UNIVERSITY OF CALIFORNIA
Santa Barbara

Higher Order Numerical Discretization on Scattered Grids

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

by

Karthik Jayaraman Raghuram

Committee in Charge:

Professor S. Chandrasekaran, Chair

Professor J. D. Gibson

Professor J. R. Gilbert

Professor M. Gu

Professor B. S. Manjunath

September 2011

The Dissertation of
Karthik Jayaraman Raghuram is approved:

Professor J. D. Gibson

Professor J. R. Gilbert

Professor M. Gu

Professor B. S. Manjunath

Professor S. Chandrasekaran, Committee Chairperson

July 2011

Higher Order Numerical Discretization on Scattered Grids

by

Karthik Jayaraman Raghuram

To my family.

Acknowledgements

My parents Jayaraman and Sita, my wife Poornima, my sister Ramya and her family, and my brother Sathish all have been and will be great sources of moral strength to me. With God's will and my family's support, I have come this far and hope to continue much further.

My journey has been an adventurous one. Through out, I have have been guided by my advisor Prof.Chandrasekaran. Much of what I learnt has been just by listening to him. He is a store house of so much knowledge that every interaction with him is sure to teach me something. He has been a friend and mentor to me too here at UCSB and continues to inspire me to be better in whatever I do. Thanks Shiv! I also would like to thank Prof.Gilbert for teaching me sparse matrix computations and providing timely advices on many occasions. I also express my gratitude to Prof.Ming, whose advices from time to time helped me. He is a mentor and a role model to me as well. My sincere thanks to my committee members Prof.Gibson and Prof.Manjunath for taking the time to review my dissertation and suggesting improvements. I thank Stefan Borieu, the UCSB Super Computing head, for the generous super computing time, and timely help in this regard too. I thank Prof.Hrushikesh at California State University, Los Angeles. I enjoyed with awe the interaction between this ace mathematician and Shiv, while I stood an amazed bystander with a little to say now and then. His proofs have

been guiding my research and have taught me some of the mathematical formalism needed. I thank Joe Moffitt for this timely help with numerical experiments and discussions. Finally, I thank my friends for all their kindness and support throughout these years.

Vita of Karthik Jayaraman Raghuram

August 2011

Education

2008	Master of Science in Electrical and Computer Engineering, University of California, Santa Barbara.
2004	Bachelor of Engineering in Electronics and Communications, PSG College of Technology, Coimbatore, India.

Experience

2006 – 2011	Graduate Student Researcher/Teaching Assistant/Staff Research Associate, University of California, Santa Barbara.
06-2010 – 09-2010	Intern Software Development Engineer, Microsoft Corporation, Speech Core Development Team, Redmond.
06-2009 – 09-2009	Summer Research Intern, AccelerEyes Inc, Atlanta.
06-2008 – 08-2008	Summer Research Intern, AT&T Shannon Labs, Florham Park.
06-2007 – 09-2007	IPod Software Intern, Apple Inc, Cupertino.
05-2004 – 08-2006	Software Systems Engineer, Texas Instruments India Pvt Ltd, Bangalore.

Selected Publications

S. Chandrasekaran, K. R. Jayaraman, M. Gu, H. N. Mhaskar, M. Gu and J. Moffitt: "Higher Order Numerical Discretization Methods using Minimum Sobolev Norm," In *Proc. International Conference on Computational Science, (ICCS)*, June 2011.

S. Chandrasekaran, K. R. Jayaraman, J. Moffitt, H. N. Mhaskar and S. Pauli : "Minimum Sobolev Norm Schemes and Applications in Image Processing," In *Proc. SPIE Imaging Sciences and Technology, (SPIE IS&T)*, Jan 2010.

G. D. Bhokare, J. R. Karthik, V. M. Gadre: "Modified EW Coding using the M-Band Wavelet Transform and its applications to Image Compression,", In *IEEE Conference on Statistical Signal Processing, (IEEE SSP)*, July 2006.

S. Chandrasekaran, K. R. Jayaraman, M. Gu, H. N. Mhaskar, M. Gu and J. Moffitt: "Minimum Sobolev Norm Methods for BiHarmonic Type Equations," In preparation *Special Issue of the Journal of Computational and Applied Mathematics*, To be Submitted July 2011.

S. Chandrasekaran, K. R. Jayaraman and H. N. Mhaskar: "Minimum Sobolev norm interpolation with trigonometric polynomials on the torus," In preparation

S. Chandrasekaran, K. R. Jayaraman and H. N. Mhaskar: "High order finite difference schemes for solving elliptic partial differential equations," In preparation

Patents

K. Sanjeev and K. R. Jayaraman "Image watermarking based on sequency and wavelet transforms", No. 7742619

K. J. Raghuram and P. Lafon "Techniques for Efficient Dithering", No. 7864191

Awards and Honors

Outstanding Teaching Assistant Award for 2010–2011 by the Department of Electrical and Computer Engineering, UC Santa Barbara

Academic Senate Doctoral Student Travel Award for 2011

Selected to attend the Gene Goloub International Summer School on Numerical Linear Algebra with full financial support, Brindisi, Italy June 2011.

Gold Medalist and Top Rank holder, Department of Electronics and Communications, PSG College of Technology, May 2004.

Best Outgoing Student Award, Alumni Association of the Department of Electronics and Communications, PSG College of Technology, May 2004.

Abstract

Higher Order Numerical Discretization on Scattered Grids

Karthik Jayaraman Raghuram

Beginning with the suppression of Runge Phenomenon which arises in equi-spaced polynomial interpolation, we present the Minimum Sobolev Norm interpolation technique which we generalize to produce Finite Difference (MSNFD) type weights for differential operators. It is shown that these weights yield higher order approximations to these operators with increasing stencil sizes, and that the idea generalizes to non-uniform grids easily. Thus perhaps for the first time, a systematic means of producing higher order FD weights on irregular grids is discussed. After the basic theoretical discussions on the interpolation process and local convergence of weights, the idea of solving elliptic PDEs using these MSNFD weights is discussed. A set of extensive numerical experiments on standard second order problems as well as the Exterior Laplace problem and the biharmonic equations are discussed. In the absence of a theory for global convergence of the PDE solution, various numerical results that validate our claim of a higher order FD method on non-uniform grids are presented. The concluding section discusses the short-comings associated with solving ill-conditioned problems such as the bi-

harmonic equation and the scattering problem. The idea of lifting using Div-Curl systems is presented as a possible future work and extension.

Professor S. Chandrasekaran
Dissertation Committee Chair

Contents

List of Figures

List of Tables

Chapter 1

Introduction

One of the most surprising facts in the theory of interpolation and approximation is that the simplest and the most natural approach to synthesis leads to failure, or rather, to an impossibility

Philip J. Davis - Author of "Interpolation and Approximation"

1.1 Overview

This thesis is primarily concerned with the efficient solution of partial differential equations (PDEs), systems in which one tries to recover an unknown function from measurements of its derivatives. In order to solve such systems, we use a discrete representation of the unknown function to produce an algebraic system of equations. These are then solved using standard or specialized linear algebra techniques. A method of solving PDEs (a PDE Solver) is regarded efficient if it uses as

few measurements and as few unknown samples to get to a target accuracy. The minimum number of such samples is prescribed by the Nyquist criterion, which requires at least two points per wavelength corresponding to the largest frequency in the solution to the PDE. Hence an efficient solver would be one whose operating point is as close to this rate as possible. However, a more efficient representation may lead to increased complexity in recovering the solution as is the case with signal compression. Hence, a more realistic measure is just the wall-clock time taken to get to a target accuracy i.e. the time taken to solve the discrete system obtained from the PDE.

In solving a PDE, we have a discrete representation of the unknown function. To illustrate this idea, we consider a simple differential equation $u' = f$. Let us discretize the solution at N points $u(x_i) = u_i$. Let us also suppose that at some point y, we have some weights w_i such that

$$\sum_{i=0}^{K} w_i(y) u_i \approx u'(y). \tag{1.1}$$

Then if we discretize the PDE at M points y_i, we can write the discrete system

$$Wu = f, \tag{1.2}$$

where W is an $M \times N$ matrix that has the weights $w_i(y_j)$ in row j. The important matters for us are the generation of accurate weights w_i and solution to the system. As another example, the Helmholtz equation specifies a combination of the second

partial derivatives and a scaled function value. This equation arises naturally in solving for time snapshots of a traveling wave. If ∇ denotes the vector gradient operator $\begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{bmatrix}$, then the two dimensional Helmholtz equation is given by:

$$\nabla.\nabla u + k^2 u = f$$

for any real k. Another example is the diffusion-advection equation, which involves a combination of second and first partial derivatives,

$$\nabla.A\nabla u + \nabla.(\epsilon u) = f.$$

There are global and local approaches to solving PDEs. The former uses all the underlying samples and equations at once to recover the solution in the entire domain. While this seems to be the most natural approach, and several powerful global methods exist, this may not be good idea for many reasons. For example, if the solution to the underlying PDE is mostly smooth, except in a small neighborhood, the solution has a low Nyquist rate except in a small locality. For example, a PDE modeling fluid flow would require high Nyquist rate at regions where the flow is turbulent. Similarly, a wave traveling through a waveguide would require highly Nyquist rate at sharp corners. A global approach however could be oblivious to this fact, and require a very high sampling rate to resolve the resolution compared to a local solver, which could exploit a selectively high

sampling rate in the vicinity of the non-smooth neighborhood. An example of a global method is a spectral method, that solves for the Fourier coefficients of the solution in entire domain.

In addition to the above possible inefficiency, global approaches lead to dense systems of equations. If the structure of such systems is known, such as low-numerical rank off-diagonal blocks, then these can be solved efficiently using techniques such as Fast Multipole Methods [20], and Hierarchical Semiseparable Representations [5]. But we still end up having to solve for a larger number of spectral coefficients since we are solving for a high-frequency solution. Although global methods tend to have high rates of convergence, their numerical stability and the need for specialized solvers limits their utility.

A local approach to a PDE Solver on the other hand, uses only a neighborhood of samples to reconstruct the underlying solution at a given point. Such methods work well with adaptive gridding schemes that selectively boost the sampling rate in the vicinity of specific regions. Local discretizations lead to sparse banded systems and such systems can be efficiently solved using direct or iterative methods. The popular Finite-Difference (FD) and Finite Element (FEM) methods both employ local discretizations. A related advantage of a local method is that the accuracy (or inaccuracy) in one region does not affect the convergence in the other

regions. Also, using adaptive gridding, one can selectively boost the accuracy in specific regions if needed.

Note that our goal here is to have an effective discretization strategy, showcased in a PDE Solver. For this an FD method is used, although a similar FEM setup may well be in order. An FD method has an elegance to it in that it is never bogged down by details. Its primary ideas are just those of sampling, approximation and reconstruction. Further, FD solvers are very robust in the sense of the types of PDEs they can handle. An FD solver can be built to handle almost any type of PDE with minimum modifications. In my recent discussion with some eminent parallel computing experts it also came to light that super computing architectures, and computing architectures in general are most suited to FD type PDE Solvers than FEM type PDE Solvers. We therefore discuss and work with FD type of PDE Solvers.

To summarize the main goal, this thesis presents an effective process of numerical discretization, and its application in a Higher Order FD PDE Solver.

1.2 Interpolation and Approximation

This section introduces the basic ideas in numerical discretization. The related mathematical notation is set up, and will be used in the remainder of this thesis.

Interpolation may be regarded as the process of reconstructing a function, given its sample values. Interpolation helps us *read between* samples by using the function thus reconstructed. Often, the reconstructed function takes the form of a polynomial which approximates the actual function. The polynomial is chosen so that it matches the prescribed function values. Let us consider interpolation in one dimension. Let $x_1, x_2, x_3, ..., x_N$ denote N points in the interval $[-1, 1]$. Let $f_1, f_2, f_3, ..., f_N$ denote the prescribed sample values of some function $f(x)$. Suppose we wish to use polynomials to approximate $f(x)$ given f_i. An important question that arises is : "*What is the order of the interpolating polynomial?*".

Traditional wisdom in interpolation is to choose a polynomial whose order is the number of samples being interpolated, N. An interpolation to N samples using an order N polynomial is termed Lagrange interpolation. The interpolating polynomial (interpolant) in this case is unique since the polynomial is completely specified by N coefficients, and there are as many interpolating conditions. However, this process has serious disadvantages and we resort to polynomials of order larger than N. Let M denote the order of the interpolating polynomial $p(x)$. Then, the interpolation constraint is given by

$$p(x_i) = f(x_i), \ i = 1, 2, ...N. \tag{1.3}$$

Let $p(x)$ be represented in a Chebyshev basis,

$$p(x) = \sum_{m=0}^{M-1} a_m T_m(x), \tag{1.4}$$

where $T_m = \cos m \cos^{-1}(x)$, the order m Chebyshev polynomial. In general, we do not restrict ourselves to the Chebyshev basis, although this choice has certain advantages described later. The interpolation constraints then become

$$\sum_{m=0}^{M-1} a_m T_m(x_i) = f(x_i), i = 1, 2, ...N. \tag{1.5}$$

Since we have a system of equations, it is natural to consider the Linear Algebraic representation as follows. Let V denote the matrix as below,

$$V = \begin{bmatrix} T_0(x_1) & T_1(x_1) & T_2(x_1) & ... & T_{M-1}(x_1) \\ T_0(x_2) & T_1(x_2) & T_2(x_2) & ... & T_{M-1}(x_2) \\ . & . & . & . & . \\ . & . & . & . & . \\ T_0(x_N) & T_1(x_N) & T_2(x_N) & ... & T_{M-1}(x_N) \end{bmatrix}.$$

Let $\mathbf{a}, \mathbf{f}$ denote the vector of coefficients and prescribed samples as below.

$$\mathbf{a} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ . \\ . \\ . \\ a_{M-1} \end{bmatrix}, \quad \mathbf{f} = \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ . \\ . \\ . \\ f_{N-1} \end{bmatrix}$$

The linear algebraic interpolation problem is then to find $\mathbf{a}$ such that,

$$V\mathbf{a} = \mathbf{f}. \tag{1.6}$$

For Lagrange interpolation, $N = M$. Hence, V is a square matrix of full rank. Hence the system of equations has a unique solution for $\mathbf{a}$ as expected. If $M < N$, then, we have a tall-skinny over determined system and there *may* not be an exact solution. However, we may still produce an approximant that has minimum least square error by solving the above system. This is often the case when several noisy measurements are available. If $M > N$, we have infinitely many solutions to the system. In this case, one is left with the option of picking particular solutions, that satisfy some additional constraints. One traditional way of solving the above system is to pick that $\mathbf{a}$ whose 2-norm is minimum. If we interpret $\mathbf{a}$ as the 'Fourier' coefficients, then this corresponds to a minimum energy interpolant.

1.3 Runge Phenomenon

In this section, we discuss an important but perhaps little known observation in interpolation theory. This observation is the main reason one seldom sees polynomial approximations over neighborhoods larger than a few samples, as with Splines. Consider a function $f(x) = \frac{1}{1+25x^2}$ in the interval $[-1, 1]$. Figure 1.1 in page 9 shows the prescribed samples at $N = 5, 9$ equidistant points and the corresponding Lagrange interpolant. Surprisingly, the 9 point interpolant is much worse than the 5 point interpolant in approximating $f(x)$. These wild oscillations and the resultant divergence of the interpolant from the underlying function is known as Runge Phenomenon.

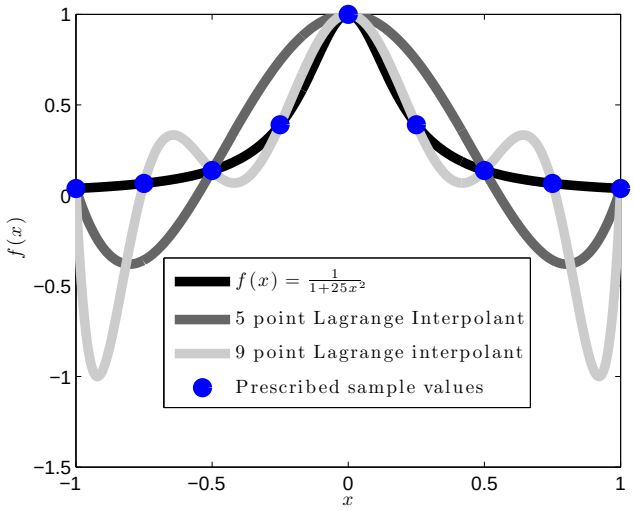


Figure 1.1: One dimension Lagrange Interpolation

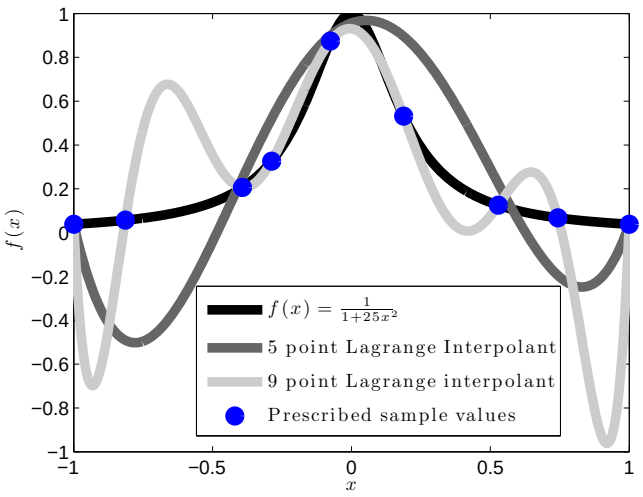


Figure 1.2: One dimension Lagrange Interpolation

This phenomenon was first studied by Runge, and hence called so [11]. Runge's observation was that equispaced interpolation of the function $\frac{1}{1+x^2}$ in $|x| \leq 5$ diverges in the interval $|x| > 3.63$. The divergence is attributed to the complex singularity at $\pm i$. Beginning with Runge, a series of negative results followed revealing that a convergent interpolatory process is not obvious and that Lagrange Interpolation on equispaced points can in fact even diverge everywhere for a continuous function's samples. Figure 1.2 depicts the Runge phenomenon on scattered points.

Table 1.1 depicts Runge's observation for $f(x) = \frac{1}{1+25x^2}$. As the number of interpolating points and hence the order of the Lagrange interpolant is increased

from 5 to 50, errors at $x = -0.821, 0.91$ grow unbounded. This illustrates the fundamental bottleneck with using Lagrange interpolation on equispaced nodes.

At this point we stop and ask the important question *"Why are we interested in higher order interpolation?"*. While highly accurate interpolation is definitely useful, the main application of interest for us is that of solving PDEs. Given a linear interpolatory process, it is possible to systematically move from interpolation to production of FD weights of arbitrary width to provide increasing accuracy. This shall be proved constructively using the interpolatory process we call the Minimum Sobolev Norm (MSN) interpolation. Through MSN, we construct highly accurate weights using increasing number of interpolation samples as shall be shown later.

As another example of the Runge phenomenon, consider a two dimensional function $f(x, y) = \frac{1}{1+100x^2+100y^2}$ as shown in Figure 1.3 . The Lagrange interpolant to this function at 11 equispaced nodes is shown in Figure 1.4. One again, we see that Runge Phenomenon renders Lagrange interpolation on an equispaced grid useless! We therefore need an efficient interpolation mechanism, that could be used to produce accurate FD weights to solve PDEs.

Table 1.1: Point-wise Lagrange interpolation error

x	Order 5	Order 9	Order 50
-8.31e-01	4.27e-01	5.57e-01	5.47e+00
-5.25e-01	5.37e-02	5.32e-02	1.28e-04
1.21e-01	2.06e-01	8.75e-02	1.11e-06
2.60e-01	3.54e-01	1.45e-02	8.16e-07
9.10e-01	3.14e-01	1.04e+00	8.89e+02

Figure 1.3: Two dimensional Runge phenomenon.

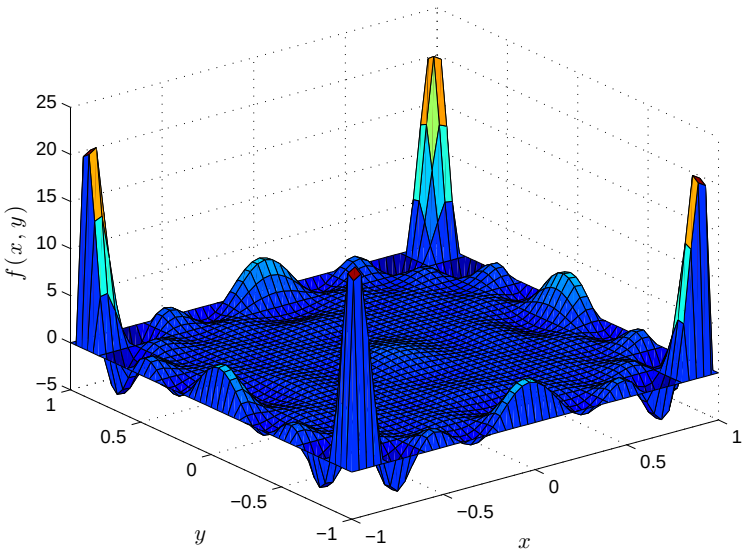
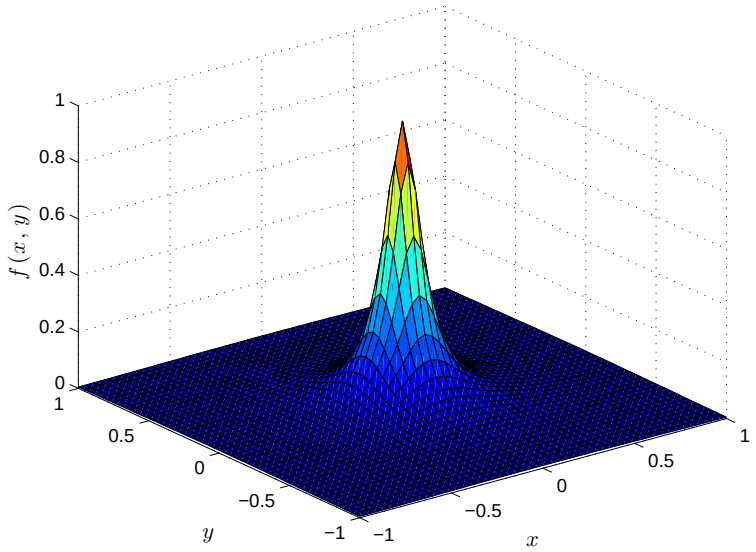


Figure 1.4: A two dimensional Runge type function



1.4 Traditional Finite Difference Weights

In this section we consider traditional methods of producing FD weights. By FD weights we refer to a discrete form of differential operators, that we could use to solve PDEs. Such weights are usually computed using a Taylor series expansion of the underlying solution. While this may be useful in one dimension and on equispaced (regular) grids, using Taylor series has some problems in higher dimensions, particularly on scattered grids. For scattered grids, it is useful to think of FD weights as computed from a Lagrange interpolant. But such weights are not useful over large stencils due to Runge phenomenon; since the underlying interpolant diverges, so would the weights! We expand on these key concepts

below. In addition, the weights themselves are not uniquely specified using Taylor expansions as shall be shown below.

Consider the standard five-point stencil given in Figure 1.5. Let the grid points be given by $(x_0, y_0),\ (x_0 \pm h, y_0),\ (x_0, y_0 \pm h)$. Suppose we wish to compute weights α_i such that,

$$\begin{aligned} f_{xx} + f_{yy} &= \alpha_0 f(x_0 - h, y_0) + \alpha_1 f(x_0, y_0) + \alpha_2 f(x_0 + h, y_0) + \\ &\quad \alpha_3 f(x_0, y_0 - h) + \alpha_4 f(x_0, y_0 + h), \end{aligned} \tag{1.7}$$

where $f_{xx} = \frac{\partial^2 f}{\partial x^2}$ and so on. Taylor series expansion of each of the above terms leads to the following system of equations,

$$f(x_0, y_0) = f(x_0, y_0) \tag{1.8}$$

$$\begin{aligned} f(x_0 \pm h, y_0) &= f(x_0, y_0) \pm h f_x(x_0, y_0) + \frac{h^2}{2!} f_{xx}(x_0, y_0) \pm \frac{h^3}{3!} f_{xxx}(x_0, y_0) \\ &\quad +... \end{aligned} \tag{1.9}$$

$$\begin{aligned} f(x_0, y_0 \pm h) &= f(x_0, y_0) \pm h f_y(x_0, y_0) + \frac{h^2}{2!} f_{yy}(x_0, y_0) \pm \frac{h^3}{3!} f_{yyy}(x_0, y_0) \\ &\quad +... \end{aligned} \tag{1.10}$$

Using equations 1.8 through 1.10 in equation 1.7, we have

$$\begin{aligned} f_{xx} + f_{yy} &= \left(\sum_{i=0}^{4} \alpha_i\right) f(x_0, y_0) + h(\alpha_2 - \alpha_0) f_x(x_0, y_0) + \\ & h(\alpha_4 - \alpha_3) f_y(x_0, y_0) + \frac{h^2}{2!}(\alpha_2 + \alpha_0) f_{xx}(x_0, y_0) + \\ & \frac{h^2}{2!}(\alpha_3 + \alpha_4) f_{yy}(x_0, y_0) + \frac{h^3}{3!}(\alpha_2 - \alpha_0) f_{xxx}(x_0, y_0) + \\ & \frac{h^3}{3!}(\alpha_4 - \alpha_3) f_{yyy}(x_0, y_0) + ... \end{aligned} \tag{1.11}$$

To solve for the α_i, we use equation 1.11 to setup appropriate constraints on the α_i so that the left and right hand sides of the equation match to the desired accuracy. To uniquely define these weights, so that the error bound on them is known, we can at most specify five constraints. We set lower order derivative coefficients to zero, giving,

$$\sum_{i=0}^{4} \alpha_i = 0 \tag{1.12}$$

$$\alpha_2 - \alpha_0 = 0 \tag{1.13}$$

$$\alpha_4 - \alpha_3 = 0 \tag{1.14}$$

In addition to these three constrains that make sure the error is at least $O(h^2)$, we further want the coefficients of the second order terms be one. We therefore get

$$\frac{h^2}{2!}(\alpha_2 + \alpha_0) = 1 \tag{1.15}$$

$$\frac{h^2}{2!}(\alpha_4 + \alpha_3) = 1. \tag{1.16}$$

We solve the exact system defined by equations 1.12 through 1.16 to get the weights as

$$(\alpha_i)_{i=0}^4 = \left(\frac{1}{h^2}, -\frac{4}{h^2}, \frac{1}{h^2}, \frac{1}{h^2}, \frac{1}{h^2}\right).$$

Using these weights, we can see that the error in approximating $f_{xx} + f_{yy}$ is $\frac{h^2}{12} f_{xxxx} + \frac{h^2}{12} f_{yyyy} + O(h^4)$.

A few important observations are in order. Suppose we wish to use a nine point stencil instead, as shown in figure 1.6. In this case, we have nine unknowns corresponding to the weights. However, there are only six derivative terms in the expansion up to second order, namely, $(f, f_x, f_y, f_{xx}, f_{xy,f_{yy}})$. Hence, there is *no* unique way to pick these weights. If we decide to include a higher order term and set its coefficients to zero, we would have coefficients corresponding to $(f_{xxx}, f_{yyy}, f_{xxy}, f_{yyx})$, giving a total of ten equations for nine unknowns. There is no solution that satisfies this over constrained system in general. For any of the nine point weights we have bounded the error only to the same order as with the five point stencil, there is no gain in using these additional weights, expect for particular advantages with the Laplacian operator. However, this is only the beginning of the complication.

The equations themselves and hence the coefficients depend on the grid spacing h. For irregular grids, the Taylor expansion and hence these equations need to locally setup. So the elegance of the pre-computed weights disappears once the

grid becomes irregular. Rather than consider the Taylor series expansion to the solution and compute FD weights, on may also use interpolation to compute FD weights. To illustrate this, consider the five point one-dimensional discretization for the first derivative. In this case, we are given $(f(x_0), f(x_1), f(x_2), f(x_3), f(x_4))$ to approximate the derivative at some point x as shown in figure 1.7. The Lagrange interpolant to these five samples is given by,

$$p(x) = \sum_{i=0}^{4} f(x_i) l_i(x) \tag{1.17}$$

$$l_i(x) = \Pi_{j=0, j\neq i}^{4} \frac{x - x_j}{x_i - x_j}. \tag{1.18}$$

Note that the point-wise interpolation weights at x are given by $l(x)$. In order to construct weights for f'', we simply apply the same operator to the interpolant. Since this is a Linear interpolatory operator, the weights simply correspond to l_i'' as seen in equation 1.19. Thus, in the case of non-uniform grids, one can construct the Lagrange interpolant, and use it to obtain the weights for linear operators.

$$p''(x) = \sum_{i=0}^{4} f(x_i) l''_i(x) \tag{1.19}$$

However, as seen earlier, Lagrange interpolation is not guaranteed to converge due to Runge phenomenon, even on equispaced samples! Hence, Lagrange interpolation cannot be used to produce the correct FD weights.

To summarize our observations with traditional FD weights, we noted that the process of specifying weights using Taylor series expansion is inefficient, and

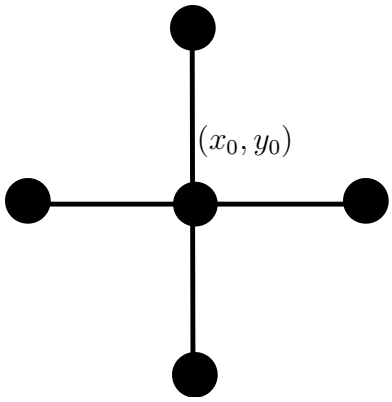


Figure 1.5: Five point stencil

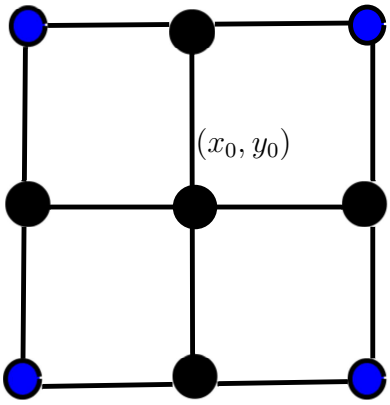


Figure 1.6: Nine point stencil

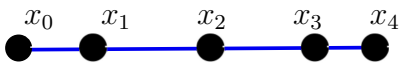


Figure 1.7: 1D Five point stencil

leads to grids that are of the 'cross' form since there is no way to harness other neighbors effectively. This process becomes complicated for non-uniform grids. Further, if we consider the FD weights as arising from a Lagrange interpolant over increasing number of samples, then Runge phenomenon prevents us from finding accurate weights! Thus the problem of finding FD weights on non-uniform grids remains and we address this problem using the idea of Minimum Sobolev Norm Interpolation. The rest of this thesis is concerned with discussing the MSN method in Chapter 2, a specialized method for solving ill-conditioned Weighted Least-

Squares systems called CODA in Chapter 3, and PDE applications in Chapter 4 onwards.

Chapter 2

Minimum Sobolev Norm Interpolation

Common sense is the collection of prejudices acquired by age eighteen.
Albert Einstein

2.1 The Route to MSN

In the previous chapter, Runge Phenomenon was pointed out to be a key hurdle to be overcome, if one hopes to produce higher order FD weights. We revisit it, to gain further intuition about resolving it. Consider the function $\frac{1}{1+ax^2}$ in the interval $[-5, 5]$. This function has singularities at $\pm ai$ on the complex plane. In order to see the effect of this singularity on the Runge Phenomenon, consider

figures 2.1 through 2.3. We observe that as the singularity is moved closer to the real plane, the oscillations increase. Epperson discusses the Runge phenomenon and the various complications in Lagrange interpolation in [17]. Since we have no control over the location of the singularity of the function we are trying to interpolate, this observation is of no use. Instead, what gives a better insight is that of interpolation error analysis. As discussed in [17],[11], we see that the interpolation error depends on the derivatives of the function being interpolated. For the Runge function, derivatives become increasingly large in magnitude. Such increasing error bounds permit these wild oscillations. These derivatives also increase as the singularity in the function gets closer to the real plane as mentioned earlier in this section. We do not get into further discussions of the Runge phenomenon, as a more detailed look requires complex analysis and the theory of residues.

The Runge phenomenon exhibits itself as oscillations in the interpolant. A traditional approach to alleviate Runge phenomenon is to use specific configurations of grid points. It is known that use of the zeros of the Chebyshev polynomial as the grid points for interpolation suppresses the Runge phenomenon for the function $\frac{1}{1+ax^2}$. However, there exist continuous functions for which the Lagrange interpolant diverges everywhere [11] even using Chebyshev nodes as interpolating grid points. This result is due to G. Grünwald and (independently) J. Marcinkiewicz, who showed that there is a continuous function for which interpolation at Cheby-

Figure 2.1: Runge function and Lagrange interpolant, poles at $\pm\sqrt{0.1}i$

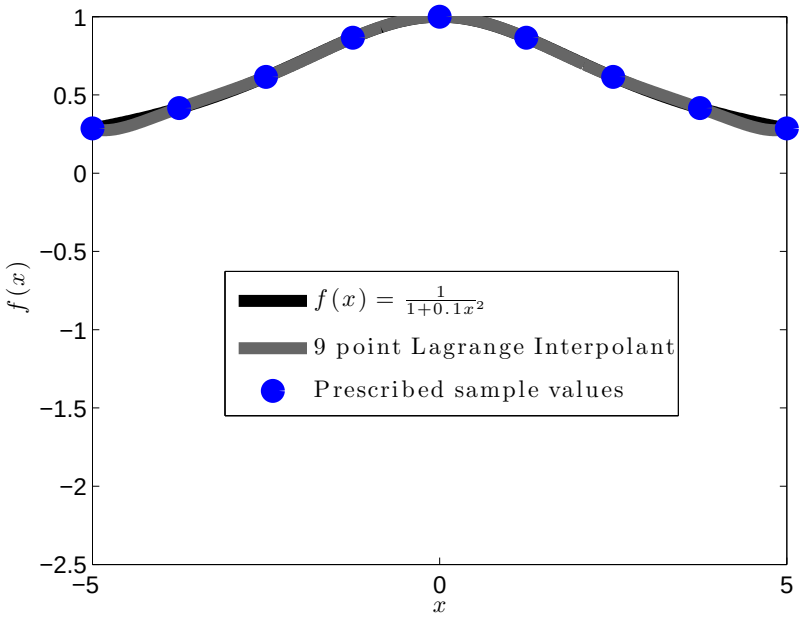


Figure 2.2: Runge function and Lagrange interpolant, poles at $\pm i$

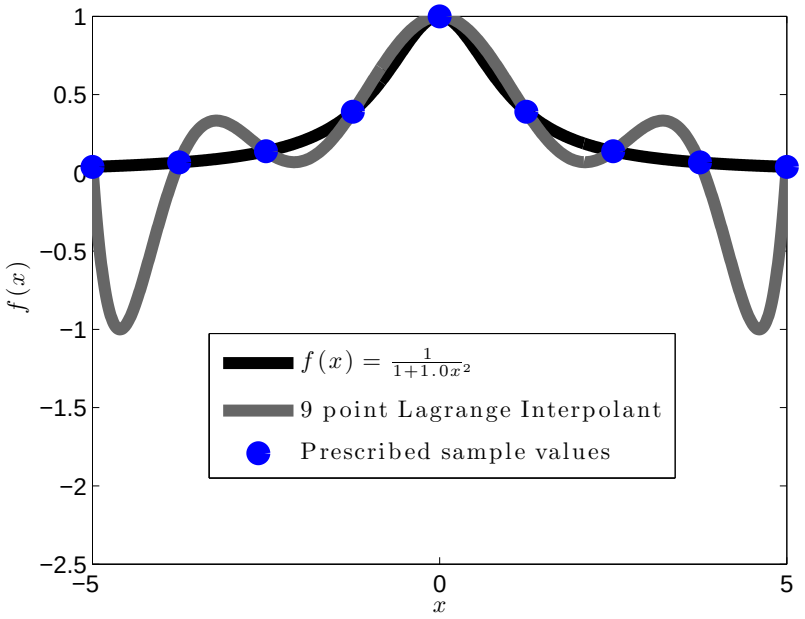


Figure 2.3: Runge function and Lagrange interpolant, poles at $\pm 10i$

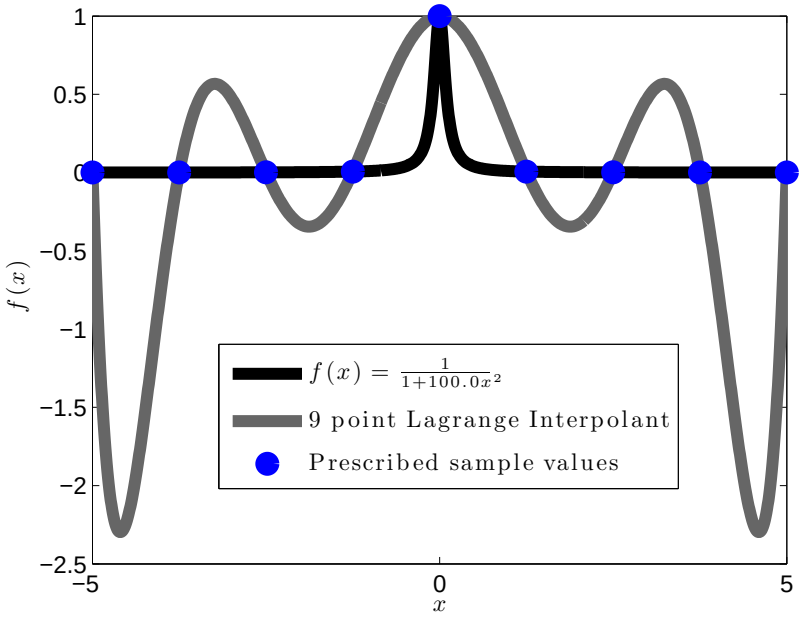
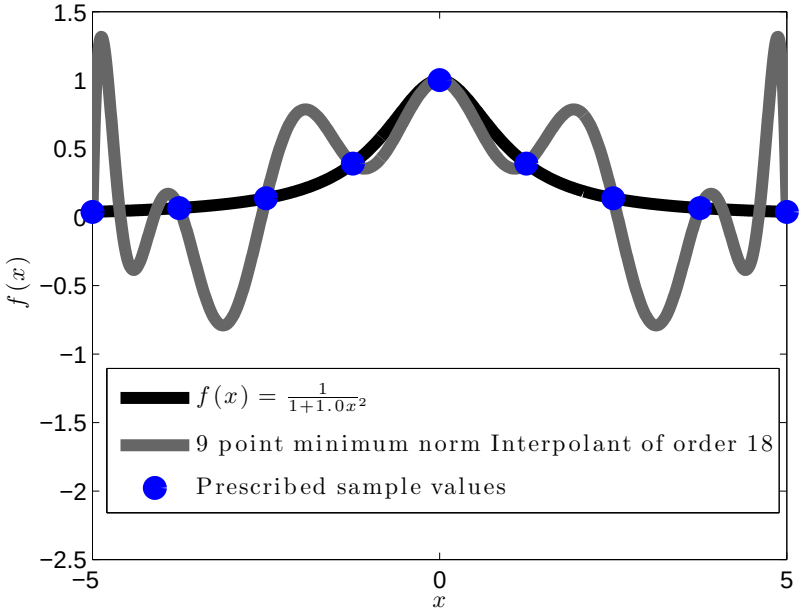


Figure 2.4: Runge function and minimum 2 norm interpolant of order 18



shev zeros does not converge. However, such an f must be rough enough so as to *not* satisfy the Dini-Lipschitz condition

$$\lim_{N\to\infty} \omega\left(\frac{1}{N}\right)\log N = 0,$$

where ω denotes the *modulus of continuity* of f. Thus, there is not a unique configuration of grid points that can alleviate Runge phenomenon during Lagrange interpolation. Even if available, such a configuration would be of little or no use, since in the context of PDE Solvers, the grid configuration is determined by several other factors. Therefore, the only degree of freedom we have is the choice of the interpolant, which is uniquely specified for Lagrange interpolation, i.e., interpolation using a polynomial of the same order as the number of grid points. Naturally, this leads us to the question of using polynomials of order larger than the number of grid points.

As seen in the introductory section on interpolation, if we let the order M of the interpolant be larger than the number of grid points N, we have an under-constrained system of equations $V\mathbf{a} = f$. Since there are infinitely many choices possible, one can pick an interpolant with the *desired* property i.e. Runge suppression. Suppose we pick the naive minimum two norm solution for interpolation, we see that we still have not suppressed the Runge oscillations as in figure 2.4. This of course makes sense because a function with a small two norm may still have

rapidly growing derivatives, something directly related to Runge phenomenon. We therefore need to consider interpolants whose derivatives are controlled.

Fejer, the Hungarian mathematician provided for an interpolation scheme in which he used a polynomial of order twice the number of grid points. He used the additional degrees of freedom to set the first derivative of the interpolating polynomial to zero at the grid points, thereby uniquely specifying the interpolant. Such schemes in which both the function value and its derivatives are specified fall into the category of Hermite interpolation. Splines too fall in this category [16], [40]. Splines are piece-wise polynomials, with derivatives up to a particular order matched at the boundary of these pieces. The over all interpolant is as smooth as the number of derivatives matched among neighboring pieces. This idea of matching point-wise derivatives becomes increasingly complicated, and in higher dimensions and arbitrary surfaces, such matching is very hard to specify [30]. Also, there exist known examples of functions for which Fejer's approach fails to converge as well [37]. Unfortunately setting the derivative of the interpolant to be zero at the interpolation points reduces the accuracy of interpolation for finite values of N. Also, this scheme fails on equispaced points. This is a result of D. L. Berman [37, Theorem 6.1]. However the key idea of using a polynomial of order larger than the number of grid points remains to be used, albeit correctly. Bernstein proposed a famous alternative scheme. He showed that by giving up

the interpolation property it is possible to produce a sequence of polynomials that converge uniformly for all continuous f with equispaced points. Since the scheme of Bernstein converges slowly it has not been of much practical interest either. However, it is to be compared with Fejer's scheme which does not converge for equispaced points. J. Szabados [37, Theorem 2.8] shows a non-linear interpolatory process that converges at equispaced grid points for all continuous function. But a linear interpolatory process is what our goal is, given that we would like to produce FD weights using the process. The Lozinskii and Kharshiladza theorem provides a near complete justification of all these observations. As per the theorem, no linear interpolatory process for producing polynomials of degree N, that also preserves all polynomials of degree N, can converge for all continuous functions. Fejer and Bernstein do not preserve all polynomials and hence produce convergent interpolatory processes; however both have practical short comings that limits their use.

Thus far, our discussions have been about divergence of polynomial interpolation to *continuous* functions. A key inference from the preceding discussions is one may need to relax this condition of convergence for all continuous functions, and instead work with a class of functions with known smoothness properties. For example, one could consider the Sobolev space $\mathbf{H}^s$, of functions whose Sobolev

norm, defined by,

$$\|f\|_s^2 = \|f^s\|_2^2 = \int_{-\infty}^{\infty} |f^s(x)|^2 dx < \infty, \tag{2.1}$$

where f^s refers to the s^{th} derivative. Note that the Sobolev space we have considered is *weak* in the sense that a true Sobolev norm would consider all derivatives of f less that s as well. We take a moment to quickly look at the Fourier series of such functions. One can also generalize the above notation for all real s if we define derivative through Fourier coefficients. Suppose we construct the Fourier series for the function $f \in \mathbf{H}^s$ defined over the interval $[-1, 1]$,

$$\hat{f}_n = \int_{-1}^{+1} f(x) e^{-jn\pi x} dx. \tag{2.2}$$

Consider the reconstruction equation,

$$f(x) = \sum_{n=-\infty}^{+\infty} \hat{f}_n e^{jn\pi x}. \tag{2.3}$$

The Fourier series may not converge, since at the periodic extension of f may not even be continuous even though f is differentiable. We therefore need a clever trick that would preserve the nicety of f.

We map the interval $[-1, 1]$ onto a circle using the mapping $x = \cos\theta$. This maps the interval $x \in [-1, 1] \rightarrow \theta \in [\pi, 2\pi]$, a semi-circle. Note that $f(\cos\theta) = f(\cos(\theta + 2\pi))$, 2π periodic and still s times continuously differentiable. If we

now consider the Fourier coefficients of f^s,

$$\begin{aligned} f^s(\theta) &= \sum_{n=-\infty}^{+\infty} \hat{f}_n \frac{d^s}{d\theta^s} e^{jn\pi\theta} \\ &= \sum_{n=-\infty}^{+\infty} \hat{f}_n n^s j^s e^{jn\pi\theta} \end{aligned} \tag{2.4}$$

Using 2.1 and 2.4, we see that

$$\|f\|_s^2 = \|f^s\|_2^2 = \sum_{n=-\infty}^{+\infty} |\hat{f}_n|^2 n^{2s} < \infty \tag{2.5}$$

$$\Rightarrow \hat{f}_n = O\left(\frac{1}{n^{s+k}}\right) \quad k > 0 \tag{2.6}$$

From 2.6 we see that specifying a function's smoothness through bounded energy in derivatives is equivalent to specifying a rate of decay of its Fourier coefficients *on the circle*.

2.2 Minimum Sobolev Norm interpolation

The MSN method draws from Fejer's approach and attempts to control derivatives of the interpolant. But instead of point-wise control over the derivative, we control the norm of the derivative uniformly throughout the interval of interpolation. The last discussions in the previous section hints at the fact that we shall be achieving this by specifying a rate of decay of the Fourier coefficients of the interpolant on the circle. For our setup, we first consider the one dimensional case, in which the grid points are given by $x_0, x_1, x_2, ...x_{N-1}$ in increasing order in $[-1, 1]$.

We assume uniqueness of these grid points. Further, we let $\mathbf{f}$, denote the vector of function values at these grid points; $\mathbf{f}_i = f(x_i)$. Let the Chebyshev Vandermonde matrix be indicated by V. The choice of Chebyshev instead of monomial basis has important advantages. Firstly, the Chebyshev basis is a well-conditioned basis (Gautschi). Secondly, under the mapping $x = \cos\theta$, the Chebyshev basis transforms into an orthogonal Fourier cosine basis. In this case, it is very simple to express the Sobolev norm of the interpolant through just a weighted two norm of the Fourier coefficients of the interpolant as shown in equation 2.10. Let $p_M(x)$ denote the M^{th} order Chebyshev polynomial given by

$$p_M(x) = \sum_{m=0}^{M-1} a_m T_m(x). \tag{2.7}$$

$$\Rightarrow p_M(\cos\theta) = \sum_{m=0}^{M-1} a_m \cos m\theta \tag{2.8}$$

and let $\mathbf{a}$ denote the vector of Fourier coefficients.

Using 2.1 and 2.4

$$\|p_M(\theta)\|_s^2 = \|\frac{d^s}{d\theta^s} p_M(\theta)\|_2^2 \tag{2.9}$$

$$= \sum_{m=0}^{M-1} |a_m|^2 m^{2s} \tag{2.10}$$

$$\approx \|D_s \mathbf{a}\|_2^2. \tag{2.11}$$

Note that in the above equation, we introduced a diagonal matrix D_s, defined as

$$D_s = \begin{bmatrix} 1^s & 0 & ... & 0 & 0 \\ 0 & 2^s & ... & 0 & 0 \\ . & . & ... & . & . \\ 0 & 0 & ... & 0 & M^s \end{bmatrix}_{M \times M} . \tag{2.12}$$

we define D_s in the above manner so that it is invertible, for the sake of ease of analysis. Later on , we shall consider the case when we have a need to choose D_s to be singular. In general, D_s can be chosen to be any suitable matrix so long as it serves our purpose of representing a Sobolev weight for the Fourier coefficients.

With this setup, we are ready to state the main problem of MSN Interpolation. We wish to find interpolating polynomials specified by the coefficients $\mathbf{a}$ so that, the s^{th} Sobolev norm as defined by equation 2.10 is minimum. In our notation, this is the optimization problem specified by equation 2.13. Through this we are finding an interpolant such that its Sobolev norm on the semi-circle is minimum.

$$\mathbf{a}^* = \arg \min_{\mathbf{a}: V\mathbf{a}=\mathbf{f}} \|D_s \mathbf{a}\|_2^2 \tag{2.13}$$

2.2.1 Solution when D_s is invertible

$$V\mathbf{a} = \mathbf{f} \tag{2.14}$$

$$\Rightarrow VD_s^{-1}D_s\mathbf{a} = \mathbf{f} \tag{2.15}$$

Equation 2.15 is an under constrained system with infinitely many solutions. However, our unique solution is the one with minimum norm as defined by equation 2.13.

$$(D_s\mathbf{a})^* = (VD_s^{-1})^\dagger\mathbf{f} \tag{2.16}$$

$$\Rightarrow \mathbf{a}^* = D_s^{-1}(VD_s^{-1})^\dagger\mathbf{f} \tag{2.17}$$

$$= D_s^{-2}V^T(VD_s^{-2}V^T)^{-1}\mathbf{f}. \tag{2.18}$$

The interpolant at some $x \in [-1, 1]$ is given by the sum

$$\begin{aligned} p_M(x) &= \sum_{m=0}^{M-1} a_M T_m(x) \\ &= V(x)\mathbf{a}^*, \end{aligned} \tag{2.19}$$

where $V(x) = [\ T_0(x) \quad T_1(x) \quad ... \quad T_{M-1}(x) \quad T_{M-1}(x)\ ]$.

Equation 2.19 can be rewritten as,

$$p_M(x) = V(x)D_s^{-2}V^T(VD_s^{-2}V^T)^{-1}\mathbf{f} \tag{2.20}$$

$$= \sum_{i=0}^{N-1} K(x, x_i)\mathbf{f}_i, \tag{2.21}$$

where $K(x, y)$ is the MSN interpolation kernel.

