Investigating Schwarz Domain Decomposition Based Preconditioners for Efficient Geophysical Electromagnetic Field Simulation

by

© *Benjamin R. Kary*

A thesis submitted to the

School of Graduate Studies

in partial fulfilment of the

requirements for the degree of

Master of *Science*

Department of *Earth Science*

Memorial University of Newfoundland

May 2017

St. John's Newfoundland & Labrador

Abstract

In this thesis, I researched and implemented a number of Schwarz domain decomposition algorithms with the intent of finding an efficient method to solve the geophysical EM problem. I began by using finite difference and finite element discretizations to investigate the domain decomposition algorithms for the Poisson problem. I found that the Schwarz methods were best used as a preconditioner to a Krylov iteration. The optimized Schwarz (OS) preconditioner outperformed the related restricted additive Schwarz (RAS) preconditioner and both of the local and global OS fixed point iterations. Using finite differences the OS preconditioner performed much better than the RAS preconditioner, but using finite element in parallel with the FEniCS assembly library, their performance was similar. The FEniCS library automatically partitions the global mesh into subdomains and produces irregular partition boundaries. By creating a serial rectangular subdomain code in FEniCS, I regained the benefit of the OS preconditioner, suggesting that the irregular partitioning scheme was detrimental to the convergence behaviour of the OS preconditioner. Based on my work for the Poisson problem, I decided to attempt both a RAS and OS preconditioned GMRES iteration for the electromagnetic problem. Due to the unstructured meshes and source/receiver refinement used in EM modelling I could not avoid the irregular mesh partitioning, and the OS preconditioner lagged the RAS preconditioner in terms of iteration count. On the bright side, the RAS preconditioner worked very well, and outperformed any of the preconditioners bundled with PETSc in terms of both iteration count and time to solution.

Acknowledgements

I would like to thank, first and foremost, my excellent co-supervisors: Colin G. Farquharson, and Ronald D. Haynes. You both seemed to find that perfect balance of guiding the ship while allowing me to do the actual steering. I would also like to thank my third committee member Alison E. Malcolm who made several useful suggestions during my annual committee meetings.

I owe a debt of gratitude to Scott P. MacLachlan, who provided me with two excellently taught courses in mathematics related to my topic of research, and who assisted me with my work on several occasions. I also greatly appreciated the scholarly advice I received from two very knowledgable researchers in the field of domain decomposition: Victorita Dolean and Felix Kwok – thank you.

Many thanks to School of Graduate Studies for their financial support through the graduate fellowship program, and to Chevron Canada for their financial support through the Rising Star Scholarship.

Finally, I would like to thank my friends and family. I could not have done this without the support of my lovely wife, Amy – thank you for driving 7,536.3 km from Nanaimo, BC to St. John's NL to let me pursue my dreams! I can't forget the huge contribution my parents made to this effort by filling half their garage and basement with our stuff for two years. Nor can I forget the spiritual support I received from the amazingly beautiful province of Newfoundland and the incredible group of happy fun time friends that I met there.

Contents

Abstract ii

Acknowledgements iii

List of Tables vii

List of Figures ix

1 Introduction 1

1.1 Motivation 1

1.2 State of the forward modelling problem 6

2 Poisson Problem 17

2.1 Poisson problem as a test case 17

2.2 Schwarz Domain Decomposition Methods 18

2.2.1 Historical Perspective 18

2.2.2 Connection to modern Schwarz DD 19

2.2.3 Discrete Schwarz Domain Decomposition 26

2.2.4 Optimized Schwarz Domain Decomposition 33

2.3 Finite Difference 38
2.3.1 One dimensional subdomain problems 38
2.3.2 Two dimensional subdomain problems 49
2.3.3 Two dimensional optimized Schwarz using the auxilliary variable method 57
2.3.4 Two dimensional optimized Schwarz as a preconditioned global iteration 61
2.4 Finite element method 65
2.4.1 Two dimensional subdomain problems 65
2.4.2 Two dimensional optimized Schwarz using the auxilliary variable method 75
2.4.3 Parallel two dimensional optimized Schwarz as a preconditioned global iteration 81

3 Electromagnetic Problem **95**
3.1 Formulation 96
3.2 Finite Element method 100
3.3 FEniCS implementation 104
3.3.1 External mesh generation 104
3.3.2 Finite element assembly for the EM problem 106
3.3.3 Optimized Schwarz preconditioner 109

4 Conclusions **130**

Bibliography **133**

A One dimensional finite difference domain decomposition with equal subdomains **145**

B Using a property of the Kronecker product to extend one dimensional discretizations into two dimensions **149**

C Finite element assembly routines **154**

D Unstructured mesh creation through GMESH geometry files **178**

E Preprocessing script HDF5.py **181**

List of Tables

2.1 Robin Boundary Condition Error for forward difference in the left subdomain 47
2.2 Robin Boundary Condition Error for centered difference in the left subdomain 47
2.3 Robin Boundary Condition Error for forward difference in the right subdomain 48
2.4 Robin Boundary Condition Error for centered difference in the right subdomain 48
2.5 Robin Boundary Condition Error for forward difference in the left subdomain 55
2.6 Robin Boundary Condition Error for centered difference in the left subdomain 55
2.7 Robin Boundary Condition Error for backward difference in the right subdomain 56
2.8 Robin Boundary Condition Error for centered difference in the right subdomain 56

2.9 Robin Boundary Condition Error for finite elements using FEniCS and my own assembly code 75

List of Figures

2.1 Two-subdomain decomposition similar to that used in Schwarz (1870) to investigate existence and uniqueness of the Laplace equation over arbitrary domains. 18

2.2 χ_1 and χ_2 where χ_1 is scaled to permit u_1 to contribute its full solution within the overlap and χ_2 correspondingly scaled so that u_2 contributes nothing (top) and where χ_1 and χ_2 scale u_1 and u_2 so that they contribute the average of the two solution in the overlap (bottom). . . . 22

2.3 A simple decomposition of the 1D mesh into two subdomains 27

2.4 A simple decomposition of the 1D mesh into two non-overlapping subdomains 40

2.5 Function u and f used in the one dimensional experiments 40

2.6 Two dimensional non-overlapping mesh partitioning into two subdomains Ω_1 and Ω_2 with interface Γ 50

2.7 Function u and f used in the two dimensional experiments 52

2.8 Coefficients involved in the application of the Dirichlet and Robin boundary equations 54

2.9 Optimal α search for a 41×41 point global mesh. 58

2.10 Error contraction with optimal $\alpha = 12$ with error measured by $\|u_{glob} - u_{DD}\|_2 / \|u_{glob}\|_2$. 59
2.11 Optimal α grid dependence where h is the space between nodes in x and y dimensions 60
2.12 Two dimensional overlapping mesh partitioning into two subdomains Ω_1 and Ω_2 with interfaces Γ_1 and Γ_2 61
2.13 Comparison of the Optimized Schwarz method as a local iteration via the auxilliary variable method (ORASAV), a preconditioner for the global fixed point iteration (ORASPFP), and a preconditioner for the global Krylov iteration (ORASPGMRES), as well as the RAS preconditioner for the global Krylov iteration (RASPGMRES). The global preconditioned iterations all naturally measure error with the residual vector, by $\|Ax - b\|_2 / \|b\|_2$, so the auxiliary variable code was altered to comply with this error format 64
2.14 Reference triangle transformation 69
2.15 Finite element triangulation of the unit square 72
2.16 Optimal α search 78
2.17 Convergence rate for optimized Schwarz using FEniCS for assembly with $\alpha = 25$ 79
2.18 Optimal α grid dependence 80
2.19 Parallel mesh generation 82
2.20 Shared Vertices 88

2.21 Comparison between the local iteration using the auxiliary variable methos (ORASAV), global fixed point iteration (ORASFP), global ORAS preconditioned GMRES iteration (ORASPGMRES), and global RAS preconditioned GMRES iteration (RASPGMRES). Square subdomain results are included to demonstrate the adverse effect of the METIS partitioning strategy on the iteration count (ORASPGMRES square, RASPGMRES square) 93

3.1 Two dimensional mesh with same refinement and partitioning strategy as the three dimensional mesh used to compare the RAS and MUMPS solutions. 116

3.2 Same mesh as Figure 3.1 but zoomed in on refinement area. 117

3.3 same mesh as Figure 3.1 but zoomed in to a scale that shows the individual cells in refinement area. 118

3.4 Predicted $Re\{H_x\}$ data along y dimension for an x directed dipole. . 119

3.5 Predicted $Im\{H_x\}$ data along y dimension for an x directed dipole. . 120

3.6 Predicted $Re\{E_z\}$ data along y dimension for an x directed dipole. . 121

3.7 Predicted $Im\{E_z\}$ data along y dimension for an x directed dipole. . 122

3.8 Comparison of preconditioner performance. 126

3.9 Wall clock timings for increasing number of processors. 127

3.10 Grid size dependent iteration count. 128

3.11 Grid size dependent wall clock time. 129

A.1 A simple decomposition of the 1D mesh into two subdomains 146

B.1 The elements of the mesh that form the matrix U 150

Chapter 1

Introduction

1.1 Motivation

Data collection practices in the field of electromagnetic (EM) geophysics have outpaced data processing technologies. Practitioners of the field would like to use data, collected using the latest technologies in marine, ground, and airborne equipment, to recover the earth model that best describes the volume of earth under study. Inversion algorithms exist today that can model such data, but data collection practices often pose a challenge for existing algorithms. Marine and airborne survey configurations energize the earth from many sources while traversing the earth, and recording data nearly continuously. This combination can very easily create datasets with tens of millions of data points. The amount of data collected easily becomes prohibitively large to allow inversion within a reasonable time frame.

The bottleneck for these inversions is obtaining the solution of the forward problem. The solution of the forward problem consists of computing the fields given a

particular earth model, and it requires solving a large and often ill-conditioned linear system of equations. Not only must the iterative inversion algorithm solve the forward problem at every iteration, but it must do so for all of the source positions at every iteration. In the following two paragraphs, I will describe a case study each in both the mining and oil and gas industries to demonstrate the cost of real-world inversion.

Yang et al. (2014) developed a strategy for reducing the cost of inversion, and applied it to a dataset collected over the Mt. Milligan porphyry copper deposit. At Mt. Milligan, the general character of the deposit is already established through geological tools and previous geophysical studies. However, improvements in algorithmic design could allow inversions to be carried out using more cells at higher accuracies; thereby, also improving the understanding of the deposit at finer scales, and at depths beyond the reach of geological tools. The strategy employed by Yang et al. (2014) reduced the overall cost of the inversion in two ways. First, they tackled the size of each forward problem by restricting it to a local mesh based on the 'footprint' of each source. Second, they reduced the number of forward problems by adaptively refining the number of source positions to reach a target data misfit throughout a series of inversion iterations in coarse to fine meshes. They began with 14362 soundings of time domain EM data collected over fourteen 2.5km long lines centered over the known deposit. Their strategy only used 744 of the total soundings. They used horizontal and vertical cell sizes of 50 and 20 metres, resulting in a 443520 cell global mesh. The total time for the inversion process was 4.3hrs using 24 processors to hit a normalized data misfit of 19, while a previous algorithm took 18hrs for a subset of the same data using the same number of processors to reach a misfit of 17.

That is a substantial reduction, but potentially at the expense of model fidelity (a misfit of 19 is rather high). They cite that it is a well known fact that modelling every sounding is not necessary, but the point at which downsampling causes a loss in model accuracy is certainly less well known. Estimating the 'footprint' of each transmitter also carries the potential for modelling inaccuracies. It would be desirable to keep as many transmitter locations as possible, and limit the forward modelling mesh in terms of both overall size and resolution as little as possible. For now, this kind of drastic downsampling is at least partially justified by the fact that the sampling rate of the receiving coil generates an along-line spacing so much greater than the across-line spacing as dictated by surveying cost. Future technologies may improve the economics of surveying at fine scales in all dimensions leading to higher data density overall, and reducing the disparity between along-line and across-line sampling rates. If or when this occurs there will likely be much less of an impetus to drastically downsample the number of soundings in favor of recovering higher resolution models using more data points. To give a sense of the cost of inverting a large dataset such as this, I can borrow a result from Yang et al. (2014). In order to demonstrate the scalability of their algorithm, they produced an equation to estimate the total time to solution, and plotted the results for a 10000 sounding survey. To give a comparison in line with the computer power used for their 744 sounding result – they predict that 24 CPUs would take somewhere in the range of 10^4 to 10^5s, or roughly 14 hrs. This cost would be incurred for each Conjugate Gradient (CG) iteration used to solve each model update. If they require ten CG iterations for each of ten Gauss Newton model updates, then the total time would be 58 days.

To find an example of an application requiring that level of data density right

now, I need not look any further than the oil and gas industry. In marine CSEM applications, a survey boat has the ability to tow an array of receivers, making the disparity between along line and across-line sampling density much less pronounced. Additionally, since many receivers at once are sensitive to each transmitter, the moving 'footprint' method may not be applied. To get a sense of the cost of an inversion in this setting, I can make an estimation from the numbers predicted by Ryhove et al. (2017), who provide a measure of the complexity of a state of the art forward problem solver for the marine CSEM problem. Ryhove et al. (2017) compared two of the most promising forward problem algorithms on the SEAM model – an industry standard thought to represent a realistic marine CSEM exploration target. They found that a multigrid preconditioned iterative solver beat a sparse low-rank approximated multifrontal direct solver, and accurately solved the forward problem for 3784 transmitter – receiver pairs in 3141 seconds. I can take that solution time and multiply it by three to get the total cost for their frequency domain algorithm, and then multiply again by 10 for the number of Conjugate Gradient iterations, and again by 10 for the Gauss-Newton iterations, and I get roughly 11 days. Eleven days is a long time to wait for a solution, particularly when a few trials may be required to refine the inversion parameters and find a satisfactory earth model. Data inversion can be considered an advanced processing step. As alluded to in the prior two examples, a number of industries currently rely on EM data. With or without inversion, EM surveying is being used to help characterize the earth in a variety of settings.

Mineral exploration professionals have quite successfully used the EM method in their search for economic concentrations of minerals (Nabighian and Macnae, 1991). In this setting, the EM method often directly indicates the highly conductive ore

minerals that exploration professionals seek. However, when disseminations of ore minerals fail to raise the bulk conductivity of the host rock enough to make the deposit "visible" to the electromagnetic method, electromagnetic data may still prove useful in searching for mineralization through indicators like conductive alteration halos and fault zones.

The oil and gas industry has recently adopted the EM method as a complement to the seismic method for its improved sensitivity to fluids. Electromagnetic data can be used to search for oil and gas deposits directly, but seismic surveys are often preferred for their higher resolution. The recent rise in popularity of the EM method is due to the discovery that the seismic velocity of a reservoir is highly susceptible to trace amounts of fizz gas in the pore fluid. The anomalous velocities recorded in regions containing fizz gas can falsely suggest the presence of hydrocarbons. Marine controlled-source EM (CSEM) has proven to be a useful tool for avoiding such false detections, since reservoir conductivity, as opposed to seismic velocity, is mostly unaffected by small amounts of gas (Constable, 2010).

There are also a large number of lesser known applications such as geothermal exploration (Muñoz, 2014), engineering (Li et al., 2014), environmental monitoring (Christensen and Halkjær, 2014), and hydrogeology (Sapia et al., 2014). Everett (2012) reviews advances in some of these, as well as applications in unexploded ordnance (UXO) detection, soils and agriculture, archaeology, hazards, and climate. Regardless of the application, EM data will always require some degree of processing to be useful. The following section highlights several past efforts to improve the forward modelling of EM data towards the ultimate goal of robust and rapid data inversion.

1.2 State of the forward modelling problem

The geophysical EM forward problem uses the low frequency, or quasistatic approximation. In theory, the low frequency assumption is used to eliminate one of the terms in Maxwell's equations in the second order frequency domain form. In practise, the low frequency assumption means that the equations produce solutions that resemble diffusion phenomena moreso than wave phenomena. There are many ways of approaching the forward problem, all of which seek to find a solution to Maxwell's equations in the quasistatic regime by translating the continuous equations into a discrete set of equations, and each has their own particular set of advantages and disadvantages. In this section, I will provide a brief overview of the work done on the geophysical EM modelling problem. For a more complete overview, see Börner (2010). I will classify the approaches first into two categories: those that use time integration and those that transform the equations into the frequency domain.

Time integration techniques can be categorized by the choice of time stepping and spatial discretization methods. Time stepping methods that rely on information from the current timestep in order to update are known as explicit time integration schemes. Those that require the solution of a linear system of equations to update are labelled as implicit. Explicit schemes would appear to have a lower cost in terms of algorithmic complexity over implicit schemes requiring linear system solves, however analysis has shown that a small time step must be used in order to solve Maxwell's equations using an explicit method (Ascher, 2008). In an attempt to overcome this restriction, some have used an explicit modified DuFort-Frankel method (Wang and Hohmann, 1993; Commer and Newman, 2004), but the current trend seems to be toward implicit

methods using backward Euler (Haber et al., 2007; Um et al., 2010; Fu et al., 2015). Finite Difference (FD), Finite Volume (FV) and Finite Element (FE) may all be employed to discretize the spatial component within time integration schemes. Finite Difference Time Domain (FDTD) techniques have long been the standard (Wang and Hohmann, 1993; Commer and Newman, 2004), but more recently Finite Element Time Domain (FETD) (Um et al., 2010; Fu et al., 2015) and Finite Volume Time Domain (FVTD) (Haber et al., 2007) have been investigated. The main advantage of the FE and FV spatial discretizations is improved meshing of complex structures through tetrahedral, OcTree, and hexahedral meshes, and the ability to adapt mesh size in regions with high or low complexity. FD, FV, and FE discretization techniques can also be applied to solve Maxwell's equations purely in the spatial dimension using Fourier transforms.

Purely spatial discretizations, often called time-harmonic, can be used to avoid time stepping complications described above, but in doing so, they expose other complications. By taking the Fourier transform of Maxwell's equations, and assuming the transformation $\frac{\partial}{\partial t} = i\omega$, the extra dimension of time is avoided, in a way. Time domain data can be recovered later through inverse Fourier transform of a number of solutions computed at different frequencies. As with the time domain discretizations, the finite difference method for the time-harmonic problem represents the old guard, with more effort currently being expended investigating alternative finite volume and finite element techniques for their improved flexibility. Integral equation methods (Wannamaker et al., 1984; Lajoie and West, 1977; Newman et al., 1986) are another class of method used to model time-harmonic EM fields in complex geometries, and which have a long history. They continue to be improved upon (Avdeev and Knizhnik,

2009), despite the limitation that they involve solving a dense linear algebra problem.

Traditional time harmonic discretizations result in poorly conditioned matrices, requiring new measures to ensure a solution in a finite and reasonably short amount of time. Once transformed into the frequency domain, usually either the E or H field is eliminated to produce a second order formulation involving an operator of the form $\nabla \times \nabla \times$. This operator guarantees a non-trivial nullspace for either of the resulting second order equations since $\nabla \times (\nabla \Phi) = 0$ for any function Φ, however the E field formulation (Weiss and Newman, 2002; Streich, 2009; Farquharson and Miensopust, 2011; Commer et al., 2011; Um et al., 2013; Haber and Ruthotto, 2014; Chung et al., 2014; Cai et al., 2015; Grayver, 2015; Yavich and Zhdanov, 2016) is generally accepted as having better numerical properties (Ren et al., 2014). Many researchers have opted to transform the equations further, by considering the vector potentials (Badea et al., 2001; Haber and Ascher, 2001; Aruliah and Ascher, 2002; Weiss, 2013; Ansari and Farquharson, 2014; Horesh and Haber, 2011; Jahandari and Farquharson, 2015). The vector potential method carries an improved condition number over the E field formulation, but has a higher complexity due to the extra degrees of freedom introduced by the potentials, and the need to satisfy a gauge condition. In all cases, the problem is cast as a large, linear system of equations, requiring linear algebra techniques to find a solution. There are two main approaches to solving the linear systems produced from any of the described discretizations: matrix factorization and substitution (direct methods), or iterative methods.

Direct methods are generally considered the most robust choice, but have historically been avoided for large problems due to their higher computational cost. Direct methods are more robust than iterative methods since they are less sensitive to the

matrix condition number. The higher a condition number, the more susceptible the solution of the linear system of equations is to small variations in the data. On its own, a high condition number isn't a problem, but with a large condition number, the small errors incurred by finite precision floating point operations and inaccurate data measurements, can produce large errors in the numerical solution. While a direct method's convergence rate is *mostly* unaffected by condition number, factorization places a large burden on memory resources and time to solution, and until recently this proved to be too much for practical problem sizes. This is much less of a concern now due to the arrival of efficient parallel sparse direct methods such as MUMPS (Schenk and Gartner, 2004) and PARDISO (Amestoy et al., 2001). Direct methods have been successfully used to solve EM problems (Streich, 2009; Schwarzbach et al., 2011; Schwarzbach and Haber, 2013; Chung et al., 2014) for up to six million degrees of freedom (Cai et al., 2015). In many other cases, problem size and memory limitations preclude the use of a direct method and preference is given to the iterative method.

Krylov subspace methods (Saad, 1995) make up the bulk of all modern uses of an iterative method for solving the discrete equations arising from discretizations of Maxwell's equations. They are computationally cheaper than direct methods, but they often require preconditioning techniques to obtain fast convergence. The Krylov iteration is based on finding an approximation to the solution by retrieving a candidate from the Krylov subspace $\kappa_m(A, r_0) = \text{span}\{r_0, Ar_0, A^2r_0, ..., A^{m-1}r_0\}$ such that the approximation forms a residual $b - Ax_m$ that is orthogonal to the subspace (Saad, 1995). The Krylov method known as Conjugate Gradient (CG) is hard to beat for symmetric, positive, and definite systems of equations. In most EM cases, General-

ized Minimal Residual (GMRES), Biconjugate Gradient Stabilized (BiCGStab), or Quasi-Minimal Residual (QMR) methods are required since the resulting equations are symmetric, but indefinite. Krylov methods are used in countless numerical investigations, and what sets one method apart from the next has more to do with preconditioning techniques than with the choice of Krylov method. Preconditioning (Meijerink and van der Vorst, 1977; Saad and van der Vorst, 2000; Saad, 1995) a linear system of equations of the form $Ax = b$ can be expressed as an equality conserving transformation in either of the left, $MAx = Mb$, or right, $AM^{-1}Mx = b$, forms. The point is to find an M which produces a system of equations that is easier to solve because it has improved spectral properties. It is also important to consider the cost of the application of the preconditioner. The best preconditioners are cheap to compute and they reduce the condition number of the system substantially. There are a number of preconditioning techniques that are based on existing linear algebra solution techniques, such as matrix factorization, matrix splitting, multigrid, and domain decomposition methods. Before discussing these, I would like to present a couple of modern approaches to preconditioning that do not fit into any of these categories.

There were two publications in which the authors made use of a Green's function approximation and a matrix splitting, respectively, to successfully accelerate a Krylov iteration. Yavich and Zhdanov (2016) used a left preconditioning strategy for the secondary electric field equations based on a Green's function approximation of the background fields. The authors estimated the condition number of a simple Green's function for the background field and highlighted its sensitivity to conductivity contrasts. They transformed the simple Green's function via scaling and shifting to

arrive at their contraction operator based preconditioner that has improved behaviour for highly contrasting conductivity. Using the contraction operator approach the authors solved a realistic marine CSEM model with close to six million unknowns to a relative tolerance of 10^{-12} in 32 BiCGStab iterations in 733 CPU seconds. Grayver and Burg (2014) implemented a block diagonal preconditioner with the real and symmetric block $C+M$ where C, M are discretizations of the curl and conductivity terms $\nabla \times (\mu^{-1}\nabla\times)$ and $\omega\sigma$. They noted that the diagonal block systems can, themselves, be solved using an auxiliary space preconditioning method, but that direct solvers were preferred for problems with less that 900000 degrees of freedom. Numerical results for the COMMEMI 3D-1 model at 10Hz show 18 iterations taking 2571 CPU seconds for 1.6M degrees of freedom.

From the right preconditioner form $AM^{-1}Mx = b$, one can observe that the choice of preconditioner $M = A$ would result in single iteration convergence, since the application of the preconditioner solves the original solution ($M^{-1}Mx \Leftrightarrow A^{-1}b$). Based on this observation, a good preconditioner might be created which uses $M = A$ but which approximates the solution process $M^{-1}Mx$ in a way that is hopefully cheaper than solving the original problem. Two such solution techniques are based on matrix factorization and splitting.

Preconditioners based on an incomplete factorization (Dongarra et al., 1998) are among the most robust, and those that are based on matrix splitting are generally the most inexpensive, and so attract many users. Incomplete factorization preconditioners are based on the same process of factorization used by direct solvers, however sparsity is preserved during factorization by enforcing limitations on fill-ins resulting from the factorization process. Incomplete LU methods are particularly widespread in

the EM modelling community due to the nature of the equations (Haber and Ascher, 2001; Mitsuhata and Uchida, 2004; Farquharson and Miensopust, 2011; Um et al., 2013; Jahandari and Farquharson, 2015; Ansari and Farquharson, 2014). These preconditioners are effective for a broad range of problems and can be tuned to produce a better and better condition number, but at the expense of time and memory cost. On the other end of the spectrum, Jacobi preconditioners are inexpensive to apply, but can sometimes fail to significantly accelerate the Krylov iteration. Jacobi preconditioners are based on the same matrix splitting into upper, lower and diagonal parts that is used to define the Jacobi iteration. They have been used in Badea et al. (2001), Weiss and Newman (2002), Nam et al. (2007), Commer et al. (2011), and Weiss (2013). Recent advances in iterative solver technology have given rise to two prominent alternatives to these established methods: multigrid (MG) and domain decomposition (DD).

Multigrid techniques (Trottenberg et al., 2001) were developed to produce an efficient error contraction for a broad range of partial differential equations (PDEs) and are attracting a lot of attention in the EM simulation world as both a standalone solver and a preconditioner. Multigrid methods are known to perform very well for elliptic problems, and their use is becoming more and more widespread all the time. They rely on a hierarchy of coarse to fine grids in which relaxation of the PDE causes an error reduction for different components of the error. This is in contrast to conventional techniques that stagnate due to their inability to reduce the error for more than a few error modes. As an iterative solver, multigrid has shown great potential, but reaching that potential often requires fine tuning the cycle of presmoothing, coarse grid correction, and post-smoothing. For this reason, it is often

fruitful to use multigrid methods as a preconditioner to the robust Krylov iteration (Trottenberg et al., 2001). Multigrid has been used in geophysics as both a solver (Li et al., 2016), and as a preconditioner (Aruliah and Ascher, 2002; Haber and Heldmann, 2007; Horesh and Haber, 2011). Jaysaval et al. (2015) used multigrid preconditioning with the BiCGStab solver to simulate a marine CSEM model using an exponential finite difference discretization for a fine grid with 127 million unknowns to a tolerance of 10^{-9} in 3656 seconds. Multigrid has been called a divide and conquer strategy since the workload is spread out among the grid hierarchy. Another important divide and conquer approach is to tackle the kernel, or null space, of the differential operator. The Auxiliary Space method is related to multigrid and pioneering work has been done for the EM problem in Hiptmair and Xu (2007) and Hiptmair and Xu (2008). Domain decomposition methods are another form of divide and conquer strategy, but with the workload spread among subdivisions of the problem domain.

Domain decomposition (DD) methods are naturally parallelizable, and have the potential to produce a good preconditioner for the geophysical EM problem. The original idea behind domain decomposition was to split a problem into smaller subdomains and solve the problem on those subdomains in a way that reproduces the solution of the original global problem (Schwarz, 1870). The original concept has since been morphed into a wide range of techniques for solving PDEs in parallel. Domain decomposition techniques can be classified into two broad categories: substructuring methods (Toselli and Widlund, 2004, chap. 4), and Schwarz methods (Dolean et al., 2015c, chap. 1). Substructuring DD methods such as Neumann-Neumann, Finite Element Tearing and Interconnecting (FETI) and Balancing Domain Decomposition by Constraints (BDDC), were developed during an effort to create parallel direct

solvers. These methods can be thought of as hybrid direct/iterative methods since they combine factorization of the interior degrees of freedom (DOFs) and relaxation of the interface DOFs. Substructuring methods have abstracted the notion of dividing the physical space into subdomains and are more concerned with splitting the matrix into subdomains (Dolean et al., 2015c). The Schwarz methods are those that have branched off from the original Schwarz concept of physical domain splitting. Today, the DD method is popular because of its ease of parallel implementation. Lions (1988) modified the original Schwarz alternating procedure so that it could be used for solving PDEs on overlapping subdomains in parallel computers. Subsequent modifications to the original Schwarz method relaxed the requirement for overlap, improved error contraction rates, and increased the parallel scalability.

To my knowledge, there are five applications of the DD method to the geophysical EM problem in the literature with the Schwarz and substructuring methods roughly equally represented. Rung-Arunwan and Siripunvaraporn (2010) used a modified hierarchical substructuring domain decomposition to solve smaller systems in serial with a direct solver in a memory limited computing environment. Bihlo et al. (2016) applied a stochastic DD method wherein Monte-Carlo simulations were used to find the stochastic representation of the solution at the interfaces. Subsequently, the subdomain problems were solved with a radial basis function based finite difference method. Bakr et al. (2013) used a non-overlapping substructuring domain decomposition technique that allowed different physics to be used in different subdomains. They then deployed a cheaper approximation in regions where a two dimensional (2D) conductivity structure dominated the model. Xiong (1999) separated the air and earth layers into subdomains to improve the condition number of the modelling

matrix using a Schwarz variant. Zyserman and Santos (2000) implemented a Schwarz DD method that used a Robin transmission condition in which the Robin parameter is chosen heuristically. This would have been very much similar to the method I use in this thesis, and will be outlined in a later chapter, but the authors recast the problem in terms of a Lagrange multiplier at the interfaces. Finally, Ren et al. (2014) is the only paper to have used a DD method to parallelize the solution of a large-scale EM problem. They chose the well known substructuring FETI-DP method – a non-overlapping DD method that defines a coarse problem from the intersecting subdomain interfaces. The solution of the coarse problem acts as a preconditioner to the interface Lagrange multipliers problem which enforces continuity of the tangential magnetic and electric fields between subdomains. The successful solution of the interface problem then allows independent solution for the interior degrees of freedom. They solved a system of equations with 3 million degrees of freedom spread over 32 processors in 2367 CPU seconds with a near linear scaling for trials of 8, 16, and 32 processors. While these papers have demonstrated the usefulness of the DD method for the geophysical EM problem in a number of non-traditional ways, there seems to be very little work done exploring the Schwarz methods as solvers or preconditioners.

The Restricted Additive Schwarz (RAS) method and the related Optimized Schwarz Method (OSM) have been developed within the last twenty years, and have been proven to work for a number of PDEs including the EM wave equation. Cai and Sarkis (1999) introduced their RAS method as a means of reducing communication cost over traditional methods, but which also happens to reduce the iteration count over other Schwarz algorithms. The Optimized Schwarz method has been developed by Martin Gander and collaborators, who have optimized Robin style transmission conditions

for Laplace's equation (Gander et al., 2001), the advection-diffusion equation (Daoud and Gander, 2010), the Helmholtz equation (Gander and Zhang, 2014), and recently for the EM wave equation (Dolean et al., 2015a). The latter publication showed that OSMs improved the scalability and convergence rate over 'classical' Schwarz methods while maintaining the cost per iteration. All of this seems to suggest that the OSM would also be well suited to EM equations under the quasistatic approximation.

In this thesis I will outline the steps I have taken towards constructing and implementing an Optimized Schwarz preconditioner for the quasistatic EM equations. Along the way, I will develop an RAS preconditioner to use as a benchmark to measure the success (or lack thereof) of the OSM preconditioner. In this chapter, I have explained the motivation for the use of an Optimized Schwarz preconditioner for the EM problem. In the next chapter, I will develop the optimized Schwarz solver and preconditioner for the Poisson problem. The RAS preconditioner is easily constructed once the components of the optimized Schwarz preconditioner are in place. I will begin the chapter by discussing the model Poisson problem (section 2.1), and then provide an overview of the Schwarz methods from the original concept through to the modern discrete parallel algorithms using the Poisson problem as a model (section 2.2). Later in the chapter, I will implement a number of key algorithms presented in section 2.2 using the finite difference discretization technique (section 2.3), followed by the same treatment while using the finite element technique (section 2.4). In the final chapter, I will demonstrate my work towards constructing the Optimized Schwarz and RAS preconditioners for the quasistatic EM problem.

Chapter 2

Poisson Problem

2.1 Poisson problem as a test case

The Poisson equation for the unknown solution u, load function f, and Dirichlet data β over the domain Ω:

$$-\nabla^2 u = f \quad \text{in } \Omega \tag{2.1a}$$

$$u = \beta \quad \text{on } \partial\Omega \tag{2.1b}$$

is an elliptic partial differential equation that is used throughout the sciences to model diffusion, and is a good model equation for developing the OSM. There has been a lot of work done on the Poisson equation and a number of methods can be used to find its solution, both numerically and analytically. For this reason, the Poisson equation makes an ideal candidate for discussing and testing numerical methods. Given an appropriate source function and boundary conditions, the Poisson equation could be used to model the geophysical potential field, or DC resistivity experiment. However

in the work that follows, I will work with a simple sinusoidal source function to avoid any complications arising from the use of a point source. In the next section, I use the Poisson equation to illustrate the main ideas behind the domain decomposition method. I will begin with the earliest domain decomposition method designed for Laplace's equation, and I will end by discussing the motivation for the development of the optimized Schwarz method and for using it, as well as all other Schwarz methods, as a preconditioner rather than a stand-alone solver. In later sections, I present the results of my numerical testing of the RAS and optimized Schwarz methods on the Poisson equation using finite difference and finite element discretization techniques.

2.2 Schwarz Domain Decomposition Methods

2.2.1 Historical Perspective

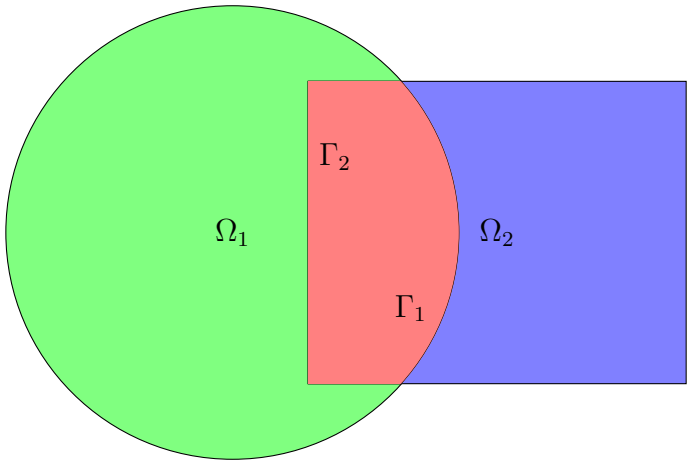

Figure 2.1: Two-subdomain decomposition similar to that used in Schwarz (1870) to investigate existence and uniqueness of the Laplace equation over arbitrary domains.

