Matrix Equation Solvers

Alex Harvey - mm13ah - ID: 200786528

1 Introduction

The traditional approach to solving PDEs numerically involves stacking all unknowns of the problem into a single vector which ignores underlying structures. This prevents methods from being used which can take advantage of the problem structure to solve the problem more efficiently. This can come at a significant cost when uncertainty is introduced into the equation. An alternative approach is to formulate the problem as a matrix equation, which can be solved using a range of different methods. This project involves exploring how this alternative formulation can be solved using matrix solvers, and how these solvers compare against each other. As an example, let $u : \Omega \to \mathbb{R}$. Then let the equation:

$$-u_{xx} - u_{yy} = f \tag{1.1}$$

be defined on $\Omega = (0,1) \times (0,1)$, with boundary conditions $u(x,y) = 0$, as shown in Figure 1.

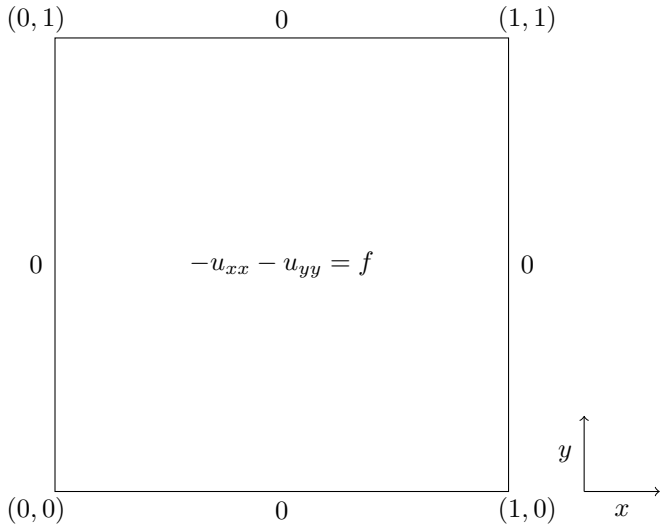

Figure 1: Domain for $-u_{xx} - u_{yy} = f$.

The domain of this PDE can be discretised into a mesh with uniform spacing h using the centred finite difference approximations:

$$u_{xx} \approx \frac{u_{i-1j} - 2u_{ij} + u_{i+1j}}{h^2} \tag{1.2}$$

$$u_{yy} \approx \frac{u_{ij-1} - 2u_{ij} + u_{ij+1}}{h^2} \tag{1.3}$$

where $u_{ij} = u(x_i, y_j)$. The mesh is shown in Figure 2.

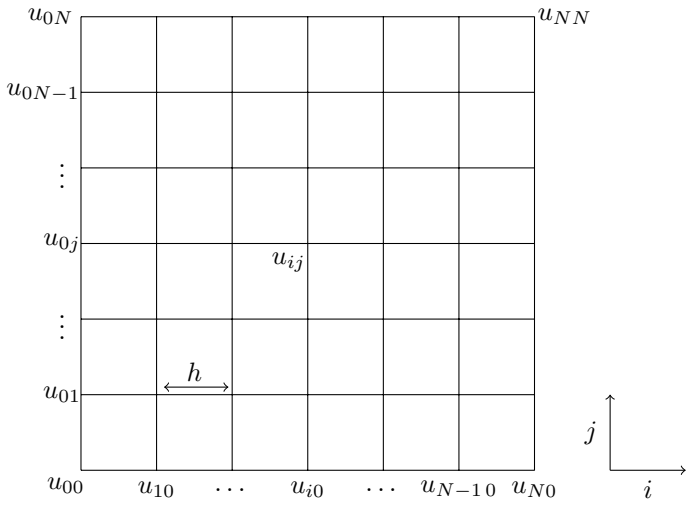

Figure 2: Discretised domain for $-u_{xx} - u_{yy} = f$.