2.2.2 Solution for a general D_s

In this section we consider the solution when D_s is not invertible. Through the matrix D_s, we are imposing Sobolev weights on the Fourier coefficients of the interpolant on the circle. By using a weight of the form m^s, we require that high frequency terms be all the more smaller, compared to the low-frequency terms. We are in effect looking for a low-pass filtered interpolant. However, we do not preserve *all* polynomials, since we are considering only those which can be expressed as a minimum sobolev norm solution. Suppose we wish to preserve certain polynomials, then the Sobolev weight in D_s corresponding to these terms is zero. Hence the matrix D_s has zero diagonal entries. With appropriate permutations it is possible to rewrite such a D_s as

$$D_s = \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \hat{D}_s \end{bmatrix} \tag{2.22}$$

The corresponding optimization problem now reads, find

$$\mathbf{a}^* = \arg \min_{V\mathbf{a}=\mathbf{f}} \|\hat{D}_s \mathbf{a_2}\|_2^2, \tag{2.23}$$

where $\mathbf{a_2}$ the $M_2 \times 1$ vector of coefficients whose Sobolev norm needs to be minimized. Let M_1 denote the number of polynomial terms preserved, corresponding to as many zero sobolev weights in D_s, $M_1 + M_2 = M$. Let us partition the

Chebyshev Vandermonde matrix V accordingly as

$$V = \begin{bmatrix} V_1 & V_2 \end{bmatrix}. \tag{2.24}$$

Using 2.24, we write 2.23 as, minimizing $\hat{D}_s \mathbf{a}_2$ subject to $V_1\mathbf{a}_1 + V_2\mathbf{a}_2 = \mathbf{f}$. We solve this problem in the manner below.

$$\begin{aligned} V_2\mathbf{a}_2 &= \mathbf{f} - V_1\mathbf{a}_1 && (2.25)\\ \Rightarrow V_2\hat{D}_s^{-1}\hat{D}_s\mathbf{a}_2 &= \mathbf{f} - V_1\mathbf{a}_1 && (2.26)\\ \Rightarrow (\hat{D}_s\mathbf{a}_2)^* &= (V_2\hat{D}_s^{-1})^\dagger(\mathbf{f} - V_1\mathbf{a}_1) && (2.27) \end{aligned}$$

We still need to pick $\mathbf{a}_1$ so that the minimum sobolev norm condition is met. Hence we solve,

$$\arg\min_{\mathbf{a}_1} \|(V_2\hat{D}_s^{-1})^\dagger(\mathbf{f} - V_1\mathbf{a}_1)\|_2^2. \tag{2.28}$$

The minimum norm solution for $\mathbf{a}_1$ is given by,

$$\mathbf{a}_1^* = ((V_2\hat{D}_s^{-1})^\dagger V_1)^\dagger (V_2\hat{D}_s^{-1})^\dagger \mathbf{f} \tag{2.29}$$

We now use equations 2.27 and 2.29 to solve for $\mathbf{a}_2$. The over all solution is given by $\mathbf{a}_1$ and $\mathbf{a}_2$.

2.2.3 Convergence with infinite order polynomials

We now consider the conditions on the smoothness of the underlying function f, so that the interpolant p_∞ converges to it. The reason why we consider ∞

order polynomials is that in this case, the coefficients of interpolation that we seek to find are actually the Fourier coefficients of $f(\cos\theta)$. In this case, since the Fourier coefficients themselves are a part of the possible solution space, and the interpolant $p_\infty(x)$ has minimum sobolev norm, we have,

$$\|p_\infty(x)\|_s^2 = \|D_s\mathbf{a}\|_2^2 < \|f\|_s^2. \tag{2.30}$$

Now if the function itself is in some sobolev space $\mathbf{H}^s$, then $\|f\|_s < \infty$ and so the sobolev norm of the interpolant is also bounded. We now consider the Arzela-Ascoli theorem of convergence. A set of functions $\mathcal{F}$ is said to equicontinuous if every one of them is equally continuous. This means that for this entire set $\mathcal{F}$ for all points x, y, if there exists an $\epsilon > 0$, such that $|f(x) - f(y)| < \epsilon, \quad \forall f \in \mathcal{F}$, then there exists δ so that $|x - y| < \delta$. Note that for a continuous function, the parameters ϵ, δ are specified for each f and for each x, y. The Arzela-Ascoli theorem states that for such an equicontinuous set $\mathcal{F}$ with a countable dense subset E, every sequence of functions $\{f_n\} \in \mathcal{F}$ uniformly converges on every compact subset over which F is defined [34].

Our plan of attack to prove convergence is as follows. We first consider the sequence of MSN interpolants with $M = \infty$, and increasing number of grid points $N_1, N_2, N_3, ...$ We show that this constitutes an equicontinuous, pointwise bounded set that satisfies the assumptions needed for Arzela-Ascoli type convergence for every sub-sequence. We then show that if we take the limit of

such a converging sub-sequence, then this limit converges point-wise to the underlying function $f(x)$. Since this is shown to be the case for every sub-sequence, the entire sequence of MSN interpolants should converge to $f(x)$.

Theorem 1. *Let $s > \frac{3}{2}$. Let $\mathbf{p}_N$ be the sequence of MSN interpolants of order ∞ at N grid points, $\{x_i : 1 \leq i \leq N, 1 \leq N < \infty\}$. If $f \in \mathbf{H}^r$, with $r \geq s$, and if x is a limit point of the set of grid points, then*

$$\lim_{N \to \infty} \mathbf{p}_N(x) = f(x).$$

Proof. Consider the norm of the first derivative of the interpolants on the circle, given by,

$$\begin{aligned}
\|p(\cos\theta)'\|_1 &= \sum_{m=1}^{\infty} |a_m| m && (2.31)\\
&= \sum_{m=1}^{\infty} |a_m| \frac{m^s}{m^{s-1}} && (2.32)\\
&\leq \sqrt{\sum_{m=1}^{\infty} |a_m|^2 m^{2s}} \sqrt{\sum_{m=1}^{\infty} \frac{1}{m^{2s-2}}}, \text{ using Cauchy-Schwarz} && (2.33)\\
&= \|D_s \mathbf{a}\|_2 \sqrt{\sum_{m=1}^{\infty} \frac{1}{m^{2s-2}}} && (2.34)\\
&\leq \|f\|_s \sqrt{\sum_{m=1}^{\infty} \frac{1}{m^{2s-2}}}, \text{ using 2.30} && (2.35)\\
&= K < \infty, \quad s > \frac{3}{2} && (2.36)
\end{aligned}$$

□

Since the first derivatives of the interpolants are uniformly bounded, using mean value theorem, we would have that

$$\frac{|p_\infty(x) - p_\infty(y)|}{|x - y|} \leq K. \tag{2.37}$$

Thus, to satisfy equicontinuity, one can pick any ϵ and $\delta = \epsilon/2K$. This is a well known results about differentiable functions. The fact that we are working with functions with uniformly bounded derivatives is *very* important in that, Arzela-Ascoli becomes transparent to us! In fact, using the power of Arzela-Ascoli, the proof even generalizes to d dimensional spaces easily.

Now that we have an equicontinuous set of interpolants $\mathbf{p}_N$, over a compact interval on the real line, every sub-sequence of interpolants (corresponding to increasing number of grid points), converges. Let us consider one such sub-sequence, and let us suppose its limit is $p(x)$. We now show that irrespective of the sub-sequence we consider, this limit converges to the underlying function. Consider some point x, which is the limit point to some sequence of grid points, $\{x_{n,N_k}\}, 0 < n \leq N_k$.

$$|p(x) - f(x)| = \lim_{n\to\infty} |p(x_{n,N_k}) - f(x_{n,N_k})| \tag{2.38}$$

Since we interpolate at the grid points, we can replace f with the sub-sequence of interpolants, $\mathbf{p}_N$.

$$|p(x) - f(x)| = \lim_{n\to\infty} |p(x_{n,N_k}) - \mathbf{p}_N(x_{n,N_k})| \tag{2.39}$$

The above is a point-wise limit, and so we can replace the RHS with max norm, as

$$|p(x) - f(x)| \leq \lim_{n\to\infty} \|p - \mathbf{p}_N\|_\infty \tag{2.40}$$

But the limit of the sub-sequence $\mathbf{p}_N$ is p. Hence, we have shown that p converges to f at any point x, so long as we pick a set of grid points, whose limit will be x. Since we picked a subsequence in general, and the limit namely f was independent of which subsequence we pick, all subsequences, and hence the entire sequence of interpolants converges point-wise to f, at limit points to the grid points.

$$|p(x) - f(x)| = 0 \text{ using the fact that } p \text{ is the limit of } \mathbf{p}_N \tag{2.41}$$

Corollary 1. *Theorem 1 still holds if* $s > \frac{1}{2}$.

The assumptions of the Arsela-Ascoli theorem are still satisfied if we have uniform Holder continuity, rather than bounded derivatives. In fact, this directly presents a mean-value form that shows equicontinuity. The reason we are going through this is of course because, $s > \frac{1}{2}$ does not even imply differentiability, but important functions such as $|x|$ are in this space. We show that a function in $\mathbf{H}^s$ has a bounded Hölder norm, and hence the family of MSN interpolants with $s > \frac{1}{2}$ is still equicontinuous. The α-Hölder norm we consider is $\left\| \frac{f(x)-f(y)}{|x-y|^\alpha} \right\|_\infty$.

Proof. Consider a function g in $\mathbf{H}^{s>\frac{1}{2}}$. We consider the function at two points θ_1, θ_2. Then, we would like to bound the above mentioned Hölder norm for g for some α. Suppose that g has Fourier coefficients $\hat{g}_m$.

$$\begin{aligned} |g(\theta_1) - g(\theta_2)| &= \left| \sum_{m=0}^{\infty} g_n \left(\cos m\theta_1 - \cos m\theta_2 \right) \right| && (2.42) \\ &\leq \sum_{m=1}^{\infty} |g_m| \left| \cos m\theta_1 - \cos m\theta_2 \right| && (2.43) \\ &= 2 \sum_{m=1}^{\infty} |g_m| \left| \sin\left(m\frac{\theta_1 - \theta_2}{2} \right) \right| \left| \sin\left(m\frac{\theta_1 + \theta_2}{2} \right) \right| && (2.44) \end{aligned}$$

We now use the observation that $|\sin(mx)| \leq |mx|^t, 0 < t \leq 1$. Using this for the term $\left|\sin\left(m\frac{\theta_1-\theta_2}{2}\right)\right|$ and continuing on the earlier argument, we have

$$\begin{aligned} |g(\theta_1) - g(\theta_2)| &\leq 2 \sum_{m=1}^{\infty} |g_m| \left| \left(m\frac{\theta_1 - \theta_2}{2} \right) \right|^t, \ 0 < t \leq 1 && (2.45) \\ &= 2^{1-t} |\theta_1 - \theta_2|^t \sum_{m=1}^{\infty} |g_m| m^t, \ 0 < t \leq 1 && (2.46) \end{aligned}$$

We now use the usual trick of multiplying and dividing by m^β.

$$\begin{aligned} |g(\theta_1) - g(\theta_2)| &\leq 2^{1-t} |\theta_1 - \theta_2|^t \sum_{m=1}^{\infty} \frac{|g_m|}{m^\beta} m^{t+\beta}, \ 0 < t \leq 1 && (2.47) \\ \frac{|g(\theta_1) - g(\theta_2)|}{|\theta_1 - \theta_2|^t} &\leq 2^{1-t} \sum_{m=1}^{\infty} \frac{|g_m|}{m^\beta} m^{t+\beta}, \ 0 < t \leq 1 && (2.48) \\ &\leq 2^{1-t} \sqrt{\sum_{m=1}^{\infty} \frac{1}{m^{2\beta}}} \sqrt{\sum_{m=1}^{\infty} |g_m|^2 m^{2t+2\beta}} && (2.49) \end{aligned}$$

The first term in the above product is bounded for $\beta > \frac{1}{2}$. Since $g \in \mathbf{H}^{s>\frac{1}{2}}$, the second term would also be bounded if $t + \beta \geq \frac{1}{2}$. This guaranteed because $\beta > \frac{1}{2}$

by choice, and $0 < t \leq 1$. We see that the function g is Hölder continuous for some t. Therefore the MSN interpolant class with $s > \frac{1}{2}$ is equicontinuous, and we can proceed using Arzela-Ascoli as before with Theorem 1. □

The above results may be easily extended to the cases where we have intervals devoid of limit points and uniform convergence over any dense subset of the interpolating interval for $s > 2$ using the Arzela-Ascoli convergence.

2.2.4 Convergence with finite order polynomials

The case for finite order polynomials is much more involved. We merely state the key results here; the proof is presented in [9]. The key idea is still to prove that the MSN interpolants of finite order still satisfy the assumptions needed by Arzela-Ascoli. For this, there exist an additional constraint on the order of the polynomial. Essentially, we want the order of the polynomial of the polynomial to be a constant factor of the *mesh norm* $I(x_N)$ associated with a set of points x_N, which we define as

$$I(x_N) = \left\lceil \frac{\pi}{\min_{i \neq j} \|\theta_i - \theta_j\|} \right\rceil, \tag{2.50}$$

where $\theta_i = \cos^{-1} x_i$. We therefore require that $M = cI(x_N)$ in order that

$$\|p_M\|_s \leq K\|f\|_s \leq \infty. \tag{2.51}$$

Once we have bounded Sobolev norms, one can use the techniques presented to show an induced bound on a Hölder norm and hence equicontinuity. With these in place, we can appeal to Arzela-Ascoli again to prove convergence. The paper by Chandrasekaran and Mhaskar [9] proves such convergence for arbitrary p Sobolev norms, and arbitrary dimensions. The Arzela-Ascoli theorem is really general and extremely powerful indeed!

2.2.5 Construction of the MSN Interpolant

For the one-dimensional case, we saw how the MSN interpolant is constructed earlier. The only comment that needs to be made is that, one need to choose the order of the interpolating polynomial based on the mesh norm as required by our theorem. As a rule of thumb, for the equispaced grid setting, the order of the interpolant needs to twice the number of grid points in each dimension. For the general setting, the order of the polynomial is set to be three to four times the mesh norm for each dimension.

Consider a set of grid points, $\mathbf{x}_i = \{x_i, y_i\}_{i=0}^{N}$. Let the corresponding mesh norm be

$$I(\mathbf{x}_i) = \left\lceil \frac{\pi}{\min_{i \neq j} \|\theta_i - \theta_j\|_2} \right\rceil, \tag{2.52}$$

where $\theta_i = \{\cos^{-1} x_i, \cos^{-1} y_i\}$. We set $M = \frac{4}{\lceil \min_{i \neq j} \|\theta_i - \theta_j\|_2 \rceil}$ in practice.

In two dimensions, the Chebyshev basis takes polynomials of the form

$$\mathbf{T}_{m,n}(\mathbf{x}) = T_m(x)T_n(y),\ 0 \leq m < M_x,\ 0 \leq n < M_y, \tag{2.53}$$

product of one dimensional polynomials.

We now write the Chebyshev Vandermonde matrix as

$$V = \begin{bmatrix} \mathbf{T}_{0,0}(\mathbf{x}_0) & \mathbf{T}_{0,1}(\mathbf{x}_0) & ... & \mathbf{T}_{0,M_y-1}(\mathbf{x}_0) & ... & \mathbf{T}_{M_x-1,M_y-1}(\mathbf{x}_0) \\ \mathbf{T}_{0,0}(\mathbf{x}_1) & \mathbf{T}_{0,1}(\mathbf{x}_1) & ... & \mathbf{T}_{0,M_y-1}(\mathbf{x}_1) & ... & \mathbf{T}_{M_x-1,M_y-1}(\mathbf{x}_1) \\ . & . & . & . & . & . \\ \mathbf{T}_{0,0}(\mathbf{x}_{N-1}) & \mathbf{T}_{0,1}(\mathbf{x}_{N-1}) & ... & \mathbf{T}_{0,M_y-1}(\mathbf{x}_{N-1}) & ... & \mathbf{T}_{M_x-1,M_y-1}(\mathbf{x}_{N-1}) \end{bmatrix} \tag{2.54}$$

The Sobolev weight matrix D_s is defined to correspond to a Sobolev norm as below,

$$D_s = \begin{bmatrix} (1+0^2+0^2)^{\frac{s}{2}} & 0 & 0 & ... & 0 \\ 0 & (1+0^2+1^2)^{\frac{s}{2}} & 0 & ... & 0 \\ . & . & . & . & . \\ 0 & 0 & 0 & ... & (1+(M_x-1)^2+(M_y-1)^2)^{\frac{s}{2}} \end{bmatrix} \tag{2.55}$$

With the V and D_s defined, one can construct the MSN interpolant as described. The interpolatory process remains the same irrespective of dimensionality. In fact, we readily even see how things extend to any higher dimension! The rate of convergence of the interpolant with increasing grid density is an important

parameter. We discuss this in the light of local discretizations we use to produce FD weights in a later chapter. We do pay a penalty in the rate with increase in dimensions, as the case called the 'curse of dimensionality'. Also, if the grid points are not in $[-1, 1]$, then we must use an appropriate mapping function $\phi(\mathbf{x})$ before we use the Chebyshev basis at these points.

The construction of the interpolant involves the construction of the pseudo-inverse of VD_s^{-1}. This is often carried by first performing a QR or QL factorization of $D_s^{-1}V^T$. However, the QR factorization of such badly row-scaled matrices is inaccurate, unless carried out with complete pivoting, or in rank-revealing manner. Such factorizations require a special treatment, through the use of complete orthogonal decompositions [22]. We present one such efficient method using SVDs, which we call CODA. This is described in a subsequent chapter in conjunction with the computation of FD weights. For now, we shall assume that any QR factorization involving the bad row-scaling has been carried out though CODA.

2.2.6 Numerical Results for Interpolation - 1D

In this section, we present a series of numerical examples of 1D MSN interpolation.

Table 2.1: MSN Interpolation error for Runge function

		s						
N	M	2	4	6	8	10	12	14
30	60	1.8e-3	4.2e-4	1.1e-3	1.8e-2	1.4e-1	7.3e-1	2.7e+00
100	200	4.9e-6	1.7e-9	9.9e-12	7.3e-13	9.1e-12	1.1e-9	5.1e-8
311	622	3.3e-7	7.5e-12	1.8e-14	1.4e-13	1.6e-12	1.8e-10	8.6e-8
512	1024	9.6e-8	7.7e-13	1.7e-14	2.7e-13	9.1e-12	2.6e-8	1.2e-5

Runge type function $f(x) = \frac{1}{1+x^2},\ x \in [-5, 5]$

Table 2.1 presents the maximum relative reconstruction error using MSN interpolants to the function $f(x) = \frac{1}{1+x^2}$ over the interval $[-5, 5]$. Note that this was the same example Runge had used to observe the oscillations. The table contains the number of grid points N, the order of the interpolant M (chosen to be twice the number of grid points), and the maximum relative reconstruction error for $s = 2, 4, 6, 8, 10, 12, 14$. The error is measured as $\frac{\|f(x)-p_M(x)\|_\infty}{\|f(x)\|_\infty}$ where x is a set of 1024 equispaced points in the interval $[-5, 5]$.

Runge type function $f(x) = \frac{1}{1+25x^2}$

Table 2.2: MSN Interpolation error for $f(x) = \frac{1}{1+25x^2}$

		s						
N	M	2	4	6	8	10	12	14
30	60	1.8e-3	4.2e-4	1.1e-3	1.8e-2	1.4e-1	7.3e-1	2.6e+00
100	200	4.9e-6	1.7e-9	9.9e-12	7.3e-13	8.9e-12	1.0e-9	5.1e-8
311	622	3.3e-7	7.5e-12	3.0e-14	2.2e-13	4.3e-12	5.3e-11	7.1e-8
512	1024	9.7e-8	7.8e-13	5.6e-15	9.2e-14	4.4e-12	7.4e-8	4.0e-5

Table 2.2 presents the maximum relative reconstruction error using MSN interpolants to the function $f(x) = \frac{1}{1+25x^2}$ over the interval $[-1, 1]$. The error is measured over a set of 1024 equispaced points in the interval $[-1, 1]$. Note that this function has its complex poles closer from the real plane than the previous function. In order to be visually convincing as well, see below, the MSN Interpolant and the actual function super-imposed. They are barely discernible from each other in figure 2.5. The figure uses 30 equispaced samples, with $s = 2$ and a polynomial of order 60.

Figure 2.5: MSN Interpolant to $f(x) = \frac{1}{1+25x^2}$ at 30 equispaced points.

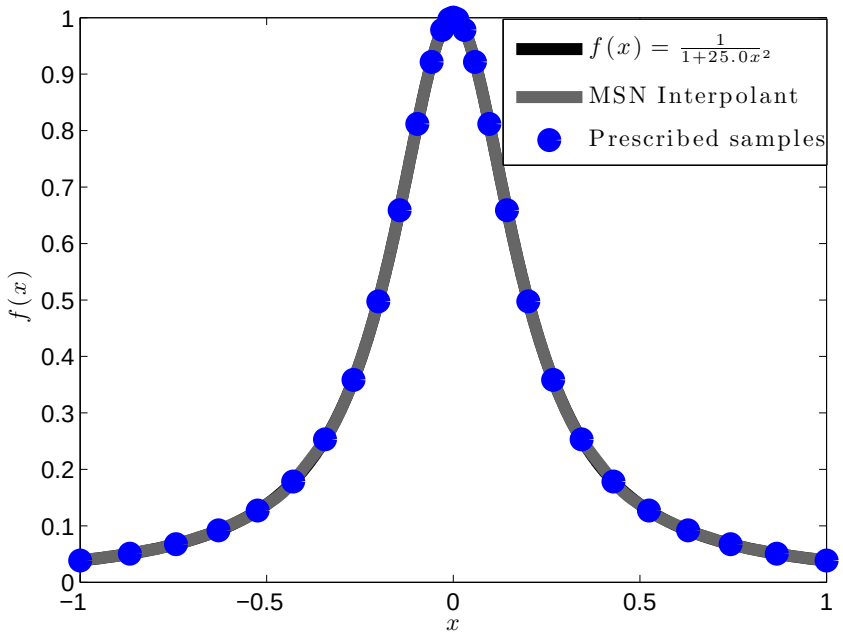


Table 2.3: MSN Interpolation error for $f(x) = \frac{1}{1+0.01x^2}$

		s						
N	M	2	4	6	8	10	12	14
30	60	2.0e-5	3.7e-9	2.0e-12	9.2e-15	7.4e-14	1.9e-13	8.8e-13
100	200	1.8e-6	3.4e-11	2.2e-14	1.4e-13	2.9e-12	7.5e-11	1.0e-9
311	622	2.0e-7	3.9e-13	9.9e-14	9.5e-13	5.7e-12	5.3e-10	1.6e-7
512	1024	5.9e-8	4.5e-14	4.1e-14	1.9e-13	1.7e-11	4.9e-8	2.4e-4

Runge type function $f(x) = \frac{1}{1+0.01x^2}$

Table 2.3 presents the maximum relative reconstruction error using MSN interpolants to the function $f(x) = \frac{1}{1+0.01x^2}$ over the interval $[-1, 1]$. The error is measured over a set of 1024 equispaced points in the interval $[-1, 1]$. Note that this function has its complex poles farther from the real plane than the previous function.

Function $f(x) = \sqrt{|x|}$

Table 2.4: MSN Interpolation error for $f(x) = \sqrt{|x|}$

		s						
N	M	0.5	1.0	1.5	2	6	10	14
30	60	2.1e-1	1.5e-1	1.3e-1	1.3e-1	1.3e-1	1.2e+00	1.9e+01
100	200	1.2e-1	6.4e-2	5.8e-2	5.7e-2	5.5e-2	5.4e-2	1.1e+02
311	622	8.2e-2	2.7e-2	2.7e-2	2.9e-2	3.1e-2	6.8e-1	7.3e+05
512	1024	6.2e-2	1.1e-2	9.3e-3	8.8e-3	8.2e-3	3.6e+00	7.2e+06

The function $\sqrt{|x|}$ is not differentiable at $x = 0$. Hence, the interpolation error is dominated by the accuracy in the vicinity of $x = 0$. To address, one may resort to a more careful choice of grid points, such as a quadratically clustered

set of points $\text{sgn}(x)x^2$, where x is still from an equispaced set of points in $[-1, 1]$. We see that the effect is immediate, and we get more accuracy at the even a 100 grid points. Of course, to do this, we have used a much larger polynomial order, since the mesh norm, which depends on the minimum sample spacing is much smaller. Due to numerical ill-conditioning, we restrict ourselves to $s \leq 4$ for this case; note that $s > \frac{1}{2}$ is sufficient for this case. We shall discuss the numerical error in more detail while summarizing the numerical results. Table 2.5 below presents the interpolation error at 1024 equispaced points in $[-1, 1]$.

Table 2.5: Square root Interpolation, quadratic grid point clustering

		s				
N	M	0.5	1.0	1.5	2	6
30	1261	3.7e-1	5.1e-2	1.4e-2	1.6e-2	4.7e+02
100	14701	2.7e-1	1.7e-2	1.4e-3	1.0e-3	8.8e+03
311	72075	1.8e-1	5.6e-3	2.7e-4	1.46e-4	8.8e+12
512	391682	1.5e-1	3.5e-3	1.3e-4	4.7e-6	4.1e+12

Lagrange and Spline interpolation on $\sqrt{|x|}$

For the sake of visual curiosity, we see how Lagrange interpolation does on this function with equispaced and quadratic points. Figure 2.6 and Figure 2.7 show Lagrange interpolation of the $\sqrt{|x|}$ function at $5, 9$ equispaced and quadratically clustered points respectively. We see that Runge oscillations render Lagrange interpolation with no hope. In addition to Lagrange interpolation, we also take a moment to compare our performance against Splines. For this, we use the cubic

spline interpolation toolbox of Matlab. We use piece-wise cubic interpolants with slope conditions matching at the edges in this case. We observe that due to their local nature, they are affected much by errors in the vicinity of $x = 0$. However, they converge extremely slowly. To get to a comparable accuracy of 10^{-4}, it was observed experimentally that $1e6$ equispaced grid points were needed. If we used quadratic clustering, the error actually worsened, and only and accuracy of 10^{-2} was observed. To fix the slow convergence one may hope to set up larger polynomials, so that higher derivatives could be matched. But this presents dual problems. Firstly, Runge oscillations will haunt us again. Secondly, the idea of matching point-wise derivatives become complicated in higher dimensions and geometries such as even a sphere.

Summary

Firstly, the Runge phenomenon has been dealt with successfully. We observe convergence for several Runge type functions. As a hard example, we considered the square root function, and convergence was observed for $s > 0.5$. In fact, we observed convergence for $s = 0.5$. Clearly, the theorem we proved can be made much sharper! We considered quadratically clustered grid points, and saw that it increases accuracy considerably. It was observed that for such a grid, one needs to resort to much higher orders of interpolants. The numerical conditioning of such

Figure 2.6: Square root function and its Lagrange interpolants

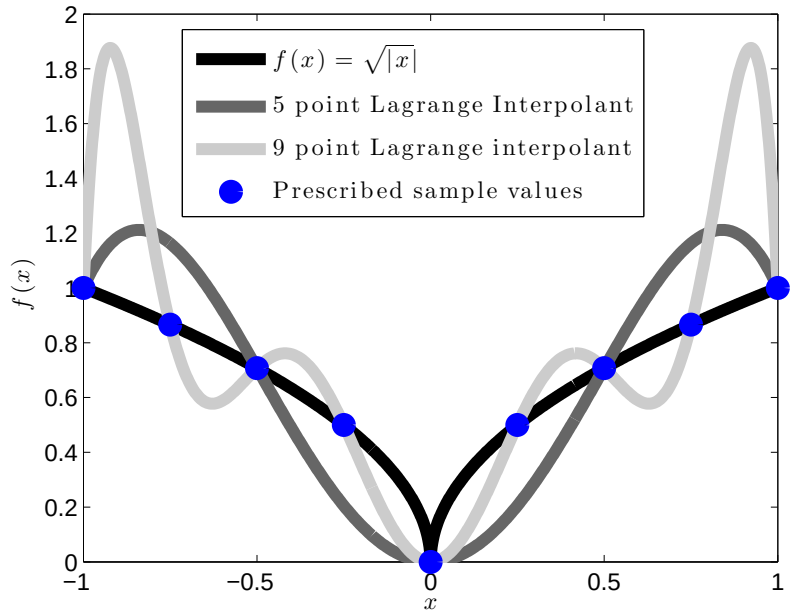
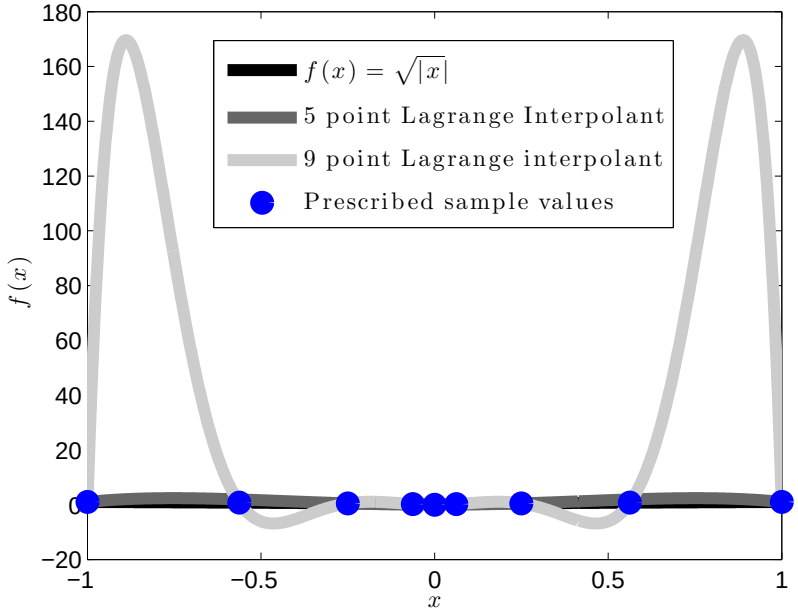


Figure 2.7: Square root function on a quadratic grid



higher order interpolation methods are still a problem. While we dealt with them partially using the CODA algorithm, to be detailed in the next chapter, we see that it still is a problem for larger M and s. However, a few comments must be made in this regard. Firstly, for rough functions such as the square root function, a small value of s will work well. Secondly, since we have high rates of convergence, the desired accuracy can be reached with relatively coarse grids, which means a more modest order for the interpolant. Thirdly, for our end goal of solving PDEs, we are going to deploy MSN locally. Of course, the size of the neighborhood determines the order of the FD method, but increasing order increases the computationally complexity as well. As shown with a simple calculation, it is not advantageous to increase the size of interpolating neighborhoods beyond a limit. Within this limit, the MSN interpolant is very well conditioned, as we shall see, even up to $s = 50$. Note that since our primarily goal was to present a convergent scheme, we did not make explicit comments about the rate of convergence. As evident in comparison with splines, MSN is a higher order interpolation method, capable of many order accuracy with comparable grid sizes.

2.2.7 Numerical results for interpolation - 2D

We now consider examples of MSN interpolation of two dimensional samples.

Smooth Runge function $\frac{1}{1+25x^2+25y^2}$

Figure 2.8 shows the MSN interpolant to the considered function at a 20×20 regular grid in $[-1,1] \times [-1,1]$. Note that in comparison with the Lagrange interpolant shown in figure 1.3 there is no Runge oscillations. The order of the polynomial was chosen to be $M = 57$ along each dimension and s was set to 2.

Figure 2.8: MSN interpolant to Runge type function in two dimensions

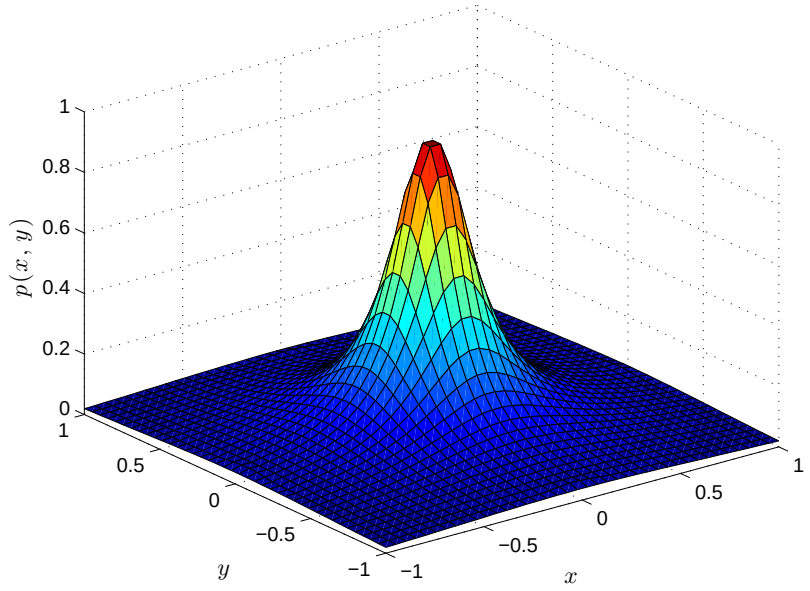


Table 2.6 shows the MSN interpolation error. N indicates the number of equispaced grid points in the interval $[-1,1] \times [-1,1]$. s was varied as $2, 4, 6, ...14$. The optimum choice of s is seen to be around s taking into account numerical issues. This function has comparable smoothness to the one-dimensional Runge example we presented in table 2.2. For this function, the interpolation example

was run until memory overflow in the computer and up to 24 hours of computation time. This was a time limit on the use of super computing nodes, with large memory. Note that we are able to interpolate to an accuracy of 7 digits. The reconstruction error for each N was measured over an equispaced grid using $4 \times N$ grid points in the interval of interpolation. The reconstruction grids were chosen not to overlap with the interpolation grids.

Table 2.6: MSN Interpolation error for $f(x) = \frac{1}{1+25x^2+25y^2}$

		s						
N	M	2	4	6	8	10	12	14
100	900	2.5e-1	2.2e-1	2.2e-1	2.2e-1	2.2e-1	2.3e-1	2.3e-1
961	8449	2.1e-3	3.5e-4	1.6e-4	2.2e-3	2.5e-2	1.6e-1	7.3e-1
6400	57600	2.2e-4	2.2e-5	4.7e-6	1.4e-6	4.6e-7	2.4e-7	1.5e-6
14641	131769	7.5e-5	5.5e-6	8.7e-7	1.9e-7	5.3e-08	1.7e-08	6.0e-09

Rough Runge function

Having convinced ourselves of the basic example, in which Runge oscillations were successfully suppressed, we now consider a rougher example. This function has several Runge type singularities, along a circle, two paraboli and a straight line too. The function is shown in Figure 2.9 and is mathematically given by,

$$\begin{aligned} f(x,y) &= \frac{1}{1+25(x^2+y-.3)^4} + \frac{1}{1+35(x+y-5.4)^2} + \\ & \quad \frac{1}{1+25(x+y^2-.5)^2} + \frac{1}{1+35(x^2+y^2-2.25)^2} \end{aligned}$$

Figure 2.9: MSN interpolant to Runge type function in two dimensions

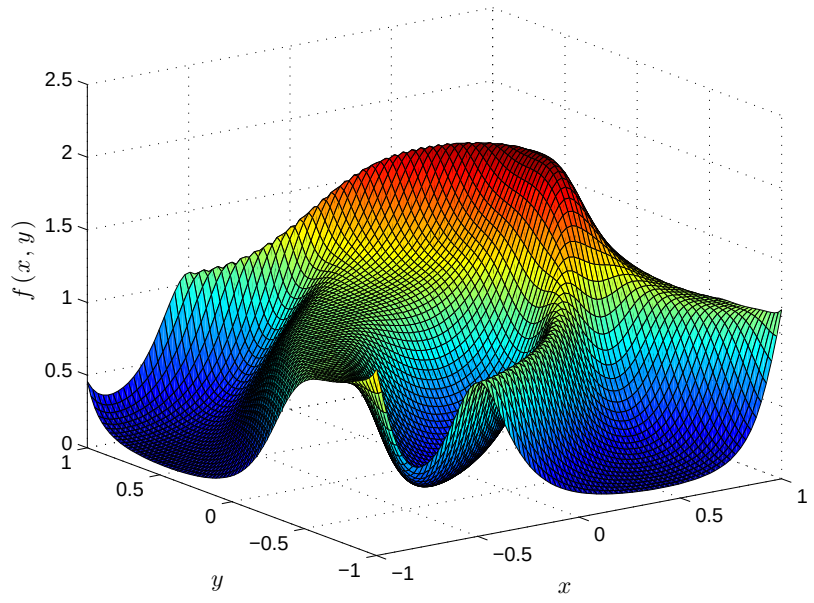


Table 2.7 below presents the Interpolation error for this function. Once again, we succeed in obtaining up to 6 digits of accuracy for the grid sizes we considered and possible could be run.

Table 2.7: MSN Interpolation error for the rough Runge type function

		s						
N	M	2	4	6	8	10	12	14
100	900	2.3e-1	2.1e-1	2.6e-1	7.8e-1	1.4	2.0	2.4
961	8449	2.1e-2	1.1e-2	1.0e-2	1.7e-2	8.0e-2	3.5e-1	1.4
6400	57600	1.6e-3	1.8e-4	3.2e-5	8.5e-6	8.0e-6	5.5e-5	4.8e-4
14641	131769	5.2e-4	3.5e-5	4.9e-6	1.0e-6	2.7e-07	1.0e-07	5.8e-08

Annular Domain

We now consider interpolation of a rough function given by

$$f(r,\theta) = \mathcal{I}m\{\sqrt{(r-.5)(0.5-r)}\sin 2|\theta|\} \tag{2.56}$$

We also complicate the domain of interpolation to an annulus with a concentric disk as shown as in figure 2.10. The front views of the function being interpolated are given in figures 2.11,2.12.

Figure 2.10: Domain of interpolation $r \leq 0.3 \cup 0.5 \leq r \leq 0.75$

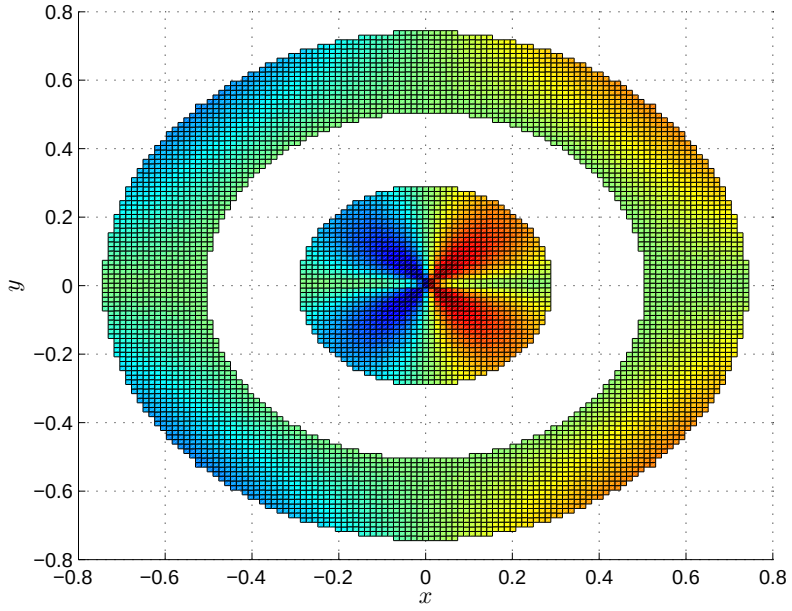


Table 2.8 gives the interpolation error over the domain specified as an annular region $0.1 \leq r \leq 0.25$. Note the orders of the interpolant. These were picked computing the mesh norm of the region being interpolated. The function itself has

Figure 2.11: Front view of the rough function

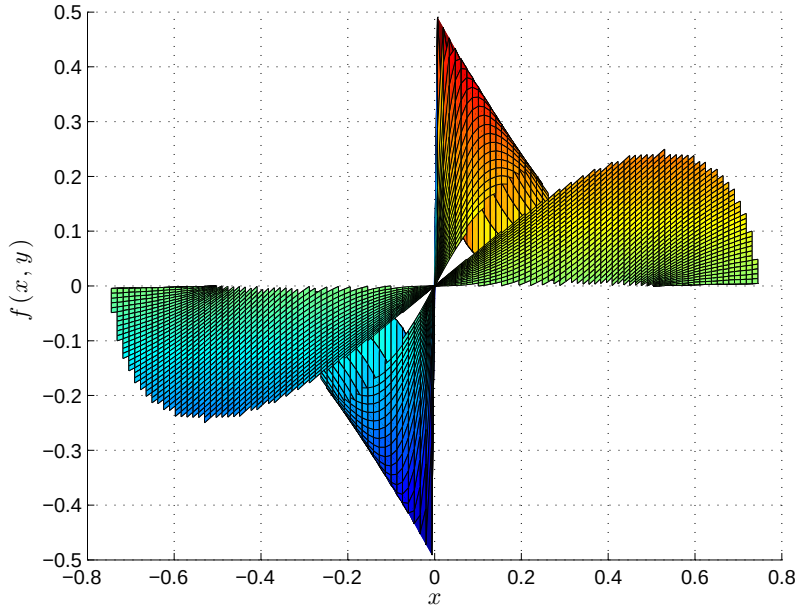
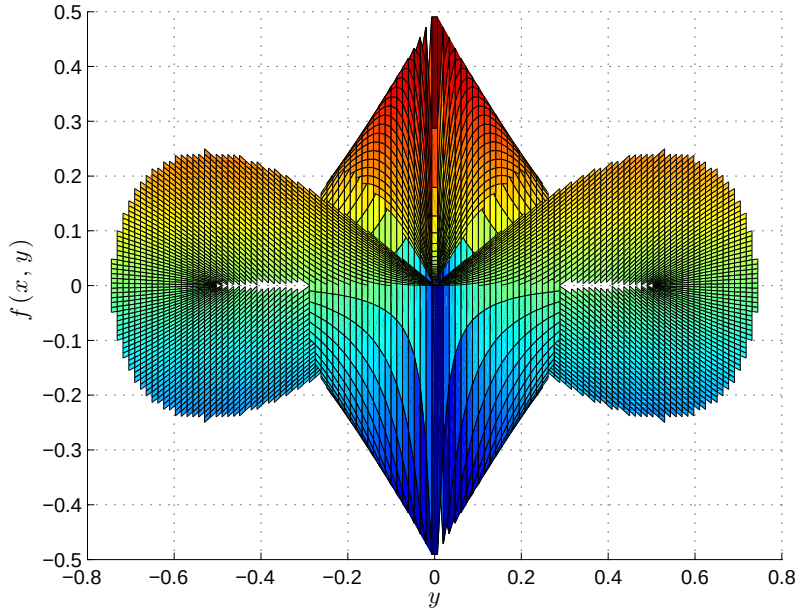


Figure 2.12: Front view of the rough function



square root type singularities over a circle. In addition there is a jump singularity at $\theta = 0, \pi$.

Table 2.8: Rough function over a complex domain

		s						
N	M	2	4	6	8	10	12	14
400	921	9.7e-1	1.1	1.2	1.2	1.2	1.3	1.3
1681	35344	1.3e-1	1.5e-1	1.6e-1	2.6e-1	4.2e-1	6.0e-1	7.5e-1
3969	85264	8.8e-2	9.8e-2	9.9e-2	1.0e-1	1.0e-1	1.9e-1	4.0e-1

Summary

The most important observation that is relevant in the global context of MSN applications is that even in two dimensions, we retain the rapid convergence properties even for complicated functions, provided sufficient samples are available. In the context of a local discretization of PDE Solutions, this is the most important property we are interested in. PDE Solutions are often locally smooth, and the solution may be rough over particular regions. The numerical issues are more predominant in the two dimensional case compared to the one dimensional case. But note that in the context of a local discretization, the grid size we considered are *extremely* large. The fact the interpolation worked stably even over such size, is very encouraging. It also means that for smaller discretizations, we can accelerate the convergence using larger values of s as the case may be.

We also demonstrated MSN's power in interpolating over complicated domains. This again comes in handy when local discretizations are over complicated parts of the domain of a PDE such as corners, exclusions etc. The observed convergence for specific singular cases over complicated domains motivates that the fact the method may succeed over singular PDEs over complicated geometries. Yet another application where in the power of MSN has been demonstrated is image segmentation. It has been shown in [8] that MSN can be used to characterize regions of the image (which may be be as complicated as the objects that need to be segmented). This requires us to perform interpolation or approximation operations over such complicated domains as illustrated in the last example.

Another possibility with using MSN to solve PDEs is to use MSN coefficients as unknowns of the solution. This would be a global pseudo-spectral type of approach. The resulting PDE discretizations would be dense, albeit with structure. It has been experimentally observed that the interpolation operator kernel has low-rank off-diagonal structure as required by HSS [5] type algorithms over cartesian grids. For general grids, one may need to use the full FMM formulation[20]. We do not pursue this approach in this thesis. Fast algorithms and kernels for the approximation case of MSN are discussed in an appendix.

Chapter 3

Complete Orthogonal Decomposition Algorithm

Give me a place to stand, and I will move the earth. Archimedes

3.1 The Weighted Least-Squares problem

We begin this chapter with a brief introduction to weighted least squares (WLS) problems. A WLS problem is usually as a minimization problem,

$$\arg\min_x \|W(Ax - b)\|_2^2, \tag{3.1}$$

where $A_{M\times N,\, M>N}$ is a tall-kinny matrix. A is assumed to have full column rank, N. W is the weight matrix and b specifies the right hand side. The case when

W is diagonal is most common. In this case, let us denote $w_1, w_2, ...w_M$ to be the diagonals of W. If we let the rows of A be

$$A = \begin{bmatrix} \text{———} & A_1^T & \text{———} \\ \text{———} & A_2^T & \text{———} \\ \text{———} & A_3^T & \text{———} \\ . & . & . \\ \text{———} & A_M^T & \text{———} \end{bmatrix}, \tag{3.2}$$

then the WLS equations are

$$\arg\min_x \sum_{i=1}^{M} |w_i(A_i^T x - b)|^2. \tag{3.3}$$

Practical situations in which such problems arise include barrier methods in optimization, finite elements, structural investigations, circuit analysis problems and in our case FD weight calculation through MSN. Further, it is natural for the weights w_i to be highly skewed, leading to a highly ill-conditioned W. [38] discusses the problems with solving WLS systems, with highly ill-conditioned positive definite diagonal W. In addition to the examples Vavasis discusses, as we shall note shortly, MSN based FD weight computation also involves ill-conditioned WLS systems.

In the notation of Vavasis, the stability of an algorithm to solve such a WLS system is defined by an error bound independent of W. Essentially, we are looking for an algorithm whose backward error can be bounded independent of the

scaling matrix W. Traditional methods for solving WLS systems, such as LU factorizations, Cholesky factorizations and even QR decomposition based methods are shown to be unstable in the above sense. In [22] Hough and Vavasis discuss the method of complete orthogonal decomposition to solve such WLS systems. They show that the proposed approach is stable. In this chapter, we draw upon their idea and present a poor man's implementation of the technique, which we called CODA. Instead of relying on the more expensive completely pivoted decomposition methods, we use library SVD routines in a rank revealing manner to obtain accurate results.

3.2 The Trick to solving WLS systems

While solving an ill-weighted LS system, the problem is essentially that due to bad row scaling, traditional algorithms end up adding two very differently scaled rows. This causes severe round off errors. If somehow, this bad row-scaling could be dealt with, say by converting it into bad column scaling instead, then one can hope that traditional methods would do better. Consider the ill row-scaled matrix A and its SVD as below,

$$WA = U\Sigma V^H \tag{3.4}$$

$$\Rightarrow DAV = U\Sigma. \tag{3.5}$$

The matrix $U\Sigma$ *if* computed correctly, would be the badly column scaled version we want. If we have a QR factorization of this *nicer* matrix, then we can write,

$$U\Sigma = QR \tag{3.6}$$

$$\Rightarrow WA = QRV^H \tag{3.7}$$

Note that we are dependent on an accurate SVD being computed. Of course, a regular SVD would fail on the badly row scaled system. But one can hope that it computes the singular values accurately up to some scale. We use idea and repeatedly use SVDs to peel more accurate singular values, together with the orthogonal columns of V. Note that we are never interested directly in the column factors U. So for the tall skinny system, a memory efficient implementation is possible, if we use only the singular values in Σ and the row orthogonal vectors in V^H. We state the algorithm with subsequent discussion leading to an informal proof below. We present substantial numerical evidence that the algorithm works, particularly in our context of MSN.

3.2.1 CODA

Consider the badly row-scaled matrix, $WA = B$. Our objective is to obtain a form $BV^H = U\Sigma$, U is well conditioned and Σ accordingly would be

ill-conditioned. We shall show that it is possible to use SVDs to achieve this. We let the refinement factor be η, the fraction of the largest singular value that could be computed accurately. A smaller value of η means more refinement, and also that we could potentially handle more severe ill-conditioning. Typically, η is set to be about 10.

Step 1 Compute top-level SVD

$$B = U\Sigma V^H \tag{3.8}$$

$$\Rightarrow BV = U\Sigma. \tag{3.9}$$

We can stop here if the SVD we to be computed accurately. But this may not be so. Hence we iterate.

Step 2 Let the singular values in Σ be $\sigma_1, \sigma_2, ...$ in decreasing order. Let $k = \arg\min_i \sigma_i > \frac{\sigma_1}{\eta}$. Split

$$BV = \begin{bmatrix} BV(*, 1:k-1) & BV(*, k:end) \end{bmatrix} \tag{3.10}$$

$$= \begin{bmatrix} \underbrace{BV_1}_{accurate} & BV_2 \end{bmatrix} \tag{3.11}$$

$$= \begin{bmatrix} U\Sigma_1 & U\Sigma_2 \end{bmatrix}. \tag{3.12}$$

We also appropriately partition $V = \begin{bmatrix} \underbrace{V_1}_{accurate} & V_2 \end{bmatrix}$

Step 3 Since BV_2 and V_2 we not accurate, we refine them further. Note that we have *peeled* off the range of singular values corresponding to a factor of η from the matrix we wanted to factorize. We perform the the refinement SVD,

$$BV_2 = \tilde{U}\tilde{\Sigma}\tilde{V}^H \tag{3.13}$$

$$\Rightarrow BV_2\tilde{V} = \tilde{U}\tilde{\Sigma}. \tag{3.14}$$

In order for algorithm to terminate there should be no singular value $\tilde{\sigma}_i < \frac{\tilde{\sigma}_{max}}{\eta}$.

