Matrix Equation Solvers

Alex Harvey - mm13ah - ID: 200786528

Contents

1 Introduction

1.1 Problem Statement

The traditional approach to solving partial differential equations (PDEs) numerically involves stacking all the 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 problem. 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 methods that solve this alternative formulation and how these solvers compare against each other.

As an example, define spatial domain $\mathcal{D} = \{(x, y) : 0 \leq x \leq 1,\ 0 \leq y \leq 1\}$ and let $u : \mathcal{D} \to \mathbb{R}$ be the solution of the equation:

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

with boundary conditions $u(x, y) = 0$, as shown in Figure 1.

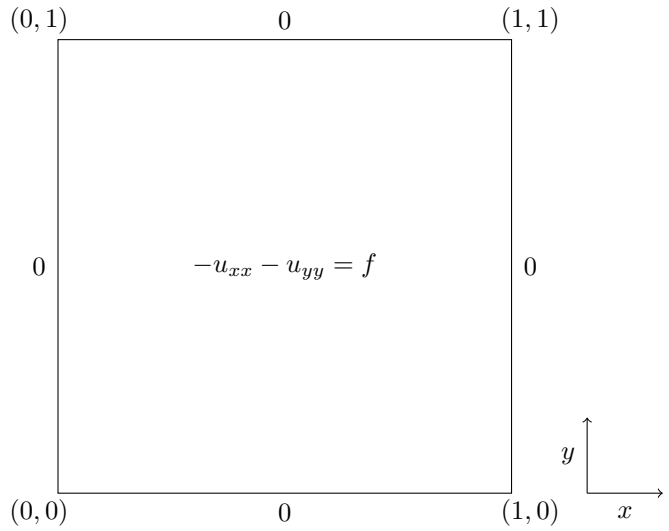

Figure 1: Domain for $-u_{xx} - u_{yy} = f$.

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.

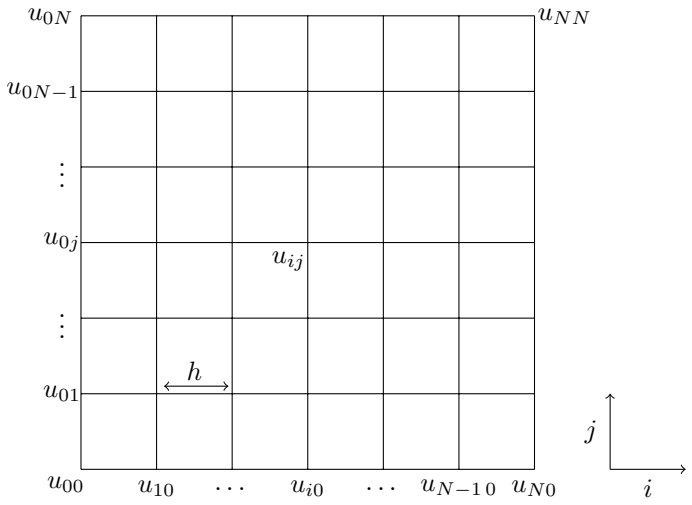

Figure 2: Discretised domain for $-u_{xx} - u_{yy} = f$.

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 previously, this ignores the underlying structure of the problem.

1.2 Sylvester Equation

A Sylvester equation is a matrix equation of the form $AX + XB = C$, where A is a $n \times n$ matrix, B is a $m \times m$ matrix, and X and C are $n \times m$ matrices. We can write (1.4) as a Sylvester equation in the form:

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

where T, U and F are of size $n \times n$. Here $T = -\frac{1}{h^2}\,\text{tridiag}(1, -2, 1)$ and $U_{ij} = u(x_i, y_j)$, where (x_i, y_j) are interior grid nodes for $i, j = 1, \ldots, n$. The system has n unknowns in each direction meaning there is a total of n^2 unknowns. The matrix equation is visualised below:

$$\frac{-1}{h^2}\begin{pmatrix} -2 & 1 & 0 & \dots & 0 & 0 \\ 1 & -2 & 1 & \dots & 0 & 0 \\ 0 & 1 & -2 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & -2 & 1 \\ 0 & 0 & 0 & \dots & 1 & -2 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & \dots & u_{1j} & \dots & u_{1n} \\ u_{21} & u_{22} & \dots & u_{2j} & \dots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots & \dots & \vdots \\ u_{i1} & u_{i2} & \dots & u_{ij} & \dots & u_{in} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ u_{n1} & u_{n2} & \dots & u_{nj} & \dots & u_{nn} \end{pmatrix}$$

$$+ \begin{pmatrix} u_{11} & u_{12} & \dots & u_{1j} & \dots & u_{1n} \\ u_{21} & u_{22} & \dots & u_{2j} & \dots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots & \dots & \vdots \\ u_{i1} & u_{i2} & \dots & u_{ij} & \dots & u_{in} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ u_{n1} & u_{n2} & \dots & u_{nj} & \dots & u_{nn} \end{pmatrix} \frac{-1}{h^2} \begin{pmatrix} -2 & 1 & 0 & \dots & 0 & 0 \\ 1 & -2 & 1 & \dots & 0 & 0 \\ 0 & 1 & -2 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & -2 & 1 \\ 0 & 0 & 0 & \dots & 1 & -2 \end{pmatrix} = F$$

There are a variety of methods that can be used to solve Sylvester equations. This project will explore and compare different methods for solving equations in this form.

1.3 Notation and Preliminaries

1.4 Organisation

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

For spatial domain $\mathcal{D} = \{(x, y) : 0 \leq x \leq 1,\ 0 \leq y \leq 1\}$ and $u : \mathcal{D} \to \mathbb{R}$, let $u(x, y) = \sin(\pi x)\sin(\pi y)$ be the exact solution of the equation:

$$\begin{aligned} -u_{xx} - u_{yy} &= f \quad \text{on} \quad \mathcal{D} \\ u &= 0 \quad \text{on} \quad \partial\mathcal{D} \end{aligned} \tag{3.1}$$

This gives:

$$\begin{aligned} u_{xx} = u_{yy} &= -\pi^2 \sin(\pi x)\sin(\pi y) \\ \implies f &= 2\pi^2 \sin(\pi x)\sin(\pi y) \end{aligned} \tag{3.2}$$

The equation to be be solved is therefore:

$$\begin{aligned} -u_{xx} - u_{yy} &= 2\pi^2 \sin(\pi x)\sin(\pi y) \quad \text{on} \quad \mathcal{D} \\ u &= 0 \quad \text{on} \quad \partial\mathcal{D} \end{aligned} \tag{3.3}$$

Using the centred finite difference approximations ((1.2) and (1.3)) with uniform grid spacing h to discretise the domain of this system, we have:

$$\begin{aligned} -\tfrac{1}{h^2}\left(u_{i-1j} - 2u_{ij} + u_{i+1j}\right) - \tfrac{1}{h^2}\left(u_{ij-1} - 2u_{ij} + u_{ij+1}\right) \\ = 2\pi^2 \sin(\pi x_i)\sin(\pi y_j) \end{aligned} \tag{3.4}$$

to be solved at each grid point for $i, j = 1, \ldots n$, where n is the total number of unknowns in each direction. The matrix form of this equation is therefore:

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

where $T = -\frac{1}{h^2}\,\text{tridiag}(1, -2, 1)$, $U_{ij} = u(x_i, y_j)$ and $F_{ij} = 2\pi^2 \sin(\pi x_i)\sin(\pi y_j)$ are all matrices of size $n \times n$ and (x_i, y_j) are interior grid nodes for $i, j = 1, \ldots, n$.

This form allows us to explore different methods for solving this equation and compare them to the exact solution $u = \sin(\pi x)\sin(\pi y)$. A plot of the exact solution, with $n = 1000$, is given in Figure 3.

