

```

cellHeight = cellWidth;
% cell centers
[xgr ygr] = meshgrid((cellWidth/2:cellWidth:1-cellWidth/2) ...
    ,(cellHeight/2:cellHeight:1-cellHeight/2));
xgr = kron(xgr,ones(3)); ygr = kron(ygr,ones(3));
% integration points and weights
[xgc ygc] = meshgrid([-sqrt(3/5) 0 sqrt(3/5)],[-sqrt(3/5) 0 sqrt(3/5)]);
xgr = xgr + kron(ones(floor(1/cellWidth)),cellWidth/2*xgc);
ygr = ygr + kron(ones(floor(1/cellHeight)),cellHeight/2*ygc);
wCorner = 25/81; wEdge = 40/81; wCenter = 64/81;
wg = kron(ones(floor(1/cellWidth)), ...
    [wCorner wEdge wCorner; wEdge wCenter wEdge; ...
    wCorner wEdge wCorner])*cellHeight/2*cellWidth/2;

% free boundary integration points
yfb = kron((cellWidth/2:cellWidth:1-cellWidth/2)',ones(3,1));
yfb = yfb + kron(ones(floor(1/cellWidth),1), ...
    cellWidth/2*[-sqrt(3/5); 0; sqrt(3/5)]);
xfb = ones(size(yfb));
wgfb = kron(ones(floor(1/cellWidth),1),cellWidth/2*[5/9; 8/9; 5/9]);

%%
h = 0.2;

[Ph dPhdx dPhdy] = ApproximationFunctions2D([xh(:) yh(:)],h);
xg = [xgr(:); xfb(:)]; yg = [ygr(:); yfb(:)]; wg = [wg(:); wgfb(:)];
[pd dpd_dx dpd_dy] = parallelDistance2D(xh(:),yh(:),h,xg,yg);
rf = 1;
phi = zeros(size(xg,1),length(xh));
dphidx = zeros(size(xg,1),length(xh));

```

```

dphidy = zeros(size(xg,1),length(xh));
hreg = exp(1.5*([1;2;2;3;3;3;4;4;4;4;5;5;5;5;5;6;6;6;6;6;6])-25);

%%
for i = 1:length(xg)
    x = [xg(i) yg(i)];
    [P dPdx dPdy] = ApproximationFunctions2D(x,h);
    dx = xh(:)-ones(length(xh(:)),1)*xg(i);
    dy = yh(:)-ones(length(yh(:)),1)*yg(i);
    [wh dwhdx dwhdy] = CubicSplineAt2DPoint(dx,dy,h);
    wnonzero = wh > 0;
    wh = wh(wnonzero);
    dwhdx = dwhdx(wnonzero);
    dwhdy = dwhdy(wnonzero);
    Q = rf*pd(i)*diag(hreg);
    dQdx = rf*dpd_dx(i)*diag(hreg);
    dQdy = rf*dpd_dy(i)*diag(hreg);
    Ph_c = Ph(wnonzero,:);
    ei = eye(sum(wnonzero));
    deidx = zeros(sum(wnonzero));
    deidy = zeros(sum(wnonzero));
    a = zeros(size(Ph_c,2),sum(wnonzero));
    dadx = zeros(size(Ph_c,2),sum(wnonzero));
    dady = zeros(size(Ph_c,2),sum(wnonzero));
    minm = 6;
    for j = [minm size(Ph_c,2)]
        pj = Ph_c(:,1:j);
        if(pd(i) > 1e-7 || size(pj,2)<size(pj,1))
            A = (pj'*diag(wh)*pj+sqrt(j-minm)*Q(1:j,1:j));
            dAdx = (pj'*diag(dwhdx)*pj+sqrt(j-minm)*dQdx(1:j,1:j));

```

```

        dAdy = (pj'*diag(dwhdy)*pj+sqrt(j-minm)*dQdy(1:j,1:j));
        B = (pj'*diag(wh));
        dBdx = (pj'*diag(dwhdx));
        dBdy = (pj'*diag(dwhdy));
        aj = A\B*ei;
        dajdx = (-A\ (dAdx*(A\B))+A\dBdx)*ei+A\B*deidx;
        dajdy = (-A\ (dAdy*(A\B))+A\dBdy)*ei+A\B*deidy;
    else
        aj = diag(1./hreg(1:j))*pj'*((pj*diag(1./hreg(1:j))*pj')\ei);
        dajdx = diag(1./hreg(1:j))*pj' ...
            *((pj*diag(1./hreg(1:j))*pj')\deidx);
        dajdy = diag(1./hreg(1:j))*pj' ...
            *((pj*diag(1./hreg(1:j))*pj')\deidy);
    end
    ei = ei - pj*aj;
    deidx = deidx - pj*dajdx;
    deidy = deidy - pj*dajdy;
    a(1:j,:) = a(1:j,:) + aj;
    dadx(1:j,:) = dadx(1:j,:) + dajdx;
    dady(1:j,:) = dady(1:j,:) + dajdy;
end
phi(i,wnonzero) = P*a;
dphidx(i,wnonzero) = dPdx*a + P*dadx;
dphidy(i,wnonzero) = dPdy*a + P*dady;
end

%% Stiffness matrix and load vector
phifb = phi(end-length(xfb(:))+1:end,:);

phi(end-length(xfb(:))+1:end,:) = [];