Case 1: Suppose the termination condition is met, we show how to obtain the desired factorization form. Consider

$$B\begin{bmatrix} V_1 & V_2\tilde{V} \end{bmatrix} = \begin{bmatrix} BV_1 & BV_2\tilde{V} \end{bmatrix} \tag{3.15}$$

$$= \begin{bmatrix} U\Sigma_1 & \tilde{U}\tilde{\Sigma} \end{bmatrix} \tag{3.16}$$

$$= U\begin{bmatrix} \Sigma_1 & U^T\tilde{U}\tilde{\Sigma} \end{bmatrix}. \tag{3.17}$$

Now the columns of U form an orthogonal for the column space of A. On the other hand, $\tilde{U}$ forms a basis for the column space of AV_2. But,

$$AV_2 = U\Sigma_2 = \begin{bmatrix} U_1 & U_2 \end{bmatrix}\begin{bmatrix} 0 \\ \hat{\Sigma}_2 \end{bmatrix} = U_2\hat{\Sigma}_2. \tag{3.18}$$

Hence, the columns of $\tilde{U}$ span the columns of U_2. Also note that the columns of U are orthonormal be definition. Therefore,

$$U^T\tilde{U} = \begin{bmatrix} U_1^T \\ \\ U_2^T \end{bmatrix} \tilde{U} = \begin{bmatrix} 0 \\ \\ I \end{bmatrix}. \tag{3.19}$$

We therefore get the desired form,

$$B \begin{bmatrix} V_1 & V_2\tilde{V} \end{bmatrix} = B\overline{V} = U \begin{bmatrix} \Sigma_1 & \begin{pmatrix} 0 \\ \\ \tilde{\Sigma} \end{pmatrix} \end{bmatrix} = U\overline{\Sigma}. \tag{3.20}$$

Case 2: Suppose there are some singular values that are smaller than of $\frac{\tilde{\sigma}_{max}}{\eta}$, then we repeat the refinement as before, and split

$$BV_2 = \begin{bmatrix} BV_2\tilde{V}_1 & BV_2\tilde{V}_2 \end{bmatrix}. \tag{3.21}$$

Step 4: Once we have the desired column scaled form, we can perform a QR factorization,

$$B\overline{V} = QR. \tag{3.22}$$

Step 5: The WLS system is solved, as

$$QR\overline{V}^T x \overset{LS}{=} Wb \tag{3.23}$$

$$\Rightarrow x_{LS} = VR^{-1}Q^TWb \tag{3.24}$$

3.3 Discussion of the algorithm

In the previous section, we described an algorithm to compute the singular value decomposition of an ill-row scaled matrix. In [22], the first step of the algorithm computes a column pivoted QR factorization, that orders the ill-conditioning in the factor R to correspond to a descending order of D. Under this assumption, the following step succeeds for them, with an unpivoted QR factorization. In CODA, we rely on the SVD to do the ordering for us, although due to the nature of implementation of the orthogonal reflections necessiated pre-ordering for the best results. The SVD transforms the bad row-scaling into a bad column scaling, and the columns are ordered roughly in decreasing order of norm so that a normal QR routine has no issues constructing good factors for it. The pinch-point for CODA is the accurate computation of SVDs.

It is known that if a matrix is perturbed by $O(\epsilon)$, then the singular subspaces of A are perturbed by $\frac{\epsilon}{\delta}$ where δ is the separation of the singular values (see [19, Theorem 8.6.5]).The condition number of our WLS system is determined by the ratio of the largest and smallest entry in W. Since we assume W to be ill-conditioned, the smallest singular value is extremely small. δ then is very small, and so the singular vectors V of WA are inaccurate. More specifically, it is known that the angular error between the computed singular vector $\hat{v}_i$ and the actual

singular vector v_i is given by

$$\theta(\hat{v}_i, v_i) \leq \frac{p(m,n)\epsilon\|A\|_2}{\text{gap}_i} \tag{3.25}$$

in LAPACK implementations **GESVD,GESDD**, where $p(m,n)$ is a moderately growing function of m, n, ϵ is the machine precision and $\text{gap}_i = \min_{i \neq j} |\sigma_i - \sigma_j|$, the smallest absolute spacing between the singular values. For the ill-conditioned WLS system, the range of singular values exceeds the machine precision. Hence there is a non-zero set of singular values smaller than ϵ. Since all these singular values are effectively zero, gap_i is very small over this set of singular values, and so the singular vectors are far from being accurate. In the refinement step of CODA, we isolate the accurately computed singular values as suggested by the heuristic tolerance η. Singular values smaller than σ_1/η are deemed inaccurate and hence are refined.

Since the first k singular vectors V_1 corresponding to the correctly computed singular values are also accurate, we apply them to the matrix WA. By the nature of SVD, we get an appropriately column-scaled matrix $U\Sigma$, whose first k columns are accurate, but the rest need to be refined. We now consider the partitioned matrix WAv_2. Since we consider a subset of the original set of columns, the largest singular value is diminished, and the smallest singular value is increased, compared to WA. In addition, we now know that the largest singular value is bounded by σ_1/η. So long as this partitioned residue WAV_2 has singular values

smaller than the machine precision, there are singular vectors that are incorrectly computed. However, we are once again able to compute the k singular values that correspond to a scale η accurately. Hence the corresponding singular vectors are also computed accurately. In the extreme case, we would end up peeling only one column and one singular vector at a time. At each iteration, the parameter gap_i corresponding to a given singular value σ_i is increased as compared to the previous iteration. We don't handle the situation where there are exactly repeated singular values; this implies a defective eigen space and non unique decompositions. Such cases are not of practical interest to us and we ignore them. Subsequent to a reliably computed SVD and $WAV = U\Sigma$, we now consider a regular QR decomposition as discussed in [22]. The proof of their solution would directly apply to CODA, with small modifications as needed corresponding to Step 1 of their algorithm. The preceeding discussion gives an informal sketch of why CODA computes an accurate column-scaled version that can be used by a standard QR routine to solve the WLS system.

3.4 MSN Interpolation as a WLS problem

We now establish the significance of CODA to MSN Interpolation. Consider the MSN intepolation problem given the Chebyshev Vandermonde matrix V at

grid points $\mathbf{x}_i$, samples $\mathbf{f}$ and the Sobolev weight matrix D. Then, as we saw in equation 2.19 the MSN interpolant is given by

$$p_M(\mathbf{x}) = V(x)D_s^{-2}V^T(VD_s^{-2}V^T)^{-1}\mathbf{f}, \tag{3.26}$$

where $V^T(\mathbf{x})$ is the Chebyshev Vandermonde matrix at $\mathbf{x}$.

We can rewrite the value of the interpolant a x as specified using a weighted sum of samples $\mathbf{f}_i$, where the weights depend on $\mathbf{x}$. In this notation, one may write the above equation as

$$p_M(\mathbf{x}) = w^T(\mathbf{x})\mathbf{f}_i \text{ , where, } w(\mathbf{x}) = (VD_s^{-2}V^T)^{-1}VD_s^{-2}V^T(\mathbf{x}). \tag{3.27}$$

We now see the connection of MSN interpolation as a solution to a WLS problem, which can be specified as

$$\arg\min_{w(\mathbf{x})} \| D_s^{-1}\left(V^T w(\mathbf{x}) - V^T(\mathbf{x})\right) \|_2^2. \tag{3.28}$$

To prove that equation 3.28 yields the same weights as MSN, consider the normal equation given by equation 3.29,

$$\begin{aligned} D_s^{-1}V^T w(\mathbf{x}) &\overset{LS}{=} D_s^{-1}V^T(\mathbf{x}) \\ \Rightarrow VD_s^{-2}V^T w(\mathbf{x}) &= VD_s^{-2}V^T(\mathbf{x}) \qquad (3.29) \\ \Rightarrow w(\mathbf{x}) &= (VD_s^{-2}V^T)^{-1}VD_s^{-2}V^T(\mathbf{x}). \qquad (3.30) \end{aligned}$$

From equations 3.30 and 3.27 we see that MSN interpolation can be set up as a WLS system, which can be solved reliably using CODA. Of course the idea of com-

puting the weight at every point is redundant, since the coefficients a completely specify the polynomial throughout the interval of interpolation. For interpolation, one can still use the idea of computing a more accurate orthogonal decomposition using CODA to compute these coefficients a. The WLS formulation is useful in the construction of FD weights for linear operators through MSN as shall be shown in the next chapter.

3.5 Numerical results

In this section, we compare the performance of methods to solve ill-conditioned WLS systems. We consider random systems as well as MSN interpolation systems, with Sobolev weights. We also consider random weights. A known random solution was used to generate the RHS of these systems. The maximum relative error between the computed solution and the actual solution is plotted as a function of increasing ill conditioning. The ill conditioning is chosen to be a diagonal matrix with a negative exponential scaling. This scaling parameter s is increased to increase the ill conditioning. These experiments were carried out in Matlab. Perhaps due to manner of implementation of the SVD as well as the QR factorizations, the ordering of the diagonal weights seemed to matter. Although this not completely clear, there are indications that the ordering of weights in such

systems does matter, as reference by Vavasis in [38]. Hence, sorting the equations in the order of weights was considered. Comparison was drawn between QR factorization and CODA under different ordering of the weights.

3.5.1 Random matrices, random weights

For this purpose, we chose a random matrix of V of size 500×50. The matrix is row scaled with a random positive diagonal matrix, 35 of whose entries, were randomly amplified by 10^{16}. Also to control the ill-conditioning in a systematic manner, an exponential parameter s was varied as $s = \{2, 10, 30, 50\}$. The final scaling matrix is of the form D^{-s}, where D is a diagonal matrix, whose entries were randomly picked as above. We compute the QR factorization, the SVD and the CODA. We then use these factorizations to solve the WLS system. The RHS for the system is generated by picking a set of 10 random vectors, and multiplying them by $D^{-s}V$. The maximum relative solution error is then computed for the three cases. Since we pick random matrices, we average these errors over 1000 choices of matrices and weights for each s. The table 3.1 below presents the results of solving the WLS system by the above methods. In addition, the table shows the forward error for the QR factorization computed as $\frac{\left\|QR-D^{-s}V\right\|_\infty}{\|D^{-s}V\|_\infty}$ in column 2. The parameter η for CODA was set at 100. The CODA WLS Solver used descending order of weights.

Table 3.1: Solution Error for solving random ill-conditioned WLS systems

s	QR forward error	QR soln err	SVD Soln err	CODA Soln err
2	1.5e-15	4.1e-09	4.1e-09	2.2e-14
10	3.3e-16	5.9e+01	5.9e+01	4.3e-13
30	1.5e-16	1.1e+55	1.1e+55	5.3e-01
50	1.2e-16	5.5e+110	2.0e+110	3.6e+00

In addition to the random sweep above, we perform a more careful sweep of s so as to find a workable range for CODA. We also vary η to see its effect. Figure 3.1 presents a comparison of QR, SVD and CODA accuracies, over $s = \{1, 3, 5, ...50\}$. Figure 3.2 below show experimental results for $\eta = 10$. No marked change is observed between the two values. The most important observation is that CODA has made the error almost independent of the ill-conditioning over this range of s. However, this result was generated over only 100 random matrices, and a sever case for just one is enough to throw the average error out of range. In any case, we set the safe range of operation to be $s \leq 15$.

Results for $\eta = 10^8$ are shown in figure 3.3.

3.5.2 MSN Systems

We now consider a specific MSN system and compare the accuracy of CODA against a QR decomposition based method. We use a $2D$ Chebyshev Vandermonde matrix V over a 10×10 grid, with an order 40 polynomial used along each dimension. The diagonal matrix D_s^{-1} was the 1600×1600 Sobolev weight

Figure 3.1: Comparison of WLS Solution error for Random System, $\eta = 100$

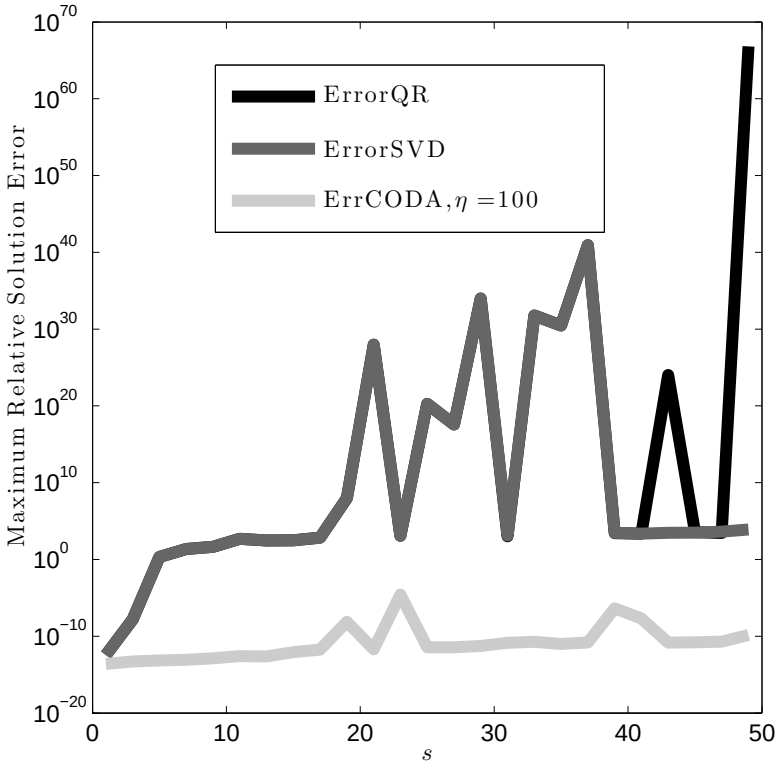


Figure 3.2: Comparison of WLS Solution error for Random System $\eta = 10$

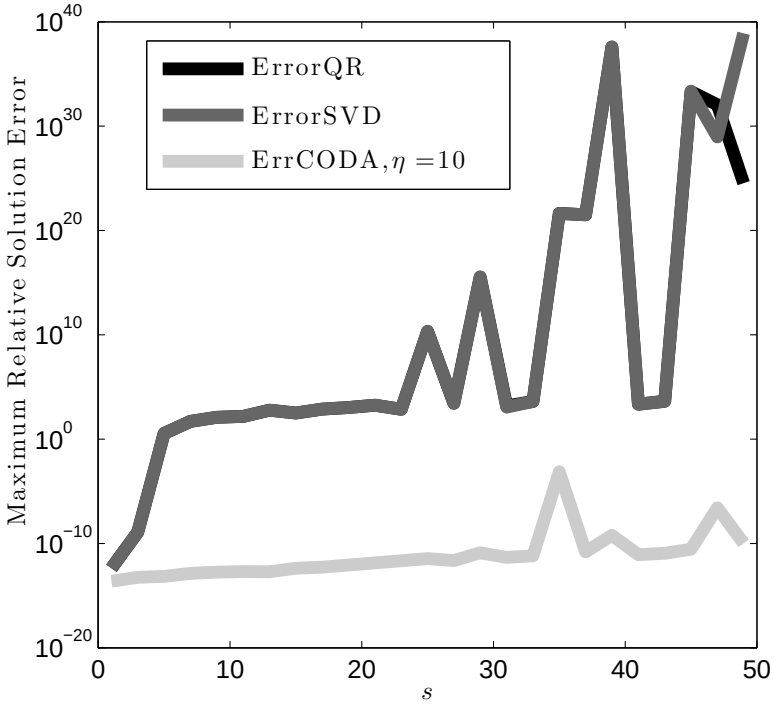
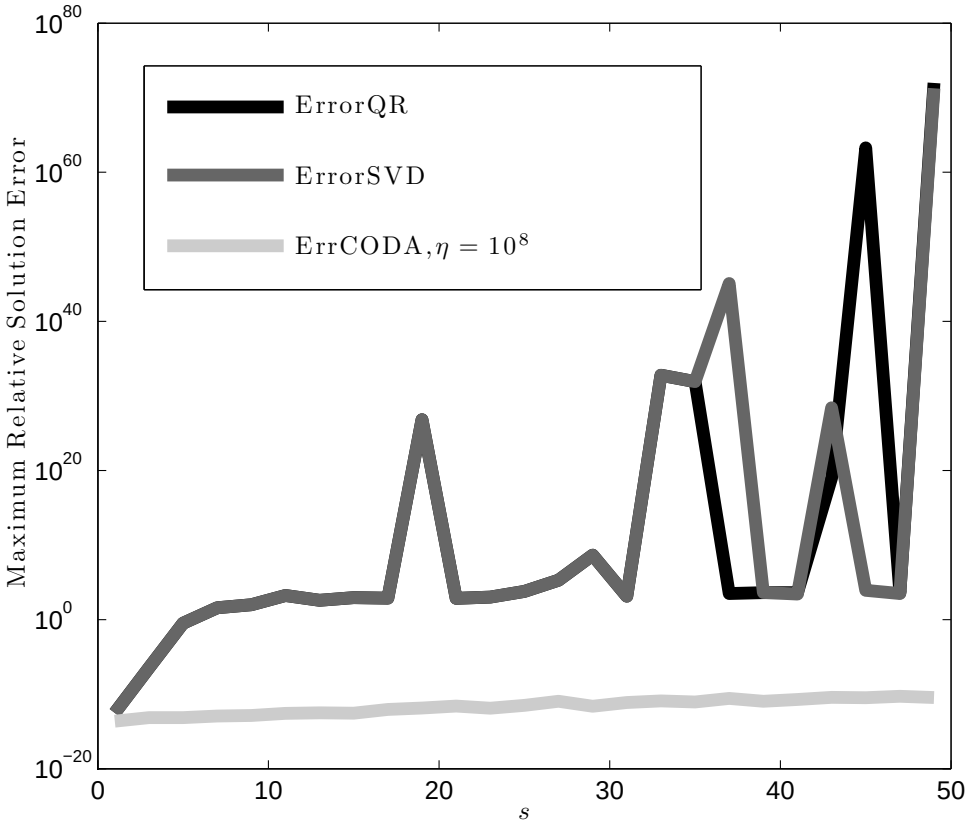


Figure 3.3: Comparison of WLS Solution error for Random System $\eta = 10^8$



matrix as in equation 2.55. Figure 3.4 shows the solution error. We now compare various ordering of the weights as well. As mentioned earlier, sorting the weights in descending order gives a definite advantage. The WLS system solved was $D_s^{-1}V^T \overset{LS}{=} D_s^{-1}b$. The RHS was once again generated using known solution vectors. The maximum relative error over 10 such solution vectors is plotted for $s = \{1, 2, ...60\}$.

Figure 3.4: WLS Solution accuracy for two dimensional MSN System, $\eta = 10$

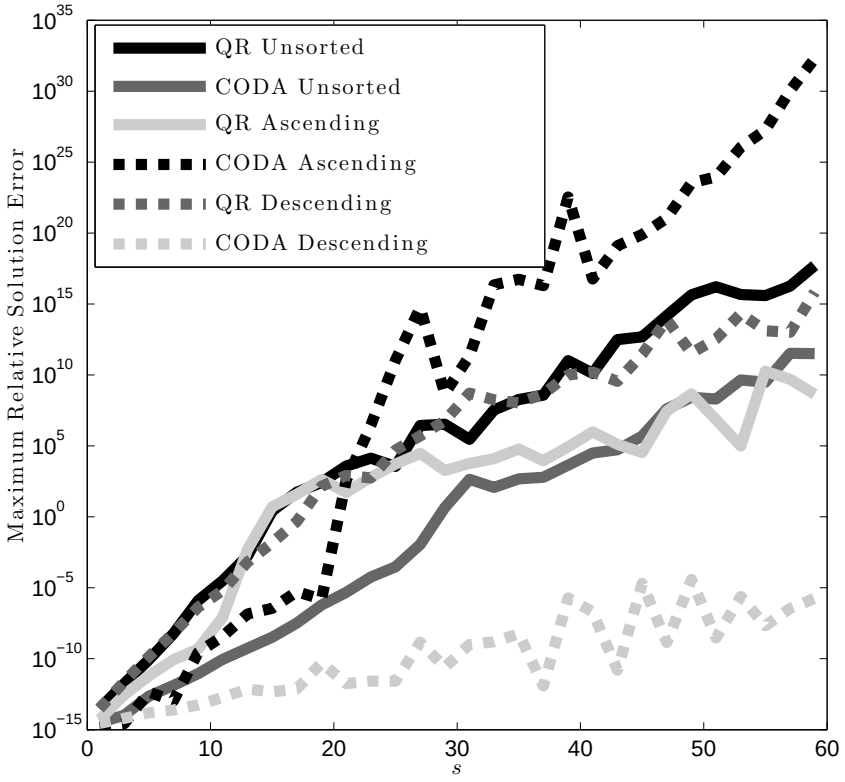
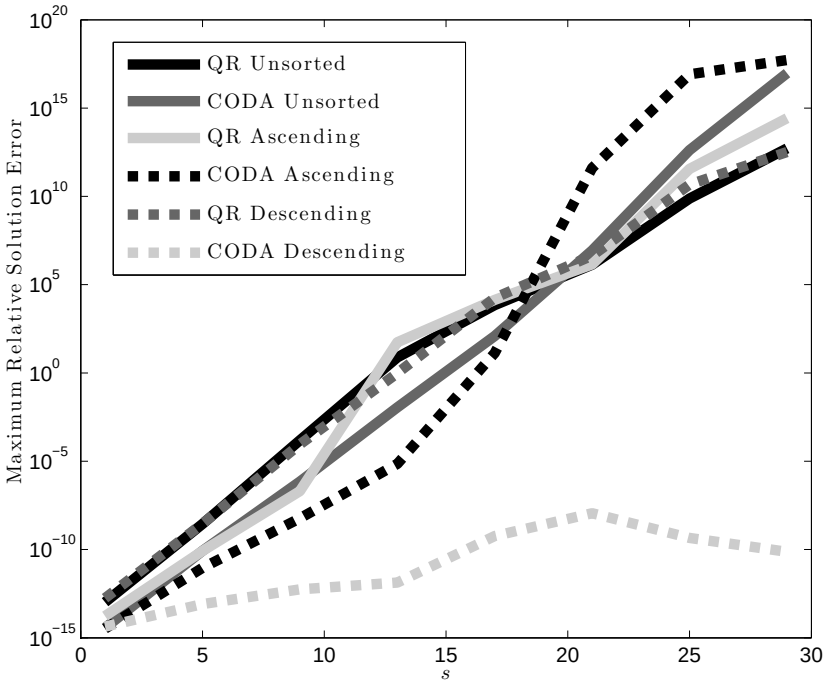


Figure 3.5 presents the accuracy comparison for a larger MSN system, over a 15×15 grid, with an order 60 polynomial. Once again, CODA with weights in descending order is seen to be most stable.

Figure 3.5: WLS Solution accuracy over a 15×15 grid



3.6 Summary

This chapter introduced CODA to accurately solving highly ill-conditioned WLS systems of a particular type. The relation of WLS systems to MSN interpolation was shown, and as shall be evident in the next chapter, CODA enables us to

compute highly accurate FD weights that we use to solve PDEs. The key idea is that through CODA, we have changed the problem from a bad row-scaled system to a bad column-scaled system. Experimental results and the theoretical discussion clearly show that CODA works reliably over wide scale of ill-conditioning for varying systems. We thus have a stable method of solving for MSN weights over a wide range of s and for reasonably large stencils. In subsequent developments we conservatively use $s = 15$.

Chapter 4

Solving PDEs using MSN : The MSNFD weights

If I have seen further than others, it is by standing upon the shoulders of giants. Isaac Newton

4.1 Introduction and Notation

In this chapter, we consider the main application of the MSN method, namely solving PDEs. We focus on two dimensional elliptic PDEs for our work. Three dimensional PDE solvers are much more complicated and not in our scope of consideration, although all that we discuss usually generalize to three dimensions unless otherwise noted.

Consider a generalized PDE of the form

$$\sum_i \sum_j \frac{\partial^{i+j} u(x,y)}{\partial x^i y^j} = f(x,y),\ (x,y) \in \Omega. \tag{4.1}$$

Such a PDE is usually solved in conjunction with appropriate boundary conditions, with each condition specified specified as

$$\sum_i \sum_j \frac{\partial^{i+j} u(x,y)}{\partial x^i \partial y^j} = g(x,y),\ (x,y) \in \partial\Omega, \tag{4.2}$$

where $\partial\Omega$ denotes the boundary. The theory of PDEs is vast, and we shall not get into details of it. For our purposes, we shall draw from the extensive array of well-posed and known PDEs, in textbooks and literature. For a introduction to PDEs, the textbook by Renardy and Rogers serves well [33]. A second order PDE of the form

$$a_{11}(x,y)\frac{\partial^2 u}{\partial x^2} + \frac{a_{12}(x,y)}{2}\frac{\partial^2 u}{\partial x \partial y} + \frac{a_{21}(x,y)}{2}\frac{\partial^2 u}{\partial y \partial x} + a_{22}(x,y)\frac{\partial^2 u}{\partial y^2} + ... = f(x,y) \tag{4.3}$$

is said to be *elliptic* if the matrix, given by

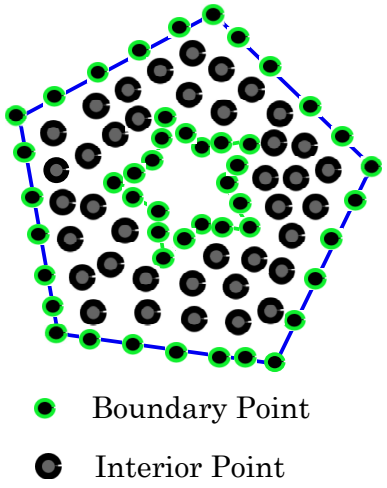
$$A = \begin{bmatrix} a_{11}(x,y) & a_{12}(x,y)/2 \\ a_{21}(x,y)/2 & a_{22}(x,y) \end{bmatrix} \tag{4.4}$$

is definite for all $(x,y) \in \Omega$. This means that for all two dimensional vectors $\mathbf{p}$, $\mathbf{p}^T A\mathbf{p} > 0, \forall \mathbf{p}$ or $\mathbf{p}^T A\mathbf{p} < 0, \forall \mathbf{p}$. In terms of eigenvalues of A, all of them are of the same sign, and none are zero. If the eigenvalues are of both signs and non-zero, the system is *hyperbolic*. If A is singular, the PDE is *parabolic*. Famous examples

of these three class of PDEs are the Laplace's equation, the wave equation and the heat equation respectively. In general, we are concerned with PDE systems whose leading term coefficients are strictly of the same sign, and these are *elliptic*.

Since we are interested in a discrete approximation to the unknown solution, let $\mathbf{x}_i = (x_i, y_i)$ be a set of grid-points in the domain Ω. In addition, we need to discretize the PDE and the Boundary Condition as well at points we call the specification points $\mathbf{y}_i$. In many cases, the grid points may be the same as the specification points. If a specification point is an interior point, i.e if $\mathbf{y}_i \in \Omega$, the PDE as per equation 4.1 is specified. Otherwise, if the point is a boundary point, the boundary equation is specified. Figure 4.1 gives an example of domain with indication of interior and boundary specifications.

Figure 4.1: PDE Domain and discretization points



With these discretization points defined, we can define a discrete system of equations,

$$\left.\begin{array}{rcl} \sum_i \sum_j \frac{\partial^{i+j} u}{\partial x^i y^j}(\mathbf{y}_k) & = & f(\mathbf{y}_k),\ \mathbf{y}_k \in \Omega \\ & & \\ \sum_i \sum_j \frac{\partial^{i+j} u}{\partial x^i \partial y^j}(\mathbf{y}_k) & = & g(\mathbf{y}_k),\ \mathbf{y}_k \in \partial\Omega. \end{array}\right\} \tag{4.5}$$

We now use that discretization of the solution u, to approximate the operators in the LHS of equation 4.5. In a finite-difference type of scheme, we use a set of weights $w(\mathbf{y}_k)$ such that

$$\sum_i \sum_j \frac{\partial^{i+j} u}{\partial x^i y^j}(\mathbf{y}_k) \approx w^T(\mathbf{y}_k) u(\mathbf{x}). \tag{4.6}$$

The size of the weight vector depends on the number of samples $u(\mathbf{x}_l)$ we use to approximate the RHS. For a FD scheme using an N point *stencil*, we use an N point neighborhood of $\mathbf{y}_k$ from the set $\mathbf{x}_l$ and have an N dimensional weight vector at most. In the vicinity of the boundary, the stencil size reduces, and there may be fewer than N neighbors present; the weights are chosen to take care of this. In the introductory chapter, the traditional method of producing FD weights using Taylor series was discussed. There were problems of uniqueness, and divergence using this approach. In this chapter, we discuss a systematic method of producing FD weights using MSN interpolation. The method we discuss works on arbitrary stencil sizes and grid spacings. The convergence of the local solution directly follows from convergence of MSN interpolation.

4.2 MSNFD Weights

In this section, we discuss the approximation of differential operators using MSN. Consider the problem of finding weights so that equation 4.6 is true. Since we using a local approximation to the underlying solution, we consider a neighborhood of a specification point $\mathbf{y}_l$, say $\delta_l = \{\mathbf{x}_n : \|\mathbf{x}_n - \mathbf{y}_l\|_\infty < \xi_l\}$. The size of neighborhood ξ_l lets us pick the size of the stencil N_l we wish to use for the local approximation. Increasing ξ_l, N_l produces increasingly accurate approximations and hence higher order of weights. In the following equations, we use a multi-index $\mathbf{m}$ with $\mathbf{T_m} = T_{m,n,...r} = T_m T_n ... T_r, |\mathbf{m}| = m + n + ...r$. In the Chebyshev basis, the local solution is represented as

$$u(\mathbf{y}_l) = \sum_{|\mathbf{m}|=0}^{\infty} a_{\mathbf{m}} \mathbf{T_m}(\mathbf{y}_l). \tag{4.7}$$

Let us denote the linear differential operator $\sum_i \sum_j \frac{\partial^{i+j}}{\partial x^i y^j}$ as $\mathcal{D}^{(i,j)}$. Then one can apply the linear differential operator to the above equation to get

$$\mathcal{D}^{(i,j)} u(\mathbf{y}_l) = \sum_{|\mathbf{m}|=0}^{\infty} \hat{a}_{\mathbf{m}} \mathbf{T_m}(\mathbf{y}_l). \tag{4.8}$$

However, what we seek are compact approximations of the local solution, and the derivatives. Hence, we consider the local MSN interpolant to the solution at $\{\mathbf{x}_k \in \delta_l\}$. Since we know that the MSN interpolant converges to the underlying solution and that the derivatives of Chebyshev polynomials can be used to represent the

derivatives of the solution, we write the following equations,

$$\mathbf{T_m}(\mathbf{y}_l) \approx w(\mathbf{y}_l)^T \mathbf{T_m}(\delta_l) \tag{4.9}$$

$$\mathcal{D}^{(i,j)}\mathbf{T_m}(\mathbf{y}_l) \approx \hat{w}(\mathbf{y}_l)^T \mathbf{T_m}(\delta_l), \tag{4.10}$$

where $w, \hat{w}$ are computed using the MSN WLS formulation discussed in the previous chapter. While the first of the above equations is quite clearly just MSN interpolation through weights, the second equation is not immediately obvious. Intuitively, since Chebyshev polynomials are a basis for the underlying solution, if we can differentiate the Chebyshev polynomials accurately, then the same weights would also serve to differentiate the solution which is just a linear combination of Chebyshev polynomials. Since we need compact weights, we use local approximations to Chebyshev polynomials through the MSN interpolant, and use the interpolatory process to locally differential Chebyshev polynomials as well.

To approximate the solution, and hence Chebyshev polynomials at $\mathbf{y}_l$ using the MSN interpolant at $\mathbf{x}_l \in \delta_l$, we have a WLS system to solve for weights w_l such that

$$w_l = \arg\min_w \| D_s^{-1} \left(V^T w - V^T(\mathbf{y}_l) \right) \|_2^2. \tag{4.11}$$

In this equation V is the Chebyshev Vandermonde matrix at the local interpolating grid points $\mathbf{x}_k \in \delta_l$ and $V(\mathbf{y}_l)$ is a Chebyshev Vandermonde row matrix composed of Chebyshev polynomials evaluated at $\mathbf{y}_l$. In the previous chapter, we showed

how the above system corresponds to MSN interpolation through coefficients of Chebyshev polynomials. In order to find FD weights that approximate $\mathcal{D}^{(i,j)}$, we set us a modified WLS problem of the form,

$$w_l = \arg\min_w \| D_s^{-1} \left(V^T w - \hat{V}^T(\mathbf{y}_l) \right) \|_2^2, \tag{4.12}$$

where V is the Chebyshev Vandermonde matrix at the local interpolating grid points $x_k \in \delta_l$ and $\hat{V}$ is a row-matrix composed of derivatives of the Chebyshev polynomials, as below.

$$\hat{V}^T(\mathbf{y}) = \mathcal{D}^{(i,j)} V^T(\mathbf{y}_l) = \begin{bmatrix} \mathcal{D}^{(i,j)}\mathbf{T_0}(\mathbf{y}_l) \\ \mathcal{D}^{(i,j)}\mathbf{T_1}(\mathbf{y}_l) \\ \mathcal{D}^{(i,j)}\mathbf{T_2}(\mathbf{y}_l) \\ . \\ . \\ . \\ \mathcal{D}^{(i,j)}\mathbf{T_{M-1}}(\mathbf{y}_l) \end{bmatrix}. \tag{4.13}$$

Each equation in 4.12 is written as below,

$$\mathcal{D}^{(i,j)}\mathbf{T_m}(\mathbf{y}_l) \quad = \quad (1 + ||\mathbf{m}||_2^2)^{-\frac{s}{2}} w^T(\mathbf{y}_l)\mathbf{T_m}(\delta_l). \tag{4.14}$$

The LHS of the above equation is at a specification point $\mathbf{y}_l$, where we discretize the PDE or the Boundary condition. On the RHS, we have a linear combination of the solution at grid points $\mathbf{x} \in \delta_l$, a neighborhood of $\mathbf{y}_l$. Through the Sobolev

weights $(1+||\mathbf{m}||_2^2)^{-\frac{s}{2}}$, we are giving preferential treatment to the equations corresponding to lower-order polynomials compared to higher order polynomials. This is yet another view of looking at MSN interpolation, where in we suppress high frequency oscillations.

For every specification point $\mathbf{y}_l$ we compute the MSNFD weights using a WLS system as above. We solve for the weights using CODA, as needed by ill-conditioned WLS systems. We can then assemble the FD matrix corresponding to the PDE specified by equations 4.1, 4.2 as follows. Once the weights are computed, we have

$$\mathcal{D}^{(i,j)}\mathbf{T_m}(\mathbf{y}_l) \approx \sum_{k=0}^{N_l} w_k(\mathbf{y}_l)u(\mathbf{x}_k), \mathbf{x}_k \in \delta_l. \tag{4.15}$$

Since the discretization is local, we have a sparse system of equations of the form

$$\begin{bmatrix} F \\ \\ G \end{bmatrix} u = \begin{bmatrix} f \\ \\ g \end{bmatrix}, \tag{4.16}$$

where F and G correspond to discretizations of the interior and boundary equations respectively. If we have N_i interior specification points and N_b boundary specification points, we have $N_i \times N$ and $N_b \times N$ matrices F and G respectively. We also assumed that we have N grid points $\mathbf{x}$ where the known u is discretized. The over all system is therefore $(N_i + N_b) \times N$. If we have as many specification points in total as many grid points, then we have a square sparse system of

equations. Each row l of the sparse matrix contains weights w_l at columns that columns k that correspond to grid points $\mathbf{x}_k \in \delta_l$.

We now state the local convergence result without proof. The proof is due to Mhaskar and is presented in [6]. As per the proof, the local discretization error

$$\left|\mathcal{D}^{\mathbf{k}}u(\mathbf{y}_l) - \mathcal{D}^{\mathbf{k}}p(\mathbf{y}_l)\right| = \left|\mathcal{D}^{\mathbf{k}}u(\mathbf{y}_l) - w_l^T u(\delta_l)\right| \leq c\xi_l^{s-q/p-\|\mathbf{k}\|_1}. \tag{4.17}$$

In the bound above, ξ_l is the size of the neighborhood δ_l, s is the Sobolev parameter, q is the dimension of the problem, p is the p-norm used for defining the Sobolev norm, $\mathbf{k}$ is the multi-index corresponding to the derivative. If we consider a fixed number of points per stencil, and keep increasing the number of gird points in the domain, we get decreasing neighborhood size ξ_l. As $\xi_l \to 0$, the error reduces to zero. The rate of convergence is given as ξ^s at best, with penalties due to dimensionality and the order of the derivative itself. In this sense, as we refine the grid further, we get increasingly accurate local discretizations, Note that these discretizations are coupled as the stencils of nearby points overlap leading to a coupled system of equations. If not, we would end up solving for local solutions with no bearing to the over-all solution. Unfortunately, a proof of global convergence of the MSNFD solution is still work in progress. However, in what follows, extensive numerical evidence of the convergence of the MSNFD solution is presented over a wide class of PDEs. Also, though not proved yet, increasing the stencil size also leads to increasing accuracy of solution and increasing order

of convergence as shall be seen numerically. Consider an $N \times N$ regular grid, and an stencil with L points per dimension, i.e. and $L \times L$ stencil. Then, the area per stencil is $\frac{L}{N}^2$. The error bound above may then be written as

$$\left|\mathcal{D}^{\mathbf{k}}u(\mathbf{y}_l) - \mathcal{D}^{\mathbf{k}}p(\mathbf{y}_l)\right| \leq c \left(\frac{L}{2N}\right)^{s-q/p-\|\mathbf{k}\|_1}, \tag{4.18}$$

where L is held fixed, and N increases. In this light, we rewrite the error bound to get our rate of convergence for two dimensions and the 2-Sobolev norm as

$$\left|\mathcal{D}^{\mathbf{k}}u(\mathbf{y}_l) - \mathcal{D}^{\mathbf{k}}p(\mathbf{y}_l)\right| \leq \hat{c}\ N^{-s+1+\|\mathbf{k}\|_1}. \tag{4.19}$$

This is only the local error bound though. The global solution and its convergence are observed to be dependent on the stencil size and the grid size, but the exact relationship is not known yet. In the following section, we consider some motivation and background on higher order methods.

4.3 On order of convergence and complexity

The order of a PDE solver is the rate at which the numerical solution converges to the actual solution, with increasing grid fineness. The measure of fineness or coarseness of a grid is usually the smallest grid spacing. For uniform grids, the local grid spacing is the same for the entire domain and so the above definitions are clear. Nevertheless, number of grid points shall be used as a measure of fineness

of the grid, and the following numerical experiments shall primarily document the error in the PDE Solution with increasing grid points and increasing stencil sizes per neighborhood.

A higher order method clearly needs a coarser grid to get to a target accuracy, than a finer grid. Also, In the case of MSNFD, experimental evidence suggests that the rate of convergence increases with increasing stencil size. However, the also means that there are more weights per row of the sparse discrete matrices F and G defined earlier and hence increases computational complexity in solving the PDE system. Although not known analytically yet, it has been observed numerically that the order of convergence as a function of the stencil half-width $k = \frac{L-1}{2}$ is N^{-k}. This may be viewed as the *Nyquist picture* of higher order methods. From the perspective Nyquist theory, the smallest number of samples required to resolve the underlying solution is at least two samples per wavelength. An efficient PDE solver, strives to be as close to this rate as possible. Higher the order of the PDE solver, the closer the operating point to the Nyquist rate. However there are computational aspects that need to be considered too.

If we have an $L \times L$ stencil, then we will have a sparse matrix of bandwidth at most L^2. If we let N be the total number of specification points and grid points, then we have an $N \times N$ sparse banded matrix. The number of non-zeros in the sparse matrix is given by $O(NL^2)$. The computational cost for solving

such a sparse matrix under optimal ND ordering is $O(N^3L^3)$. When $L^2 = N^2$ corresponding to a full-dense matrix of dimension $N^2 \times N^2$, we get the known asymptotic order of $O(N^6)$ [12] [19]. Based on out observation, we have the error converging at $O\left(N^{-\frac{L-1}{2}}\right)$ while the computational complexity increases with L as $O(N^3L^{3)}$. We converge exponentially fast with increasing stencil size, with a polynomial increase in computational cost.

Given that increasing order leads to increase in computational complexity, the law of diminishing returns seems to hold true beyond a particular order. If we consider a target accuracy of ϵ, then with an L point stencil, the rate of convergence to this accuracy is $O\left(N^{-\frac{L-1}{2}}\right)$. Ignoring the constants, the minimum grid size to get to this accuracy can be computed as

$$N^{-\frac{L-1}{2}} = \epsilon \tag{4.20}$$

$$\Rightarrow \frac{1}{N} = \epsilon^{\frac{2}{L-1}} \tag{4.21}$$

$$\Rightarrow N_\epsilon = \epsilon^{-\frac{2}{L-1}}. \tag{4.22}$$

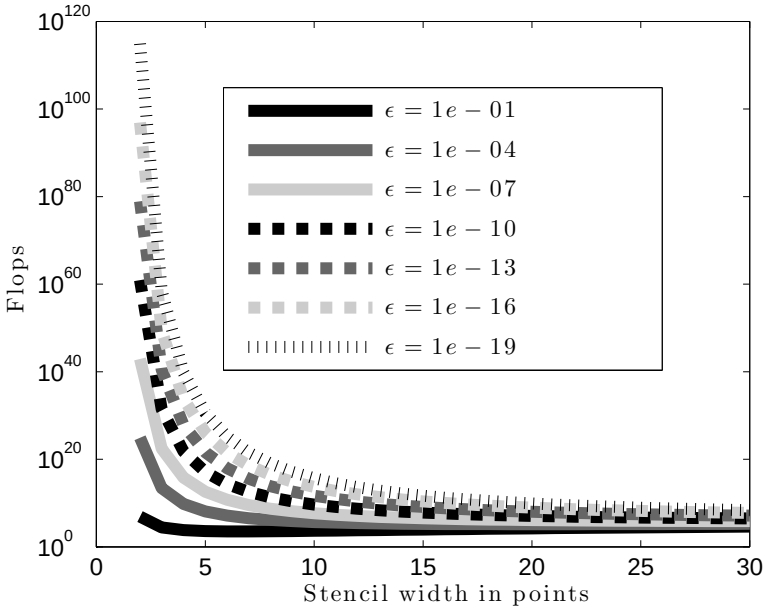
Since we have a computational complexity of $O(N^3L^3)$, the total flops and hence the time taken can be estimated as

$$t_\epsilon = N_\epsilon^3 L^3 \tag{4.23}$$

$$= \epsilon^{-\frac{6}{L-1}} L^3. \tag{4.24}$$

Figure 4.2 considers the time taken as the number of points in the stencil is increased for various target accuracies. The first inference from the plot is that increasing the order does decrease the computational complexity for any reasonable target accuracy. The second rather surprising observation is that increasing the order does not continue to decrease computational cost significantly. As is observed, increasing the stencil width beyond 20 seems to yield no obvious decrease in flops to get to a target accuracy. Nevertheless, the advantage of going in for a higher order method itself tremendous.

Figure 4.2: Order Vs Computations for varying target accuracies



4.4 Finite Element and Other higher order FD methods

With a higher order method, while 'regular' problems may be solved even more efficiently, or to much greater accuracy, 'hard' problems which were previously impossible to tackle can now be handled better. In this category are higher PDEs such as the biharmonic equation, a 4^{th} order PDE, exterior Laplace and exterior Helmholtz problems. Equations with a biharmonic leading term arise naturally in elasticity problems, Navier-Stokes equations for fluid flow etc. Exterior problems too are extremely important. The exterior Laplace problem arises naturally in magneto hydrodynamics, electromagnetics etc, while the exterior Helmholtz equation is omni-present through steady-state solutions of the wave-equation, scattering and inverse scattering problems.

Just as we consider a higher order FD method, there are higher order FEM methods as well. Two well known PDE Solver software packages available in the public domain, which incorporate higher order FEM are *Hermes* and *dealII*. Hermes is due to Pavel Solin and his collaborators at the University of Neveda at Reno [39], while dealII is due to Bangerth and his collaborators [3]. The field of finite elements is vast and well studied. The area of higher order finite elements is rapidly growing and FEM techniques are becoming increasingly popular

in several engineering applications. In our numerical experiments, we compare our performance with those of dealII. These serve two purpose. Firstly, by comparing against a standard well documented library such as dealII, we gain valuable credibility. Secondly, since we do not have exact results on our order, we control the order of the dealII solver and establish our order in comparison with dealII. dealII is a library that has been developed over a decade nearly and is still actively improved. Further it is built using C++ a low level language, and can be expected to serve as an excellent benchmark. No further discussion of FEM is presented; there are several well written textbooks in this area.

Several other higher order FD methods have also been tried before. For example, a very comparable method is that due to Wright and Fohnberg [42], where in radial basis functions (RBFs) are used to produce local FD weights. However, RBFs are known to be notoriously ill-conditioned. Further in the methods prescribed in [42], the local grid is modified to boost accuracy. As is evident from their paper increasing the smoothness of the basis function leads to saturation. The paper describes second and fourth order accurate weights, but not any higher orders. In comparable examples we shall comment on the performance of the RBF-FD technique. Also, the algorithm relies on gridding a domain locally. This may lead to problems of reuse, consistency of the grid used etc. In comparison, the MSNFD method is generic, and robust. The manner of increasing the order

of the MSNFD method is merely that of specifying a larger window size. Another serious question with the RBF-FD method is that since the RBF converges to the Lagrange or Hermite interpolant in the limit of smoothness (which increases order), the Runge phenomenon would cause divergence. Hence the method may not converge for higher orders. Other methods that I'm aware of are very specialized to particular types of problems and at most 4^{th} order.

4.5 Numerical Accuracy of MSNFD weights - 1D

In this section, we calculate MSNFD weights in one and two dimensions that correspond to derivatives in a few local grid configurations. We consider the residue in computing these weights, which correspond to the error in computing the derivatives of Chebyshev polynomials using these weights. The recurrence

formulae for the derivatives of Chebyshev polynomials are as below,

$$\begin{aligned} T_m(x) &= 2xT_m(x) - T_{m-1}(x) && (4.25)\\ \Rightarrow T'_m(x) &= 2xT'_m - T'_{m-1}(x) + 2T_m(x) && (4.26)\\ \Rightarrow T''_m(x) &= 2xT''_m - T''_{m-1}(x) + 4T'_m(x) && (4.27)\\ \Rightarrow T'''_m(x) &= 2xT'''_m - T'''_{m-1}(x) + 6T''_m(x) && (4.28)\\ \Rightarrow T''''_m(x) &= 2xT''''_m - T''''_{m-1}(x) + 6T'''_m(x) && (4.29)\\ \Rightarrow T'''''_m(x) &= 2xT'''''_m - T'''''_{m-1}(x) + 6T''''_m(x). && (4.30) \end{aligned}$$

The weights corresponding to derivatives up to 5^{th} order are computed by setting a WLS system of the form

$$\arg\min_w \left\| D_s^{-1} \left(V^T w - \begin{bmatrix} V_x^T & V_{xx}^T & V_{xxx}^T & V_{xxxx}^T & V_{xxxxx}^T \end{bmatrix} \right) \right\|_2^2, \qquad (4.31)$$

where the Vandermonde matrix V is evaluated at grid points $\mathbf{x}$ and the derivative Vandermonde vectors V_x etc are evaluated at the specification point $\mathbf{y}$. We first consider the weights themselves for varying number of stencil points. We first consider a regular grid of spacing h below. Tables 4.1, 4.2 and 4.3 specify the MSNFD weights. Note that the weights are only specified to some precision as the space permitted. The actual weights may have significant information in the trailing (unseen) digits. Hence for actual weights, the code should be used to generate them to full precision and used. These weights were generated with

$s = 15$ and using a polynomial based on the mesh norm of the grid points. The three point weights are the same as the Taylor series based FD weights. None of these weights are zero, there are truly higher order weights generated by MSNFD. As a further confirmation, we plot the logarithm of the absolute value of the second derivative weights with a 31 point stencil below in Figure 4.3.

Table 4.1: Three point MSNFD weights

Term	$w(\mathbf{y}-h)$	$w(\mathbf{y})$	$w(\mathbf{y}+h)$
hu'	-5.0e-1	2.0e-18	5.0e-1
h^2u''	1.00	-2.00	1.00

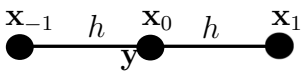
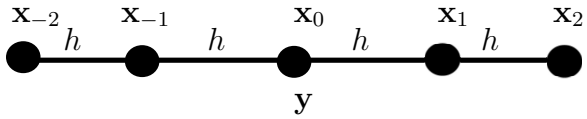


Table 4.2: Five point MSNFD weights

	hu'	h^2u''	h^3u'''	h^4u''''
$w(\mathbf{y}-2h)$	8.333e-02	-8.336e-02	-5.000e-01	1.000e+00
$w(\mathbf{y}-h)$	-6.667e-01	1.333e+00	1.000e+00	-4.002e+00
$w(\mathbf{y})$	-3.770e-17	-2.500e+00	1.505e-17	6.002e+00
$w(\mathbf{y}+h)$	6.667e-01	1.333e+00	-1.000e+00	-4.002e+00
$w(\mathbf{y}+2h)$	-8.334e-02	-8.336e-02	5.000e-01	1.000e+00

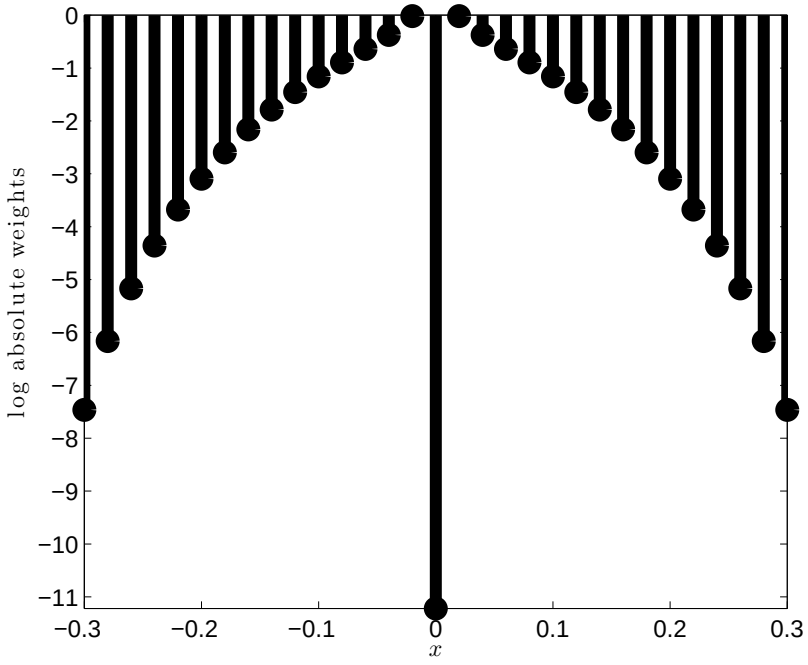


The next immediate question is the accuracy of these weights. As a measure of accuracy, we consider the error in differentiating the Chebyshev polynomials using these weights of different lengths. The derivatives were all evaluated at

Table 4.3: Nine point MSNFD weights

	hu'	h^2u''	h^3u'''	h^4u''''	h^5u'''''
$w(\mathbf{y}-4h)$	3.614e-03	-1.843e-03	-2.953e-02	3.015e-02	1.691e-01
$w(\mathbf{y}-3h)$	-3.835e-02	2.586e-02	3.022e-01	-4.079e-01	-1.515e+00
$w(\mathbf{y}-2h)$	2.006e-01	-2.016e-01	-1.413e+00	2.844e+00	4.367e+00
$w(\mathbf{y}-h)$	-8.006e-01	1.603e+00	2.038e+00	-8.188e+00	-4.867e+00
$w(\mathbf{y})$	4.706e-16	-2.851e+00	-4.778e-15	1.144e+01	2.387e-14
$w(\mathbf{y}+h)$	8.006e-01	1.603e+00	-2.038e+00	-8.188e+00	4.867e+00
$w(\mathbf{y}+2h)$	-2.006e-01	-2.016e-01	1.413e+00	2.844e+00	-4.367e+00
$w(\mathbf{y}+3h)$	3.835e-02	2.586e-02	-3.022e-01	-4.079e-01	1.515e+00
$w(\mathbf{y}+4h)$	-3.614e-03	-1.843e-03	2.953e-02	3.015e-02	-1.691e-01

Figure 4.3: 31 point MSNFD weights for first derivative



the point $\mathbf{y} = 0.1923$, a random choice of point. A centered stencil was used to compute these derivatives at this point. Table 4.4 below shows the relative error in derivatives of Chebyshev polynomials up to order 10, up to 4^{th} derivative using a 5 point stencil. For example, the error corresponding to the first derivative was computed as

$$e_x = \frac{\|V' w_x - V_x^T\|_\infty}{\|V_x^T\|_\infty}, \tag{4.32}$$

where w_x is the MSNFD weight corresponding to the first derivative. Similar errors were computed for other higher derivatives as well. Since a five point stencil is insufficient to compute the 5^{th} derivative, we do have those results here. Table 4.5 shows the corresponding errors with a 9 point stencil and Table 4.6 shows the error in computing these derivatives up to order 30. The observation, as expected is that with increasing stencil size, the derivatives are computed more and more accurately, for increasingly larger orders of polynomials. The observation again as expected is that higher derivatives are computed less accurately than lower derivatives. Intuitively, since derivatives of a differentiable function tend to become rougher, it becomes increasingly difficult to calculate their derivative accurately.