The Schwarz method was originally conceived as a means to investigate the existence and uniqueness of solutions to PDEs on arbitrary domain shapes, but has been adopted as a way to solve PDEs in parallel. I begin this section with a bit of an historical overview since I think it sets up the need for OSMs nicely, and it demonstrates the use of domain decomposition methods for the Poisson problem. The idea behind the domain decomposition method is as old as the Schwarz (1870) paper, in which the author uses alternating solutions within an overlapping circle and square, like the one shown in Figure 2.1, to prove existence and uniqueness of a solution for the hybrid shape. The alternating Schwarz procedure for the Poisson problem generates the $(k+1)^{th}$ iterate u^{k+1} by Algorithm 1, with load function f using the two subdomains Ω_1 and Ω_2, subdomain solutions u_1^k and u_2^k, Dirichlet data g, and interfaces Γ_1 and Γ_2. More recently, this algorithm was adapted for use on parallel computers by Lions (1988). The parallel Algorithm 2 is attained by simply taking Dirichlet data at all interfaces from the local solutions of the neighboring subdomain from the previous iteration (u_{3-i}^k). Lion's parallel adaptation of the original Schwarz method is the starting point for a wide variety of related parallel iterative methods, many of which are written in algebraic terms, and iterate on a global solution u^k or residual r^k rather than the local solutions (u_i^k) seen here.

2.2.2 Connection to modern Schwarz DD

Most modern domain decomposition literature discusses the iterative method in terms of the global iterates u^k rather than the local iterates u_i^k. This is an especially natural choice when the end goal is to provide a preconditioner to the global Krylov subspace

Algorithm 1 Original Schwarz alternating procedure

while $u_1^k \neq u_2^k$ **do**

solve:

$$-\nabla^2 u_1^{k+1} = f \quad \text{in } \Omega_1 \tag{2.2}$$

$$u_1^{k+1} = \beta \quad \text{on } \partial\Omega_1 \setminus \Gamma_1 \tag{2.3}$$

$$u_1^{k+1} = u_2^k \quad \text{on } \Gamma_1, \tag{2.4}$$

then solve:

$$-\nabla^2 u_2^{k+1} = f \quad \text{in } \Omega_2 \tag{2.5}$$

$$u_2^{k+1} = \beta \quad \text{on } \partial\Omega_2 \setminus \Gamma_2 \tag{2.6}$$

$$u_2^{k+1} = u_1^{k+1} \quad \text{on } \Gamma_2, \tag{2.7}$$

end while

Algorithm 2 P.L. Lions parallel Schwarz algorithm

while $u_1^k \neq u_2^k$ **do**

for $i = \{1, 2\}$ **do**

solve in parallel:

$$-\nabla^2 u_i^{k+1} = f \quad \text{in } \Omega_i \tag{2.8}$$

$$u_i^{k+1} = \beta \quad \text{on } \partial\Omega_i \setminus \Gamma_i \tag{2.9}$$

$$u_i^{k+1} = u_{3-i}^k \quad \text{on } \Gamma_i, \tag{2.10}$$

end for

end while

iteration. The terms 'local' and 'global' are potentially misleading. They can be used to distinguish between local and global parts of a parallel data structure, or local and global parts of the domain. When I use the terms, I will mean the latter. This way, when I refer to the global iterate u^k in the context of a parallel algorithm, I will mean that the update occurs on a vector defined for the global domain, while the vector is distributed across an array of processors. As I will demonstrate next, the global iteration algorithms rely on a series of operations on local entities, requiring a pair of operators designed to map between local and global domains. In order to begin using the global iterate, I must introduce the extension and partition of unity operators, $E_{1,2}$ and $\chi_{1,2}$, which are designed so that

$$u = E_1(\chi_1 u^\infty) + E_2(\chi_2 u^\infty). \tag{2.11}$$

That is, $\chi_{1,2}$ is designed so that subdomains contribute an amount that would exactly reconstruct the global solution once the local solutions reach their fixed point (u^∞). One way to create $\chi_{1,2}$ is to have either χ_1 or χ_2 be equal to one for the whole overlap and the other equal to zero so that one local solution contributes its full amount, and the other contributes nothing (Figure 2.2). Another feasible way is to have both χ_1 and χ_2 scale the local iterates in the overlap so that they contribute their average. $E_{1,2}$ maps the functions in $\Omega_{1,2}$ to Ω by extending the functions with a value of zero outside of $\Omega_{1,2}$. The global counterpart of Algorithm 2 using the extension and partition of unity operators is provided in Algorithm 3. However, it is even more common to see a global Schwarz algorithm written as a global residual iteration rather than a global solution iteration. An example is provided in Algorithm 4.

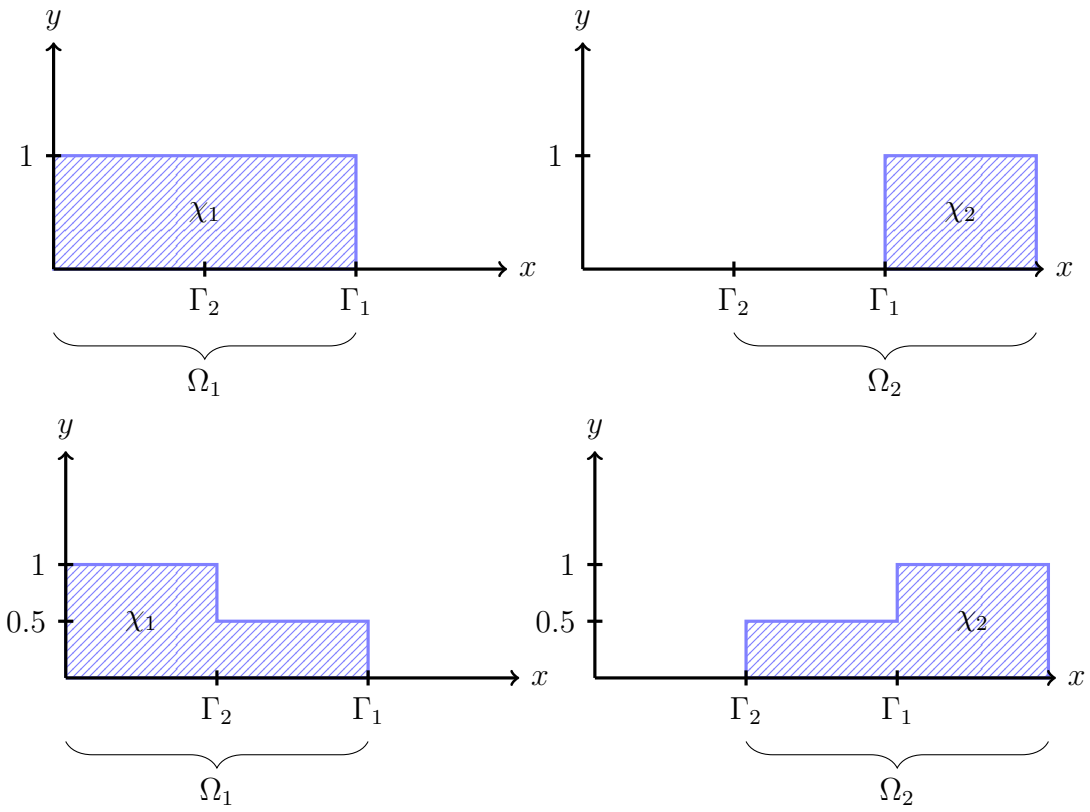

Figure 2.2: χ_1 and χ_2 where χ_1 is scaled to permit u_1 to contribute its full solution within the overlap and χ_2 correspondingly scaled so that u_2 contributes nothing (top) and where χ_1 and χ_2 scale u_1 and u_2 so that they contribute the average of the two solution in the overlap (bottom).

The global residual iteration is, in fact, equivalent to the original Schwarz alternating method. Dolean et al. (2015c), for example, provides a proof which begins by asserting that in order for the algorithms to be equivalent, the following must be true

$$u^k = E_1(\chi_1 w_1^k) + E_2(\chi_2 w_2^k), \tag{2.19}$$

where the u^k is the solution from the global residual Algorithm 4, and $w_{1,2}^k$ are the

Algorithm 3 Schwarz global iteration algorithm

while $u^{k+1} \neq u^k$ **do**

for $i = \{1, 2\}$ **do**

solve in parallel:

$$-\nabla^2 u_i^{k+1} = f \qquad \text{in } \Omega_i \tag{2.12}$$

$$u_i^{k+1} = \beta \qquad \text{on } \partial\Omega_i \setminus \Gamma_i \tag{2.13}$$

$$u_i^{k+1} = u_{3-i}^k \quad \text{on } \Gamma_{3-i}, \tag{2.14}$$

update:

$$u^{k+1} = \sum_{i=1}^{2} E_i(\chi_i u_i^{k+1}) \tag{2.15}$$

end for

end while

solutions from the original Schwarz algorithm. I can assume that with appropriate extension and partition of unity operators, the initial solution guesses can be stitched by

$$w^0 = E_1(\chi_1 w_1^0) + E_2(\chi_2 w_2^0), \tag{2.20}$$

and that the same is true for a particular iterate w^k

$$w^k = E_1(\chi_1 w_1^k) + E_2(\chi_2 w_2^k), \tag{2.21}$$

and I can attempt to complete the proof by induction. From 2.18, I have

$$u^{k+1} = u^k + E_1(\chi_1 v_1^k) + E_2(\chi_2 v_2^k). \tag{2.22}$$

Algorithm 4 Global residual iteration

while $r^k > \text{tol}$ **do**

compute:

$$r^k(x) = f(x) + \nabla^2 u^k(x). \tag{2.16}$$

for $i = \{1, 2\}$ **do**

$r_i \leftarrow$ **restrict** r

solve in parallel:

$$\begin{aligned} -\nabla^2 v_i^k(x) &= r_i^k(x) \qquad \text{in } \Omega_i \\ v_i^k(x) &= 0 \qquad\quad\ \ \text{on } \partial\Omega_i. \end{aligned} \tag{2.17}$$

update:

$$u^{k+1}(x) = u^k(x) + \sum_i E_i(\chi_i v_i^k(x)) \tag{2.18}$$

end for

end while

If I now let $u^k = w^k$ and substitute by Equation (2.21), I get

$$u^{k+1} = E_1(\chi_1 w_1^k) + E_2(\chi_2 w_2^k) + E_1(\chi_1 v_1^k) + E_2(\chi_2 v_2^k), \tag{2.23}$$

and I can then collect terms for subdomain one and two as

$$u^{k+1} = E_1(\chi_1(w_1^k + v_1^k)) + E_2(\chi_2(w_2^k + v_2^k)). \tag{2.24}$$

Now all that remains is to show that $u_i^{k+1} = w_i^k + v_i^k$ for $i = 1, 2$. I already know that u_i^{k+1} satisfies Equation (2.12). If I can show that $w_i^k + v_i^k$ also satisfies Equation (2.12) then the equality must be true. I start by expanding for subdomain one

$$-\nabla^2(w_1^k + v_1^k) = -\nabla^2 w_1^k - \nabla^2 v_1^k, \tag{2.25}$$

and noting that, from Equation (2.17), $-\nabla^2 v_1^k = r^k$, giving

$$-\nabla^2(w_1^k + v_1^k) = -\nabla^2 w_1^k + r^k. \tag{2.26}$$

Now, from the definition of the residual

$$-\nabla^2(w_1^k + v_1^k) = -\nabla^2 w_1^k + (\nabla^2 w_1^k + f) = f, \tag{2.27}$$

giving the first of the boundary value problem (BVP) equations. To satisfy the second equation, I now assert that $w_1^k + v_1^k = u_2^k$ on Γ_1 by considering the equation $u^k = E_1(\chi_1^k u_1^k) + E_2(\chi_2^k u_2^k)$ at the interface Γ_1 where $\chi_1 = 0$ and $\chi_2 = 1$. With both equations of the BVP satisfied for $w_1^k + v_1^k$, I can state that $w_1^k + v_1^k = u_1^{k+1}$ and similarly that $w_2^k + v_2^k = u_2^{k+1}$. Therefore $u^{k+1} = E_1(\chi_1 w_1^{k+1}) + E_2(\chi_2 w_2^{k+1})$, and I conclude by induction that the two algorithms are equivalent.

The Restricted Additive Schwarz (RAS) algorithm is based on a global iteration credited to Cai and Sarkis (1999), who discovered it by accident while trying to improve communication cost by having a single subdomain contribute its fully weighted solution in the overlap. The RAS algorithm is a global residual iteration in the form of Algorithm 4 with the definition of the partition of unity function set so that one subdomain contributes its solution in the overlap while the other contributes nothing. The algorithm is very popular thanks in part to its algebraic implementation as a preconditioner in the linear algebra package PETSc. I will demonstrate in the next section how the discrete RAS method can be thought of as a preconditioner. This particular use of the RAS method will be important to my work as a benchmark while attempting to create optimized Schwarz preconditioners for both the Poisson and EM problems.

The Lions algorithm provided a window to domain decomposition based parallel solution of PDEs that would take advantage of emerging multi-processor technology. However, in order to solve a PDE in a computer, the PDE, as well as any methods for its solution, must first be discretized. In the next section, I will overcome this hurdle by creating discrete algebraic counterparts of the partition of unity and extension function for use with finite difference or finite element discretizations of the PDE.

2.2.3 Discrete Schwarz Domain Decomposition

The discrete Schwarz algorithms are formed by combining algebraic operations with the discrete counterparts of the extension and partition of unity operators. I will start with the extension operator, but it is common to first define the related discrete

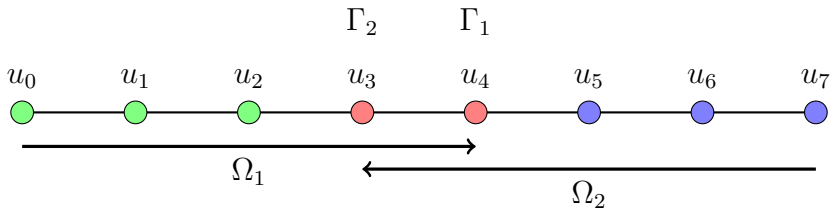

Figure 2.3: A simple decomposition of the 1D mesh into two subdomains

restriction operator. The space Ω and its subspaces Ω_1 and Ω_2 in the continuous RAS algorithm can be discretized in one dimensions by taking a finite set of points in the x dimension. In two dimensions, a discretization may involve either taking points in both x and y along a rectilinear grid suited to finite difference methods, or at the nodes of a triangulation suited to the finite element approach. In three dimensions, the same is done in x, y, and z dimensions in rectilinear fashion for the finite difference method, or tetrahedral for the finite element method. In each case, the functions taken from those spaces $u \in \Omega$, $u_1 \in \Omega_1$, $u_2 \in \Omega_2$ of the continuous algorithm are represented by vectors in the discrete version. Naturally, the restriction operator must take a vector in Ω and transform it into a vector in Ω_1 or Ω_2. The simplest algebraic operator that accomplishes this task is the Boolean matrix which contains rows of the identity matrix designed to pick out elements of the vector that belong to a particular subdomain (Dolean et al., 2015c). To give a small example, consider the one dimensional mesh consisting of seven points along the x axis as in Figure 2.3. To accomplish restriction of a function $u \in \Omega$ into the function $u_1 \in \Omega_1$, I construct a matrix with the first five rows of a 7x7 identity matrix. Left multiplication with such a matrix (R_1) yields a $u_1 \in \Omega_1$ as demonstrated below.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \end{pmatrix} = \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \end{pmatrix} \tag{2.28}$$

The same can be done to extract the vector $u_2 \in \Omega_2$ by using the last 5 rows of a 7x7 identity matrix:

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \end{pmatrix} = \begin{pmatrix} u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \end{pmatrix}. \tag{2.29}$$

The reason I started with the restriction operator is that I find its action to be more intuitive, and because the extension operator is obtained by simply taking the transpose of the restriction operator. The transpose operator

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}^T = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \tag{2.30}$$

can be used to right multiply a vector $u_1 \in \Omega_1$ back into $u \in \Omega$

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \end{pmatrix} = \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \\ 0 \\ 0 \end{pmatrix}. \tag{2.31}$$

The discrete partition of unity is now all that is left in order to write down the discrete RAS algorithm. Recall that the partition of unity operator is meant to correct the overlap between two subdomain functions that are being added together. The matrix for this operation consists of weighted diagonal entries such that only one subdomain contributes its fully weighted solution, or so that both subdomains contribute half of their solution (Figure 2.2) (Dolean et al., 2015c). The matrices that would perform these operations are

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.32}$$

and

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 0.5 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 0.5 & 0 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \tag{2.33}$$

respectively. By definition, the restriction matrices R_i and partition of unity matrices D_i must obey the following equality

$$I = \sum_i R_i^T D_i R_i \tag{2.34}$$

where the sum over all subdomains $i = 1, ..., N$ produces the identity matrix. This is a convenient debugging tool when coding the RAS algorithm, and a quick check with any linear algebra library will verify that this holds true for the operators described above.

The partition of unity and restriction/extension operators provided here are for one dimensional discretizations. To get the two dimensional counterparts, I use a property of the Kronecker product that I will introduce in a later section and which I have fully explained in Appendix B. For now, these operators can be thought of as either one or two dimensional entities with the common symbols R and D with the assumption that the two dimensional versions require a further step. I will now construct the discrete RAS algorithm by translating the continuous equations 2.16, 2.17 and 2.18, into discrete matrix operations and then assembling them into a two part formula. The residual computation 2.16 translates with little alteration except that the continuous operator ∇^2 is now represented by the discretized differential operator A. This gives the discrete equation for the vectors r^k, u^k and f:

$$r^k = Au^k + f. \tag{2.35}$$

I will call the i^{th} restriction matrix R_i, so that I can write the local correction step in equation 2.17 for the global residual r^k as

$$A_i^{-1} R_i r^k = v_i^k. \tag{2.36}$$

To prepare for summation, the local correction must be scaled by the partition of unity and mapped into the global space by the extension operator, or the transpose of the restriction operator. These steps are represented by the matrix multiplications

$$R_i^T D_i v_i^k. \tag{2.37}$$

The discrete RAS algorithm can now be fully stated by combining steps 2.36 and 2.37, by eliminating v_i^k, summing the contributions from each subdomain and updating u^k, as shown in Algorithm 5.

Algorithm 5 Algebraic restricted additive Schwarz algorithm

while $r^k >$ tol **do**

compute:

$$r^k = Au^k + f. \tag{2.38}$$

update in parallel:

$$u^{k+1} = u^k + \sum_i R_i^T D_i A_i^{-1} R_i r^k. \tag{2.39}$$

end while

The RAS algorithm is based on Dirichlet transmission conditions, meaning that they use a Dirichlet BC on the interface between subdomains. I have assumed that the A_i matrices in Algorithm 5 are the discretizations of BVPs with Dirichlet conditions on the exterior boundaries and interfaces. It turns out that instead of discretizing the subdomain matrices from scratch, they may be simply restricted out of the global matrix A that is already constructed for the residual calculation. Dolean et al. (2015c)

construct the RAS method using a restriction of the global system matrix to form the subdomain matrices $A_i = R_i A R_i^T$. Regardless of whether the matrix is restricted, or discretized from scratch, or whether one choice of partition of unity is preferred over another, an important observation can be made about the structure of Equation (2.39) in Algorithm 5. It has the general form of a fixed point iteration, and it may be more advantageous to use the preconditioner with another kind of iteration, as I'll discuss next.

Global Schwarz iterations are equivalent to fixed point iterations preconditioned by the local solutions of subdomain problems, but theory suggests that the preconditioner would be better served as an accelerator to a Krylov method (Dolean et al., 2015c). The fixed point iteration has the form $u^{k+1} = u^k + M^{-1}(b - Ax^k)$ where M^{-1} is a preconditioner. Given this definition, I can identify the Schwarz preconditioner in Equation (2.39) as $\sum_i R_i^T D_i A_i^{-1} R_i$. Dolean et al. (2015c) demonstrates that the fixed point iteration produces an approximation to the solution lying in the space defined by powers of the iteration matrix $M^{-1}P$: $\text{Span}\{M^{-1}r^0, (M^{-1}P)M^{-1}r^0, ..., (M^{-1}P)^n M^{-1}r^0\}$. Furthermore, Dolean showed that a Krylov method will generate a better approximation to the solution in the same space, requiring fewer iterations, while matching the cost per iteration of the fixed point method. The Schwarz preconditioner is thus much better utilized when partnered with a Krylov iteration. I can provide this preconditioner to, for example, the GMRES algorithm which constructs the orthogonal bases

$$\text{Span}\{r_0, M^{-1}Ar_0, ...(M^{-1}A)^{m-1}r_0\}, \tag{2.40}$$

or

$$\text{Span}\{r_0, AM^{-1}r_0, ..., (AM^{-1})^{m-1}r_0\}, \tag{2.41}$$

depending on whether left of right preconditioning is preferred. In either case the Krylov algorithms only need to have a method for applying the preconditioner to a vector $m : v \mapsto M^{-1}v$. The use of the Schwarz preconditioner to accelerate a Krylov iteration isn't the only way to improve upon the original Schwarz method, however. The RAS algorithm relies on Dirichlet transmission conditions, but new algorithms have been developed using Neumann and what may be interpreted as generalized transmission conditions, consisting of a weighted combination of Neumann and Dirichlet conditions. The weighted combination of Neumann and Dirichlet conditions is known as a Robin condition, and the weighting parameter in this condition has become the subject of a new field of domain decomposition research: the Optimized Schwarz Method.

2.2.4 Optimized Schwarz Domain Decomposition

Optimized Schwarz methods (OSMs) use Robin transmission conditions and can improve the convergence properties of the classical RAS algorithm for many problems. I have already mentioned P.L. Lions as the progenitor of the parallel Schwarz alternating algorithm, but in his concluding paper on the Schwarz method, he introduces a Schwarz based algorithm that converges without overlap (Lions, 1988). Today this is known as the Optimized Schwarz method, and it differs from the classical Schwarz method only by its transmission condition. Lions found that the Poisson problem con-

verged without overlap when he replaced the Dirichlet transmission condition in the "classical" Schwarz algorithm with a Robin style transmission condition. Subsequent to this discovery, researchers have analyzed the new Robin transmission conditions and discovered many beneficial properties over the classical conditions. Gander (2006) summarizes the benefits of the OSM as:

"

1. *They converge necessarily faster than classical Schwarz methods, at the same cost per iteration.*
2. *There are simple optimization procedures to determine the best parameters to be used in the transmission conditions, sometimes even closed formulas, depending on the problem solved.*
3. *Classical Schwarz implementations need only a small change in the implementation, in the information exchange routine, to benefit from the additional performance.*
4. *Optimized Schwarz methods can be used with or without overlap.*

"

The OSM can be expressed in terms of both a local and a global iterate, just as in the RAS method. The two new methods are provided in Algorithms 6 and 7.

For the local iteration algorithm, there is one last modification that is often employed to avoid discretization of the normal derivative in the Robin condition. The auxiliary variable method introduces the definition

$$g_i^k := -\frac{\partial u_{3-i}^k}{\partial n_{3-i}} + \alpha u_{3-i}^k, \tag{2.48}$$

so that Equation (2.44) becomes

$$\frac{\partial u_i^{k+1}}{\partial n_i} + \alpha u_i^{k+1} = g_i^k. \tag{2.49}$$

Algorithm 6 Optimized Schwarz as a local iteration

while $u_1^k \neq u_2^k$ **do**

for $i = \{1, 2\}$ **do**

solve in parallel

$$-\nabla^2 u_i^{k+1} = f \qquad \text{in } \Omega_i \tag{2.42}$$

$$u_i^{k+1} = \beta \qquad \text{on } \partial\Omega_i \setminus \Gamma_i \tag{2.43}$$

$$\frac{\partial u_i^{k+1}}{\partial n_i} + \alpha u_i^{k+1} = \frac{\partial u_{3-i}^{k}}{\partial n_{3-i}} + \alpha u_{3-i}^{k} \qquad \text{on } \Gamma_i, \tag{2.44}$$

end for

end while

By adapting Definition (2.48) for a g_i at step $k+1$, I can re-write the update formula (Equation (2.44)) as

$$g_i^{k+1} = -\frac{\partial u_{3-i}^{k+1}}{\partial n_{3-i}} + \alpha u_{3-i}^{k+1} = -(\frac{\partial u_{3-i}^{k+1}}{\partial n_{3-i}} + \alpha u_{3-i}^{k+1}) + 2\alpha u_{3-i}^{k+1} = -g_{3-i}^{k} + 2\alpha u_{3-i}^{k+1} \tag{2.50}$$

where the first equality is just an algebraic manipulation, and the third equality comes from Equation (2.49). The parallel optimized Schwarz algorithm can now be written in two steps, as demonstrated in Algorithm 8. The global iteration is written in discrete algebraic form in the same way that was done for the global RAS iteration, except that the matrix consists of the discretization of a subdomain problem with Robin transmission conditions, $A_{Robin,i}$. The resulting preconditioner $\sum_i R_i^T D_i A_{Robin,i}^{-1} R_i$ is known as the Optimized Restricted Additive Schwarz (ORAS) preconditioner and may be used with the fixed point iteration in Algorithm 9 or with a Krylov iteration. Just like with the RAS preconditioner, the ORAS preconditioner

Algorithm 7 Optimized Schwarz as a global iteration

while $r^k > \text{tol}$ **do**

compute:

$$r^k(x) = f(x) + \nabla^2 u^k(x). \tag{2.45}$$

for $i = \{1, 2\}$ **do**

$r_i \leftarrow$ **restrict** r

solve in parallel:

$$\begin{aligned} -\nabla^2 v_i^k(x) &= r_i^k(x) && \text{in } \Omega_i \\ v_i^k(x) &= 0 && \text{on } \partial\Omega_i \\ \frac{\partial u_i^k}{\partial n} + \alpha u_i^k &= 0 && \text{on } \Gamma \end{aligned} \tag{2.46}$$

end for

update:

$$u^{k+1}(x) = u^k(x) + \sum_i E_i(\chi_i v_i^k(x)) \tag{2.47}$$

end while

is supplied to the Krylov iteration as a routine which applies the preconditioner to a vector. The Krylov iteration then uses the routine at every iteration to construct its left or right preconditioned orthogonal basis (Equations (2.40) and (2.41)). In the next two sections, I will use finite difference and finite elements to discretize the Robin subdomain problems in a two subdomain configuration and demonstrate the ORAS preconditioned fixed point and Krylov methods for solving the Poisson problem.

Algorithm 8 Optimized Schwarz as a local iteration using the auxiliary variable method

while $g_1^k \neq g_2^k$ **do**

solve:

$$-\nabla^2 u_i^{k+1} = f \qquad \text{in } \Omega_i \tag{2.51}$$

$$u_i^{k+1} = \beta \qquad \text{on } \partial\Omega_i \setminus \Gamma_i \tag{2.52}$$

$$\frac{\partial u_i^{k+1}}{\partial n} + \alpha u_i^{k+1} = g_{3-i}^k \quad \text{on } \Gamma_i, \tag{2.53}$$

update:

$$g_i^{k+1} = -g_{3-i}^k + 2\alpha u_{3-1}^{k+1} \tag{2.54}$$

end while

Algorithm 9 Algebraic Optimized Schwarz algorithm

while $r^k >$ tol **do**

compute:

$$r^k = A_{Robin,i} u^k + f \tag{2.55}$$

update in parallel:

$$u^{k+1} = u^k + \sum_i R_i^T D_i A_{Robin,i}^{-1} R_i r^k, \tag{2.56}$$

end while

2.3 Finite Difference

2.3.1 One dimensional subdomain problems

The foundation of any domain decomposition method is the correct solution of the subdomain problems equipped with the appropriate transmission conditions. In order to ensure that the subdomain problems could be solved accurately, I tested the subdomain solutions using a finite difference discretization of the Laplacian operator with Robin transmission conditions using a numerical experiment which compares $O(h)$ and $O(h^2)$ approximations to the Robin boundary condition. In this section, I describe that experiment and present results for the simple one dimensional Robin subdomain problem. The experiment consisted of the following:

1. Impose a known solution.

2. Derive the first and second derivatives from the known solution in order to compute the Robin condition and right hand side of a simple model problem.

3. Solve the PDE equipped with a Robin BC using the Robin condition and right hand side derived from the known solution.

4. Check that the numerical solution of the single domain Robin problem matches the original imposed solution as the mesh is refined. The rate at which the error decreases with refinement should also match the theoretical discretization error. For a mesh size decrease of a factor of two, $O(h)$ discretizations of the Robin condition should produce an error that decreases by a factor of two and $O(h^2)$ discretizations should produce a factor of four error reduction.

I performed this experiment for the one dimensional Poisson problem (Equation (2.1)), and demonstrated that the subdomain problems with Robin transmission conditions fulfill the criteria described in the last step of the experiment described above. I will now describe how I carried out the experiment using finite differences and provide a table that demonstrates the correct error behaviour.

The success of the experiment relies heavily on the appropriate application of the Robin transmission condition and uses the ghost point method for the discretization. In the following, I consider the subdomain problems arising from the non-overlapping decomposition of the global domain into two equally sized subdomains. In one dimension, the decomposition consists of a number of nodes along the x axis and partitioned so that the middle node is shared by both subdomains as demonstrated in Figure 2.4.

The first three steps of the experiment require a solution, and its first and second derivatives. I began by choosing a solution that is smooth within the domain $\Omega = \Omega_1 \cup \Omega_2$, and has a non-zero value and derivative on the boundary $\partial\Omega$.

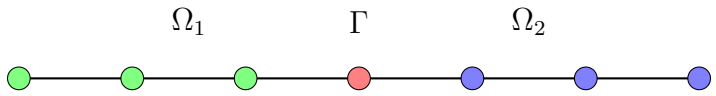

Figure 2.4: A simple decomposition of the 1D mesh into two non-overlapping subdomains

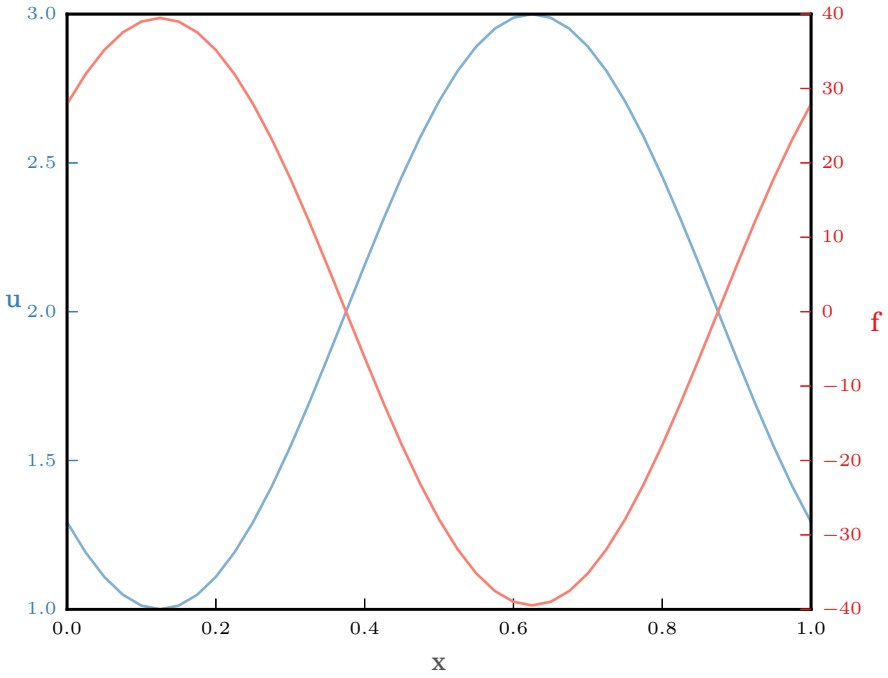

Figure 2.5: Function u and f used in the one dimensional experiments

$$u = \sin(2\pi x - \frac{3\pi}{4}) + 2. \tag{2.57}$$

Then, I used the first derivative,

$$\frac{\partial u}{\partial x} = 2\pi \cos(2\pi x - \frac{3\pi}{4}), \tag{2.58}$$

to build the Robin BC, and the second derivative,

$$\frac{\partial^2 u}{\partial x^2} = -4\pi^2 sin(2\pi x - \frac{3\pi}{4}) =: f, \tag{2.59}$$

to build the right hand side (Figure 2.5). In step three of the experiment, I solved the subdomain problems with a Robin BC. Before tackling the details of the transmission boundary discretization, I will demonstrate the finite difference discretization of the Laplacian operator with Dirichlet conditions on the outer boundary. The second order centered finite difference stencil for the Laplacian is

$$\frac{u_{i-1} - 2u_i + u_{i+1}}{h^2}, \tag{2.60}$$

so I can write the discrete form of the Poisson equation for the interior nodes in a mesh as

$$\frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} = f_i, \tag{2.61}$$

or

$$u_{i-1} - 2u_i + u_{i+1} = h^2 f_i, \quad \text{for } i = \{1, ..., n-1\}. \tag{2.62}$$

In order to apply a Dirichlet boundary condition at, for example $i = 0$ in Ω, I combined the two equations

$$\begin{aligned} u_0 - 2u_1 + u_2 &= h^2 f_1 \\ u_0 &= \beta_0, \end{aligned} \tag{2.63}$$

to give

$$-2u_1 + u_2 = h^2 f_1 - \beta_0. \tag{2.64}$$

I used the same procedure to derive an equation for the boundary at $i = n$ in Ω, where $u_n = \beta_n$, resulting in

$$u_{n-2} - 2u_{n-1} = h^2 f_{n-1} - \beta_n. \tag{2.65}$$

I formed the finite difference matrix for the global Poisson problem in Ω row by row by Equations (2.60), (2.64), and (2.65), producing the discrete linear problem to solve for the global solution vector u:

$$\begin{pmatrix} -2 & 1 & & & & & \\ 1 & -2 & 1 & & & & \\ & 1 & -2 & 1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & 1 & -2 & 1 & \\ & & & & 1 & -2 & \end{pmatrix} \cdot \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{n-2} \\ u_{n-1} \end{pmatrix} = \begin{pmatrix} h^2 f_1 - \beta_0 \\ h^2 f_2 \\ h^2 f_3 \\ \vdots \\ h^2 f_{n-2} \\ h^2 f_{n-1} - \beta_n. \end{pmatrix} \tag{2.66}$$

However, to carry out the experiment, I needed to discretize the Robin problem. To do this, I used a similar discrete linear system in which a Robin BC was imposed at the interface marked as Γ in Figure 2.4. I did this both for the left and right domains (Ω_1 and Ω_2), and considered a scheme with both the Laplacian and the Robin BC discretized with a centered difference stencil, as well as a scheme that discretized the Robin BC with a forward difference discretization.