```

```

dphidx(end-length(xfb(:))+1:end,:) = [];
dphidy(end-length(xfb(:))+1:end,:) = [];
wg(end-length(xfb(:))+1:end,:) = [];

Bphi = reshape([dphidx dphidy]',size(dphidx,2),[]);

%%

K = zeros(size(phi,2));
for i = 1:length(xgr(:))
    K = K + wg(i)*(Bphi(2*(i-1)+1:2*(i-1)+2,:)' ...
        *k*Bphi(2*(i-1)+1:2*(i-1)+2,:));
end
%%
% F = kron(phi(:,12:end)'*diag(sparse(wg))*-ones(size(phi,1),1),[0;1]);
F = [zeros(size(phifb,2)-22,1); phifb(:,end-10:end)] ...
    *diag(sparse(wgfb))*ones(size(phifb,1),1)];
U = [zeros(11,1); K(12:end,12:end)\F];

figure(1), meshc(xgr,ygr,reshape(phi*U,size(xgr,1),size(xgr,2)));

toc

return

%%

[uK sK vK] = svd(K);
for k = size(K,2):-1:1
figure(3), meshc(xgr,ygr,reshape(phi*sK(k,k)*vK(1:2:end,k) ...
    ,size(xgr,1),size(xgr,2)));

```

```
title(sprintf('Eigenmode with frequency %0.2f Hz',sqrt(sK(k,k))/(2*pi)))
xlabel('x')
ylabel('y')
pause(0.6)
end
```

## B.4 Model Problem: Cantilever Beam Under Plane Stress

This section contains the Matlab implementation of the Moving Regularized Interpolation method to produce 2D shape functions for use in a plane stress problem. The code is contained in the file *cantileverBeam.m* in the folder *Cantilever Beam*, which is in the *Model Problems* folder of the accompanying disk.

```
clear, clc

tic

%% Initialization
% Nodes
xh = (0:0.02:1)';
yh = (-0.05:0.02:0.05)';
[xh yh] = meshgrid(xh,yh);
%xh(xh~=0&xh~=1) = xh(xh~=0&xh~=1) ...
    + random('norm',0,1.4e-3,size(xh(xh~=0&xh~=1)));
%yh(yh~=-0.05&yh~=0.05) = yh(yh~=-0.05&yh~=0.05) ...
```

```

    + random('norm',0,1.4e-3,size(yh(yh~=-0.05&yh~=0.05)));
figure, plot(xh,yh,'o'),grid
% Constitutive
young = 200e9;nu=0.3;
Dmat = (young/(1-nu^2))*[1 nu 0;nu 1 0;0 0 (1-nu)/2];

%%
cellWidth = .1/20;
cellHeight = cellWidth;
% cell centers
[xgr ygr] = meshgrid((cellWidth/2:cellWidth:1-cellWidth/2) ...
    ,(-.05+cellHeight/2:cellHeight:.05-cellHeight/2));
xgr = kron(xgr,ones(3)); ygr = kron(ygr,ones(3));
% integration points and weights
[xgc ygc] = meshgrid([-sqrt(3/5) 0 sqrt(3/5)],[-sqrt(3/5) 0 sqrt(3/5)]);
xgr = xgr + kron(ones(size(xgr)/3),cellWidth/2*xgc);
ygr = ygr + kron(ones(size(ygr)/3),cellHeight/2*ygc);
wCorner = 25/81; wEdge = 40/81; wCenter = 64/81;
wg = kron(ones(floor(1/cellWidth)),[wCorner wEdge wCorner; ...
    wEdge wCenter wEdge; wCorner wEdge wCorner])*cellHeight/2*cellWidth/2;

% free boundary integration points
yfb = kron((-0.05+cellWidth/2:cellWidth:.05-cellWidth/2)',ones(3,1));
yfb = yfb + kron(ones(floor(.1/cellWidth),1), ...
    cellWidth/2*[-sqrt(3/5); 0; sqrt(3/5)]);
xfb = ones(size(yfb));
wgfb = kron(ones(floor(.1/cellWidth),1), ...
    cellWidth/2*[5/9; 8/9; 5/9]);

%%