Instead of Chebyshev polynomials, consider the derivatives of the Runge function $\frac{1}{1+25x^2}$. This is a classical example of a function whose derivatives grow unbounded and hence partly being responsible for the Runge phenomenon itself.

Table 4.4: 5 point relative error in differentiating Chebyshev polynomials

Order	T'_m	T''_m	T'''_m	T''''_m
0	1.2e-17	0	2.8e-19	1.9e-19
1	7.4e-16	4.2e-16	1.1e-16	2.3e-17
2	1.2e-10	7.2e-11	1.9e-11	4.0e-12
3	5.3e-07	3.1e-07	8.3e-08	1.7e-08
4	2.4e-04	1.4e-04	3.8e-05	8.1e-06
5	3.7e-02	2.2e-02	5.9e-03	1.2e-03
6	3.9e-02	4.2e-02	2.1e-03	3.9e-03
7	1.1e-01	4.4e-02	2.6e-02	2.2e-03
8	9.6e-02	1.2e-01	4.2e-03	1.5e-02
9	2.3e-01	4.4e-02	7.0e-02	1.8e-03

Table 4.5: 9 point relative error in differentiating Chebyshev polynomials

Order	T'_m	T''_m	T'''_m	T''''_m	T'''''_m
0	3.9e-17	0.0e+00	1.3e-18	3.7e-19	4.4e-21
1	2.7e-17	0.0e+00	1.1e-19	0.0e+00	1.7e-20
2	8.4e-17	1.9e-17	4.7e-18	2.0e-18	3.9e-19
3	1.3e-13	9.9e-14	1.7e-14	8.6e-15	1.8e-15
4	8.8e-11	6.5e-11	1.2e-11	5.7e-12	1.2e-12
5	1.7e-08	1.3e-08	2.2e-09	1.1e-09	2.2e-10
6	1.1e-06	7.9e-07	1.4e-07	6.8e-08	1.4e-08
7	4.4e-05	3.3e-05	5.7e-06	2.9e-06	5.7e-07
8	5.2e-04	3.7e-04	7.1e-05	3.1e-05	7.4e-06
9	8.7e-03	6.6e-03	1.1e-03	5.8e-04	1.1e-04

Table 4.6: 31 point relative error in differentiating Chebyshev polynomials

Order	T'_m	T''_m	T'''_m	T''''_m	T'''''_m
0	1.5e-16	8.3e-18	4.6e-18	3.1e-19	1.7e-19
1	1.5e-17	1.0e-18	2.9e-19	0.0e+00	1.1e-20
2	3.4e-16	8.1e-18	0.0e+00	9.2e-19	0.0e+00
3	1.9e-17	4.1e-18	6.2e-20	6.2e-19	2.2e-20
4	1.8e-16	7.7e-18	0.0e+00	5.5e-19	1.7e-19
5	3.8e-17	1.2e-17	5.9e-19	0.0e+00	6.6e-20
6	6.1e-17	1.1e-17	2.3e-18	4.0e-19	1.7e-19
7	9.9e-17	0.0e+00	8.6e-19	9.2e-19	1.9e-19
8	2.7e-16	3.9e-17	3.5e-18	3.8e-18	7.0e-19
9	5.3e-17	2.0e-16	1.0e-17	2.4e-17	3.4e-18
10	1.1e-14	9.9e-15	1.1e-15	1.1e-15	1.1e-16
11	1.8e-14	3.3e-14	1.8e-15	4.0e-15	6.1e-16
12	1.1e-12	1.0e-12	1.2e-13	1.1e-13	1.1e-14
13	8.5e-13	2.0e-12	1.5e-13	2.4e-13	4.4e-14
14	5.1e-11	4.6e-11	5.3e-12	5.0e-12	5.2e-13
15	1.6e-11	5.3e-11	6.1e-12	6.6e-12	1.5e-12
16	1.2e-09	1.0e-09	1.2e-10	1.1e-10	1.2e-11
17	3.1e-11	6.8e-10	1.3e-10	8.8e-11	2.9e-11
18	1.5e-08	1.3e-08	1.6e-09	1.4e-09	1.6e-10
19	2.7e-09	3.5e-09	1.5e-09	5.2e-10	2.9e-10
20	9.4e-08	8.2e-08	1.0e-08	8.7e-09	1.1e-09
21	3.5e-08	5.5e-09	8.9e-09	3.0e-11	1.5e-09
22	2.1e-07	1.8e-07	2.5e-08	1.9e-08	2.8e-09
23	1.5e-07	1.3e-07	1.6e-08	1.4e-08	1.8e-09
24	7.9e-07	7.2e-07	7.9e-08	7.8e-08	7.6e-09
25	1.5e-07	2.2e-07	8.8e-08	3.2e-08	1.7e-08
26	4.2e-06	3.6e-06	4.8e-07	3.8e-07	5.2e-08
27	2.4e-06	1.6e-06	3.6e-07	1.6e-07	4.9e-08
28	5.5e-06	5.4e-06	5.0e-07	5.9e-07	4.1e-08
29	3.8e-07	3.1e-06	7.4e-07	4.1e-07	1.6e-07

In Table 4.7 below each column corresponds to various derivatives for a given stencil size, which is varies from 5 up to 100. All these results and the above were obtained on a 100 point grid in $[-1, 1]$. As can be observed, we get increasingly accurate derivatives for increasing stencil size, although beyond about 100 points, numerical errors began to interfere with the accuracy. The Sobolev parameter s was again set to 15, the largest value we prefer considering numerical issues.

Table 4.7: Error in differentiating the Runge function $\frac{1}{1+25x^2}$ at $x = -0.5378$

derivative	5pt	12pt	19pt	31pt	45pt	71pt	100pt
f'	7.2e-06	5.2e-11	1.3e-13	4.3e-14	1.3e-14	2.2e-13	7.6e-10
f''	7.8e-05	8.8e-10	1.8e-12	1.9e-14	2.1e-13	5.7e-13	5.7e-10
f'''	4.9e-03	3.9e-08	1.0e-10	4.2e-12	6.4e-13	6.0e-11	1.7e-10
f''''	2.2e-02	5.4e-07	1.2e-09	4.7e-12	3.5e-11	4.3e-10	1.4e-08
f'''''	9.5e-01	1.2e-05	3.9e-08	1.5e-09	1.5e-09	2.8e-08	6.8e-09

Table 4.8 below presents these errors for the negative exponential function at $x = 0$. The exponential function's derivatives are all exponents as well. So this test is a clear indicator of the loss in accuracy in computing weights corresponding to higher derivatives. For the 5^{th} derivative, the accuracy is several orders lower than that for the the first derivative with a 31 point stencil, where numerical issues are not yet prominent. Table 4.9 below presents these errors for the square root function at the $x = 0.354$.

We now consider increasing grid size, with the same number of points per stencil. We again consider the Runge function $\frac{1}{1+25x^2}$as an example with $s = 15$ and a 13 point stencil. Table 4.10 presents the convergence results with increasing

Table 4.8: Error in differentiating e^{-x} at $x = 0.0$

derivative	5pt	12pt	19pt	31pt	45pt	71pt	100pt
1	1.2e-09	5.8e-15	1.9e-14	3.0e-14	2.9e-14	1.2e-14	1.2e-13
2	8.6e-05	4.5e-13	2.1e-12	2.1e-13	3.3e-12	8.5e-12	5.5e-13
3	8.0e-05	1.4e-10	2.3e-10	2.3e-10	6.3e-10	7.9e-10	1.3e-09
4	1.0e+00	4.0e-09	3.1e-08	5.2e-09	6.4e-08	1.6e-07	2.9e-08
5	1.6e+00	3.0e-07	3.7e-06	5.4e-06	6.1e-06	2.8e-05	1.8e-05

Table 4.9: Error in differentiating $\sqrt{1+x^2}$ at $x = 0.354$

derivative	5pt	12pt	19pt	31pt	75pt	100pt
1	9.0e-08	8.2e-14	3.0e-14	1.4e-13	4.5e-14	1.8e-14
2	5.6e-06	2.4e-12	1.0e-12	6.3e-13	1.0e-11	4.6e-12
3	8.7e-04	2.3e-11	9.2e-11	9.8e-10	1.6e-10	3.4e-10
4	5.0e-02	1.5e-08	3.4e-08	5.3e-08	1.2e-07	9.6e-08
5	1.1e+00	6.1e-08	1.7e-07	5.3e-07	1.7e-06	3.3e-07

N. We measure the error in computing the derivatives of the above function at $x = 0.354$. As expected, we see convergence with increasing grid size. However, as the order of the derivative increases, the weights become numerically inaccurate since the Chebyshev polynomials themselves have ill-behaved derivatives. Therefore, we see the turn around in accuracy with increasing grid sizes. While this is not entirely encouraging, the hope is that with a higher order method, one need not have to reach such fine grids as to see the turn around. For example, we see that we have 6 digits of accuracy for the 5^{th} derivative at a 320 point grid. This is very significant and as many as one can hope to get given the ill-conditioned nature of this operator. Further, there are no issues for even the 4^{th} derivative, which we could compute to 9 digits before a turn around. Table 4.11 presents results for

the same problem with a 7 point stencil instead. Comparing these two tables, it is evident that increasing the stencil size does lead to increased rate of convergence!

Table 4.10: Error with increasing grid size, with 13 point stencil

Term	20	40	80	160	320	640	1200
f'	1.0e-02	4.8e-06	4.4e-11	2.2e-12	6.9e-15	8.1e-15	4.7e-14
f''	2.3e-02	2.0e-04	1.3e-08	1.7e-10	1.3e-12	3.2e-12	1.4e-11
f'''	2.4e-01	2.2e-04	7.9e-08	4.0e-09	5.9e-11	4.0e-10	2.4e-09
f''''	4.0e-01	1.6e-02	3.9e-06	2.1e-07	3.7e-09	3.1e-08	2.1e-07
f'''''	2.5e+01	1.0e-02	2.3e-04	3.0e-05	2.1e-06	1.3e-05	6.0e-04

Table 4.11: Error with increasing grid size, with 7 point stencil

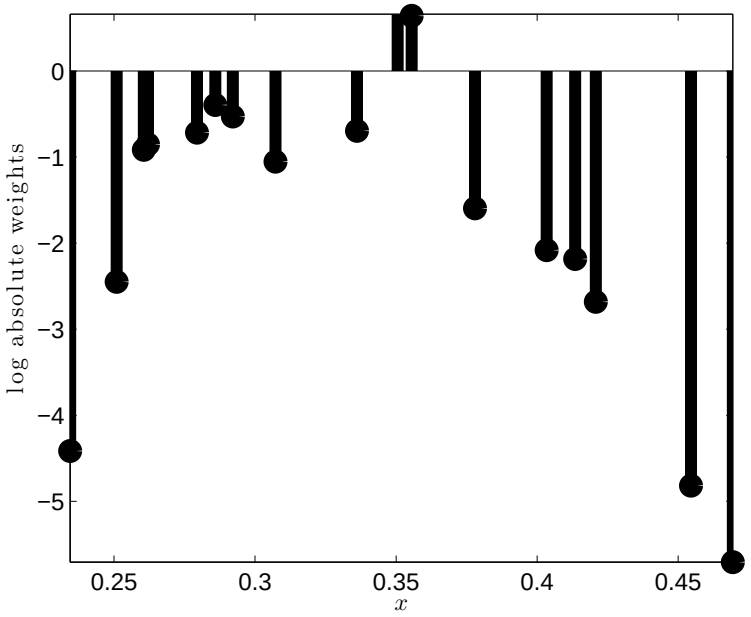
Term	20	40	80	160	320	640	1200
f'	1.4e-02	7.8e-05	7.7e-08	1.8e-08	4.6e-10	2.0e-12	4.5e-13
f''	1.7e-02	2.2e-03	1.1e-04	1.2e-06	2.9e-09	7.4e-10	1.5e-10
f'''	2.8e-01	7.8e-03	1.6e-04	3.2e-05	2.6e-06	6.5e-08	4.2e-08
f''''	1.1e-02	1.5e-01	2.9e-02	1.1e-03	4.3e-06	1.2e-05	7.9e-06
f'''''	1.9e+01	2.6e+00	4.6e-01	1.9e-01	4.9e-02	6.7e-03	1.3e-02

As a final set of numerical results in one dimension, we consider the results on a scattered grid. For this purpose, we pick as stencil size, say 17 points. We perturb a regular grid randomly to get an irregular grid. We once again consider the Runge function at a randomly chosen point. Table 4.12 below presents the error in computing the derivative of the Runge function at $x = 0.354$ using a random grid. Random grids can be extremely ill-conditioned. A reasonable one has been picked for illustration. Nevertheless, the method succeeded in producing the weights, shown in Figure 4.5.

Table 4.12: Error in derivatives of $\frac{1}{1+25x^2}$ with 17 random points

Derivative	Error
f'	3.0e-14
f''	1.1e-11
f'''	1.4e-10
f''''	1.1e-08
f'''''	9.3e-07

Figure 4.4: 17 point MSNFD weights for first derivative in a randomized grid

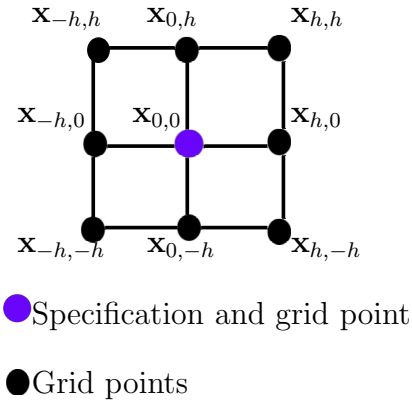


We have thus seen substantial numerical evidence of the MSNFD approach in 1D. We do not present any results of solving ODEs using MSNFD weights. We would directly be presenting more general PDE results.

4.6 Numerical Accuracy of MSNFD weights - 2D

The standard 9 point stencil is shown below in Figure 4.5 for illustration. In a similar manner, we shall consider square stencils, for illustrative purposes. There is nothing that precludes the use of any other neighborhood choice. For simplicity we restrict ourselves to rectangular regions in what follows.

Figure 4.5: Standard nine point stencil in two dimensions



4.6.1 Higher order weights for two dimensional operators

Consider a regular square stencil centered at the origin. We consider the approximation of the following operators at the origin,

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \tag{4.33}$$

$$\nabla^4 = \frac{\partial^4}{\partial x^4} + \frac{\partial^4}{\partial y^4} + 2\frac{\partial^4}{\partial x^2 \partial y^2}. \tag{4.34}$$

Equation 4.33 represents the two dimensional Laplacian operator, while equation 4.34 represents the Biharmonic operator. These are the most important operators in engineering perhaps that arise in a variety of scenarios and so we consider them. The smallest possible stencil for the Biharmonic operator is 13 point. Standard stencils for regular grids for these operators can be found in standard textbooks. For example, Abramowitz and Stegun [1] provides a catalog of known regular stencils using Taylor series expansions. Comparison of differentiation accuracy with these standard weights are also considered.

To set the references for comparison, we consider the standard weights and compute the error in computing derivatives of two functions. Table 4.13 below presents the error in computing the derivatives of a Runge type function and an exponential function near the origin, for various documented FD weights on regular grids. Note that the Biharmonic operator is extremely difficult to deal with and the loss in order for this operator. The standard 35 point weights given

fourth order accuracy for regular grids for this operator. However, there are no known general procedures to produce weights on irregular grids.

Table 4.13: Standard FD accuracy for Laplacian and Biharmonic Operators

	$\frac{1}{1+25x^2+25y^2}$		$\frac{\sin y^2}{1+100x^2+e^{y^2}}$	
Stencil pts	∇^2	∇^4	∇^2	∇^4
5	2.4e-03	-	5.0e-05	-
9	1.6e-06	-	4.2e-10	-
13	-	2.8e-03	-	1.2e-03
35	-	3.2e-06	-	1.2e-06

We now consider MSNFD weights on square stencils of varying size. The following example shows how these weights are produced over any stencil size in a robust manner. Table 4.14 below presents the 9 point weight for the Laplacian. Since the Biharmonic operator needs wider stencils, we move on to larger stencils. The weights have been truncated to one significant digit for reasons of space. The accurate weights are computed by a Matlab code to be presented in the appendix. Tables 4.15 through 4.18 present higher order Laplacian and Biharmonic operator weights. To compute these weights, $s = 15$ was used for the sake of consistency and numerical stability. The order of the interpolants was based on the mesh norm of the stencil.

Table 4.14: 9 point MSNFD weight for the Laplacian

1.1e-16	1.0e+00	2.7e-16
1.0e+00	-4.0e+00	1.0e+00
3.3e-16	1.0e+00	2.2e-16

Table 4.15: 25 point MSNFD weight for the Laplacian

-4.0e-05	1.7e-04	-8.4e-02	1.7e-04	-4.0e-05
1.7e-04	-7.3e-04	1.3e+00	-7.3e-04	1.7e-04
-8.4e-02	1.3e+00	-5.0e+00	1.3e+00	-8.4e-02
1.7e-04	-7.3e-04	1.3e+00	-7.3e-04	1.7e-04
-4.0e-05	1.7e-04	-8.4e-02	1.7e-04	-4.0e-05

Table 4.16: 25 point MSNFD weight for the BIHarmonic Operator

1.5e-02	-2.3e-01	1.4e+00	-2.3e-01	1.5e-02
-2.3e-01	3.6e+00	-1.1e+01	3.6e+00	-2.3e-01
1.4e+00	-1.1e+01	2.5e+01	-1.1e+01	1.4e+00
-2.3e-01	3.6e+00	-1.1e+01	3.6e+00	-2.3e-01
1.5e-02	-2.3e-01	1.4e+00	-2.3e-01	1.5e-02

Table 4.17: 49 point MSNFD weight for the Laplacian

-1.7e-05	1.3e-04	-4.0e-04	1.2e-02	-4.0e-04	1.3e-04	-1.7e-05
1.3e-04	-9.4e-04	2.8e-03	-1.5e-01	2.8e-03	-9.4e-04	1.3e-04
-4.0e-04	2.8e-03	-8.1e-03	1.5e+00	-8.1e-03	2.8e-03	-4.0e-04
1.2e-02	-1.5e-01	1.5e+00	-5.5e+00	1.5e+00	-1.5e-01	1.2e-02
-4.0e-04	2.8e-03	-8.1e-03	1.5e+00	-8.1e-03	2.8e-03	-4.0e-04
1.3e-04	-9.4e-04	2.8e-03	-1.5e-01	2.8e-03	-9.4e-04	1.3e-04
-1.7e-05	1.3e-04	-4.0e-04	1.2e-02	-4.0e-04	1.3e-04	-1.7e-05

Table 4.18: 49 point MSNFD weight for the Biharmonic Operator

7.5e-04	-7.0e-03	4.4e-02	-2.4e-01	4.4e-02	-7.0e-03	7.5e-04
-7.0e-03	7.0e-02	-5.2e-01	2.9e+00	-5.2e-01	7.0e-02	-7.0e-03
4.4e-02	-5.2e-01	4.7e+00	-1.5e+01	4.7e+00	-5.2e-01	4.4e-02
-2.4e-01	2.9e+00	-1.5e+01	3.4e+01	-1.5e+01	2.9e+00	-2.4e-01
4.4e-02	-5.2e-01	4.7e+00	-1.5e+01	4.7e+00	-5.2e-01	4.4e-02
-7.0e-03	7.0e-02	-5.2e-01	2.9e+00	-5.2e-01	7.0e-02	-7.0e-03
7.5e-04	-7.0e-03	4.4e-02	-2.4e-01	4.4e-02	-7.0e-03	7.5e-04

We now consider the benchmark functions and present the error in differentiating these functions at the origin using MSNFD weights. Table 4.19 consolidates a few observations. As expected, increasing the stencil size produces increasing orders. We shall consider the convergence with increasing grid size and stencil size once again in solution to PDEs. The local discretization error for the PDEs is an even stronger measure of accuracy, since it includes the errors at corners and irregular geometries. Compared to the Traditional FD weights we see that using MSNFD, we can generalize to much larger stencils as needed. It must be noted that increasing the stencil size beyond about 100 beings to cause numerical errors and hence saturation or loss of accuracy. This has also been justified from computational viewpoint, where in beyond such large stencils, the computational advantages of a higher order method begins to reduce. Perhaps the more significant advantage of MSNFD is its ability to produce weights for irregular data. Since the weights themselves have no meaning in this situation, we directly present convergence results with irregular grids. For example, for the same problems we considered, since the idea is to approximate derivatives in the center of the stencil, a grid point choice with more points closer to the origin may be helpful. For this we consider a regular grid, but use a transformation of the form $x = |x|^{\alpha} \operatorname{sgn} x,\ \alpha > 0$. This clusters points closer to the origin as we seek. Table 4.20 below compares a regular grid with clustered grid, with the

cluster parameter α set to 1.1 and 1.2. Clearly, clustering points near the origin is seen to improve the accuracy by two digits! The penalty paid for this increase in accuracy is through increased computational complexity. The irregular grid has a larger mesh norm than the regular grids. The subsequent order of polynomials M is shown in the table in the second column. It is important to note that *no* Taylor series based FD weights are known for irregular grids although they are inevitable in engineering when adapting grids to geometry. For example, a sharp corner or a singularity may warrant higher grid density near it to resolve the solution to sufficient accuracy.

Table 4.19: MSN FD accuracy for Laplacian and Biharmonic Operators

Stencil	$\frac{1}{1+25x^2+25y^2}$		$\frac{\sin y^2}{1+100x^2+e^{y^2}}$	
	∇^2	∇^4	∇^2	∇^4
3×3	2.5e-03	-	5.0e-05	-
5×5	1.5e-06	2.3e-03	4.2e-10	4.8e-06
7×7	3.0e-10	1.1e-06	4.0e-12	2.6e-08
9×9	7.0e-13	1.8e-09	7.3e-15	6.7e-11
11×11	1.2e-11	5.7e-08	2.8e-14	2.4e-10

Table 4.20: Boosting accuracy using an irregular 7×7 stencil

α	M	$\frac{1}{1+25x^2+25y^2}$		$\frac{\sin y^2}{1+100x^2+e^{y^2}}$	
		∇^2	∇^4	∇^2	∇^4
1.0	82	3.0e-10	1.1e-06	4.0e-12	2.6e-08
1.1	262	3.4e-11	3.7e-07	5.0e-13	1.1e-08
1.2	827	9.0e-12	9.6e-08	2.8e-14	1.7e-09

4.7 Summary

In this chapter, an extensive study of MSNFD weights was presented. Starting with a sound motivation for higher order accurate FD weights, details about the computation of MSNFD weights was presented. Extensive numerical results were considered in one and two dimensions and the challenge in computing good weights for higher order derivatives was illustrated. The set of one and two dimensional weights on regular grids were shown. Their accuracy was compared to Taylor series based FD weights. With MSNFD, much higher orders of accuracy were obtained and stencils of arbitrary sizes were generated with ease. Accuracies of up to 15 digits for the Laplacian and 11 digits for the biharmonic operator were shown to be attainable. Another important feature also a key point of this thesis, namely that of FD weights on irregular grids were considered. As an illustration of this, a clever choice of an irregular grid was shown to further improve the accuracy for the biharmonic operator using MSNFD. These experiments are key to deciding the operating point for the PDE Solver to be discussed in the next chapter. Elementary PDE concepts were presented in the introduction as a part of the motivation. The next chapter deals with the implementation aspects and presents extensive numerical results from solving several classes of PDEs using MSNFD.

Chapter 5

Solving PDEs using MSN :

Numerical Results

I think, therefore I am. Descartes

5.1 Construction of the MSNFD PDE Solver

In the previous chapter, we introduced PDEs and their discretization using MSNFD. Extensive numerical evidence and some theory was presented about the generation of Finite Difference type weights using MSN. In this chapter, we use these higher order weights to solve several PDEs. Before entering the realm of numerical results, we first describe the architecture of the PDE Solver including some software aspects.

The PDE Solver we are about to describe is for two dimensions and targets elliptic PDEs. We shall not concern ourselves with any PDE theory, but rather consider well-posed problems. That is not to say that these are easy problems. In fact there many well-posed seemingly innocuous PDEs that are numerical nightmares, for example the exterior problems, and the biharmonic type problems. Given a PDE specification such as the geometry, coefficients, boundary conditions etc, a PDE Solver has two main components. The first is the construction of MSNFD weights corresponding to the derivative terms in the PDE over the specified geometry. The second is that of assembling a large sparse matrix corresponding to the PDE using the coefficients and the computed weights. Subsequently, the large sparse system is solved and the solution is thus computed.

We had introduced the notation of specification points, at which the PDE is specified, and grid points at which the solution is to be computed. The first sub-task of a PDE Solver concerns of computing the stencil points in a given neighborhood of each specification point from the grid points. Once the stencil points have been found, the local WLS system to compute the MSNFD weights is then setup and solved. These are the weights that are used to approximate the PDE at that specification point. The process is repeated for each specification point. For a reasonably dense grid, the stencil at any specification point overlaps with that of neighboring specification points. Hence we get a coupled system of

equations. If an irregular grid may cause decoupling for a fixed stencil size, one needs to adapt the size of the stencil to achieve this coupling. The MSNFD Solver implemented has the provision to adapt the grid size to include at least a specified number of grid points in the smallest possible stencil. This possibility of adaption is important in irregular grids particularly, wherein local gradients in grid density may case extreme variations in mesh norm, and subsequent ill-conditioning and instability. Each stencil is scaled to occupy the interval $[-1, -1] \times [-1, 1]$ before computing the weights. The Chebyshev polynomials are a stable basis in this interval and hence this operation.

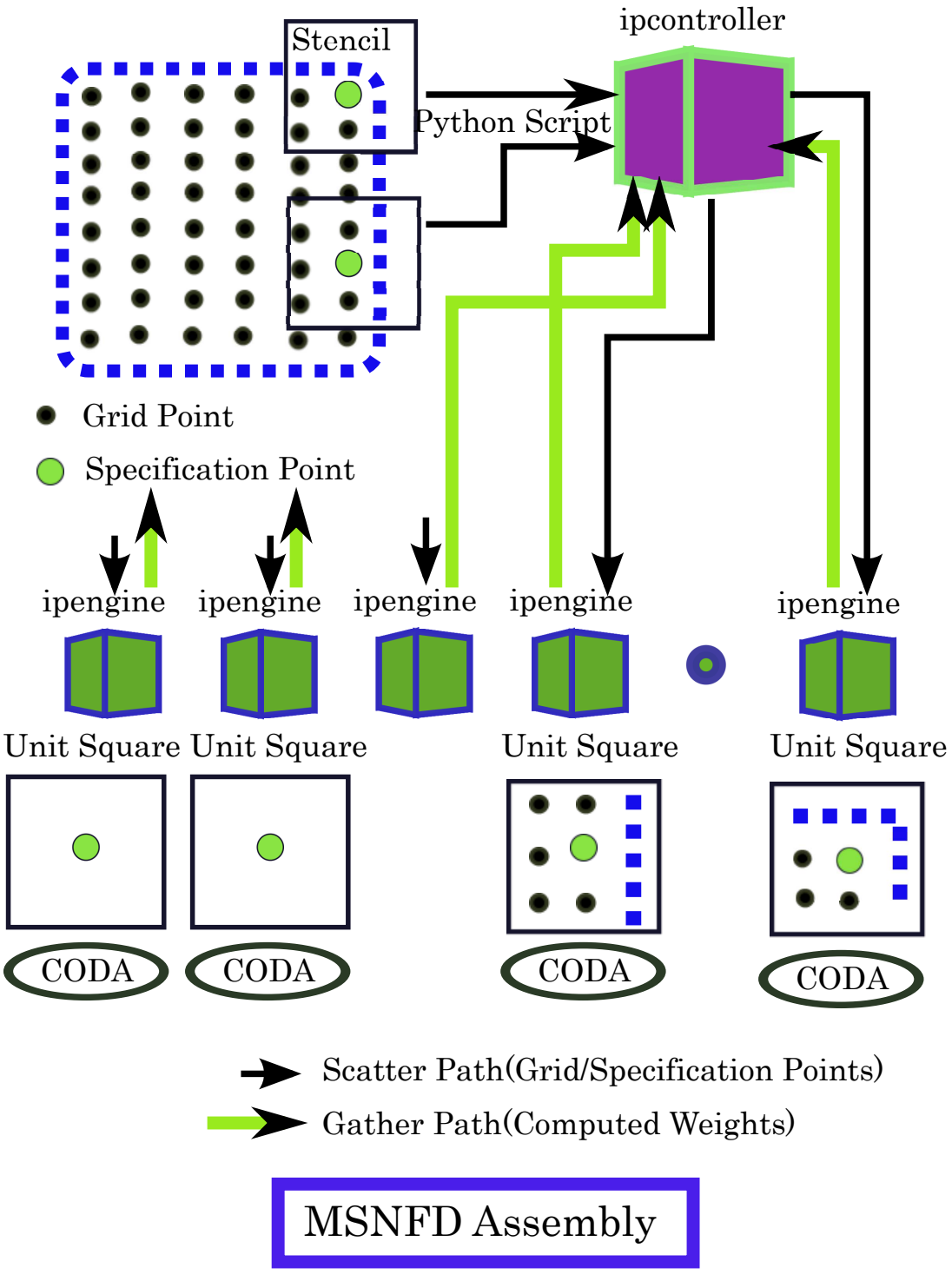
The time taken to compute the weights for the given domain involves multiple SVDs over matrices of dimension around 100×1000 at each point say. This is a moderately intensive task, and in practice computing weights over a large number of specification points over a fine grid can take a very long time. However, as is observed, the computation of weights is readily parallel. Each of the WLS computations is independent of each other. Also, these weights once computed for a given geometry over the specified grid points and specification points are re-usable over several PDEs. In fact, if all higher order weights are also computed, then several order PDEs may be solved using the same weights as well. For regular grids, these weights need to computed at only one point in the far-interior of the domain where there are no boundary influences. This far-interior optimization

can further save weight computation time for particular geometries and grids. The PDE Solver implemented provides for this feature as well, but for the sake of generality, we do not use it by default.

While our initial software of choice was Matlab by Mathworks [29], in our experience, Matlab does not scale well for parallelism. While through its C Mex interface one can make use of Matlab and an underlying MPI layer to do parallel computations, we wanted an easier higher level solution. Our answer came in the form of the Python programming language. Python through its numerous packages such as Numpy, Scipy and Sympy makes the transition from Matlab a breeze [23]. In addition to being sufficiently high level by providing ready data representations for matrices and arrays, it also have an excellent parallelism interface through its iPython parallel feature [31]. iPython at first glance is merely a shell interface to Python, but with it is the Multi-Engine Client (MEC) interface that facilitates parallelism.

Figure 5.1 illustrates the key ideas behind the parallelism in the MSNFD assembly stage. iPython uses the idea of an iPython Controller (ipcontroller) and several iPython engines (ipengine). Each iPython engine is essentially a worker, that can interpret Python code and run commands. Almost all of our parallel computing resources were through the NSF Teragrid initiative, and through the UCSB Super Computing resource support due to Dr.Stefan Borieu. In a parallel

Figure 5.1: MSNFD Assembly process



computing environment, a single ipcontroller and several ipengines, perhaps one per core depending on the memory needs are launched. The ipengines *register* with a controller, specified at the time of launch or by means of a controller file present in a known common path. This needs a shared file system access between the host node running the ipcontroller and the clients running the ipengines. The controller-engine communication happens through socket secure or unsecure socket communication over ssh or rsh.

iPython parallelism uses the simple notions of scatter, execute and gather. To elucidate further, the specification points, at each of which a WLS system is solved using CODA, are distributed amongst the several engines by the controller. The data slicing is automatic in our case; the specification points are equally split up amongst all the engines except perhaps the last, which may have the reminder after an equal split. In addition to scattering the specification points, a function calls enables the broadcast of functions, and global parameters needed for the local function execution. An execute call again as part of the MEC interface executes the local WLS computation in core over a subset of specification points assigned to that core. While asynchronous calls are possible, we use the simpler synchronous blocking parallel call, in which the total time taken is the time taken by the longest running process. Thus, after all the weight computations are over, the execute call returns. We then call the gather function over all the local weight

variables. The gather function stacks these weights each of which is an array as a list. We post process these lists into matrices as needed by us. Since the order of the scatter is the same as the order of the gather, there is no need for extensive book-keeping. The result of the assembly call is a data file that contains the input PDE specifications, grid points and specification points, stencil sizes, as well as the weights. The weights are stored as ordered triplets (i, j, w), where each triplet corresponds to the i^{th} specification point, j^{th} grid point and the corresponding weight for the operator in question. This is also the most natural form to create and store a sparse matrix later on.

Since each of the SVDs computationally intensive compared to communication overheads corresponding to scatter and gather, we do not concern ourselves with those aspects. However I wish to point out a caveat with regard to SVDs in lapack. The default SVD function called by packages such as Matlab or Python's scipy is the *dgesdd* call. This function is the divide-and-conquer based SVD that is faster that its cousin, *dgesvd*. However for our purposes of CODA, note that we never have to compute the Us corresponding to the column space and right null-space. We are only interested in the singular values and the right singular vectors in V. The divide-and-conquer function call does *not* provide for this option. As easily seen, this causes huge memory overheads for tall-skinny matrices particularly. Hence we use the somewhat slower, albeit more memory efficient

(in our case) and robust *dgesvd* function call. Python provides an easily usable Foreign Function Interface library as well. Using its *ctypes* package, one can easily interface C library functions such as lapack. Thus from an installed clapack library, the desired function call can be exported to Python. The Enthought Python Distribution is a free-for-academic use enterprise Python package that readily comes with all the necessary packages and libraries. It provides a one-shot install in several platforms and using this to install Python on all of the super computers we used was a breeze. Mayavi in Enthought's python is an excellent scientific visualization tool [32].

Once the weights have been generated and stored, the reminder of the PDE Solver concerns itself with assembling the large sparse matrix corresponding to the PDE. This matrix is constructed as a diagonally scaled linear combination of finite-difference type banded matrices corresponding to the various derivative terms in the PDE. Another aspect here is that of choosing the boundary weights and equations at the appropriate specification points. Once the sparse matrix and the RHS have been assembled, we perform a row-norm equilibration. We consider each row of the matrix and divide it by its row norm. Since the norms of the weights for each row are also precomputed and stored, this is an inexpensive operation as well. Row equilibration was experimentally observed to improve the condition number by a few orders and hence important. In contrast column equilibration

was not observed to give a similar advantage and hence not performed. The scaled system is them solved using SuperLU's sparse LU factorization algorithm [26]. For our purposes we do not consider any fill-reducing orderings but for the ones provided by SuperLU on the fly although for more complicated cases, we do use a geometric nested dissection ordering facilitated by the mesh part library due to Gilbert et al [18], as well as Metis [24]. If one needs to solve tall-skinny systems as against square systems, then SuiteSparse provides the necessary tools through UMFPACK [14], [13]. Last but not the least, a few rounds of iterative refinement are performed. For computing the residue, the various weight matrices, each at a different scale of h are applied separately to improve accuracy. Also, the LU factors are applied to the PDE term and the boundary term of the RHS separately and the solution is then added. For iterative refinement, these components are subtracted separately. The over solution is this computed and returned to the testing script. For the purpose of estimating the condition number, we use the Laub-Kenney type of condition number estimate, as described in [25].

With such a set up, a wrapper script tests the assemble and solves. Several aspects such as geometry generation, and coefficient generation, as well as symbolic computation of the RHS using any known solution are in place. Components of geometry generation and coefficient generation are parallelized. Grid geometries

are saved and reused if available. Having described the tools of our trade, we now being to look at the numerical results.

5.2 Problems on the Square

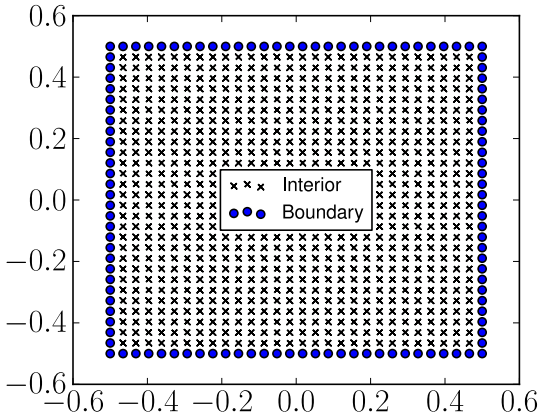
We start with two numerical examples on the Square. The first is a standard second order equation, with a negative definite leading term. This is a *nice* PDE is this sense. The right hand side of the PDE is chosen to satisfy the solution given by a Runge function $u = \frac{1}{1+x^2+y^2}$. The problem description is as per Figure 5.2. Figure 5.3 depicts the Runge function used as the solution as well. From this basic example, we clearly see that the solver works correctly. The numerical results below comprise of the local discretization error, the condition number, and the solution accuracy, in addition to the accuracy in the least square solution to MSN weights over this geometry. In Table 5.1 the grid spacing shown in the first column is decreased by increasing the number of grid points in the interval defined by the geometry. The next four columns present the local discretization error. This error is measure by using the actual solution and the finite-difference matrix to compute the RHS. The maximum relative error in the RHS is the maximum error due to local approximations. Each of the first four columns contains the local discretization error for varying local stencil sizes. The stencil sizes are varied

to roughly correspond to $3\times3, 7\times7, 13\times13, 17\times17$ stencils respectively. The last four columns contain the condition number estimate corresponding to these stencil sizes. It can be observed that increasing stencil size causes only a small increase in the condition number. However, the accuracy of weight computations drops for larger stencils due to the ill-conditioned nature of the local MSN systems, as was seen in the previous chapter. Figure 5.4 shows the LS error in weight computations. This is nothing but the worst case error in differentiating the Chebyshev polynomials with the MSNFD weights. As we had observed in the previous chapter, accuracy drops with increasing stencil sizes due to ill-conditioning. However, this does not necessarily mean that the PDE solution suffers in accuracy as is observed below. However, if the PDE itself has a high condition number, then the sensitivity of these weights being to matter.

Table 5.2 contains the maximum relative error in the solution for varying stencil sizes, and the computed order of the solution. The accuracy at the smallest grid spacing is the chosen reference. We see that increasing stencil sizes increases the order of the accuracy as expected. We have a 7^{th} order accurate solution, and the best over all accuracy of 14 digits is obtained with this. In contrast the lower first order method has only half the digits at even five times the grid size. For the example illustrated, the order drops for higher order stencils at smaller grid spacings. This is because, we are already close to the machine precision and

any instability in weight computations exhibits itself very blatantly at such fine accuracies. Nevertheless, we expect the accuracy to just saturate in the vicinity of 12 digits for decreasing grid spacing, until the maximum. In contrast, future examples shall make it clear that there is a great benefit in higher order stencils indeed, as expected, sans this saturation phenomenon.

Figure 5.2: Square Domain, *nice* problem



$$\begin{aligned} \nabla^2 u - u &= f, \text{ in } \Omega \\ u &= g, \text{ in } \partial\Omega \\ u &= \frac{1}{1+x^2+y^2} \end{aligned}$$

Table 5.1: Discretization Error and Condition Number for Square Domain

	Local Discretization Error				Condition Number Estimate			
$\frac{1}{h}$	9	49	169	289	9	49	169	289
30	9.5e-4	4.1e-5	1.6e-07	9.3e-09	1.3e+4	1.9e+4	2.2e+4	2.2e+4
100	8.2e-5	8.5e-7	2.8e-10	4.0e-11	9.6e+5	1.4e+6	1.5e+6	1.6e+6
200	2.0e-5	1.1e-7	4.8e-11	7.6e-11	1.1e+7	1.6e+7	1.8e+7	1.8e+7
500	1.2e-5	7.7e-9	3.1e-10	9.8e-10	2.7e+8	3.9e+8	4.4e+8	4.5e+8

As a second example, we consider a much harder Helmholtz problem. This problem in indefinite, since the large positive term in the PDE makes the spectrum two-sided. These problems are potentially singular, nevertheless ill-conditioned.

Figure 5.3: The Runge function for the *nice* problem

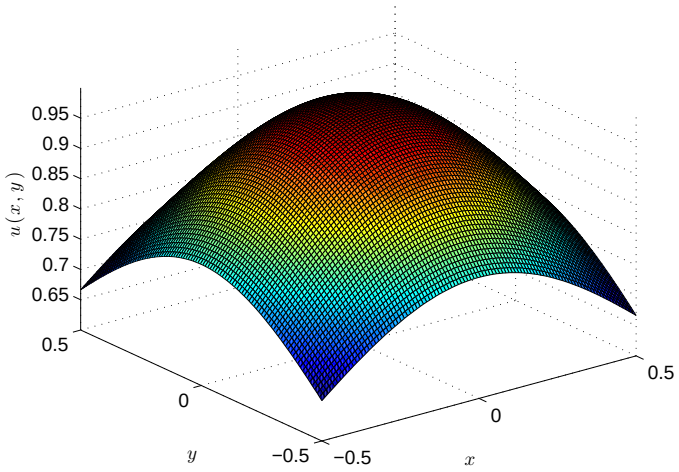
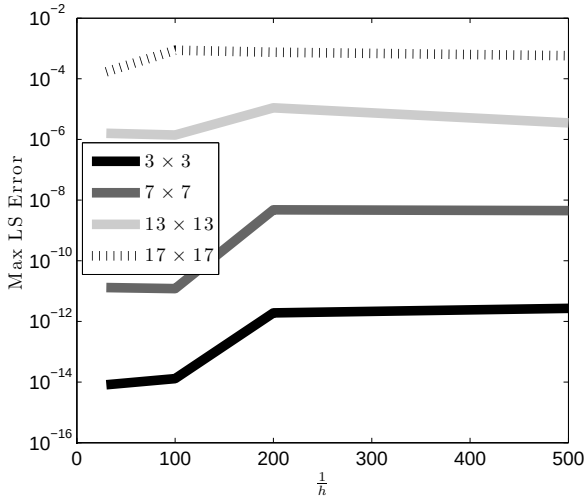


Table 5.2: Solution Error and Order for *nice* problem on the Square

	9		49		169		289	
$\frac{1}{h}$	Error	Order	Error	Order	Error	Order	Error	Order
30	1.9e-04	-	2.2e-07	-	1.2e-09	-	6.9e-11	-
100	1.6e-05	2.1	3.8e-10	5.3	1.8e-13	7.3	4.5e-14	6.1
200	4.0e-06	2.0	1.3e-11	5.1	1.4e-12	3.6	9.9e-13	2.2
500	6.3e-07	2.0	4.5e-12	3.8	3.2e-12	2.1	1.7e-12	1.3

Figure 5.4: Maximum Least Square Error in MSN weight computations



In addition to the problem being Helmholtz, we also have a rather nasty solution of choice as in Figures 5.6 and 5.7. Figure 5.5 shows the geometry and the PDE details. We solve this PDE subject to Neumann boundary conditions, i.e., we prescribe normal components of the solution's gradient on the boundary. At this point, I wish to point out a shortcoming the implemented wrapper to test the solver. We do not have the code to compute arbitrary normal derivative components to the boundary. The current solver code can handle arbitrary robin type boundary conditions though.

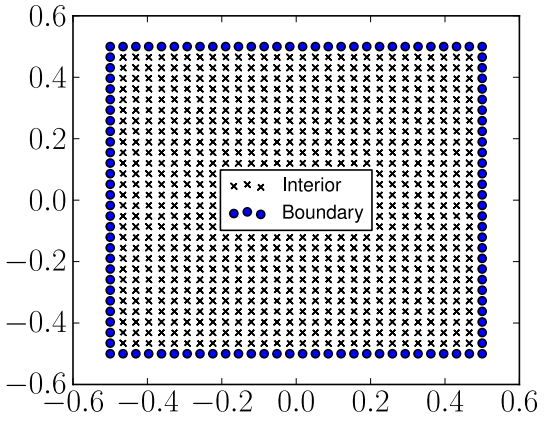
The results below contain the error in the solution and the order for varying stencil sizes. Since the solution is so rough (see the logarithmic plot of the solution in Figure 5.7 being so rough!), the Nyquist rate needed is pretty high. The lower order method (which we know is close to using FD type weights) has a hard time converging at all, while the higher order stencils obtain 6 digits of accuracy at $\frac{1}{500}$ grid spacing. Note that the largest stencil we have, 17×17 has a maximum order closer to 7!. However, these are still problems on a regular square grid. We shall demonstrate our key point, that of obtaining higher orders on irregular geometries below. However, before we move, a comment on the weight computations near the boundary is in order. At the boundary, we do not do anything special. Rather, we still pick a stencil centered at the specification point and solve a WLS system. These one-sided weights are all we use at the boundary. It would be reasonable to

attribute the condition number of the over-all system to the scale in the weights close to the boundary. This is perhaps the only manner in which the geometry of the domain expresses itself in the condition number of the system.

Figure 5.5: Square Domain, *Helmholtz* problem

$$\nabla^2 u + 10000u = f, \text{ in } \Omega$$

$$\nabla u.\hat{n} = g, \text{ in } \partial\Omega$$



$$u = \frac{\sin(10x + 201y^2)}{1 + 900(x^2 + y - 0.1)^2} + \frac{1}{1 + 721(x + y - 0.3)^2} + \frac{e^{-x^2}}{1 + 1000(x + y^2 - 0.25)^2} + \frac{1}{1 + 1120(x^2 + y^2 - 0.5)^2}$$

Table 5.3: Solution Error and Order for Helmholtz Problem on the Square

	9		49		169		289	
$\frac{1}{h}$	Error	Order	Error	Order	Error	Order	Error	Order
30	2.8e-01	-	7.8e-01	-	9.0e+00	-	2.5e+02	-
100	1.9e-01	0.3	1.3e-01	1.6	1.6e-01	3.4	2.1e-01	6.0
200	4.4e-01	-0.2	1.0e-02	2.3	5.2e-03	4.0	2.9e-03	6.0
500	2.2e-02	0.9	1.1e-03	2.3	3.3e-05	4.0	2.6e-06	6.5

Figure 5.6: The rough function for the *Helmholtz* problem - Top View

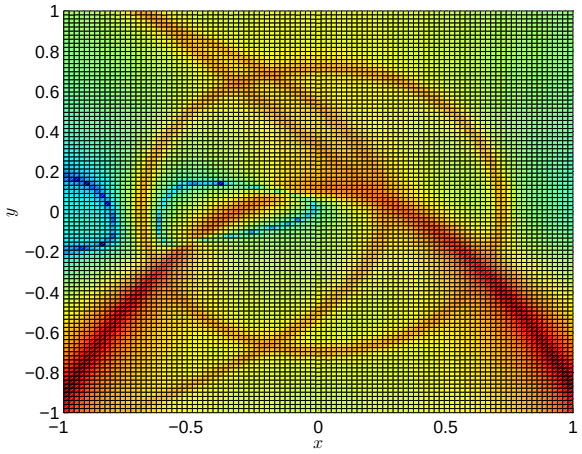
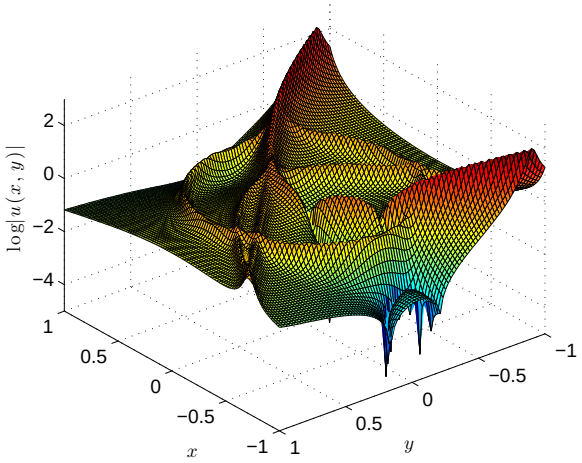


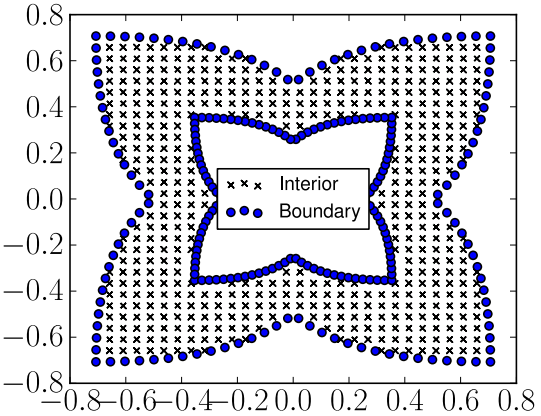
Figure 5.7: The rough function for the *Helmholtz* problem - Prespective



5.3 Variable Coefficient Problems

Having numerically established the order of our solver in a Square domain, we now consider more complex geometries. In addition, we consider variable coefficient PDEs, with positive coefficients that oscillate. If at all we chose the RHS not to be oscillatory, then we have a *multi-scale* problem. We discuss an example of such a problem in a later section. The first variable coefficient example we consider is given by Figure 5.8. The solution considered to this problem, also oscillatory, is shown in Figure 5.9.