The Robin boundary condition has the form

$$\frac{\partial u}{\partial n} + \alpha u = g. \tag{2.67}$$

I discretized the normal derivative, for the left subdomain, looking outside the mesh using a ghost point method (see Neumann BC discretization techniques in, for example, LeVeque (2007)). I applied the condition by eliminating the ghost point in the corresponding row of the system matrix, removing the right Dirichlet condition and adding a row to the matrix in Equation (2.66). The forward difference discretization of the first derivative is

$$\frac{u(x+h) - u(x)}{h}. \tag{2.68}$$

Using the ghost point u_G, I wrote the discrete Robin condition in Ω_1 using forward differences in the direction of the outward facing normal n_1 as

$$\frac{u_G - u_n}{h} + \alpha u_n = g. \tag{2.69}$$

from which u_G was found to be

$$u_G = hg + (1 - h\alpha)u_n. \tag{2.70}$$

The right subdomain Ω_2 features a left pointing normal vector so that the forward difference discretization produces a negative normal derivative. The resulting Robin condition is

$$-\frac{u_0 - u_G}{h} + \alpha u_0 = g, \tag{2.71}$$

and u_G was determined to be

$$u_G = hg + (1 - h\alpha)u_0. \tag{2.72}$$

In each case, I eliminated u_G by substituting into the ghost point in the Laplacian stencil centered at u_n and u_0 for Ω_1 and Ω_2,

$$\frac{u_{n-1} - 2u_n + (hg + (1 - h\alpha)u_n)}{h^2} = f_n, \tag{2.73}$$

and

$$\frac{(hg + (1h\alpha)u_0) - 2u_0 + u_1}{h^2} = f_0. \tag{2.74}$$

I then re-arranged to produce the equations

$$u_{n-1} - (h\alpha + 1)u_n = h^2 f_n - hg, \tag{2.75}$$

and

$$-(h\alpha + 1)u_0 + u_1 = h^2 f_0 - hg. \tag{2.76}$$

The Robin conditions were applied by removing the Dirichlet conditions in Equation (2.66) and adding the Robin discretization as a row to the bottom of the matrix for the left subdomain and the top of the matrix for the right subdomain. The resulting Robin subdomain matrices for the left and right subdomains were

$$\begin{pmatrix} -2 & 1 & & & & \\ 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & & \\ & & 1 & -2 & 1 & \\ & & & 1 & -(h\alpha+1) \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{n-1} \\ u_n \end{pmatrix} = \begin{pmatrix} h^2 f_1 - \beta_0 \\ h^2 f_2 \\ h^2 f_3 \\ \vdots \\ h^2 f_{n-1} \\ h^2 f_n - hg \end{pmatrix}. \tag{2.77}$$

and

$$\begin{pmatrix} -(h\alpha+1) & 1 & & & & & \\ 1 & -2 & 1 & & & & \\ & 1 & -2 & 1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & 1 & -2 & 1 & \\ & & & & 1 & -2 \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_{n-2} \\ u_{n-1} \end{pmatrix} = \begin{pmatrix} h^2 f_0 - hg_0 \\ h^2 f_1 \\ h^2 f_2 \\ \vdots \\ h^2 f_{n-2} \\ h^2 f_{n-1} - \beta_n \end{pmatrix}, \tag{2.78}$$

I also discretized the normal derivative in the Robin boundary condition using a second order centered finite difference stencil

$$\frac{u(x+h) - u(x-h)}{2h}. \tag{2.79}$$

Following the same procedure as before, I arrived at the following equations for the left and right subdomains at the interface

$$2u_{n-1} - 2(h\alpha+1)u_n = h^2 f_n - 2hg, \tag{2.80}$$

and

$$-2(h\alpha+1)u_0 + 2u_1 = h^2 f_0 - 2hg. \tag{2.81}$$

The discrete systems that I used for the centered approximation were thus

$$\begin{pmatrix} -2 & 1 & & & & & \\ 1 & -2 & 1 & & & & \\ & 1 & -2 & 1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & 1 & -2 & 1 & \\ & & & & 2 & -2(h\alpha+1) & \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{n-1} \\ u_n \end{pmatrix} = \begin{pmatrix} h^2 f_1 - \beta_0 \\ h^2 f_2 \\ h^2 f_3 \\ \vdots \\ h^2 f_{n-1} \\ h^2 f_n - 2hg_n \end{pmatrix} \tag{2.82}$$

and

$$\begin{pmatrix} -2(h\alpha+1) & 2 & & & & & \\ 1 & & -2 & 1 & & & \\ & & 1 & -2 & 1 & & \\ & & & \ddots & \ddots & \ddots & \\ & & & & 1 & -2 & 1 \\ & & & & & 1 & -2 \end{pmatrix} \cdot \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_{n-2} \\ u_{n-1} \end{pmatrix} = \begin{pmatrix} h^2 f_0 + 2hg_0 \\ h^2 f_1 \\ h^2 f_2 \\ \vdots \\ h^2 f_{n-2} \\ h^2 f_{n-1} - \beta_n \end{pmatrix}. \tag{2.83}$$

Using Equations (2.77), (2.78), (2.82), and (2.83), I performed the experiment described at the beginning of the section for both right and left subdomains, using both first and second order discretizations of the normal derivative. The errors for these, and all subsequent experiments, were measured in the infinity norm and calculated in a relative fashion. That is, they had the form

$$\frac{\|u_1 - u_2\|_\infty}{\|u_1\|_\infty}, \tag{2.84}$$

where u_1 is a known solution, and u_2 a computed solution. Tables 2.1, 2.2, 2.3, and 2.4 show the behavior of the error as the mesh was successively refined by a factor of 2.

Table 2.1: Robin Boundary Condition Error for forward difference in the left subdomain

h	error	factor
0.1	0.073775	—
0.05	0.035120	2.100659
0.025	0.017052	2.059623
0.0125	0.008417	2.025860
0.00625	0.004181	2.013007

Table 2.2: Robin Boundary Condition Error for centered difference in the left subdomain

h	error	factor
0.1	0.014815	—
0.05	0.003639	4.071685
0.025	0.000902	4.034285
0.0125	0.000225	4.000472
0.00625	0.000056	4.000750

Table 2.3: Robin Boundary Condition Error for forward difference in the right subdomain

h	error	factor
0.1	0.083585	—
0.05	0.044267	1.888198
0.025	0.022654	1.954096
0.0125	0.011479	1.973454
0.00625	0.005778	1.986821

Table 2.4: Robin Boundary Condition Error for centered difference in the right subdomain

h	error	factor
0.1	0.015097	—
0.05	0.003831	3.940538
0.025	0.000950	4.033638
0.0125	0.000237	4.004259
0.00625	0.000059	4.001064

Theory suggests that the use of first order accurate forward difference discretization of the normal derivative in the Robin condition should reduce the accuracy of the whole scheme to $\mathcal{O}(h)$, despite the second order accurate discretization of the Laplacian. This would translate into an error that improves by a factor of two for every doubling of the number of nodes in the discretization. However, the centered

difference discretization should remain as $\mathcal{O}(h^2)$ accurate, and the centered schemes were expected to produce an error that improves by a factor of four under the same refinement. Both the centered and forward schemes performed as predicted.

The one dimensional local optimized Schwarz iteration has some unique properties. In one dimension, for two subdomains, the optimal α *and* the iteration count are both equal to two. Perhaps more remarkably, the behaviour remains the same for all grid sizes. When more subdomains are added, the iteration count remains two, but the optimal α reflects the number of subdomains. In Appendix A, I have provided the analysis that explains the behaviour of the one dimensional experiments, but I do not include the details of the experiment here to avoid repetition with the two dimensional exposition that follows.

2.3.2 Two dimensional subdomain problems

In this section, I simply extend the one dimensional formulations covered in the previous section to two dimensions (Figure 2.6) and carry out the same experiment. In two dimensions I used the following solution u to get the first and second derivatives,

$$u = \sin(2\pi x - \frac{3\pi}{4}) \sin(2\pi y - \frac{3\pi}{4}) + 2. \tag{2.85}$$

Figure 2.7 demonstrates this solution and load function f that is found by taking two derivatives of the solution. The extension from one to two dimensional finite difference modelling is greatly simplified through the use of the Kronecker product. The following is a known property of the Kronecker product:

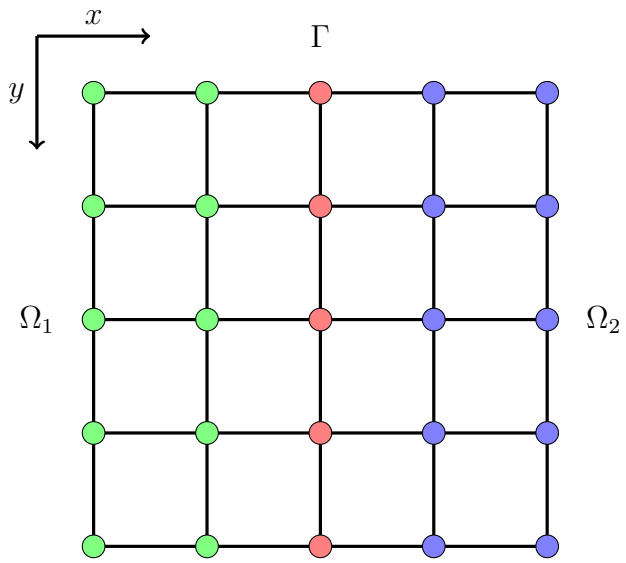

Figure 2.6: Two dimensional non-overlapping mesh partitioning into two subdomains Ω_1 and Ω_2 with interface Γ

$$\text{vec}(AXB) = (B^T \otimes A)\text{vec}(X). \tag{2.86}$$

Appendix B provides the details of how this property can be used to extend the one dimensional finite difference discretizations into two dimensional matrices for use with a vectorized right hand side. To summarize the conclusions of the appendix: the two dimensional matrix is created from the one dimensional matrix by the operation

$$(A_x \otimes I_y) + (I_x \otimes A_y), \tag{2.87}$$

where A_x and A_y are one dimensional discretizations of the PDE like those described in section 2.3.1. In a similar fashion, I can also form the two dimensional restriction and partition of unity matrices with the Kronecker product identity. If $R_{x,i}$ is the i^{th} restriction matrix for the x dimension, and $R_{y,i}$ is the i^{th} restriction in the y

dimension, then the two dimensional restriction is constructed by

$$R_i = (R_{x,i} \otimes I_y) + (I_x \otimes R_{y,i}), \tag{2.88}$$

and the same can be done for the i^{th} partition of unity matrix D_i. Although these matrices do not appear in the auxilliary variable algorithm, I rely on these matrices in my code to stitch together subdomain solutions so that I can compute the error against the single domain solution.

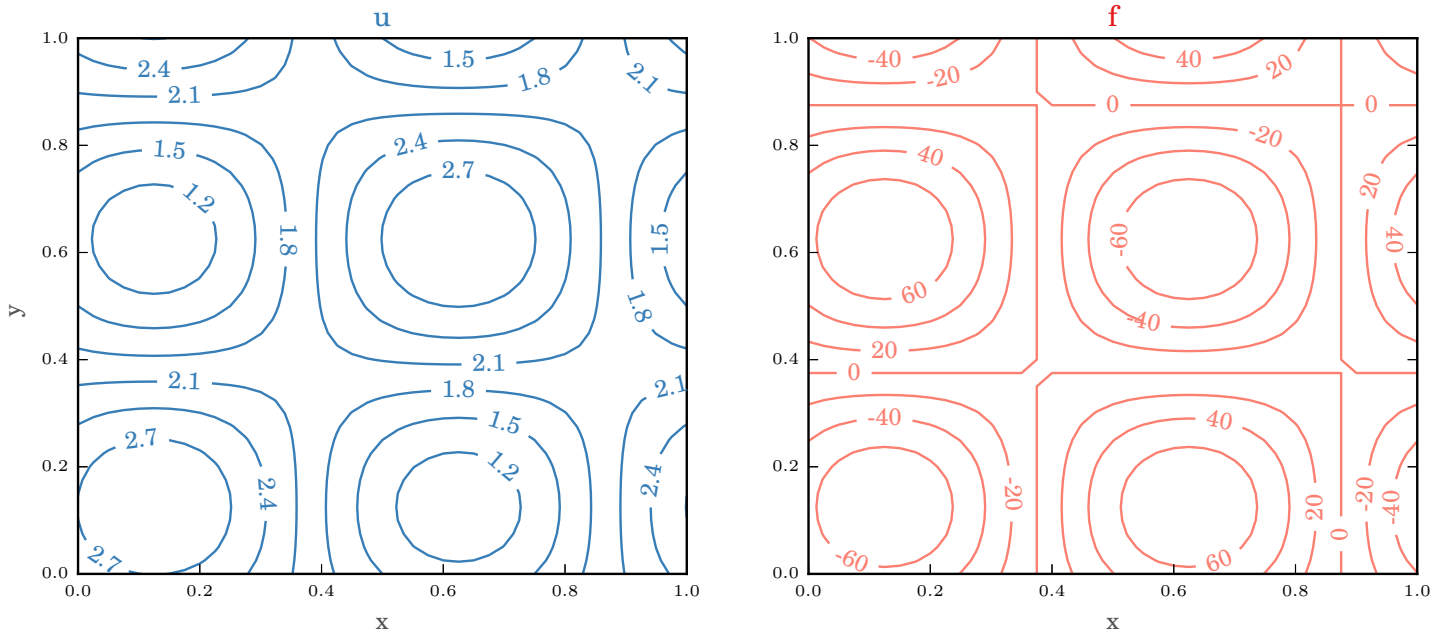

Figure 2.7: Function u and f used in the two dimensional experiments

In two dimensions, I discretized the right hand side for the Poisson problem as a matrix F, with the rows and columns representing discrete sampling of the domain in the x and y dimensions. Just like the one dimensional case where the load vector f was adjusted for the application of the Dirichlet and Robin BCs, I did the same to the load matrix F. Instead of removing an element from the first and last entry of the vector f, I removed a full column and two rows from the outside of the matrix F. Then, I updated the remaining exterior rows and columns, to reflect the deflation of the system matrix A during the application of the Dirichlet conditions as described for the one dimensional problem in Equations 2.63 and 2.64. Figure 2.8 shows the nodes represented by a matrix F for a small example of a left subdomain problem. The colored nodes indicate what is being updated in the boundary condition application. Red nodes indicate the nodes that are removed for the Dirichlet condition. Green nodes represent nodes that have a Dirichlet value subtracted from them. The blue nodes show coefficients involved in the Robin boundary condition and get updated as in Equation 2.75 or 2.80 for either a forward or centered difference. Finally, in this small example the two black nodes are the only interior nodes for which the full finite difference stencil (Equation (2.60)), applies.

The treatment for the centered difference method is nearly identical to that of the forward difference method described previously. The only alteration to the previous method is that the blue nodes in Figure 2.8 now receive an update in the form of Equation (2.80). I will leave this detail out, and also omit the description of the right subdomain forward and centered difference methods since they parallel those just described.

I performed the same experiment described in the last section on the two di-

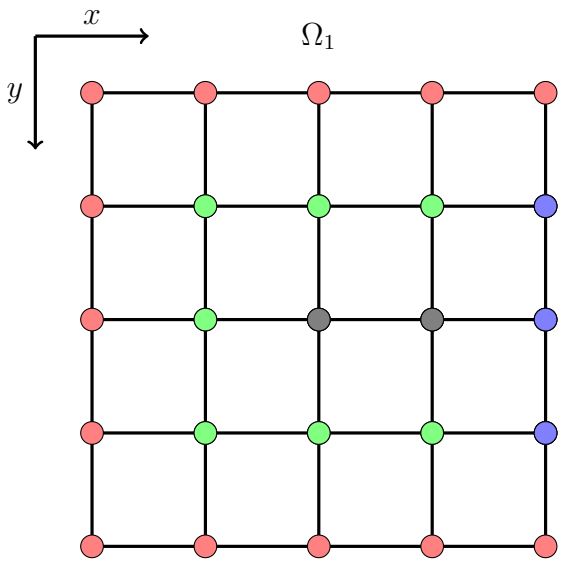

Figure 2.8: Coefficients involved in the application of the Dirichlet and Robin boundary equations

mensional discretizations resulting from the Kronecker product trick applied to the previously verified one dimensional discretizations, and with a right hand side produced by vectorizing the updated matrix F. Tables 2.5, 2.6, 2.7, and 2.8 summarize the error behaviour resulting from uniform grid refinement in a two-dimensional single domain solution for the left using forward and centered differences, and for the right using backward and centered differences.

In two dimensions, the error behaved as predicted by theory. The error decreased like $\mathcal{O}(h^2)$ for the centered difference Robin condition discretizations, and like $\mathcal{O}(h)$ for the forward difference Robin condition discretizations in both the left and right subdomain problems.

Table 2.5: Robin Boundary Condition Error for forward difference in the left subdomain

h	error	factor
0.1	0.100904	—
0.05	0.048585	2.076876
0.025	0.023496	2.067809
0.0125	0.011583	2.028536
0.00625	0.005745	2.016098

Table 2.6: Robin Boundary Condition Error for centered difference in the left subdomain

h	error	factor
0.1	0.013011	—
0.05	0.003224	4.035075
0.025	0.000806	3.998773
0.0125	0.000201	4.003159
0.00625	0.000050	3.999835

Table 2.7: Robin Boundary Condition Error for backward difference in the right subdomain

h	error	factor
0.1	0.094586	—
0.05	0.046506	2.033853
0.025	0.023225	2.002388
0.0125	0.011505	2.018631
0.00625	0.005724	2.010188

Table 2.8: Robin Boundary Condition Error for centered difference in the right subdomain

h	error	factor
0.1	0.006814	—
0.05	0.001704	3.998420
0.025	0.000430	3.964414
0.0125	0.000108	3.969812
0.00625	0.000027	3.981297

2.3.3 Two dimensional optimized Schwarz using the auxilliary variable method

Numerical experiments show that the optimized Schwarz method can be successfully applied as a local iteration on the subdomain solutions for the two dimensional Poisson problem. In the previous section, I constructed the left and right subdomain problems using finite differences in two dimensions. I also demonstrated that the problems could be solved with an error that behaves as $\mathcal{O}(h)$ and $\mathcal{O}(h^2)$ when I use a first and second order discretization of the Robin boundary condition. In this section I will use the tested second order discretization to carry out the optimized Schwarz method as a local iteration on the subdomain solutions u_1 and u_2. In section 2.2, I provided a two-step optimized Schwarz algorithm (Algorithm 8) that first solves Robin subdomain problems, then updates the Robin data. I will now present the numerical results of the two dimensional finite difference OSM as a local iteration using this auxilliary variable algorithm.

For the local iteration, I performed three separate experiments. The first consisted of a parameter sweep to find an optimal α for the Robin transmission condition. The second experiment used the optimal α discovered in the first experiment to find the optimal iteration count for the chosen grid size. The final experiment exhibited the grid dependence on the optimal α by sweeping through grid sizes, and α values. For all experiments, I considered the non-overlapping decomposition of a square mesh into two equal halves. The first two experiments did so for a 41×41 point global mesh, resulting in two 41×21 point meshes. Figure 2.9 shows that the optimal α is 12 for the 41×41 mesh. Figure 2.10 provides the optimal convergence behaviour

using $\alpha = 12$, and indicates that the problem can be solved to a precision of 10^{-6} in 25 iterations. Figure 2.11 demonstrates the growth of the optimality parameter under a shrinking grid size (blue curve). Gander (2006) analyzed a related elliptical PDE and showed that the optimal α could be related to the grid size h through a function with the form $Ch^{-1/3}$ where C is a constant scaling factor which depends on the smallest frequency possible for the given mesh. The numerical grid dependence provides a close match to the analytical form with $C = 3.65$ (red curve).

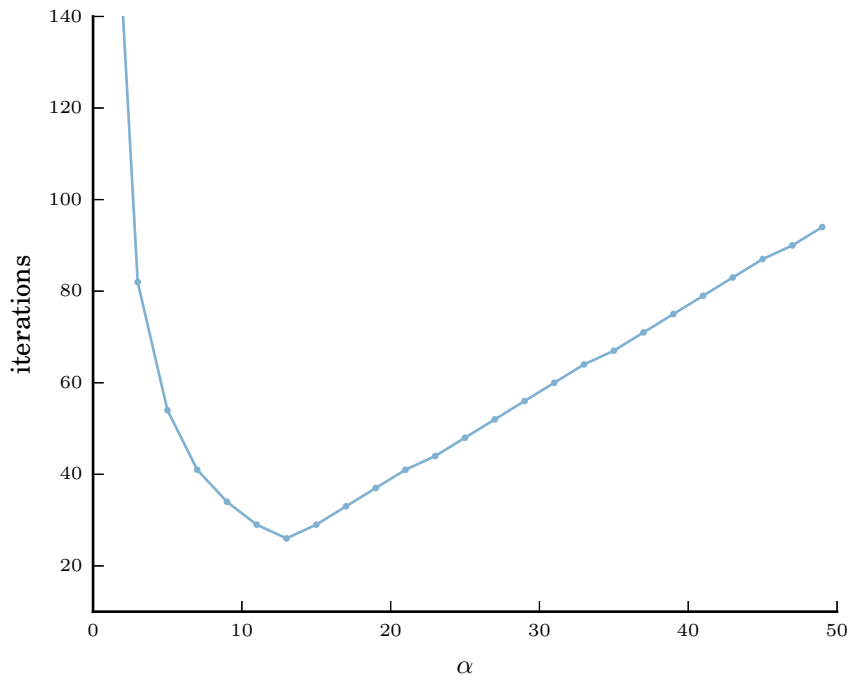

Figure 2.9: Optimal α search for a 41×41 point global mesh.

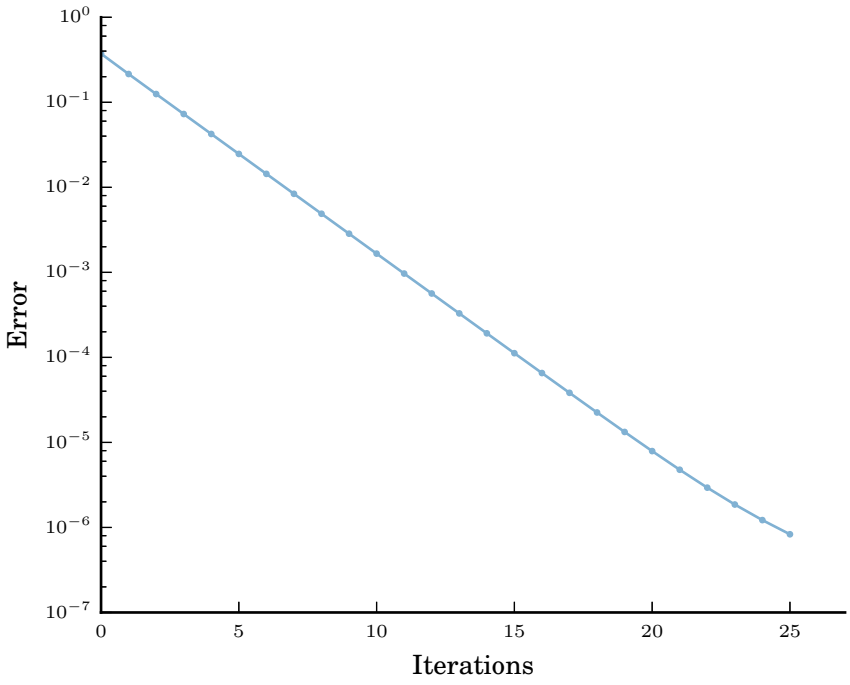

Figure 2.10: Error contraction with optimal $\alpha = 12$ with error measured by $\|u_{glob} - u_{DD}\|_2 / \|u_{glob}\|_2$.

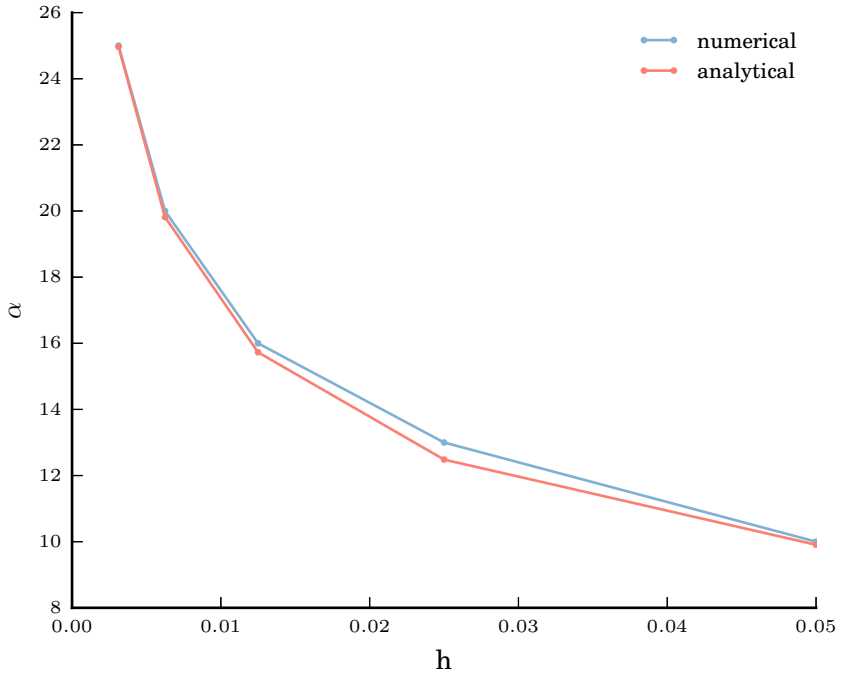

Figure 2.11: Optimal α grid dependence where h is the space between nodes in x and y dimensions

2.3.4 Two dimensional optimized Schwarz as a preconditioned global iteration

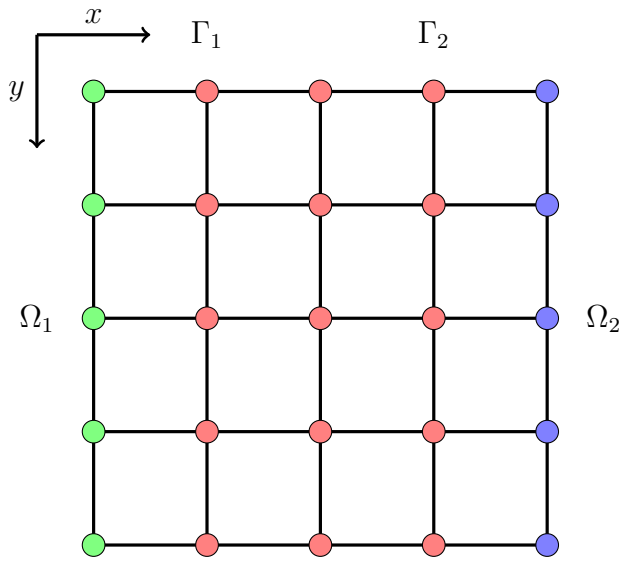

Figure 2.12: Two dimensional overlapping mesh partitioning into two subdomains Ω_1 and Ω_2 with interfaces Γ_1 and Γ_2

The following numerical experiments for the global iterations demonstrate that the fixed point iteration must only be used with overlapping subdomains, and can be problematic for poor choices of α, whereas the Krylov iteration produces a good error contraction when accelerated with the RAS preconditioner and better still with the Optimized Restricted Additive Schwarz (ORAS) preconditioner. I used an optimized Schwarz method to precondition both the global fixed point iteration and the global GMRES iteration using the same preconditioner routine (Equation (2.56) in Algorithm 9). The preconditioned global fixed point iteration resulted in a diverging error for the non-overlapping partitioning strategy used in previous experiments. Previous

work by St-Cyr et al. (2007) explains that ORAS preconditioned fixed point iteration produces an inconsistent matrix splitting and cannot be used in non-overlapping configurations. A consistent splitting implies that $A_i R_i u = R_i f + \sum_{j \neq i} B_{ij} R_j u$ for restriction matrix R_i, subdomain system matrix A_i, and transmission operator B_{ij}. The local iteration using the auxiliary variable method contains the transmission operator B_{ij} whose trace is the vector g_j containing all the Robin contributions $-2hg$ from, for example, Equation (2.80). In non-overlapping configurations, the transmission operator compensates for the fact that the normal derivative stencils built into A_i do not overlap. In the global ORAS fixed point iteration, the transmission operator is omitted and the splitting is no longer consistent. The FEniCS finite element assembly library that I use for the EM experiments allows a very limited range of options for the overlap. In order to make comparisons to the finite element experiments on the Poisson problem in later sections, I introduced a small overlap by adding a layer of mesh vertices to each subdomain on either side of the previously non-overlapping partition Γ. This created two new interfaces for each subdomain, Γ_1 and Γ_2 as can be seen in Figure 2.12. The following results are for the domain decomposition of the same 41×41 node global mesh, but which includes this new small overlap. Figure 2.13 compares the various different Schwarz algorithms including the local iteration from the last section with the error measured by taking the norm of the residual instead of the solution as in past experiments.

The local iteration using the auxiliary variable method and the global fixed point iteration for the OSM were found to be equivalent when the local iteration error was measured as the norm of the residual, just as the theory in Section 2.2 for the continuous RAS method hinted that they might be. When I converted the auxiliary

variable code to measure the residual instead of the solution error, I found that I needed to re-optimize α. The auxilliary variable method was optimized with $\alpha = 12$, whereas the global fixed point provided optimal convergence with $\alpha = 17$. The preconditioned GMRES iteration also provided a unique optimality condition, $\alpha = 3$. These examples demonstrate that the global preconditioned fixed point iteration performs as well as the auxiliary variable method while avoiding a somewhat tricky discretization of the transmission operator. However, to match the local iteration performance, a small overlap was required. The examples also demonstrate that the ORAS preconditioner outperforms the RAS preconditioner when using a finite difference discretization over a rectilinear mesh. More importantly, the examples make it clear that using the ORAS preconditioner within a Krylov iteration is the preferred approach.

The results provided in this section are in general agreement with the findings in Gander et al. (2001). I did not find any similar papers for the Poisson problem using a finite element discretization, so these finite difference results will be the only benchmark for the experiments that follow. In the next section, I will perform the same tests for a finite element discretization. I will build up to the domain decomposition experiments by performing the Robin boundary test as I did in this section. However, I will jump straight to the two dimensional case this time.

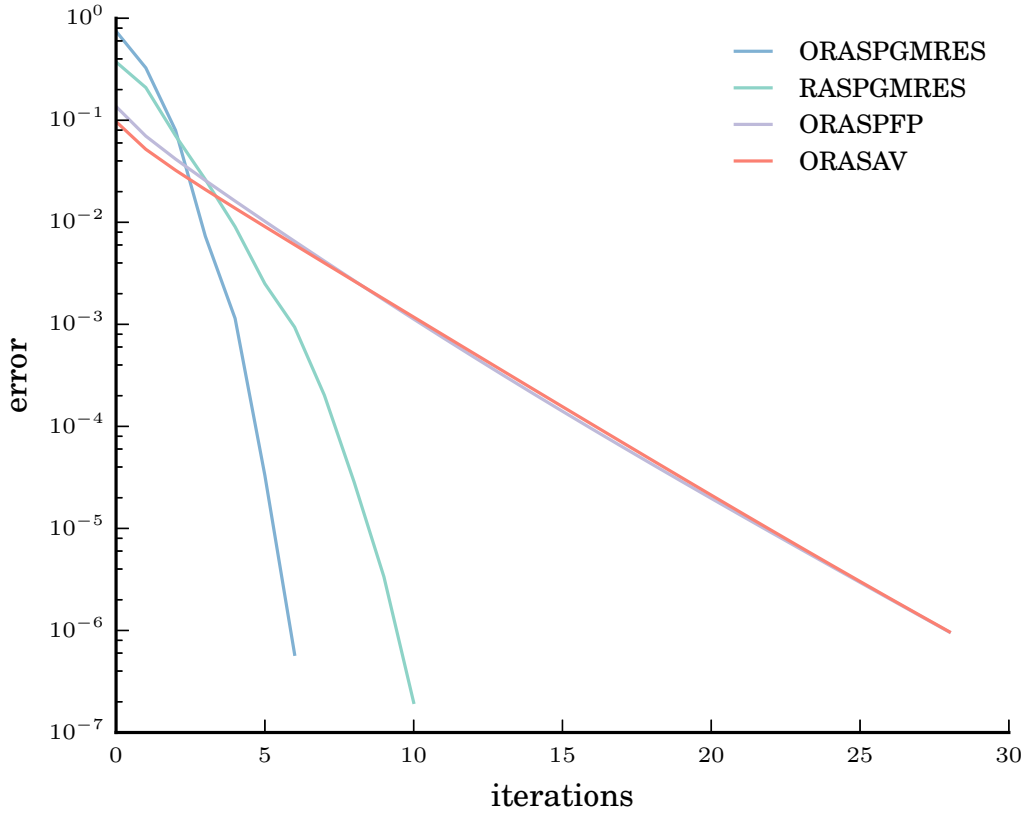

Figure 2.13: Comparison of the Optimized Schwarz method as a local iteration via the auxilliary variable method (ORASAV), a preconditioner for the global fixed point iteration (ORASPFP), and a preconditioner for the global Krylov iteration (ORASPGMRES), as well as the RAS preconditioner for the global Krylov iteration (RASPGMRES). The global preconditioned iterations all naturally measure error with the residual vector, by $\|Ax - b\|_2 / \|b\|_2$, so the auxiliary variable code was altered to comply with this error format

2.4 Finite element method

2.4.1 Two dimensional subdomain problems

The finite element method provides the flexibility to implement unstructured mesh refinement for the geophysical EM problem. The finite difference method was a good place to start for its ease of use and efficiency for problems, like the Poisson problem studied here, that allow for a simple structured discretization. However, in order to solve the geophysical EM problem efficiently and accurately, the discretization of the domain will need to be refined around the point-like source and the observation points. The finite element method is a natural choice for such a scenario, as it can handle unstructured meshes arising from local mesh refinement. In order to use finite elements to solve the EM geophysical problem with a Schwarz DD based preconditioner, I needed to build an understanding of the OSM within a finite element context. What follows is my description of the same experiments carried out for the Poisson problem in the last section, but using the finite element method to discretize both the global and subdomain equations.