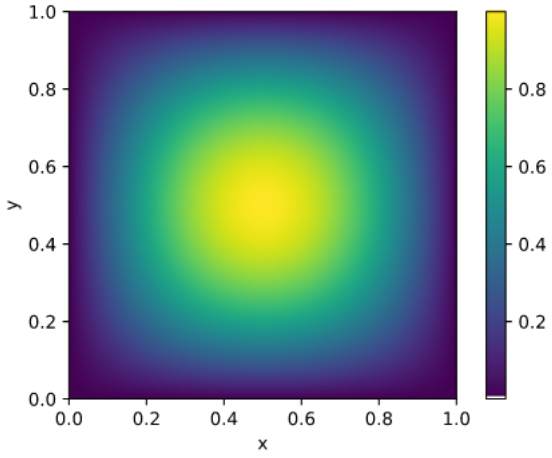

Figure 3: Plot of the solution with $n = 1000$.

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

- n: The 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 (that is, the approximate time complexity). 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.

It is worth noting that as the total number of unknowns and therefore the problem size is n^2, the best time complexity that an optimal solver can achieve is $O(n^2)$, as it must compute a solution for each unknown.

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:

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

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 Table 1. As can be seen from the results, both errors decrease as the problem size n is increased. The eoc is very close to 2 for all n, demonstrating the convergence of the algorithm. As n is increased the eog grows beyond 3, which shows this algorithm has worse than cubic time complexity for large n.

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

n	Time(s)	$\|\|u-u_h\|\|_{L^\infty}$	$\|\|u-u_h\|\|_{L^2}$	eoc	eog
125	0.18141	5.1807×10^{-5}	2.5904×10^{-5}	-	-
250	0.60371	1.3054×10^{-5}	6.5275×10^{-6}	2.0001	1.7346
500	4.1663	3.2767×10^{-6}	1.6384×10^{-6}	1.9999	2.7868
1000	36.113	8.2082×10^{-7}	4.1041×10^{-7}	2.0000	3.1157
2000	404.48	2.0539×10^{-7}	1.0267×10^{-7}	2.0001	3.4855

Table 1: 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.

For (3.5) we have $A = B = T$, $T = T^*$, $X = U$ 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 Table 2. These results show that both errors decrease as n is increased. Also the eoc is close enough to 2 to conclude that this algorithm converges. The eog demonstrates that this algorithm has an approximate cubic time complexity.

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}	2.0001	2.9124
500	77.888	3.2767×10^{-6}	1.6384×10^{-6}	1.9999	3.0096
1000	640.22	8.2088×10^{-7}	4.1044×10^{-7}	2.0000	3.0391
2000	5093.1	2.0467×10^{-7}	1.0237×10^{-7}	2.0052	2.9919

Table 2: Results using the Bartels-Stewart algorithm.

By breaking this algorithm down into its component parts, we can time each step to see which is the most costly. The results of doing so are given in Table 3, where the steps are:

1. Compute the Schur form $T = PRP^*$ (using `linalg.schur` from the SciPy library)
2. Solve the $R^*V + VR = P^*FP$ for V, which is a triangular system
3. Compute the solution $U = PVP^*$

As can be seen from the results in Table 3, the most costly part of the algorithm is by far the second step, which is due to the fact that solving the triangular system uses back substitution which requires multiple nested `for` loops.

n	1	2	3	Total
125	0.012667	1.2357	0.00023031	1.2486
250	0.048434	9.6218	0.0010102	9.6712
500	0.26262	77.619	0.0064600	77.888
1000	0.083478	639.34	0.043818	640.22
2000	5.1290	5087.5	0.41275	5093.1

Table 3: Timings for each step 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 Table 4. The errors here are similar to the errors given in Table 2. Again the eoc is close enough to 2 to conclude that this algorithm converges. However, the eog seems to vary depending on the problem size n.