```

```

h = 0.03;

[Ph dPhdx dPhdy] = ApproximationFunctions2D([xh(:) yh(:)],h);
xg = [xgr(:); xfb(:)]; yg = [ygr(:); yfb(:)]; wg = [wg(:); wgfb(:)];
[pd dpd_dx dpd_dy] = parallelDistance2D(xh(:),yh(:),h,xg,yg);
rf = 1;
phi = zeros(size(xg,1),length(xh));
dphidx = zeros(size(xg,1),length(xh));
dphidy = zeros(size(xg,1),length(xh));
hreg = exp(1.5*([1;2;2;3;3;3;4;4;4;4;5;5;5;5;5;6;6;6;6;6;6])-20);

%%
for i = 1:length(xg)
    x = [xg(i) yg(i)];
    [P dPdx dPdy] = ApproximationFunctions2D(x,h);
    dx = xh(:)-ones(length(xh(:)),1)*xg(i);
    dy = yh(:)-ones(length(yh(:)),1)*yg(i);
    [wh dwhdx dwhdy] = CubicSplineAt2DPoint(dx,dy,h);
    wnonzero = wh > 0;
    wh = wh(wnonzero);
    dwhdx = dwhdx(wnonzero);
    dwhdy = dwhdy(wnonzero);
    Q = rf*pd(i)*diag(hreg);
    dQdx = rf*dpd_dx(i)*diag(hreg);
    dQdy = rf*dpd_dy(i)*diag(hreg);
    Ph_c = Ph(wnonzero,:);
    ei = eye(sum(wnonzero));
    deidx = zeros(sum(wnonzero));
    deidy = zeros(sum(wnonzero));
    a = zeros(size(Ph_c,2),sum(wnonzero));

```

---

```

dadx = zeros(size(Ph_c,2),sum(wnonzero));
dady = zeros(size(Ph_c,2),sum(wnonzero));
minm = 3;
for j = [minm size(Ph_c,2)]
    pj = Ph_c(:,1:j);
    if(pd(i) > 1e-9 || size(pj,2)<size(pj,1))
        A = (pj'*diag(wh)*pj+sqrt(j-minm)*Q(1:j,1:j));
        dAdx = (pj'*diag(dwhdx)*pj+sqrt(j-minm)*dQdx(1:j,1:j));
        dAdy = (pj'*diag(dwhdy)*pj+sqrt(j-minm)*dQdy(1:j,1:j));
        B = (pj'*diag(wh));
        dBdx = (pj'*diag(dwhdx));
        dBdy = (pj'*diag(dwhdy));
        aj = A\B*ei;
        dajdx = (-A\'(dAdx*(A\B))+A\'dBdx)*ei+A\B*deidx;
        dajdy = (-A\'(dAdy*(A\B))+A\'dBdy)*ei+A\B*deidy;
    else
        aj = diag(1./hreg(1:j))*pj'*((pj*diag(1./hreg(1:j))*pj')\ei);
        dajdx = diag(1./hreg(1:j))*pj' ...
            *((pj*diag(1./hreg(1:j))*pj')\deidx);
        dajdy = diag(1./hreg(1:j))*pj' ...
            *((pj*diag(1./hreg(1:j))*pj')\deidy);
    end
    ei = ei - pj*aj;
    deidx = deidx - pj*dajdx;
    deidy = deidy - pj*dajdy;
    a(1:j,:) = a(1:j,:) + aj;
    dadx(1:j,:) = dadx(1:j,:) + dajdx;
    dady(1:j,:) = dady(1:j,:) + dajdy;
end
phi(i,wnonzero) = P*a;

```

```

    dphidx(i,wnonzero) = dPdx*a + P*dadx;
    dphidy(i,wnonzero) = dPdy*a + P*dady;
end

phifb = phi(end-length(xfb(:))+1:end,:);

phi(end-length(xfb(:))+1:end,:) = [];
dphidx(end-length(xfb(:))+1:end,:) = [];
dphidy(end-length(xfb(:))+1:end,:) = [];
wg(end-length(xfb(:))+1:end,:) = [];

Bphi = [kron(dphidx,[1 0]), kron(dphidy,[0 1]), ...
        kron(dphidx,[0 1])+kron(dphidy,[1 0])];
Bphi = reshape(Bphi',size(Bphi,2)/3,[],)';

K = zeros(2*size(phi,2));
for i = 1:length(xgr(:))
    K = K + wg(i)*(Bphi(3*(i-1)+1:3*(i-1)+3,:))' ...
        *Dmat*Bphi(3*(i-1)+1:3*(i-1)+3,:));
end

%%

D = 0.1;
I = D^3/12; P = 1;
Ffb = P/(2*I)*(D^2/4-yfb.^2);
F = [zeros(2*(size(phifb,2)-12),1); ...
     kron(phifb(:,end-5:end))*diag(sparse(wgfb))*Ffb,[0;-1])];
U = [zeros(12,1); K(13:end,13:end)\F];
Ux = U(1:2:end);
Uy = U(2:2:end);