The finite element method consists of finding the weak form of the equations, choosing an approximating subspace, and then selecting the best approximation from that subspace by way of the Galerkin method (see Gockenback (2006) for introductory theory and implementation). To use this tool to carry out the DD algorithms, I began by testing the Robin discretizations. Once again, the subdomain problem I would like to solve is

$$-\nabla^2 u = f \qquad \text{in } \Omega \tag{2.89}$$

$$\frac{\partial u}{\partial n} + \alpha u = g \qquad \text{on } \Gamma \tag{2.90}$$

$$u = \beta \qquad \text{on } \partial\Omega \setminus \Gamma. \tag{2.91}$$

I began the FEM by writing the weak form of the equations. I multiplied equation 2.89 by a test function v and integrated over the domain Ω, to get

$$\int_\Omega \nabla^2 u \cdot v = \int_\Omega f \cdot v. \tag{2.92}$$

Using integration by parts, I transformed this into

$$-\int_\Omega (\nabla u) \cdot (\nabla v) + \int_{\partial\Omega} \frac{\partial u}{\partial n} \cdot v = \int_\Omega f \cdot v. \tag{2.93}$$

I applied the Robin BC using the method outlined for the Neumann BC in Gockenback (2006), but with the extra step of rearranging the Robin BC to isolate the normal derivative. I began by multiplying equation 2.90 by a test function and integrating over the surface $\partial\Omega$. Then I split the surface $\partial\Omega$ into Dirichlet boundary and interface components and noted that the test function v vanishes on the Dirichlet boundary. This allowed me to apply the weak Robin boundary condition through substitution of

$$\int_{\partial\Omega} \frac{\partial u}{\partial n} \cdot v = \int_\Gamma \frac{\partial u}{\partial n} \cdot v = \int_\Gamma g \cdot v - \int_\Gamma \alpha u \cdot v, \tag{2.94}$$

into equation 2.93 to give,

$$-\int_{\Omega}(\nabla u)\cdot(\nabla v)-\int_{\Gamma}\alpha u\cdot v=\int_{\Omega}f\cdot v-\int_{\Gamma}g\cdot v. \tag{2.95}$$

I then wrote this in terms of the bilinear and linear forms, $a(u,v)$ and $L(v)$, as

$$a(u,v)=L(v). \tag{2.96}$$

The weak form of the continuous Poisson equation is stated as

$$\text{find a } \; u\in V \mid a(u,v)=L(v) \qquad \forall v\in V \tag{2.97}$$

where

$$V=\left\{v\in H^1(\Omega)\,\middle|\, v=0 \text{ on } \partial\Omega\right\} \tag{2.98}$$

and

$$H^1=\left\{v \;\middle|\; \int_{\Omega}|v|^2\,dx<\infty, \int_{\Omega}|\nabla v|^2\,dx<\infty\right\}. \tag{2.99}$$

To find a discrete solution u_h, I triangulated the domain Ω to produce a mesh T_h. Then, I used the Ritz-Galerkin approximation and searched the finite-dimensional subspace $V_h\subset V$ associated with T_h for the 'best' solution. The discrete weak form became

$$\text{find a } \; u_h\in V_h \mid a(u_h,v)=L(v) \qquad \forall v\in V_h. \tag{2.100}$$

I chose V_h to be the space of piece-wise linear functions with nodal basis functions

$$\phi_i = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{if } i \neq j, \end{cases} \tag{2.101}$$

and I assembled and solved the system $KU = F$ where

$$K_{ij} = a(\phi_i, \phi_j), \tag{2.102}$$

and

$$F_j = L(\phi_j). \tag{2.103}$$

To assemble this system, I looped through triangles in the mesh and accumulated contributions to the coefficients in the stiffness matrix K and load vector F. The contributions came from approximating the integrals of equation 2.95 by a Gaussian quadrature rule with an appropriate order for the integrand. In other words, I approximated the two dimensional integral of a function F over the triangle k by

$$\iint_k F(x,y)dxdy \approx A_k \sum_{i=0}^{N} w_i F(P(\xi_i, \eta_i), Q(\xi_i, \eta_i)), \tag{2.104}$$

where I used the transformations P and Q to perform the approximation on the reference triangle (Figure 2.14). This step allowed me to use Gauss points and weights, ξ_i, η_i and w_i, that have previously been established for the reference triangle (Gockenback, 2006). I computed the area of the triangle by the formula

$$A_k = \frac{(x_1 - x_0) \cdot (y_2 - y_0) - (x_2 - x_0) \cdot (y_1 - y_0)}{2}. \tag{2.105}$$

The same approach was taken for one dimensional boundary integrals by transforming to the reference interval and computing the length of the interval.

The integrals of the bilinear form containing only linear basis functions are approximated exactly with a linear quadrature rule, but I used a second order quadrature rule for the load function integral due to the non-linear character of the function f.

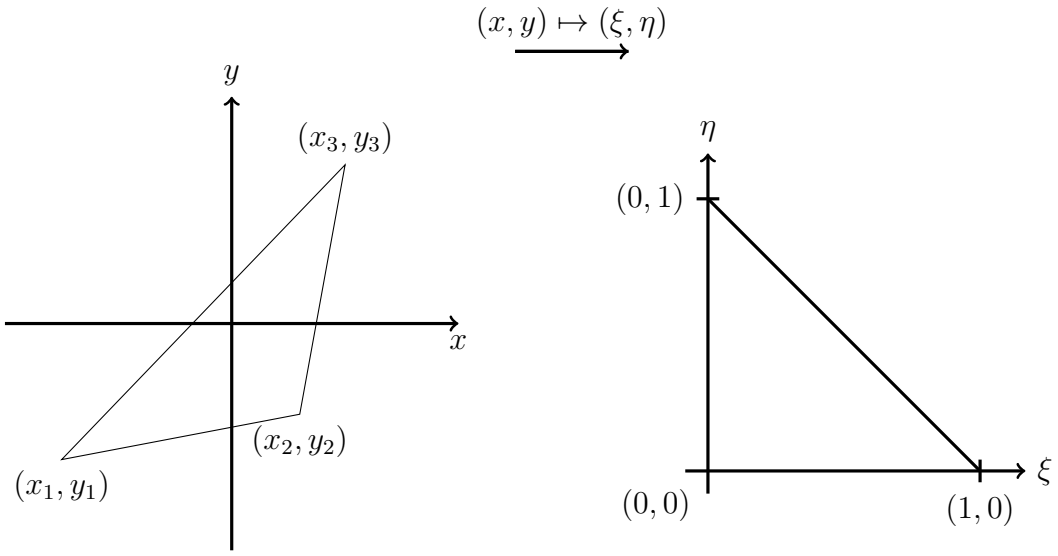

Figure 2.14: Reference triangle transformation

The subdomain problem required function definitions for the load f, Robin data g, i^{th} basis function ϕ_i and gradient of the i^{th} basis function $\nabla\phi_i$. The load and Robin data function definitions were a straightforward translation of the mathematical function into Python syntax. I computed basis functions individually using the geometry of each triangle. I did this by inverting the matrix

$$\begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix}^{-1} = \begin{bmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3, \end{bmatrix} \tag{2.106}$$

as was done in Gockenback (2006), to solve for the coefficients a_i, b_i, c_i of the i^{th} basis function $a_i + b_i x + c_i y$. The basis function gradients were extracted from the inverted matrix as $\nabla\phi_i = [b_i, c_i]^T$. Once assembly was finished, I applied the boundary conditions using the strong form. That is, I enforced the condition that $u = \beta$ by replacing rows of the assembled matrix with rows of the identity matrix and replacing the assembled load vector rows with the Dirichlet values β.

I reproduced the results of the FEniCS codes that generated the Robin boundary condition test and the local iteration by auxiliary variable OSM with my own pure Python assembly routines using the procedure outlined at the beginning of this chapter. Since one of the goals for my own pure Python code was to investigate the features in FEniCS that I felt I needed a better understanding of, the code was essentially left in a state that matches the performance of the FEniCS code. I do not include results from my own code that match those achieved with FEniCS, but I do include the results of the Robin boundary test since they deviated slightly from the FEniCS implementation. The critical finite element assembly routines from my code are provided in Appendix C for reference. My assembly code established that the FEniCS library uses a methodology similar enough to that described in Gockenback (2006) that I could replicate the main numerical properties of the finite element discretizations of the Robin test and OSM. The difference between a FEniCS assembly code and a homespun assembly code like mine is that FEniCS is optimized and generalized as I will discuss next.

The FEniCS library provides a professional implementation of the various components of the finite element assembly routines described above. Finite element assembly libraries provide fast optimized code that avoid the many non-trivial tasks involved

in assembly from scratch. In the description of my finite element assembly routine, I avoided discussing details like the creation of the structured mesh, and its associated data structures which store connectivity between various geometrical entities. However, this is in fact a significant portion of the work involved in performing the assembly from scratch. Not only is the book-keeping for the mesh an onerous task, but so is the book-keeping that allows mapping from the geometrical entities of the mesh to their associated indices in the matrix (ie, degrees of freedom). The meshing for even the most basic structured mesh occupied nearly 100 lines of my code. More importantly, these lines of code consist mostly of unavoidable loops over the mesh elements. In Python, loops are known to be very slow compared to compiled languages. This is where a professional finite element assembly package is so beneficial. In FEniCS, I get highly optimized assembly routines that rely heavily on wrapped, compiled C++ code. I also get to replace those 100 lines of code with just the one line: `mesh = UnitSquareMesh(nx,nx)`, the result of which is a structured triangulation of the unit square, like the one depicted in Figure 2.15 with `nx` intervals in each direction.

I will now demonstrate how I used FEniCS to replicate the components of my own assembly code, while highlighting some of the advantages in using the library approach. In my description of the finite element method, I previously stated that I would choose the space of linear functions with nodal basis functions. This choice was deeply embedded into my assembly code by providing the functional form of the basis functions to the Gaussian quadrature routines. In FEniCS, this choice is made at a very high level of abstraction through the `FunctionSpace()` object. That meant that for my Poisson problem, I could create an instance of this object via `V =`

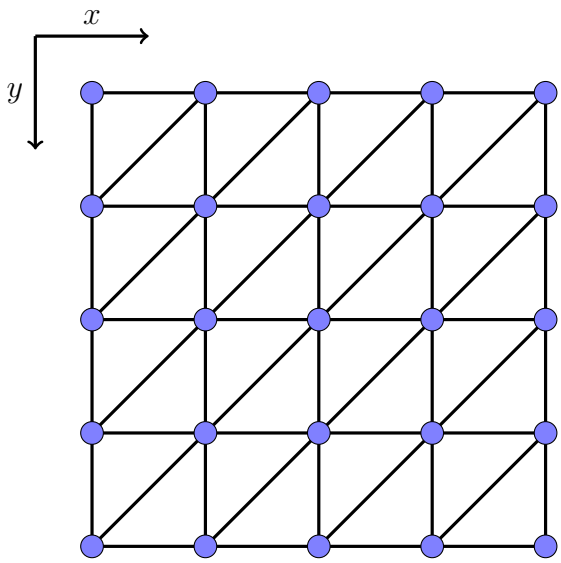

Figure 2.15: Finite element triangulation of the unit square

`FunctionSpace(mesh, Lagrange, 1)` for the space of piece-wise linear functions. To assemble the system of equations for the subdomain problem, I first defined the test and trial functions to be used in the declaration of the bilinear and linear forms. Then, I transcribed the mathematical description of the load and Robin data functions into C++ syntax. Finally, I declared the forms using test and trial functions along with the load and Robin data functions, and made a call to `assemble()`. These steps are shown in Listing 2.1.

```
1  u = TrialFunction(V)
2  v = TestFunction(V)
3
4  f = Expression('(-8.*pi*pi) * ( sin((2.0*pi*x[0]) - (0.75*pi)) * \
5                  sin((2.0*pi*x[1]) - (0.75*pi)) )', \
6                  degree=2)
7
```

```
g = Expression('(2.0*pi*cos(2.0*pi*x[0] - (0.75*pi))*sin(2*pi*x[1] -
  (0.75*pi))) + \
                alpha*((sin((2.0*pi*x[0]) - (0.75*pi)) * \
                sin((2.0*pi*x[1]) - (0.75*pi))) + 2)', \
                alpha=alpha, \
                degree=2)

e = - inner(grad(u), grad(v))*dx - inner(alpha*u,v)*ds(1)
L =   inner(f,v)*dx - inner(g,v)*ds(1)

A = assemble(a)
b = assemble(L)
```

Listing 2.1: Finite element assembly of the subdomain problem in FEniCS.

To assemble the Robin boundary integrals, I relied on an additional FEniCS object `Measure()` which was instantiated with a function that marks the appropriate boundary so that the assembler knew to integrate over the interface only. These details were contained within the form definition as `ds(1)` where the boundary marking function and `Measure()` objects were created by the code shown in Listing 2.2

```
class Interface(SubDomain):
    def inside(self, x, on_boundary):
        tol = 1E-14
        return abs(1.0 - x[0]) < tol

interface = Interface()
markers  = MeshFunction("size_t", mesh, 1)
interface.mark(markers, 1)
```

```
9
10  ds = Measure("ds")[markers]
```

Listing 2.2: Boundary marking and measure instatiation for the interface integrals.

The last step that I described in my assembly routine was to apply the strong form of the Dirichlet boundary condition. In FEniCS, I accomplished this by creating a `DirichletBC()` object, again using the boundary marking function (Listing 2.3).

```
1   class Global_outer_boundary(SubDomain):
2       def inside(self, x, on_boundary):
3           tol = 1E-14
4           return on_boundary and \
5                  (abs(x[0]) < tol or \
6                   abs(x[1]) < tol or \
7                   abs(1.0 - x[1]) < tol)
8
9   global_outer_boundary = Global_outer_boundary()
10  global_outer_boundary.mark(markers,2)
11
12  bc = DirichletBC(V, ufunc, markers, 2)
```

Listing 2.3: Dirichlet boundary setup.

I supplied the function `ufunc` for the subdomain problem in the same way that `f` and `g` were defined for the bilinear and linear form definitions. Once this was complete, I applied the strong form Dirichlet boundary condition via `bc.apply(A)`

I performed the Robin test using the FEniCS code described above and found that this produced the correct error behaviour. When I solved the resulting subdomain problem for the same series of meshes as was done for the finite difference discretiza-

tion and my own assembly code, I also found that the error decreased like $\mathcal{O}(h^2)$. The errors and contraction rates of both the FEniCS assembly code and my assembly code can be seen in Table 2.9. Although the errors and rates did not match, the difference is explained by how each code evaluated the function `g` within the Gaussian quadrature routine, as discussed earlier.

Table 2.9: Robin Boundary Condition Error for finite elements using FEniCS and my own assembly code

h	FEniCS error	factor	my assembly error	factor
0.1	0.012958	—	0.015056	—
0.05	0.003295	3.932681	0.003815	3.946615
0.025	0.000828	3.979381	0.000949	4.019510
0.0125	0.000207	3.996521	0.000237	3.996148
0.00625	0.000052	3.998863	0.000059	3.999188

2.4.2 Two dimensional optimized Schwarz using the auxilliary variable method

The finite element subdomain solver tested in the last section can be bootstrapped to carry out the auxilliary variable OSM. In the last section, I demonstrated and tested a finite element subdomain solver with Robin transmission conditions that is suitable for an optimized Schwarz domain decomposition algorithm. I will now use those subdomain solvers within Algorithm 8 to perform the local iteration variant

of the OSM. Later, I will also demonstrate the global iteration (Algorithm 9), and preconditioned Krylov methods using FEniCS, but this will require a transition to parallel computing to take advantage of the parallel data structures created when any FEniCS code is run in Message Passing Interface (MPI) mode. While these data structures proved to be useful for programming the OSM algorithms, FEniCS was by no means designed to implement DD algorithms. FEniCS is a finite element assembly library foremost. I used FEniCS to assemble both global and local problems, and I pieced the other components of the algorithms together from FEniCS functions that happened to be suitable for the DD. For now, I must introduce a few extra steps needed to carry out the local iteration by the auxilliary variable method so that I can compare results to those found using the finite difference discretization.

The finite element discretization technique requires that the original local iteration algorithm be modified slightly. To start the bootstrap, I need to supply an initial guess for the value of the Robin condition. Since g is multiplied by a test function and integrated in the finite element method, I can assume this to be an initial guess for the integrated g. This is a reasonable assumption whenever the initial guess is simply the zeros vector, like in my case. After the subdomain solves, the finite element implementation requires an extra step consisting of multiplying the subdomain solution by a test function and integrating along the interface. Algorithm 10 includes these new details.

The auxilliary variable OSM using finite elements produced different results for the same experiments carried out using finite differences. I used the new Algorithm 10 to carry out the same experiments for the finite element method as I did for the finite difference discretization. In Figure 2.16, the optimal α is 25, whereas for finite

Algorithm 10 Optimized Schwarz as a local iteration with finite elements

while $\int_\Gamma g_1^k \cdot v \neq \int_\Gamma g_2^k \cdot v$ **do**

for $i = \{1, 2\}$ **do**

solve in parallel:

$$-\int_{\Omega_i} (\nabla u_i) \cdot \nabla v_i - \int_{\Gamma_i} \alpha u_i|_{\Gamma_i} \cdot v_i = \int_{\Omega_i} f_i \cdot v_i - \int_{\Gamma_i} g_i \cdot v_i \tag{2.107}$$

update in parallel:

$$\int_\Gamma g_i^{k+1} \cdot v_i = 2\alpha \int_\Gamma u_{3-i}^k|_\Gamma \cdot v_i - \int_\Gamma -g_{3-i}^k \cdot v_i \tag{2.108}$$

end for

end while

difference, it was 13. This figure also shows that the α curve for the FE discretization is much flatter than for FD, meaning there are a larger number of α values for which the convergence would be near optimal. The iteration count for the finite element implementation, as seen in Figure 2.17, is also twice as high as the finite difference implementation. This could be related to the fact that, although the discretizations share the number and orientation of mesh nodes, the finite element mesh bisects the square cells of the finite difference mesh, effectively cutting the area of the cell in two. I was surprised to find that I could not locate any discussion of this in the literature, so it remains an open question how to best compare finite element and finite difference DD results. Finally, in Figure 2.18, I studied the grid dependence of the optimal α value. The finite element result differred from that of the finite difference implementation and I was not able to produce as close of a fit to the analytical form $Ch^{-1/3}$ in Gander (2006). The optimal α and the scaling factor for

the analytical curve C found for the finite element discretization are roughly double those found with finite differences. Additionally, the curve is less steep between the first two grid size samples and lags the analytical curve initially, then it steepens to the point where it finishes at a lower optimal α than the analytical curve. Overall, the numerical curve is at least comparable to the analytical curve in that it demonstrates a decaying optimal α with increasing grid size, and the rate of the decay is not too far off from the analytical curve $Ch^{-1/3}$ when $C = 7.18$.

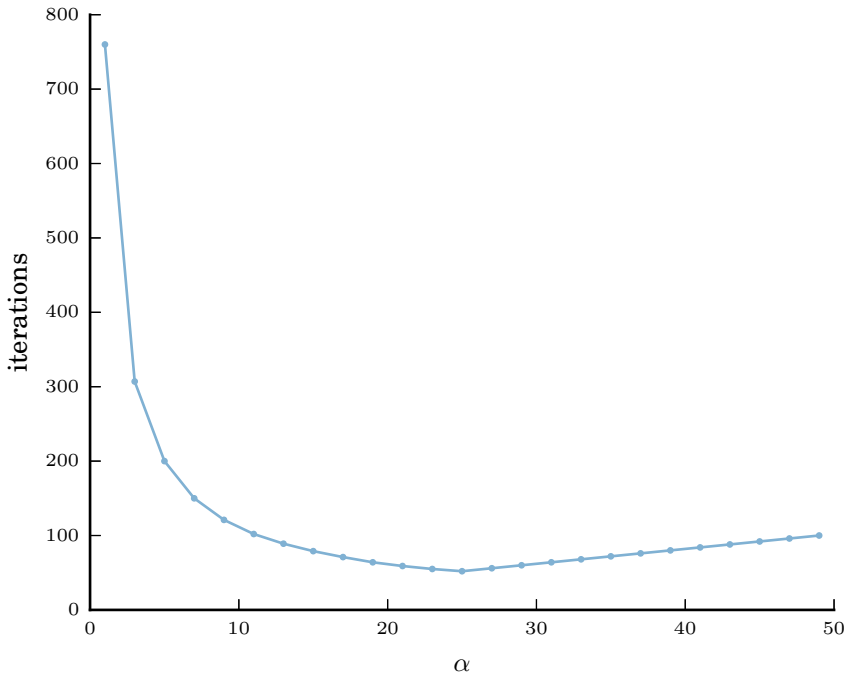

Figure 2.16: Optimal α search

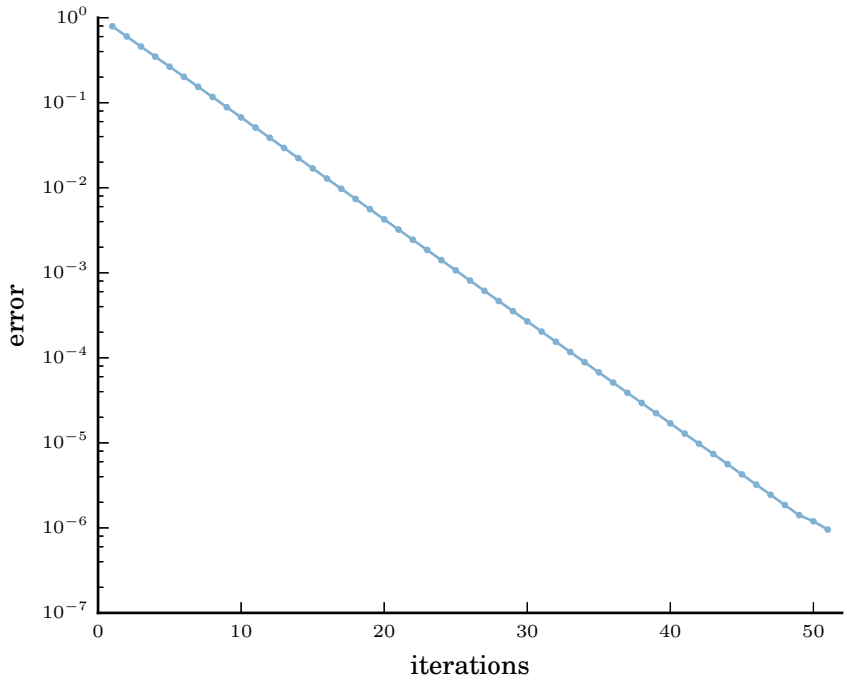

Figure 2.17: Convergence rate for optimized Schwarz using FEniCS for assembly with $\alpha = 25$

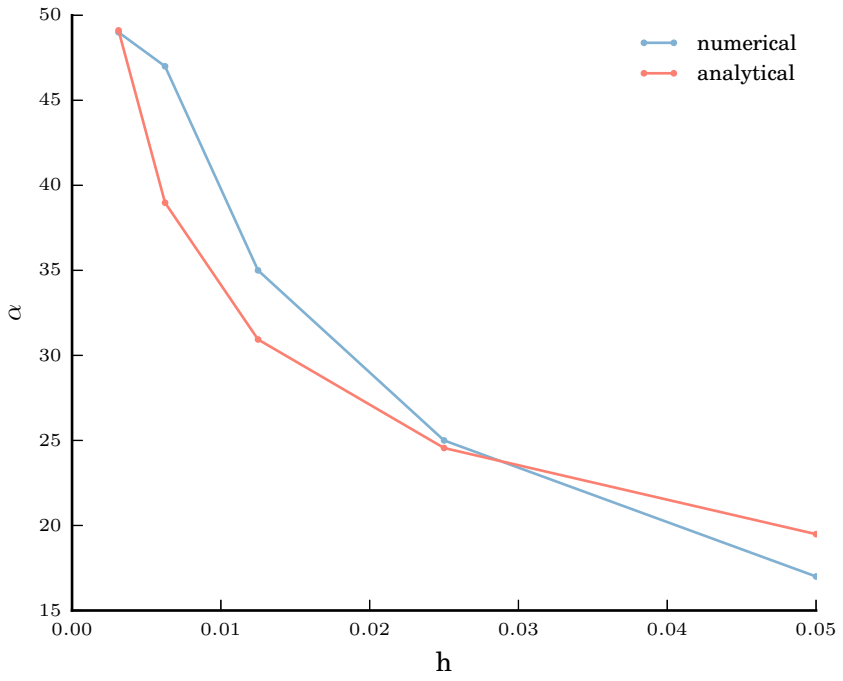

Figure 2.18: Optimal α grid dependence

2.4.3 Parallel two dimensional optimized Schwarz as a preconditioned global iteration

The global iteration variants of the OSM are implemented by replicating the action of the extension and partition of unity operators, which is simplified using FEniCS's parallel capabilities. In the last section, I demonstrated the behaviour of the local iteration using the auxilliary variable method discretized by finite elements. In Section 2.2, I presented an algorithm (Algorithm 9) that used algebraic extension and partition of unity operators to perform a global iteration. I stated then that the fixed point preconditioner could also be used as a preconditioner to a Krylov iteration. In this section, I break down the algebraic operations in this preconditioner and demonstrate the equivalent operations I used to precondition the global fixed point and Krylov iterations. These algebraic actions are most easily performed using parallel data structures provided by FEniCS – itself piggy-backing on MPI and PETSc functions. For this reason, from this point on, all of the algorithms will be carried out in parallel using FEniCS. FEniCS natively supports parallel programming using an MPI model that mirrors the back-end linear algebra package PETSc. This means that all of the assembly code I presented while discussing the subdomain problems is also valid for a parallel run. I can simply initiate a parallel run at the command line using `mpirun` and FEniCS's assembly routines will produce a distributed PETSc stiffness matrix and load vector based on a partitioning of the nodes of the mesh provided by the METIS package. The METIS package uses a parallel multilevel k-way graph-partitioning algorithm (LaSalle and Karypis, 2013) which performs partitioning in a way that seeks to create load balancing meshes for MPI code. The partitions

created by the METIS library do not necessarily create square or rectangular interfaces, even in a structured mesh setting. A typical METIS partitioned mesh might look something like Figure 2.19.

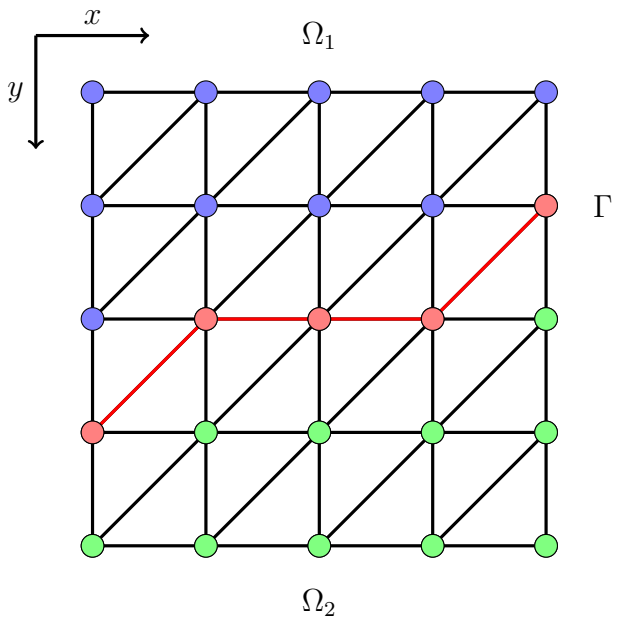

Figure 2.19: Parallel mesh generation

In order to define a custom preconditioner in FEniCS, I inherited from the `PETScUserPreconditioner()` class and overloaded the `solve()` function that operates on pre-declared, partitioned PETSc input and output vectors. The pseudocode in Algorithm 11 outlines the parallel steps involved in setting the output vector from the input, that forms the new definition for the `solve()` function, and which replicates the action of the algebraic operations in the optimized Schwarz preconditioner of Algorithm 9. In line 2, I use the `gather()` MPI routine to collect DOFs of the input vector r for subdomains with an arbitrary overlap; the operation is equivalent

to the algebraic restriction $R_i r^k$. In line 3, I map the gathered DOFs into the local DOF order required for use with local subdomain matrices A_r. In line 4, I replicate the subdomain solve operations $A_{r_i}^{-1} R_i r^k$ by solving the subdomain problems with the local subdomain matrices and gathered/mapped local residual vectors. I replicate the action of the partition of unity matrix $D_i A_{r_i}^{-1} R_i r^k$ in line 5. Finally, in line 6, I map subdomain solutions into global DOF order and accumulate them into the output vector to complete the action of applying the optimized Schwarz preconditioner on an arbitrary vector; it is equivalent to the prolongation and summation $\sum_i R_i^T D_i A_{r_i}^{-1} R_i r^k$. Since the new definition for the `solve()` method is general, I can use the inherited preconditioner class to either carry out the global fixed point iteration, or provide an outer Krylov iteration with the preconditioning routine for constructing either left of right preconditioning as described in Saad (1995). In order to accomplish any of the steps of the pseudocode in Algorithm 11 using FEniCS, I had to create local meshes and associated data structures to get around the lack of FEniCS functionality for solving subproblems within the global mesh data structures.

Algorithm 11 Overloading solve() for input vector **r**, and output vector **Mr**

1: **function** solve(**Mr**, **r**)
2: $z \leftarrow$ gather subdomain DOFs from **r**
3: $b \leftarrow$ map z into local DOF order
4: **Solve:** $x \leftarrow A_r^{-1} b$
5: $u \leftarrow x$ scaled by partition of unity
6: **Mr** $\leftarrow$ Accumulate u in global DOF order
7: **end function**

Notice that there was nothing explicitly parallel about the instructions in Algorithm 11. That's because FEniCS functions are written in such a way that they may be used in serial or parallel without altering the way they are called. It is assumed that when a function is called on a vector in parallel, that the instructions provided by the function are carried out on the local part of the vector when run in parallel. Furthermore, when I call upon a variable in a parallel code, I need to think about the program returning only the local part. There is no better example than in the construction of the subdomain meshes. In order to construct the required subdomain components of the Schwarz algorithms, I needed to first form separate meshes from the local and shared portions of the global partitioned mesh. To create these, I initialized new, empty, local meshes, iterated over the local part of the global mesh and copied vertices and cells into the new mesh. These actions were performed by the FEniCS code in Listing 2.4.

```
1  mesh_local = Mesh(mpi_comm_self())
2  mesh_editor = MeshEditor()
3  mesh_editor.open(mesh_local, tdim, tdim)
4  mesh_editor.init_vertices(mesh.num_vertices())
5  mesh_editor.init_cells(mesh.num_cells())
6
7  cell_num = 0
8  used_verts = []
9  for c in mesh.cells():
10     for v in c:
11         if v.index() not in used_verts:
12             mesh_editor.add_vertex_global(v.index(),
13                                           v.global_index(),
```

```
14                                                  np.array([v.x(0),
15                                                            v.x(1)])
16
17          mesh_editor.add_cell(cell_num, c[0], c[1], c[2])
18          used_verts.append(c[0])
19          used_verts.append(c[1])
20          used_verts.append(c[2])
21          cell_num += 1
22
23  mesh_editor.close()
```

Listing 2.4: Copying local parts of the global distributed mesh to create local meshes

I created the new mesh with `mpi_comm_self()` so that the linear algebra backend did not attempt to split the mesh over the processors. I used the global version of the `add_vertex()` because, as it turned out, this was an important simplification for the Poisson problem. More importantly, I also found that this step was required for the EM problem. I discovered the hard way (ie, after several weeks of debugging) that the numbering of incident mesh entities was only preserved when I used the global version. When I used a local version, the numbering of the edges in the local mesh and the orientations of the edges within each cell did not coincide. For the EM problem, this was totally destructive to the DD formulation since the curl of the basis functions depends on the cell orientations. For the Poisson problem, using the global version did not cost anything extra, and it saved me from creating a map between the local and global edge indices.

Solutions generated in the local meshes will not, in general, have the same ordering as their global counterparts, so I needed to create a mapping of local to global DOFs.

Lines 3 and 6 of Algorithm 11 both require a means to map from a local DOF ordered vector to a global DOF ordered vector. I accomplished this by looping through the global mesh once and storing a dictionary that had a sorted vertex tuple as a key and a DOF index as a value. Then, I looped through the local mesh and related a local DOF index to a global one through the sorted vertex tuple. As a matter of convenience, I built the partition of unity operator at the same time. Both of these task were accomplished by the loops in Listing 2.5.

```
1  partition_of_unity = [1] * mesh.num_edges()
2  dof_g2l = [0] * mesh.num_edges()
3  for ci in range(mesh_local.num_cells()):
4    cg = Cell(mesh, ci)
5    cl = Cell(mesh_local,ci)
6    cell_dofs = [dofmap.local_to_global_index(i) for i in dofmap.
   cell_dofs(ci)]
7    cell_dofs_local = dofmap_local.cell_dofs(ci)
8
9    edge_data = {}
10   for e in edges(cg):
11       vs = []
12       for v in vertices(e):
13           vs.append(v.index())
14       vk = tuple(sorted(vs))
15
16       edge_data[vk] = [cell_dofs[cg.index(e)],
17                        1./(len(e.sharing_processes()) + 1)]
18
19   for e in edges(cl):
```

```
        vs = []
        for v in vertices(e):
            vs.append(v.index())
        vk = tuple(sorted(vs))

        dof_g2l[cell_dofs_local[cl.index(e)]] = edge_data[vk][0]
        partition_of_unity[cell_dofs_local[cl.index(e)]] = edge_data[vk
    ][1]

dofg2l_IS = PETSc.IS().createGeneral(dof_g2l, comm=PETSc.COMM_SELF)
```

Listing 2.5: Global to local DOF map and parition of unity

There is often more than one way to carry out an operation in FEniCS, and care should be taken to ensure that, whatever method is selected, it acts on the appropriate DOFs. In FEniCS, a vector is just a view into a vector in the linear algebra back-end format. Since I use PETSc as my back-end, the vector layout is that of a partitioned PETSc vector. The layout depends on which ghost mode I choose through `parameters["ghost_mode"] = "shared_vertex"`. With `shared_vertex` selected, I had access to a layer of cells and the associated degrees of freedom on the far side of the true mesh interface, as shown in Figure 2.20. The reason I bring this up is because it is important to keep in mind what local part is being accessed in many operations. For example, if I used the usual mesh entity iteration scheme recommended in the FEniCS documentation (Listing 2.6).

```
for c in Cells(mesh):
    for e in Edges(mesh):
```

Listing 2.6: FEniCS recommended mesh entity iteration procedure

in the construction of the global to local DOF map, I would get only the owned entities. However, if I used the iteration `for ci in range(mesh.num_cells()):`, I would also get access to the overlapping cells in the ghost range. This insight was crucial for constructing submeshes that overlap. For this same reason, I could not simply ask for the local part of the incoming vector. Instead, I was required to `gather()` local vectors to gain access to both owned and shared DOFs provided with the `shared_vertex` option activated.