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}	2.0001	0.92964
500	1.9383	3.2767×10^{-6}	1.6384×10^{-6}	1.9999	4.0850
1000	4.0920	8.2084×10^{-7}	4.1042×10^{-7}	2.0000	1.0780
2000	32.029	2.0503×10^{-7}	1.0259×10^{-7}	2.0027	2.9685
4000	279.95	4.9949×10^{-8}	2.4876×10^{-8}	2.0377	3.1277

Table 4: 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 [5] can be used to solve the Sylvester equation $AX + XB = C$. 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, $\mu_1, \ldots, \mu_m$ are the eigenvalues values of B and the columns of P and Q are the eigenvectors of A and B, respectively. Letting $\widetilde{C} = P^{-1}CQ$, the solution is then:

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

In the case of (3.5), $A = B = T$, $X = U$ 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 and the columns of P are the eigenvectors of T. Letting $\widetilde{F} = P^{-1}FP$, the solution is then:

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

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 Table 5. The errors are again similar to the previous methods. However, here the eoc 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`. It is likely that the approximations become less accurate as n is increased. The eog here is however slightly better than the previous methods.

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}	2.0000	1.9870
500	0.59718	3.2767×10^{-6}	1.6384×10^{-6}	1.9999	2.1250
1000	2.2211	8.2086×10^{-7}	4.1043×10^{-7}	1.9999	1.8950
2000	11.571	2.0467×10^{-7}	1.0237×10^{-7}	2.0053	2.3812
4000	70.984	4.9966×10^{-8}	2.4876×10^{-8}	2.0350	2.6170
8000	466.83	8.6360×10^{-9}	4.1223×10^{-9}	2.5330	2.7173

Table 5: Results using a similarity transformation, calculating the eigenvalues and eigenvectors using `numpy.linalg.eig`.

Similarly to the Bartels-Stewart algorithm, 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 $\widetilde{F} = P^{-1}FP$
4. Calculate $\widetilde{U}$, where $\widetilde{u}_{ij} = \frac{\widetilde{f}_{ij}}{\lambda_i + \lambda_j}$
5. Calculate solution $U = P\widetilde{U}P^{-1}$

The results of doing so are shown in Table 6.

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

Table 6: 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 in Table 6, 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 Table 7, and timings for each step are given in Table 8. Similar errors are given by this method, however the eoc is closer to 2 for $n \leq 8000$. When $n = 16000$ the eoc increases significantly, however the errors here are so small that this is likely due to the impact of roundoff errors. The eog is also significantly less demonstrating that using this algorithm and computing the eigenvalues explicity achieves a near square time complexity. Also this is the only method so far that has been able to solve the problem with $n = 16000$ without running out of memory.

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}	2.0001	1.9137
500	1.2021	3.2767×10^{-6}	1.6384×10^{-6}	1.9999	1.9836
1000	5.5904	8.2082×10^{-7}	4.1041×10^{-7}	2.0000	2.2174
2000	20.247	2.0541×10^{-7}	1.0270×10^{-7}	2.0000	1.8567
4000	82.243	5.1314×10^{-8}	2.5656×10^{-8}	2.0018	2.0222
8000	360.35	1.2814×10^{-8}	6.4065×10^{-9}	2.0020	2.1314
16000	1771.9	7.7761×10^{-10}	3.8764×10^{-10}	4.0429	2.2978

Table 7: 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
16000	892.51	168.51	217.68	288.92	204.25	1771.9

Table 8: Timing results for each step of the similarity transformation method, calculating the eigenvalues and eigenvectors explicitly.

As can be seen from the results in Table 8, calculating the eigenvalues and eigenvectors explicitly outperforms calculating them using the NumPy library when n is large.

3.1.4 Shifted System

A projection method [5] 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 $B = WSW^{-1}$, where $S = \text{diag}(s_1, \ldots, s_n)$ are the eigenvalues of B and the columns of W are the eigenvectors of B
2. Compute $\hat{C} = CW$
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 $T = WSW^{-1}$, where $S = \text{diag}(s_1, \ldots, s_n)$ are the eigenvalues of T and the columns of W are the eigenvectors of T
2. Compute $\hat{F} = FW$
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 Table 9. The errors are approximately the same as the errors given previously. The eoc is approximately 2 and the eog is > 3 for large n demonstrating that this algorithm converges with a worse than cubic time complexity.