```

```
toc

% exact displacements
Uyex = -P/(6*young*I)*(3*nu*ygr.^2.*(1-xgr) ...
        +(4+5*nu)*D^2*xgr/4+(3*1-xgr).*xgr.^2);
Uxex = P/(6*young*I)*ygr.*((6*1-3*xgr).*xgr ...
        +(2+nu).*(ygr.^2-D^2/4));

%%
figure(1), subplot(1,2,1)
meshc(xgr,ygr,reshape(phi*Uy,size(xgr,1),size(xgr,2)))
xlabel('x'), ylabel('y'), zlabel('u_y (MRI)')
figure(1), subplot(1,2,2), meshc(xgr,ygr,Uyex);
xlabel('x'), ylabel('y'), zlabel('u_y (exact)')
figure(2), subplot(1,2,1)
meshc(xgr,ygr,reshape(phi*Ux,size(xgr,1),size(xgr,2)))
xlabel('x'), ylabel('y'), zlabel('u_x (MRI)')
zlim([-2e-9 2e-9])
figure(2), subplot(1,2,2), meshc(xgr,ygr,Uxex);
xlabel('x'), ylabel('y'), zlabel('u_x (exact)')
zlim([-2e-9 2e-9])
return

%%
ysec = abs(ygr-0.0005635)<1e-4;
figure(1), subplot(1,2,1)
p=plot(xgr(ysec),phi(ysec,:)*Ux,xgr(ysec),Uxex(ysec),'--r'), grid
xlabel('x'), ylabel('u_x at y=0'), legend('MRI','Exact')
set(p,'LineWidth',2)
```

```

ysec = abs(ygr-0.04944)<1e-4;
figure(1), subplot(1,2,2)
p=plot(xgr(ysec),phi(ysec,:)*Ux,xgr(ysec),Uxex(ysec),'--r'), grid
xlabel('x'), ylabel('u_x at y=0.5'), legend('MRI','Exact')
set(p,'LineWidth',2)

ysec = abs(ygr-0.0005635)<1e-4;
figure(2), subplot(1,2,1)
p=plot(xgr(ysec),phi(ysec,:)*Uy,xgr(ysec),Uyex(ysec),'--r'), grid
xlabel('x'), ylabel('u_y at y=0'), legend('MRI','Exact')
set(p,'LineWidth',2)

ysec = abs(ygr-0.04944)<1e-4;
figure(2), subplot(1,2,2)
p=plot(xgr(ysec),phi(ysec,:)*Uy,xgr(ysec),Uyex(ysec),'--r'), grid
xlabel('x'), ylabel('u_y at y=0.5'), legend('MRI','Exact')
set(p,'LineWidth',2)

%%
[uK sK vK] = svd(K);
for k = size(K,2):-1:1
figure(3)
meshc(xgr,ygr,reshape(phi*sK(k,k)*vK(1:2:end,k),size(xgr,1),size(xgr,2)));
title(sprintf('Eigenmode with frequency %0.2f Hz',sqrt(sK(k,k))/(2*pi)))
xlabel('x')
ylabel('y')
pause(0.6)
end
%%
for i = 1:size(phi,2)

```

```
figure(3), meshc(xgr,ygr,reshape(phi(:,i),size(xgr,1),size(xgr,2)))
xlabel('x')
ylabel('y')

pause(0.6)
end

%%

patch = K*ones(size(K,2),1);
figure, meshc(xgr,ygr,reshape(phi*patch(1:2:end),size(xgr,1),size(xgr,2)));
xlabel('x'), ylabel('y')
```

## B.5 GPU-accelerated EFG Implementation

This section contains the code of the GPU implementation of the Element-Free Galerkin method that was developed in the current work. The files are included in the folder *GPU Implementation* of the accompanying disk.

### B.5.1 Data Structures

```
#ifndef _QUADRATURE_NODE_H
#define _QUADRATURE_NODE_H

struct QuadraturePoint
{
```

```
unsigned int *d_num_quad_points;
float *d_coordinates;
float *d_weights;
unsigned int *d_contributionCounts;
unsigned int *d_contributionIndices;
unsigned int *d_total_contributions;
unsigned int *d_interactionSetIndices;
unsigned int *d_total_interactions;
float *d_momentMatrix;
float *d_derivMomentMatrix;
float *d_momentVector;
float *d_derivMomentVector;
};

struct MeshNode
{
unsigned int *d_num_nodes;
float *d_influenceDomainRadius;
float *d_coordinates;
};

struct ContributionData
{
unsigned int *d_meshNodeAddress;
unsigned int *d_quadPointAddress;
unsigned int *d_localContributionAddress;
float *d_distanceToNode;
float *d_weight;
float *d_deriv_weight;
```

```
};

#endif // #ifndef _QUADRATURE_NODE_H
```