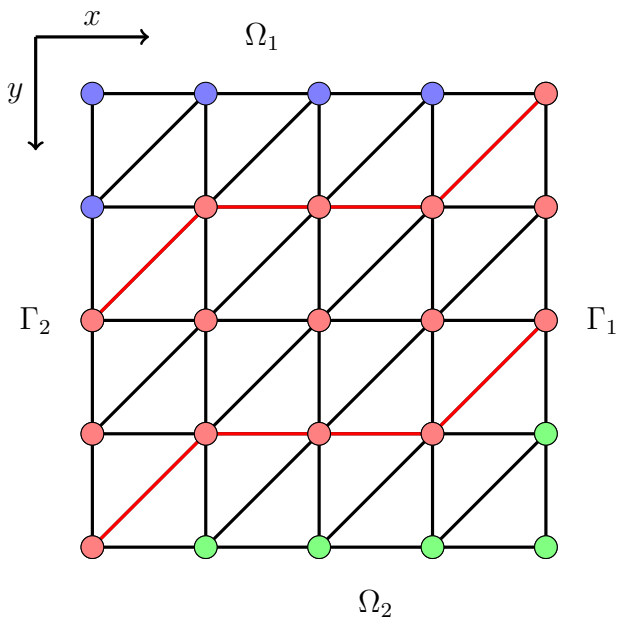

Figure 2.20: Shared Vertices

I created the partition of unity function using the `sharing_processes()` method available for the `MeshEntity` class. The function provides a list of processes that share the i^{th} edge, whose length indicates the number of sharing processes. To set the weight for the i^{th} edge, I map the edge index to the corresponding DOF index

and set that entry of the partition of unity array to $1/n + 1$, where n is the length of the shared processes list. This choice of weighting resulted in an averaging of the overlapping DOFs when I accumulate local solution vectors. At the end of the code snippet, I included a step that creates a PETSc index set (IS) from the global to local DOF map. The index set let me perform all of the preconditioner steps in the PETSc back-end. The index set was also useful for implementing the RAS method where I obtained the subdomain matrices by restriction of the global stiffness matrix. Listing 2.7 accomplishes the restriction $A_i = RAR^T$ using the Python wrapper for PETSc (PETSc4py) function `getSubMatrices()`.

```
Amat = as_backend_type(A).mat()
A_restr = Amat.getSubMatrices(dofg2l_IS, dofg2l_IS, submats=None)[0]
```

Listing 2.7: Restricting the global matrix for the RAS method

I have now described all of the components required to implement either a RAS or ORAS preconditioner. Listing 2.8 shows the actual code that I used to overload `solve()` function with the action of applying a Schwarz based (in this case ORAS) preconditioner to the input vector.

```
solver_local = LUSolver(A_local)
solver_local.parameters['reuse_factorization'] = True
class SchwarzPreconditioner(PETScUserPreconditioner):

  def __init__(self):
      PETScUserPreconditioner.__init__(self)
      self.dofg2l_IS = dofg2l_IS
      self.V_local = V_local
      self.partition_of_unity = partition_of_unity
```

```
        self.solver_local = solver_local

    def solve(self, Minvr, r):

        # Equivalent to lines 2 and 3 of pseudocode
        z = r.vec().getSubVector(self.dofg2l_IS)

        # Equivalent to line 4 of pseudocode
        u = Function(self.V_local)
        self.solver.solve(u.vector(), PETScVector(z))

        # Equivalent to line 5 of pseudocode
        ug = self.partition_of_unity * (u.vector().array())

        # Equivalent to line 6 of pseudocode
        Minvr.vec().setValues(self.dofg2l_IS, ug, addv=True)
        Minvr.vec().assemblyBegin()
        Minvr.vec().assemblyEnd()

ORAS = SchwarzPreconditioner(dofg2l_IS, V_local, partition_of_unity,
  solver_local))
```

Listing 2.8: User defined OSM preconditioner

All of the steps in Algorithm 11 provided at the beginning of this section are present in Listing 2.8. I used comments in the code snippet to associate lines of actual code with lines of the pseudocode for the algebraic operations implied by the optimized Schwarz preconditioner (Equation (2.56)). It might not be particularly clear how the proper stitching of subdomain solutions is accomplished within this

code. The partition of unity is applied to all overlapping DOFs prior to setting the global vector, and the `setValues()` function is called with the flag `addv=True` which means that DOFs are accumulated into the global outgoing vector and overlapping entries are added. The combination of this addition and the partition of unity weighting are designed so that the average of the overlapping entries are assigned to the global vector.

There are many ways to solve the subdomain problems in a DD method, but using a direct method allowed me to reuse factorizations and reduce the cost per iteration. I chose to solve the subdomain problems exactly by LU factorization and found that, by reusing the factorization, I cut the cost of each iteration by a significant amount. This choice is also motivated by the fact that EM subdomain problems will have the same ill-conditioning issue that affects their global counterpart. In this case, a LU factorization of the subdomain problems is a very natural choice. To reuse the factorization in FEniCS, I set the PETSc `LUSolver` object and its parameters to reuse the factorization as can be seen at the top of the preconditioner definition.

In order to use the preconditioner, I needed to manipulate a PETSc `ksp()` object. The code in Listing 2.9 sets up and solves the global problem with the ORAS preconditioner for a global stiffness matrix A and load vector b.

```
1 solver = PETScKrylovSolver("gmres", ORAS)
2 solver.set_operator(A)
3
4 u = Function(V)
5 solver.solve(u.vector(), b)
```

Listing 2.9: Solving the global problem with an ORAS preconditioner

Figure 2.21 combines the local auxiliary variable iteration with the global fixed point, the optimized Schwarz preconditioned GMRES iteration, and the RAS preconditioned GMRES iteration for comparison. Since I found significant variation in the optimal α values for each optimized Schwarz algorithm, I compared each algorithm's performance when run with their respective optimum. The global fixed point iteration diverged for a significant number of α values, but when it did converge with the optimal $\alpha = 46$, it did so in nearly as few iterations (52) as the local auxiliary variable iterations (49). The non-converging behaviour is similar to that seen in the finite difference experiments, but the window of acceptable α values is smaller in these finite element experiments. The ORAS preconditioner did, however, work well as a supplement to the GMRES iteration, reducing the iteration count to 12. Interestingly, the RAS preconditioner reduced the error by the same amount in only one extra iteration; whereas the finite difference discretization of the optimized Schwarz preconditioner reduced the iteration count by 4 iterations. In order to investigate this further, I created an experiment to explore the effect of the METIS partitioning strategy on the iteration count. As I mentioned earlier, the METIS partitioning algorithm produces uneven interfaces between subdomains (see Figure (2.19)). I have yet to find a paper that discusses this aspect of the OSM, but the results of my simple experiment demonstrate that the choice of partitioning strategy affects the convergence properties. My experiment consisted of creating a rectangular partitioning of the same mesh used in the earlier experiments and carrying out the same ORAS preconditioned Krylov iteration. I found that the rectangular partitioning strategy reduced the iteration count so that the difference in performance between RAS and ORAS preconditioners is more in line with my expectations based on the finite differ-

ence results where the optimized Schwarz preconditioner improved the convergence over the RAS preconditioner by 40%.

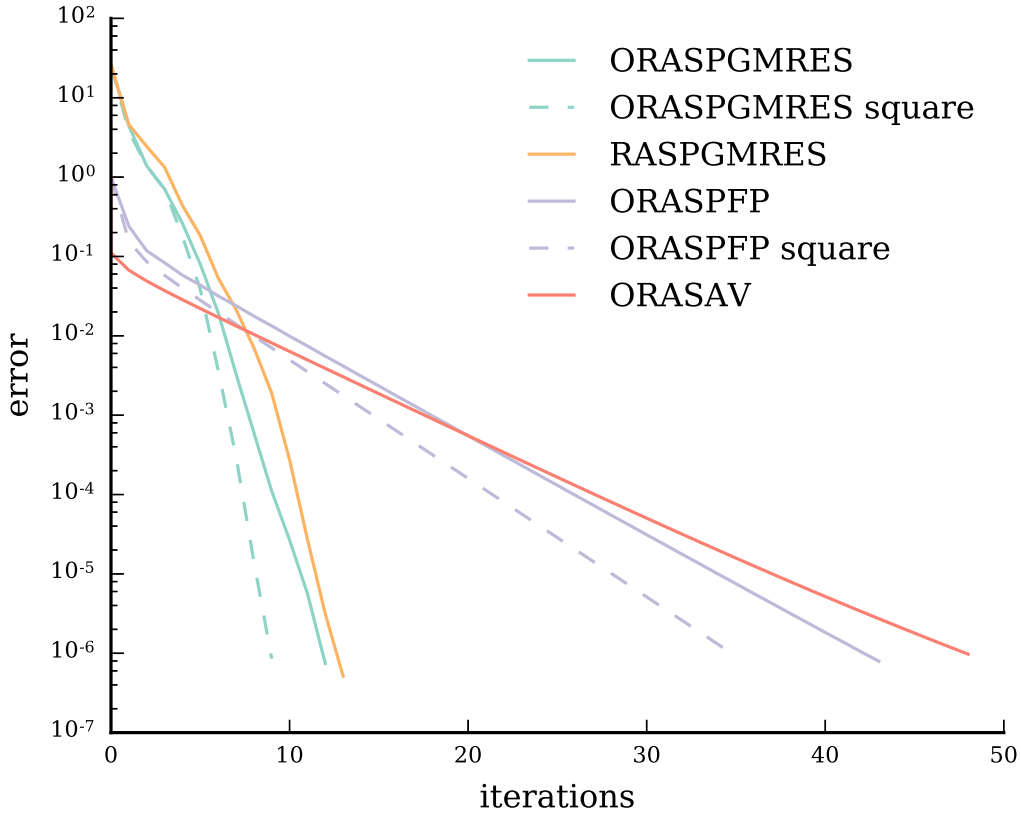

Figure 2.21: Comparison between the local iteration using the auxiliary variable methos (ORASAV), global fixed point iteration (ORASFP), global ORAS preconditioned GMRES iteration (ORASPGMRES), and global RAS preconditioned GMRES iteration (RASPGMRES). Square subdomain results are included to demonstrate the adverse effect of the METIS partitioning strategy on the iteration count (ORASPGMRES square, RASPGMRES square)

Based on the performance of these experiments, and those of the finite difference discretization, it is clear that using the ORAS preconditioner with the Krylov iteration is more profitable in terms of iteration count than any of the other methods. What is not so clear, is whether the Optimized Schwarz approach is worth the extra work when the RAS preconditioner performs so well. The decision is even muddier when a METIS partitioning strategy is used.

This concludes my study of the optimized Schwarz method for the Poisson problem. I have developed the OSM to solve the Poisson problem using both a finite difference and finite element discretization in two dimensions for each of the local auxilliary variable, global fixed point and preconditioned Krylov iterations. In the next section, I will build the global optimized Schwarz preconditioner to use in conjunction with a GMRES iteration for the 3D EM problem. This represents a significant jump in complexity in regards to the book-keeping of local to global degrees of freedom. The reason for this is that I must solve the EM problem in a mixed finite element space to compensate for FEniCS's lack of support for complex variables. In other aspects, such as changing the weak form for the new physics, and jumping from two to three dimensions, the transition to the EM problem is greatly simplified by FEniCS. Based on the results in this chapter, I can expect to find cost/benefit ratio rise for the ORAS preconditioner in comparison to the RAS preconditioner. I must use METIS to partition the meshes for the geophysical EM problem where unstructured meshes are required to accurately and efficiently model the fields under the influence of a point-like source. Furthermore, the partitions created by METIS are likely to be even more uneven than for the structured meshes used in this chapter.

Chapter 3

Electromagnetic Problem

I demonstrated in the last chapter that the OSM could be used to solve the 2D Poisson problem using finite elements or finite difference discretizations. I used the two discretizations to perform the OSM as a local iteration, global fixed point iteration, and as a preconditioner to the GMRES iteration. I also created an RAS preconditioner to serve as a benchmark to measure the success of the ORAS preconditioner. In the finite difference experiments, the ORAS preconditioner outperformed the RAS by a substantial amount, but in the finite element experiments, the margin was much smaller. By altering the partitioning strategy in the finite element implementation, I was able to recover the extra benefit of the ORAS preconditioner in terms of iteration count. However, I will not be able to use such a partitioning strategy for the geophysical EM problem due to the unstructured mesh imposed by refinement in the vicinity of the source. The Poisson problem results in well conditioned system matrices, and I could probably find a number of methods that outperform my OSM preconditioned GMRES iteration. In fact, I chose the GMRES method only because I knew that I

would be requiring it for the EM problem. For symmetric positive definite systems, the conjugate gradient (CG) iteration is a much better choice as it carries a fixed cost per iteration compared to the GMRES iteration whose cost grows with every iteration. In this section, I will attempt to carry out the OSM preconditioning strategy for the indefinite EM equations, *requiring* the use of the GMRES method. Compared to simple preconditioners like Jacobi and SOR, the OSM preconditioner is expensive due to the requirement to fully solve nearly the same physical problem fully on each subdomain. However, in this case, the cost of the preconditioner is more easily justified if it can reduce the number of increasingly expensive GMRES iterations. I will begin this chapter by describing the formulation of the EM equations specific to the geophysical setting where the EM fields perturbations interact with a lossy earth.

3.1 Formulation

The geophysical electromagnetic fields are modelled using Maxwell's equations along with some constitutive equations involving the physical properties of the earth. Maxwell's equations relate the electric ($\vec{E}$), magnetic ($\vec{H}$), electric displacement ($\vec{D}$), magnetic induction ($\vec{B}$), and electric current density ($\vec{J}$) vector fields, through Faradays law

$$\nabla \times \vec{E} = -\frac{\partial \vec{B}}{\partial t}, \tag{3.1}$$

and Ampère's law

$$\nabla \times \vec{H} = \vec{J} + \frac{\partial \vec{D}}{\partial t}. \tag{3.2}$$

The constitutive relations are

$$\vec{B} = \mu_0 \vec{H}, \tag{3.3a}$$

$$\vec{D} = \epsilon_0 \vec{E}, \tag{3.3b}$$

$$\vec{J} = \sigma(\vec{x})\vec{E}, \tag{3.3c}$$

where μ_0 and ϵ_0 are the permeability and permittivity of free space. In the typical geophysical experiment, conductivity varies throughout the earth and would be represented by $\sigma(\vec{x})$ as seen here in Ohm's law. However, for the work in this thesis, I treat conductivity as a constant with respect to the spatial variable $\vec{x}$ since my experiments use a wholespace conductivity model. The constitutive equations are empirical relationships that are considered valid for earth materials. They can be used to eliminate the magnetic induction and electric displacement fields from the equations, leaving

$$\nabla \times \vec{E} = -\mu_0 \frac{\partial \vec{H}}{\partial t} \tag{3.4a}$$

$$\nabla \times \vec{H} = \sigma \vec{E} + \epsilon_0 \frac{\partial \vec{E}}{\partial t} \tag{3.4b}$$

Ampère's law can be modified to allow the modelling of an active source by adding an impressed current density ($\vec{J}_{imp}$). These changes give the usual time domain equations

$$\nabla \times \vec{E} = -\mu_0 \frac{\partial \vec{H}}{\partial t} \tag{3.5a}$$

$$\nabla \times \vec{H} = \sigma \vec{E} + \epsilon_0 \frac{\partial \vec{E}}{\partial t} + \vec{J}_{imp}. \tag{3.5b}$$

These can be expressed in the frequency domain by assuming a time dependence of

$$\frac{\partial}{\partial t} = i\omega. \tag{3.6}$$

Under this transformation, the first order frequency domain equations are

$$\nabla \times \vec{E} = -i\omega\mu_0\vec{H} \tag{3.7a}$$

$$\nabla \times \vec{H} = \sigma\vec{E} + i\omega\epsilon_0\vec{E} + \vec{J}_{imp}, \tag{3.7b}$$

and I get the second order frequency domain equation by taking the curl of Faraday's law and eliminating of the magnetic field by substitution of the $\nabla \times \vec{H}$ term:

$$\nabla \times \nabla \times \vec{E} + i\omega\mu_0\sigma\vec{E} + \omega^2\epsilon_0\vec{E} = -i\omega\mu_0\vec{J}_{imp}. \tag{3.8}$$

Typical earth conductivities and survey frequencies guarantee that $\omega^2\epsilon_0\vec{E} << |i\omega\mu_0\sigma\vec{E}|$ (Grant and West, 1965), so it is common to drop the term involving ϵ_0 from the equations. This is known as the quasistatic approximation. Boundary conditions for Equation (3.8) are typically imposed by considering a large enough domain to approximate infinity at the boundary so that the tangential component of the electric field vanishes there:

$$\hat{n} \times \vec{E} = 0. \tag{3.9}$$

Together, these two equations give the boundary value problem (BVP) for the electric field:

$$\nabla \times \nabla \times \vec{E} + i\omega\mu_0\sigma\vec{E} = -i\omega\mu_0\vec{J}_{imp} \qquad \text{in } \Omega, \tag{3.10a}$$

$$\hat{n} \times \vec{E} = 0 \qquad \text{on } \partial\Omega. \tag{3.10b}$$

In the optimized Schwarz method, I will need to solve subdomain problems with a Robin style boundary condition along subdomain interfaces. The natural combination of Neumann and Dirichlet conditions for the wavy EM problem is the impedance boundary condition.

$$(\nabla \times \vec{E} \times \hat{n}) + i\alpha(\hat{n} \times \vec{E} \times \hat{n}) = g. \tag{3.11}$$

With proper units for α, the impedance boundary condition may be used to approximate the infinite absorbing boundary condition within a finite domain for EM wave equations in perfect resistors (Monk, 2003). There is no guarantee that this boundary condition is the appropriate one for our lossy EM problem, so instead of using the physically meaningful $\alpha = ik$ with wavenumber k and imaginary unit i, I proposed to keep α as a dimensionless optimization parameter. I hoped that this would yield an optimized Schwarz method for the quasistatic EM equations. The associated BVP for the subdomain problem with the impedance BC is

$$\nabla \times \nabla \times \vec{E} + i\omega\mu_0\sigma\vec{E} = -i\omega\mu_0\vec{J}_{imp} \qquad \text{in } \Omega \tag{3.12a}$$

$$\hat{n} \times \vec{E} = 0 \qquad \text{on } \partial\Omega \setminus \Gamma \tag{3.12b}$$

$$(\nabla \times \vec{E} \times \hat{n}) + i\alpha(\hat{n} \times \vec{E} \times \hat{n}) = g \qquad \text{on } \Gamma. \tag{3.12c}$$

Dolean et al. (2015a) proposed a second order impedance style BC to improve the convergence properties over the 'classical' impedance BC. The operator for this method consists of the sum of three terms, the first of which is the original impedance BC and the remaining two containing the second order operators $\nabla_\tau \nabla_\tau \cdot$ and $\nabla_\tau \times \nabla_\tau \times$ for the tangential direction τ. Compared to the first order impedance BC which required the optimization of a single parameter, the second order condition requires a multi-parameter optimization. I mention the second order formulation because it is known to have improved convergence properties, but due to the complexity of the condition, I did not attempt to implement it in this thesis.

3.2 Finite Element method

In the previous section, I derived the second order formulation of Maxwell's equations for the electric field. In order to apply the finite element method, I use the methodology of Monk (2003) and write the weak form of the equations by multiplying by a test function and integrating over the problem domain:

$$\int_\Omega \nabla \times \nabla \times \vec{E} \cdot \vec{N} + \int_\Omega i\omega\mu_0\sigma\vec{E} \cdot \vec{N} = -\int_\Omega i\omega\mu_0\vec{J}_{imp} \cdot \vec{N}. \tag{3.13}$$

Here, I have written the equations with a generic test function $\vec{N}$, but in fact the choice of function space and corresponding basis function can have a profound effect on the resulting performance of the method. Typical Lagrange basis functions were demonstrated in Farquharson and Miensopust (2011) which required the imposition of a gauge condition in order to find accurate solutions. The gauge condition was determined to be required due to the extra condition in EM problems that tangential

electric fields must be conserved across element boundaries. This issue arises in finite difference discretization techniques as well. Hyman and Shashkov (2001) developed a 'mimetic' finite difference discretization in which the discrete operators preserve certain properties of their continuous conterparts such as tangential electric field conservation. A common way to impose the tangential electric field continuity in the finite element method is to seek a solution $\vec{E}$ in the space $H(\text{curl}, \Omega) = \{\vec{E} \in (L^2(\Omega))^3 : \nabla \times \vec{E} \in (L^2(\Omega))^3\}$ (Jin, 2002). The first order shape function from this space is a curl conforming vector function of the form $\vec{N_i} = \lambda_n \nabla \lambda_m - \lambda_m \nabla \lambda_n$ for the nodal shape functions $\lambda_k = a_k + b_k x + c_k y$ on an edge defined by nodes m and n.

With a sufficiently large mesh, I can approximate the condition that the tangential electric field disappears at infinity by imposing a Dirichlet condition at the boundary of a mesh. This translates to seeking a solution in the new space $H_0(\text{curl}, \Omega) = \{\vec{E} \in H(\text{curl}, \Omega) : n \times \vec{E}|_\Gamma = 0\}$. I use integration by parts to reduce the order of the integrand $\nabla \times \nabla \times \vec{E}$ so that I can approximate the integral using the first order shape functions I described earlier. In doing so, I introduce a surface integral over the boundary $\partial\Omega$:

$$\begin{aligned} \int_\Omega (\nabla \times \vec{E}) \cdot (\nabla \times \vec{N}) - \int_{\partial\Omega} (\nabla \times \vec{E} \times \hat{n}) \cdot \vec{N} + i\omega\mu_0 \int_\Omega \sigma \vec{E} \cdot \vec{N} \\ = \quad -i\omega\mu_0 \int_\Omega \vec{J}_{imp} \cdot \vec{N}. \end{aligned} \tag{3.14}$$

Now that there is a boundary integral, it is possible to impose Dirichlet or Robin style boundary conditions according to the problem at hand. For the OSM, I will need to apply both Dirichlet and Robin boundary conditions over different portions of the boundary. This can easily be accommodated by splitting the boundary integral

into portions marked for Dirichlet and Robin boundary condition application:

$$\int_{\partial\Omega}(\nabla\times\vec{E}\times\hat{n})\cdot\vec{N}=\int_{\partial\Omega/\ \Gamma}(\nabla\times\vec{E}\times\hat{n})\cdot\vec{N}+\int_{\Gamma}(\nabla\times\vec{E}\times\hat{n})\cdot\vec{N} \tag{3.15}$$

For the homogeneous Dirichlet boundary condition on $\partial\Omega$, the boundary integral simply vanishes. The Robin boundary condition is applied by multiplying Equation (3.11) by a test function and integrating along the interface Γ

$$\int_{\Gamma}(\nabla\times\vec{E}\times\hat{n})\cdot\vec{N}+\int_{\Gamma}i\alpha(\hat{n}\times\vec{E}\times\hat{n})\cdot\vec{N}=\int_{\Gamma}g\cdot\vec{N}. \tag{3.16}$$

I then rearrange and substitute the condition into the Robin boundary integral of Equation (3.15), which itself is inserted into Equation (3.14) to give a weak form for the most general problem in which a homogeneous Dirichlet condition has been imposed on a portion of the domain, and a Robin condition on another

$$\begin{aligned}\int_{\Omega}(\nabla\times\vec{E})\cdot(\nabla\times\vec{N})+\int_{\Gamma}i\alpha(\hat{n}\times\vec{E}\times\hat{n})\cdot\vec{N}+i\omega\mu_0\int_{\Omega}\sigma\vec{E}\cdot\vec{N}\\ =\quad -i\omega\mu_0\int_{\Omega}\vec{J}_{imp}\cdot\vec{N}-\int_{\Gamma}g\cdot\vec{N}.\end{aligned} \tag{3.17}$$

Now I have the most general equation required to complete an implementation of an OSM preconditioner for the quasistatic EM equations. I can use this formulation directly for the subdomain problems that arise from the domain decomposition, and I can simply ignore the interface integrals in Equation (3.17) to get the relevant equation for the global problem. In either case, I can state the problem as: Find $\vec{E}\in H_0(curl,\Omega)$ such that $a(\vec{E},\vec{N})=L(\vec{N})$ where I have collected the integrals of the left hand side into the sesquilinear form $a(\vec{E},\vec{N})$ (the sesquilinear form is a

generalization of the more common bilinear form for simple finite element spaces) and integrals of the right side into the linear form $L(\vec{N})$. Note that if the goal is to precondition a Krylov iteration by accumulating subdomain solutions, the last term of Equation (3.17) need not be discretized since the preconditioner routine only needs a discretization of the bilinear form for the subdomain problem.

The next step is to apply Galerkin's method and replace $\vec{E}$ by the linear combination $\sum_j^n \vec{E}_j \vec{N}_j$ and the original test function $\vec{N}$ by $\vec{N}_i$. Unfortunately, FEniCS was not designed to handle complex arithmetic, so a little extra work is required to accommodate the fact that the system of equations is complex. A well known strategy for this situation is to solve, at once, for the real and imaginary components of the solution. This can be accomplished by solving the block system

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}, \tag{3.18}$$

where x_1 are a set of degrees of freedom representing the real components of the solution, and x_2 are another set of degrees of freedom for the imaginary components. In order to do this, I extended Galerkin's method for the real and complex components $E_{R,j}$ and $E_{C,j}$; ie: I let $\vec{E} = \sum_j^n (E_{R,j} + iE_{C,j})\vec{N}_j$, and inserting this into 3.17 to give

$$\begin{aligned} \sum_{j=1}^{n} E_{R,j} \left(\int_\Omega (\nabla \times \vec{N}_i) \cdot (\nabla \times \vec{N}_j) + i\omega\mu_0 \int_\Omega \sigma \vec{N}_i \cdot \vec{N}_j - i\alpha \int_\Gamma (\hat{n} \times \vec{N}_i \times \hat{n}) \cdot \vec{N}_j \right) \\ + \quad i \sum_{j=1}^{n} E_{C,j} \left(\int_\Omega (\nabla \times \vec{N}_i) \cdot (\nabla \times \vec{N}_j) + i\omega\mu_0 \int_\Omega \sigma \vec{N}_i \cdot \vec{N}_j - i\alpha \int_\Gamma (\hat{n} \times \vec{N}_i \times \hat{n}) \cdot \vec{N}_j \right) \\ = \quad -i\omega\mu_0 \int_\Omega \vec{J}_{imp} \cdot \vec{N}. \end{aligned} \tag{3.19}$$

I will now introduce a few symbolic abstractions for clarity. Let integrals $\int_\Omega(\nabla \times \vec{N}_i)\cdot(\nabla\times\vec{N}_j)$, $\omega\mu_0\int_\Omega \sigma\vec{N}_i\cdot\vec{N}_j$, $\alpha\int_\Gamma(\hat{n}\times\vec{N}_i\times\hat{n})\cdot\vec{N}_j$, and $\omega\mu_0\int_\Omega \vec{J}_{imp}\cdot\vec{N}$ be represented by C, M_σ, M_α, and S. In terms of the newly introduced symbols, I get the following by distributing the imaginary component into the terms of the sum,

$$\sum_{j=1}^{n} E_{R,j}\left(C + iM_\sigma + iM_\alpha\right) + \sum_{j=1}^{n} E_{C,j}\left(iC - M_\sigma - M_\alpha\right) \tag{3.20}$$

and I can collect the real and imaginary components into the 2x2 block system

$$\begin{pmatrix} C & -(M_\sigma + M_\alpha) \\ M_\sigma + M_\alpha & C \end{pmatrix} \begin{pmatrix} E_{R,j} \\ E_{C,j} \end{pmatrix} = \begin{pmatrix} 0 \\ S \end{pmatrix}. \tag{3.21}$$

In the next section, I will demonstrate how I used FEniCS and PETSc to assemble and solve a mixed function space formulation of the complex system of equations from the bilinear and linear forms of a subdomain problem. I will then show how I set up and used my custom preconditioner to accelerate the global Krylov iteration for the EM problem.

3.3 FEniCS implementation

3.3.1 External mesh generation

Mesh refinement is an important component to geophysical EM modelling, as it allows computational effort to be directed to areas where the fields are changing the most rapidly, and where observations are made. In order to accomplish mesh refinement for my EM experiment, I used the GMESH finite element mesh generator (Geuzaine and Remacle, 2009). This tool can be run through a graphical user interface (GUI)

program, or through a command line interface in conjunction with geometry files. I took the latter approach, and I will refer throughout this section to the geometry file provided in Appendix D. To accomplish mesh refinement in GMESH, I used the concept of an attractor field. The attractor field concept takes as input the refinement location, radius, fine and coarse mesh size parameters. GMESH then incorporates the attractor field into the overall meshing routine, and creates a mesh that transitions smoothly from the fine mesh size to the coarse mesh size within the specified radius, and centered on the refinement location. In Appendix D, I create points for the source and receiver attractor fields in lines 18-33. The points are then defined as attractors in lines 51-64.

GMESH can also be used to build earth models, although I have only used it to create the simplest of earth model: the wholespace. In GMESH, physical properties are assigned to the various geometrical entities (Points, Lines, Surfaces, and Volumes). In Appendix D, I create the physical volume representing the wholespace in line 49. In order to create more complicated earth models, a user could layer more physical volumes in this way, or through the GUI.

Poor mesh quality is known to affect the conditioning of the EM equations. In particular, tetrahedra with large radius-edge ratios have been found to limit the accuracy of solutions (Jahandari and Farquharson, 2015). GMESH has a feature that optimizes the mesh quality. I have not carefully studied its effect, but I used it for all the meshes; the optimization step can be seen at the end of the geometry file provided in D.

In order to assemble the EM equations in FEniCS for a GMESH mesh, I converted the .msh files created by GMESH using the dolfin-convert scripts provided with every

FEniCS installation. This generates a collection of xml files that describe the mesh and physical properties of the defined model. In order to run FEniCS in parallel, I generated an HDF5 file containing the mesh, physical properties, and parameters for each trial in serial first. This was carried out as a pre-processing step using a script HDF5mesh.py (Appendix E). The HDF5mesh.py code also sets up the active EM source assembly by locating the cell containing the source location, a task that is necessarily done in serial. Once the HDF5mesh.py code has been run in serial, the physical properties, source cell, and mesh can all be loaded, in parallel thanks to the HDF5 technology, into the main EM codes which then assemble the EM equations over the particular mesh.

3.3.2 Finite element assembly for the EM problem

Many of the steps in the preceding section can be translated easily into the syntax provided by the finite element assembly library FEniCS. These include the selection of a function space and associated basis function, definition of the bilinear and linear forms, stiffness matrix and load vector assembly, and application of Dirichlet boundary conditions. In this section I will demonstrate the FEniCS syntax required to accomplish these tasks for a subdomain problem. The following assumes that I already have a `mesh` saved to file in HDF5 format along with a mesh function (`boundaries`) that contains markers for the Dirichlet boundary conditions.

To begin with, I imported the library, mesh, and mesh function that contains the boundary markers. I then created an $H(\text{curl}, \Omega)$ function space each for the real and imaginary components, and combined them into a mixed function space. This

establishes a mapping of the degrees of freedom (DOF) on the mesh so that the N edges of the mesh have two DOFs, one for the real part and one for the imaginary (Listing 3.1).

```
from dolfin import *
mesh = Mesh()
hdf = HDF5File(mesh.mpi_comm(), "test.h5", "r")
hdf.read(mesh, "/mesh", False) # read in the mesh
hdf.read(boundaries, "/boundaries") # read in the boundary function
hdf.close()

Vr = FunctionSpace(mesh, "Nedelec 1st kind H(curl)", 1)
Vc = FunctionSpace(mesh, "Nedelec 1st kind H(curl)", 1)
V = Vr * Vc
```

Listing 3.1: Function Spaces

I used the `boundaries` mesh function to define the homogeneous Dirichlet boundary condition. Since I have a vector field in three dimensions with real and imaginary components, the `Expression()` that I used to provide the zero Dirichlet condition had 6 entries of the zero value as seen in Listing 3.2.

```
bc = DirichletBC(V, Expression(('0.0', '0.0', '0.0', '0.0', '0.0', '
  0.0')), boundaries, dirichlet)
```

Listing 3.2: Dirichlet Conditions

To set up the Robin boundary condition I needed to use the same `boundaries` mesh function to construct a `Measure()` that allowed me to assign portions of the boundary to integrate over (Listing 3.3).

```
ds = Measure('ds', domain=mesh, subdomain_data=boundaries)
```

Listing 3.3: Measure

In order to construct the sesquilinear and linear forms for the problem so that FEniCS could take care of the assembly, I first needed to create `TrialFunction()` and `TestFunction()` objects (Listing 3.4). The mixed function space meant that two distinct objects were created for each of the trial and test functions. The trial and test function served as a means to set the various blocks of the matrix Equation (3.21) in the sesquilinear form:

```
Nri, Nci = TrialFunctions(V)
Nrj, Ncj = TestFunctions(V)
```

Listing 3.4: Trial and test functions

The syntax used for creating the sesquilinear form from the test and trial functions was remarkably close to the mathematical notation of the form (Equation (3.20)). Listing 3.5 demonstrates the form as well as the two lines needed to then assemble the form and apply the Dirichlet BC.

```
a_robin = + inner(curl(Nri_local), curl(Nrj_local))*dx_local \
          + inner(curl(Nci_local), curl(Ncj_local))*dx_local \
          - alpha_r * inner(cross(cross(n,Nri_local),n), Nrj_local)*ds
  (1) \
          + alpha_c * inner(cross(cross(n,Nci_local),n), Nrj_local)*ds
  (1) \
          - alpha_c * inner(cross(cross(n,Nri_local),n), Ncj_local)*ds
  (1) \
          - alpha_r * inner(cross(cross(n,Nci_local),n), Ncj_local)*ds
  (1) \
```

```
            + omega*mu * inner(sigma*Nri_local, Ncj_local)*dx_local \
            - omega*mu * inner(sigma*Nci_local, Nrj_local)*dx_local

Ar = assemble(ar)
bc.apply(Ar)
```

Listing 3.5: Bilinear form

I used the various test and trial functions to assemble the different integrals within the block system (Equation (3.21)). The combination `Nri` and `Nrj` assembles the upper left block, `Nci` and `Ncj` the lower right block, `Nri` and `Ncj` the lower left, and `Nci` and `Nrj` the upper right.

This concludes the description of the method I used to assemble a subdomain problem, but it is not in fact possible to perform this operation on a subset of the global mesh in the latest build of the FEniCS library. To solve a PDE in parallel, I needed to create individual meshes on each processor, so that I could solve the subdomain problem. In the next section I will demonstrate how to create the subdomain meshes from the global distributed mesh, solve the subdomain problem on the individual meshes, and map the solutions back into a DOF ordering that matches that of the global mesh.

3.3.3 Optimized Schwarz preconditioner

In the last section, I covered how to build the new sesquilinear and linear forms for the geophysical EM problem. In Section 2.4.3, I presented a method for setting up an optimized Schwarz preconditioner for Poisson, from which I can borrow several components for the EM problem. The biggest complication in transitioning to the EM

problem is the extra book-keeping involved in the local to global degrees of freedom. In the EM problem, I have a mixed function space for the real and imaginary parts of the equation, and my degrees of freedom reside on the edges.

The first step in the process of forming a Schwarz preconditioner is, once again, to copy over local parts of the mesh into separate local meshes using `MeshEditor()` as explained in the last chapter for the Poisson problem. These details do not change for the EM problem and will not be included here. However, the next step is to create a mapping of the local and global degrees of freedom, and this will certainly be different for the EM problem since the degrees of freedom now reside on the edges of the mesh. The following code is very similar to Listing 2.5, but creates a dictionary of edge data rather than vertex data to correctly map the new edge based DOF (Listing 3.6).

```
1  partition_of_unity = [1] * mesh.num_edges()*2
2  dof_g2l = [0] * mesh.num_edges()*2
3  for ci in range(mesh_local.num_cells()):
4
5      cg = Cell(mesh, ci)
6      cl = Cell(mesh_local, ci)
7
8      cell_dofs = [dofmap.local_to_global_index(i) for i in dofmap.
       cell_dofs(ci)]
9      cell_dofs_local = dofmap_local.cell_dofs(ci)
10
11
12     edge_data = {}
13     for e in edges(cg):
14         vs = []
```

```
15          for v in vertices(e):
16              vs.append(v.index())
17          vk = tuple(sorted(vs))
18
19          edge_data[vk] = [(cell_dofs[cg.index(e)], cell_dofs[cg.index(e)
    +6]),
20                           1./(len(e.sharing_processes()) + 1)]
21
22      for e in edges(cl):
23          vs = []
24          for v in vertices(e):
25              vs.append(v.index())
26          vk = tuple(sorted(vs))
27
28          dof_g2l[cell_dofs_local[cl.index(e)]] = edge_data[vk][0][0]
29          dof_g2l[cell_dofs_local[cl.index(e)+6]] = edge_data[vk][0][1]
30
31          partition_of_unity[cell_dofs_local[cl.index(e)]] = edge_data[vk
    ][1]
32          partition_of_unity[cell_dofs_local[cl.index(e)+6]] = edge_data[
    vk][1]
33
34  dofg2l_IS = PETSc.IS().createGeneral(dof_g2l, comm=PETSc.COMM_SELF)
```

Listing 3.6: Local to global edge map

Since I needed to use the edge mapping on vectors that contain a real and imaginary DOF entry for each edge, I built this into the mapping by assigning an i and $i+1$ map for each edge that I encountered. This is a particularly important point: In

mixed function spaces, FEniCS stores Vectors and Matrices so that DOFs, for a given edge, from each function space, are stored consecutively. The global to local DOF map is based on this structure and is the fundamental link between the subdomain problems and the global problem, as it is used to gather DOFs from the incoming vector, and to set DOFs into the outgoing vector. The last line of the code converts the map into a PETSc index set so that I can perform the gathering and setting with the PETSc backend. To avoid having to loop through the mesh twice, I built the partition of unity operator at the same time as the DOF map using the edge data dictionary. The partition of unity is constructed one DOF at a time in line 19. I used `sharing_processes()` to get the partition of unity weight for the i^{th} edge. Just as for the Poisson problem, this creates a partition of unity that produces the average of the overlapping DOFs.