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}	2.0001	2.2436
500	2.1851	3.2767×10^{-6}	1.6384×10^{-6}	1.9999	2.3250
1000	19.498	8.2082×10^{-7}	4.1041×10^{-7}	2.0000	3.1576
2000	194.13	2.0541×10^{-7}	1.0271×10^{-7}	2.0000	3.3156
4000	2452.5	5.1347×10^{-8}	2.5674×10^{-8}	2.0009	3.6592

Table 9: Results using a projection method to solve n linear systems.

The steps of this method can also be broken down into component parts and each part timed. The results of doing so are given in Table 10. Unsuprisingly, the most costly part of the algorithm is solving the n linear systems.

n	1	2	3	4	Total
125	0.057421	0.0043032	0.3028	7.1764×10^{-7}	0.092085
250	0.23967	0.0041912	0.19177	0.00046206	0.43609
500	0.85408	0.0079994	1.3200	0.0030336	2.1851
1000	3.3709	0.026276	16.079	0.021675	19.498
2000	13.278	0.50065	180.81	0.16621	194.13
4000	53.300	1.8301	2396.1	1.2866	2452.5

Table 10: Timing results of each step using a projection method to solve n linear systems.

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 Table 11, using a convergence tolerance of 10^{-9}. The results show errors similar to all the direct methods. The eoc demonstrates that the algorithm converges very well. Although this algorithm is much faster than all the previous direct methods, the eog demonstrates that when n is significantly large, the algorithm exhibits a worse than cubic time complexity, which is shown here when $n = 8000$.

n	Time(s)	$\|\|u - u_h\|\|_{L^\infty}$	$\|\|u - u_h\|\|_{L^2}$	eoc	eog
125	0.014182	5.1807×10^{-5}	2.5904×10^{-5}	-	-
250	0.042605	1.3054×10^{-5}	6.5275×10^{-6}	2.0001	1.5870
500	0.14136	3.2767×10^{-6}	1.6384×10^{-6}	1.9999	1.7303
1000	0.54308	8.2082×10^{-7}	4.1041×10^{-7}	2.0000	1.9418
2000	2.3162	2.0541×10^{-7}	1.0271×10^{-7}	2.0000	2.0925
4000	9.8487	5.1379×10^{-8}	2.5689×10^{-8}	2.0000	2.0882
8000	103.34	1.2848×10^{-8}	6.4239×10^{-9}	2.0000	3.3913

Table 11: 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 [7] 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 [4] a modified conjugate gradient (MCG) algorithm is proposed, which is adapted for solving Sylvester Equations. The general algorithm for solving $AX + XB = C$, where A, B, C and X are all $n \times n$ matrices, is as follows:

1. Choose initial guess $X^{(0)}$ and calculate:
   - $R^{(0)} = C - AX^{(0)} - X^{(0)}B$
   - $Q^{(0)} = A^T R^{(0)} + R^{(0)} + R^{(0)}B^T$
   - $Z^{(0)} = Q^{(0)}$
2. If $R^{(k)} < \text{tol}$ or $Z^{(k)} < \text{tol}$, stop. Else, go to 3.
3. Calculate:
   - $\gamma_k = \frac{[R^{(k)}, R^{(k)}]}{[Z^{(k)}, Z^{(k)}]}$, where $[R, R] = \text{tr}(R^T R)$ is the trace of the matrix $R^T R$
   - $X^{(k+1)} = X^{(k)} + \gamma_k Z^{(k)}$
   - $R^{(k+1)} = C - AX^{(k+1)} - X^{(k+1)}B$
   - $Q^{(k+1)} = A^T R^{(k+1)} + R^{(k+1)} + R^{(k+1)}B^T$

   and return to Step 2.