Figure 5.8: Non-Square Domain, variable coefficient

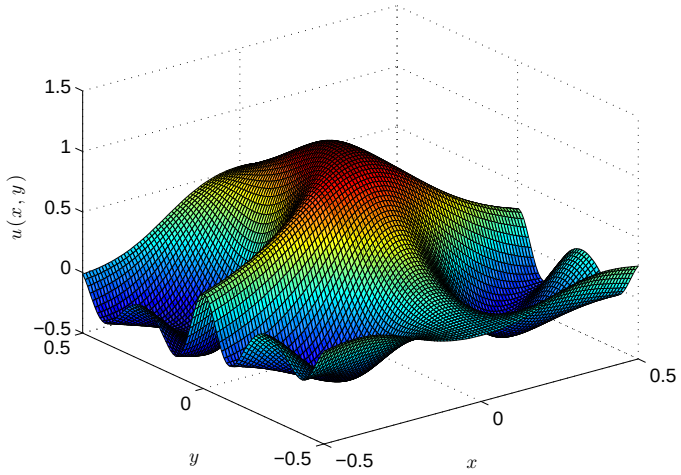


$$
\begin{aligned}
a_{11}\frac{\partial^2 u}{\partial x^2} + a_{22}\frac{\partial^2 u}{\partial y^2} - u &= f, \text{ in } \Omega \\
u &= g, \text{ in } \partial\Omega
\end{aligned}
$$

$$
\begin{aligned}
a_{11} &= \frac{1}{1+\cos 0.01x^2 + \tan 0.01y^2} \\
a_{22} &= \frac{1}{1+\sin 0.01x^2 + \cos 0.01y^2} \\
u &= \frac{\sin\left(2y\cos 6x\right)}{1+10x^2+10y^2} \\
&\quad + \frac{\cos\left(10x\sin 6y\right)}{1+10x^2+10y^2}
\end{aligned}
$$

Table 5.4 presents the solution error, and the order for two different stencil sizes. From this table it is evident that higher order of convergence is observed in the solution using MSNFD even on this non-uniform grid case. In addition,

Figure 5.9: Solution to variable coefficient problems



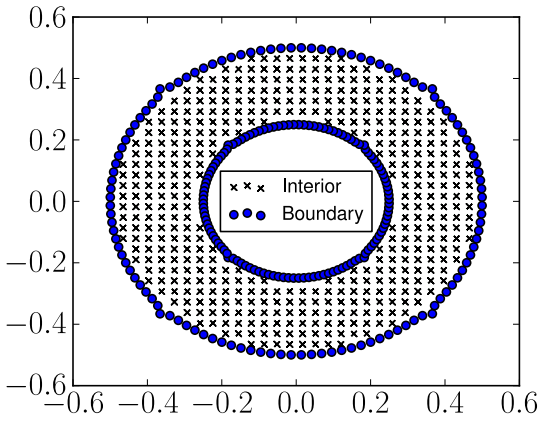
the fact that we have variable coefficients posed little or no change in the way the problem is discretized or solved. This is a well-known advantage of FD type methods over other methods such as FEM or integral equation based methods, which have a hard time even setting up these kind of problems. The grid chosen for this case (and for all cases) is just an equi-spaced, except that we restrict the grid points to those in the geometry. Also, a simple cleanup routine is used the boundary to prevent an interior grid point chosen too close a boundary point. We thus are left with a non-uniform grid.

Table 5.4: Solution Error and Order for Variable Coefficient Problem 1

	30		130	
$\frac{1}{h}$	Error	Order	Error	Order
30	2.2e-02	-	8.5e-02	-
100	1.5e-04	4.1	2.9e-04	4.7
200	9.6e-06	4.1	2.1e-07	6.8
500	2.0e-07	4.1	3.5e-10	6.9

Figure 5.10 presents the variable coefficient example on a second non-square geometry. The solution considered is the same as in Figure 5.9. The geometry is almost an annulus but for sharp corners in the out and inner boundaries. Once again, we restrict an equi-spaced grid to the interior of this geometry and discard grid points too close to the boundaries. This example as with the previous case is computed using Dirichlet boundary conditions.

Figure 5.10: Non-Square Domain, variable coefficient



$$
\begin{aligned}
a_{11}\frac{\partial^2 u}{\partial x^2} + a_{22}\frac{\partial^2 u}{\partial y^2} - u &= f, \text{ in } \Omega \\
u &= g, \text{ in } \partial\Omega \\
a_{11} &= \frac{1}{1+\cos 0.01x^2 + \tan 0.01y^2} \\
a_{22} &= \frac{1}{1+\sin 0.01x^2 + \cos 0.01y^2} \\
u &= \frac{\sin\left(2y\cos 6x\right)}{1+10x^2+10y^2} \\
&\quad + \frac{\cos\left(10x\sin 6y\right)}{1+10x^2+10y^2}
\end{aligned}
$$

Table 5.5 presents the solution error and order for this second variable coefficient example. Note the solution accuracy of 11 digits, corresponding to an order of 6 or more! The above two variable coefficient examples on non-square geometries serve to illustrate the key advantaged of MSNFD. Firstly, the method achieves higher order on non-uniform grids as well. Secondly, it can tackle vari-

Table 5.5: Solution Error and Order for Variable Coefficient Problem 2

	60		200	
$\frac{1}{h}$	Error	Order	Error	Order
30	2.2e-03	-	5.2e-04	-
100	3.0e-07	7.4	5.9e-08	7.5
200	7.0e-09	6.7	2.3e-09	6.5
500	7.0e-11	6.1	7.0e-11	5.6

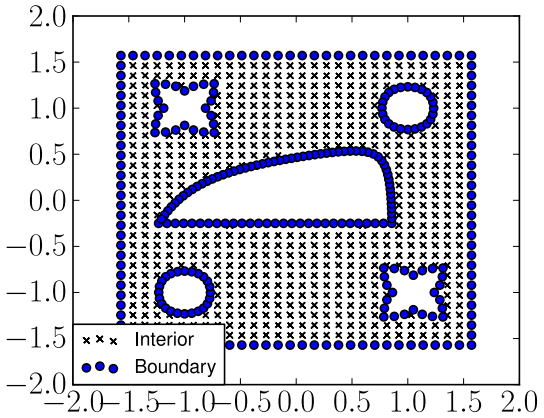
able coefficient problems with the same ease as any other problem. We shall later consider more complex variable coefficient examples with the exterior problems. We next consider more complex geometries for second order problems.

5.4 Other Complicated Geometries

We first consider the problem shown in Figure 5.11. This is a simple negative definite equation with Dirichlet data. However the geometry comprises of multiple holes in it! The solution which is oscillatory is shown in Figure 5.12. The corresponding results are as per Table 5.6. Note the large condition number. Nevertheless, we get to accuracies of 5 digits. The order of convergence for the largest stencil shown is 5. The point in this example is again that we are able to handle very complicated domains of choice, as well as get to higher order on these domains using our method.

We consider a much more trick domain, namely a Spiral. The solution is peaky runge type function shown in Figure 5.14. The geometry induced condition

Figure 5.11: A Complex Geometry with multiple holes



$$\begin{aligned} \nabla^2 u - u &= f, \text{ in } \Omega \\ u &= g, \text{ in } \partial\Omega \end{aligned}$$

$$\begin{aligned} u = &\frac{\sin(10x + 21y^2)}{1 + 10(x^2 + y - 0.1)^2} \\ &+ \frac{1}{1 + 11(x + y - 0.3)^2} \\ &+ \frac{e^{-x^2}}{1 + 10(x + y^2 - 0.25)^2} \\ &+ \frac{1}{1 + 20(x^2 + y^2 - 0.5)^2} \end{aligned}$$

Figure 5.12: Solution to problem on complex geometry

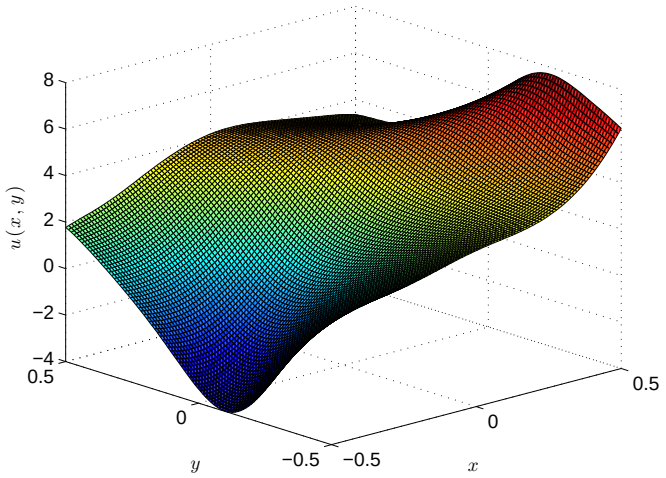
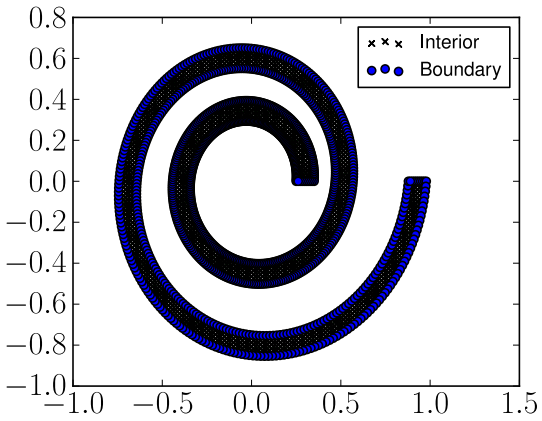


Table 5.6: Results Geometry with Multiple Holes

	30			90		
$\frac{1}{h}$	Error	Order	Cond No	Error	Order	Cond No
30	1.0e+01	-	1.3e+05	9.2e+01	-	6.8e+06
100	2.4e-01	3.1	2.0e+05	3.5e-01	4.6	1.1e+07
200	1.9e-02	3.3	1.3e+06	1.9e-02	4.5	1.3e+08
500	7.7e-04	3.4	3.1e+07	9.1e-05	4.9	6.5e+07

number is high. While this may be alleviated to an extent by a more conservative cleanup of grid points close to the boundary, we make no special efforts. This is to illustrate that we can get to accuracies as large as 7 digits even on such complicate geometries. The order of the method is also shown below. The order is a maximum of 3.3, but saturates and drops. However, this is expected since the stencil sizes used did not increase themselves. Although we increased stencil sizes, the increase in the grid density needed to increase the stencil size used is much larger than what we tried. Also, due to large condition number, the law of diminishing returns kicks in and since we are close to the best possible accuracy, the error begins to saturate, stalling the order. In comparison, a FEMs will drop order near the ever curving boundary. Since the curvature varies from one point to another, even circular elements would not allow exact boundary condition specification with FEMs.

Figure 5.13: Spiral Geometry, *Helmholtz* problem



$$\nabla^2 u + 10^5 u = f, \text{ in } \Omega$$

$$u = g, \text{ in } \partial\Omega$$

$$u = \frac{1}{1 + 100x^2 + 100y^2}$$

Figure 5.14: Solution to problem on complex geometry

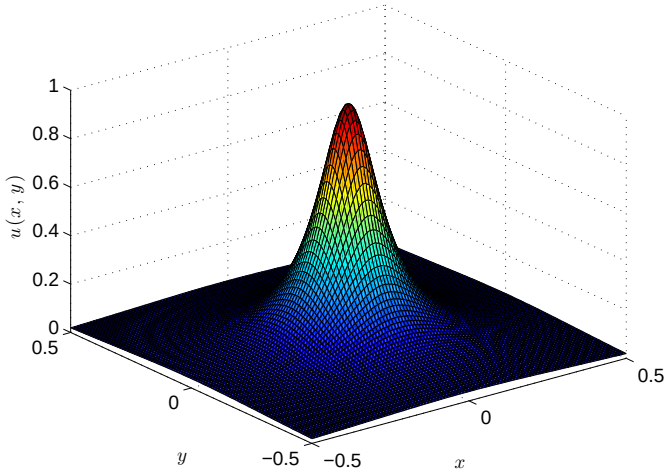


Table 5.7: Solution Error, Condition Number and Order for Spiral Problem

$\frac{1}{h}$	Solution Err	Order	Cond No	Stencil Size
30	3.7e-05	-	5.5e+05	34
100	1.2e-06	3.1	1.9e+07	58
200	5.7e-06	3.3	1.0e+10	60
500	3.7e-07	1.6	4.8e+10	64

5.5 Summary

A more extensive set of numerical results are available in our paper [7] and technical report [6]. In [7], we have presented comparison of MSNFD with dealii, a higher order FEM package. From the results it is clear that we have a more robust higher order method than dealii. Note that we compare quite favorably with deallii in terms of a coarser grid in Figure 12 in [7]. For the sake of completeness, we republish those results here. Figure 5.15 compares MSNFD and dealii on the following problem as given in the dealii tutorial step 7.

$$-\nabla^2 u + u = f,\ (x, y) \in [-1, 1]^2 \tag{5.1}$$

$$u = g_1,\ x = 1 \text{ or } y = 1 \tag{5.2}$$

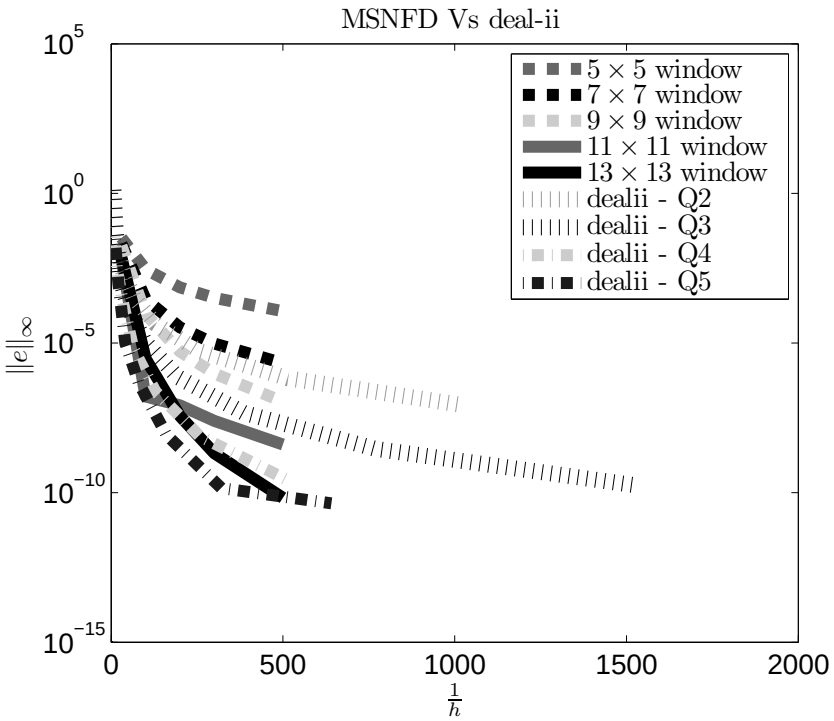
$$\nabla u.\hat{n} = g_2,\ x = -1 \text{ or } y = -1. \tag{5.3}$$

We let the solution to be $u = \sum_{i=1}^{3} e^{-\frac{|\mathbf{x}-\mathbf{x}_i|}{\sigma^2}}$, with the centers for the exponentials being $\left(\frac{-1}{2}, \frac{1}{2}\right), \left(\frac{-1}{2}, \frac{-1}{2}\right), \left(\frac{1}{2}, \frac{-1}{2}\right)$, and $\sigma = \frac{1}{3}$. However, instead of the $\mathbf{L}^2$ and $\mathbf{H}^1$ errors we are interested in the maximum errors, since these are capture boundary losses if any. Figure 5.15 below plots the solution error with reducing grid spacing for MSNFD for varying stencil sizes. We also dealii with increasing refinement, and the grid spacing is chosen commensurate with the number of triangles chosen by dealii. The order of the dealii quadrilateral elements is increased until $Q5$. Both were run until the solver ran out of memory and swap space (total of 56GB) in the

same computer. Firstly, this test provides credibility to MSNFD as a higher order method, capable of competing quite well with deallii. Although we compare only on square geometries, we have shown that we retain our higher order properties on very general geometries including the spiral. Further, dealii's advantage on curved geometries using higher order element mappings provide only a constant improvement in the errors and not a boost in order. Thus the order of the methods remains the same even with custom elements. Secondly, the boundary conditions used by dealii are modified. Since the boundary is approximated by boundary elements, the boundary conditions are also modified and reevaluated on this new boundary. This in our opinion is not correct. In practical situations, one would not have a function to evaluate these values at the new boundary, and some kind of interpolation is the only inaccurate possibility. The MSNFD method has no such problems. While our method approximates boundaries as decided by the sampling rate, it reliably uses the provided boundary conditions on the boundary and not modify it in any way. In this sense, we calculate a solution to the exact problem and not an approximate problem. Thus while higher order FEs possible, they are not as efficient in representing the PDE discretely compared MSNFD, since MSNFD required a coarser grid compared to dealii to get to a target accuracy. Further the MSNFD method is quite robust and required little or no code modifications, except for choice of a larger stencil size to scale to higher

order. Also, in situations such as variable coefficient scenarios or singular PDEs, the theoretical drudgery often precludes the use of higher order elements as with the biharmonic PDEs.

Figure 5.15: MSNFD Vs dealii



We do not present a direct comparison of time to target accuracy, since dealii is a C++ package, much more efficient in its implementation. We believe the Matlab comparison to be fairer, both Python and Matlab are at sufficiently similar level of abstraction. We also present comparison with the Matlab FEM toolbox, in several figures in [7] and again we compare quite favorably. We also present our results for another extremely hard problem, namely the multi-scale problem in section

5.4 of [7]. Multi scale problems and the difficulty with them and numerical results are dealt with in [41] by Shu et al. We have thus tested our mettle sufficiently for second order problems. We have thus garnered extensive numerical evidence in favor of the MSNFD method for solving PDEs. Through these second order PDE examples, we have shown the efficacy of MSNFD as a higher order method on complex non-uniform geometries and grids. What remains is to consider the hardest problems of the lot, namely exterior problems and biharmonic problems. We consider these in the next chapter.

Chapter 6

Numerical Results for Special Problems

To strive, to seek, to find, and not to yield. Alfred Lord Tennyson

6.1 The Exterior Laplace Problem

The Exterior Laplace problem is where we solve for Laplace's equation outside of a domain [28]. This means that we are dealing with an infinite domain for solving our PDE. Formally we specify the problem as below. in equation 6.1. In Figure 6.1 the hashed region is the infinite exterior in which the PDE needs to be

solved.

$$\nabla^2 u = 0,\ (x,y) \in \mathbf{R}^2 \backslash \Omega \tag{6.1}$$

$$u = g,\ (x,y) \in \partial\Omega. \tag{6.2}$$

However, in order for the solution to be meaningful at ∞, an additional homogenous Dirichlet or Neumann condition is specified as in Equations 6.3, 6.4.

$$u = 0,\ x = \pm\infty \text{ or } y = \pm\infty \tag{6.3}$$

or

$$\nabla u.\hat{n} = 0,\ x = \pm\infty \text{ or } y = \pm\infty. \tag{6.4}$$

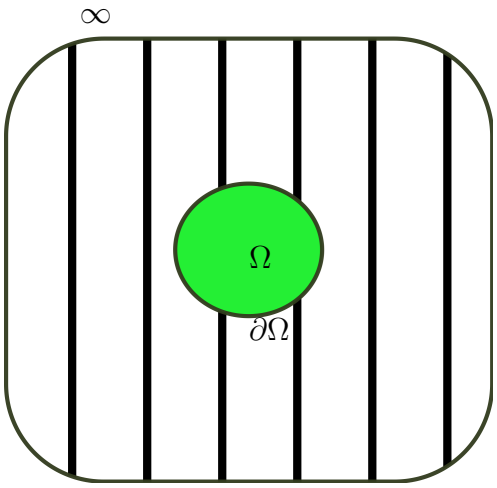


Figure 6.1: Domain of Exterior Laplace problem

Since we are dealing with an infinite domain which are infeasible to discretize, we either need to compactify the domain using some mapping, or as FEM ap-

proaches indicate, approximate the infinite boundary conditions using a finite boundary layer in the vicinity of the domain, and solve a finite domain problem now. I do not prefer the latter approach, since in my view, it works around the problem by approximating it rather than solving it directly. Related work in the field of VLSI that applying FD methods for capacitance computations solve an exterior poisson problem, for example, see [43], [15] and these approaches too find approximation solutions rather than exact ones. We consider a rather simple contraction mapping using inverse tangent functions. We also translate the boundary conditions appropriately so that they continue to be exact. We then solve the resultant PDE using MSNFD. This rather seemingly naive technique seems to yield very good results as is shown in a later section.

6.1.1 Compactification and the Resultant PDE

Let x, y denote the variables in the infinite domain. Consider the transformation of variables $x = \tan x, y = \tan y$. Under this transformation the partial derivative terms are modified as under,

$$\frac{\partial}{\partial x} \rightarrow \cos^2 x \frac{\partial}{\partial x} \tag{6.5}$$

$$\frac{\partial^2}{\partial x^2} \rightarrow \cos^4 x \frac{\partial^2}{\partial x^2} - 2\cos^3 x \sin x \frac{\partial}{\partial x}. \tag{6.6}$$

Subsequently, the interior boundary is transformed, $\partial\Omega \rightarrow \tan^{-1}\partial\Omega$, and the infinite exterior boundary is now mapped into a square $\left[-\frac{\pi}{2},\frac{\pi}{2}\right]^2$ as in Figure 6.2 below.

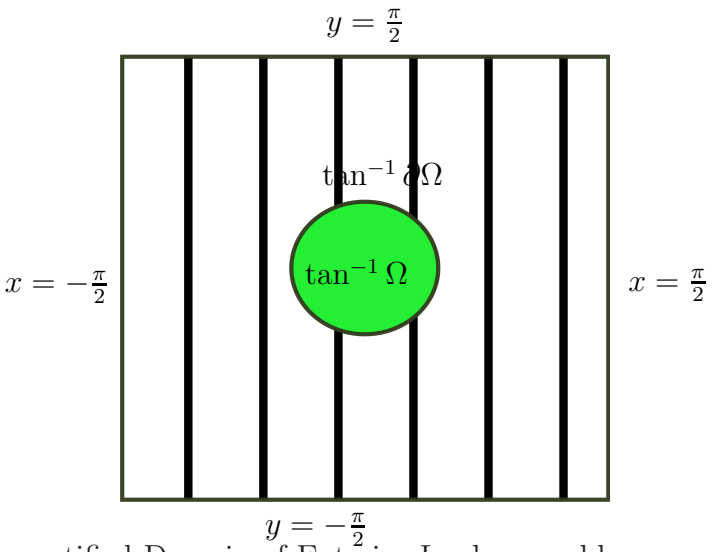


Figure 6.2: Compactified Domain of Exterior Laplace problem

Let

$$\mathcal{E}\{u\} = \cos^4 x\frac{\partial^2 u}{\partial x^2} + \cos^4 y\frac{\partial^2 u}{\partial y^2} - (2\cos^3 x\sin x - \cos^2 x)\frac{\partial u}{\partial x} \tag{6.7}$$

$$-(2\cos^3 y\sin y - \cos^2 y)\frac{\partial u}{\partial y}. \tag{6.8}$$

The original exterior Laplace problem is now of the form

$$\mathcal{E}\{u\} = 0 \tag{6.9}$$

$$(x,y) \in \left[-\frac{\pi}{2},\frac{\pi}{2}\right]^2 \setminus \tan^{-1}\Omega$$

$$u = g,\ (x,y) \in \tan^{-1}\partial\Omega \tag{6.10}$$

$$u = 0, x = \pm\frac{\pi}{2} \text{ or } y = \pm\frac{\pi}{2}. \tag{6.11}$$

Instead of Dirichlet condition at the exterior boundary, we could also use an appropriate homogenous Neumann condition.

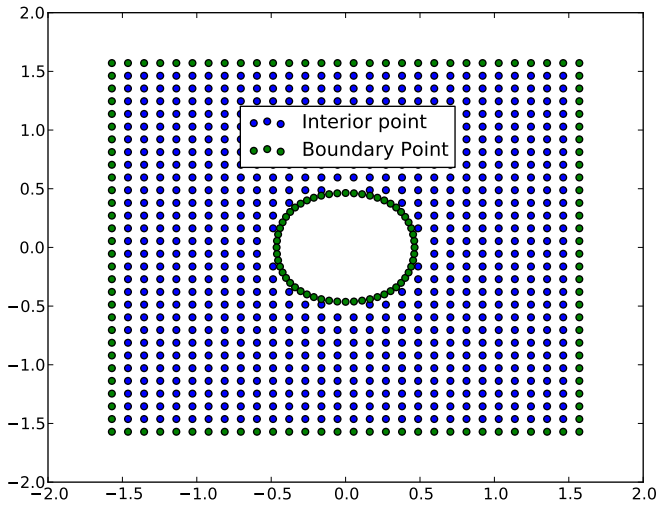
The original rather nice problem is now a much harder variable coefficient problem, which is singular on the exterior boundary. Although we do not discretize the PDE on the exterior boundary, we do so in its vicinity, making it an ill-conditioned problem. In addition, we have the first derivative term with variable coefficients which makes the problem indefinite again. However, the MSNFD method was seen to perform quite well in the case of variable coefficients as well as singular problems. The following section on numerical results indicates that the MSNFD method can be used to solve exterior Laplace problems as well with ease.

6.1.2 Numerical Results

We first consider the solution in the exterior of a circular geometry. The compactified domain and the grid are as shown in Figure 6.3 below. A known solution to the Laplace's equation is its Green Function at the origin, which takes the form $G(x, y) = \log r$ (see [36] page 273). Since this function is unbounded at $r = \infty$, this cannot be a solution to the exterior Laplace problem. We therefore consider its derivatives, which are bounded at ∞ and have bounded normal derivatives as well. With these as test functions, we compute the numerical solution given

Dirichlet data on the interior boundary and homogenous data on the exterior boundary and measure the accuracy.

Figure 6.3: Compactified domain corresponding to exterior of a circle



In the domain above, we assume the solution to be

$$u = \frac{x+y}{x^2+y^2} = \frac{\partial G}{\partial x} + \frac{\partial G}{\partial y}, \tag{6.12}$$

where G is the Green's function for the two dimensional Laplace's equation. The domain is the exterior of the circle defined by $\sqrt{x^2+y^2} \geq 0.5$.

Table 6.1 below documents the solution error obtained for the above problem. The first column contains the grid spacing used, while the next two columns present the solution error obtained using homogenous Neumann and Dirichlet conditions in that order. The last two columns compare the condition number obtained with the two types of boundary conditions. Note the large condition

numbers associated with this problem. We therefore do not carry any further results with larger stencils at this point. However, the below example does make the point that the MSNFD method can deal with such extremely ill-conditioned PDEs quite well and converge to the right solution. Following this basic example, we consider more complicated geometries, and PDEs.

Table 6.1: Exterior Laplace outside a circle with a 39 point stencil

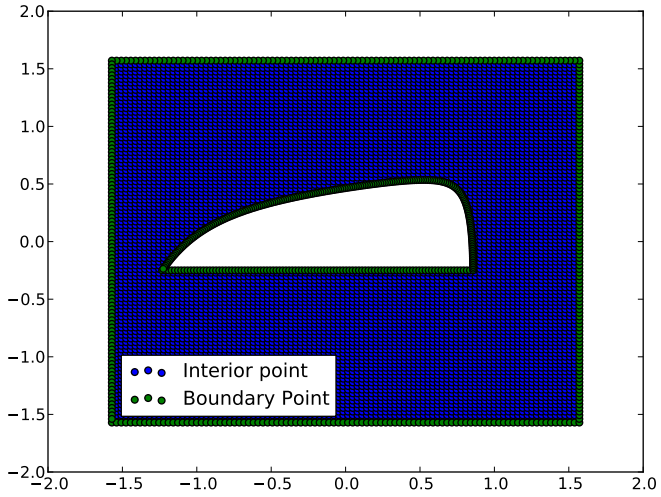
	Solution Error		Condition No	
$\frac{1}{h}$	Neumann	Dirichlet	Neumann	Dirichlet
100	2.4e-02	1.1e-03	1.6e+08	5.0e+07
200	1.2e-02	5.5e-04	4.4e+09	1.2e+09
500	4.7e-03	2.2e-04	3.2e+11	8.0e+10

As a second example we consider the exterior of a half tear-drop domain as shown below in Figure 6.4. The choice of this shape is due to its similarity to an *airfoil*. We consider the same problem as in the previous domain, with just the homogenous Dirichlet condition on the exterior boundary. Table 6.2 below presents the convergence results and the condition number.

Table 6.2: Exterior Laplace outside a half tear-drop with a 42 point stencil

$\frac{1}{h}$	Solution Error	Condition No	Local Discretization Error
30	2.3e-02	4.0e+05	1.4e+00
100	1.4e-03	3.9e+07	3.0e-02
200	7.0e-04	9.5e+08	7.8e-03
500	2.8e-04	6.4e+10	1.3e-03

Figure 6.4: Compactified exterior of half tear-drop shape



We now consider a variable coefficient example of an exterior problem. We consider a negative definite second order PDE, given by

$$\mathcal{E}\{u\} - \frac{u}{1+x^2+y^2} = f, \tag{6.13}$$

where u is chosen to be the same function corresponding to the derivative of the Green's function. We solve this problem in the exterior of the half tear-drop shape as in Figure 6.4. Table 6.3 below summarizes the condition numbers, the solution errors as well as the local discretization errors for this problem. Note that for this example, due to the extremely difficult nature of the problem the solution error saturates, although the local discretization error continues to converge. The condition number is getting larger all the same, and it would be a wise guess to

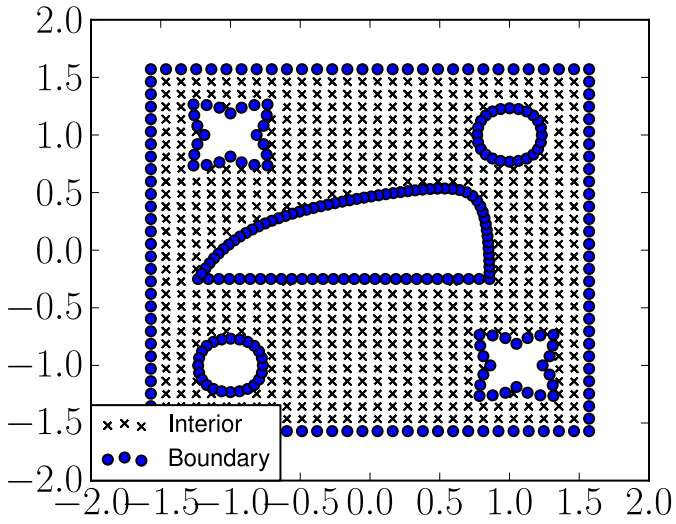
assume the stalling to be due to increasing accuracy battling increasing numerical errors.

Table 6.3: Exterior Variable coefficient outside a half tear-drop, 42 pt stencil

$\frac{1}{h}$	Solution Error	Condition No	Local Discretization Error
30	2.2e-02	3.5e+05	1.4e+00
100	2.7e-05	9.3e+06	3.0e-02
200	1.4e-05	1.1e+07	7.8e-03
500	1.4e-05	2.7e+08	1.3e-03

As a final example of the exterior Laplace type problem, we consider the domain with multiple holes, as in Figure 6.5.

Figure 6.5: Compactified domain corresponding to exterior of multiple objects



In this geometry, we first consider the variable coefficient problem,

$$\mathcal{E}\{u\} - (1.0 + \sin\frac{x}{100} + \sin\frac{y}{100}) = f, \tag{6.14}$$

where u is chosen to be the same function corresponding to the derivative of the Green's function. About 5 digits of accuracy was obtained, and no special setup was required to handle this kind of problem.

Table 6.4: Exterior Variable coefficient outside multiple objects, 45 pt stencil

$\frac{1}{h}$	Solution Error	Condition No	Local Discretization Error
30	2.3e-02	2.5e05	1.3e+00
100	2.8e-05	4.7e06	3.0e-02
200	1.4e-05	2.3e07	7.8e-03
500	1.4e-05	8.4e08	1.3e-03

As a second example, we present a positive shift to the spectrum to mimic a Helmholtz type setup the PDE. The problem takes the form in equation 6.15 below.

$$\mathcal{E}\{u\} + 10000u = f. \tag{6.15}$$

The chosen solution is the same as previously mentioned in equation 6.12. For this example, we obtained surprisingly good results. Perhaps the shift in spectrum was not as harmful as expected, although the Helmholtz problems themselves are much harder than this. The primary difficulty with the actual Helmholtz problem (zero RHS) is that the Green's function is oscillatory and very slowly decaying. Further the boundary conditions at ∞ are not as simple as with the Laplace case (see [10] page 65). We shall discuss this briefly in the concluding chapter, but in short, a more elaborate mechanism is need if we need to solve the true exterior Helmholtz problem. Table 6.5 below presents the numerical solution convergence

together with the condition numbers. Note the continued improvement in residue and the stalling of accuracy around 5 digits.

Table 6.5: Exterior indefinite problem

$\frac{1}{h}$	Solution Error	Condition No	Local Discretization Error
30	2.9e-04	4.7e+03	2.1e-04
100	8.7e-05	2.9e+05	3.2e-06
200	7.8e-05	1.2e+08	8.1e-07
500	1.4e-05	5.7e+08	1.3e-07

Finally we consider a much harder problem, with the same variable coefficients, but the chosen solution is,

$$u = -\frac{3y}{(x^2+y^2)^{1.5}} = \frac{\partial^2 G}{\partial y^2}. \tag{6.16}$$

The solution error is as shown in Table 6.4. We see that the method gets to a single digit of accuracy, something significant for such difficult problems, and perhaps in engineering applications in the presence of noise in measurements.

Table 6.6: Hard Exterior Variable coefficient problem

$\frac{1}{h}$	Solution Error	Condition No
30	0.15	2.8e+05
100	0.43	3.7e+06
200	0.18	2.5e+07
500	0.18	7.5e+08

6.1.3 Summary

We thus have garnered sufficient numerical evidence to showcase the MSNFD approach to solve the tough exterior Laplace problem in a variety of situations

as may arise in practice. This is still only a proof-of-concept level of detail. For particular applications, several optimizations may be possible as needed, such as specific scalings, adaption of grid sizes, other compactifications, etc.These are all possible extensions and perhaps part of future work in this topic. The concluding chapter discusses the exterior Helmholtz problems.

6.2 Fourth Order problems

In this last section on numerical results, we present those corresponding to fourth order PDEs which we call the biharmonic type problems. Strictly, speaking the term 'biharmonic' corresponds to PDEs with the leading term

$$\nabla^4 = \frac{\partial^4}{\partial x^4} + \frac{\partial^4}{\partial y^4} + 2\frac{\partial^4}{\partial x^2 y^2}. \tag{6.17}$$

We however misuse it only harmlessly, to refer to any PDE with fourth order leading terms. Of course, the actual biharmonic equation's leading term is positive definite and the cross-term serves to make it more so. Nevertheless, for the sake of simplicity, we shall refer to fourth order PDEs as being biharmonic. A biharmonic PDE needs two boundary conditions in conjunction with the PDE in order to be well-posed. These boundary conditions are usually picked to be one of Dirichlet type and an other of Neumann type, although general robin boundary pair could be used as well.

A biharmonic PDE thus takes the following form for us,

$$h_{11}\frac{\partial^4}{\partial x^4} + h_{22}\frac{\partial^4}{\partial y^4} + a_{11}\frac{\partial^2}{\partial x^2} + a_{22}\frac{\partial^2}{\partial y^2} + b_1\frac{\partial}{\partial x} + b_2\frac{\partial}{\partial y} + cu = f \tag{6.18}$$

$$e_{11}\frac{\partial}{\partial x} + e_{12}\frac{\partial}{\partial y} + d_1 u = g_1 \tag{6.19}$$

$$e_{21}\frac{\partial}{\partial x} + e_{22}\frac{\partial}{\partial y} + d_2 u = g_2 \tag{6.20}$$

where equation 6.18 holds in the interior of a domain Ω and the other two equations 6.19 and 6.20 hold on the boundary $\partial\Omega$. Biharmonic equations play a very important role in several engineering problems. For example, the Navier Stokes equation, elasticity equations, diffusion equations, all have fourth order spatial components. A lot of work has been done on specific fourth order problems; we attempt to remain general and understand the performance of the MSNFD method, and in general higher order methods for these higher order PDEs. The work by Greer et all is a good reference on this topic [21]. They however discuss a level set approach to converting three dimensional geometries into a level set surface and using a finite difference scheme to solve it. The paper however considers only convergence of the iterative solver for solution, and not the convergence of the computed solution to the actual solution! The paper mentions first and second order convergence for these problems. Our MSNFD method can be used in their framework as well and may in fact provide for a very powerful method to handle implicit geometries through level sets. For ill-conditioned situations such as this,

the iteratively computed solution may have no semblance to the actual solution, since the the eigen vectors and values may have little or no separation between them. Hence any Krylov subspace method would have problems, except in the presence of very good pre-conditioners perhaps. Another reference [2] generates Taylor series type grid weights symbolically. But no effort is made to tackle the Runge phenomenon and one may not expect the method to generalize to higher order. The emphasis of the paper is in the implementation of a Multigrid type of method to handle it [4]. We do not in this work pay particular attention to the solver part while this is certainly possible, and in fact something is important for complicated two dimensional cases as well as three dimensional PDEs.

There are two key challenges to higher order PDE problems. First is that of a *right* discretization. In its continuous form as in equations 6.18 to 6.20, biharmonic PDEs are well-posed and solvable. However we need to make sure the discrete form continues to remain so. Further, for practical computational considerations, one may wish to retain a square system of equations to yield a unique solution. Both the under-determined and over determined setups may need specialized care to the solver, at least a much more computationally intensive QR factorization for sparse matrices. The second is related to the condition number of the discrete system. The condition number associated with a fourth order PDE is $O(N^4)$. This means that if we have a regular grid of size 1000×1000, then the condition

number is already 10^{12}. The quantity $\epsilon_{mach}\kappa$, the product of the machine precision and the condition number is the best floating point accuracy one can hope out of the solution, which is now 10^{-4}. Any numerical results we obtain at such large grid sizes are very susceptible to numerical errors. 10000×10000 grid, there is no hope of any numerical accuracy at all! These are short-comings of any direct discretization of such an ill-conditioned operator. However, there *is* a black magic effect called regularization by discretization which may still save us in some situations.

With a second order method, the best accuracy we hope to obtain $O(10^{-6})$. But at this point numerical errors themselves are as large as 10^{-4}. So the solution we obtained may all be meaningless! However with a higher order method, there is still some hope, as we may obtain a much larger accuracy at coarser grids. Nevertheless, the fact that we are directly discretizing a fourth order PDE may not be correct. This is something we are investigating further. In this section, we consider a direct discretization of biharmonic PDEs using MSNFD. We present numerical results of convergence. We also make the observation regarding a *turn-around* of the solution accuracy due to increasing numerical instability. In the concluding chapter, we then attempt to fix the solution using a naive approach to drop the condition number of the system. We then carry on a discussion that reveals at the heart of such approach lie the 'Div-Curl' problem, which needs a

very careful treatment, not addressed by us in this thesis and part of our future work perhaps.

6.2.1 Numerical Results

We now begin with a simple example of solving a biharmonic PDE in a square domain. Consider a discretization with n_i interior points and n_b boundary points as the grid points. Let the grid points serve as specification points as well. Then we would be left with a tall-skinny system, since we would have $n_i + n_b$ unknowns and $n_i + 2n_b$ equations, since at each boundary point we have two boundary equations. In order to avoid having a specialized solver, we use a rather simple idea to get a square system. We peel off a layer of the interior specification points close to the boundary points. This layer, which we call a Ghost layer is now used to place stencils and enforce boundary conditions at nearby boundary points. For each of these Ghost points, we find the nearest boundary point and we specify that point's equation, only that we use a stencil centered at the Ghost point. At these Ghost points, the PDE is not specified, as these are coupled to the boundary equations through the stencil.

Our first example, as shown in Figure 6.6 shows the interior points, boundary points and the Ghost points. The PDE solved over this grid is given in equations 6.21 through 6.23. We have chosen to use a Dirichlet condition and

a Neumann condition on the boundary. As a solution, we chose the smooth Runge function $u = \frac{1}{1+x^2+y^2}$. Tables 6.2.1 and 6.2.1 below provides the numerical results for this problem. Table 6.2.1 presents the condition number and local discretization errors. We consider two different stencil sizes , 5×5 and 7×7. The high condition number of these problems is the first important observation. As expected, increasing the stencil size increases the local discretization accuracy. However, due to the large condition number, the solution error does not exhibit the same trend. While we are able to achieve an accuracy of 9 digits with a 49 point stencil, the solution accuracy turns around at the smallest grid spacing we consider, $\frac{1}{500}$. This is because the numerical errors involved in inverting the associated sparse matrix are penalizing us more severely than the gain we have with our larger stencil. However, it is important to note that we managed to get to 9 digits which may be sufficient for most applications, and perhaps even the best possible for this problem.

$$\frac{\partial^4 u}{\partial x^4} + \frac{\partial^4 u}{\partial y^4} + u = f, \text{ in } [-0.5, 0.5]^2 \tag{6.21}$$

$$u = g_1, \ x = \pm 0.5, \ y = \pm 0.5 \tag{6.22}$$

$$\nabla u.\hat{n} = g_2, \ x = \pm 0.5, \ y = \pm 0.5 \tag{6.23}$$

Figure 6.6: Ghost points in a square domain

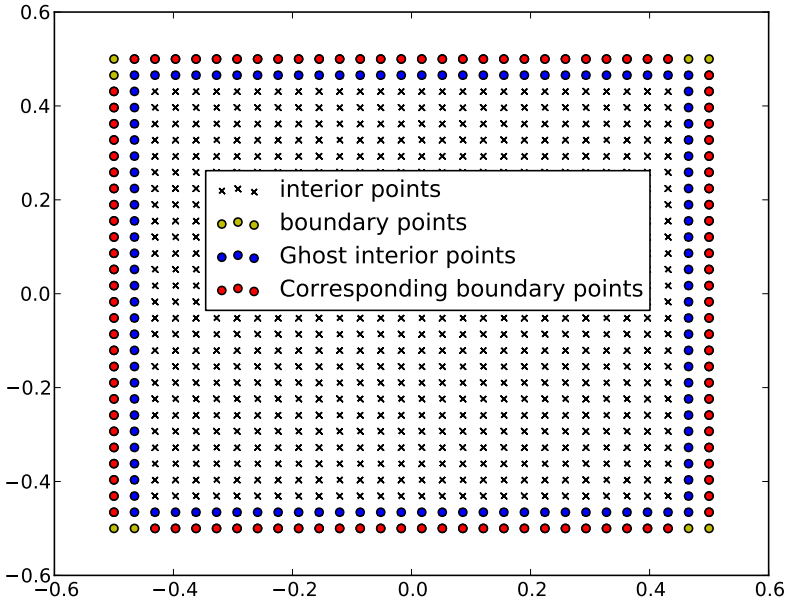


Table 6.7: Numerical Results for biharmonic PDE on a Square

$\frac{1}{h}$	Condition No		Discretization Error	
	25	49	25	49
30	1.0e+06	1.8e+06	5.7e-03	1.4e-03
100	8.5e+08	1.4e+09	4.3e-04	8.8e-05
200	4.0e+10	7.1e+10	5.2e-05	1.0e-05
500	6.1e+12	1.0e+13	6.0e-05	2.2e-05

Table 6.8: Numerical Results for biharmonic PDE on a Square

$\frac{1}{h}$	25		49	
	Error	Order	Error	Order
30	2.1e-04	-	2.5e-06	-
100	1.4e-05	2.7	9.2e-09	4.7
200	7.1e-07	3.0	6.0e-09	3.2
500	2.8e-06	1.5	4.5e-07	0.6

As far as comparable methods with FEM techniques go, there are two comments. Firstly, the complexity of conformal finite elements needed is much higher. In typically used weak-formulations, the PDE is integrated twice by parts, to search for a local solution in $\mathbf{H}^2$ space. Further, one needs to match sufficient number of moments at the edges of the elements for the over-all solution to be sufficiently smooth. The overall function we are looking for though is four-times differentiable. Hence the search space for the solution is pretty large and this is inefficient. On the other hand, one could use a H^3 local discretization, but this is so complicated that it is almost never used. The most common approach to enforcing smoothness of the overall solution is to use a weak continuity of the normal derivative, as needed by the discontinuous Galerkin methods. A recent reference on the topic of solving such higher order PDEs using an immersed FEM technique is [27]. In an exhaustive search of existing results, it was not possible to find a comparable example for the biharmonic PDE. A recent interaction with a researcher regarded to be an expert on higher order FEM techniques also confirmed this difficulty. However, our examples are absolute in their own right. We pick known solutions and measure the errors and the orders, which are golden standard in any case. The reference mention above, discusses an approach to handle discontinuity in the domain corresponding to jumpy coefficients in the PDE. Note that such a variable coefficient case is quite easy and natural to handle with

our FD type approach. Together with its high order, it can solve such complicated problems without the need for any specialized setup. Note that in general, a variable coefficient case is not even amenable to a weak representation, precluding the use of any higher order conformal elements.

The second example we present is a variable coefficient biharmonic PDE on a square domain. The problem is as per equations 6.24 to 6.26. The solution used to test this problem was $u = \frac{1+\sin(10x+10y+0.25)^2}{1+10(x^2+y-0.3)^2}$. Numerical accuracy and order are documented in Table 6.2.1 below. A quartic order of convergence was observed for this problem using a 49 point stencil and an accuracy of 7 digits was obtained.

$$(1+x^2+y^2)\frac{\partial^4 u}{\partial x^4} + (1+x^2+y^2)\frac{\partial^4 u}{\partial y^4} + u = f, \text{ in } [-0.5, 0.5]^2 \tag{6.24}$$

$$u = g_1, \; x = \pm 0.5, \; y = \pm 0.5 \tag{6.25}$$

$$\nabla u.\hat{n} = g_2, \; x = \pm 0.5, \; y = \pm 0.5 \tag{6.26}$$

Table 6.9: Numerical Results for variable coefficient biharmonic PDE

	25		49	
$\frac{1}{h}$	Error	Order	Error	Order
30	3.5e-02	-	1.7e-02	-
100	2.6e-03	2.1	8.2e-05	4.5
200	5.8e-04	2.2	4.4e-06	4.4
500	7.3e-05	2.2	4.0e-07	3.8

A third example with a much rougher solution over the square domain is considered next. The solution to the PDE in equation 6.24 is now

$$u = \frac{1}{1+1000(x^2+y-0.3)^2} + \frac{1}{1+1000(x+y-0.4)^2} + \frac{1}{1+1000(x+y-0.4)^2} + \frac{1}{1+1000(x+y^2-0.5)^2} + \frac{1}{1+1000(x^2+y^2-0.25)^2}. \tag{6.27}$$

This extremely rough function is as per Figure 6.7. The PDE itself as per equations 6.21 to 6.23 and Figure 6.6. The corresponding solution convergence and order as in Table 6.2.1. Note the order to be as high as 6 for the problem.

Figure 6.7: Rough Runge function

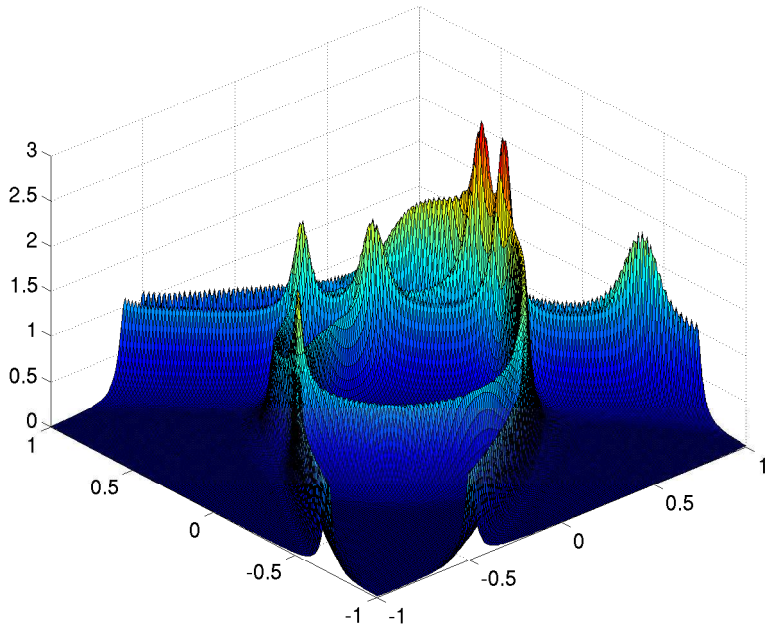


Table 6.10: Numerical Results for hard Runge problem

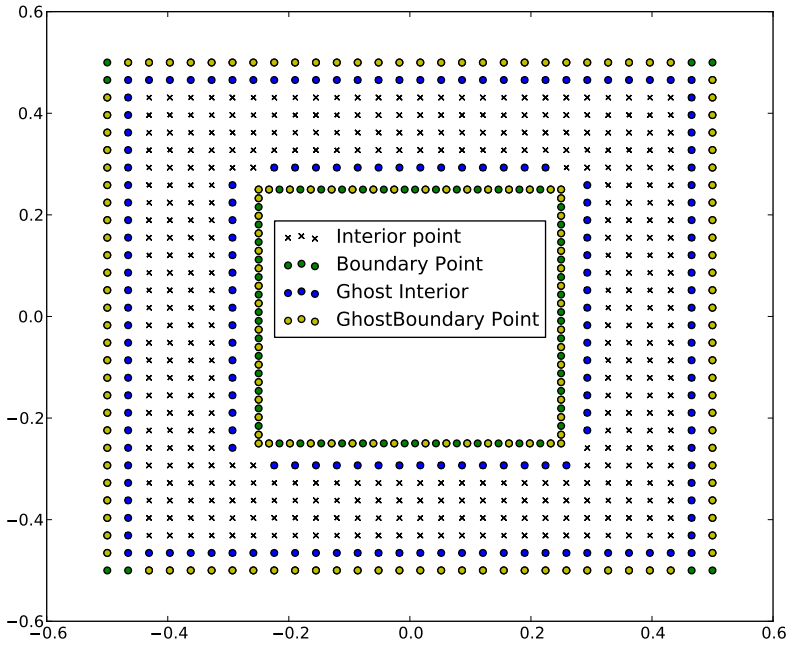
	25		49	
$\frac{1}{h}$	Error	Order	Error	Order
30	3.5e+02	-	3.6e+02	-
100	2.3e-01	6.1	3.5e-01	5.8
200	4.8e-02	4.7	9.7e-03	5.5
500	5.0e-03	4.0	2.6e-04	5.0

We present one final example for the biharmonic case, namely a more complicated domain as in Figure 6.8. We now have two Ghost layers corresponding to the two boundaries. Solving the standard biharmonic problem which we had as the first example on this geometry yielded an accuracy of 7 digits with $h = \frac{1}{200}$. We are restricted in software to square domains, since we do not have the code in place to compute the normal derivatives for arbitrary boundaries.