The preconditioner definition in Listing 2.8 remains the same for the EM problem although it will now be instantiated with a DOF map, subdomain solver and associated factorized Robin subdomain matrices, and partition of unity that are constructed for the mixed function space representation of the complex number system. When an outer Krylov solver calls the preconditioner with a mixed function space vector, the preconditioner solves a subdomain problem with a mixed function space matrix assembled from the form in Listing 3.5 and the local part of the incoming vector. The preconditioner then applies the partition of unity to the resulting real and imaginary components of the solution. Finally, it sets the result into the outgoing vector's real and imaginary DOF entries according to the map created in Listing 3.6.

Before I present the performance of the Schwarz preconditioners for the EM problem, I will demonstrate the accuracy of the solutions found by both a direct and

preconditioned iterative approach. In order to verify the numerical solution, I will use the analytic solutions provided in Ward and Hohmann (1988) which gives the following equations for the non-zero field components given an x directed magnetic dipole in a uniform wholespace sampled at $z = 0$ along the y dimension. Here, $r = \sqrt{x^2 + y^2 + z^2}$, $k = -\sqrt{i\omega\mu\sigma}$, I is the electric current, and A is the area of the dipole:

$$H_x = \frac{IA}{4\pi r^3}\left[\frac{x^2}{r^2}\left(-k^2r^2 + 3ikr + 3\right) + \left(k^2r^2 - ikr - 1\right)\right]e^{-ikr} \tag{3.22}$$

and

$$E_z = \frac{i\omega\mu IA}{4\pi r^2}\left(ikr + 1\right)e^{-ikr}\left(\frac{-y}{r}\right). \tag{3.23}$$

In order to get the correct components to compare with the analytic solutions, I needed to split my E field solution into its components and solve a small variational problem to get the H field. I split the electric field into components by projecting the solution into a vector function space with the code in Listing 3.7.

```
Vv = VectorFunctionSpace(mesh, 'DG', 1)
Er, Ec = E.split()
Er_vec = project(Er, Vv, solver_type='cg')
Ec_vec = project(Ec, Vv, solver_type='cg')
Erx = Er_vec.sub(0)
Ecx = Ec_vec.sub(0)
Ery = Er_vec.sub(1)
Ecy = Ec_vec.sub(1)
Erz = Er_vec.sub(2)
```

```
Ecz = Ec_vec.sub(2)
```

Listing 3.7: Projection into the discontinuous Galerkin vector function space

The H and E fields are related through Faraday's law by

$$i\omega\mu\vec{H} = -\nabla \times \vec{E}, \tag{3.24}$$

so I can form a variational problem by multiplying by a test function and integrating

$$i\omega\mu \int_\Omega \vec{H} \cdot \vec{v} = \int_\Omega \nabla \times \vec{E} \cdot \vec{v}. \tag{3.25}$$

I followed the same procedure to form the block system for the real and imaginary parts as was done for the electric field equation, and translated this into the FEniCS code in Listing 3.8.

```
Er, Ec = E.split()

wr, wc = TrialFunctions(V)
vr, vc = TestFunctions(V)

a = omega*mu*inner(wr, vc)*dx - omega*mu*inner(wc,vr)*dx

L = - inner(Er, curl(vr))*dx \
    - inner(Ec, curl(vc))*dx

H = Function(V)
solve(a == L, H)
```

Listing 3.8: Solving a variational problem for the H field given the E field solution

Finally, by discretizing Equations 3.22 and 3.23, and comparing them to the solutions of both a parallel direct (MUMPS), and preconditioned GMRES solver, I found that my finite element solutions were a reasonable match to the analytical solutions. I used a 437610 tetrahedra mesh with a high degree of refinement about the source and receiver locations to solve the system accurately. In Figures 3.1, 3.2, and 3.3, I give a series of successively zoomed in images of a two dimensional mesh that was constructed with the same refinement strategy as the three dimensional meshes used for the experiment (I used this work-around because I was not able to extract a reasonable image of a slice through the partitioned three dimensional mesh). The data for each of the analytical, MUMPS, and RAS preconditioned GMRES solvers are provided in Figures 3.4, 3.5, 3.6, and 3.7. The analytical solution in blue is very accurately recovered by the MUMPS solution in red, and the preconditioned GMRES iteration when it is run to a tolerance of 10^{-19} (green). The two-norm of the error between the MUMPS solution and the RAS preconditioned solution was 10^{-14}, suggesting that the true error of the iterative solution is not exactly reflected in the norm of the residual provided by the PETSc GMRES function. Nevertheless, the solution seems acceptable, at least by visual standards.

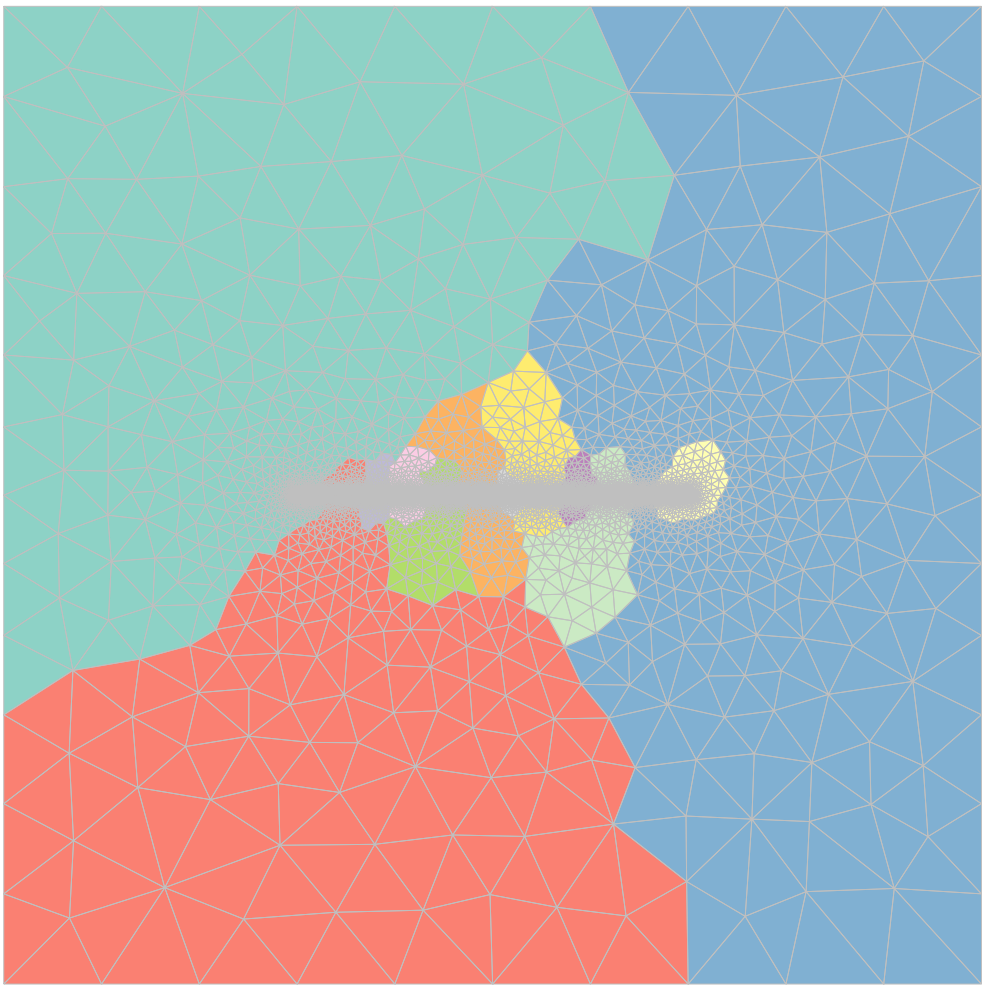

Figure 3.1: Two dimensional mesh with same refinement and partitioning strategy as the three dimensional mesh used to compare the RAS and MUMPS solutions.

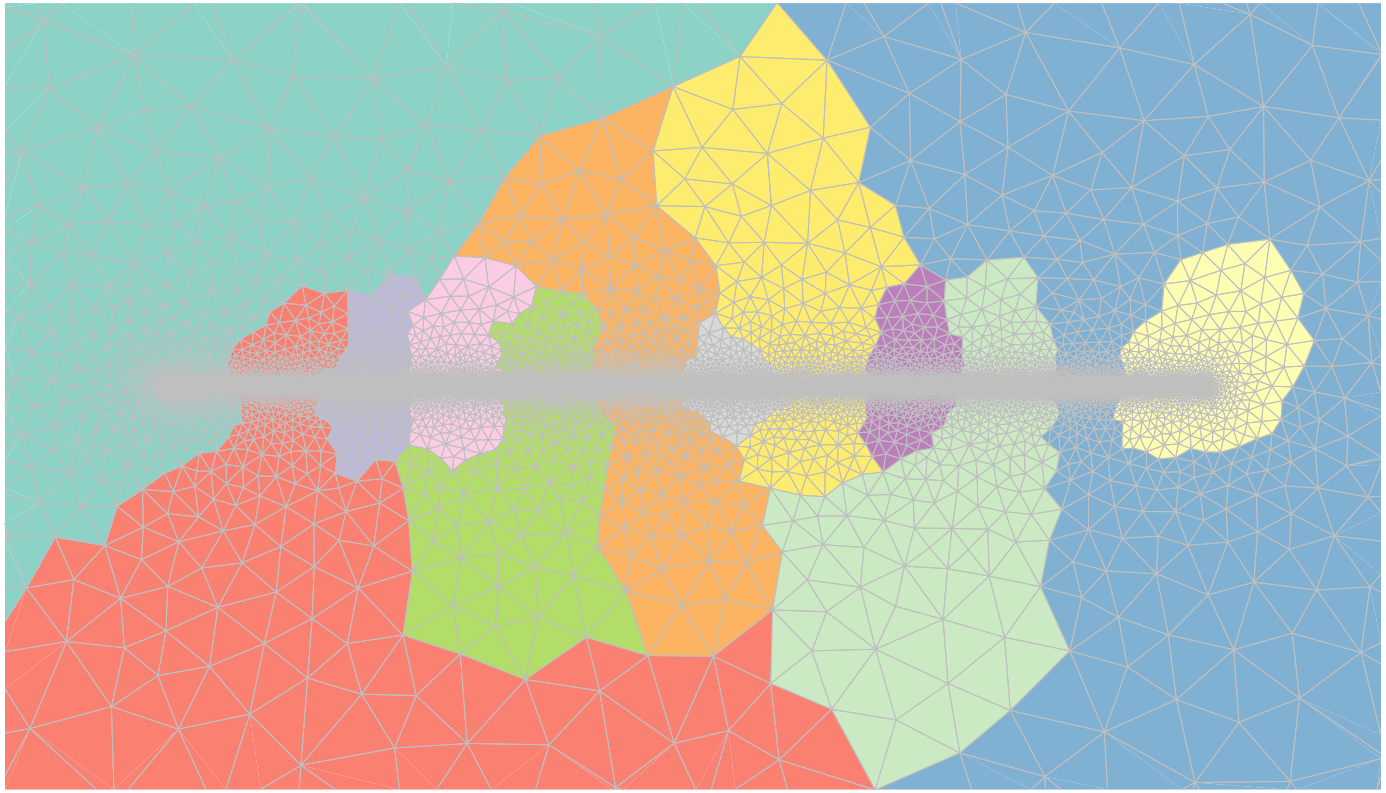

Figure 3.2: Same mesh as Figure 3.1 but zoomed in on refinement area.

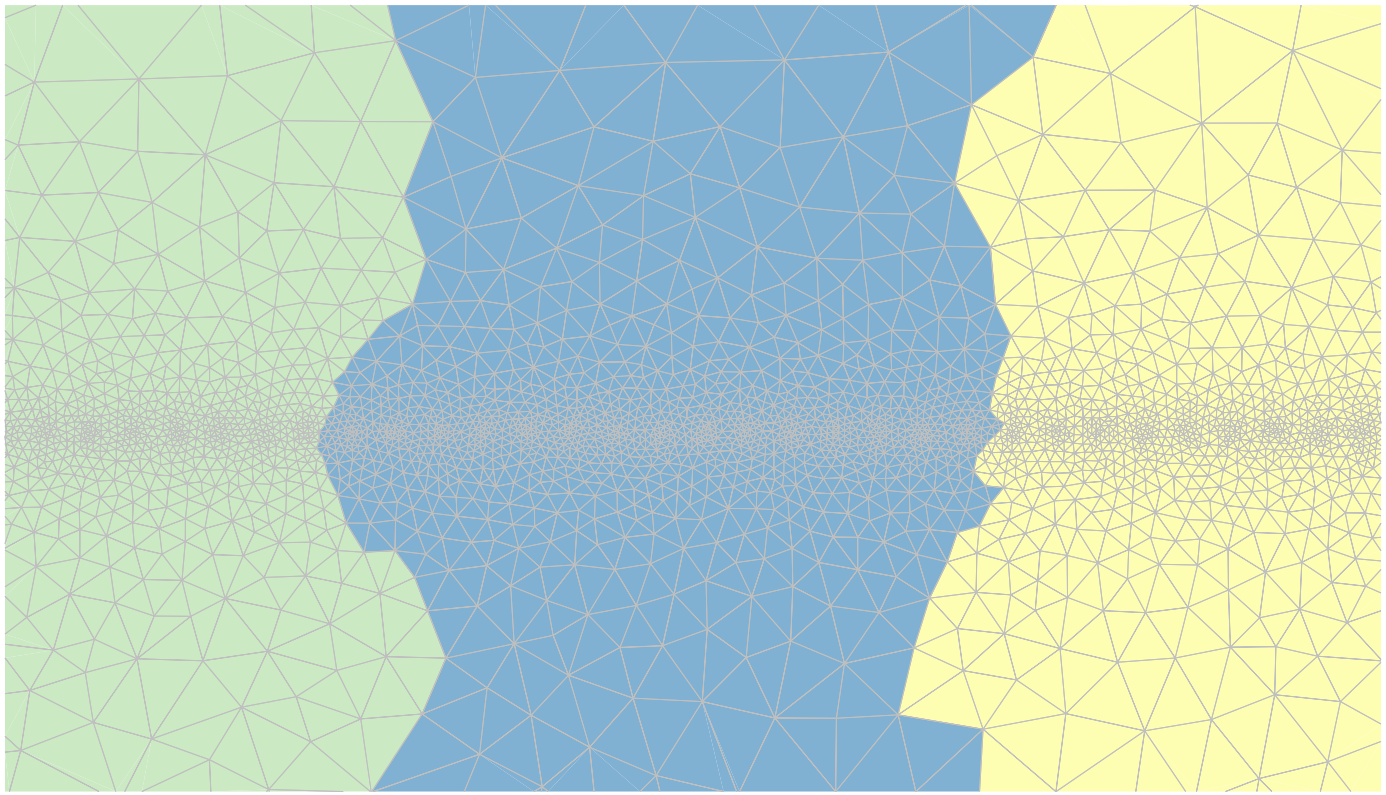

Figure 3.3: same mesh as Figure 3.1 but zoomed in to a scale that shows the individual cells in refinement area.

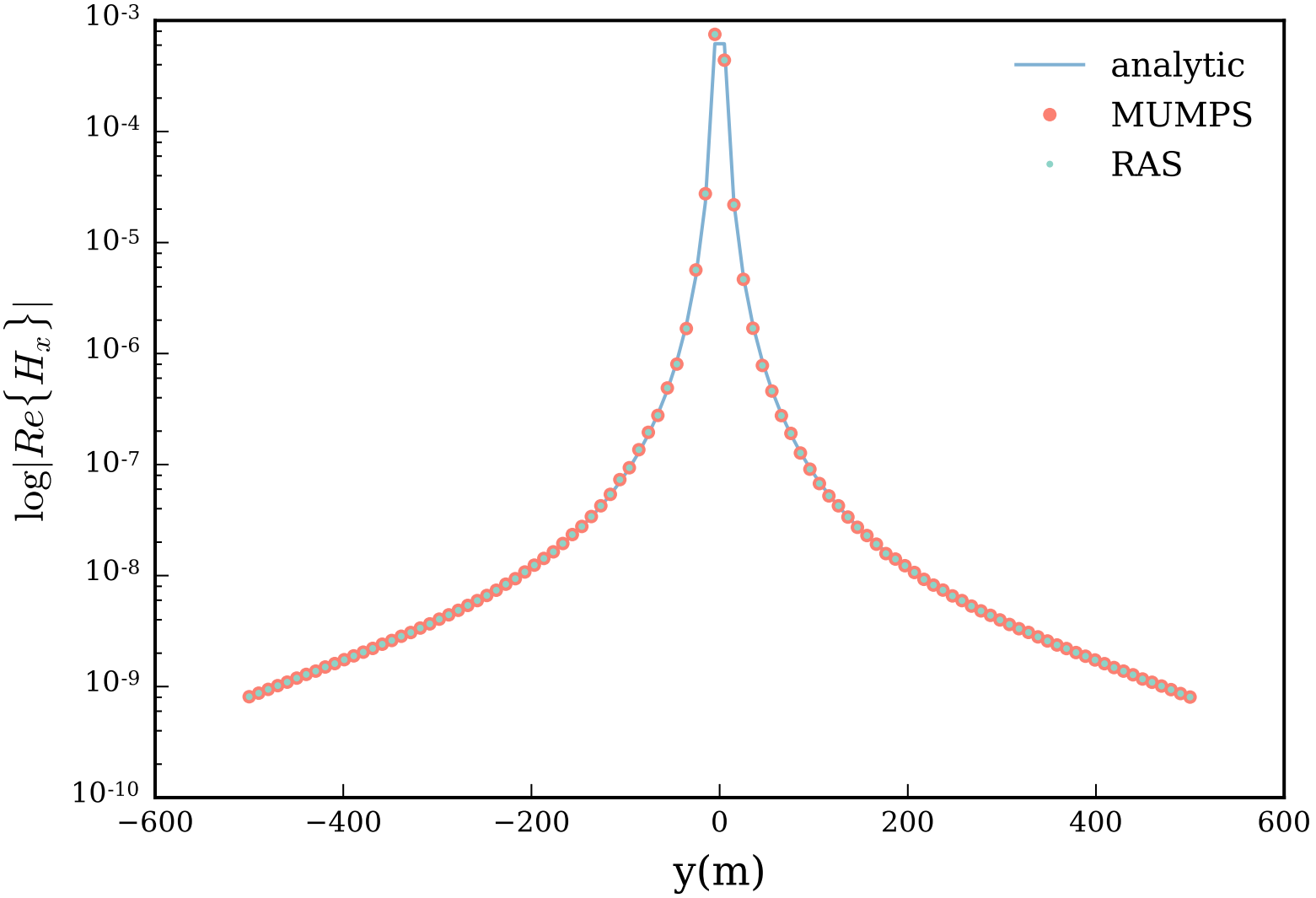

Figure 3.4: Predicted $Re\{H_x\}$ data along y dimension for an x directed dipole.

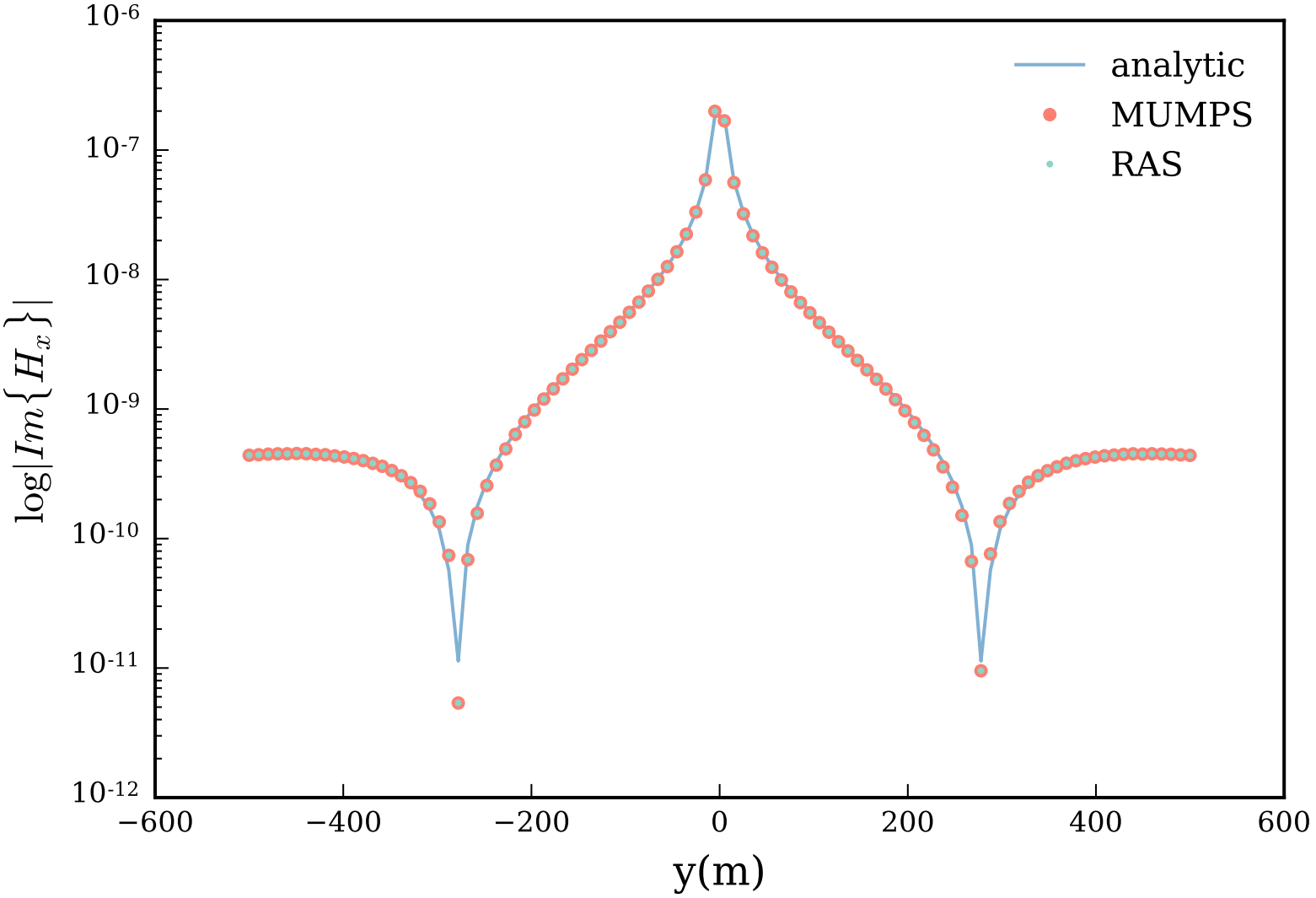

Figure 3.5: Predicted $Im\{H_x\}$ data along y dimension for an x directed dipole.

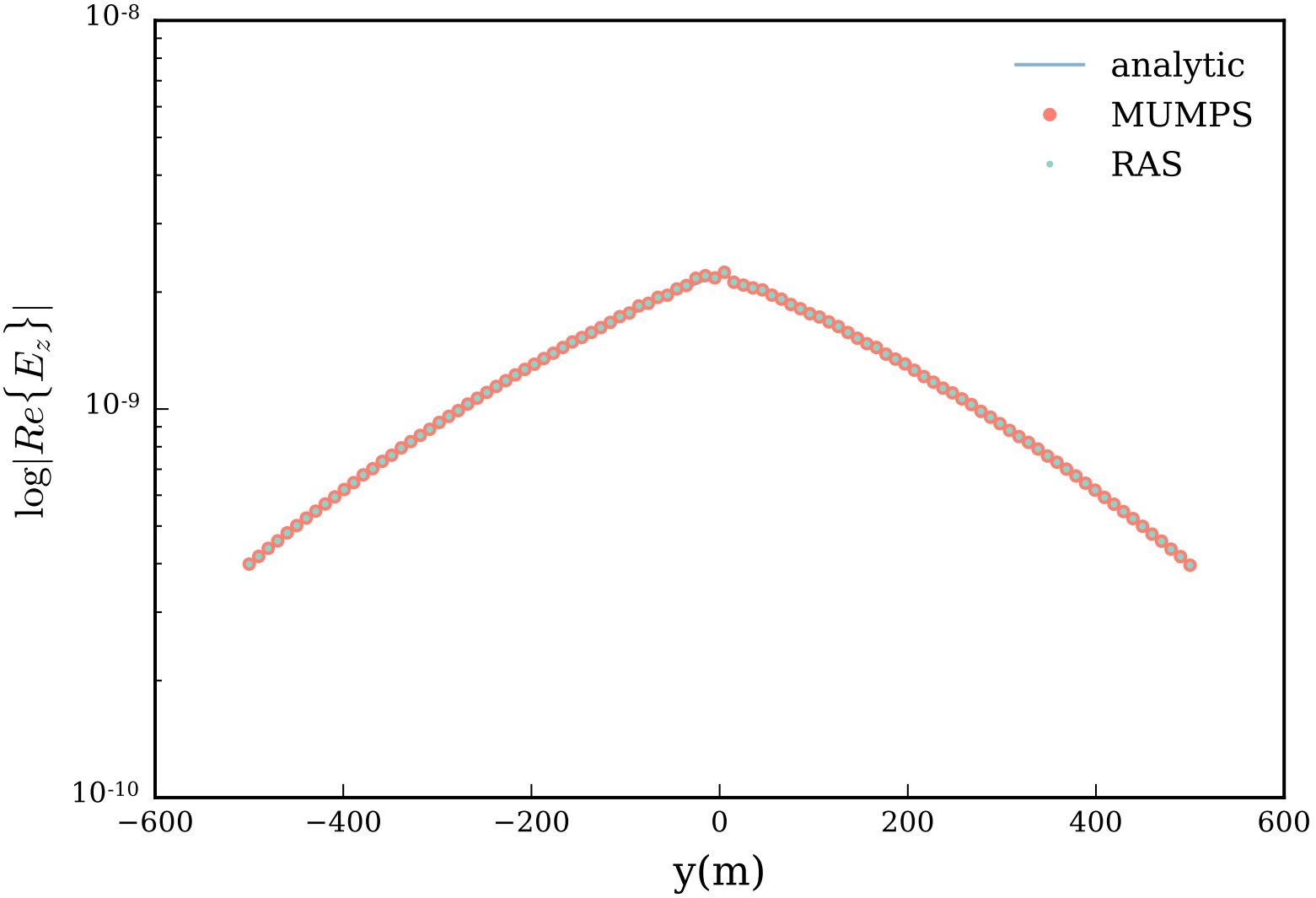

Figure 3.6: Predicted $Re\{E_z\}$ data along y dimension for an x directed dipole.

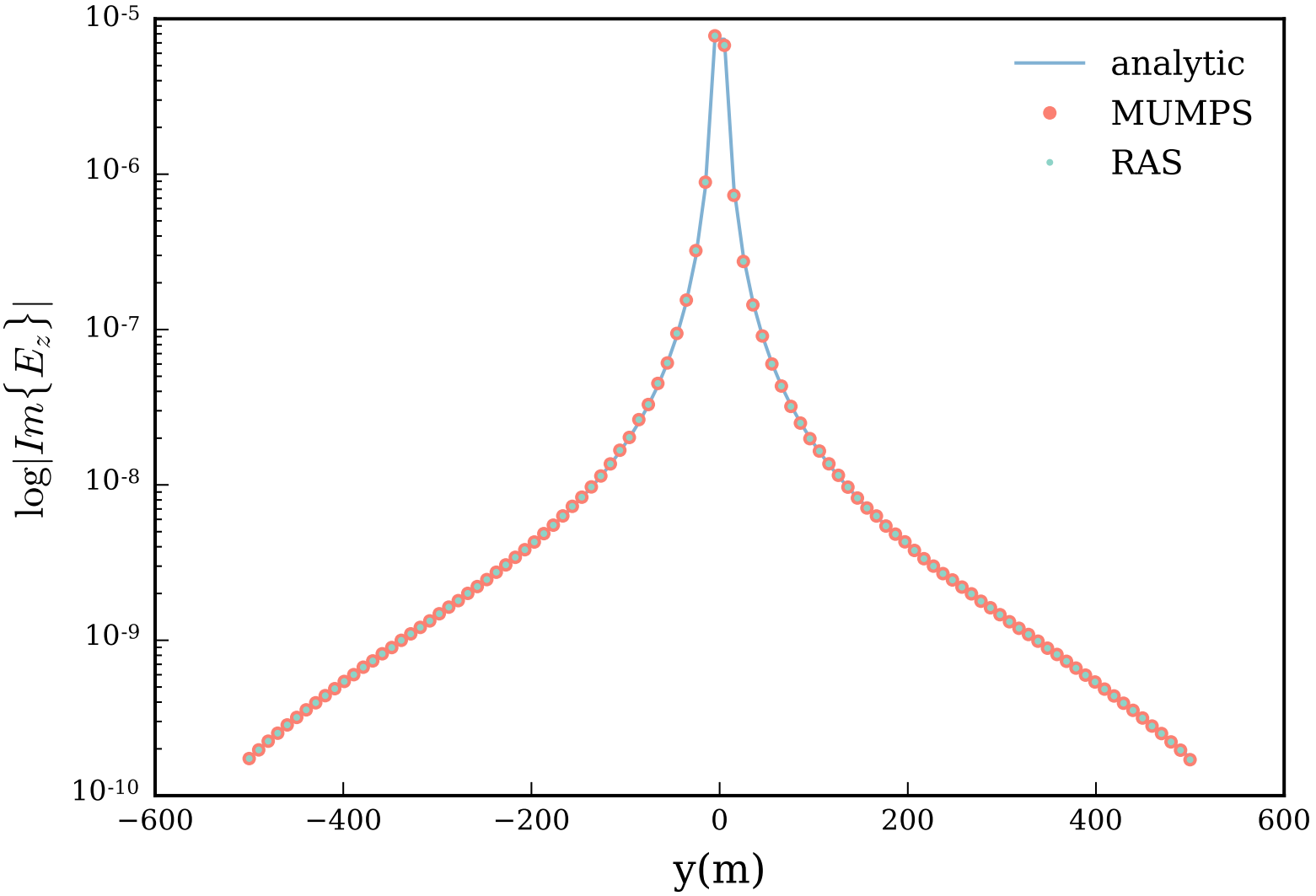

Figure 3.7: Predicted $Im\{E_z\}$ data along y dimension for an x directed dipole.

Now that I have confirmed the accuracy of my solution, I can address the performance of the two Schwarz preconditioners. FEniCS has a handful of built-in parallel preconditioners that I can use to compare against my own preconditioners. This includes an implementation of the RAS algorithm discussed in the last chapter. Figure 3.8 provides a performance comparison of my optimized Schwarz preconditioner against my own RAS implementation, a successive over-relaxation (SOR), multigrid (AMG), and PETSc's RAS preconditioner for a larger overlap than I used for my RAS. The reason for the different overlaps is that PETSc defines its overlap starting with a single layer of cells, whereas the `shared_vertex` option that I used provided two layers of cells on either side of the partition. I have also provided the unpreconditioned iteration for comparison. For the Optimized Schwarz preconditioner, I performed a trial and error optimization of the parameter α that yielded a range of values under which the algorithm performed 'optimally'. The value settled on for the final run was 10^9. The only preconditioners that were able to reduce the error beyond 10^{-8} were the domain decomposition based preconditioners, although I suspect that a multigrid variant could be constructed that would compete with the DD approach. However, this would likely require the same level of detailed study that I have put into my DD preconditioners. To get the most out of the PETSc RAS preconditioner, I found that I needed to set the subdomain solvers to solve the subdomain problems exactly with an LU decomposition, and set the ASMType to RESTRICT to implement the partition of unity particular to the RAS algorithm. Under these conditions, the PETSc RAS performs better in terms of iteration count than my RAS implementation when a larger overlap was deployed, and worse for a smaller overlap. Finally, the most noteworthy result here is that the RAS preconditioners beat my ORAS precon-

ditioner by a small margin. From the analysis done for the Poisson problem, I suspect this to be caused by the uneven partitions created by the METIS graph partitioning algorithm. If the irregular partition boundaries produce an inconsistent splitting analogous to the way the non-overlapping ORAS preconditioned fixed point iteration did for the Poisson problem, then it is possible that the RAS method that restricts the subdomain matrices out of the global matrix produces a consistent splitting even in the presence of an uneven interface. However, unlike the fixed point iteration, I would not expect the Krylov iteration to diverge. Rather, the ORAS preconditioned Krylov iteration might simply suffer a small performance penalty compared to the RAS resulting in the observed behaviour. The METIS approach is required for the geophysical EM problem due to the unstructured meshes imposed by modelling the discrete dipole source. So it appears that for the geophysical EM problem, the RAS preconditioner may actually be preferred over the ORAS preconditioner. However, iteration count does not provide the full story. Time to solution is another arguably more important metric for the success of an iterative method.

In Figure 3.9 I provide the timing for my ORAS and RAS preconditioner implementations, as well as the PETSc RAS preconditioner and the FEniCS code for assembly and preconditioner setup. The time to assemble the global matrix and right hand side decreased subtly between two and twelve processors, and took a negligible amount of time overall. The setup code that was necessary for both the ORAS and RAS preconditioner methods took up a little over half of the time to solution and scaled close to linearly between two and eight processors. The total time to solution using both ORAS and RAS was very similar despite the fact that the RAS method avoids assembly of the subdomain matrices by restricting the global matrix. The

ORAS and RAS preconditioners fell short of linear scaling between two and eight processors. I used a separate scale for the PETSc timing since it took much longer to compute than my own preconditioners. Although the PETSc took between one and two orders of magnitude longer to solve the EM problem using similar parameters, its scaling was slightly better than linear between two and eight processors. The difference in performance between my RAS and PETSc's RAS preconditioner could be explained by the communication burden imposed by the larger overlap of the PETSc version. However, if that were true then I would have expected the time gap between the PETSc and my RAS preconditioner to increase with the number of processors, not decrease as it does in Figure 3.9.