In the case of the matrix equation $TU + UT = F$, the algorithm is as follows:

1. Choose initial guess $U^{(0)}$ and calculate:
   - $R^{(0)} = F - TU^{(0)} - U^{(0)}T$
   - $Q^{(0)} = TR^{(0)} + R^{(0)} + R^{(0)}T$
   - $Z^{(0)} = Q^{(0)}$
2. If $R^{(k)} < \text{tol}$ or $Z^{(k)} < \text{tol}$, stop. Else, go to 3.
3. Calculate:

- $\gamma_k = \frac{[R^{(k)}, R^{(k)}]}{[Z^{(k)}, Z^{(k)}]}$
- $U^{(k+1)} = U^{(k)} + \gamma_k Z^{(k)}$
- $R^{(k+1)} = F - TU^{(k+1)} - U^{(k+1)}T$
- $Q^{(k+1)} = TR^{(k+1)} + R^{(k+1)} + R^{(k+1)}T$

and return to Step 2.

Results using this algorithm with a convergence tolerance of 10^{-3} are given in Table 12. These results show that, even when using this relatively large convergence tolerance, this algorithm is slow. Also the errors does not always decrease as n is increased, which is reflected in the eoc. The errors here are also bigger than those for previous methods for large n, which is due to the large convergence tolerance. The eog demonstrates that this algorithm has a worse than cubic time complexity for large n.

n	Time(s)	$\|\|u - u_h\|\|_{L^\infty}$	$\|\|u - u_h\|\|_{L^2}$	eoc	eog
125	11.006	1.1451×10^{-6}	5.7254×10^{-7}	-	-
250	53.234	3.7599×10^{-5}	1.8800×10^{-5}	-5.0662	2.2741
500	397.11	4.6673×10^{-5}	2.3337×10^{-5}	-0.31294	2.8991
1000	3865.5	3.3353×10^{-5}	1.6677×10^{-5}	0.48547	3.2830

Table 12: Results obtained using the MCG algorithm.

3.2.4 Preconditioned MCG

[4] also proposes a preconditioned version of the MCG algorithm which accelerates the speed of convergence. This version of the algorithm solves $AX + XB = C$ by solving the equation $X^{(k+1)} = \widetilde{A}X^{(k)}\widetilde{B} + \widetilde{C}$, with:

$$\widetilde{A} = I + 2(\alpha A - I)^{-1}$$

$$\widetilde{B} = I + 2(\alpha B - I)^{-1}$$

$$\widetilde{C} = -2\alpha(\alpha A - I)^{-1}C(\alpha B - I)^{-1}$$

where α is a parameter that needs to be chosen and I is the identity matrix. [4] suggests choosing $\alpha = \pm\sqrt{n}/||A||_F$ or $\alpha = \pm\sqrt{n}/||B||_F$, where $||A||_F = \sqrt{[A, A]} = \sqrt{\text{tr}(A^T A)}$, for good convergence. Once α has been chosen and $\widetilde{A}, \widetilde{B}$ and $\widetilde{C}$ have been calculated, the algorithm is as follows:

1. Choose initial guess $X^{(0)}$ and calculate:
   - $R^{(0)} = -\widetilde{C} + X^{(0)} - \widetilde{A}X^{(0)}\widetilde{B}$
   - $Q^{(0)} = \widetilde{A}^T R^{(0)}\widetilde{B}^T + R^{(0)}$
   - $Z^{(0)} = Q^{(0)}$

2. If $R^{(k)} < \text{tol}$ or $Z^{(k)} < \text{tol}$, stop. Else, go to 3.

3. Calculate:
   - $\gamma_k = \frac{[R^{(k)}, R^{(k)}]}{[Z^{(k)}, Z^{(k)}]}$
   - $X^{(k+1)} = X^{(k)} + \gamma_k Z^{(k)}$
   - $R^{(k+1)} = -\widetilde{C} + X^{(k+1)} - \widetilde{A} X^{(k+1)} \widetilde{B}$
   - $Q^{(k+1)} = \widetilde{A}^T R^{(k+1)} \widetilde{B}^T - R^{(k+1)}$
   - $\eta_k = \frac{[R^{(k+1)}, R^{(k+1)}]}{[R^{(k)}, R^{(k)}]}$
   - $Z^{(k+1)} = Q^{(k+1)} + \eta_k Z^{(k)}$

   and return to Step 2.