The discretised form of this PDE can then be solved by computing the equation:

$$f_{ij} = -\frac{1}{h^2}\big(u_{i-1j} - 2u_{ij} + u_{i+1j}\big) - \frac{1}{h^2}\big(u_{ij-1} - 2u_{ij} + u_{ij+1}\big) \tag{1.4}$$

at each internal grid point, meaning the system has n^2 unknowns with $n = N-2$. The traditional approach to solving this discretised form would be to write (1.4) as:

$$f_{ij} = -\frac{1}{h^2}\big(u_{i-1j} + u_{ij-1} - 4u_{ij} + u_{i+1j} + u_{ij+1}\big) \tag{1.5}$$

and then stack all unknowns u_{ij} into a single vector U, resulting in the linear system $AU = F$. As stated, this ignores the underlying structure of the problem.

By keeping (1.4) in its original form we can instead write the system as a matrix equation:

$$TU + UT = F \tag{1.6}$$

with $T = -\frac{1}{h^2}\,\text{tridiag}(1, -2, 1)$ and $U_{ij} = u(x_i, y_j)$ where (x_i, y_j) are interior grid nodes for $i, j = 1, \ldots, n$. This equation is in the form of a Sylvester equation $AX + XB = C$, with $A = B = T$, $X = U$ and $C = F$, and there many different methods that can be used to solve equations of this type. This project will explore different methods for solving equations in this form.

2 Scope and Schedule

2.1 Aim

The aim of this project is to first study, implement and compare a range of matrix equation solvers. Following this, a specific problem will be derived with the help of my supervisor so that these solvers may be used and compared for a suitable application.

2.2 Objectives

The objectives of this project are as follows:

- To carry out an extensive, in-depth literature review on methods (both direct and iterative) for solving matrix equations from a wide range of sources. To decide which of these methods are appropriate to implement and to gain a solid understanding of how they work.
- To use and expand upon my programming experience to implement the chosen methods for solving matrix equations to solve the specified problem.
- To evaluate the implementation by comparing and contrasting the methods implemented to try to decide which is the best method for solving the given problem.
- To derive a suitable application equation so that the methods studied in this project can be applied to a specific problem.
- To clearly present the work carried out during the project by using and building upon my report writing skills.

2.3 Deliverables

The deliverables of this project include:

- The final report that will include the details of the matrix solvers that have been studied, how the solvers were implemented, an evaluation and comparison of the implemented solvers, an analysis of how these solvers were used to solve the chosen application problem, and finally an evaluation of the success of the project.
- Code that successfully implements the chosen matrix solvers so that they solve the given problem.

2.4 Methodology

The methodology of this project will first involve studying academic publishings to gain an understanding of various methods for solving matrix equations. Python will be used as the programming language of choice for the implementation because of my familiarity with it, the extensive amount of documentation available for it and the excellent libraries it has available (e.g. NumPy and SciPy). GitHub will be used for version control and the final report will be written using LaTeX.

2.5 Tasks, milestones and timeline

The steps of this project will be divided into iterations, with the problem in each iteration becoming successively more complex and difficult to solve. This is because understanding is a key part of this project, and so each iteration will build on the understanding of the last. Each iteration will consist of studying and applying matrix methods to the problem, implementing them in Python to solve the problem, evaluating the results and write up. Also rough deadlines will be given for when each iteration should be completed by, to ensure the project is on track at any given stage.

The iterations are as follows:

- Introductory problem: $-u_{xx} - u_{yy} = 2\pi^2 \sin(\pi x) \sin(\pi y)$ - deadline June 1st
- Problem introducing uncertainty: $-\varepsilon u_{xx} - \varepsilon u_{yy} = 2\pi^2 \sin(\pi x) \sin(\pi y)$ - deadline June 22nd
- A Poisson equation on a surface defined by a height map (not yet derived) - deadline July 13th
- Application reaction-diffusion equation (not yet derived) - deadline August 3rd

If the project deadlines are met the remaining time will be dedicated to project evaluation, write up and any possible project extensions.