The last two figures (Figure 3.10 and Figure 3.11) demonstrate the behaviour of the Schwarz preconditioners for increasing grid size using the full twelve processors available to me. In terms of iteration count, both Schwarz preconditioners scaled at least linearly judging by the roughly doubling iteration count for the factor of five increase in tetrahedra between 100000 and 500000. However, in terms of time to solution, the Schwarz preconditioners scaled worse than linearly with the time increasing by a factor of nine for the same range of tetrahedra, suggesting that there are inefficiencies in the setup portion of the code.

Based on the evidence provided in these experiments, I would have to recommend the RAS over the ORAS preconditioner for the geophysical EM problem. Due to the requirement for unstructured mesh refinement about dipole sources in these experiments, the ORAS preconditioner did not seem to meet its expectations. I know of one improvement that can be made on the current ORAS implementation, and one other that could benefit both the RAS and ORAS preconditioners, but I will save the

discussion of these for the next, and concluding, chapter.

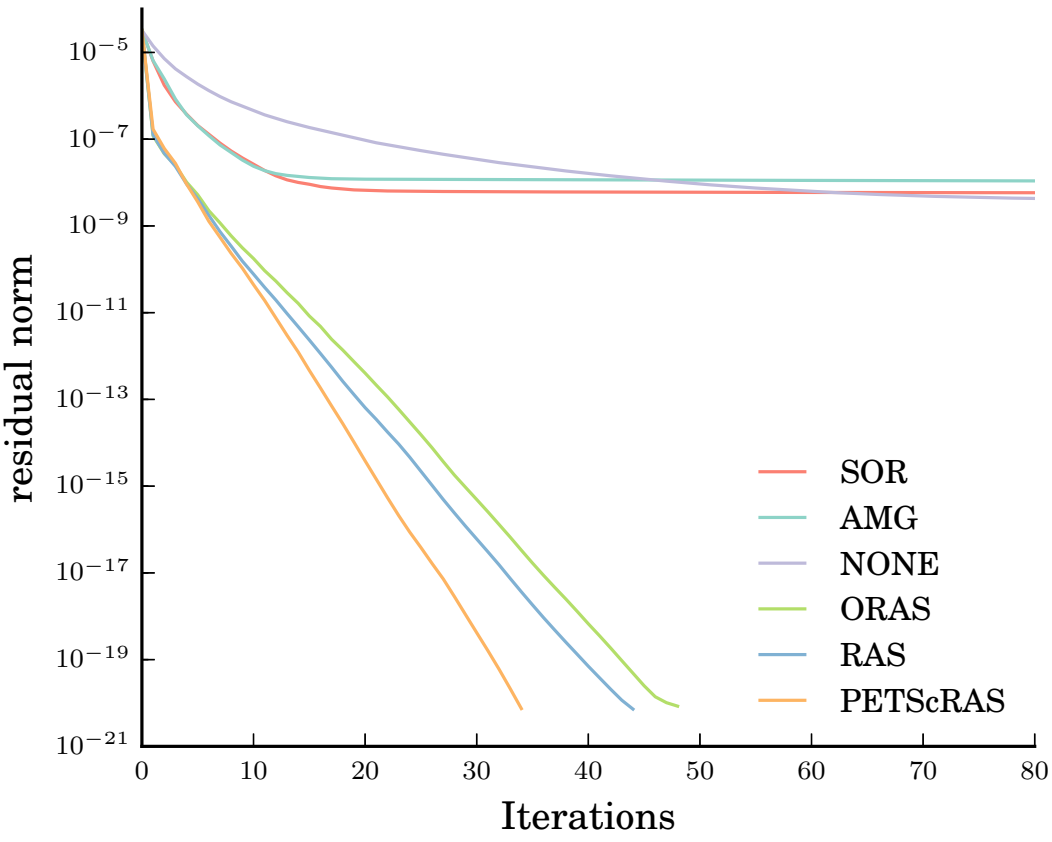

Figure 3.8: Comparison of preconditioner performance.

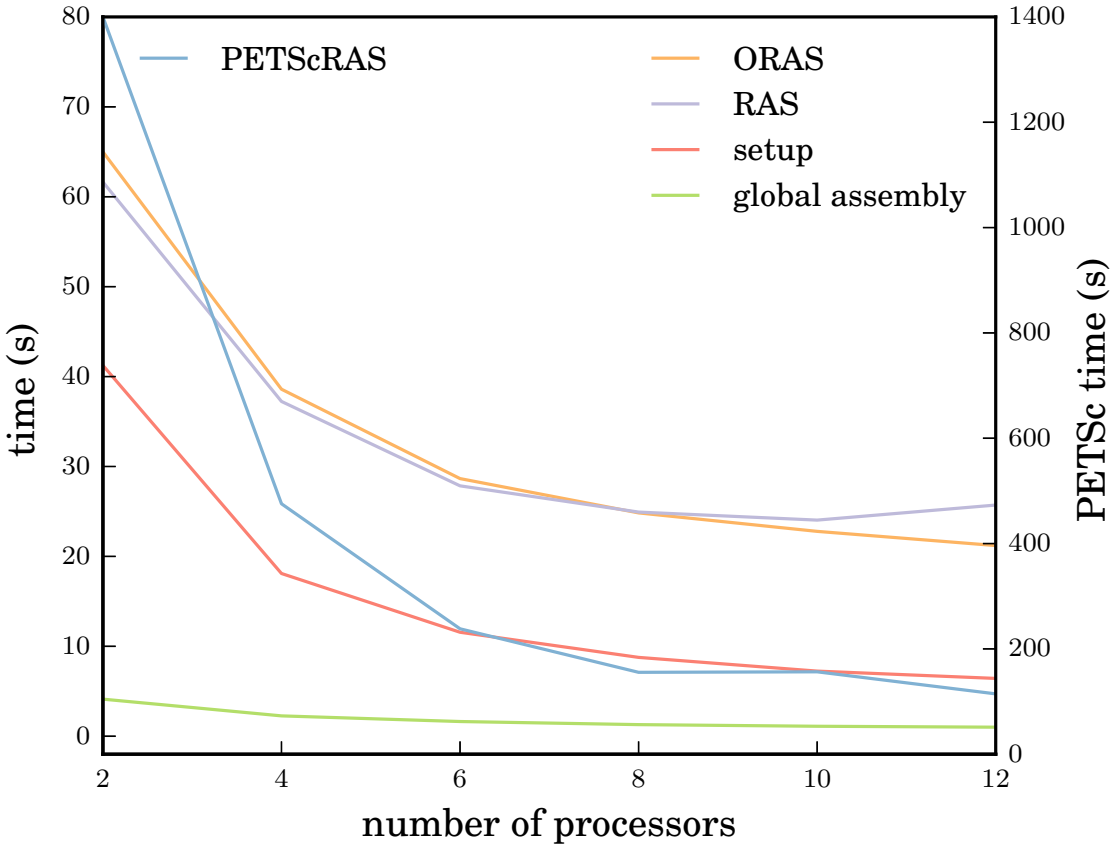

Figure 3.9: Wall clock timings for increasing number of processors.

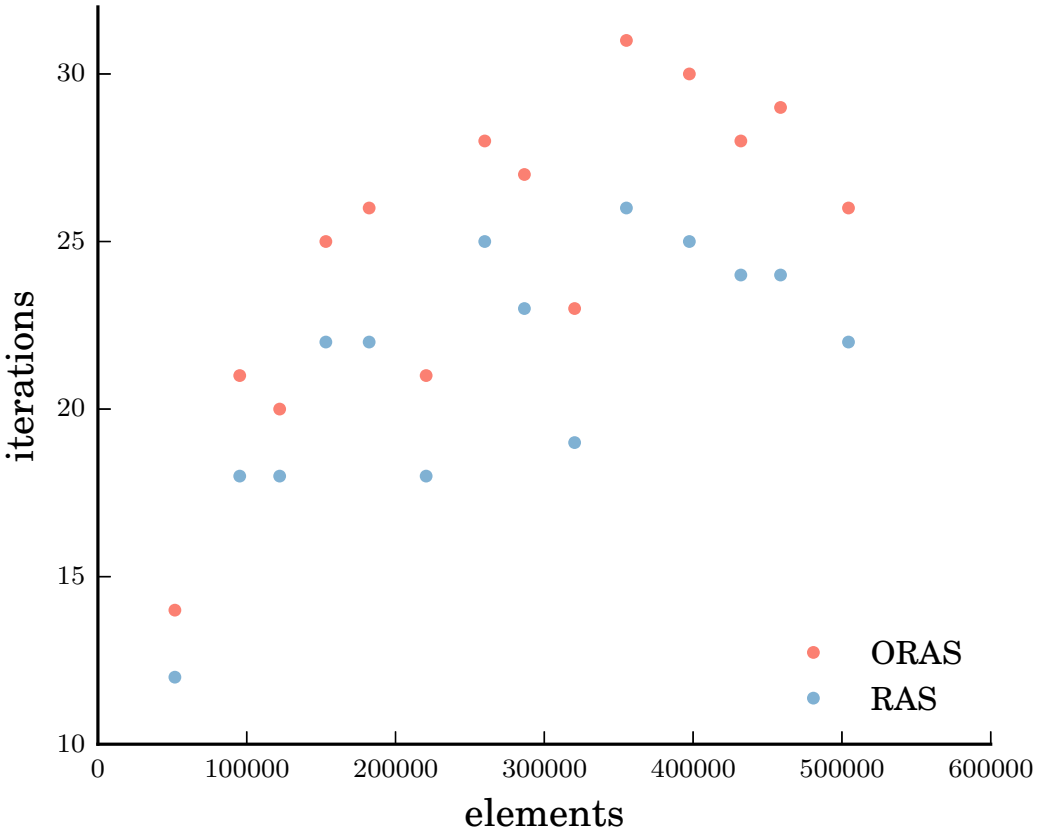

Figure 3.10: Grid size dependent iteration count.

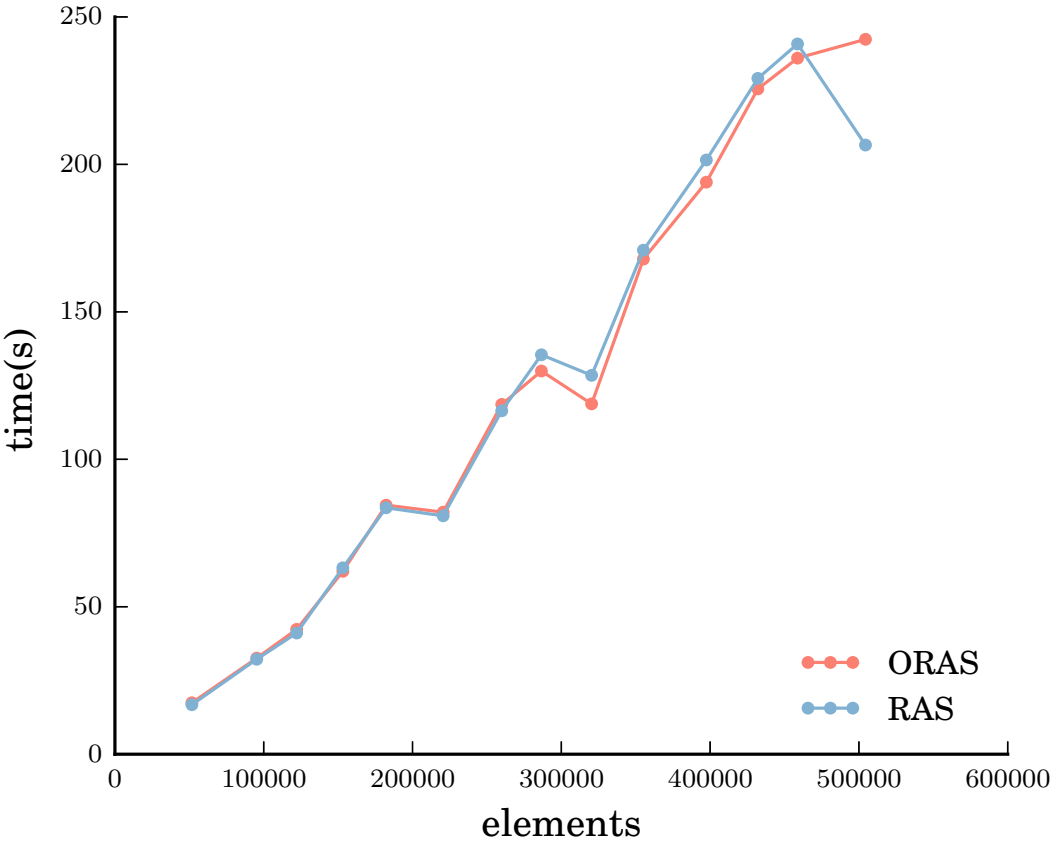

Figure 3.11: Grid size dependent wall clock time.

Chapter 4

Conclusions

The geophysical EM problem poses a significant computational challenge. Not only is the size of the computational problem generated by industrial applications a hurdle, but the physics underlying the geophysical EM experiment makes matters worse. The nullspace of the curl operator creates a highly ill-conditioned matrix when the second order electric field equations are discretized with the FEM. I investigated Schwarz preconditioners in an attempt to mitigate the ill-conditioning problem. The optimized Schwarz method showed great promise when applied to the Poisson problem, but the magnetic dipole source in the geophysical EM problem created a need for unstructured mesh refinement, leading to partitioning strategies that reduced the ORAS preconditioner performance. Nonetheless, I was able to develop a preconditioner that outperformed the options available through PETSc, and there may still be ways to improve upon my Schwarz preconditioners.

Multilevel DD preconditioners are known to outperform single level versions like those that I implemented. A multilevel DD preconditioner adds a coarse grid correc-

tion step, and it can be added to either the RAS or ORAS preconditioning routines to create a composite preconditioner. Migliorati and Quarteroni (2011), for example, show that a multilevel additive Schwarz preconditioner improved the convergence behaviour for the Poisson equation from 155 to 39 iterations by adding a second level.

The ORAS preconditioner can be improved further by considering the second order optimization of the impedance condition. Second order optimizations seek to conserve not only the first derivative of the solution at the interface, but the second derivative. There are several different versions of the second order optimization which have been shown to reduce the iteration count over the first order optimization discussed here (Dolean et al., 2015b). The second order optimizations contain more than one optimization parameter, and so require careful analysis to optimize compared to the scalar α that can be optimized by trial and error, like in this thesis.

On top of the improvements to the DD methods, there may be other ways to increase the efficiency. The DD method could be extended to either the A/Phi potential formulation or the time domain formulation of EM equations. Since the condition number depends on the quality of the underlying mesh, a more careful study of the meshing and, in particular, the refinement procedure could result in an improved performance. The poor scaling of the time to solution with respect to grid size suggests that the algorithm contains some inefficiencies in the setup phase since the number of iterations scaled well as the grid size increased. The portion of the setup which relies on looping through the mesh elements would be a likely area in need of improvement. If it were possible to wrap the iterations in C++ code, I think this could improve the time scaling. Finally, the FEniCS library is in a period of rapid growth, and I am anticipating two updates that could greatly improve my code: Support for complex

arithmetic, and the ability to solve problems within a submesh. The former might only improve code readability, but the latter would remove the need to loop through mesh elements to create the local to global DOF maps and likely improve the time scalability issue just mentioned.

As long as the trend in computing technology is towards multicore technologies, a need for parallel algorithms will exist. The Domain Decomposition strategy used in this thesis fulfills this niche nicely and I would not be surprised if a DD technique is used in the first commercial inversion software for the geophyscial EM equations – a benchmark that cannot be too far away.

Bibliography

Patrick R. Amestoy, Iain S. Duff, Jean-Yves L'Excellent, and Jacko Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *Society for Industrial and Applied Mathematics*, 23(1):15–41, 2001.

Seyedmasoud Ansari and Colin G. Farquharson. 3D finite-element forward modeling of electromagnetic data using vector and scalar potentials and unstructured grids. *Geophysics*, 79(4):E149–E165, 2014.

D. Aruliah and U. M. Ascher. Multigrid Preconditioning for Krylov Methods for Time-Harmonic Maxwell's Equations in Three Dimensions. *SIAM Journal on Scientific Computing*, 24(2):702–718, 2002.

Uri M. Ascher. *Numerical Methods for Evolutionary Differential Equations*. SIAM, Philadelphia, PA, 2008.

Dmitry Avdeev and Sergei Knizhnik. 3D Integral Equation Modeling with a Linear Dependence on Dimensions. *Geophysics*, 74(5):F89–F94, 2009.

Eugene A. Badea, Mark E. Everett, Gregory A. Newman, and Oszkar Biro. Finite-Element Analysis of Controlled-Source Electromagnetic Induction Using Coulomb-Gauged Potentials. *Geophysics*, 66(3):786–799, 2001.

Shaaban A. Bakr, David Pardo, and Trond Mannseth. Domain decomposition Fourier finite element method for the simulation of 3D marine CSEM measurements. *Journal of Computational Physics*, 255:456–470, 2013.

Alexander Bihlo, Colin G. Farquharson, Ronald D. Haynes, and J. Concepcion Loredo-Osti. Probabilistic domain decomposition for the solution of the two-dimensional magnetotelluric problem. *Computational Geosciences*, pages 1–13, 2016.

Ralph Uwe Börner. Numerical modelling in geo-electromagnetics: Advances and challenges. *Surveys in Geophysics*, 31(2):225–245, 2010.

Hongzhu Cai, Č. Martin, and Michael S. Zhdanov. Three-dimensional parallel edge-based finite element modeling of electromagnetic data with field redatuming. pages 1012–1017, 2015.

Xiao-Chuan Cai and Marcus Sarkis. A Restricted Additive Schwarz Preconditioner for General Sparse Linear Systems. *SIAM Journal on Scientific Computing*, 21(2): 792–797, 1999.

Niels B. Christensen and Max Halkjær. Mapping pollution and coastal hydrogeology with helicopter–borne transient electromagnetic measurements. *Exploration Geophysics*, 45(4):243–254, 2014.

Yonghyun Chung, Jeong Sul Son, Tae Jong Lee, Hee Joon Kim, and Changsoo Shin. Three-dimensional modelling of controlled-source electromagnetic surveys using an edge finite-element method with a direct solver. *Geophysical Prospecting*, 62(6): 1468–1483, 2014.

Michael Commer and Gregory Newman. A parallel finite-difference approach for 3D transient electromagnetic modeling with galvanic sources. *Geophysics*, 69(5): 1192–1202, 2004.

Michael Commer, F. R. Maia, and Gregory A. Newman. Iterative Krylov solution methods for geophysical electromagnetic simulations on throughput-oriented processing units. *International Journal of High Performance Computing Applications*, 26(4):378–385, 2011.

Steven Constable. Ten years of marine CSEM for hydrocarbon exploration. *Geophysics*, 75(5):75A67, 2010.

Daoud S. Daoud and Martin J. Gander. Overlapping Schwarz Waveform Relaxation for Advection Reaction Diffusion Equations. 2010.

Victorita Dolean, Martin J. Gander, Stephane Lanteri, Jin Fa Lee, and Zhen Peng. Effective transmission conditions for domain decomposition methods applied to the time-harmonic curl-curl Maxwell's equations. *Journal of Computational Physics*, 280:232–247, 2015a.

Victorita Dolean, Martin J. Gander, Stephane Lanteri, Jin Fa Lee, and Zhen Peng. Effective transmission conditions for domain decomposition methods applied to the time-harmonic curl-curl Maxwell's equations. *Journal of Computational Physics*, 280:232–247, 2015b.

Victorita Dolean, Pierre Jolivet, and Frederic Nataf. *An Introduction to Domain Decomposition Algorithms*. SIAM, Philadelphia, PA, 2015c.

J.J. Dongarra, I.S. Duff, D.C. Sorensen, and H.A. Van der Vorst. *Numerical Linear Algebra for High-Performance Computers*. SIAM, Philadelphia, PA, 1998.

Mark E. Everett. Theoretical Developments in Electromagnetic Induction Geophysics with Selected Applications in the Near Surface. *Surveys in Geophysics*, 33(1):29–63, 2012.

Colin G. Farquharson and Marion P. Miensopust. Three-dimensional finite-element modelling of magnetotelluric data with a divergence correction. *Journal of Applied Geophysics*, 75(4):699–710, 2011.

Haohuan Fu, Yingqiao Wang, Evan Schankee Um, Jiarui Fang, Tengpeng Wei, Xiaomeng Huang, and Guangwen Yang. A parallel finite-element time-domain method for transient electromagnetic simulation. *Geophysics*, 80(4):E213–E224, 2015.

Martin J. Gander. Optimized Schwarz Methods. *SIAM Journal On Numerical Analysis*, 44(2):699–731, 2006.

Martin J. Gander and Hui Zhang. Optimized schwarz methods with overlap for the helmholtz equation. *Lecture Notes in Computational Science and Engineering*, 98 (11371287):207–215, 2014.

Martin J. Gander, Laurence Halpern, and Frederic Nataf. Optimized Schwarz Method for Laplace's Equation. In Tony Chan, Takashi Kako, Hideo Kawarada, and Olivier Pironneau, editors, *12th International Conference on Domain Decomposition Methods*, 2001.

Christophe Geuzaine and Jean François Remacle. Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009.

Mark S. Gockenback. *Understanding and Implementing the Finite Element Method*. SIAM, Philadelphia, PA, 2006.

F.S. Grant and Gordon Fox West. *Interpretation Theory in Applied Geophysics*. McGraw-Hill, New York, 1965.

Alexander V. Grayver. Parallel three-dimensional magnetotelluric inversion using adaptive finite-element method. Part I: Theory and synthetic study. *Geophysical Journal International*, 202(1):584–603, 2015.

Alexander V. Grayver and Markus Burg. Robust and scalable 3-D geo-electromagnetic modelling approach using the finite element method. *Geophysical Journal International*, 198(1):110–125, 2014.

E Haber and U M Ascher. Fast Finite Volume Simulation of 3D Electromagnetic Problems With Highly Discontinuous Coefficients . *SIAM Journal on Scientific Computing*, 22(6):1943–1961, 2001.

E. Haber and L. Ruthotto. A multiscale finite volume method for Maxwell's equations at low frequencies. *Geophysical Journal International*, 199(2):1268–1277, 2014.

Eldad Haber and Stefan Heldmann. An octree multigrid method for quasi-static Maxwell's equations with highly discontinuous coefficients. *Journal of Computational Physics*, 223(2):783–796, 2007.

Eldad Haber, Douglas W. Oldenburg, and R. Shekhtman. Inversion of time domain three-dimensional electromagnetic data. *Geophysical Journal International*, 171(2): 550–564, 2007.

Ralf Hiptmair and Jinchao Xu. Nodal Auxiliary Space Preconditioning in H (curl) and H (div) Spaces. *SIAM Journal on Numerical Analysis*, 45(6):2483–2509, 2007.

Ralf Hiptmair and Jinchao Xu. Auxiliary space preconditioning for edge elements. *IEEE Transactions on Magnetics*, 44(6):938–941, 2008.

Lior Horesh and Eldad Haber. A Second Order Discretization of Maxwell's Equations in the Quasi-Static Regime on OcTree Grids. *SIAM Journal on Scientific Computing*, 33(5):2805–2822, 2011.

J. M. Hyman and M. Shashkov. Mimetic Finite Difference Methods for Maxwell's Equations and the Equations of Magnetic Diffusion. *Progress in Electromagnetics Research*, 32:89–121, 2001.

H. Jahandari and C.G. Farquharson. Finite-volume modelling of geophysical electromagnetic data on unstructured grids using potentials. *Geophysical Journal International*, 202(3):1859–1876, 2015.

Piyoosh Jaysaval, Daniil V. Shantsev, and Sébastien de la Kethulle de Ryhove. Efficient 3-D controlled-source electromagnetic modelling using an exponential finite-difference method. *Geophysical Journal International*, 203(3):1541–1574, 2015.

J. Jin. The Finite Element Method in Electromagnetics, 2002.

J. J. Lajoie and G. F. West. Electromagnetic response of a conductive inhomogeneity in a layered earth. *International Journal of Rock Mechanics and Mining Sciences & Geomechanics Abstracts*, 14(2):33, 1977.

Dominique LaSalle and George Karypis. Multi-threaded graph partitioning. *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium, IPDPS 2013*, pages 225–236, 2013.

R. J. LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems*. SIAM, Philadelphia, PA, 2007.

Gang Li, Lili Zhang, and Bo Han. Stable Electromagnetic Modeling Using a Multigrid Solver on Stretching Grids: The Magnetotelluric Example. *IEEE Geoscience and Remote Sensing Letters*, 13(3):334–338, 2016.

S. Li, H. Sun, X. Lu, and X. Li. Three-dimensional Modeling of Transient Electromagnetic Responses of Water-bearing Structures in Front of a Tunnel Face. *Journal of Environmental & Engineering Geophysics*, 19:13–32, 2014.

Pierre-Louis Lions. On the Schwarz alternating method. I, 1988.

J. A. Meijerink and H. A. van der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Mathematics of Computation*, 31(137):148–148, 1977.

Carl D. Meyer. *Matrix Analysis and Applied Linear Algebra*. SIAM, Philadelphia, PA, 2000.

Giovanni Migliorati and Alfio Quarteroni. Multilevel Schwarz methods for elliptic partial differential equations. *Computer Methods in Applied Mechanics and Engineering*, 200(25-28):2282–2296, 2011.

Yuji Mitsuhata and Toshihiro Uchida. 3D magnetotelluric modeling using the T-Ω finite-element method. *Geophysics*, 69(1):108–119, 2004.

Peter Monk. *Finite Element Methods for Maxwell's Equations*. Oxford University Press Inc., New York, 2003.

Gerard Muñoz. Exploring for Geothermal Resources with Electromagnetic Methods. *Surveys in Geophysics*, 35(1):101–122, 2014.

Misac N. Nabighian and C. Macnae. Time Domain Electromagnetic Prospecting Methods. In Misac N. Nabighian, editor, *Electromagnetic Methods in Applied Geophysics*, chapter 6, pages 427–520. Society of Exploration Geophysicists, 2 edition, 1991.

Myung Jin Nam, Hee Joon Kim, Yoonho Song, Tae Jong Lee, Jeong Sul Son, and Jung Hee Suh. 3D magnetotelluric modelling including surface topography. *Geophysical Prospecting*, 55(2):277–287, 2007.

Gregory A. Newman, Gerald W. Hohmann, and Walter L. Anderson. Transient electromagnetic response of a three-dimenional body in a Layered Earth. *Geophysics*, 51(8):1608–1627, 1986.

Zhengyong Ren, Thomas Kalscheuer, Stewart Greenhalgh, and Hansruedi Maurer.

A finite-element-based domain-decomposition approach for plane wave 3D electromagnetic modeling. *Geophysics*, 79(6):E255–E268, 2014.

Tawat Rung-Arunwan and Weerachai Siripunvaraporn. An efficient modified hierarchical domain decomposition for two-dimensional magnetotelluric forward modelling. *Geophysical Journal International*, 183(2):634–644, 2010.

Kethulle De Ryhove, Patrick R. Amestoy, Alfredo Buttari, Jean-yves L. Excellent, and Theo Mary. Large-scale 3-D EM modelling with a Block Low-Rank multifrontal direct solver. *Geophysical Journal International*, 209:1558–1571, 2017.

Yousef Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelphia, PA, 2nd edition, 1995.

Yousef Saad and Henk A. van der Vorst. Iterative solution of linear systems in the 20th century. *Journal of Computational and Applied Mathematics*, 123(1):1–33, 2000.

V. Sapia, A. Viezzoli, F. Jorgensen, G. A. Oldenborger, and M. Marchetti. The Impact on Geological and Hydrogeological Mapping Results of Moving from Ground to Airborne TEM. *Journal of Environmental and Engineering Geophysics*, 19(1): 53–66, 2014.

Olaf Schenk and Klaus Gartner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Generation Computer Systems*, 20(3):475–487, 2004.

H. A. Schwarz. Über einen Grenzübergang durch alternierendes Verfahren. *Vierteljahrsschrift der Naturforschenden Gesellschaft*, 15(15):272–286, 1870.

Christoph Schwarzbach and Eldad Haber. Finite element based inversion for time-harmonic electromagnetic problems. *Geophysical Journal International*, 193(2): 615–634, 2013.

Christoph Schwarzbach, Ralph Uwe Börner, and Klaus Spitzer. Three-dimensional adaptive higher order finite element simulation for geo-electromagnetics – a marine CSEM example. *Geophysical Journal International*, 187(1):63–74, 2011.

A St-Cyr, M. J. Gander, and S. J. Thomas. Optimized Multiplicative, Additive, and Restricted Additive Schwarz Preconditioning. *SIAM Journal on Scientific Computing*, 23(29):2402 – 2425, 2007.

Rita Streich. 3D finite-difference frequency-domain modeling of controlled-source electromagnetic data: Direct solution and optimization for high accuracy. *Geophysics*, 74(5):F95–F105, 2009.

Andrea Toselli and Olof Widlund. *Domain Decomposition Methods - Algorithms and Theory*, volume 34. Springer-Verlag, Berlin, Heidelberg, 2004.

Ulrich Trottenberg, Cornelis Oosterlee, and Anton Schuller. *Multigrid*. Elsevier, London, 1 edition, 2001.

Evan Schankee Um, Jerry M. Harris, and David L. Alumbaugh. 3D time-domain simulation of electromagnetic diffusion phenomena: A finite-element electric-field approach. *Geophysics*, 75(4):F115–F126, 2010.

Evan Schankee Um, Michael Commer, and Gregory A. Newman. Efficient preconditioned iterative solution strategies for the electromagnetic diffusion in the

earth: Finite-element frequency-domain approach. *Geophysical Journal International*, 193(3):1460–1473, 2013.

T. Wang and G. W. Hohmann. A finite-difference, time-domain solution for three-dimensional electromagnetic modelling. *Geophysics*, 58(6):797–809, 1993.

Philip E. Wannamaker, Gerald W. Hohmann, and William A. SanFilipo. Electromagnetic modeling of threedimensional bodies in layered earths using integral equations, 1984.

Stanley H. Ward and Gerald W. Hohmann. Electromagnetic theory for geophysical application. In Misac N. Nabighian, editor, *Electromagnetic Methods in Applied Geophysics*, volume 1, chapter 4, pages 13–196. Society of Exploration Geophysicists, 1988.

Chester J. Weiss. Project APhiD: A Lorenz-gauged A-Phi decomposition for parallelized computation of ultra-broadband electromagnetic induction in a fully heterogeneous Earth. *Computers and Geosciences*, 58:40–52, 2013.

Chester J. Weiss and Gregory A. Newman. Electromagnetic induction in a fully 3-D anisotropic earth. *Geophysics*, 67(4):1104, 2002.

Zonghou Xiong. Domain decomposition for 3D electromagnetic modeling. *Earth Planets Space*, 51:1013–1018, 1999.

Dikun Yang, Douglas W. Oldenburg, and Eldad Haber. 3-D inversion of airborne electromagnetic data parallelized and accelerated by local mesh and adaptive soundings. *Geophysical Journal International*, 196(3):1492–1507, 2014.

Nikolay Yavich and Michael S. Zhdanov. Contraction pre-conditioner in finite-difference electromagnetic modelling. *Geophysical Journal International*, 206(3): 1718–1729, 2016.

Fabio I. Zyserman and Juan E. Santos. Parallel finite element algorithm with domain decomposition for 3D MT modelling. *Applied Geophysics*, 2000.

Appendix A

One dimensional finite difference domain decomposition with equal subdomains

In order to investigate the curious behavior of the one dimensional finite difference domain decomposition, I use the error equations. I assume $e = u - u_{true}$, so that by subtracting the true solution from u in Equation (2.1), the problem takes the form:

Solve:

$$\nabla^2 e = 0 \quad \text{in } \Omega \tag{A.1a}$$

$$e = 0 \quad \text{on } \partial\Omega, \tag{A.1b}$$

I now choose a small discretization of the one dimensional space between zero and one. For $N = 5$, I have two subdomains with three nodes each as shown in Figure

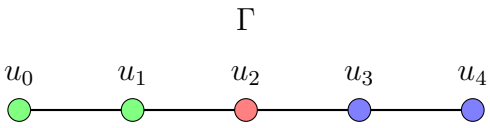

Figure A.1: A simple decomposition of the 1D mesh into two subdomains

A.1. The discrete system of equations for the left side with outer boundary conditions and an interface condition can be written as

$$-2e^0_{u1} + e^0_{u2} = 0 \tag{A.2a}$$

$$e^0_{u1} - (1 + h\alpha)e^0_{u2} = -hg^0_u. \tag{A.2b}$$

Since I have two equations and two unknowns, it is straightforward to solve this system analytically. From Equation (A.2a), I get that

$$e^0_{u2} = 2e^0_{u1}, \tag{A.3}$$

from which I may substitute e^0_{u2} into Equation (A.2b) to get

$$e^0_{u1} = \frac{hg^0_u}{1 + 2h\alpha}. \tag{A.4}$$

Then I use this in conjunction with Equation (A.3) and find

$$e^0_{u2} = \frac{2hg^0_u}{1 + 2h\alpha}. \tag{A.5}$$

The solution for the right subdomain follows the same steps and I obtain

$$e^0_{u1} = \frac{hg^0_u}{1+2h\alpha}, \qquad e^0_{u2} = \frac{2hg^0_u}{1+2h\alpha}, \qquad e^0_{v0} = \frac{2hg^0_v}{1+2h\alpha}, \qquad e^0_{v1} = \frac{hg^0_v}{1+2h\alpha}, \qquad . \tag{A.6}$$

I now update g^0_u and g^0_v through the familiar equations

$$g^1_u = -g^0_v + 2\alpha e^0_{v0}, \qquad g^1_v = -g^0_u + 2\alpha e^0_{u2}, \tag{A.7}$$

and substitute the expressions for e^0_{u1}, e^0_{u2}, e^0_{v0}, and e^0_{v1} from Equations (A.6). This gives me

$$g^1_u = -g^0_v + \frac{4h\alpha g^0_v}{1+2h\alpha} = \left(\frac{4h\alpha}{1+2h\alpha} - 1\right) g^0_v = \left(\frac{2h\alpha - 1}{1+2h\alpha}\right) g^0_v, \tag{A.8}$$

$$g^1_v = \left(\frac{2h\alpha - 1}{1+2h\alpha}\right) g^0_u. \tag{A.9}$$

This is followed by another round of subdomain solves to find the errors at iteration 1, now in terms of g^1_u and g^1_v. Since these will be equivalent to Equations (A.6) but with the superscript 1, I skip directly to the result from the substitution of the expressions for g^1_u and g^1_v.

$$e^1_{u1} = \left(\frac{h(2h\alpha - 1)}{(1+2h\alpha)^2}\right) g^0_v, \qquad e^0_{u2} = \left(\frac{2h(2h\alpha - 1)}{(1+2h\alpha)^2}\right) g^0_v, \tag{A.10}$$

$$e^0_{v0} = \left(\frac{2h(2h\alpha - 1)}{(1+2h\alpha)^2}\right) g^0_u, \qquad e^0_{v1} = \left(\frac{h(2h\alpha - 1)}{(1+2h\alpha)^2}\right) g^0_u. \tag{A.11}$$

From my five point discretization, I have that $h = 1/4$, and since I want to analyze the result with $\alpha = 2$ I substitute these values into the expressions for the error and find that the bracketed term in the numerator vanishes.