6.3 Summary

We thus have considered several numerical results that stand evidence to the power MSNFD in handling two hard classes of problems in the form of Exterior Laplace problems and biharmonic PDEs. As with the exterior problem, we are not attempting to specialize our solver to a particular PDE, but rather have a general purpose higher order solver. At this point we take up on the turn-around in accuracy that was observed with Table 6.2.1. The accuracy of the solution from a grid spacing of $\frac{1}{200}$ to $\frac{1}{500}$ *reduced* by two digits due to the numerical errors.

Figure 6.8: Domain with a hole for the biharmonic PDE solution



We attribute this to the high condition number of the biharmonic system. As an effort to alleviate this, one may attempt to rewrite the fourth order PDE as a system of first order PDEs. If done right, this would drop the condition number of the system to first order, although leading to a much larger tall-skinny system of equations. An initial attempt to lift this biharmonic equation to a square system led to a system that was 7 times larger. Since the memory requirements were prohibitive ($>$ 512GB) for the solver, single precision experiments were resorted to. While this partially alleviated the problem of turn around by postponing it, the condition number remained high. A correctly lifted system as per the FOLS idea (see [35]) would involve a Div-Curl system. In our initially lifting attempt, this was not understood, and so the additional homogenous curl condition on a gradient term was not incorporated and hence unsuccessful. The next chapter discusses this further, summarizes the key ideas presented in this thesis, and draws the straws for the future.

Chapter 7

Conclusions and Extensions

Through this thesis, a robust solution has been proposed for the problem of producing higher order finite difference weights on scattered grids. The problem with traditional finite difference weights was demonstrated, their shortcomings being non-uniqueness, non-convergence, and non-generalizability to scattered grids. Finite element based ideas have difficulty handling cases of variable coefficients and the order of these methods is dependent on having matching elements for exact boundary conditions. In many if not most situations, an approximate boundary condition is imposed and a nearby problem is solved. The MSNFD approach on the other hand, being a finite difference type approach handles the variable coefficient case effectively. In addition, the order of the method was shown to be high on various geometries and problems through sufficient numerical exper-

iments. Beginning with extensive numerical results for higher order MSN interpolation, theory and results were provided for the process of computing higher order MSNFD weights. The show-case application of our interpolation technique was a PDE Solver. In the absence of a proof of convergence of the global solution using MSNFD, a variety or second order problems were first considered, and results of convergence, condition number and discretization error were provided. Following these, we consider two extremely hard problems, namely, the exterior Laplace problem and the biharmonic problem. The former problem is very important in circuit design, magneto hydro dynamics and several other applications. The biharmonic problem is important for elasticity, diffusion and several other applications. While each of these cases deserve a special investigation, we put the MSNFD method to test using these hard problems to confirm its promise as a viable, and practical higher order FD method. It would be fair to say that the MSNFD method indeed withstands these tests and in fact goes beyond. At present, my research is directed towards solving the scattering problem using MSNFD approach. The problem is known to be extremely hard, and deserves specialized setup. In this section, we consider a brief overview of attempts and a summary of the current status. We also give a little more detail concerning our efforts towards lifting and resolving the turn around for the biharmonic problem. These clearly provide the road map for possible extensions.

7.1 Lifting the BiHarmonic Problem

7.1.1 Lifting a fourth order ODE

Consider a simple fourth order ODE of the form

$$u'''' + u = f,\ x \in [-1, 1] \tag{7.1}$$

$$u(-1) = g_{-1}, \quad u'(-1) = h_{-1}, \quad u(1) = g_1, \quad u'(1) = h_1, \tag{7.2}$$

where g and h are appropriate boundary functions.

Upon lifting, we rewrite the above equation as the system,

$$u = u_0 \tag{7.3}$$

$$u' = u_1 = u_0' \tag{7.4}$$

$$u'' = u_2 = u_1' \tag{7.5}$$

$$u''' = u_3 = u_2' \tag{7.6}$$

$$u_3' + u_0 = f. \tag{7.7}$$

The key difference here is that we are solving a system of first order equations. In order to construct a square system once again, we use the Ghost point idea for the ODE and the boundary conditions. For the auxiliary equations that relate the various lifted variables, we use all the specification points (same as grid points), so that the overall system is still square. Note that this is a self imposed

Figure 7.1: Condition Number of the fourth order ODE

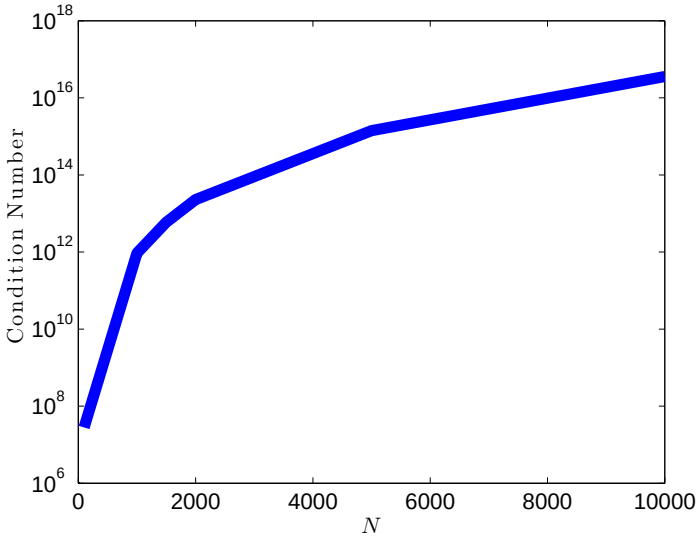


Figure 7.2: Error and residual of the fourth order ODE

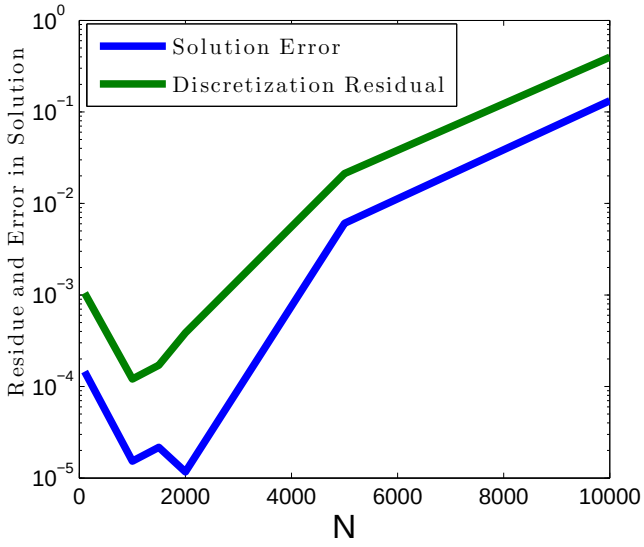


Figure 7.3: Condition Number of the lifted fourth order ODE

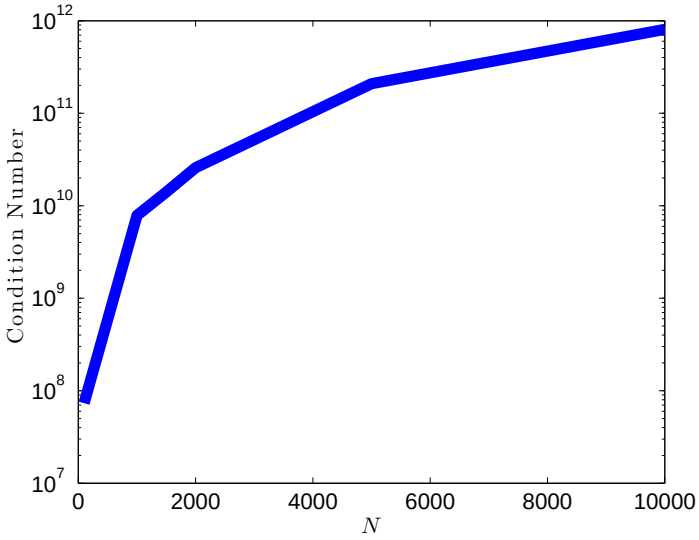
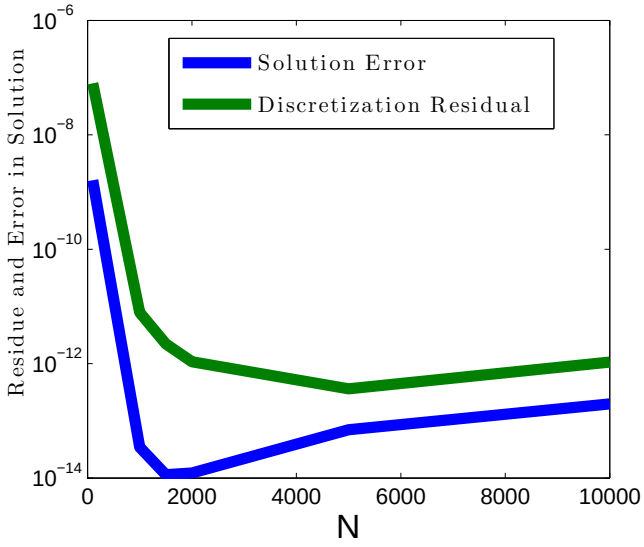


Figure 7.4: Error and residual of the lifted fourth order ODE



restriction. There is no strict need for us to solve only a square system, except that of computational ease and that of a unique solution. Figures 7.1 through 7.4 compare the solution error and condition number due to lifting. From these figures, we clearly see the advantage of lifting, in terms of condition number and the subsequent alleviation of the turn around in accuracy. While this proved simple enough in one dimension, things are not especially so in higher dimensions.

7.1.2 Lifting the two dimensional biharmonic PDEs

We now try to adopt a similar approach of lifting a biharmonic PDE using first order terms. The biharmonic PDE we consider is

$$\frac{\partial^4 u}{\partial x^4} + \frac{\partial^4 u}{\partial y^4} + u = f,\ (x, y) \in \Omega \tag{7.8}$$

$$u = g,\ (x, y) \in \partial\Omega \tag{7.9}$$

$$\nabla u.\hat{n} = h,\ (x, y) \in \partial\Omega, \tag{7.10}$$

where $\hat{n}$ is the unit normal to the boundary. We rewrite this equation using first order terms as follows.

$$u = u_0 \tag{7.11}$$

$$\mathbf{u}_1 = \nabla u = \begin{bmatrix} \frac{\partial u}{\partial x} \\ \\ \frac{\partial u}{\partial y} \end{bmatrix} = \begin{bmatrix} u_{11} \\ \\ u_{12} \end{bmatrix} \tag{7.12}$$

$$u_2 = \frac{\partial u_{11}}{\partial x} = \frac{\partial^2 u}{\partial x^2} u_3 = \frac{\partial u_{12}}{\partial y} = \frac{\partial^2 u}{\partial y^2} \tag{7.13}$$

$$u_4 = \frac{\partial u_2}{\partial x} = \frac{\partial^3 u}{\partial x^3} \tag{7.14}$$

$$u_5 = \frac{\partial u_3}{\partial y} = \frac{\partial^3 u}{\partial y^3} \tag{7.15}$$

$$\frac{\partial u_4}{\partial x} + \frac{\partial u_5}{\partial y} + u_0 = f. \tag{7.16}$$

This system is 7 times larger than the original system of equations. Due to the extremely large nature of the system, and considerable bandwidth, the memory requirements needed to carry out an LU factorization could not be met by any of the super computers we have access to. We thus needed an alternative way to verify our hypothesis. We therefore resorted to single precision arithmetic. With single precision, the accuracy of floating point numbers is about 9 digits. The tolerable condition number therefore is reduced a lot, and the turn around could be simulated much more easily. Table 7.1 below shows the error and condition number for the biharmonic PDE problem with $u = \frac{1}{1+x^2+y^2}$ as the solution, in single precision arithmetic. Table 7.2 provides these results after lifting the system as above. We see that, while the turn around has been postponed and the

maximum possible accuracy has been improved by 3 digits, the condition number is still to high. Even though the discretization error converges, the large condition number causes the solution to computed incorrectly. Thus, the turn around problem remains! An important take away from these experiments is that the condition number increase has not yet been alleviated as expected with lifting; we still observe N^4 order of scaling. We believe this is because the system has not been lifted correctly.

Table 7.1: Biharmonic error in Single Precision

$\frac{1}{h}$	Solution Error	Cond No	Discretization Err
30	3.2e-04	9.9e+05	2.7e-02
100	1.1e-01	8.1e+08	5.3e+00
200	1.1e+00	5.6e+09	7.2e+01

Table 7.2: Biharmonic error in Single Precision with Lifting

$\frac{1}{h}$	Solution Error	Cond No	Discretization Err
30	5.7e-04	7.8e+07	2.5e-04
100	1.5e-04	2.3e+10	1.7e-05
200	1.4e+00	1.0e+16	4.5e-06

Perhaps the problem with above lifted system is that linear dependencies are still present in the lifted system. For example, the gradient term $\mathbf{u}_1$ also has a zero curl,

$$\nabla \times \mathbf{u}_1 = 0. \tag{7.17}$$

This leads to an implicit *Div-Curl* system that needs to be resolved. Thus our focus is now Div-Curl systems. In fact, as we shall see, in any lifted procedure,

one would have to invariably employ a Div-Curl setup and hence, handling this problem is quite important.

7.2 The Exterior Helmholtz Problem

The exterior Helmholtz problem is primary to all scattering and inverse scattering problems. A fundamental difference between the exterior Laplace and Helmholtz problems is that the former continues to be definite, whereas the latter is indefinite. Also, the fundamental solution, also called the Green's function is smooth for the Laplacian, whereas for the Helmholtz problem, the fundamental solution is that Hankel function, whose real and imaginary parts are oscillatory Bessel functions. In the presence of a compactification, all these oscillations are compressed into a finite domain, leading to an accordion type compression. In addition, the maximum frequency in the compressed domain is infinite and so is the Nyquist rate to resolve it. However, one may consider the fact that the Green's function decays, and so for a given tolerance, one may be able to adequately sample it so that any aliasing cause is smaller than the tolerance. However, the Hankel function decays extremely slowly as $O(\sqrt{r})$. Thus, one would need to be at $r = 10^6$ for 3 digits relative amplitude. At such a distance, the compressed frequency would be very large and adequate sampling would be impossible.

All hope is not lost though. Since we know that the Green's function is oscillatory, we can assume the solution to contain these oscillations, and instead solve the PDE for the much smoother far-field pattern and other components. The exterior Helmholtz problem takes the form below.

$$\nabla^2 u + k^2 u = 0,\ (x, y) \in \mathbf{R}^2 \backslash \Omega \tag{7.18}$$

$$u = u^i + u^s = 0,\ (x, y) \in \partial\Omega, \tag{7.19}$$

where u^i is the incident wave component, u^s is the scattered wave component. In addition, we have Sommerfiled Radiation condition to make sure we retrieve the out-going scattered wave solution instead of the standing wave solution, as below.

$$\lim_{r\to\infty} \sqrt{r} \left(\frac{\partial u}{\partial r} + iku \right) = 0. \tag{7.20}$$

We solve the exterior Helmholtz equation for the scattered wave, given the incident wave. The boundary condition is to enforce zero total radiation on the surface of the conductor. While setting up and solving a polar Helmholtz solver, assuming that the solution takes the form

$$u = \frac{e^{ikr}}{\sqrt{r}} \hat{u}, \tag{7.21}$$

subject to the compactification $r \to \tan r$.

We now consider the modification of the Polar Helmholtz PDE under the above transformations. The polar Helmholtz equation is given by

$$u_{rr} + \frac{1}{r}u_r + u_{\theta\theta} + k^2 u = 0. \tag{7.22}$$

If $u = \frac{e^{ikr}}{\sqrt{r}}\hat{u}$, the above equation becomes

$$\hat{u}_{rr} + 2ik\hat{u}_r + \frac{1}{r^2}\hat{u}_{\theta\theta} + \frac{\hat{u}}{4r^2} = 0. \tag{7.23}$$

We compactify the domain here using the transformation $r \to \tan r$. There for the infinite exterior maps to the region $\left\{[0, \frac{\pi}{2}] \times [0, 2\pi]\right\} \setminus \Omega$. Note that the exterior boundary is naturally skewed, with the height being 6 times the length due to our transformation. We could handle this either through a scaled transformation, or just different rate of sampling along one axis. We tried the latter approach in our experiments. Under the inverse tangent compactification, the equation above further changes as below,

$$\cos^2 r\hat{u}_{rr} - 2\cos r \sin r\hat{u}_r + 2ik\sin^2 r\hat{u}_r + \hat{u}_{\theta\theta} + \frac{\hat{u}}{4} = 0 \tag{7.24}$$

in the compactified domain.Another important point to note is that the Sommerfield radiation conditions also naturally transform to the exact conditions given by,

$$\sin r\cos r\hat{u}_r - \frac{\hat{u}}{2} = 0, \quad |r| = \frac{\pi}{2}. \tag{7.25}$$

By rewriting the scattering equation in the compactified notation, *after* capturing the oscillations explicitly, we are assured that there is only a finite Nyquist frequency to be met. The solution for the scattering is known to take the form (see 3.63 in [10]),

$$u = \frac{e^{ikr}}{\sqrt{r}} \left\{ u_\infty(\hat{r}) + O\left(\frac{1}{r}\right) \right\}, \; r \to \infty. \tag{7.26}$$

By capturing all the oscillations and the slowly decaying components in the solution, we are left with a much smoother part, which actually is accelerated in its decay by the compactification. However, the problem is still singular and highly ill-conditioned. The severe ill-conditioning of the system, combined with the high Nyquist rate required has made this problem quite formidable. The obvious way out is once again to lift and solve a system of equations including a Div-Curl system. Thus once again, we see that the Div-Curl system plays a crucial role in solving lifted ill-conditioned systems and hence deserves further investigation. In addition, the elusive proof of convergence of the MSNFD solution is also an important work for the future.

7.3 Concluding Remarks

We have thus shown that the MSNFD method and hence the MSN idea is an important one. The potential applications of such a fundamental idea are

numerous, ranging from Image processing, to PDE solutions. We have showcased the most important perhaps easy to reach application of the MSN interpolation idea, through a PDE solver. The solver in three dimensions is of great industrial significance and of substantial commercial value.

The biharmonic problem in particular has been dealt well by MSNFD. Another difficult problem which I think has been solved through MSNFD is the exterior Laplace problems. I hope that this benefits the semiconductor industry for circuit parameter calculations and Magneto Hydrodynamic problems. Other industrial applications may now be considered and solved using MSNFD.

However, the hardest of the problems, namely, the biharmonic and the exterior Helmholtz problems need further work to be dealt with completely. Our experience with these hardest problems also brings out the importance of first order Div-Curl systems for handling the large condition number. It has been my fortune and privilege to work on this idea during the course of my PhD. Besides learning so much, it has provided me the great opportunity of contributing something useful for all! The battle won perhaps, but the war goes on!

Bibliography

[1] M. Abramowitz and I. Stegun. *Handbook of mathematical functions with formulas, graphs, and mathematical tables*. Number v. 55, no. 1972 in Applied mathematics series. U.S. Govt. Print. Off., 1964.

[2] I. Altas, J. Dym, M. M. Gupta, and R. P. Manohar. Multigrid solution of automatically generated high-order discretizations for the biharmonic equation. *SIAM Journal on Scientific Computing*, 19(5):1575–1585, 1998.

[3] W. Bangerth, R. Hartmann, and G. Kanschat. deal.II – a general purpose object oriented finite element library. *ACM Trans. Math. Softw.*, 33(4):24/1–24/27, 2007.

[4] W. L. Briggs, V. E. Henson, and S. F. McCormick. *A multigrid tutorial (2nd ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[5] S. Chandrasekaran, P. Dewilde, M. Gu, W. Lyons, and T. Pals. A fast solver for hss representations via sparse matrices. *SIAM Journal on Matrix Analysis and Applications*, 29(1):67–81, 2008.

[6] S. Chandrasekaran, K. R. Jayaraman, M. Gu, H. N. Mhaskar, and J. Moffitt. Msnfd : A higher order finite difference method for solving elliptic pdes on scattered points. *Technical Report*, 2011.

[7] S. Chandrasekaran, K. R. Jayaraman, M. Gu, H. N. Mhaskar, and J. Mofftt. Higher order numerical discretization methods with sobolev norm minimization. *Procedia CS*, 4:206–215, 2011.

[8] S. Chandrasekaran, K. R. Jayaraman, J. Moffitt, H. N. Mhaskar, and S. Pauli. Minimum sobolev norm schemes and applications in image processing. volume 7535, page 753507. SPIE, 2010.

[9] S. Chandrasekaran and H. N. Mhaskar. A construction of linear bounded interpolatory operators on the torus. *ArXiv e-prints*, Nov. 2010.

[10] D. Colton and R. Kress. *Inverse acoustic and electromagnetic scattering theory*. Applied mathematical sciences. Springer, 1998.

[11] P. J. Davis. Interpolation and approximation. 1963.

[12] T. Davis. *Direct methods for sparse linear systems*. Fundamentals of algorithms. Society for Industrial and Applied Mathematics, 2006.

[13] T. A. Davis. Suitesparseqr: Algorithm 9xx: Suitesparseqr, a multifrontal multithreaded sparse qr factorization package. *ACM Trans. Math. Softw.*

[14] T. A. Davis. Algorithm 832: Umfpack v4.3—an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw.*, 30:196–199, June 2004.

[15] C. N. Dorny. Finite-difference approximation of the exterior problem for poisson's equation. *Journal of Computational Physics*, 2(4):363 – 380, 1968.

[16] J. Epperson. *An introduction to numerical methods and analysis*. Wiley-Interscience, 2007.

[17] J. F. Epperson. On the runge example. *Am. Math. Monthly*, 94:329–341, April 1987.

[18] J. Gilbert, G. Miller, and S.-H. Teng. Geometric mesh partitioning: implementation and experiments. In *Parallel Processing Symposium, 1995. Proceedings., 9th International*, pages 418 –427, apr 1995.

[19] G. Golub and C. Loan. *Matrix computations*. Johns Hopkins studies in the mathematical sciences. Johns Hopkins University Press, 1996.

[20] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations, 1987.

[21] J. B. Greer, A. L. Bertozzi, and G. Sapiro. Fourth order partial differential equations on general geometries. *Journal of Computational Physics*, 216(1):216 – 246, 2006.

[22] P. Hough and S. A. Vavasis. Complete orthogonal decomposition for weighted least squares. *SIAM J. Matrix Anal. Appl*, 18:369–392, 1995.

[23] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001–.

[24] G. Karypis and V. Kumar. Metis - unstructured graph partitioning and sparse matrix ordering system, version 2.0. Technical report, 1995.

[25] C. S. Kenney, A. J. Laub, and M. S. Reese. Statistical condition estimation for linear systems. *Siam Journal on Scientific Computing*, 19, 1998.

[26] X. S. Li. An overview of superlu: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.*, 31:302–325, September 2005.

[27] T. Lin, Y. Lin, W.-W. Sun, and Z. Wang. Immersed finite element methods for 4th order differential equations. *Journal of Computational and Applied Mathematics*, 235(13):3953 – 3964, 2011. Engineering and Computational Mathematics: A Special Issue of the International Conference on Engineering and Computational Mathematics, 27-29 May 2009.

[28] Y. Lung-An. Numerical methods for exterior problems. 2006.

[29] T. Mathworks. Matlab, 2010a.

[30] H. Mhaskar and D. Pai. *Fundamentals of approximation theory*. CRC Press, 2000.

[31] F. Pérez and B. E. Granger. IPython: a System for Interactive Scientific Computing. *Comput. Sci. Eng.*, 9(3):21–29, May 2007.

[32] P. Ramachandran and G. Varoquaux. Mayavi: 3D Visualization of Scientific Data. *Computing in Science & Engineering*, 13(2):40–51, 2011.

[33] M. Renardy and R. Rogers. *An introduction to partial differential equations*. Texts in applied mathematics. Springer, 2004.

[34] W. Rudin. *Real and complex analysis*. Mathematics series. McGraw-Hill, 1987.

[35] J. Strain. Locally-corrected spectral methods and overdetermined elliptic systems. *Journal of Computational Physics*, 224(2):1243 – 1254, 2007.

[36] G. Strang. *Computational science and engineering*. Wellesley-Cambridge Press, 2007.

[37] J. Szabados and V. P. Interpolation of functions. 1990.

[38] S. A. Vavasis. Stable numerical algorithms for equilibrium systems. *SIAM J. Matrix Anal. Appl*, 15:1108–1131, 1992.

[39] T. Vejchodsky, P. Soln, and M. Ztka. Modular hp-fem system hermes and its application to maxwell's equations. *Mathematics and Computers in Simulation*, 76(1-3):223 – 228, 2007. Mathematical Modelling and Computational Methods in Applied Sciences and Engineering.

[40] G. Wahba. *Spline models for observational data*. CBMS-NSF regional conference series in applied mathematics. Society for Industrial and Applied Mathematics, 1990.

[41] W. Wang, J. Guzmán, and C. Shu. The multiscale discontinuous galerkin method for solving a class of second order elliptic problems with rough coefficients. *International Journal of Numerical Analysis and Modeling*, v8:28–47, 2011.

[42] G. B. Wright and B. Fornberg. Scattered node compact finite difference-type formulas generated from radial basis functions. *J. Comput. Phys.*, 212:99–123, February 2006.

[43] A. Zemanian. A finite-difference procedure for the exterior problem inherent in capacitance computations for vlsi interconnections. *Electron Devices, IEEE Transactions on*, 35(7):985 –992, jul 1988.

Appendices

Appendix A

The MSN Interpolation Kernel

The idea of an MSN Kernel was introduced in the MSN interpolation chapter, where equation 2.19 could be rewritten as,

$$p_M(x) = V(x)D_s^{-2}V^T(VD_s^{-2}V^T)^{-1}\mathbf{f} \tag{A.1}$$

$$= \sum_{i=0}^{N-1} K(x, x_i)\mathbf{f}_i. \tag{A.2}$$

In turn the kernel can written as a summation,

$$K(\theta_i, \theta_j) = 1 + \sum_{m=1}^{M-1} \frac{\cos m\theta_i \cos m\theta_j}{(m)^{2s}},\ \theta_i = \cos^{-1} x_i \tag{A.3}$$

$$= 1 + 0.5 \sum_{m=1}^{M-1} \frac{\cos\left(0.5m(\theta_i + \theta_j)\right) + \cos\left(0.5m|\theta_i - \theta_j|\right)}{(m)^{2s}} \tag{A.4}$$

Summations of the form

$$S_r(t) = \sum_{m=1}^{\infty} \frac{\cos mt}{m^r} \tag{A.5}$$

occur as representation of the Clausen's Integral (see [1] page 1005). Thus, our kernel for infinite order interpolants, for integer values of s can be written in closed form as polynomials of order $2s$. For example, with $s = 2$, we can write

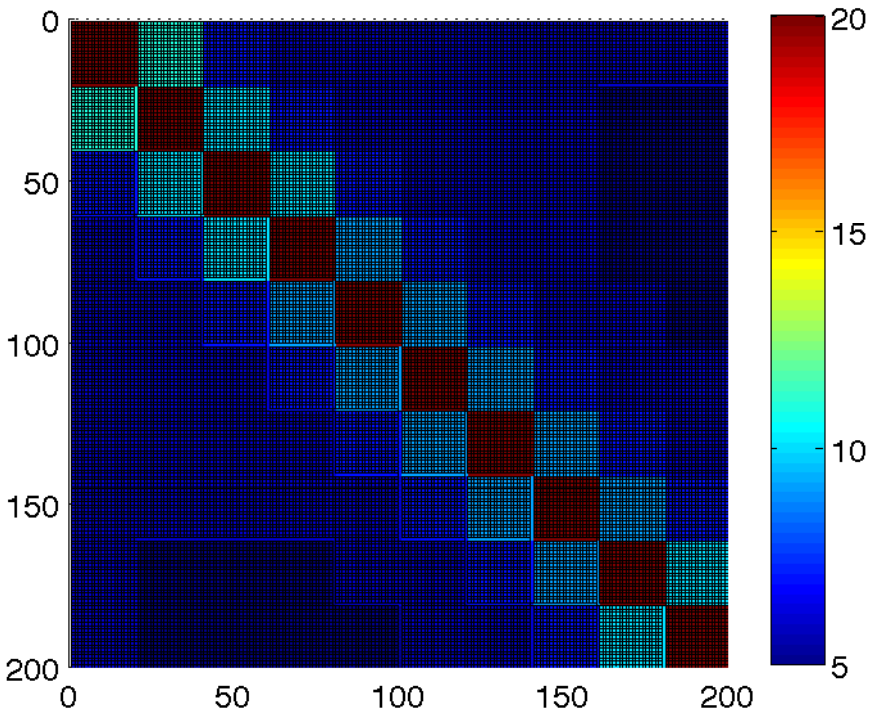
the infinite order kernel matrix entries as the summation

$$K_{i,j} = 1 + 0.5\left\{S_{2s}(0.5(\theta_i+\theta_j)) + S_{2s}(0.5|\theta_i-\theta_j|)\right\} \tag{A.6}$$

$$= 1 + 0.5\left\{\frac{\pi^4}{90} - \frac{\pi^2(\theta_i+\theta_j)^2}{48} + \frac{\pi(\theta_i+\theta_j)^3}{96} - \frac{(\theta_i+\theta_j)^4}{768}\right\}$$
$$+0.5\left\{\frac{\pi^4}{90} - \frac{\pi^2|\theta_i-\theta_j|^2}{48} + \frac{\pi|\theta_i-\theta_j|^3}{96} - \frac{|\theta_i-\theta_j|^4}{768}\right\}. \tag{A.7}$$

The odd powers in the second summation term $S_{2s}(0.5|\theta_i - \theta_j|)$ behave like the $|x|$ function about the diagonal $\theta_i = \theta_j$. Away from the diagonal, these terms are extremely smooth. This leads to a dense diagonal and low-rank off diagonal structure for the kernel matrix K. An example indicating block ranks is shown in Figure **??** for a 200 point kernel. Using this fact, one can construct fast interpolation algorithms as well as techniques to solve PDEs using the idea of Fast Multipole Methods.

Figure A.1: Block rank structure, 20×20 blocks



In the paper [8], results of a HSS based implementation for MSN interpolation is discussed. However, the algorithm implemented is just an LU factorization

based solver, which as we know is extremely unstable given the ill-conditioned nature of such systems. One needs a much more sophisticated implementation of a WLS solver using CODA through HSS or FMM for numerical stability. The paper referenced above also discuss an interesting and powerful extension of MSN interpolation to an approximation scheme by a regularization. This approximation scheme called the Generalized MSN (GMSN) has been applied to perform a basic image segmentation as discussed in the paper.

Appendix B

Software Details

B.1 Code organization

All functions of the PDE solver, starting from geometry generation, to the sparse solve are exported by a python module file, called the 'core' file. For each kind of PDE we'd like to solve, we thus have a wrapper that calls the core file functions, and set the problem up. The stages for setting up and solving a PDE system are as follows. The problem is defined by setting up the solution (if known) and generating the appropriate RHS function, symbolically. At this time, the coefficients of the PDE if analytically known may be specified, although not necessary. The coefficients may be set to unity and the actual coefficients be employed only at the solving stage. The next step is the generation of interior and boundary points corresponding to the problem's geometry. Several geometries are available in the core, and the module is easily extensible to add new geometry. The geometry generation is parametric and the boundary points are ordered to form closed polygon. The core also supplies a parallel point-in-polygon test for help with geometry generation. Once the geometry is generated, the assembly routine can be called with the unity coefficients, to generate the weights and save them. A simple flag is used to enable the weight generation. Otherwise, to assemble a sparse matrix and solve the PDE, the various function evaluations, corresponding to the coefficients of the PDE and the RHS are carried out. The reference solution is also evaluated to compute the residue. The evaluation is carried out in parallel on as many ipengine instances as available. Once all the requisite evaluations are in place, the solver function is called. The function returns the computed solution, the condition number estimate, the residue as well as the time taken for the various stages. Additional diagnostics may be added with ease as well. The wrapper then computes the solution error and saves the results into a data file.

The coefficients are compressed as saved as a gzip file, due to their large size. Pickling as provided by python is used to marshal a variety of data formats and structures.

B.2 Core python functions

In this section, we publish the various functions in the MSNFD's core module. We make extensive use of Numpy, Scipy and iPython's MEC interfaces. The python scripts that follow in order are the second order core modules, the exterior Laplace test wrapper and the biharmonic wrapper. These capture the essential functionality and method for extending the implementation to other problems, and higher dimensions perhaps. The core module contains extensions to a Hyperbolic solver, that tackles a one dimensional hyperbolic wave equation as a fully implicit two dimensional problem. The core also contains an implementation of a multi-grid type of approach, although it is not investigated further. All python modules and wrappers are available in our website http://scg.ece.ucsb.edu/software.html.