3 Matrix Solvers

As an example problem, let the exact solution, u, of (1.1) be defined as:

$$u = \sin(\pi x)\sin(\pi y) \tag{3.1}$$

which gives:

$$u_{xx} = u_{yy} = -\pi^2 \sin(\pi x)\sin(\pi y) \tag{3.2}$$

$$\implies f = 2\pi^2 \sin(\pi x)\sin(\pi y) \tag{3.3}$$

The PDE to be solved is now:

$$-u_{xx} - u_{yy} = 2\pi^2 \sin(\pi x)\sin(\pi y) \tag{3.4}$$

at each grid point, or equivalently:

$$TU + UT = F \tag{3.5}$$

as a matrix equation. Here $T = -\frac{1}{h^2}\,\text{tridiag}(1, -2, 1)$, $U_{ij} = u(x_i, y_j)$ and $F_{ij} = 2\pi^2 \sin(\pi x_i)\sin(\pi y_j)$, where (x_i, y_j) are interior grid nodes for $i, j = 1, \ldots, n$. As stated previously this equation is in the form of a Sylvester equation and can be solved using a range of different methods. A graph of the exact solution, with $n = 1000$, is given below:

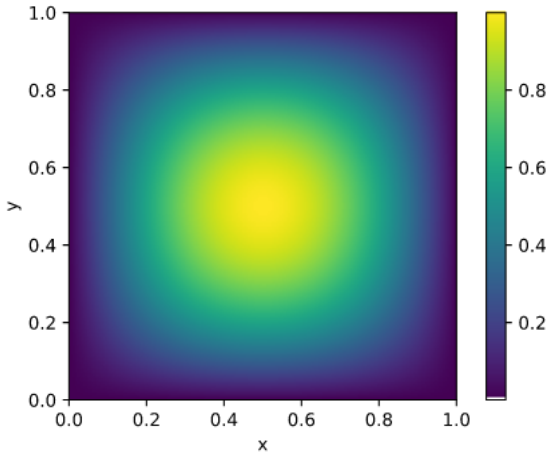

Figure 3: Plot of the solution with $n = 1000$.

Throughout the following section, the following measurements are given to evaluate the performance of each of the methods implemented:

- n: The total number of unknowns in each direction for the system. The total number of unknowns is n^2.

- Time(s): The time taken in seconds for the method to compute the solution to the problem.
- $||u - u_h||_{L^\infty}$: Measures the maximum difference between the actual solution and computed solution for each u. Defined as:

$$||u - u_h||_{L^\infty} = \max_{ij}|u(x_i, y_i) - u_{ij}|$$

- $||u - u_h||_{L^2}$: A measure of error that takes into account the difference between all actual and computed solutions, as well as the step size. Defined as:

$$||u - u_h||_{L^2} = \sqrt{h^2 \sum |u(x_i, y_j) - u_{ij}|^2}$$

- Experimental order of convergence (eoc): Measures the rate of convergence of a method as the problem size is increased, which should approach 2 as the step size is increased. Defined as:

$$\text{eoc}(i) = \frac{\log(E_i/E_{i-1})}{\log(h_i/h_{i-1})}$$

  where E_i is the error and h_i is the mesh size at level i.

- Empirical order of growth (eog): Measures the order of growth of the execution time of an algorithm as the problem size is increased. Defined as:

$$\text{eog}(i) = \frac{\log(t_i/t_{i-1})}{\log(n_i/n_{i-1})}$$

  where t_i is the total execution time and n_i is the problem size at level i.