If the same procedure is followed for finer discretizations, the same result will occur. For example, when I use seven points instead of five I have four point subdomains and the second round of subdomain solves yields for the left subdomain

$$e^1_{u1} = \frac{h(3h\alpha - 1)}{3h\alpha + 1}, \qquad e^1_{u2} = 2e^1_{u1}, \qquad e^1_{u3} = 3e^1_{u1}. \tag{A.12}$$

With the extra two points added between zero and one, I now have $h = 1/6$ and again the bracketed term in the numerator disappears.

It looks like for every choice of discretization, the solution to the error equations after two rounds of the optimized Schwarz procedure will give a term in the form $(\frac{n-1}{N-1}\alpha - 1)$ where n is the number of subdomain nodes and N is the global number of nodes. The choice of $\alpha = 2$ can now be seen to produce 2-iteration convergence since n is always smaller than N by a factor of two for non-overlapping methods where the domain is split in two equal halves.

Appendix B

Using a property of the Kronecker product to extend one dimensional discretizations into two dimensions

The action of the Kronecker product is to create a block matrix in the following manner

$$A \otimes B = \begin{pmatrix} a_{11}B & a_{12}B & \dots \\ a_{21}B & a_{22}B & \dots \\ \vdots & \vdots & \ddots \end{pmatrix} \tag{B.1}$$

Note also that the operator denoted by vec() can refer to an unraveling of the matrix represented by Figure B.1 in row, or column major ordering, giving

$$\begin{pmatrix} u_{11} \\ u_{12} \\ u_{13} \\ u_{21} \\ u_{22} \\ u_{23} \\ u_{31} \\ u_{32} \\ u_{33} \end{pmatrix} \quad \text{or} \quad \begin{pmatrix} u_{11} \\ u_{21} \\ u_{31} \\ u_{12} \\ u_{22} \\ u_{32} \\ u_{13} \\ u_{23} \\ u_{33} \end{pmatrix}. \tag{B.2}$$

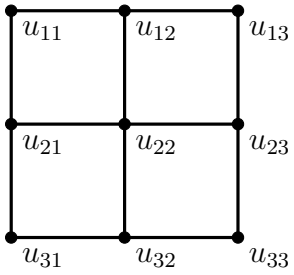

Figure B.1: The elements of the mesh that form the matrix U

The Kronecker product property (Equation (2.86)) holds for any combination of matrices A, X and B, but I would like to apply the property in order to ease the assembly of a two dimensional system of equations of the form $A_{2D}u = f$, where A_{2D} is a finite difference discretization of the two dimensional Laplacian operator,

$$\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}, \tag{B.3}$$

and f is a load vector in row or column major order. I define the matrix $A =$

A_{1D} as the one dimensional system matrix arising from the discretization of the Laplacian, $X = U$ as the matrix representing the solution of the Poisson problem in two dimensions, and $B = I$ as the Identity matrix (a convenient choice that fulfills Equation (2.86) without affecting the PDE that I want to apply the property to), and I write the multiple right hand sides linear algebra problem $A_{1D}U = F$ with a matrix F that discretizes the load function in two dimensions. The solution U would be the matrix formed by columns of one dimensional solutions of the form $A_{1D}u_i = f_i$. To apply the Kronecker product property, I transform the matrix equation by right multiplying the left side by the innocuous identity matrix and vectorizing both sides

$$\text{vec}(A_{1D}UI) = \text{vec}(F), \tag{B.4}$$

and then applying the property

$$(I^T \otimes A_{1D})\text{vec}(U) = \text{vec}(F). \tag{B.5}$$

With this choice for the matrix B, the kronecker product $I^T \otimes A_{1\text{D}}$ simply gives

$$I^T \otimes A_{1\text{D}} = \begin{pmatrix} A_{1\text{D}} & 0 & 0 & \dots \\ 0 & A_{1\text{D}} & 0 & \dots \\ 0 & 0 & A_{1\text{D}} & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}, \tag{B.6}$$

where the matrix A_{1D} is repeated in a block diagonal fashion as many times as there are rows in the identity matrix. The structure of this large sparse matrix is exactly the form needed to build one of the components of the matrix A_{2D}. The matrix in Equation (B.6) can be either the x or y component of the Laplacian depending on

whether I choose to vectorize F in row or column major order. If I choose to vectorize F with row major ordering as in the left vector in Equation (B.2), then A_{1D} must have the correct dimensions to difference along the x axis, and $I^T = I$ must have as many rows as there are mesh points along the y axis.

By choosing the order of vec(F), I fix the action of the matrix in Equation (B.6) to work in a particular dimension and in order to define a similar matrix for the other dimension, I need to use the Kronecker product in a new fashion. Keeping with the choice of row major ordering, I propose that the following use of the Kronecker product property yields a two dimensional finite difference matrix that discretizes the y component of the Laplacian

$$IUA_{1D}^T = (A_y \otimes I_x)\text{vec}(U), \tag{B.7}$$

where A_y is sized for the y dimension, and I_x for the x dimension. The Kronecker product resulting from this transformation has the block form

$$A_{1D} \otimes I = \begin{pmatrix} a_{11}I_x & a_{12}I_x & a_{13}I_x & \dots \\ a_{21}I_x & a_{22}I_x & a_{23}I_x & \dots \\ a_{31}I_x & a_{32}I_x & a_{33}I_x & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}, \tag{B.8}$$

where the differencing occurs along the y component since the coefficients of the one dimensional discrete Laplacian have been multiplied by the identity matrix with as many rows as there are mesh points in the x dimension. This has the effect of spreading out the finite difference coefficients so that they pick out element of vec(U) to re-create the stencil down the columns of U. Since the two dimensional Laplacian

can be constructed by adding the two components as in Equation (B.3), I can find the two dimensional solution of Poisson's equation by forming and solving the matrix equation

$$((A_x \otimes I_y) + (I_x \otimes A_y))u = \text{vec}(F). \tag{B.9}$$

Notice that I can now solve the two dimensional problem using only one dimensional discretizations for the extra cost of computing a Kronecker product and vectorizing the right hand side. I find this method especially useful when dealing with boundary conditions since these are much more easily implemented in one dimension. The matrices A_x and A_y are exactly those that I used in the one-dimensional experiments, so the only task left is to update the two-dimensional F for the boundary conditions, vectorize it and then solve Equation (B.9). For a more mathematical treatment of this method, see Meyer (2000).

Appendix C

Finite element assembly routines

```
1
2 ## gaussian_quadrature_2d ##
3
4 def gaussian_quadrature_2d(funcs, order, x, y):
5
6     N0 = lambda chi, eta: 1 - chi - eta
7     N1 = lambda chi: chi
8     N2= lambda eta: eta
9     P = lambda chi, eta: x[0]*(1 - chi - eta) + x[1]*chi + x[2]*eta
10    Q = lambda chi, eta: y[0]*(1 - chi - eta) + y[1]*chi + y[2]*eta
11
12    if order == 1:
13        w   = [1.]
14        chi = [1/3.]
15        eta = [1/3.]
16
17    if order == 2:
```

```
18          w   = [1/3., 1/3., 1/3.]
19          chi = [1/6., 2/3., 1/6.]
20          eta = [1/6., 1/6., 2/3.]
21
22      Area = abs(((x[1] - x[0])*(y[2] - y[0])) - ((x[2] - x[0])*(y[1] - y
        [0])))/2
23
24      f_int = 0
25
26      for i in range(len(w)):
27          f = 1
28          for j in range(len(funcs)):
29              f *= funcs[j](P(chi[i], eta[i]), Q(chi[i], eta[i]))
30          f_int += Area * w[i] * f
31
32      return f_int
33
34  ## gaussian_quadrature_1d ##
35
36  def gaussian_quadrature_1d(funcs, order, y):
37
38      if order == 1:
39          w   = [2.]
40          chi = [0.]
41
42      if order == 2:
43          w   = [1., 1.]
44          chi = [-1/np.sqrt(3), 1/np.sqrt(3)]
```

```
45
46      if order == 3:
47          w   = [5/9., 8/9., 5/9.]
48          chi = [-np.sqrt(3/5.), 0, np.sqrt(3/5.)]
49
50      f_int = 0
51
52      for i in range(len(w)):
53          f = 1
54          for j in range(len(funcs)):
55              #print funcs[j], (y[1] + (((y[0] - y[1])/2.) * (1 + chi[i]))
    ), funcs[j](y[1] + (((y[0] - y[1])/2.) * (1 + chi[i])))
56              f *= funcs[j](y[1] + (((y[0] - y[1])/2.) * (1 + chi[i])))
57          f_int += ((y[1] - y[0])/2.) * w[i] * f
58
59      return f_int
60
61 ## get_nodes ##
62
63 def get_nodes(nx, ny, X, Y, dirichlet_values, robin_values):
64
65      nodes = {}
66
67      for i in range(nx*ny):
68
69          nodes[i] = {'coords' : [X.reshape(nx*ny)[i], Y.reshape(nx*ny)[i
    ]]}
70
```

```
71          if not np.isnan(dirichlet_values.reshape(nx*ny)[i]):
72              nodes[i]['diri'] = dirichlet_values.reshape(nx*ny)[i]
73
74          if not np.isnan(robin_values.reshape(nx*ny)[i]):
75              nodes[i]['robin'] = robin_values.reshape(nx*ny)[i]
76
77      return nodes
78
79  ## get_triangles ##
80
81  def get_triangles(nx, ny, nodes):
82
83      triangles = {}
84      e = 0
85      n = 0
86
87      for r in range(ny-1):
88          for s in range(nx-1):
89
90              triangles[e] = {'v0' : r*nx + s,
91                              'v1' : r*nx + s+1,
92                              'v2' : r*nx + s+nx}
93
94              robin_nodes = []
95              dirichlet_nodes= []
96
97              for v in ['v0', 'v1', 'v2']:
98
```

```
99                if 'robin' in nodes[triangles[e][v]].keys():
100                   robin_nodes.append(triangles[e][v])
101
102               if 'diri' in nodes[triangles[e][v]].keys():
103                   dirichlet_nodes.append(triangles[e][v])
104
105               else:
106
107                   if 'free' not in nodes[triangles[e][v]].keys():
108                       nodes[triangles[e][v]]['free'] = n
109                       n+=1
110
111           if len(robin_nodes) == 2:
112
113               triangles[e]['R'] = robin_nodes
114
115           if True:
116
117               triangles[e]['D'] = dirichlet_nodes
118
119           e+=1
120
121           triangles[e] = {'v0' : r*nx + s+1,
122                           'v1' : r*nx + s+1+nx,
123                           'v2' : r*nx + s+nx}
124
125           robin_nodes = []
126           dirichlet_nodes= []
```

```
127
128             for v in ['v0', 'v1', 'v2']:
129
130                 if 'robin' in nodes[triangles[e][v]].keys():
131                     robin_nodes.append(triangles[e][v])
132
133                 if 'diri' in nodes[triangles[e][v]].keys():
134                     dirichlet_nodes.append(triangles[e][v])
135
136                 else:
137
138                     if 'free' not in nodes[triangles[e][v]].keys():
139                         nodes[triangles[e][v]]['free'] = n
140                         n+=1
141
142             if len(robin_nodes) == 2:
143
144                 triangles[e]['R'] = robin_nodes
145
146             if len(dirichlet_nodes) > 0:
147
148                 triangles[e]['D'] = dirichlet_nodes
149
150             e+=1
151
152     return triangles
153
154
```

```
155 ## assemble_interface_vector ##
156
157 def assemble_interface_vector(nodes, triangles, f, qord):
158
159     vector = np.zeros(len(nodes.keys()))
160     local_vertices = ['v0', 'v1', 'v2']
161
162     for k in triangles.keys():
163         M = []
164         if 'R' in triangles[k].keys():
165
166             for v in local_vertices:
167                 M.append([nodes[triangles[k][v]]['coords'][0], \
168                           nodes[triangles[k][v]]['coords'][1]])
169
170             M = np.hstack([np.ones((3,1)), np.array(M)])
171
172             Minv = np.linalg.inv(np.array(M))
173             y_interval = [nodes[triangles[k]['R'][0]]['coords'][1], \
174                           nodes[triangles[k]['R'][1]]['coords'][1]]
175
176             for v in local_vertices:
177                 inverse_vertex_map = {triangles[k][v]: int(v[-1]) \
178                                       for v in local_vertices}
179
180
181             def f_interp(y):
182
```

```
183                 phi0 = Minv[0,inverse_vertex_map[triangles[k]['R'][0]]
    + \
184                        Minv[1,inverse_vertex_map[triangles[k]['R'][0]]
    * \
185                        nodes[triangles[k]['R'][0]]['coords'][0] + \
186                        Minv[2,inverse_vertex_map[triangles[k]['R'][0]]
    * y
187
188                 phi1 = Minv[0,inverse_vertex_map[triangles[k]['R'][1]]
    + \
189                        Minv[1,inverse_vertex_map[triangles[k]['R'][1]]
    * \
190                        nodes[triangles[k]['R'][1]]['coords'][0] + \
191                        Minv[2,inverse_vertex_map[triangles[k]['R'][1]]
    * y
192
193                 return (phi0 * f[triangles[k]['R'][0]]) + (phi1 * f[
    triangles[k]['R'][1]])
194
195             for r in range(2):
196                 vr = triangles[k]['R'][r]
197                 phi_eval = lambda y: Minv[0,inverse_vertex_map[vr]] + \
198                                      Minv[1,inverse_vertex_map[vr]] * \
199                                      nodes[triangles[k]['R'][0]]['coords
    '][0] + \
200                                      Minv[2,inverse_vertex_map[vr]] * y
201                 I = gaussian_quadrature_1d([f_interp, phi_eval], qord,
    y_interval)
```

```
202                 vector[triangles[k]['R'][r]] += I
203
204     return vector
205
206
207 ## assemble system ##
208
209 def assemble_system(nodes, triangles, alpha, f_eval, order):
210
211     # Initialize storage
212     K = ssp.lil_matrix((len(nodes.keys()), len(nodes.keys())))
213     Kr = ssp.lil_matrix((len(nodes.keys()), len(nodes.keys())))
214     Ktest = ssp.lil_matrix((len(nodes.keys()), len(nodes.keys())))
215     F = np.zeros(len(nodes.keys()))
216
217     #for k in triangles.keys():
218     #    if 'robin' in trianges[k].keys():
219
220     for k in triangles.keys():
221
222         # Form M from coordinates of vertices and invert to recover the
    hat
223         # function parameters for the current triangle
224
225         M = []
226         local_vertices = ['v0', 'v1', 'v2']
227
228         for v in local_vertices:
```

```
229             M.append(nodes[triangles[k][v]]['coords'])
230
231         M = np.hstack([np.ones((3,1)), np.array(M)])
232         Minv = np.linalg.inv(np.array(M))
233
234         # Area of the kth triangle
235         Area = abs((M[1,1] - M[0,1]) * (M[2,2] - M[0,2]) -
236                    (M[2,1] - M[0,1]) * (M[1,2] - M[0,2]))/2
237
238         # Compute the centroid
239         c = [np.sum(M[:,1])/3., np.sum(M[:,2])/3.]
240
241         # Accumulate contributions to the stiffness matrix
242
243         for r in range(3):
244             for s in range(r,3):
245
246                 i = min([triangles[k][local_vertices[r]],
247                          triangles[k][local_vertices[s]]])
248                 j = max([triangles[k][local_vertices[r]],
249                          triangles[k][local_vertices[s]]])
250
251                 K[i,j] -=  Area * np.dot(Minv[1:,r], Minv[1:,s])
252
253         # Accumulate contributions to the load vector
254
255         #I = Area * f_eval(c[0],c[1]) / 3.
256
```

```
257
258         for r in range(3):
259
260             phi_eval = lambda x, y: Minv[0,r] + Minv[1,r]*x + Minv[2,r]*
    y
261             funcs = [f_eval, phi_eval]
262             I = gaussian_quadrature_2d(funcs, order, M[:,1], M[:,2])
263
264             i = triangles[k][local_vertices[r]]
265             F[i] += I
266
267         # Accumulate contributions to Robin Mass Matrix
268
269         if 'R' in triangles[k].keys():
270
271             y_interval = [nodes[triangles[k]['R'][0]]['coords'][1], \
272                           nodes[triangles[k]['R'][1]]['coords'][1]]
273
274             for v in local_vertices:
275                 inverse_vertex_map = {triangles[k][v]: int(v[-1]) \
276                                       for v in local_vertices}
277             for r in range(2):
278                 for s in range(r,2):
279                     vr = triangles[k]['R'][r]
280                     vs = triangles[k]['R'][s]
281                     phi_eval1 = lambda y: Minv[0,inverse_vertex_map[vr]]
    + \
282                                           Minv[1,inverse_vertex_map[vr]]
```

```
    * \
283                                         nodes[triangles[k]['R'][0]]['
    coords'][0] + \
284                                         Minv[2,inverse_vertex_map[vr]]
    * y
285
286                     phi_eval2 = lambda y: Minv[0,inverse_vertex_map[vs]]
     + \
287                                         Minv[1,inverse_vertex_map[vs]]
    * \
288                                         nodes[triangles[k]['R'][0]]['
    coords'][0] + \
289                                         Minv[2,inverse_vertex_map[vs]]
    * y
290
291                     I = alpha*gaussian_quadrature_1d([phi_eval1,
    phi_eval2], order, y_interval)
292                     K[triangles[k]['R'][r], triangles[k]['R'][s]] -= I
293                     Ktest[triangles[k]['R'][r], triangles[k]['R'][s]] -=
     I
294
295
296     K += ssp.triu(K,1).T
297     return K,F
298
299
300 ## assemble G ##
301
```

```
302 def assemble_G(nodes, triangles, g_eval):
303
304     G = np.zeros(len(nodes.keys()))
305
306     for k in triangles.keys():
307
308         if 'R' in triangles[k].keys():
309
310             if nodes[triangles[k]['R'][0]]['coords'][0] - \
311                nodes[triangles[k]['R'][1]]['coords'][0] < \
312                1e-6:
313
314                 L = abs(nodes[triangles[k]['R'][0]]['coords'][1] -
315                         nodes[triangles[k]['R'][1]]['coords'][1])
316
317                 c =(nodes[triangles[k]['R'][0]]['coords'][1] +
318                     nodes[triangles[k]['R'][1]]['coords'][1])/2.
319
320             else:
321
322                 L = abs(nodes[triangles[k]['R'][0]]['coords'][0] -
323                         nodes[triangles[k]['R'][1]]['coords'][0])
324
325                 c = (nodes[triangles[k]['R'][0]]['coords'][0] +
326                      nodes[triangles[k]['R'][1]]['coords'][0])/2.
327
328             I = L*g_eval(c)/2.
329             for r in range(2):
```

```
330                 G[triangles[k]['R'][r]] += I
331
332     return G
333
334
335 ## apply boundary conditions ##
336
337 def apply_BCs(nodes, K, F):
338
339     for i in nodes.keys():
340
341         if 'diri' in nodes[i].keys():
342             K[i,:] = 0
343             K[i,i] = 1.
344             F[i] = nodes[i]['diri']
345
346     return K, F
347
348
349 def assemble_global_weak_form(dirichlet_values, nodes, triangles, Area,
    f_eval):
350
351
352     # Initialize storage
353     nf = np.sum(np.isnan(dirichlet_values))
354     K = ssp.lil_matrix((nf,nf))
355     A = ssp.lil_matrix((nf,nf))
356     F = np.zeros(nf)
```

```
357     G = np.zeros(nf)
358     DK =  np.zeros(nf)
359     DA =  np.zeros(nf)
360
361     for k in triangles.keys():
362
363
364         # Form M from coordinates of vertices and invert to recover the
    hat
365         # function parameters for the current triangle
366
367         M = []
368         local_vertices = ['v0', 'v1', 'v2']
369
370         for v in local_vertices:
371             M.append(nodes[triangles[k][v]]['coords'])
372
373         M = np.hstack([np.ones((3,1)), np.array(M)])
374         Minv = np.linalg.inv(np.array(M))
375
376         # Compute the centroid
377         c = [np.sum(M[:,1])/3., np.sum(M[:,2])/3.]
378
379
380         # Accumulate contributions to the stiffness matrix
381
382         for r in range(3):
383             for s in range(r,3):
```

```
384
385             if ('free' in nodes[triangles[k][local_vertices[r]]].
    keys()  and \
386                 'free' in nodes[triangles[k][local_vertices[s]]].
    keys()):
387
388                 i = min(nodes[triangles[k][local_vertices[r]]]['free
    '],
389                         nodes[triangles[k][local_vertices[s]]]['
    free'])
390                 j = max(nodes[triangles[k][local_vertices[r]]]['free
    '],
391                         nodes[triangles[k][local_vertices[s]]]['
    free'])
392
393                 K[i,j] -=  Area * np.dot(Minv[1:,r], Minv[1:,s])
394
395     # Accumulate contributions to the load vector
396
397     I = Area * f_eval(c[0],c[1]) / 3.
398     for r in range(3):
399         if 'free' in nodes[triangles[k][local_vertices[r]]].keys():
400             i = nodes[triangles[k][local_vertices[r]]]['free']
401             F[i] += I
402
403     # Apply the Dirichlet condition through the weak form
404
405     # First build up the function D = sum(d_i * Grad Phi_i)
```

```
406         if 'D' in triangles[k].keys():
407             gradD = np.zeros(2)
408             for r in range(3):
409                 if triangles[k][local_vertices[r]] in triangles[k]['D']:
410                     gradD += nodes[triangles[k][local_vertices[r]]]['
    diri'] * \
411                                 Minv[1:,r]
412 
413         # Then accumulate into DK
414             for r in range(3):
415                 if 'free' in nodes[triangles[k][local_vertices[r]]].keys
    ():
416                     i = nodes[triangles[k][local_vertices[r]]]['free']
417                     DK[i] += Area * gradD.dot(Minv[1:,r])
418 
419     K += ssp.triu(K,1).T
420 
421     return K, F, DK
```

Listing C.1: Custom module containing function definitions for the FE assembly of the 2D Poisson problem

```
1 from dolfin import *
2 
3 import numpy as np
4 import matplotlib.pyplot as plt
5 from mpl_toolkits.mplot3d import Axes3D
6 import scipy.sparse as ssp
7 import scipy.sparse.linalg as ssl
```

```
8
9  import logging
10 logging.getLogger("FFC").setLevel(logging.ERROR)  # holds back fenics
       messages
11
12 nx = 21 # number of nodes along x dimension
13 ny = 41
14 qord = 2
15 alpha = 25
16 interp_degree = 6
17 x = np.linspace(0,1,ny)
18 y = np.linspace(0,1,ny)
19 X,Y = np.meshgrid(x,y)
20
21 utrue = (np.sin(2.*np.pi*X - (0.75*np.pi)) * \
22          np.sin(2.*np.pi*Y- 0.75*np.pi) + 2).reshape(ny*ny)
23
24 ufunc = Expression('(sin((2.0*pi*x[0]) - (0.75*pi)) * \
25                      sin((2.0*pi*x[1]) - (0.75*pi))) + 2', degree=
       interp_degree)
26
27 f = Expression('(-8.*pi*pi) * ( sin((2.0*pi*x[0]) - (0.75*pi)) * \
28                  sin((2.0*pi*x[1]) - (0.75*pi)) )', degree=interp_degree
       )
29
30 mesh1 = UnitSquareMesh(nx-1, ny-1, 'left')
31 mesh2 = UnitSquareMesh(nx-1, ny-1, 'left')
32 meshG = UnitSquareMesh(ny-1, ny-1, 'left')
```

```
33
34 mesh1.coordinates()[:,0] = mesh1.coordinates()[:,0]/2
35 mesh2.coordinates()[:,0] = (mesh2.coordinates()[:,0]/2) + 0.5
36
37 V1 = FunctionSpace(mesh1, 'Lagrange', 1)
38 V2 = FunctionSpace(mesh2, 'Lagrange', 1)
39 VG = FunctionSpace(meshG, 'Lagrange', 1)
40
41 d2v1 = dof_to_vertex_map(V1)
42 d2v2 = dof_to_vertex_map(V2)
43 d2vG = dof_to_vertex_map(VG)
44
45 v2d1 = vertex_to_dof_map(V1)
46 v2d2 = vertex_to_dof_map(V2)
47 v2dG = vertex_to_dof_map(VG)
48
49 class Interface(SubDomain):
50     def inside(self, x, on_boundary):
51         tol = 1E-14   # tolerance for coordinate comparisons
52         return abs(0.5 - x[0]) < tol
53
54 class outer_boundary(SubDomain):
55     def inside(self, x, on_boundary):
56         tol = 1E-14
57         return on_boundary and \
58             (abs(x[0]) < tol or abs(1.0 - x[0]) < tol or abs(x[1]) < tol
    or abs(1.0 - x[1]) < tol)
59
```

```
60 def global_outer_boundary(x, on_boundary):
61     tol = 1E-14
62     return on_boundary and \
63         (abs(x[0]) < tol or abs(1.0 - x[0]) < tol or abs(x[1]) < tol or
    abs(1.0 - x[1]) < tol)
64
65 bcG = DirichletBC(VG, ufunc, global_outer_boundary)
66
67 sub_domains1 = MeshFunction("size_t", mesh1, mesh1.topology().dim() - 1)
68 sub_domains2 = MeshFunction("size_t", mesh2, mesh2.topology().dim() - 1)
69 sub_domains1.set_all(5)
70 sub_domains2.set_all(5)
71
72 interiors1 = MeshFunction("size_t", mesh1, mesh1.topology().dim())
73 interiors2 = MeshFunction("size_t", mesh2, mesh1.topology().dim())
74 interiorsG = MeshFunction("size_t", meshG, mesh1.topology().dim())
75 interiors1.set_all(0)
76 interiors2.set_all(0)
77 interiorsG.set_all(0)
78
79 interface1 = Interface()
80 interface2 = Interface()
81 outer = outer_boundary()
82
83 interface1.mark(sub_domains1, 2)
84 interface2.mark(sub_domains2, 1)
85
86 outer.mark(sub_domains1, 3)
```

```
87 outer.mark(sub_domains2, 3)
88
89 bc1 = DirichletBC(V1, ufunc, sub_domains1, 3)
90 bc2 = DirichletBC(V2, ufunc, sub_domains2, 3)
91
92 uG = TrialFunction(VG)
93 vG = TestFunction(VG)
94 dxG = Measure("dx")[interiorsG]
95 aG = - inner(grad(uG), grad(vG))*dxG(0, metadata={'quadrature_degree':
       qord})
96 LG = inner(f,vG)*dxG(0, metadata={'quadrature_degree': qord})
97
98 b = assemble(LG)
99 A = assemble(aG)
100 bcG.apply(b)
101 bcG.apply(A)
102
103 uG = Function(VG) # Compute solution - default is LU
104 solve(A, uG.vector(), b, 'lu')
105
106
107 uL = TrialFunction(V1)
108 uR = TrialFunction(V2)
109
110 vL = TestFunction(V1)
111 vR = TestFunction(V2)
112
113 B1 = np.zeros(nx)
```

```
114 B1[-1] = 1
115 B1 = ssp.kron(ssp.eye(ny), B1).todense()
116 B2 = np.zeros(nx)
117 B2[0] = 1
118 B2 = ssp.kron(ssp.eye(ny), B2).todense()
119
120 # Define Forms
121 ds1 = Measure("ds")[sub_domains1]
122 ds2 = Measure("ds")[sub_domains2]
123 dx1 = Measure("dx")[interiors1]
124 dx2 = Measure("dx")[interiors2]
125
126 a1 = - inner(grad(uL), grad(vL))*dx1(0, metadata={'quadrature_degree':
        qord}) - inner(alpha*uL,vL)*ds1(2, metadata={'quadrature_degree':
        qord})
127 a2 = - inner(grad(uR), grad(vR))*dx2(0, metadata={'quadrature_degree':
        qord}) - inner(alpha*uR,vR)*ds2(1, metadata={'quadrature_degree':
        qord})
128
129 A1 = assemble(a1); bc1.apply(A1);
130 A2 = assemble(a2); bc2.apply(A2)
131
132 g1 = Constant(0)
133 g2 = Constant(0)
134
135 G1 = assemble(inner(g1, vL)*dx1(2, metadata={'quadrature_degree': qord})
        )
136 G2 = assemble(inner(g2, vR)*dx2(1, metadata={'quadrature_degree': qord})
```

```
    )

L1 = assemble(inner(f,vL)*dx1(0, metadata={'quadrature_degree': qord}))
L2 = assemble(inner(f,vR)*dx2(0, metadata={'quadrature_degree': qord}))

maxiter = 100
niter = 0
tol = 1e-6
errors = []
err = 1

# Iterate

while err > tol and niter < maxiter:

    b1 = L1 - G1
    b2 = L2 - G2

    bc1.apply(b1)
    bc2.apply(b2)

    u1 = Function(V1)
    solve(A1, u1.vector(), b1, 'lu')
    u2 = Function(V2)
    solve(A2, u2.vector(), b2, 'lu')

    u1np = u1.vector().array()[v2d1].reshape(ny,nx)
    u2np = u2.vector().array()[v2d2].reshape(ny,nx)
```

```
164
165     uGnp = np.zeros((ny,ny))
166     uGnp[:,0:nx-1] = u1np[:,0:-1]
167     uGnp[:,nx:] = u2np[:,1:]
168     uGnp[:,nx-1] = 0.5*(u1np[:,-1] + u2np[:,0])
169     uDD = Function(VG)
170     uDD.vector()[:] = uGnp.reshape(ny*ny)[d2vG]
171
172     err = norm(uG.vector() - uDD.vector(), 'linf')/norm(uG.vector(), '
        linf')
173     errors.append([niter, err])
174
175     U1 = assemble(inner(u1,vL)*ds1(2, metadata={'quadrature_degree':
        qord}))
176     U2 = assemble(inner(u2,vR)*ds2(1, metadata={'quadrature_degree':
        qord}))
177
178     G1_old = G1[v2d1].copy()
179     G1.set_local(np.dot(B1.T, (2.0*alpha*np.dot(B2, U2[v2d2]) - np.dot(
        B2, G2[v2d2]))).T)[d2v1])
180     G2.set_local(np.dot(B2.T, (2.0*alpha*np.dot(B1, U1.array()[v2d1]) -
        np.dot(B1, G1_old)).T)[d2v2])
181
182     niter +=1
183
184 errors = np.array(errors)
```

Listing C.2: Main code for solving the 2D Poisson problem using the auxiliary variable optimized Schwarz method with my custom finite element assembly routines

Appendix D

Unstructured mesh creation through GMESH geometry files

```
1
2 // Define sizes and volume limits
3 coarse_size= 1000;
4 fine_size = 10;
5 xl = -2500;
6 xu = 2500;
7 yl = -2500;
8 yu = 2500;
9 zl = -2500;
10 zu = 2500;
11
12 // Points for base of volume
13 Point(1) = {xl,yl,zl,coarse_size};
14 Point(2) = {xu,yl,zl,coarse_size};
15 Point(3) = {xu,yu,zl,coarse_size};
```

```
16 Point(4) = {xl,yu,zl,coarse_size};
17
18 // Source refinement
19 Point(5) = {0,0,0,fine_size};
20
21 // Receiver refinement
22 point_num = 6;
23 recv_spacing = 10;
24 obsy = 0; // y offset for line of observations in x direction
25 n_recvs = 201; // odd for symmetry
26 For i In {1:n_recvs}
27
28   Point(point_num) = {0, recv_spacing*i - recv_spacing*(n_recvs + 1)/2,
       0, fine_size};
29   //Printf("point number is: %g, ", newp) ;
30   //Physical Point("Receiver") = newp;
31   point_num += 1;
32
33 EndFor
34 // Lines for base of volume
35 Line(1) = {1,2};
36 Line(2) = {2,3};
37 Line(3) = {3,4};
38 Line(4) = {4,1};
39
40 // Volume extrusion
41 Line Loop(5) = {4, 1, 2, 3};
42 Plane Surface(6) = {5};
```

```
43 Extrude {0, 0, 2*zu} {
44   Surface{6};
45 }
46
47 // Create boundary condition physicals
48 Physical Surface("Dirichlet") = {19, 15, 27, 6, 23, 28};
49 Physical Volume("wholespace") = {1}; // without this I get a 2D mesh
       when I save??
50
51 // Build refinement around source and receiver points
52 Field[1] = Attractor;
53 Field[1].NodesList = {5:6+n_recvs-1};
54 Field[2] = Threshold;
55 Field[2].IField = 1;
56 Field[2].LcMin = fine_size;
57 Field[2].LcMax = coarse_size;
58 Field[2].DistMin = 0.00001;
59 Field[2].DistMax = 2000;
60
61 // Use minimum of all the fields as the background field
62 Field[3] = Min;
63 Field[3].FieldsList = {2};
64 Background Field = 3;
65
66 // Optimize meshing
67 Mesh.Optimize = 1;
```

Listing D.1: gmesh geometry file specifying the wholespace model with refinement around the source and receiver locations

Appendix E

Preprocessing script HDF5.py

```
1
2 import numpy as np
3 from dolfin import *
4 import pickle
5 import os
6
7 # Set the parameters for the problem
8 p = {'filename' : "wholespace_mesh_437610e",
9       'omega' : 2*pi*300.0,
10      'mu' : (4.*pi) * 1e-07,
11      'sigma' : 0.01,
12      'I' : 1.,
13      'S' : 1.,
14      'xs' : [0., 0., 0.],
15      'yobs' : 100}
16
17 os.system('dolfin-convert ' + p['filename'] + '.msh ' + p['filename'] +
```

```
    '.xml')
# Read in the mesh, physical properties, and boundary data
mesh = Mesh(p['filename'] + ".xml")
boundaries = MeshFunction("size_t", mesh, p['filename'] + "_facet_region
    .xml")
earth = MeshFunction("size_t", mesh, p['filename'] + "_physical_region.
    xml")

# Set up the point source

# Locate the source cell
source_pt = Point(0,0,0)
bbt = BoundingBoxTree()
bbt.build(mesh)
source_cell = bbt.compute_entity_collisions(source_pt)[0]

# Create source function
source_func = MeshFunction("size_t", mesh, 3)
source_func.set_all(0)
source_func.array()[source_cell] = 1

# Add a source_cell_volume to the parameter dictionary
p['source_cell_volume'] = Cell(mesh, source_cell).volume()

# Find the limits of the mesh
p['xlim'] = (min(mesh.coordinates()[:,0]), max(mesh.coordinates()[:,0]))
p['ylim'] = (min(mesh.coordinates()[:,1]), max(mesh.coordinates()[:,1]))
p['zlim'] = (min(mesh.coordinates()[:,2]), max(mesh.coordinates()[:,2]))
```

```
43
44 # Pickle the parameters and save
45 pickle.dump( p, open( "save.p", "wb" ) )
46
47 # Write the mesh and mesh functions to an hdf5 file for parallel import
48 hdf = HDF5File(mesh.mpi_comm(), p['filename'] + ".h5", "w")
49 hdf.write(mesh, "/mesh")
50 hdf.write(earth, "/subdomains")
51 hdf.write(boundaries, "/boundaries")
52 hdf.write(source_func, "/source")
53 hdf.close()
```

Listing E.1: Serial preprocessing script for subsequent parallel modeling code