Listings

Listing B.1: Second Order Core Function

```
import numpy as np
import time as tm
import scipy as sp
import scipy.sparse as sps
import scipy.sparse.linalg as spla
import numpy.linalg as la
import sympy as sym
from IPython.kernel import client
from ctypes import *
#mkl = cdll.LoadLibrary("clapack.so")
mkl = cdll.LoadLibrary("/Library/Frameworks/Python.framework/Versions
    /2.7/lib/python2.7/site-packages/scipy/lib/lapack/clapack.so")
dgesvd = mkl.dgesvd_
import pylab as pl
import cPickle as pkl
import gzip as gz

###NEW SVD using DGESVD
def ctypes_svd(B, jobU, jobVt):
  #Row major to colun major conversion
  m = B.shape[0]
  n = B.shape[1]
  A = np.ravel(B, order='F')
  if jobU == 'N':
    U = np.array('')
  else:
    U = np.zeros([m,m], order = 'F')

  S = np.zeros([1, min(m,n)], order = 'F')
  if jobVt == 'N':
```

```
    VT = np.array('')
  else:
    VT = np.zeros([n,n], order = 'F')

  INFO = c_int(0)
  LWORK = -1
  WORK = (c_double*1)()

  dgesvd( byref(c_char(jobU)), byref(c_char(jobVt)), byref(c_int(m)),
       byref(c_int(n)), A.ctypes.data_as(c_void_p), byref(c_int(m)),
      S.ctypes.data_as(c_void_p), U.ctypes.data_as(c_void_p), byref(
      c_int(m)), VT.ctypes.data_as(c_void_p), byref(c_int(n)),WORK,
      byref(c_int(LWORK)), byref(INFO))

  LWORK =int(WORK[0])

  WORK = (c_double * LWORK)()

  dgesvd( byref(c_char(jobU)), byref(c_char(jobVt)), byref(c_int(m)),
       byref(c_int(n)), A.ctypes.data_as(c_void_p), byref(c_int(m)),
      S.ctypes.data_as(c_void_p), U.ctypes.data_as(c_void_p), byref(
      c_int(m)), VT.ctypes.data_as(c_void_p), byref(c_int(n)), WORK,
      byref(c_int(LWORK)), byref(INFO))

  return np.mat(U).copy(), S.copy(), np.mat(VT).copy()

#Outermost function definition, CODA_WLS
#Accepts the ill-scaled matrix D, and the tall-skinny matrix
#A. It returns the three factors, q, r and v, such that
#DA = qrvT (basically from an svd), but the qr are derieved from a
#more 'manageable' matrix.
def CODA_WLS(D, A, eta):
    Av,v = refine_v(D*A, eta);
    q,r = la.qr(Av);
    return q,r,v

def refine_v(A, eta):
    #u,s,vh = la.svd(A, full_matrices=False)
    u,s,vh = ctypes_svd(A, 'N','A')
    v = vh.H
    Av = A*v
    Ao = np.matrix('')
    vo = np.matrix('')
    iter = 0
    while True:
        A1, A2, v1, v2, s, term = compute_refine(Av, v, eta, s)
#         print iter
```

```
            iter = iter + 1
            if Ao.nbytes == 0:
                Ao = np.bmat('A1')
            elif A1.nbytes != 0:
                Ao = np.bmat('Ao,A1')
#           Ao = [Ao A1]
            if vo.nbytes == 0:
                vo = np.bmat('v1')
            elif v1.nbytes != 0:
                vo = np.bmat('vo,v1')
#           vo = [vo v1]
            Av = A2
            v = v2
            if term == 1:
                break
        if Ao.nbytes == 0:
            Ao = np.bmat('A2')
        elif A2.nbytes != 0:
            Ao = np.bmat('Ao,A2')
#            Ao = [Ao A2]
        if vo.nbytes == 0:
            vo = np.bmat('v2')
        elif v2.nbytes != 0:
            vo = np.bmat('vo,v2')
#            vo = [vo v2]
        return Ao, vo

def compute_refine(Av, v, eta, s):
    s = s.flatten()
    tol = s[0]/eta
    k = np.nonzero(s<tol)
    if np.any(k) == False:
        A1 = Av
        A2 = np.matrix('')
        v1 = v
        v2 = np.matrix('')
        term = 1
        return A1, A2, v1, v2, s, term

    k = k[0][0]
    A1 = np.matrix(Av[:,0:k])
    A2 = np.matrix(Av[:,k:])
    v1 = np.matrix(v[:,0:k])
    v2 = np.matrix(v[:,k:])
    if np.any(A2) == False:
        term = 1
    else:
```

```
        term = 0
    #u,s,vh = la.svd(A2, full_matrices=False)
    u,s,vh = ctypes_svd(A, 'N','A')
    v = vh.H
    A2 = A2*v
    v2 = v2*v
    return A1, A2, v1, v2, s, term

#Solves the weighted LS system DAx ~ Db
def WLS_Solve(D, A, b):
    #Sort D
    d = np.diag(D)
    srt_ind = np.argsort(d)
    if srt_ind.shape[0] == 1:
        srt_ind = np.fliplr(srt_ind)
    else:
        srt_ind = np.flipud(srt_ind)
    Ds = D[srt_ind,:]
    q,r,v = CODA_WLS(Ds, A, 10.0**2.0)
    #DAx = qrv.Tx = Db
    #v.T*x = r\(q.T*D*b)
    #x = v*(r\(q.T*D*b))
    Db = Ds*b
    z = q.T*Db
    w = la.solve(r,z)
    x = v*w
    return x

def test_CODA_WLS_Unit(s):
    M = 6400
    N = 400
    P  = 10
    err = np.zeros([s.size,1])
    err_ref = np.zeros([s.size,1])
    for i in np.arange(0,s.size,1):
        D = np.diag(np.arange(1,M+1,1.)**(-s[i]))
        A = np.mat(np.random.rand(M,N))
        xo = np.mat(np.random.rand(N,P))
        b = A*xo
        x = WLS_Solve(D, A, b)
        err[i] = np.max(np.abs(x-xo))/np.max(np.abs(xo))
        x_ref = la.lstsq(D*A, D*b)[0]
        err_ref[i] = np.max(np.abs(x_ref-xo))/np.max(np.abs(xo))

    return err,err_ref, s

#Scales 2D coordinates to the range -1,1
```

```
def scale_2D(x,y,ax,bx,ay,by):
    xs = 2.0*(x-ax)/(bx-ax) - 1
    ys = 2.0*(y-ay)/(by-ay) - 1
    return xs,ys

#Computes the 2D Vandermonde Matrix
def vandermonde2D(x,y,Mx,My):
    V = np.zeros([x.size,Mx*My])
    for i in range(x.size):
        V[i,:] = np.kron(np.cos(np.arccos(x[i])*range(Mx)),np.cos(np.
            arccos(y[i])*range(My)))
    return np.mat(V)

#Computes the meshnorm corresponding to x,y
def meshnorm2D(xa,ya):
    x = np.arccos(xa)
    y = np.arccos(ya)
    #We use Numpy's broadcast capability!! Awesome Numpy, way to go.
    #expecting column arrays/vectors
    if x.shape[1] != 1 or y.shape[1] != 1:
        print 'Error in dimension'
    dx = x-x.flatten()
    dy = y-y.flatten()
    d = (np.array(dx)**2.0 + np.array(dy)**2.0)**(0.5)
    dmin = np.min(np.min(d[d>1e-6]))
    mnorm = int(round(3.*np.pi/dmin))
    return mnorm

#Computes the MSN Weight obtained by solving the MSN-LS system with
    the given
#s, that 'interpolates' the function in b, at xk,yk using the values
    at x,y
def MSN2DWt(x,y,ax,bx,ay,by,s,b, Mx, My):
    #Using the same polynomial lengths - this may be wasteful. Need
        to refine this.
    V = vandermonde2D(x, y, Mx, My)
    #Set up the diagonal weights
    D = np.zeros([Mx*My,Mx*My])
    p=0
    for m in range(Mx):
        for n in range(My):
            D[p,p] = (1+m**2+n**2)**(-0.5*s)
            p = p + 1
    xo = np.mat(np.random.rand(V.shape[0], 10))
    bo = V.T*xo
    b1 = np.bmat('b, bo')
    #Get weights using the CODA WLS Solver
```

```
    w = WLS_Solve(D, V.T, b1)
    x_WLS = w[:,b.shape[1]:]
    errLS = np.max(np.abs(x_WLS-xo))/np.max(np.abs(xo))
    return w, errLS

#Computes parial derivatives of the Vandermonde matrix upto 4th order
    and evaluates it at points
#x,y, order of polynomials is Mx, My; assumes x, y are scaled and in
    range [-1,1]
def computeVanderFunctionals2D(xc,yc,Mx,My):
    N = xc.size
    if N != 1:
        print 'Only singlepoint computation is supported as of now
            for functionals of V2'
        return -1

    #Compute the degree 0 Vandermonde
    Vkx = np.mat(np.cos(np.arccos(xc)*range(Mx)))
    Vky = np.mat(np.cos(np.arccos(yc)*range(My)))

    V1k = np.zeros([N, Mx]); V2k = np.zeros([N, Mx]);
    V3k = np.zeros([N, Mx]); V4k = np.zeros([N, Mx]);

    V1k[:,0] = 0;   V1k[:,1] = 1;
    V2k[:,0] = 0;   V2k[:,1] = 0;
    V3k[:,0] = 0;   V3k[:,1] = 0;
    V4k[:,0] = 0;   V4k[:,1] = 0;
    for m in np.arange(1,Mx-1,1):
        V1k[:, m+1] = (2*xc*V1k[:, m] - V1k[:, m-1] + 2*Vkx[:, m])
        V2k[:, m+1] = (2*xc*V2k[:, m] - V2k[:, m-1] + 4*V1k[:, m])
        V3k[:, m+1] = (2*xc*V3k[:, m] - V3k[:, m-1] + 6*V2k[:, m])
        V4k[:, m+1] = (2*xc*V4k[:, m] - V4k[:, m-1] + 8*V3k[:, m])

    Vxxxx = np.kron(V4k, Vky);    Vxxx = np.kron(V3k, Vky);
    Vxx = np.kron(V2k, Vky);     Vx = np.kron(V1k, Vky);

    V1kx = V1k;   V2kx = V2k;
    V3kx = V3k;   V4kx = V4k;

    V1k = np.zeros([N, My]); V2k = np.zeros([N, My]);
    V3k = np.zeros([N, My]); V4k = np.zeros([N, My]);

    V1k[:,0] = 0;   V1k[:,1] = 1;
    V2k[:,0] = 0;   V2k[:,1] = 0;
    V3k[:,0] = 0;   V3k[:,1] = 0;
    V4k[:,0] = 0;   V4k[:,1] = 0;
    for m in np.arange(1,My-1,1):
```

```
            V1k[:, m+1] = (2*yc*V1k[:, m] - V1k[:, m-1] + 2*Vky[:, m])
            V2k[:, m+1] = (2*yc*V2k[:, m] - V2k[:, m-1] + 4*V1k[:, m])
            V3k[:, m+1] = (2*yc*V3k[:, m] - V3k[:, m-1] + 6*V2k[:, m])
            V4k[:, m+1] = (2*yc*V4k[:, m] - V4k[:, m-1] + 8*V3k[:, m])

        Vyyyy = np.kron(Vkx, V4k);    Vyyy = np.kron(Vkx, V3k);
        Vyy = np.kron(Vkx, V2k);    Vy = np.kron(Vkx, V1k);

        V1ky = V1k;     V2ky = V2k;
        V3ky = V3k;     V4ky = V4k;

        V1xy = np.kron(V1kx, V1ky);     V2xy = np.kron(V2kx, V2ky);
        V3xy = np.kron(V3kx, V3ky);     V4xy = np.kron(V4kx, V4ky);
        Vk = np.kron(Vkx, Vky)    ;
        return Vxx, Vx, Vyy, Vy, V1xy, Vk, Vxxxx, Vyyyy, V2xy

#The weight generation function for computing the MSN-FD weights
    corresponding to
#various differential operators
def compute_MSN_weight(x,y,xk,yk,ax,bx,ay,by,s,a11,a12,a21,a22,a11_x,
    a12_x,a21_y,a22_y):
    #scale the points to th range
    xs,ys = scale_2D(x,y,ax,bx,ay,by)
    xks,yks = scale_2D(xk,yk,ax,bx,ay,by)
#     xs = x; ys = y;xks = xk; yks = yk;
    mnorm = meshnorm2D(xs, ys); Mx = mnorm; My = mnorm;
    #Compute the functionals at xc,yc
    Vxx, Vx, Vyy, Vy, V1xy, Vk, Vxxxx, Vyyyy, V2xy =
        computeVanderFunctionals2D(xks,yks,Mx,My)
    lx = 0.5*(bx-ax); ly = 0.5*(by-ay);
    VxxT = (lx**-2.)*Vxx.T; VyyT = (ly**-2.)*Vyy.T; V1xyT = (lx*ly)
        **(-1.)*V1xy.T;
    VxT = (lx**-1.)*Vx.T;VyT = (ly**-1)*Vy.T;VkT = Vk.T;
    rhs = np.bmat('VxxT, VxT, VyyT, VyT, V1xyT,VkT')
    w, errLS = MSN2DWt(xs,ys,ax,bx,ay,by,s,rhs, Mx, My)
    cxx = w[:,0]
    cx = w[:,1]
    cyy = w[:,2]
    cy = w[:,3]
    cxy = w[:,4]
    c = w[:,5]
    t11 = a11*cxx + a11_x*cx
    t12 = a12*cxy + a12_x*cy
    t21 = a21*cxy + a21_y*cx
    t22 = a22*cyy + a22_y*cy
    c2 = t11+t12+t21+t22
    return c2,cxx,cyy,cx,cy,c, errLS
```

```
def get_MSN_weights_Unit(x,y,xo,yo,ax,bx,ay,by,s,a11,a12,a21,a22,
    a11_x,a12_x,a21_y,a22_y):
    #Evaluate the functionals d/dx^2, d/dx, d/dy^2, d/dy at xo,yo
        using x,y
    #Try the exponential function, this is the easiest!
    c2mat = np.zeros([xo.size, x.size])
    cxxmat = np.zeros([xo.size, x.size])
    cxmat = np.zeros([xo.size, x.size])
    cyymat = np.zeros([xo.size, x.size])
    cymat = np.zeros([xo.size, x.size])
    cimat = np.zeros([xo.size, x.size])
    errLSi = np.zeros(xo.shape)

    for i in range(xo.size):
        c2,cxx,cyy,cx,cy,c, errLS = compute_MSN_weight(x,y,xo[i],yo[i
            ],ax,bx,ay,by,s,a11[i][0],a12[i][0],a21[i][0],a22[i][0],
            a11_x[i][0],a12_x[i][0],a21_y[i][0],a22_y[i][0])
        c2mat[i][:] = c2.T;
        cxxmat[i][:] = cxx.T; cxmat[i][:] = cx.T;
        cyymat[i][:] = cyy.T; cymat[i][:] = cy.T;
        cimat[i][:] = c.T;
        errLSi[i] = errLS

    return np.mat(c2mat), np.mat(cxxmat), np.mat(cxmat), np.mat(
        cyymat), np.mat(cymat), np.mat(cimat), errLSi

def get_MSNWts_Parallel(x,y,xo,yo,ax,bx,ay,by,s,a11,a12,a21,a22,a11_x
    ,a12_x,a21_y,a22_y):
    #Now the parallel part!
    #import all libraries in the engines
    mec = client.MultiEngineClient()
    mec.execute('import numpy as np')
    mec.execute('import scipy as sp')
    mec.execute('import numpy.linalg as la')
    mec.execute('from scipy import sparse')
    mec.execute('import sympy as sym')
    mec.execute('from ctypes import *')
    mec.execute('mkl = cdll.LoadLibrary("clapack.so")')
    mec.execute('dgesvd = mkl.dgesvd_')

    #push asll functions to the engines
    mec.push_function(dict(WLS_Solve=WLS_Solve))
    mec.push_function(dict(compute_refine=compute_refine))
    mec.push_function(dict(refine_v=refine_v))
    mec.push_function(dict(CODA_WLS=CODA_WLS))
    mec.push_function(dict(scale_2D=scale_2D))
```

```
    mec.push_function(dict(vandermonde2D=vandermonde2D))
    mec.push_function(dict(meshnorm2D=meshnorm2D))
    mec.push_function(dict(MSN2DWt=MSN2DWt))
    mec.push_function(dict(ctypes_svd=ctypes_svd))
    mec.push_function(dict(computeVanderFunctionals2D=
        computeVanderFunctionals2D))
    mec.push_function(dict(compute_MSN_weight=compute_MSN_weight))
    mec.push_function(dict(get_MSN_weights_Unit=get_MSN_weights_Unit)
        )
    mec.push(dict(x=x))
    mec.push(dict(y=y))
    mec.push(dict(ax=ax))
    mec.push(dict(bx=bx))
    mec.push(dict(ay=ay))
    mec.push(dict(by=by))
    mec.push(dict(s=s))

    mec.scatter('xo',xo)
    mec.scatter('yo',yo)
    mec.scatter('a11',a11)
    mec.scatter('a12',a12)
    mec.scatter('a21',a21)
    mec.scatter('a22',a22)
    mec.scatter('a11_x',a11_x)
    mec.scatter('a12_x',a12_x)
    mec.scatter('a21_y',a21_y)
    mec.scatter('a22_y',a22_y)

    mec.execute('c2mat,cxxmat,cxmat,cyymat,cymat,cimat,errLSi=
        get_MSN_weights_Unit(x,y,xo,yo,ax,bx,ay,by,s,a11,a12,a21,a22
        ,a11_x,a12_x,a21_y,a22_y)')

    c2mat = mec.gather('c2mat')
    cxxmat = mec.gather('cxxmat')
    cxmat = mec.gather('cxmat')
    cyymat = mec.gather('cyymat')
    cymat = mec.gather('cymat')
    cimat = mec.gather('cimat')
    errLSi = mec.gather('errLSi')

    return c2mat, cxxmat, cxmat, cyymat, cymat, cimat, errLSi

#jsm:
#The nearest neighbor finding algorithm
def find_nn2(x,y,xk,yk,L,b_start_indx):
    #Find the nearest neighbours in a window of hald-width L,centered
        at xk,yk
```

```
    ax = xk - L; bx = xk  + L;
    ay = yk - L; by = yk + L;

    #x,y that fall in this window are the neighbours
    b1 =  x > ax; b2 = x < bx; b3 = y > ay; b4 = y < by;
    biL = np.all(np.bmat('b1 b2 b3 b4'), axis=1)

    iL = np.mat(range(x.size))
    iL = iL[biL.T]

    #if all iL indeces are less than b_start_indx then we are far
        interior (fi)
    #since nothing in the stencil touches a boundary (assuming
        interior equi-gridding)
    if iL.size == 0:
      #Empty neighborhood
      fi = False
    else:
      fi = iL.max() < b_start_indx
    xL = x[biL]; yL = y[biL];
    xL.resize(np.size(xL),1);yL.resize(np.size(yL),1);
    return xL, yL, iL, fi

#Computes the MSN weights using the nearest neighbors, obtained using
    the find_nn2 function
def compute_MSNWts_nn2(x,y,xo,yo,ind_xo,l,s,a11,a12,a21,a22,a11_x,
    a12_x,a21_y,a22_y, c2FI,cxxFI,cyyFI,cxFI,cyFI,cFI,errLSFI,
    b_start_indx):
    c2List = list()
    cxxList = list()
    cxList = list()
    cyyList = list()
    cyList = list()
    ciList = list()
    jList = list()
    iList = list()
    errLSi = np.zeros(xo.shape)
    norm2 = np.zeros(xo.shape)
    normxx = np.zeros(xo.shape)
    normx = np.zeros(xo.shape)
    normyy = np.zeros(xo.shape)
    normy = np.zeros(xo.shape)
    normi = np.zeros(xo.shape)
    bandWidth = np.zeros(xo.shape)

    adaption = False
    dl = 0.25*l #Set the adaption parameter
```

```
    Wsize = 9;
    for i in range(xo.size):
      if adaption == True:
      #Adaption
        while(1):
          xL, yL, jL, fi_flag = find_nn2(x, y, xo[i], yo[i], l,
              b_start_indx)
          xL1, yL1, jL1, fi_flag1 = find_nn2(x, y, xo[i], yo[i], l-dl
              , b_start_indx) #Test one size smaller
          xL2, yL2, jL2, fi_flag2 = find_nn2(x, y, xo[i], yo[i], l+dl
              , b_start_indx) #Test one size smaller

          print xL.size, xL1.size, xL2.size
          if xL.size == Wsize:
            break
          #Now we have a stencil that is big or small. It still may
              be the smallest stencil to take atleast 9 points.
          #Check
          if xL.size > Wsize:
            if xL1.size < Wsize:
              break
            elif xL1.size == Wsize:
              (xL, yL, jL, fi_flag) = (xL1, yL1, jL1, fi_flag1)
              l = l-dl;
              break
            else:
              l = l-dl;
              continue;
          else:
            if xL2.size >= Wsize:
              (xL, yL, jL, fi_flag) = (xL2, yL2, jL2, fi_flag2)
              l = l+dl;
              break;
            else:
              l = l + dl;
              continue;
        #Turn off the FI flag
        fi_flag = False
      else:
        xL, yL, jL, fi_flag = find_nn2(x, y, xo[i], yo[i], l,
            b_start_indx)
        fi_flag = False

#         ax = np.min(xL); ay = np.min(yL);
#         bx = np.max(xL); by = np.max(yL);
      ax = xo[i,0]-l;bx = xo[i,0]+l;
      ay = yo[i,0]-l;by = yo[i,0]+l;
```

```
        if fi_flag == True:
          c2=c2FI
          cxx=cxxFI
          cyy=cyyFI
          cx=cxFI
          cy=cyFI
          c=cFI
          errLS=errLSFI
        else:
          c2,cxx,cyy,cx,cy,c,errLS = compute_MSN_weight(xL,yL,xo[i],yo[
              i],ax,bx,ay,by,s,a11[i][0],a12[i][0],a21[i][0],a22[i][0],
              a11_x[i][0],a12_x[i][0],a21_y[i][0],a22_y[i][0])

        c2List.append(c2)
        cxxList.append(cxx)
        cxList.append(cx)
        cyyList.append(cyy)
        cyList.append(cy)
        ciList.append(c)
        jList.append(jL)
        iList.append(ind_xo[i]*np.ones(jL.shape))
        errLSi[i] = errLS
        norm2[i] = la.norm(c2)
        normxx[i] = la.norm(cxx)
        normx[i] = la.norm(cx)
        normyy[i] = la.norm(cyy)
        normy[i] = la.norm(cy)
        normi[i] = la.norm(c)
        bandWidth[i] = c2.size

    c2mat = list2mat(c2List)
    cxxmat = list2mat(cxxList)
    cxmat = list2mat(cxList)
    cyymat = list2mat(cyyList)
    cymat = list2mat(cyList)
    cmat = list2mat(ciList)
    cjmat = list2mat(jList)
    cimat = list2mat(iList)
    return c2mat, cxxmat, cxmat, cyymat, cymat,cmat, cimat, cjmat,
        errLSi, norm2, normxx, normx, normyy, normy, normi, bandWidth

#Convert a list of vectors into one long list, this shall later be
    used to assemble the sparse
#matrix
def list2mat(x):
    r = np.mat('').reshape(0,1)
```

```
    #Converts the entries in thie list to matrix flattening each
        component.
    while x != []:
        v = x.pop()
        if v.shape[0] != 1 and v.shape[1] != 1:
            print 'Non vector tuple component. Cannot be flattened'
            return -1

        #Is a row vector
        if v.shape[0] == 1:
            v = v.T

        if r.size == 0:
            r = np.mat(v)
        else:
            r = np.bmat('r;v')

    return r

#Compute the MSN weights with nearest neigbour detection in Parallel
def get_MSNWts_nn2_Parallel(x,y,xo,yo,l,s,a11,a12,a21,a22,a11_x,a12_x
    ,a21_y,a22_y, c2FI,cxxFI,cyyFI,cxFI,cyFI,cFI,errLSFI,b_start_indx
    ):
    #Now the parallel part!
    #import all libraries in the engines
    mec = client.MultiEngineClient()
    mec.execute('import numpy as np')
    mec.execute('import scipy as sp')
    mec.execute('import numpy.linalg as la')
    mec.execute('import scipy.sparse as sps')
    ##mec.execute('import scipy.sparse.linalg as spla')
    mec.execute('import sympy as sym')
    mec.execute('from ctypes import *')
    mec.execute('mkl = cdll.LoadLibrary("clapack.so")')
    mec.execute('dgesvd = mkl.dgesvd_')

    #push asll functions to the engines
    mec.push_function(dict(WLS_Solve=WLS_Solve))
    mec.push_function(dict(compute_refine=compute_refine))
    mec.push_function(dict(refine_v=refine_v))
    mec.push_function(dict(CODA_WLS=CODA_WLS))
    mec.push_function(dict(scale_2D=scale_2D))
    mec.push_function(dict(vandermonde2D=vandermonde2D))
    mec.push_function(dict(meshnorm2D=meshnorm2D))
    mec.push_function(dict(ctypes_svd=ctypes_svd))
    mec.push_function(dict(MSN2DWt=MSN2DWt))
```

```
mec.push_function(dict(computeVanderFunctionals2D=
    computeVanderFunctionals2D))
mec.push_function(dict(compute_MSN_weight=compute_MSN_weight))
mec.push_function(dict(list2mat=list2mat))
mec.push_function(dict(find_nn2=find_nn2))
mec.push_function(dict(compute_MSNWts_nn2=compute_MSNWts_nn2))
mec.push(dict(x=x))
mec.push(dict(y=y))
mec.push(dict(s=s))
mec.push(dict(l=l))
mec.push(dict(c2FI=c2FI))
mec.push(dict(cxxFI=cxxFI))
mec.push(dict(cyyFI=cyyFI))
mec.push(dict(cxFI=cxFI))
mec.push(dict(cyFI=cyFI))
mec.push(dict(cFI=cFI))
mec.push(dict(errLSFI=errLSFI))
mec.push(dict(b_start_indx=b_start_indx))

mec.scatter('xo',xo)
mec.scatter('yo',yo)
mec.scatter('ind_xo', range(xo.size))
mec.scatter('a11',a11)
mec.scatter('a12',a12)
mec.scatter('a21',a21)
mec.scatter('a22',a22)
mec.scatter('a11_x',a11_x)
mec.scatter('a12_x',a12_x)
mec.scatter('a21_y',a21_y)
mec.scatter('a22_y',a22_y)

mec.execute('c2mat,␣cxxmat,␣cxmat,␣cyymat,␣cymat,cmat,␣cimat,␣
    cjmat,␣errLSi,␣norm2,␣normxx,␣normx,␣normyy,␣normy,␣normi,␣
    bandWidth␣=␣compute_MSNWts_nn2(x,y,xo,yo,ind_xo,l,s,a11,a12,
    a21,a22,a11_x,a12_x,a21_y,a22_y,␣c2FI,cxxFI,cyyFI,cxFI,cyFI,
    cFI,errLSFI,b_start_indx)')

c2mat = mec.gather('c2mat')
cxxmat = mec.gather('cxxmat')
cxmat = mec.gather('cxmat')
cyymat = mec.gather('cyymat')
cymat = mec.gather('cymat')
cmat =  mec.gather('cmat')
cimat = mec.gather('cimat')
cjmat = mec.gather('cjmat')
errLSi = mec.gather('errLSi')
norm2 = mec.gather('norm2')
```

```
    normxx = mec.gather('normxx')
    normx = mec.gather('normx')
    normyy = mec.gather('normyy')
    normy = mec.gather('normy')
    normi = mec.gather('normi')
    bandWidth = mec.gather('bandWidth')

    return c2mat, cxxmat, cxmat, cyymat, cymat,cmat, cimat,cjmat,
        errLSi, norm2, normxx, normx, normyy, normy, normi, bandWidth

def point_in_poly(x,y,poly):
    n = poly.shape[0]
    inside = False

    p1x = poly[0,0]
    p1y = poly[0,1]
    for i in range(n+1):
        p2x = poly[i % n,0]
        p2y = poly[i % n,1]
        if y > min(p1y,p2y):
            if y <= max(p1y,p2y):
                if x <= max(p1x,p2x):
                    if p1y != p2y:
                        xinters = (y-p1y)*(p2x-p1x)/(p2y-p1y)+p1x
                    if p1x == p2x or x <= xinters:
                        inside = not inside
        p1x,p1y = p2x,p2y

    return inside

def point_poly_parallel(xei1, yei1, poly_in):
  mec = client.get_multiengine_client()
  mec.execute('import numpy as np')
  mec.execute('from numpy import *')
  mec.push(dict(poly_in=poly_in))
  mec.push_function(dict(point_in_poly=point_in_poly))
  mec.push_function(dict(point_poly_unit=point_poly_unit))
  mec.scatter('xei1',xei1);  mec.scatter('yei1',yei1);
  mec.execute('v=point_poly_unit(xei1, yei1, poly_in)')
  v = mec.gather('v')
  return v

def point_poly_unit(xei1, yei1, poly_in):
  v = np.zeros(xei1.shape) == 1.0;
  for i in range(xei1.size):
    v[i] = point_in_poly(xei1[i], yei1[i], poly_in);
  return v
```

```
#Function that performs the conformal mapping of the given set of
#Coordinates from star t square to circle using the parameter p.
def mesh_MapFromRect(x,y, p):
    s = ((np.abs(x)**p+np.abs(y)**p)**(1./p))/(0.0001 + np.abs(x)**2.
        + np.abs(y)**2.)**0.5
    xo = s*x;yo = s*y;
    return xo,yo

def generate_grid(Nx,Ny,xo,yo,w,h, type,p):
    #xbh, ybh = morph_square(xo,yo, w, h,4*N-4,p)
    xbh, ybh = morph_square_equi(xo,yo, w, h, Nx, Ny,p)
    poly = np.bmat('xbh,ybh')
    ax = np.min(xbh);      bx = np.max(xbh);
    ay = np.min(ybh);      by = np.max(ybh);

    if type == 'random':
        x = ax+(bx-ax)*np.random.rand(N*N,1)
        y = ay+(by-ay)*np.random.rand(N*N,1)
    else:
        x = np.linspace(ax, bx,Nx)
        y = np.linspace(ay, by,Ny)
        X,Y = np.meshgrid(x,y)
        x = X.reshape(X.size, 1)
        y = Y.reshape(Y.size, 1)

    is_interior = point_poly_parallel(x,y,poly)

    xi = x[is_interior]
    yi = y[is_interior]

    xi = xi.reshape(xi.size, 1)
    yi = yi.reshape(yi.size, 1)

    thr = 0.5/min(Nx, Ny)
    xsi, ysi = cleanup_boundary(xbh, ybh, xi, yi, thr)
    return xsi, ysi, xbh, ybh

def morph_square_equi(xo, yo, w, h, Nx, Ny, p):
    Nx = 2*Nx-2; Ny = 2*Ny-2;
    ax = xo-w/2.0
    ay = yo-h/2.0
    bx = xo+w/2.0
    by = yo+h/2.0

    bsx = 2#We have 2 x boundary segments and 2 y boundary segments.
    bsy = 2
```

```
    bs = bsx+bsy

    Nx = round(Nx/float(bsx))*bsx
    Ny = round(Ny/float(bsy))*bsy

    x = np.zeros([Nx+Ny,1])
    y = np.zeros([Ny+Nx,1])

    nbsx = Nx/bsx
    nbsy = Ny/bsy
    if p==100.0:

      nbsy=nbsx
      theta = np.linspace(3.0/2.0*np.pi,2*np.pi,nbsx+1)

      x[0:nbsx,0] = 0.5*np.cos(theta[0:(nbsx)])
      y[0:nbsx,0] = 0.5*np.sin(theta[0:(nbsx)])

      theta = np.linspace(0.0,0.5*np.pi,nbsx+1)
      y[nbsx:nbsx+nbsy,0] = 0.5*np.sin(theta[0:(nbsx)])
      x[nbsx:nbsx+nbsy,0] = 0.5*np.cos(theta[0:(nbsx)])

      theta = np.linspace(0.5*np.pi,np.pi,nbsx+1)
      x[nbsx+nbsy:2*nbsx+nbsy,0] = 0.5*np.cos(theta[0:(nbsx)])
      y[nbsx+nbsy:2*nbsx+nbsy,0] = 0.5*np.sin(theta[0:(nbsx)])

      theta = np.linspace(np.pi,1.5*np.pi,nbsx+1)
      y[2*nbsx+nbsy:2*nbsx+2*nbsy,0] = 0.5*np.sin(theta[0:(nbsx)])
      x[2*nbsx+nbsy:2*nbsx+2*nbsy,0] = 0.5*np.cos(theta[0:(nbsx)])

      x = np.arctan(x); y = np.arctan(y);
    elif p==200.0:#Half Tear drop
      ax = -1.5
      ay = 0.0
      bx = 2.0
      by = 0.5

      bsx = 1
      bsy = 1
      bs = bsx+bsy

      Nx = round(Nx/float(bsx))*bsx
      Ny = round(2*Ny/float(bsy))*bsy

      x = np.zeros([Nx+Ny,1])
      y = np.zeros([Ny+Nx,1])
```

```
nbsx = Nx/bsx
nbsy = Ny/bsy

#Bottom, starting at left
NC = 400;
xth = np.linspace(ax,bx,NC).reshape([1,NC])
yth = ax*np.ones([1,NC])

xth = np.arctan(xth)
yth = np.arctan(yth)

al =((xth[0,1:]-xth[0,0:-1])**2.0+(yth[0,1:]-yth[0,0:-1])**2.0)
    **0.5
#alth is arclength corresponding of each point on the curve
#from the morphed equisampled square
#th will be the parameter values on the square which gave them
alth = np.hstack([0.0,np.cumsum(al)])
th = np.linspace(0.0,alth[-1],NC)
#t_eq equi spaced arclengths
#p_eq paramater values that correspond to equispaced arclengths
     on
#the curve xth,yth
t_eq = np.linspace(0.0,alth[-1],nbsx+1)
p_eq = sp.interp(t_eq,alth,th)
p_eq = p_eq[0:-1]/p_eq[-1]
p_eq.reshape(p_eq.size, 1)

x[0:nbsx,0] = (bx-ax)*p_eq+ax
y[0:nbsx] = ay*np.ones([p_eq.size,1])

#Now the half tear shape.
NC = 400;
th = np.linspace(np.pi, 2*np.pi, NC);
xth = (bx-ax)*np.sin(th/2.0)+ax
yth = -np.sin(th)

xth = np.arctan(xth)
yth = np.arctan(yth)

al =((xth[1:]-xth[0:-1])**2.0+(yth[1:]-yth[0:-1])**2.0)**0.5
alth = np.hstack([0,np.cumsum(al)])

t_eq = np.linspace(0.0  ,alth[-1],nbsy+1)

p_eq = np.interp(t_eq,alth,th)
p_eq = p_eq[0:-1]
p_eq.reshape(p_eq.size, 1)
```

```
    x[nbsx:nbsx+nbsy,0] = (bx-ax)*(np.sin(p_eq/2.0))+ax
    y[nbsx:nbsx+nbsy,0] = -np.sin(p_eq)

    x = np.arctan(x)-0.25; y = np.arctan(y)-0.25;
  else:
    for iter in range(bsx):
      for jter in range(bsy):
        k = iter*bsx+jter;
        if k==0:
            #Bottom, starting at left
            NC = 400;
            xth = np.linspace(ax,bx,NC)
            yth = ax*np.ones([1,NC])
            xth,yth = mesh_MapFromRect(xth,yth,p)
            al =((xth[0,1:]-xth[0,0:-1])**2.0+(yth[0,1:]-yth
                [0,0:-1])**2.0)**0.5
            #alth is arclength corresponding of each point on the
                curve
            #from the morphed equisampled square
            #th will be the parameter values on the square which
                gave them
            alth = np.hstack([0.0,np.cumsum(al)])
            th = np.linspace(0.0,alth[-1],NC)
            #t_eq equi spaced arclengths
            #p_eq paramater values that correspond to equispaced
                arclengths on
            #the curve xth,yth
            t_eq = np.linspace(0.0,alth[-1],nbsx+1)
            p_eq = sp.interp(t_eq,alth,th)
            p_eq = p_eq[0:-1]/p_eq[-1]
            p_eq.reshape(p_eq.size, 1)

            x[0:nbsx,0] = (bx-ax)*p_eq+ax
            y[0:nbsx] = ay*np.ones([p_eq.size,1])
        elif k==1:
            #Right, starting at bottom
            NC = 400;
            yth = np.linspace(ay,by,NC)
            xth = bx*np.ones([1,NC])
            xth,yth = mesh_MapFromRect(xth,yth,p)

            al =((xth[0,1:]-xth[0,0:-1])**2.0+(yth[0,1:]-yth
                [0,0:-1])**2.0)**0.5
            alth = np.hstack([0,np.cumsum(al)])
            th = np.linspace(0,alth[-1],NC)
```

```
            t_eq = np.linspace(0,alth[-1],nbsy+1)
            p_eq = sp.interp(t_eq,alth,th)
            p_eq = p_eq[0:-1]/p_eq[-1]

            p_eq.reshape(p_eq.size, 1)
            y[nbsx:nbsx+nbsy,0] = (by-ay)*p_eq+ay
            x[nbsx:nbsx+nbsy] = bx*np.ones([p_eq.size,1])
        elif k==2:
            #Top starting at right
            NC = 400;
            xth = np.linspace(bx,ax,NC)
            yth = by*np.ones([1,NC])
            xth,yth = mesh_MapFromRect(xth,yth,p)

            al =((xth[0,1:]-xth[0,0:-1])**2.0+(yth[0,1:]-yth
                [0,0:-1])**2.0)**0.5
            alth = np.hstack([0,np.cumsum(al)])
            th = np.linspace(0,alth[-1],NC)

            t_eq = np.linspace(0,alth[-1],nbsx+1)
            p_eq = np.interp(t_eq,alth,th)
            p_eq = p_eq[0:-1]/p_eq[-1]

            p_eq.reshape(p_eq.size, 1)
            x[nbsx+nbsy:2*nbsx+nbsy,0] = -1.0*(bx-ax)*p_eq+bx
            y[nbsx+nbsy:2*nbsx+nbsy] = by*np.ones([p_eq.size,1])
        else:
            #Left, starting at top
            NC = 400;
            yth = np.linspace(by,ay,NC)
            xth = ax*np.ones([1,NC])
            xth,yth = mesh_MapFromRect(xth,yth,p)

            al =((xth[0,1:]-xth[0,0:-1])**2.0+(yth[0,1:]-yth
                [0,0:-1])**2.0)**0.5
            alth = np.hstack([0,np.cumsum(al)])
            th = np.linspace(0,alth[-1],NC)

            t_eq = np.linspace(0,alth[-1],nbsy+1)
            p_eq = np.interp(t_eq,alth,th)
            p_eq = p_eq[0:-1]/p_eq[-1]

            p_eq.reshape(p_eq.size, 1)
            y[2*nbsx+nbsy:2*nbsx+2*nbsy,0] = -1.0*(by-ay)*p_eq+by
            x[2*nbsx+nbsy:2*nbsx+2*nbsy] = ax*np.ones([p_eq.size
                ,1])
    x,y = mesh_MapFromRect(x,y,p)
```

```
    return x,y

#Square geom
def Square(xo, yo, w, h, N):
    ax = xo-w/2.0
    ay = yo-h/2.0
    bx = xo+w/2.0
    by = yo+h/2.0

    bs = 4
    N = round(N/float(bs))*bs
    x = np.zeros([N,1])
    y = np.zeros([N,1])
    #The parameter
    t = np.arange(0,1,1.0/N)
    nbs = N/bs
    for k in range(bs):
        if k==0:
            #Bottom, starting at left
            t1 = t[k*nbs:(k+1)*nbs]
            a0 = t[k*nbs]
            b0 = t[(k+1)*nbs]
            a1 = ax;
            b1 = bx;
            #Map [a0,b0] -> [a1,b1] using a linear transform
            a = (a1-b1)/(a0-b0)
            b = (a0*b1-b0*a1)/(a0-b0)
            x[k*nbs:(k+1)*nbs,0] = a*t1+b
            y[k*nbs:(k+1)*nbs] = ay*np.ones([t1.size,1])
        elif k==1:
            #Right, starting at bottom
            t1 = t[k*nbs:(k+1)*nbs]
            a0 = t[k*nbs]
            b0 = t[(k+1)*nbs]
            a1 = ay;
            b1 = by;
            #Map [a0,b0] -> [a1,b1] using a linear transform
            a = (a1-b1)/(a0-b0)
            b = (a0*b1-b0*a1)/(a0-b0)
            y[k*nbs:(k+1)*nbs,0] = a*t1+b
            x[k*nbs:(k+1)*nbs] = by*np.ones([t1.size,1])
        elif k==2:
            #Top starting at right
            t1 = t[k*nbs:(k+1)*nbs]
            a0 = t[k*nbs]
```

```
            b0 = t[(k+1)*nbs]
            a1 = bx
            b1 = ax
            #Map [a0,b0] -> [a1,b1] using a linear transform
            a = (a1-b1)/(a0-b0)
            b = (a0*b1-b0*a1)/(a0-b0)
            x[k*nbs:(k+1)*nbs,0] = a*t1+b
            y[k*nbs:(k+1)*nbs] = by*np.ones([t1.size,1])
        else:
            #Right, starting at top
            t1 = t[k*nbs:(k+1)*nbs]
            a0 = t[k*nbs]
            b0 = t[(k+1)*nbs-1]+1.0/N
            a1 = by
            b1 = ay
            #Map [a0,b0] -> [a1,b1] using a linear transform
            a = (a1-b1)/(a0-b0)
            b = (a0*b1-b0*a1)/(a0-b0)
            y[k*nbs:(k+1)*nbs,0] = a*t1+b
            x[k*nbs:(k+1)*nbs] = ax*np.ones([t1.size,1])

    return x,y

#creates morphed geometry outlines
def morph_square(xo, yo, w, h, N,p):
    x,y = Square(xo,yo,w,h,N)
    x,y = mesh_MapFromRect(x,y,p)
    return x,y

#cleans up interior points near the boundary
def cleanup_boundary(xb, yb, xi, yi, t):
    survivor = np.ones([xi.size, 1]) > 0
    for i in range(xb.size):
        d = ((xi-xb[i,0])**2.0+(yi-yb[i,0])**2.0)**0.5
        s = d > t
        s = np.bmat('s_survivor')
        survivor = np.all(s,1)

    xs = xi[survivor]
    ys = yi[survivor]
    return xs.reshape(xs.size, 1), ys.reshape(ys.size, 1)

#Function to compute the RHS of the general second order equation
#Uses symbolic arithmetic
def varPDE2_Evalfg(a11, a12, a21, a22, b1, b2, c, d,e1,e2,u):
    f = sym.diff(a11*sym.diff(u,'x'),'x') + sym.diff(a12*sym.diff(u,'
        x'),'y') + sym.diff(a21*sym.diff(u,'y'),'x') + sym.diff(a22*
```

```
        sym.diff(u,'y'),'y') + b1*sym.diff(u,'x') + b2*sym.diff(u,'y'
        ) + c*u
    g = e1*sym.diff(u,'x') + e2*sym.diff(u,'y') + d*u
    return f,g

def PDE2_Assemble(xi, yi, xb, yb, a, b, c, d, e, f, g, l, s,
    MAGIC_NUMBER, uref_flag, uref):
    a11 = a[0]; a12 = a[1]; a21 = a[2]; a22 = a[3];
    a11_x = a[4]; a12_x = a[5]; a21_y = a[6]; a22_y = a[7];
    x = np.bmat('xi;xb'); y= np.bmat('yi;yb');
    xo = x; yo = y;
    N = x.size;
    ind_xo = range(xo.size)

    #From jsm's code version:
    #Find some far interior point and compute its weights
    #If the nearest neighbourhood does not have a boundary point
    #We call it a far-interior point.
    ni = xi.size
    for ii in range(ni):
        xL, yL, iL, fi = find_nn2(x,y,xo[ii],yo[ii],l,ni)
        if fi == True:
            ax = xo[ii,0]-l;bx = xo[ii,0]+l;
            ay = yo[ii,0]-l;by = yo[ii,0]+l;
            c2FI,cxxFI,cyyFI,cxFI,cyFI,cFI,errLSFI=compute_MSN_weight
                (xL,yL,xo[ii],yo[ii],ax,bx,ay,by,s,a11[ii][0],a12[ii
                ][0],a21[ii][0],a22[ii][0],a11_x[ii][0],a12_x[ii][0],
                a21_y[ii][0],a22_y[ii][0])
            break

    if fi==False:
      (c2FI,cxxFI,cyyFI,cxFI,cyFI,cFI,errLSFI) =
          (-1,-1,-1,-1,-1,-1,-1)

    #Get the coefficients
    timea1 = tm.time()
    #c2mat, cxxmat, cxmat, cyymat, cymat, cmat, cimat, cjmat, errLSi,
        norm2, normxx, normx, normyy, normy, normi, bandWidth =
        compute_MSNWts_nn2(x,y,xo,yo,ind_xo,l,s,a11,a12,a21,a22,a11_x
        ,a12_x,a21_y,a22_y, c2FI,cxxFI,cyyFI,cxFI,cyFI,cFI,errLSFI,
        ni)
    c2mat, cxxmat, cxmat, cyymat, cymat, cmat, cimat, cjmat, errLSi,
        norm2, normxx, normx, normyy, normy, normi, bandWidth =
        get_MSNWts_nn2_Parallel(x,y,xo,yo,l,s,a11,a12,a21,a22,a11_x,
        a12_x,a21_y,a22_y, c2FI,cxxFI,cyyFI,cxFI,cyFI,cFI,errLSFI, ni
        )
    timea2 = tm.time()
```

```
    c_data_o = (c2mat, cxxmat, cxmat, cyymat, cymat, cmat, cimat,
        cjmat, errLSi, norm2, normxx, normx, normyy, normy, normi,
        bandWidth, timea1, timea2)
    c_data_i = (x,y,xo,yo,l,s,a11,a12,a21,a22,a11_x,a12_x,a21_y,a22_y
        , c2FI,cxxFI,cyyFI,cxFI,cyFI,cFI,errLSFI, ni)
    c_data = (c_data_i, c_data_o)

    cdata_fname = "PDE2_Solve_cdata_%s.cdata.gz" %(MAGIC_NUMBER)
    cdata_fobj = gz.open(cdata_fname, "wb")
    pkl.dump(c_data, cdata_fobj, pkl.HIGHEST_PROTOCOL)
    cdata_fobj.close
    return c_data

#Set up the PDE Solver
def PDE2_Solve(xi, yi, xb, yb, a, b, c, d, e, f, g, l, s,
    MAGIC_NUMBER, uref_flag, uref):
    a11 = a[0]; a12 = a[1]; a21 = a[2]; a22 = a[3];
    a11_x = a[4]; a12_x = a[5]; a21_y = a[6]; a22_y = a[7];
    x = np.bmat('xi;xb'); y= np.bmat('yi;yb');
    xo = x; yo = y;
    N = x.size;
    ind_xo = range(xo.size)

    cdata_fname = "PDE2_Solve_cdata_%s.cdata.gz" %(MAGIC_NUMBER)
    cdata_fobj = gz.open(cdata_fname)
    c_data = pkl.load(cdata_fobj)
    cdata_fobj.close

    (c_data_i, c_data_o) = c_data
    (c2mat, cxxmat, cxmat, cyymat, cymat, cmat, cimat, cjmat, errLSi,
        norm2, normxx, normx, normyy, normy, normi, bandWidth,
        timea1, timea2) = c_data_o

    #Create the sparse matrices
    S2mat = sps.coo_matrix((np.array(c2mat.T)[0], (np.array(cimat.T)
        [0], np.array(cjmat.T)[0])), shape=(N, N))
    Sxxmat = sps.coo_matrix((np.array(cxxmat.T)[0], (np.array(cimat.T
        )[0], np.array(cjmat.T)[0])), shape=(N, N))
    Sxmat = sps.coo_matrix((np.array(cxmat.T)[0], (np.array(cimat.T)
        [0], np.array(cjmat.T)[0])), shape=(N, N))
    Syymat = sps.coo_matrix((np.array(cyymat.T)[0], (np.array(cimat.T
        )[0], np.array(cjmat.T)[0])), shape=(N, N))
    Symat = sps.coo_matrix((np.array(cymat.T)[0], (np.array(cimat.T)
        [0], np.array(cjmat.T)[0])), shape=(N, N))
    #set up the diagonal matrices corresponding to b, c, d and e
    b1 = b[0]; b2 = b[1];
```

```
e1 = e[0]; e2 = e[1];

b1Diag = sps.spdiags(b1.flatten(),0,N,N);
b2Diag = sps.spdiags(b2.flatten(),0,N,N);
e1Diag = sps.spdiags(e1.flatten(),0,N,N);
e2Diag = sps.spdiags(e2.flatten(),0,N,N);
cDiag = sps.spdiags(c.flatten(),0,N,N);
dDiag = sps.spdiags(d.flatten(),0,N,N);

#Set up the FD Matrix.
FDi = S2mat + b1Diag*Sxmat + b2Diag*Symat + cDiag
FDb = e1Diag*Sxmat + e2Diag*Symat + dDiag

#Now extract equations from FDi corresponding to xi,yi and those
    from FDb corresponding
#to xb,yb'
FDi = FDi[0:xi.size][:];
FDb = FDb[xi.size:][:];
FD = sps.bmat([[FDi],[ FDb]]);
FD = FD.tocsc();

#The FD Matrix is setup. Need to balance it.
print norm2.shape, b1.shape, normx.shape, b2.shape, normy.shape,
    c.shape
row_scli = (norm2 + np.abs(b1)*normx+np.abs(b2)*normy+np.abs(c))
    **(-1.0)
row_sclb = (np.abs(e1)*normx+np.abs(e2)*normy+np.abs(d))**(-1.0)
row_scli = row_scli[0:xi.size]/xi.size**0.5;
row_sclb = row_sclb[xi.size:]/xb.size**0.5;

row_scl = np.bmat('row_scli;_row_sclb')
row_scale = sps.spdiags(np.array(row_scl).flatten(),0,N,N, "csr")
    ;

#Not column scaling yet.RHS scaling
f = f[0:xi.size];g = g[xi.size:];

zi = np.zeros([xb.size,1])
rhs1 = np.bmat('f;zi');
zb = np.zeros([xi.size,1])
rhs2 = np.bmat('zb;g');

if uref_flag != -1:
  fgref = FD*uref
  resids = np.abs(fgref-rhs1-rhs2)
  resid = np.max(resids)/np.max(np.abs(rhs1+rhs2))
else:
```

```
    resid = 0;

#Row scale the FD Matrix
FD = row_scale*FD;

xo = np.mat(np.random.rand(N, 10))
#Unit normalize xo
for i in range(10):
    xo[:,i] = xo[:,i]/np.max(np.abs(xo[:,i]))
#Compute the RHS
bo = FD*xo
#compute the maximum inf norm
normFD = np.max(np.max(np.abs(bo)))

rhs1 = row_scale*rhs1
rhs2 = row_scale*rhs2

x1 = np.zeros([N, 10])
#Solve the sparse system
timesp1 = tm.time()
LU = spla.dsolve.splu(FD)
#Apply the factors to one RHS at a time.
u1 = LU.solve(np.array(rhs1).flatten())
u2 = LU.solve(np.array(rhs2).flatten())
timesp2 = tm.time()

Fim = (row_scale[0:xi.size,:]*S2mat, row_scale[0:xi.size,:]*
    b1Diag*Sxmat, row_scale[0:xi.size,:]*b2Diag*Symat, row_scale
    [0:xi.size,:]*cDiag)
Fbm = (row_scale[xi.size:,:]*e1Diag*Sxmat , row_scale[xi.size
    :,:]*e2Diag*Symat , row_scale[xi.size:,:]*dDiag)

u1r, r1 = iter_refine_FD(4, 3, Fim, Fbm, xi.size, u1, rhs1, LU,
    2)
u2r, r2 = iter_refine_FD(4, 3, Fim, Fbm, xi.size, u2, rhs2, LU,
    2)

u = u1+u2
ur = u1r + u2r

for i in range(1,11):
    x1[:,i-1] = LU.solve(np.array(xo[:,i-1]).flatten())

normFDi = np.max(np.max(np.abs(x1)))
wallis = (np.pi*(N-1)*0.5)**0.5
condNo = wallis*normFD*normFDi
```

```
    errLS =np.max(abs(errLSi))

    return ur,x,y,condNo, resid, errLS, np.max(bandWidth), timea2-
        timea1, timesp2-timesp1

def PDE2_AssembleNSolve(xi, yi, xb, yb, a, b, c, d, e, f, g, l, s,
    MAGIC_NUMBER, uref_flag, uref):
  c_data = PDE2_Assemble(xi, yi, xb, yb, a, b, c, d, e, f, g, l, s,
      MAGIC_NUMBER, uref_flag, uref)
  return PDE2_Solve(xi, yi, xb, yb, a, b, c, d, e, f, g, l, s,
      MAGIC_NUMBER, uref_flag, uref)

def PDE2_HyperAssemble(xi, yi, xb, yb, a, b, c, d, e, f, g, h, bexcl,
    baddl, l, s, MAGIC_NUMBER, uref_flag, uref):
    a11 = a[0]; a12 = a[1]; a21 = a[2]; a22 = a[3];
    a11_x = a[4]; a12_x = a[5]; a21_y = a[6]; a22_y = a[7];
    x = np.bmat('xi;xb'); y= np.bmat('yi;yb');
    xo = x; yo = y;
    N = x.size;
    ind_xo = range(xo.size)

    ni = xi.size;
    ind = np.arange(x.size)
    ind_excl = ind[bexcl.flatten()] #Indices for excluded boundary
        equations
    ind_addl = ind[baddl.flatten()] #Indices for additional boundary
        equations
    nb2 = ind_addl.size;
    nb1 = x.size - ni - nb2;
    print ni, nb1, nb2

    g_excl = g[ind_excl]; g_addl = h[ind_addl];
    g_excl = g_excl.reshape(g_excl.size, 1);     g_addl = g_addl.
        reshape(g_addl.size, 1);
    zbb = np.zeros([nb2,1]);     zb = np.zeros([ni+nb1,1]);
    g = np.bmat('g_excl;zbb'); h = np.bmat('zb;g_addl');


    #From jsm's code version:
    #Find some far interior point and compute its weights
    #If the nearest neighbourhood does not have a boundary point
    #We call it a far-interior point.
    for ii in range(ni):
        xL, yL, iL, fi = find_nn2(x,y,xo[ii],yo[ii],l,ni)
        if fi == True:
            ax = xo[ii,0]-l;bx = xo[ii,0]+l;
            ay = yo[ii,0]-l;by = yo[ii,0]+l;
```

```
            c2FI,cxxFI,cyyFI,cxFI,cyFI,cFI,errLSFI=compute_MSN_weight
                (xL,yL,xo[ii],yo[ii],ax,bx,ay,by,s,a11[ii][0],a12[ii
                ][0],a21[ii][0],a22[ii][0],a11_x[ii][0],a12_x[ii][0],
                a21_y[ii][0],a22_y[ii][0])
            break

    #Get the coefficients
    timea1 = tm.time()
    #c2mat, cxxmat, cxmat, cyymat, cymat, cmat, cimat, cjmat, errLSi,
         norm2, normxx, normx, normyy, normy, normi, bandWidth =
        compute_MSNWts_nn2(x,y,xo,yo,ind_xo,l,s,a11,a12,a21,a22,a11_x
        ,a12_x,a21_y,a22_y, c2FI,cxxFI,cyyFI,cxFI,cyFI,cFI,errLSFI,
        ni)
    c2mat, cxxmat, cxmat, cyymat, cymat, cmat, cimat, cjmat, errLSi,
        norm2, normxx, normx, normyy, normy, normi, bandWidth =
        get_MSNWts_nn2_Parallel(x,y,xo,yo,l,s,a11,a12,a21,a22,a11_x,
        a12_x,a21_y,a22_y, c2FI,cxxFI,cyyFI,cxFI,cyFI,cFI,errLSFI, ni
        )
    timea2 = tm.time()

    c_data_o = (c2mat, cxxmat, cxmat, cyymat, cymat, cmat, cimat,
        cjmat, errLSi, norm2, normxx, normx, normyy, normy, normi,
        bandWidth, timea1, timea2)
    c_data_i = (x,y,xo,yo,l,s,a11,a12,a21,a22,a11_x,a12_x,a21_y,a22_y
        , c2FI,cxxFI,cyyFI,cxFI,cyFI,cFI,errLSFI, ni)
    c_data = (c_data_i, c_data_o)

    cdata_fname = "PDE2Hyper_Solve_cdata_%s.cdata.gz" %(MAGIC_NUMBER)
    cdata_fobj = gz.open(cdata_fname, "wb")
    pkl.dump(c_data, cdata_fobj, pkl.HIGHEST_PROTOCOL)
    cdata_fobj.close
    return c_data

def PDE2_HyperSolve(xi, yi, xb, yb, a, b, c, d, e, f, g, h, bexcl,
    baddl, l, s, MAGIC_NUMBER, uref_flag, uref):
    a11 = a[0]; a12 = a[1]; a21 = a[2]; a22 = a[3];
    a11_x = a[4]; a12_x = a[5]; a21_y = a[6]; a22_y = a[7];
    x = np.bmat('xi;xb'); y= np.bmat('yi;yb');
    xo = x; yo = y;
    N = x.size;
    ind_xo = range(xo.size)

    ni = xi.size;
    ind = np.arange(x.size)
    ind_excl = ind[bexcl.flatten()] #Indices for excluded boundary
        equations
```

```
ind_addl = ind[baddl.flatten()] #Indices for additional boundary
    equations
nb2 = ind_addl.size;
nb1 = x.size - ni - nb2;

g_excl = g[ind_excl]; g_addl = h[ind_addl];
g_excl = g_excl.reshape(g_excl.size, 1);     g_addl = g_addl.
    reshape(g_addl.size, 1);
zbb = np.zeros([nb2,1]);     zb = np.zeros([ni+nb1,1]);
g = np.bmat('g_excl;zbb'); h = np.bmat('zb;g_addl');

#Try to read from file instead.
cdata_fname = "PDE2Hyper_Solve_cdata_%s.cdata.gz" %(MAGIC_NUMBER)
cdata_fobj = gz.open(cdata_fname)
c_data = pkl.load(cdata_fobj)
cdata_fobj.close

(c_data_i, c_data_o) = c_data
(c2mat, cxxmat, cxmat, cyymat, cymat, cmat, cimat, cjmat, errLSi,
     norm2, normxx, normx, normyy, normy, normi, bandWidth,
    timea1, timea2) = c_data_o
#TODO: Assert if the inputs match before using the outputs.

#Create the sparse matrices
S2mat = sps.coo_matrix((np.array(c2mat.T)[0], (np.array(cimat.T)
    [0], np.array(cjmat.T)[0])), shape=(N, N))
Sxxmat = sps.coo_matrix((np.array(cxxmat.T)[0], (np.array(cimat.T
    )[0], np.array(cjmat.T)[0])), shape=(N, N))
Sxmat = sps.coo_matrix((np.array(cxmat.T)[0], (np.array(cimat.T)
    [0], np.array(cjmat.T)[0])), shape=(N, N))
Syymat = sps.coo_matrix((np.array(cyymat.T)[0], (np.array(cimat.T
    )[0], np.array(cjmat.T)[0])), shape=(N, N))
Symat = sps.coo_matrix((np.array(cymat.T)[0], (np.array(cimat.T)
    [0], np.array(cjmat.T)[0])), shape=(N, N))
#set up the diagonal matrices corresponding to b, c, d and e
b1 = b[0]; b2 = b[1];
e1 = e[0]; e2 = e[1];

b1Diag = sps.spdiags(b1.flatten(),0,N,N);
b2Diag = sps.spdiags(b2.flatten(),0,N,N);
e1Diag = sps.spdiags(e1.flatten(),0,N,N);
e2Diag = sps.spdiags(e2.flatten(),0,N,N);
cDiag = sps.spdiags(c.flatten(),0,N,N);
dDiag = sps.spdiags(d.flatten(),0,N,N);

#Set up the FD Matrix.
FDi = S2mat + b1Diag*Sxmat + b2Diag*Symat + cDiag
```

```
FDb = e1Diag*Sxmat + e2Diag*Symat + dDiag
FDb1 = Symat.tocsc()

FDb = FDb[ind_excl][:]
FDb1 = FDb1[ind_addl][:]

FDb = sps.bmat([[FDb],[ FDb1]]).tocsc();

#Now extract equations from FDi corresponding to xi,yi and those
    from FDb corresponding
#to xb,yb`
FDi = FDi[0:xi.size][:];
FDb = FDb[xi.size:][:];

FD = sps.bmat([[FDi],[ FDb]]);
FD = FD.tocsc();

#The FD Matrix is setup. Need to balance it.
row_scli = (norm2 + np.abs(b1)*normx+np.abs(b2)*normy+np.abs(c))
    **(-1.0)
row_sclb = (np.abs(e1)*normx+np.abs(e2)*normy+np.abs(d))**(-1.0)
row_sclb1= (normy)**(-1.0)

row_sclb = row_sclb[ind_excl].reshape(ind_excl.size, 1)
row_sclb1 = row_sclb1[ind_addl].reshape(ind_addl.size, 1)
row_sclb = np.bmat('row_sclb;␣row_sclb1')

row_scli = row_scli[0:xi.size]/xi.size**0.5;
row_sclb = row_sclb[xi.size:]/xb.size**0.5;

row_scl = np.bmat('row_scli;␣row_sclb')
row_scale = sps.spdiags(np.array(row_scl).flatten(),0,N,N, "csr")
    ;

#Not column scaling yet.RHS scaling
f = f[0:xi.size];g = g[xi.size:];h = h[xi.size:];

zi = np.zeros([xb.size,1])
rhs1 = np.bmat('f;zi');
zb = np.zeros([xi.size,1])
rhs2 = np.bmat('zb;g');
rhs3 = np.bmat('zb;h');

if uref_flag != -1:
  fgref = FD*uref
  resids = np.abs(fgref-rhs1-rhs2-rhs3)
  resid = np.max(resids)/np.max(np.abs(rhs1+rhs2+rhs3))
```

```
    else:
      resid = 0;

    FDu = FD;
    #Row scale the FD Matrix
    FD = row_scale*FD;

    xo = np.mat(np.random.rand(N, 10))
    #Unit normalize xo
    for i in range(10):
        xo[:,i] = xo[:,i]/np.max(np.abs(xo[:,i]))
    #Compute the RHS
    bo = FD*xo
    #compute the maximum inf norm
    normFD = np.max(np.max(np.abs(bo)))

    rhs1 = row_scale*rhs1
    rhs2 = row_scale*rhs2
    rhs3 = row_scale*rhs3

    x1 = np.zeros([N, 10])
    #Solve the sparse system
    timesp1 = tm.time()

    LU = spla.dsolve.splu(FD,'MMD_AT_PLUS_A')
    #Apply the factors to one RHS at a time.
    u1 = LU.solve(np.array(rhs1).flatten())
    u2 = LU.solve(np.array(rhs2).flatten())
    u3 = LU.solve(np.array(rhs3).flatten())
    timesp2 = tm.time()

    Fim = (row_scale[0:xi.size,:]*S2mat, row_scale[0:xi.size,:]*
        b1Diag*Sxmat, row_scale[0:xi.size,:]*b2Diag*Symat, row_scale
        [0:xi.size,:]*cDiag)
    Fb1 = (row_scale[ni:ni+nb1,:]*e1Diag*Sxmat , row_scale[ni:ni+nb1
        ,:]*e2Diag*Symat , row_scale[ni:ni+nb1,:]*dDiag)
    Fb2 = (row_scale[ni+nb1:,:]*Symat)

    u1r, r1 = iter_refine_FDHyper(4, 3, 1, Fim, Fb1, Fb2, ni, nb1,
        nb2, u1, rhs1, LU, 2)
    u2r, r2 = iter_refine_FDHyper(4, 3, 1, Fim, Fb1, Fb2, ni, nb1,
        nb2, u2, rhs2, LU, 2)
    u3r, r3 = iter_refine_FDHyper(4, 3, 1, Fim, Fb1, Fb2, ni, nb1,
        nb2, u3, rhs3, LU, 2)

    u = u1+u2+u3
    ur = u1r + u2r + u3r
```

```
    for i in range(1,11):
        x1[:,i-1] = LU.solve(np.array(xo[:,i-1]).flatten())

    normFDi = np.max(np.max(np.abs(x1)))
    wallis = (np.pi*(N-1)*0.5)**0.5
    condNo = wallis*normFD*normFDi

    errLS =np.max(abs(errLSi))
    return u,x,y,condNo, resid, errLS, np.max(bandWidth), timea2-
        timea1, timesp2-timesp1, FD

#Solver customized for the Hyperbolic equations. Basically, includes
    flags for additional and excluded
#boundary conditions. Additional boundary data in 'h'.
def PDE2_HyperAssembleNSolve(xi, yi, xb, yb, a, b, c, d, e, f, g, h,
    bexcl, baddl, l, s, MAGIC_NUMBER, uref_flag, uref):
    c_data = PDE2_HyperAssemble(xi, yi, xb, yb, a, b, c, d, e, f, g,
        h, bexcl, baddl, l, s, MAGIC_NUMBER, uref_flag, uref)
    return PDE2_HyperSolve(xi, yi, xb, yb, a, b, c, d, e, f, g, h,
        bexcl, baddl, l, s, MAGIC_NUMBER, uref_flag, uref)

#Iterative refinement with two independent boundary sets
def iter_refine_FDHyper(di, db1, db2, Fi, Fb1, Fb2, ni, nb1, nb2, x,
    b, LU, K):
  if K == 0:
    r = np.zeros(b.shape)
  #Allocate memory for the residue. Same as b
  for k in range(K):
    r = b.copy()
    for i in range(di):
      t  = Fi[i]*x
      r[0:ni] = r[0:ni]  -  t.reshape(t.size, 1)
    for j in range(db1):
      t  = Fb1[j]*x
      r[ni:ni+nb1] =  r[ni:ni+nb1]   -  t.reshape(t.size, 1)
    for j in range(db2):
      t  = Fb2[j]*x
      r[ni+nb1:] =  r[ni+nb1:]   -  t.reshape(t.size, 1)
    d = LU.solve(np.array(r).flatten())
    x = x + d
  return x, r

#K step iterative refinement for the FD case. Involves some more
    tricks.
#Fi has di component matrices, Fb has db components. x is split [ni;
    nb]
```

```
def iter_refine_FD(di, db, Fi, Fb, ni,x, b, LU, K):
  if K == 0:
    r = np.zeros(b.shape)
  #Allocate memory for the residue. Same as b
  for k in range(K):
    r = b.copy()
    for i in range(di):
      t  = Fi[i]*x
      r[0:ni] = r[0:ni]  -  t.reshape(t.size, 1)
    for j in range(db):
      t  = Fb[j]*x
      r[ni:] =  r[ni:]   -  t.reshape(t.size, 1)
    d = LU.solve(np.array(r).flatten())
    x = x + d
  return x, r

#k step iterative refinement for a square system Ax=b
def iter_refine(A, x, b, LU, k):
  d = np.zeros(x.shape)
  for i in range(k):
    r = b-A*x
    d = LU.solve(r.flatten())
    x = x + d
  return x, r

#Locally interpolate u, x, y at x1, y1
def MSN_local_interp(x, y, x1, y1, l, s, u, ni):
    xo = x1; yo = y1;
    N = x.size
    N1 = x1.size
    ind_xo = range(xo.size)

    a11 = np.zeros([x1.size, 1])
    a12 = np.zeros([x1.size, 1])
    a21 = np.zeros([x1.size, 1])
    a22 = np.zeros([x1.size, 1])
    a11_x = np.zeros([x1.size, 1])
    a12_x = np.zeros([x1.size, 1])
    a21_y = np.zeros([x1.size, 1])
    a22_y = np.zeros([x1.size, 1])

    for ii in range(ni):
        xL, yL, iL, fi = find_nn2(x,y,x[ii],y[ii],l,ni)
        if fi == True:
            ax = x[ii,0]-l;bx = x[ii,0]+l;
            ay = y[ii,0]-l;by = y[ii,0]+l;
```

```
                c2FI,cxxFI,cyyFI,cxFI,cyFI,cFI,errLSFI=compute_MSN_weight
                    (xL,yL,x[ii],y[ii],ax,bx,ay,by,s,a11[ii][0],a12[ii
                    ][0],a21[ii][0],a22[ii][0],a11_x[ii][0],a12_x[ii][0],
                    a21_y[ii][0],a22_y[ii][0])
                break

    #Get the coefficients
    c2mat, cxxmat, cxmat, cyymat, cymat, cmat, cimat, cjmat, errLSi,
        norm2, normxx, normx, normyy, normy, normi, bandWidth =
        get_MSNWts_nn2_Parallel(x,y,xo,yo,l,s,a11,a12,a21,a22,a11_x,
        a12_x,a21_y,a22_y, c2FI,cxxFI,cyyFI,cxFI,cyFI,cFI,errLSFI, ni
        )

    #Create the sparse matrices
    Imat = sps.coo_matrix((np.array(cmat.T)[0], (np.array(cimat.T)
        [0], np.array(cjmat.T)[0])), shape=(N1, N))
    #Imat shold be an N1xN sparse matrix; use it to compute the
        interpolated values.
    u_out = Imat*u
    return u_out

def MSNFDMG_PDE2Solve(param1, param2):
  #Run at the coarser resolution
  xi1, yi1, xb1, yb1, a1, b1, c1, d1, e1, f1, g1, l1, s1,
      MAGIC_NUMBER, uref_flag1, uref1  = param1
  u1,x1,y1,condNo1, resid1, errLS1, bw1, ta1, ts1, c_data1 =
      PDE2_Solve(xi1, yi1, xb1, yb1, a1, b1, c1, d1, e1, f1, g1, l1,
      s1, MAGIC_NUMBER, uref_flag1, uref1)

  #Run at the finer resolution and obtain the FD Matrix itself. This
      does the interpolation and refinement
  xi2, yi2, xb2, yb2, a2, b2, c2, d2, e2, f2, g2, l2, s2,
      MAGIC_NUMBER, uref_flag2, uref2  = param2
  u2,x2,y2, u12, u12r, condNo2, resid2, errLS2, bw2, ta2, ts2,
      c_data2 =  PDE2MG_Solve(xi2, yi2, xb2, yb2, a2, b2, c2, d2, e2
      , f2, g2, l2, s2, uref_flag2, uref2, x1, y1, u1, l1, s1)
  return u2,x2,y2,u12, u12r, condNo2, resid2, errLS2, bw2, ta2, ts2,
      c_data2

#This is the multiscale refinement type of PDE2 Solver.
def PDE2MG_Solve(xi, yi, xb, yb, a, b, c, d, e, f, g, l, s,
    MAGIC_NUMBER, uref_flag, uref, x11, y11, u11, l11, s11):
     a11 = a[0]; a12 = a[1]; a21 = a[2]; a22 = a[3];
     a11_x = a[4]; a12_x = a[5]; a21_y = a[6]; a22_y = a[7];
     x = np.bmat('xi;xb'); y= np.bmat('yi;yb');
     xo = x; yo = y;
     N = x.size;
```

```
    ind_xo = range(xo.size)


#Original code base.
#     #Get the coefficients
#     #c2mat, cxxmat, cxmat, cyymat, cymat, cmat, cimat, cjmat,
    errLSi, norm2, normxx, normx, normyy, normy, normi, bandWidth =
     compute_MSNWts_nn2(x,y,xo,yo,ind_xo,l,s,a11,a12,a21,a22,a11_x,
    a12_x,a21_y,a22_y)
#     timea1 = tm.time()
#     c2mat, cxxmat, cxmat, cyymat, cymat, cmat, cimat, cjmat, errLSi
    , norm2, normxx, normx, normyy, normy, normi, bandWidth =
    get_MSNWts_nn2_Parallel(x,y,xo,yo,l,s,a11,a12,a21,a22,a11_x,a12_x
    ,a21_y,a22_y)
#     timea2 = tm.time()

    #From jsm's code version:
    #Find some far interior point and compute its weights
    #If the nearest neighbourhood does not have a boundary point
    #We call it a far-interior point.
    ni = xi.size
    for ii in range(ni):
        xL, yL, iL, fi = find_nn2(x,y,xo[ii],yo[ii],l,ni)
        if fi == True:
            ax = xo[ii,0]-l;bx = xo[ii,0]+l;
            ay = yo[ii,0]-l;by = yo[ii,0]+l;
            c2FI,cxxFI,cyyFI,cxFI,cyFI,cFI,errLSFI=compute_MSN_weight
                (xL,yL,xo[ii],yo[ii],ax,bx,ay,by,s,a11[ii][0],a12[ii
                ][0],a21[ii][0],a22[ii][0],a11_x[ii][0],a12_x[ii][0],
                a21_y[ii][0],a22_y[ii][0])
            break

    #Get the coefficients
    timea1 = tm.time()
    #c2mat, cxxmat, cxmat, cyymat, cymat, cmat, cimat, cjmat, errLSi,
        norm2, normxx, normx, normyy, normy, normi, bandWidth =
        compute_MSNWts_nn2(x,y,xo,yo,ind_xo,l,s,a11,a12,a21,a22,a11_x
        ,a12_x,a21_y,a22_y, c2FI,cxxFI,cyyFI,cxFI,cyFI,cFI,errLSFI,
        ni)
    c2mat, cxxmat, cxmat, cyymat, cymat, cmat, cimat, cjmat, errLSi,
        norm2, normxx, normx, normyy, normy, normi, bandWidth =
        get_MSNWts_nn2_Parallel(x,y,xo,yo,l,s,a11,a12,a21,a22,a11_x,
        a12_x,a21_y,a22_y, c2FI,cxxFI,cyyFI,cxFI,cyFI,cFI,errLSFI, ni
        )
    timea2 = tm.time()
```

```
c_data_i = (c2mat, cxxmat, cxmat, cyymat, cymat, cmat, cimat,
    cjmat, errLSi, norm2, normxx, normx, normyy, normy, normi,
    bandWidth)
c_data_o = (x,y,xo,yo,l,s,a11,a12,a21,a22,a11_x,a12_x,a21_y,a22_y
    , c2FI,cxxFI,cyyFI,cxFI,cyFI,cFI,errLSFI, ni)
c_data = (c_data_i, c_data_o)

cdata_fname = "PDE2MG_Solve_cdata_%s.cdata" %(MAGIC_NUMBER)
cdata_fobj = gz.open(cdata_fname, 'wb')
pkl.dump(c_data, cdata_fobj, pkl.HIGHEST_PROTOCOL)
cdata_fobj.close

#Create the sparse matrices
S2mat = sps.coo_matrix((np.array(c2mat.T)[0], (np.array(cimat.T)
    [0], np.array(cjmat.T)[0])), shape=(N, N))
Sxxmat = sps.coo_matrix((np.array(cxxmat.T)[0], (np.array(cimat.T
    )[0], np.array(cjmat.T)[0])), shape=(N, N))
Sxmat = sps.coo_matrix((np.array(cxmat.T)[0], (np.array(cimat.T)
    [0], np.array(cjmat.T)[0])), shape=(N, N))
Syymat = sps.coo_matrix((np.array(cyymat.T)[0], (np.array(cimat.T
    )[0], np.array(cjmat.T)[0])), shape=(N, N))
Symat = sps.coo_matrix((np.array(cymat.T)[0], (np.array(cimat.T)
    [0], np.array(cjmat.T)[0])), shape=(N, N))
#set up the diagonal matrices corresponding to b, c, d and e
b1 = b[0]; b2 = b[1];
e1 = e[0]; e2 = e[1];

b1Diag = sps.spdiags(b1.flatten(),0,N,N);
b2Diag = sps.spdiags(b2.flatten(),0,N,N);
e1Diag = sps.spdiags(e1.flatten(),0,N,N);
e2Diag = sps.spdiags(e2.flatten(),0,N,N);
cDiag = sps.spdiags(c.flatten(),0,N,N);
dDiag = sps.spdiags(d.flatten(),0,N,N);

#Set up the FD Matrix.
FDi = S2mat + b1Diag*Sxmat + b2Diag*Symat + cDiag
FDb = e1Diag*Sxmat + e2Diag*Symat + dDiag

#Now extract equations from FDi corresponding to xi,yi and those
    from FDb corresponding
#to xb,yb`
FDi = FDi[0:xi.size][:];
FDb = FDb[xi.size:][:];
FD = sps.bmat([[FDi],[ FDb]]);
FD = FD.tocsc();

#The FD Matrix is setup. Need to balance it.
```

```
row_scli = (norm2 + np.abs(b1)*normx+np.abs(b2)*normy+np.abs(c))
    **(-1.0)
row_sclb = (np.abs(e1)*normx+np.abs(e2)*normy+np.abs(d))**(-1.0)
row_scli = row_scli[0:xi.size]/xi.size**0.5;
row_sclb = row_sclb[xi.size:]/xb.size**0.5;

row_scl = np.bmat('row_scli;row_sclb')
row_scale = sps.spdiags(np.array(row_scl).flatten(),0,N,N, "csr")
    ;

#Not column scaling yet.RHS scaling
f = f[0:xi.size];g = g[xi.size:];

zi = np.zeros([xb.size,1])
rhs1 = np.bmat('f;zi');
zb = np.zeros([xi.size,1])
rhs2 = np.bmat('zb;g');

if uref_flag != -1:
  fgref = FD*uref
  resids = np.abs(fgref-rhs1-rhs2)
  resid = np.max(resids)/np.max(np.abs(rhs1+rhs2))
else:
  resid = 0;

#Row scale the FD Matrix
FD = row_scale*FD;

xo = np.mat(np.random.rand(N, 10))
#Unit normalize xo
for i in range(10):
    xo[:,i] = xo[:,i]/np.max(np.abs(xo[:,i]))
#Compute the RHS
bo = FD*xo
#compute the maximum inf norm
normFD = np.max(np.max(np.abs(bo)))

rhs1 = row_scale*rhs1
rhs2 = row_scale*rhs2

x1 = np.zeros([N, 10])
#Solve the sparse system
timesp1 = tm.time()
LU = spla.dsolve.splu(FD)
#Apply the factors to one RHS at a time.
u1 = LU.solve(np.array(rhs1).flatten())
u2 = LU.solve(np.array(rhs2).flatten())
```

```
    timesp2 = tm.time()

    Fim = (row_scale[0:xi.size,:]*S2mat, row_scale[0:xi.size,:]*
        b1Diag*Sxmat, row_scale[0:xi.size,:]*b2Diag*Symat, row_scale
        [0:xi.size,:]*cDiag)
    Fbm = (row_scale[xi.size:,:]*e1Diag*Sxmat , row_scale[xi.size
        :,:]*e2Diag*Symat , row_scale[xi.size:,:]*dDiag)

    u1r, r1 = iter_refine_FD(4, 3, Fim, Fbm, xi.size, u1, rhs1, LU,
        2)
    u2r, r2 = iter_refine_FD(4, 3, Fim, Fbm, xi.size, u2, rhs2, LU,
        2)

    u = u1+u2
    ur = u1r + u2r

    for i in range(1,11):
        x1[:,i-1] = LU.solve(np.array(xo[:,i-1]).flatten())

    normFDi = np.max(np.max(np.abs(x1)))
    wallis = (np.pi*(N-1)*0.5)**0.5
    condNo = wallis*normFD*normFDi

    errLS =np.max(abs(errLSi))

    #The Multiscale refinmenet part begins here. First interpolate
        the lower resolution
    #solution to higher resolution.
    u12 = MSN_local_interp(x11, y11, x, y, l11, s11, u11)

    u12r, r12 = iter_refine_FD(4, 3, Fim, Fbm, xi.size, u12, rhs1+
        rhs2, LU, 0)

    return ur,x,y, u12r, u12,condNo, resid, errLS, np.max(bandWidth),
         timea2-timea1, timesp2-timesp1
```