## B.5.2 Host Code

```
////////////////////////////////////
// Program main
////////////////////////////////////
int
main( int argc, char** argv)
{
    //Properties();
    //getchar();
    meshlessTest( argc, argv);
}

void allocateNodeMemory(MeshNode *nodeData, unsigned int num_nodes)
{
    unsigned int node_count_mem_size = sizeof( float);
    unsigned int influence_domain_radius_mem_size = sizeof( float);
    unsigned int node_location_mem_size = sizeof( float) * (num_nodes);

    checkCudaErrors( cudaMalloc( (void**)
        &(nodeData->d_num_nodes), node_count_mem_size ) );
    checkCudaErrors( cudaMalloc( (void**)
        &(nodeData->d_influenceDomainRadius),
        influence_domain_radius_mem_size) );
}
```

```
checkCudaErrors( cudaMalloc( (void**)
    &(nodeData->d_coordinates),
    node_location_mem_size) );
}

void allocateQuadPointMemory(QuadraturePoint *qNodeData,
    unsigned int num_quad_points)
{
    unsigned int quadrature_node_count_mem_size
        = sizeof( unsigned int);
    unsigned int quadrature_node_location_mem_size
        = sizeof( float) * (num_quad_points);
    unsigned int quadrature_node_weight_mem_size
        = sizeof( float) * (num_quad_points);
    unsigned int contribution_count_mem_size
        = sizeof( unsigned int) * (num_quad_points);
    unsigned int contribution_indices_mem_size
        = sizeof( unsigned int) * (num_quad_points);
    unsigned int contribution_total_mem_size
        = sizeof( unsigned int);
    unsigned int interaction_indices_mem_size
        = sizeof( unsigned int) * (num_quad_points);
    unsigned int interaction_total_mem_size
        = sizeof( unsigned int);
    unsigned int node_moment_matrix_mem_size
        = 3 * sizeof( float) * (num_quad_points);
    unsigned int moment_vector_mem_size
        = 2 * sizeof( float) * (num_quad_points);

    // allocate memory
```

```

checkCudaErrors( cudaMalloc( (void**)
    &(qNodeData->d_num_quad_points), quadrature_node_count_mem_size) );
checkCudaErrors( cudaMalloc( (void**)
    &(qNodeData->d_coordinates), quadrature_node_location_mem_size) );
checkCudaErrors( cudaMalloc( (void**)
    &(qNodeData->d_weights), quadrature_node_weight_mem_size) );
checkCudaErrors( cudaMalloc( (void**)
    &(qNodeData->d_contributionCounts), contribution_count_mem_size) );
checkCudaErrors( cudaMalloc( (void**)
    &(qNodeData->d_contributionIndices), contribution_indices_mem_size) );
checkCudaErrors( cudaMalloc( (void**)
    &(qNodeData->d_total_contributions), contribution_total_mem_size) );
checkCudaErrors( cudaMalloc( (void**)
    &(qNodeData->d_interactionSetIndices), interaction_indices_mem_size) );
checkCudaErrors( cudaMalloc( (void**)
    &(qNodeData->d_total_interactions), interaction_total_mem_size) );
checkCudaErrors( cudaMalloc( (void**)
    &(qNodeData->d_momentMatrix), node_moment_matrix_mem_size) );
checkCudaErrors( cudaMalloc( (void**)
    &(qNodeData->d_derivMomentMatrix), node_moment_matrix_mem_size) );
checkCudaErrors( cudaMalloc( (void**)
    &(qNodeData->d_momentVector), moment_vector_mem_size) );
checkCudaErrors( cudaMalloc( (void**)
    &(qNodeData->d_derivMomentVector), moment_vector_mem_size) );
}

void allocateContributionMemory
    (ContributionData *contributionData, unsigned int num_contributions)
{
    unsigned int node_address_mem_size