3.1 Direct Methods

3.1.1 Kronecker Product

A naive approach to solving this system is to use the Kronecker product to rewrite (3.5) as a standard vector linear system. The Sylvester equation $AX + XB = C$ can be written as the standard vector linear system:

$$\mathcal{A}x = c \tag{3.6}$$

with $\mathcal{A} = I \otimes A + B^* \otimes I$, where I is the identity matrix, B^* denotes the conjugate transpose of B, $x = \text{vec}(X)$ and $c = \text{vec}(C)$.[1]

For the system in (3.6), we have $A = B = T$, $X = U$, $C = F$ and $T = T^*$, so the standard linear system is:

$$\mathcal{T}u = \mathcal{F} \tag{3.7}$$

[1]The vec operator reshapes a matrix into a vector by stacking the columns one after another.

where $\mathcal{F} = \text{vec}(F)$, $\mathcal{T} = I \otimes T + T \otimes I$ and $u = \text{vec}(U)$.

This is the exact linear system that would be obtained from equation (1.5), i.e. stacking all unknowns u_{ij} into a single vector in the first place. Since the matrix $\mathcal{T}$ is sparse, this equation can be solved using a standard direct sparse solver. This approach provides a good base case for comparison. Results solving this linear system using the direct sparse solver `sparse.linalg.spsolve` from the SciPy library are shown in Figure 4.

n	Time(s)	$\|\|u - u_h\|\|_{L^\infty}$	$\|\|u - u_h\|\|_{L^2}$	eoc	eog
125					

Figure 4: Results obtained from solving the linear system $\mathcal{A}x = c$ using the direct solver `sparse.linalg.spsolve` from the SciPy library.

3.1.2 Bartels-Stewart Algorithm

The Bartels-Stewart algorithm [1] can be used to solve the Sylvester equation $AX + XB = C$. In the general case the algorithm is as follows:

1. Compute the Schur forms $A^* = PRP^*$ and $B = QSQ^*$
2. Solve $R^*V + VS = P^*CQ$ for V
3. Compute $X = PVQ^*$

where A^* denotes the conjugate transpose of A.

In this case $A = B = T$, $T = T^*$, $X = U$ and $C = F$ so the algorithm is as follows:

1. Compute the Schur form $T = PRP^*$
2. Solve $R^*V + VR = P^*FP$ for V
3. Compute $U = PVP^*$

Results using this algorithm are shown in Figure 5.

n	Time(s)	$\|\|u - u_h\|\|_{L^\infty}$	$\|\|u - u_h\|\|_{L^2}$	eoc	eog
125	1.2486	5.1807×10^{-5}	2.5904×10^{-5}	-	-
250	9.6712	1.3054×10^{-5}	6.5275×10^{-6}	1.9887	2.9124
500	77.888	3.2767×10^{-6}	1.6384×10^{-6}	1.9942	3.0096
1000	640.22	8.2088×10^{-7}	4.1044×10^{-7}	1.9970	3.0391
2000	5093.1	2.0561×10^{-7}	1.0276×10^{-7}	1.9973	2.9919

Figure 5: Results using the Bartels-Stewart algorithm.

The SciPy library has a built in solver for solving Sylvester equations, `linalg.solve_sylvester`, which uses the Bartels-Stewart algorithm. Results using this solver are given in Figure 6.

n	Time(s)	$\|\|u-u_h\|\|_{L^\infty}$	$\|\|u-u_h\|\|_{L^2}$	eoc	eog
125	0.059959	5.1807×10^{-5}	2.5904×10^{-5}	-	-
250	0.11421	1.3054×10^{-5}	6.5275×10^{-6}	-	-
500	1.9383	3.2767×10^{-6}	1.6384×10^{-6}	-	-
1000	4.0920	8.2084×10^{-7}	4.1042×10^{-7}	-	-
2000	32.029	2.0503×10^{-7}	1.0259×10^{-7}	-	-
4000	279.95	4.9949×10^{-8}	2.4876×10^{-8}	-	-

Figure 6: Results using SciPy's `linalg.solve_sylvester`.

As can be seen from the results above, using the built-in SciPy solver results in a significant speed-up in time as n is increased. This is likely because it makes use of of LAPACK, which is an optimised software library for solving linear algebra problems.

3.1.3 Similarity Transformation

A similarity transformation [3] can be used to solve the Sylvester equation $AX + XB = C$, where A is a $n \times n$ matrix and B is a $m \times m$ matrix.