Listing B.2: Exterior Laplace Wrapper

```
import numpy as np
import time as tm
import scipy as sp
import scipy.sparse as sps
import scipy.sparse.linalg as spla
import numpy.linalg as la
import pylab as pl
import sympy as sym
import pickle as pkl
```

```
from IPython.kernel import client
from ctypes import *
mkl = cdll.LoadLibrary("clapack.so")
dgesvd = mkl.dgesvd_
from MSNFD_CORE_Split import *
#from enthought.mayavi import mlab

t0=0.0
GRID_FILE = True
#GRID_FILE = False
def eval_coeffs_extPar(x,y, ff, gf, a11f, a12f, a21f, a22f, a11_xf,
    a12_xf, a21_yf, a22_yf, b1f, b2f, cf, df, e1f, e2f, uf, uxf, uyf)
    :
    mec = client.get_multiengine_client()
    mec.execute('import numpy as np')
    mec.execute('from numpy import *')
    mec.push_function(dict(eval_coeffs_ext = eval_coeffs_ext))

    mec.push_function(dict(ff=ff, gf=gf, a11f=a11f, a12f=a12f, a21f=
        a21f, a22f=a22f, a11_xf=a11_xf, a12_xf=a12_xf, a21_yf=a21_yf,
         a22_yf=a22_yf, b1f=b1f, b2f=b2f, cf=cf, df=df, e1f=e1f, e2f=
        e2f, uf=uf, uxf=uxf, uyf=uyf))

    mec.scatter('x',x); mec.scatter('y',y);

    mec.execute('a11,a12,a21,a22, a11_x, a12_x, a21_y, a22_y, b1, b2,
        c, d,e1, e2, f, g, uref = eval_coeffs_ext(x,y, ff, gf, a11f,
        a12f, a21f, a22f, a11_xf, a12_xf, a21_yf, a22_yf, b1f, b2f,
        cf, df, e1f, e2f, uf, uxf, uyf)')

    a11 = mec.gather('a11')
    a12 = mec.gather('a12')
    a21 = mec.gather('a21')
    a22 = mec.gather('a22')
    a11_x = mec.gather('a11_x')
    a12_x = mec.gather('a12_x')
    a21_y = mec.gather('a21_y')
    a22_y = mec.gather('a22_y')
    b1 = mec.gather('b1')
    b2 = mec.gather('b2')
    c = mec.gather('c')
    d = mec.gather('d')
    e1 = mec.gather('e1')
    e2 = mec.gather('e2')
    f = mec.gather('f')
    g = mec.gather('g')
    uref = mec.gather('uref')
```

```
    a = (a11,a12,a21,a22, a11_x, a12_x, a21_y, a22_y);
    b = (b1, b2); e = (e1, e2);

    return a,b,c,d,e,f,g,uref

def eval_coeffs_ext(x,y, ff, gf, a11f, a12f, a21f, a22f, a11_xf,
    a12_xf, a21_yf, a22_yf, b1f, b2f, cf, df, e1f, e2f, uf, uxf, uyf)
    :
    a11 = np.zeros([x.size,1]);
    a12 = np.zeros([x.size,1]);
    a21 = np.zeros([x.size,1]);
    a22 = np.zeros([x.size,1]);
    a11_x = np.zeros([x.size,1]);
    a12_x = np.zeros([x.size,1]);
    a21_y = np.zeros([x.size,1]);
    a22_y = np.zeros([x.size,1]);
    b1 = np.zeros([x.size,1]);
    b2 = np.zeros([x.size,1]);
    c = np.zeros([x.size,1]);
    d = np.zeros([x.size,1]);
    e1 = np.zeros([x.size,1]);
    e2 = np.zeros([x.size,1]);
    f = np.zeros([x.size,1]);
    g = np.zeros([x.size,1]);
    uref = np.zeros([x.size,1]);
#      a11 = a11f(np.array(x), np.array(y))
#      a12 = a12f(np.array(x), np.array(y))
#      a21 = a21f(np.array(x), np.array(y))
#      a22 = a22f(np.array(x), np.array(y))
#      a11_x = a11_xf(np.array(x), np.array(y))
#      a12_x = a12_xf(np.array(x), np.array(y))
#      a21_y = a21_yf(np.array(x), np.array(y))
#      a22_y = a22_yf(np.array(x), np.array(y))
#      b1 = b1f(np.array(x), np.array(y))
#      b2 = b2f(np.array(x), np.array(y))
#     c = cf(np.array(x), np.array(y))
#     f = ff(np.array(x), np.array(y))

    for i in range(x.size):
        a11[i] = a11f(x[i], y[i])
        a12[i] = a12f(x[i], y[i])
        a21[i] = a21f(x[i], y[i])
        a22[i] = a22f(x[i], y[i])
        a11_x[i] = a11_xf(x[i], y[i])
        a12_x[i] = a12_xf(x[i], y[i])
        a21_y[i] = a21_yf(x[i], y[i])
```

```
        a22_y[i] = a22_yf(x[i], y[i])
        b1[i] = b1f(x[i], y[i])
        b2[i] = b2f(x[i], y[i])
        c[i] = cf(x[i], y[i])
        f[i] = ff(x[i], y[i])
        uref[i] = uf(x[i], y[i])
        if np.allclose(x[i], np.pi/2, rtol=1e-4):
          e1[i] = 0.0
          e2[i] = 0.0
          d[i] = 1.0
          g[i] = 0.0;#uxf(x[i], y[i])#Should be zero actually; some
              error here but thats what we get.
        elif np.allclose(x[i], -np.pi/2, rtol=1e-4):
          e1[i] = 0.0
          e2[i] = 0.0
          d[i] = 1.0
          g[i] = 0.0;#-uxf(x[i], y[i])#Should be zero actually; some
              error here but thats what we get.
        elif np.allclose(y[i], np.pi/2, rtol=1e-4):
          e1[i] = 0.0
          e2[i] = 0.0
          d[i] = 1.0
          g[i] = 0.0;#uyf(x[i], y[i])#Should be zero actually; some
              error here but thats what we get.
        elif np.allclose(y[i], -np.pi/2, rtol=1e-4):
          e1[i] = 0.0
          e2[i] = 0.0
          d[i] = 1.0
          g[i] = 0.0;#-uyf(x[i], y[i])#Should be zero actually; some
              error here but thats what we get.
        else:
            e1[i] = e1f(x[i], y[i])
            e2[i] = e2f(x[i], y[i])
            d[i] = df(x[i], y[i])
            g[i] = gf(x[i], y[i])

    return a11,a12,a21,a22, a11_x, a12_x, a21_y, a22_y, b1, b2, c, d,
        e1, e2, f, g, uref

#Test the PDE Solver for an exterior problem
def PDE_ExtTest(Nx,Ny, p, L, asmbFlag, slvFlag):
    #MAGIC="_ExtLaplaceDum_Nx%dNy%dp%dL%d" % (Nx,Ny, p, L)
    #MAGIC="_ExtLaplaceCyl_Nx%dNy%dp%dL%d" % (Nx,Ny, p, L)
    MAGIC="_ExtLaplaceTRDrop_Nx%dNy%dp%dL%d" % (Nx,Ny, p, L)
    GRID ="_GridTRDropAct_Nx%dNy%dp%d" % (Nx, Ny, p)
    ##################################TIMING
        ######################################
```

```
    print "Starting the PDE_ExtTest:%f" % (tm.time() - t0)
    ##################################TIMING
        ######################################

    x,y = sym.symbols('xy', commutative=True)
    #u = sym.sin((10**9)*2*3.14*x+(10**6)*2*3.14*y)
    #u = sym.cos(10.0*sym.tan(x)+10.0*sym.tan(y))/(1+sym.tan(x)**2+
        sym.tan(y)**2)
    u = (sym.tan(x)+sym.tan(y))/(sym.tan(x)**2+sym.tan(y)**2)
    #u = (sym.tan(x)*sym.log((sym.tan(x))**2 + (sym.tan(y))**2)+sym.
        tan(y)*sym.atan(sym.tan(y)/sym.tan(x)))/((sym.tan(x))**2 + (
        sym.tan(y))**2)**2
    #u = 1.0
    #u = sym.cos((sym.tan(x)**2+sym.tan(y)**2)**(0.5))/(sym.tan(x)
        **2+sym.tan(y)**2)**(0.5)
    #u = 1/(1+1000*x**2+1000*y**2)
    #u = 1/(1+1000*(x**2+y-0.3)**2) + 1/(1+1000*(x+y-0.4)**2)+
        1./(1+1000*(x+y**2-0.5)**2) + 1./(1+1000*(x**2+y**2-0.25)**2)
    ######################################################################
    #EQUATION SETUP HERE:
    ######################################################################
    #Standard helmoltz equation
    ######################################################################
    a11 = sym.cos(x)**4; a12 = 0; a21 = 0; a22 = sym.cos(y)**4;
#       a11_x = sym.diff(a11,'x'); a12_x = sym.diff(a12,'x');
#       a21_y = sym.diff(a21,'y'); a22_y = sym.diff(a22,'y');
    a11_x = 0;      a12_x = 0;
    a21_y = 0;      a22_y = 0;
    b1 = -2*(sym.cos(x)**3)*sym.sin(x); b2 = -2*(sym.cos(y)**3)*sym.
        sin(y);
    c = -1.0/(1.0+100*x**2+100*y**2);
    d = 1;
    e1 = 0; e2 = 0;
    ######################################################################
#       f,g = varPDE2_Evalfg(a11, a12, a21, a22, b1, b2, c, d,e1,e2,u)
    #f = a11*sym.diff(sym.diff(u,'x'),'x') + a22*sym.diff(sym.diff(u
        ,'y'),'y') + b1*sym.diff(u,'x') + b2*sym.diff(u,'y') + c*u
    f = c*u;
    #g = e1*sym.diff(u,'x') + e2*sym.diff(u,'y') + d*u
    g=u
    ux = sym.diff(u, 'x');    uy = sym.diff(u, 'y');
    ######################################################################
    ##################################TIMING
        ######################################
    print "Done with Symbolic stuff:%f" % (tm.time() - t0)
    ##################################TIMING
        ######################################
```

```
#Now f and g are available symbolically, need to evaluate them
ff = sym.lambdify((x,y),f)
gf = sym.lambdify((x,y),g)
a11f = sym.lambdify((x,y), a11)
a12f = sym.lambdify((x,y), a12)
a21f = sym.lambdify((x,y), a21)
a22f = sym.lambdify((x,y), a22)
a11_xf = sym.lambdify((x,y), a11_x)
a12_xf = sym.lambdify((x,y), a12_x)
a21_yf = sym.lambdify((x,y), a21_y)
a22_yf = sym.lambdify((x,y), a22_y)
b1f = sym.lambdify((x,y), b1)
b2f = sym.lambdify((x,y), b2)
cf = sym.lambdify((x,y), c)
df = sym.lambdify((x,y), d)
e1f = sym.lambdify((x,y), e1)
e2f = sym.lambdify((x,y), e2)
uf = sym.lambdify((x,y), u)
uxf = sym.lambdify((x,y), ux)
uyf = sym.lambdify((x,y), uy)
##################################TIMING
    ######################################
print "Done with lamdifying:%f" % (tm.time() - t0)
##################################TIMING
    ######################################
#########################################################################
#GEOMETRY SETUP HERE
#########################################################################
#SQUARE REGION
#########################################################################
xo = 0.; yo=0.#Center of the square
width = 1.0; height = 1.0;

if GRID_FILE==False:
    xi1, yi1, xb1, yb1 = generate_grid(Nx, Ny, xo,yo,np.pi,np.pi,
        'regular',p);
    xi2, yi2, xb2, yb2 = generate_grid(Nx/2, Ny/2, xo,yo,width,
       height, 'regular',200.0);
else:
    gridfname = "%s.gz" %(GRID)
    fGrid=gz.open(gridfname,"rb");
    (xi,yi,xb1,xb2,yb1,yb2) = pkl.load(fGrid)
    fGrid.close()

##################################TIMING
    ######################################
print "Done with Input grid generation:%f" % (tm.time() - t0)
```

```
    ##################################TIMING
        ######################################

    if GRID_FILE==False:
        poly_in = np.bmat('xb2 yb2');
        v = np.zeros(xi1.shape) == 1.0;
        for i in range(xi1.size):
            v[i] = point_in_poly(xi1[i], yi1[i], poly_in);

    ##################################TIMING
        ######################################
    print "Done with Input point in polygon:%f" % (tm.time() - t0)
    ##################################TIMING
        ######################################

    if GRID_FILE==False:
        xsi = xi1[~v];      ysi = yi1[~v];
        xsi = xsi.reshape(xsi.size, 1)
        ysi = ysi.reshape(ysi.size, 1)
        thr = np.pi*0.75/(2*Nx)
        xi, yi = cleanup_boundary(xb2, yb2, xsi, ysi, thr)
        gridfname = "%s.gz" %(GRID)
        fGrid=gz.open(gridfname,"wb");
        pkl.dump((xi,yi,xb1,xb2,yb1,yb2),fGrid)
        fGrid.close()


    xb = np.bmat('xb1;xb2');      yb = np.bmat('yb1;yb2');
    x = np.bmat('xi;xb');   y = np.bmat('yi;yb');


#     pl.figure()
#     pl.scatter(np.array(xi).flatten(),np.array(yi).flatten());
#     pl.scatter(np.array(xb).flatten(),np.array(yb).flatten(), c='g')
    ;

    ##################################TIMING
        ######################################
    print "Done with input cleanup boundary:%f" % (tm.time() - t0)
    ##################################TIMING
        ######################################

    #a,b,c,d,e,f,g,uref = eval_coeffs_extPar(x,y, ff, gf, a11f, a12f,
         a21f, a22f, a11_xf, a12_xf, a21_yf, a22_yf, b1f, b2f, cf, df
        , e1f, e2f, uf, uxf, uyf)
    a11,a12,a21,a22, a11_x, a12_x, a21_y, a22_y, b1, b2, c, d,e1, e2,
         f, g, uref = eval_coeffs_ext(x,y, ff, gf, a11f, a12f, a21f,
```

```
        a22f, a11_xf, a12_xf, a21_yf, a22_yf, b1f, b2f, cf, df, e1f,
        e2f, uf, uxf, uyf)
    a = (a11,a12,a21,a22, a11_x, a12_x, a21_y, a22_y);
    b = (b1, b2); e = (e1, e2);

#     mlab.figure()
#     pts = mlab.points3d(np.array(x).flatten(), np.array(y).flatten()
    , np.array(uref).flatten(), scale_mode='none', scale_factor=0.01)
#     mesh =mlab.pipeline.delaunay2d(pts)
#     surf =mlab.pipeline.surface(mesh)

    ##################################TIMING
        ######################################
    print "Done with Eval coeffs:%f" % (tm.time() - t0)
    ##################################TIMING
        ######################################

    #Set up l,s
    l = L/(x.size)**0.5
    s = 15

    if asmbFlag:
        PDE2_Assemble(xi, yi, xb, yb, a, b, c, d, e, f, g, l, s,
            MAGIC,0, uref)

    ##################################TIMING
        ######################################
    print "Done with Assemble:%f" % (tm.time() - t0)
    ##################################TIMING
        ######################################

    if slvFlag:
        u_MSN, x, y, condNo, residue, errLS, maxBand, ta, ts, resids
            = PDE2_Solve(xi.size, xb.size, a, b, c, d, e, f, g, l, s
            , MAGIC, 0, uref)
        err = np.abs(u_MSN.reshape(u_MSN.size,1)-uref)
        errMaxrel = np.max(err)/np.max(uref)
    else:
        (errMaxrel, condNo, residue, errLS, maxBand, u_MSN, x, y, ta,
            ts) = (-1, -1, -1, -1, -1, -1, -1, -1,-1,-1)

    ##################################TIMING
        ######################################
    print "Done with Solve:%f" % (tm.time() - t0)
    ##################################TIMING
        ######################################
```

```
#     pl.figure()
#     pl.scatter(np.array(x).flatten(),np.array(y).flatten());

    return errMaxrel, condNo, residue, errLS, maxBand, u_MSN, x, y,
        ta, ts

def point_poly_parallel(xei1, yei1, poly_in):
    mec = client.get_multiengine_client()
    mec.execute('import numpy as np')
    mec.execute('from numpy import *')
    mec.push(dict(poly_in=poly_in))
    mec.push_function(dict(point_in_poly=point_in_poly))
    mec.push_function(dict(point_poly_unit=point_poly_unit))
    mec.scatter('xei1',xei1);  mec.scatter('yei1',yei1);
    mec.execute('v=point_poly_unit(xei1, yei1, poly_in)')
    v = mec.gather('v')
    return v

def point_poly_unit(xei1, yei1, poly_in):
    v = np.zeros(xei1.shape) == 1.0;
    for i in range(xei1.size):
        v[i] = point_in_poly(xei1[i], yei1[i], poly_in);
    return v

def PDEExt(asmbFlag, slvFlag):
    global t0
    t0 = tm.time()
    print "Beginning!!...";     print tm.ctime();
    NList = np.array([30,100])#,200,500])
    pList = np.array([2.0])
    Llist = np.array([6.0])
    errMaxrel = np.zeros([pList.size, NList.size, Llist.size])
    condNo = np.zeros([pList.size, NList.size, Llist.size])
    residue = np.zeros([pList.size, NList.size, Llist.size])
    errLS = np.zeros([pList.size, NList.size, Llist.size])
    bandwidth = np.zeros([pList.size, NList.size, Llist.size])
    times = np.zeros([pList.size, NList.size, Llist.size])
    timea = np.zeros([pList.size, NList.size, Llist.size])
    timesp = np.zeros([pList.size, NList.size, Llist.size])
    print "Maximum_Relative_Error", "Condition_Number", "Residue", "
        Error_in_LS_System","Bandwidth",  "Wall_time_in_Seconds"
    for iter in range(pList.size):
        for kter in range(Llist.size):
            for jter in range(NList.size):
                PDE2Dump = open('PDEExtTrDrop_Jun11_varcoeff1.data','
                    a')
                t1 = tm.time()
```

```
                    errMaxrel[iter,jter,kter],condNo[iter,jter,kter],
                        residue[iter,jter,kter],errLS[iter,jter,kter],
                        bandwidth[iter, jter, kter], uMSN, x, y, ta, ts =
                         PDE_ExtTest(NList[jter], NList[jter], pList[iter
                        ], Llist[kter], asmbFlag, slvFlag)
                    t2 = tm.time()
                    times[iter,jter,kter] = t2-t1
                    timea[iter,jter,kter] = ta
                    timesp[iter,jter,kter] = ts
                    print NList[jter], Llist[kter], pList[iter],
                        errMaxrel[iter,jter,kter],condNo[iter,jter,kter],
                        residue[iter,jter,kter],errLS[iter,jter,kter],
                        bandwidth[iter, jter, kter], times[iter, jter,
                        kter], ta, ts
                    if slvFlag:
                            PDE2Obj = (NList[jter], pList[iter], Llist[
                                kter], uMSN, x, y, errMaxrel[iter,jter,
                                kter],condNo[iter,jter,kter],residue[iter
                                ,jter,kter],errLS[iter,jter,kter],
                                bandwidth[iter, jter, kter], times[iter,
                                jter, kter], ta, ts)
                            pkl.dump(PDE2Obj, PDE2Dump)
                PDE2Dump.close
    return errMaxrel, condNo, residue, errLS, bandwidth, times, timea
        , timesp

#PDEExt(True, False)
PDEExt(False, True)
```

Listing B.3: Biharmonic Wrapper

```
import numpy as np
import time as tm
import scipy as sp
import scipy.sparse as sps
import scipy.sparse.linalg as spla
import numpy.linalg as la
import pylab as pl
import sympy as sym
import pickle as pkl
from IPython.kernel import client
from ctypes import *
mkl = cdll.LoadLibrary("clapack.so")
dgesvd = mkl.dgesvd_
from MSNFD_CORE_BiHarmonicSP import *

t0=0.0
```

```
def eval_coeffs_extPar(x,y, xGb, yGb, ax, bx, ay, by, ff, gf, a11f,
    a12f, a21f, a22f, a11_xf, a12_xf, a21_yf, a22_yf, b1f, b2f, cf,
    d1f, d2f, e1f, e2f, e3f, e4f, gGf, h1f, h2f, uf, uxf, uyf):
    mec = client.get_multiengine_client()
    mec.execute('import numpy as np')
    mec.execute('import scipy.special as spsp')
    mec.execute('from numpy import *')
    mec.push_function(dict(eval_coeffs_ext = eval_coeffs_ext))

    mec.push_function(dict(ff=ff, gf=gf, a11f=a11f, a12f=a12f, a21f=
        a21f, a22f=a22f, a11_xf=a11_xf, a12_xf=a12_xf, a21_yf=a21_yf,
         a22_yf=a22_yf, b1f=b1f, b2f=b2f, cf=cf, d1f=d1f, d2f=d2f,
        e1f=e1f, e2f=e2f, e3f=e3f,e4f=e4f,gGf=gGf, h1f=h1f, h2f=h2f,
        uf=uf, uxf=uxf, uyf=uyf))

    mec.scatter('x',x); mec.scatter('y',y);
    mec.scatter('xGb',xGb); mec.scatter('yGb',yGb);
    mec.push(dict(ax=ax, bx=bx, ay=ay,by=by))

    mec.execute('a11,a12,a21,a22, a11_x, a12_x, a21_y, a22_y, b1, b2,
        c, d1,d2, e1, e2, e3, e4, f, g, gG, h1, h2, uref=
        eval_coeffs_ext(x,y,xGb, yGb, ax, bx, ay, by, ff, gf, a11f,
        a12f, a21f, a22f, a11_xf, a12_xf, a21_yf, a22_yf, b1f, b2f,
        cf, d1f, d2f, e1f, e2f, e3f, e4f, gGf, h1f, h2f, uf, uxf, uyf
        )')

    a11 = mec.gather('a11')
    a12 = mec.gather('a12')
    a21 = mec.gather('a21')
    a22 = mec.gather('a22')
    a11_x = mec.gather('a11_x')
    a12_x = mec.gather('a12_x')
    a21_y = mec.gather('a21_y')
    a22_y = mec.gather('a22_y')
    b1 = mec.gather('b1')
    b2 = mec.gather('b2')
    c = mec.gather('c')
    d1 = mec.gather('d1')
    d2 = mec.gather('d2')
    e1 = mec.gather('e1')
    e2 = mec.gather('e2')
    e3 = mec.gather('e3')
    e4 = mec.gather('e4')
    f = mec.gather('f')
    g = mec.gather('g')
    gG = mec.gather('gG')
    h1 = mec.gather('h1')
```

```
    h2 = mec.gather('h2')
    uref = mec.gather('uref')

    a = (a11,a12,a21,a22, a11_x, a12_x, a21_y, a22_y);
    b = (b1, b2); e = (e1, e2, e3, e4);
    h = (h1, h2);
    d = (d1, d2);

    return a,b,c,d,e,f,g,gG,h, uref

def eval_coeffs_ext(x,y, xGb, yGb, ax, bx, ay, by, ff, gf, a11f, a12f
    , a21f, a22f, a11_xf, a12_xf, a21_yf, a22_yf, b1f, b2f, cf, d1f,
    d2f, e1f, e2f,e3f, e4f, gGf, h1f, h2f,  uf, uxf, uyf):

    (ax1, ax2) = ax;
    (ay1, ay2) = ay;
    (bx1, bx2) = bx;
    (by1, by2) = by;

    N = x.size;
    a11 = np.zeros([N,1], dtype='single');
    a12 = np.zeros([N,1], dtype='single');
    a21 = np.zeros([N,1], dtype='single');
    a22 = np.zeros([N,1], dtype='single');
    a11_x = np.zeros([N,1], dtype='single');
    a12_x = np.zeros([N,1], dtype='single');
    a21_y = np.zeros([N,1], dtype='single');
    a22_y = np.zeros([N,1], dtype='single');
    b1 = np.zeros([N,1], dtype='single');
    b2 = np.zeros([N,1], dtype='single');
    c = np.zeros([N,1], dtype='single');

    d1 = np.zeros([N,1], dtype='single');
    e1 = np.zeros([N,1], dtype='single');
    e2 = np.zeros([N,1], dtype='single');
    f = np.zeros([N,1], dtype='single');
    g = np.zeros([N,1], dtype='single');
    h1 = np.zeros([N,1], dtype='single');
    h2 = np.zeros([N,1], dtype='single');
    uref = np.zeros([N,1], dtype='single');

    for i in range(N):
        a11[i] = a11f(x[i], y[i])
        a12[i] = a12f(x[i], y[i])
        a21[i] = a21f(x[i], y[i])
        a22[i] = a22f(x[i], y[i])
        a11_x[i] = a11_xf(x[i], y[i])
```

```
        a12_x[i] = a12_xf(x[i], y[i])
        a21_y[i] = a21_yf(x[i], y[i])
        a22_y[i] = a22_yf(x[i], y[i])
        b1[i] = b1f(x[i], y[i])
        b2[i] = b2f(x[i], y[i])
        e1[i] = e1f(x[i], y[i])
        e2[i] = e2f(x[i], y[i])
        c[i] = cf(x[i], y[i])
        d1[i] = d1f(x[i], y[i])
        f[i] = ff(x[i], y[i])
        g[i] = gf(x[i], y[i])
        h1[i] = h1f(x[i], y[i])
        h2[i] = h2f(x[i], y[i])
        uref[i] = uf(x[i], y[i])

    nG = xGb.size;
    e3 = np.zeros([nG,1], dtype='single');
    e4 = np.zeros([nG,1], dtype='single');
    gG = np.zeros([nG,1], dtype='single');
    d2 = np.zeros([nG,1], dtype='single');

    for i in range(nG):
        if abs(xGb[i] - ax1) < 1e-4:
            e3[i] = 1.0;
            gG[i] = uxf(xGb[i], yGb[i]);
        elif abs(xGb[i] - bx1) < 1e-4:
            e3[i] = 1.0;
            gG[i] = uxf(xGb[i], yGb[i]);
        elif abs(yGb[i] - ay1) < 1e-4:
            e4[i] = 1.0;
            gG[i] = uyf(xGb[i], yGb[i]);
        elif abs(yGb[i] - by1) < 1e-4:
            e4[i] = 1.0;
            gG[i] = uyf(xGb[i], yGb[i]);
        elif abs(xGb[i] - ax2) < 1e-2:
            e3[i] = 1.0;
            gG[i] = uxf(xGb[i], yGb[i]);
        elif abs(xGb[i] - bx2) < 1e-2:
            e3[i] = 1.0;
            gG[i] = uxf(xGb[i], yGb[i]);
        elif abs(yGb[i] - ay2) < 1e-2:
            e4[i] = 1.0;
            gG[i] = uyf(xGb[i], yGb[i]);
        elif abs(yGb[i] - by2) < 1e-2:
            e4[i] = 1.0;
            gG[i] = uyf(xGb[i], yGb[i]);
```

```
    return a11,a12,a21,a22, a11_x, a12_x, a21_y, a22_y, b1, b2, c, d1
        ,d2, e1, e2,e3,e4, f, g, gG, h1, h2, uref

#Test the PDE Solver
def PDE2_Test(Nx,Ny, p, L, asmbFlag, slvFlag):
    #MAGIC="_ExtLaplaceDum_Nx%dNy%dp%dL%d" % (Nx,Ny, p, L)
    MAGIC="_BH1_Nx%dNy%dp%dL%d" % (Nx,Ny, p, L)
    GRID ="_GridBH1_Nx%dNy%dp%d" % (Nx, Ny, p)
#       MAGIC="_BH1TwB_Nx%dNy%dp%dL%d" % (Nx,Ny, p, L)
#       GRID ="_GridBH1TwB_Nx%dNy%dp%d" % (Nx, Ny, p)
    ##################################TIMING
        ########################################
    print "Starting the PDE ExtTest:%f" % (tm.time() - t0)
    ##################################TIMING
        ########################################

    x,y = sym.symbols('xy', commutative=True)
    #u = 1/(1+1000*(x**2+y-0.3)**2) + 1/(1+1000*(x+y-0.4)**2)+
        1./(1+1000*(x+y**2-0.5)**2) + 1./(1+1000*(x**2+y**2-0.25)**2)
    u = 1/(1+x**2+y**2)
    ux = sym.diff(u, 'x');      uy = sym.diff(u, 'y');
    ######################################################################
    #EQUATION SETUP HERE:
    ######################################################################
    #Standard helmoltz equation
    ######################################################################
    a11 = 0.; a12 = 0; a21 = 0; a22 = 0.;
    a11_x = sym.diff(a11,'x');      a12_x = sym.diff(a12,'x');
    a21_y = sym.diff(a21,'y');      a22_y = sym.diff(a22,'y');
    b1 = 0; b2 = 0;
    c = 1.0;
    d1 = 1.; d2=1.;
    e1 = 0.; e2 = 0.; e3 = 0.; e4 = 0.;
    h1 = 1.; h2 = 1.;
    ######################################################################
    f, g, gG = varPDE2_Evalfg(h1, h2, a11, a12, a21, a22, b1, b2, c,
        d1,d2, e1,e2,e3, e4, u)
    ######################################################################
    ######################################################################
    ##################################TIMING
        ########################################
    print "Done with Symbolic stuff:%f" % (tm.time() - t0)
    ##################################TIMING
        ########################################
    #Now f and g are available symbolically, need to evaluate them
    ff = sym.lambdify((x,y),f)
    gf = sym.lambdify((x,y),g)
```

```
gGf = sym.lambdify((x,y),gG)
h1f = sym.lambdify((x,y), h1)
h2f = sym.lambdify((x,y), h2)
a11f = sym.lambdify((x,y), a11)
a12f = sym.lambdify((x,y), a12)
a21f = sym.lambdify((x,y), a21)
a22f = sym.lambdify((x,y), a22)
a11_xf = sym.lambdify((x,y), a11_x)
a12_xf = sym.lambdify((x,y), a12_x)
a21_yf = sym.lambdify((x,y), a21_y)
a22_yf = sym.lambdify((x,y), a22_y)
b1f = sym.lambdify((x,y), b1)
b2f = sym.lambdify((x,y), b2)
cf = sym.lambdify((x,y), c)
d1f = sym.lambdify((x,y), d1)
d2f = sym.lambdify((x,y), d2)
e1f = sym.lambdify((x,y), e1)
e2f = sym.lambdify((x,y), e2)
e3f = sym.lambdify((x,y), e3)
e4f = sym.lambdify((x,y), e4)
uf = sym.lambdify((x,y), u)
uxf = sym.lambdify((x,y), ux)
uyf = sym.lambdify((x,y), uy)
##################################TIMING
    ####################################
print "Done with lamdifying:%f" % (tm.time() - t0)
##################################TIMING
    ####################################
#######################################################################
#GEOMETRY SETUP HERE
#######################################################################
#SQUARE REGION
#######################################################################
xo = 0.; yo=0.#Center of the square
width = 1.0; height = 1.0;

try:
    gridfname = "%s.gz" %(GRID)
    fGrid=gz.open(gridfname,"rb");
    (xi,yi,xb,yb,xG,yG,xGb,yGb,nn_i) = pkl.load(fGrid)
    fGrid.close()
except IOError:
    print "Grid file does not exist, generating one..."
    gridfname = "%s.gz" %(GRID)
    xi1, yi1, xb1, yb1 = generate_grid(Nx,Ny,xo,yo,width,height,
        'regular',p);
```

```
        xi2, yi2, xb2, yb2 = generate_grid(Nx,Ny,xo,yo,0.5,0.5, '
            regular',p);

        poly_in = np.bmat('xb2 yb2');
        #v = np.zeros(xi1.shape) == 1.0;
        #for i in range(xi1.size):
        #    v[i] = point_in_poly(xi1[i], yi1[i], poly_in);
        v = point_poly_parallel(xi1, yi1, poly_in)
        xsi = xi1[~v];     ysi = yi1[~v];
        xsi = xsi.reshape(xsi.size, 1)
        ysi = ysi.reshape(ysi.size, 1)
        thr = 0.75*width/Nx
        xi, yi = cleanup_boundary(xb2, yb2, xsi, ysi, thr)
        xb = np.array(np.bmat('xb1;xb2')); yb = np.array(np.bmat('
            yb1;yb2'));
        xG, yG, nn_i = find_Ghost_Layer(xi, yi, xb, yb, L*width/(2*Nx
            )**0.5)
        nG = xG.size;
        xG = xG.reshape(nG, 1)
        yG = yG.reshape(nG, 1)
        xGb = np.zeros([nG, 1]);yGb = np.zeros([nG, 1]);
        #Find the nearest corresponding boundary points.
        for iter in range(nG):
            dk = (xG[iter]-xb)**2+(yG[iter]-yb)**2
            dk = dk.reshape(dk.size, 1)
            ibk = np.argsort(dk,axis=0)
            xGb[iter] = xb[ibk[0]]; #Picks the closest one
            yGb[iter] = yb[ibk[0]];
        fGrid=gz.open(gridfname,"wb");
        pkl.dump((xi,yi,xb,yb,xG,yG,xGb,yGb,nn_i), fGrid)
        fGrid.close()

    ##################################TIMING
        ########################################
    print "Done with Input grid generation:%f" % (tm.time() - t0)
    ##################################TIMING
        ########################################
#       pl.figure()
#       pl.scatter(np.array(xi).flatten(), np.array(yi).flatten(),
    marker='x')
#       pl.scatter(np.array(xb).flatten(), np.array(yb).flatten(),
    marker='o', c='g')
#       pl.scatter(np.array(xG).flatten(), np.array(yG).flatten(),
    marker='o', c='b')
#       pl.scatter(np.array(xGb).flatten(), np.array(yGb).flatten(),
    marker='o', c='y')
```

```
#     pl.legend(('Interior point', 'Boundary Point', 'Ghost Interior
    ', 'GhostBoundary Point'), loc="lower right", bbox_to_anchor
    =(0.5,0.5))
#     return -1

    x = np.bmat('xi;xb'); y = np.bmat('yi;yb');
    ni = xi.size; nb = xb.size;

    N = ni+nb;
    ax1 = xo - width/2.0;     bx1 = xo + width/2.0;
    ay1 = yo - height/2.0;    by1 = yo + height/2.0;

    ax2 = xo - 0.25;     bx2 = xo  + 0.25;
    ay2 = yo - 0.25;     by2 = yo  + 0.25;

#     print "%1.6e,%1.6e" %(ax1, ay1)
#     print "%1.6e,%1.6e" %(ax2, ay2)
#     print "%1.6e,%1.6e" %(bx1, by1)
#     print "%1.6e,%1.6e" %(bx2, by2)

    ax = (ax1, ax2);
    bx = (bx1, bx2);
    ay = (ay1, ay2);
    by = (by1, by2);

    a,b,c,d,e,f,g,gG,h,uref = eval_coeffs_extPar(x,y, xGb, yGb, ax,
        bx, ay, by, ff, gf, a11f, a12f, a21f, a22f, a11_xf, a12_xf,
        a21_yf, a22_yf, b1f, b2f, cf, d1f, d2f, e1f, e2f, e3f, e4f,
        gGf, h1f, h2f, uf, uxf, uyf)

#     pl.figure(); pl.plot(e[2]+e[3])
    zp = (e[2]+e[3] == 0.0)
    if np.any(zp):
        print "incorrect Ghost setup"
        return -1

#     print zp
#     pl.figure(); pl.scatter(xGb[zp], yGb[zp]); return -1
    ##################################TIMING
        ########################################
    print "Done with Eval coeffs:%f" % (tm.time() - t0)
    ##################################TIMING
        ########################################

    #Set up l,s
    l = L/(x.size)**0.5
    s = 15
```

```
    if asmbFlag:
        PDE2_Assemble(xi, yi, xb, yb, xG, yG, xGb, yGb, nn_i, h, a, b
            , c, d, e, f, g, gG, l, s, MAGIC, 0, uref)

    ##################################TIMING
       ######################################
    print "Done with Assemble:%f" % (tm.time() - t0)
    ##################################TIMING
       ######################################

    if slvFlag:
        u_MSN, x, y, condNo, residue, errLS, maxBand, ta, ts, resids
           = PDE2_Solve(xi, yi, xb, yb, xG, yG, xGb, yGb, nn_i, h,
           a, b, c, d, e, f, g, gG, l, s, MAGIC, 0, uref)
        err = np.abs(u_MSN.reshape(u_MSN.size,1)-uref)
        errMaxrel = np.max(err)/np.max(uref)
    else:
        (errMaxrel, u_MSN, x, y, condNo, residue, errLS, maxBand, ta,
            ts) = (-1, -1, -1, -1, -1, -1, -1, -1, -1,-1)

    return errMaxrel, condNo, residue, errLS, maxBand, u_MSN, x, y,
       ta, ts

def PDE2(asmbFlag, slvFlag):
    global t0
    t0 = tm.time()
    print "Beginning!!...";     print tm.ctime();
    NList = np.array([100])
    #Used for Dec13, 14th 1st set of data
    #pList = np.array([1.0,2.0,20.0])
    pList = np.array([2.0])
    Llist = np.array([3.0])
    errMaxrel = np.zeros([pList.size, NList.size, Llist.size])
    condNo = np.zeros([pList.size, NList.size, Llist.size])
    residue = np.zeros([pList.size, NList.size, Llist.size])
    errLS = np.zeros([pList.size, NList.size, Llist.size])
    bandwidth = np.zeros([pList.size, NList.size, Llist.size])
    times = np.zeros([pList.size, NList.size, Llist.size])
    timea = np.zeros([pList.size, NList.size, Llist.size])
    timesp = np.zeros([pList.size, NList.size, Llist.size])

    print "Maximum_Relative_Error", "Condition_Number", "Residue", "
       Error_in_LS_System","Bandwidth",  "Wall_time_in_Seconds"
    for iter in range(pList.size):
        for jter in range(NList.size):
            for kter in range(Llist.size):
              PDE2Dump = open('BH1_May3_Dum.data','a')
```

```
                t1 = tm.time()
                errMaxrel[iter,jter,kter],condNo[iter,jter,kter],
                    residue[iter,jter,kter],errLS[iter,jter,kter],
                    bandwidth[iter, jter, kter], uMSN, x, y, ta, ts =
                    PDE2_Test(NList[jter], NList[jter], pList[iter],
                    Llist[kter], asmbFlag, slvFlag)
                t2 = tm.time()
                times[iter,jter,kter] = t2-t1
                timea[iter,jter,kter] = ta
                timesp[iter,jter,kter] = ts
                print NList[jter], pList[iter], Llist[kter], errMaxrel[
                    iter,jter,kter],condNo[iter,jter,kter],residue[iter
                    ,jter,kter],errLS[iter,jter,kter], bandwidth[iter,
                    jter, kter], times[iter, jter, kter], ta, ts
                PDE2Obj = (NList[jter], pList[iter], Llist[kter], uMSN,
                     x, y, errMaxrel[iter,jter,kter],condNo[iter,jter,
                    kter],residue[iter,jter,kter],errLS[iter,jter,kter
                    ], bandwidth[iter, jter, kter], times[iter, jter,
                    kter], ta, ts)
                pkl.dump(PDE2Obj, PDE2Dump)
                PDE2Dump.close
    return errMaxrel, condNo, residue, errLS, bandwidth, times, timea
        , timesp

#PDE2(True, True) #Run with parallel qsub
PDE2(False, True) #Run with Largemem qsub
```