```

```
    = sizeof( unsigned int) * (num_contributions);
unsigned int qPoint_address_mem_size
    = sizeof( unsigned int) * (num_contributions);
unsigned int local_address_mem_size
    = sizeof( unsigned int) * (num_contributions);
unsigned int distance_to_node_mem_size
    = sizeof( float) * (num_contributions);
unsigned int contribution_weight_mem_size
    = sizeof( float) * (num_contributions);
unsigned int deriv_weight_mem_size
    = sizeof( float) * (num_contributions);

// allocate memory
checkCudaErrors( cudaMalloc( (void**)
    &(contributionData->d_meshNodeAddress), node_address_mem_size) );
checkCudaErrors( cudaMalloc( (void**)
    &(contributionData->d_quadPointAddress), qPoint_address_mem_size) );
checkCudaErrors( cudaMalloc( (void**)
    &(contributionData->d_localContributionAddress), local_address_mem_size) );
checkCudaErrors( cudaMalloc( (void**)
    &(contributionData->d_distanceToNode), distance_to_node_mem_size) );
checkCudaErrors( cudaMalloc( (void**)
    &(contributionData->d_weight), contribution_weight_mem_size) );
checkCudaErrors( cudaMalloc( (void**)
    &(contributionData->d_deriv_weight), deriv_weight_mem_size) );
}

void freeNodeMemory(MeshNode *nodeData)
{
    checkCudaErrors( cudaFree( nodeData->d_num_nodes) );
}
```

```
checkCudaErrors( cudaFree( nodeData->d_influenceDomainRadius) );
checkCudaErrors( cudaFree( nodeData->d_coordinates) );
}
```

```
void freeQuadPointMemory(QuadraturePoint *qNodeData)
{
    // free memory
    checkCudaErrors( cudaFree( qNodeData->d_coordinates) );
    checkCudaErrors( cudaFree( qNodeData->d_weights) );
    checkCudaErrors( cudaFree( qNodeData->d_contributionCounts) );
    checkCudaErrors( cudaFree( qNodeData->d_contributionIndices) );
    checkCudaErrors( cudaFree( qNodeData->d_total_contributions) );
    checkCudaErrors( cudaFree( qNodeData->d_interactionSetIndices) );
    checkCudaErrors( cudaFree( qNodeData->d_total_interactions) );
    checkCudaErrors( cudaFree( qNodeData->d_momentMatrix) );
    checkCudaErrors( cudaFree( qNodeData->d_derivMomentMatrix) );
    checkCudaErrors( cudaFree( qNodeData->d_momentVector) );
    checkCudaErrors( cudaFree( qNodeData->d_derivMomentVector) );
}
```

```
void freeContributionMemory(ContributionData *contributionData)
{
    // free memory
    checkCudaErrors( cudaFree( contributionData->d_meshNodeAddress) );
    checkCudaErrors( cudaFree( contributionData->d_quadPointAddress) );
    checkCudaErrors( cudaFree( contributionData->d_localContributionAddress) );
    checkCudaErrors( cudaFree( contributionData->d_distanceToNode) );
    checkCudaErrors( cudaFree( contributionData->d_weight) );
    checkCudaErrors( cudaFree( contributionData->d_deriv_weight) );
}
```

```
void readAndReportTimerValue(FILE *outputFile, StopwatchInterface *timer,
    char* message)
{
    sdkStopTimer( &timer );
    fprintf_s(outputFile, "%s: %f (ms)\n\n",
        message, sdkGetTimerValue( &timer ) );
    sdkStartTimer( &timer );
    sdkResetTimer( &timer );
}
```

```
void readAndReportTimerValue(FILE *outputFile, StopwatchInterface *timer,
    char *message, float *cumulativeTime)
{
    sdkStopTimer( &timer );
    float elapsedTime = sdkGetTimerValue( &timer );
    fprintf_s(outputFile, "%s: %f (ms)\n\n", message, elapsedTime );
    (*cumulativeTime) += elapsedTime;
    sdkStartTimer( &timer );
    sdkResetTimer( &timer );
}

////////////////////////////////////
//! Perform Meshless efg
////////////////////////////////////

void
meshlessTest( int argc, char** argv)
{
    shrQAStart(argc, argv);

    // use command-line specified CUDA device,
```

```
// otherwise use device with highest Gflops/s
int devID = findCudaDevice(argc, (const char**)argv);

for( unsigned int num_nodes = 3971; num_nodes <= 1024*7; num_nodes += 10)
{
    FILE *debugFileStream, *timeRecordFileStream;
    fopen_s( &timeRecordFileStream,
        "C:\\Users\\Maien\\Documents\\MSc\\Code\\GPU_times.txt", "w" );
    float memoryManagementTime = 0;
    float dataTransferTime = 0;
    float nodeComputationTime = 0;
    float quadStructureComputationTime = 0;
    float populatingContributionDataTime = 0;
    float processingMomentMatricesTime = 0;
    float processingCorrectionVectorsTime = 0;
    float populatingLoadVectorTime = 0;
    float populatingStiffnessMatrixTime = 0;

    StopwatchInterface *timer = 0;
    sdkCreateTimer( &timer );
    sdkStartTimer( &timer );

    unsigned int block_dimension = 512;
    //    unsigned int num_nodes = 2000;//1024*7;
    unsigned int num_quad_points = num_nodes;//2000;//1024*7;
    // array since data will be copied from device
    unsigned int p_num_contributions[1];
    unsigned int p_num_interactions[1];
```

```
QuadraturePoint qPoint;
MeshNode mNode;
ContributionData contributionInfo;
float *d_stiffnessMatrix, *d_loadVector, *d_constraintVector;
float *h_stiffnessMatrix =
    (float*) malloc( (num_nodes*(num_nodes+1))/2 * sizeof( float));
float *h_loadVector =
    (float*) malloc(num_nodes*sizeof( float));
float *h_constraintVector =
    (float*) malloc(num_nodes*sizeof( float));

readAndReportTimerValue(timeRecordFileStream, timer,
    "Allocating host memory", &memoryManagementTime);

allocateNodeMemory(&mNode, num_nodes);
allocateQuadPointMemory(&qPoint, num_quad_points);
checkCudaErrors( cudaMalloc( (void**) &(d_stiffnessMatrix),
    ((num_nodes*(num_nodes+1))/2) * sizeof( float) ) );
checkCudaErrors( cudaMalloc( (void**) &(d_loadVector),
    num_nodes * sizeof( float) ) );
checkCudaErrors( cudaMalloc( (void**) &(d_constraintVector),
    num_nodes * sizeof( float) ) );

readAndReportTimerValue(timeRecordFileStream, timer,
    "Allocating device memory", &memoryManagementTime);

// setup execution parameters
dim3 mNodeGrid( (num_nodes-1)/block_dimension+1, 1, 1);
dim3 mNodeThreads( block_dimension, 1, 1);
dim3 qPointThreads( block_dimension, 1, 1);
```

```

dim3  qPointGrid((num_quad_points-1)/block_dimension+1,1,1);

    // populate mesh node data
    computeNodeLocations<<< mNodeGrid,mNodeThreads>>>
        ( num_nodes, mNode.d_coordinates, mNode.d_num_nodes,
          mNode.d_influenceDomainRadius);

readAndReportTimerValue(timeRecordFileStream, timer,
    "Node location computation time", &nodeComputationTime);

// populate quadrature data
computeQuadratureStructure<<< qPointGrid, qPointThreads >>>
    ( num_quad_points, qPoint.d_num_quad_points, qPoint.d_coordinates,
      mNode.d_num_nodes, mNode.d_coordinates,
      mNode.d_influenceDomainRadius, qPoint.d_contributionCounts,
      qPoint.d_contributionIndices, qPoint.d_momentMatrix,
      qPoint.d_derivMomentMatrix, qPoint.d_interactionSetIndices,
      qPoint.d_total_contributions, qPoint.d_total_interactions);

unsigned int test[1];
checkCudaErrors( cudaMemcpy( test, qPoint.d_num_quad_points,
    sizeof( unsigned int) , cudaMemcpyDeviceToHost) );

readAndReportTimerValue(timeRecordFileStream, timer,
    "Quadrature structure computation time",
    &quadStructureComputationTime);

// get the total of contributions
checkCudaErrors( cudaMemcpy( p_num_contributions,
    qPoint.d_total_contributions, sizeof( unsigned int) ,