Assuming matrices A and B can be diagonalised, let $P^{-1}AP = \text{diag}(\lambda_1, \ldots, \lambda_n)$ and $Q^{-1}BQ = \text{diag}(\mu_1, \ldots, \mu_m)$, where $\lambda_1, \ldots, \lambda_n$ are the eigenvalues of A and $\mu_1, \ldots, \mu_m$ are the eigenvalues values of B. Let $\tilde{C} = P^{-1}CQ$. The solution is then:

$$X = P\tilde{X}Q^{-1}, \text{ with } \tilde{x}_{ij} = \frac{\tilde{c}_{ij}}{\lambda_i + \mu_j}$$

In this case, $A = B = T$, $X = U$ and $C = F$ so $P = Q$ and $P^{-1}TP = \text{diag}(\lambda_1, \ldots, \lambda_n)$, where $\lambda_1, \ldots, \lambda_n$ are the eigenvalues of T. Letting $\tilde{F} = P^{-1}FP$, the solution is:

$$U = P\tilde{U}P^{-1}, \text{ with } \tilde{u}_{ij} = \frac{\tilde{f}_{ij}}{\lambda_i + \lambda_j}$$

Using numpy.linalg.eig

Using the method above, the eigenvalues and eigenvectors can be computed using NumPy's `linalg.eig` function. The results doing this are given in Figure 7.

n	Time(s)	$\|\|u-u_h\|\|_{L^\infty}$	$\|\|u-u_h\|\|_{L^2}$	eoc	eog
125	0.034534	5.1802×10^{-5}	2.5904×10^{-5}	-	-
250	0.13690	1.3054×10^{-5}	6.5275×10^{-6}	-	-
500	0.59718	3.2767×10^{-6}	1.6384×10^{-6}	-	-
1000	2.2211	8.2086×10^{-7}	4.1043×10^{-7}	-	-
2000	11.571	2.0467×10^{-7}	1.0237×10^{-7}	-	-
4000	70.984	4.9966×10^{-8}	2.4876×10^{-8}	-	-
8000	466.83	8.6360×10^{-9}	4.1223×10^{-9}	-	-

Figure 7: Results using a similarity transformation, calculating the eigenvalues and eigenvectors using `numpy.linalg.eig`.

Here the experimental order of convergence moves away from 2 as n is increased. This is likely due to the fact that there is no general formula for calculating eigenvalues and eigenvectors for an arbitrary matrix, meaning the eigenvalues and eigenvectors are most likely approximated by NumPy's `linalg.eig`.

This method can be split into component parts and each part can be timed, to see which part is the most costly. The steps of the method are:

1. Calculate eigenvalues and eigenvectors of T
2. Diagonalise T (i.e. calculate P and P^{-1})
3. Calculate $\tilde{F} = P^{-1}FP$
4. Calculate $\tilde{U}$, where $\tilde{u}_{ij} = \frac{\tilde{f}_{ij}}{\lambda_i + \lambda_j}$
5. Calculate solution $U = P\tilde{U}P^{-1}$

The results of doing so are shown in Figure 8.

n	1	2	3	4	5	Total
125	0.013860	0.00042105	0.00075817	0.018881	0.00061440	0.034534
250	0.052402	0.00068998	0.0026329	0.078654	0.0025253	0.13690
500	0.25900	0.0020170	0.013870	0.30815	0.014147	0.59718
1000	0.91819	0.0011129	0.084650	1.1328	0.084323	2.2211
2000	5.8236	0.0035102	0.61534	4.5150	0.61360	11.571
4000	43.651	0.0074658	4.6290	18.071	4.6257	70.984
8000	325.30	0.027416	35.464	70.695	35.345	466.83

Figure 8: Timing results for each step of the similarity transformation method, calculating the eigenvalues and eigenvectors using `numpy.linalg.eig`.