```

```

cudaMemcpyDeviceToHost) );

readAndReportTimerValue(timeRecordFileStream, timer,
    "Time for reading back total of contributions", &dataTransferTime);

// get the total of interactions
checkCudaErrors( cudaMemcpy( p_num_interactions,
qPoint.d_total_interactions, sizeof( unsigned int) ,
cudaMemcpyDeviceToHost) );

readAndReportTimerValue(timeRecordFileStream, timer,
    "Time for reading back total of interactions",
    &dataTransferTime);

allocateContributionMemory(&contributionInfo, p_num_contributions[0]);

readAndReportTimerValue(timeRecordFileStream, timer,
    "Time to allocate contribution memory", &memoryManagementTime);

// populate contribution data
populateContributionData<<< qPointGrid, qPointThreads >>>
( num_quad_points, qPoint.d_num_quad_points, qPoint.d_coordinates,
  mNode.d_num_nodes, mNode.d_coordinates, mNode.d_influenceDomainRadius,
  qPoint.d_contributionCounts, qPoint.d_contributionIndices,
  contributionInfo.d_meshNodeAddress, contributionInfo.d_quadPointAddress,
  contributionInfo.d_localContributionAddress,
  contributionInfo.d_distanceToNode, contributionInfo.d_weight,
  contributionInfo.d_deriv_weight);

```

```

getLastCudaError("Kernel execution failed");
readAndReportTimerValue(timeRecordFileStream, timer,
    "Populating contribution data", &populatingContributionDataTime);

// populate moment matrices and their derivatives
dim3 contributionThreads( block_dimension, 1, 1);
dim3 contributionGrid ((p_num_contributions[0]-1)/block_dimension+1, 1, 1);
unsigned int momentMatricesMemorySize =
    3 * sizeof( float) * block_dimension;
processMomentMatrices<<< contributionGrid, contributionThreads,
    2*momentMatricesMemorySize>>>( qPoint.d_total_contributions,
    contributionInfo.d_meshNodeAddress, contributionInfo.d_quadPointAddress,
    contributionInfo.d_localContributionAddress, qPoint.d_num_quad_points,
    mNode.d_coordinates, contributionInfo.d_weight,
    contributionInfo.d_deriv_weight, qPoint.d_momentMatrix,
    qPoint.d_derivMomentMatrix);

getLastCudaError("Kernel execution failed");
readAndReportTimerValue(timeRecordFileStream, timer,
    "Computing moment matrices", &processingMomentMatricesTime);

// populate the moment vectors
processMomentVectors<<<qPointGrid,
    qPointThreads, 9 * sizeof( float) * block_dimension >>>
    ( num_quad_points, qPoint.d_num_quad_points, qPoint.d_coordinates,
    mNode.d_num_nodes, mNode.d_coordinates, qPoint.d_momentVector,
    qPoint.d_derivMomentVector, qPoint.d_momentMatrix,
    qPoint.d_derivMomentMatrix);

```

```

readAndReportTimerValue(timeRecordFileStream, timer,
    "Computing moment vectors", &processingCorrectionVectorsTime);

// populate the load vector
initializeStiffnessMatrix<<<mNodeGrid,
    mNodeThreads>>>(num_nodes, d_loadVector);
initializeStiffnessMatrix<<<mNodeGrid,
    mNodeThreads>>>(num_nodes, d_constraintVector);
populateLoadVector<<<contributionGrid,contributionThreads>>>
    (qPoint.d_total_contributions, mNode.d_coordinates,
        contributionInfo.d_meshNodeAddress,
        contributionInfo.d_quadPointAddress, qPoint.d_coordinates,
        contributionInfo.d_weight, d_loadVector, d_constraintVector,
        qPoint.d_weights, qPoint.d_momentVector);
getLastCudaError("Kernel execution failed");

readAndReportTimerValue(timeRecordFileStream, timer,
    "Populating load vector", &populatingLoadVectorTime);

// initialize stiffness matrix
int num_stiffness_matrix_entries = (num_nodes*(num_nodes+1))/2;
dim3 stiffnessMatrixThreads (block_dimension, 1, 1);
dim3 stiffnessGrid ((num_stiffness_matrix_entries-1)/
    block_dimension+1, 1, 1);
initializeStiffnessMatrix <<< stiffnessGrid, stiffnessMatrixThreads >>>
    (num_stiffness_matrix_entries, d_stiffnessMatrix);
getLastCudaError("Kernel execution failed");

readAndReportTimerValue(timeRecordFileStream, timer,

```

```

        "Initializing stiffness matrix", &populatingStiffnessMatrixTime);

// populate the stiffness matrix
dim3 interactionGrid( (p_num_interactions[0]-1)/block_dimension+1, 1, 1) ;
unsigned int indicesAndParamsMemorySize =
        6 * sizeof( unsigned int) * block_dimension;
populateStiffnessMatrix<<< interactionGrid, block_dimension,
        indicesAndParamsMemorySize >>>(qPoint.d_total_interactions,
        qPoint.d_num_quad_points, qPoint.d_interactionSetIndices,
        qPoint.d_contributionCounts,qPoint.d_contributionIndices,
        mNode.d_num_nodes, mNode.d_coordinates,
        contributionInfo.d_meshNodeAddress,
        contributionInfo.d_weight, contributionInfo.d_deriv_weight,
        d_stiffnessMatrix, qPoint.d_weights, qPoint.d_momentVector,
        qPoint.d_derivMomentVector);

readAndReportTimerValue(timeRecordFileStream, timer,
        "Populating stiffness matrix", &populatingStiffnessMatrixTime);

// check if kernel execution generated any errors
getLastCudaError("Kernel execution failed");

checkCudaErrors( cudaMemcpy( h_stiffnessMatrix,
        d_stiffnessMatrix, sizeof( float) * (num_nodes*(num_nodes+1)/2),
        cudaMemcpyDeviceToHost) );

checkCudaErrors( cudaMemcpy( h_loadVector,
        d_loadVector, sizeof( float) * num_nodes,
        cudaMemcpyDeviceToHost) );

```

```
checkCudaErrors( cudaMemcpy( h_constraintVector, d_constraintVector,
                           sizeof( float) * num_nodes,
                           cudaMemcpyDeviceToHost) );

readAndReportTimerValue(timeRecordFileStream, timer,
    "Time for reading back stiffness matrix and load and constraint vectors"
    , &dataTransferTime);

sdkStopTimer( &timer );
    printf( "Processing time: %f (ms)\n\n", sdkGetTimerValue( &timer ) );
    sdkDeleteTimer( &timer );

fclose(timeRecordFileStream);

// allocate mem for the result on host side
    float* h_odata = (float*) malloc( num_nodes * sizeof( float));
// copy result from device to host
    checkCudaErrors( cudaMemcpy( h_odata, mNode.d_coordinates,
                               num_nodes * sizeof( float) ,
                               cudaMemcpyDeviceToHost) );

float* h_qdata = (float*)
    malloc( num_quad_points * sizeof( float));
checkCudaErrors(
    cudaMemcpy( h_qdata, qPoint.d_coordinates,
               num_quad_points * sizeof( float) ,
               cudaMemcpyDeviceToHost) );

unsigned int* h_contidx = (unsigned int*)
```

```
    malloc( num_quad_points * sizeof( unsigned int));
checkCudaErrors( cudaMemcpy( h_contidx,
    qPoint.d_contributionIndices,
        num_quad_points * sizeof( unsigned int) ,
    cudaMemcpyDeviceToHost) );

unsigned int* h_qidx = (unsigned int*)
    malloc( num_quad_points * sizeof( unsigned int));
checkCudaErrors( cudaMemcpy( h_qidx,
    qPoint.d_interactionSetIndices,
        num_quad_points * sizeof( unsigned int) ,
    cudaMemcpyDeviceToHost) );

float *h_contributionWeights = (float*)
    malloc ( p_num_contributions[0] * sizeof( float));
checkCudaErrors( cudaMemcpy( h_contributionWeights,
    contributionInfo.d_weight, p_num_contributions[0] * sizeof( float) ,
    cudaMemcpyDeviceToHost) );

float *h_momentMatrices = (float*)
    malloc ( 3 * sizeof( float) * (num_quad_points));
checkCudaErrors( cudaMemcpy( h_momentMatrices,
    qPoint.d_momentMatrix, 3 * sizeof( float) * (num_quad_points),
    cudaMemcpyDeviceToHost) );

fopen_s( &debugFileStream, "C:\\Users\\Maïen\\Documents\\MSc\\Code\\gpuTimesF
fprintf_s(debugFileStream,"%d, %d, %f, %f, %f, %f, %f, %f, %f, %f\n", num
fclose(debugFileStream);

// cleanup memory
```

```
free(h_odata);
free(h_qdata);
    free(h_qidx);
free(h_contributionWeights);
free(h_momentMatrices);
free(h_stiffnessMatrix);
free(h_loadVector);
free(h_constraintVector);
freeNodeMemory(&mNode);
freeQuadPointMemory(&qPoint);
freeContributionMemory(&contributionInfo);

checkCudaErrors( cudaFree( d_stiffnessMatrix) );
checkCudaErrors( cudaFree( d_loadVector) );
checkCudaErrors( cudaFree( d_constraintVector) );
    cudaDeviceReset();

}

shrQAFinishExit(argc, (const char **)argv, QA_PASSED);

}
```