Calculating eigenvalues and eigenvectors explicitly

As can be seen from the results above, the most costly part of this method is calculating the eigenvalues and eigenvectors. As T is a matrix in Toeplitz form,

the eigenvalues and eigenvectors can be calculated directly as:

$$\lambda_i = \frac{2}{h^2}\Big(\cos\Big(\frac{i\pi}{n+1}\Big) - 1\Big)$$

and:

$$t_{ij} = \sqrt{\frac{2}{n+1}}\sin\Big(\frac{ij\pi}{n+1}\Big)$$

Results using this method for calculating the eigenvalues are eigenvectors are given in Figure 9, and timings for each step are given in Figure 10.

n	Time(s)	$\|\|u-u_h\|\|_{L^\infty}$	$\|\|u-u_h\|\|_{L^2}$	eoc	eog
125	0.080673	5.1807×10^{-5}	2.5904×10^{-5}	-	-
250	0.30396	1.3054×10^{-5}	6.5275×10^{-6}	-	-
500	1.2021	3.2767×10^{-6}	1.6384×10^{-6}	-	-
1000	5.5904	8.2082×10^{-7}	4.1041×10^{-7}	-	-
2000	20.247	2.0541×10^{-7}	1.0270×10^{-7}	-	-
4000	82.243	5.1314×10^{-8}	2.5656×10^{-8}	-	-
8000	360.35	1.2814×10^{-8}	6.4065×10^{-9}	-	-

Figure 9: Results using a similarity transformation, calculating the eigenvalues and eigenvectors explicity.

n	1	2	3	4	5	Total
125	0.057252	0.00025916	0.0045545	0.018010	0.00059748	0.080673
250	0.21954	0.00047016	0.0081279	0.073295	0.0025220	0.30396
500	0.88125	0.0012608	0.018555	0.28724	0.013819	1.2021
1000	3.7073	0.00096869	0.67056	1.1304	0.081200	5.5904
2000	13.812	0.0092294	1.3690	4.4495	0.60793	20.247
4000	54.907	0.0068946	4.9050	17.381	4.5936	82.243
8000	217.26	0.026015	35.357	71.581	36.129	360.35

Figure 10: Timing results for each step of the similarity transformation method, calculating the eigenvalues and eigenvectors explicitly.

As can be seen from the results above, calculating the eigenvalues and eigenvectors explicitly vastly outperforms calculating them using the NumPy library when n is large.

3.1.4 Shifted System

A projection method [3] can be used on a Sylvester equation $AX + XB = C$ by decomposing B to obtain n linear systems, each of which can be solved

simultaneously (and in parallel) to obtain the solution X. The steps of this method are as follows:

1. Compute $B = WSW^{-1}$ with $S = \text{diag}(s_1, \ldots, s_n)$
2. Let $\hat{C} = CW$
3. For $i = 1$ to n, solve the system $(A + s_i I)(\hat{X})_i = (\hat{C})_i$
4. Compute solution $X = \hat{X}W$

where $(\hat{X})_i$ denotes the i^{th} column of $\hat{X}$.

In the case of $TU + UT = F$, the steps are as follows:

1. Compute $T = WSW^{-1}$ with $S = \text{diag}(s_1, \ldots, s_n)$
2. Let $\hat{F} = FW$
3. For $i = 1$ to n, solve the system $(T + s_i I)(\hat{U})_i = (\hat{F})_i$
4. Compute solution $U = \hat{U}W$

Results using this method are given in Figure 11.

n	Time(s)	$\|\|u - u_h\|\|_{L^\infty}$	$\|\|u - u_h\|\|_{L^2}$	eoc	eog
125	0.092085	5.1807×10^{-5}	2.5904×10^{-5}	-	-
250	0.43609	1.3054×10^{-5}	6.5274×10^{-6}	-	-
500	2.1851	3.2767×10^{-6}	1.6384×10^{-6}	-	-
1000	19.498	8.2082×10^{-7}	4.1041×10^{-7}	-	-
2000	194.13	2.0541×10^{-7}	1.0271×10^{-7}	-	-
4000	2452.5	5.1347×10^{-8}	2.5674×10^{-8}	-	-

Figure 11: Results using a similarity transformation, calculating the eigenvalues and eigenvectors explicity.

3.2 Iterative Methods

3.2.1 Kronecker Product

Similarly to Section 3.1.1, the Kronecker product can be used to write the matrix equation as a standard vector linear system. A a standard iterative solver can then be used to solve the system, which can provide a base case for comparison. Results using `scipy.sparse.linalg.cg`, which is a sparse solver that uses the conjugate gradient iterative method, are given in Figure 12, using a convergence tolerance of 10^{-9}.

n	Time(s)	$\|\|u-u_h\|\|_{L^\infty}$	$\|\|u-u_h\|\|_{L^2}$	eoc	eog
10	0.0023940	0.0066868	0.0034124	-	-
100	0.0048580	8.0611×10^{-5}	4.0315×10^{-5}	-	-
1000	0.61869	8.2082×10^{-7}	4.1041×10^{-7}	-	-
2000	2.1425	2.0541×10^{-7}	1.0271×10^{-7}	-	-
4000	23.065	5.1378×10^{-8}	2.5689×10^{-8}	-	-
8000	160.09	1.2848×10^{-8}	6.4239×10^{-9}	-	-

Figure 12: Results obtained from solving the linear system $\mathcal{A}x = c$ using the iterative solver `sparse.linalg.cg` from the SciPy library.

3.2.2 Gradient Based Method

In [4] a gradient based method for solving Sylvester equations is given. The equation $TU + UT = F$ can be written as two recursive sequences:

$$U_k^{(1)} = U_{k-1}^{(1)} + \kappa T(F - TU_{k-1}^{(1)} - U_{k-1}^{(1)}T) \tag{3.8}$$

$$U_k^{(2)} = U_{k-1}^{(2)} + \kappa(F - TU_{k-1}^{(2)} - U_{k-1}^{(2)}T)T \tag{3.9}$$

where κ represents the relative step size. The approximate solution U_k is taken as the average of these two sequences:

$$U_k = \frac{U_k^{(1)} + U_k^{(2)}}{2} \tag{3.10}$$

This solution only converges if:

$$0 < \kappa < \frac{1}{\lambda_{\max}(T^2)} \tag{3.11}$$

where $\lambda_{\max}(T^2)$ denotes the maximum eigenvalue of T^2. Using the method given previously for calculating eigenvalues we can compute:

$$\lambda_{\max}(T^2) = \frac{4}{h^4}\max\Big(\big(\cos\Big(\frac{i\pi}{n+1}\Big) - 1\big)^2\Big) \tag{3.12}$$

$\lambda_{\max}(T^2)$ therefore scales with $\frac{1}{h^4}$ meaning its reciprocal scales with h^4, implying κ will need to be significantly small as n is increased for the solution to converge. Even for small values of n, this is impractical and therefore this method is not appropriate for solving this equation.

3.2.3 Modified Conjugate Gradient

In [2] a modified conjugate gradient (MCG) algorithm is proposed, which is adapted for solving Sylvester Equations. The algorithm is as follows:

1. Choose initial guess $X^{(0)}$, let $k = 0$ and calculate:
   - $R^{(0)} = F - AX^{(0)} - X^{(0)}B$
   - $Q^{(0)} = A^T R^{(0)} + R^{(0)} + R^{(0)} B^T$
   - $Z^{(0)} = Q^{(0)}$
2. If $R^{(k)} = 0$, or $R^{(k)} \neq 0$ but $Z^{(k)} = 0$, stop. Else, calculate:

$$\gamma_k = \frac{[R^{(k)}, R^{(k)}]}{[Z^{(k)}, Z^{(k)}]}$$

   where $[R, R] = \mathrm{tr}(R^T R)$ is the trace of matrix $R^T R$.
3. Calculate:
   - $R^{(k+1)} = F - AX^{(k+1)} - X^{(k+1)}B$
   - $Q^{(k+1)} = A^T R^{(k+1)} + R^{(k+1)} + R^{(k+1)} B^T$

   and return to Step 2.

4 Uncertainty

Uncertainty can be introduced into the PDE (1.1) by introducing a random diffusion coefficient $\varepsilon = \varepsilon(\omega)$, $\varepsilon \neq 0$ such that:

$$-\varepsilon u_{xx} - \varepsilon u_{yy} = f \tag{4.1}$$

Redefining u as:

$$u(x, y, \varepsilon) = \sin(\pi x)\sin(\pi y) + \varepsilon \sin(3\pi x)\sin(5\pi y) \tag{4.2}$$

gives:

$$u_{xx} = -\pi^2 \sin(\pi x)\sin(\pi y) - 9\pi^2 \varepsilon \sin(3\pi x)\sin(5\pi y) \tag{4.3}$$

$$u_{yy} = -\pi^2 \sin(\pi x)\sin(\pi y) - 25\pi^2 \varepsilon \sin(3\pi x)\sin(5\pi y) \tag{4.4}$$

$$\implies f = 2\pi^2 \varepsilon \sin(\pi x)\sin(\pi y) + 34\pi^2 \varepsilon^2 \sin(3\pi x)\sin(5\pi y) \tag{4.5}$$

The PDE to be solved is now:

$$-\varepsilon u_{xx} - \varepsilon u_{yy} = 2\pi^2 \varepsilon \sin(\pi x)\sin(\pi y) + 34\pi^2 \varepsilon^2 \sin(3\pi x)\sin(5\pi y) \tag{4.6}$$

As before this can be formed as a matrix equation:

$$TU + UT = F \tag{4.7}$$

with $T = -\frac{\varepsilon}{h^2}\text{tridiag}(1, -2, 1)$, $U = u(x_i, y_j)$ and $F = 2\pi^2 \varepsilon \sin(\pi x_i)\sin(\pi y_j) + 34\pi^2 \varepsilon^2 \sin(3\pi x_i)\sin(5\pi y_j)$. This equation is significantly harder to solve accurately and efficiently than the equation in the previous section due to the fact that ε is an unknown coefficient.

4.1 Monte Carlo

A simple way of solving (4.7) is to use the Monte Carlo method. The Monte Carlo method works by taking M random samples of ε over its range, and then solves the PDE for each of these samples. This allows the most effective method from Section 3 to be used as a 'black box' strategy. Once solved, the mean of all computed solutions can be calculated. If the probability distribution $\rho(\omega)$ for ε is known, then the expectation of u can be computed as:

$$\mathbb{E}(u) = \int u(x, y, \varepsilon)\rho(\omega)d\omega \tag{4.8}$$

The error can be computed by comparing the mean solution and the expectation. The error is defined as:

$$\text{Error} = \left|\left|\mathbb{E}(u) - \frac{1}{M}\sum_{i=1}^{M} u_h(x, y, \varepsilon)\right|\right| \tag{4.9}$$

where M is the number of samples used in the Monte Carlo method and u_h is the solution obtained from the finite difference method using a mesh of size h.

Results are given in Figure

References

[1] R. H. Bartels and G. W. Stewart. Solution of the Matrix Equation AX + XB = C. *Commun. ACM*, 15(9):820–826, Sept. 1972.

[2] J. Hou, Q. Lv, and M. Xiao. A Parallel Preconditioned Modified Conjugate Gradient Method for Large Sylvester Matrix Equation. 2014:1–7, 03 2014.

[3] V. Simoncini. Computational Methods for Linear Matrix Equations. 58:377–441, 01 2016.

[4] J. Zhou, W. Ruirui, and Q. Niu. A Preconditioned Iteration Method for Solving Sylvester Equations. 2012, 07 2012.