To solve the matrix equation $TU + UT = F$, we instead solve the equation $U^{(k+1)} = \widetilde{T} U^{(k)} \widetilde{T} + \widetilde{F}$, with:

$$\widetilde{T} = I + 2(\alpha T - I)^{-1}$$

$$\widetilde{F} = -2\alpha(\alpha A - I)^{-1} C (\alpha B - I)^{-1}$$

Choosing $\alpha = \pm \sqrt{n} / ||T||_F$ and calculating $\widetilde{T}$ and $\widetilde{F}$, the algorithm proceeds as follows:

1. Choose initial guess $U^{(0)}$ and calculate:
   - $R^{(0)} = -\widetilde{F} + U^{(0)} - \widetilde{T} U^{(0)} \widetilde{T}$
   - $Q^{(0)} = \widetilde{T} R^{(0)} \widetilde{T} + R^{(0)}$
   - $Z^{(0)} = Q^{(0)}$

2. If $R^{(k)} < \text{tol}$ or $Z^{(k)} < \text{tol}$, stop. Else, go to 3.

3. Calculate:
   - $\gamma_k = \frac{[R^{(k)}, R^{(k)}]}{[Z^{(k)}, Z^{(k)}]}$
   - $U^{(k+1)} = U^{(k)} + \gamma_k Z^{(k)}$
   - $R^{(k+1)} = -\widetilde{F} + U^{(k+1)} - \widetilde{T} U^{(k+1)} \widetilde{T}$
   - $Q^{(k+1)} = \widetilde{T} R^{(k+1)} \widetilde{T} - R^{(k+1)}$
   - $\eta_k = \frac{[R^{(k+1)}, R^{(k+1)}]}{[R^{(k)}, R^{(k)}]}$
   - $Z^{(k+1)} = Q^{(k+1)} + \eta_k Z^{(k)}$

   and return to Step 2.

Results using the preconditioned MCG algorithm with a tolerance of 10^{-9} are given in Table 13. These results demonstrate that this method vastly outperforms the standard MCG algorithm. The errors are similar to previous methods (excluding the MCG algorithm) and the eoc is close to 2 in most cases demonstrating that this algorithm converges well apart from in the case where $n = 8000$, which could be because the error is approaching the convergence tolerance. The eog shows that although this algorithm is fast, for large n it has close to cubic time complexity.

n	Time(s)	$\|\|u - u_h\|\|_{L^\infty}$	$\|\|u - u_h\|\|_{L^2}$	eoc	eog
125	0.0010328	5.1807×10^{-5}	2.5904×10^{-5}	-	-
250	0.0040712	1.3054×10^{-5}	6.5275×10^{-6}	2.0001	1.9789
500	0.027955	3.2767×10^{-6}	1.6384×10^{-6}	1.9999	2.7796
1000	0.58508	8.2078×10^{-7}	4.1038×10^{-7}	2.0001	4.3875
2000	3.5886	2.0585×10^{-7}	1.0275×10^{-7}	1.9968	2.6167
4000	39.681	5.3078×10^{-8}	2.6012×10^{-8}	1.9561	3.4670
8000	336.49	1.7720×10^{-8}	6.6084×10^{-9}	1.5830	3.0840

Table 13: Results obtained using the preconditioned MCG algorithm.

[4] also proposes a parallel implementation of the preconditioned MCG algorithm, which would result in better performance than given above. However the parallel version of this algorithm goes beyond the scope of this project.

4 Uncertainty

Uncertainty can arise in PDEs if, for example, coefficients, boundary conditions or initial conditions are unknown. A solution to dealing with uncertainty is to model unknown parameters as random variables, and then appropriate methods can be used to solve the system.

4.1 Single Parameter

The simplest way to introduce uncertainty into a PDE is by introducing a single unknown parameter. Define the spatial domain $\mathcal{D}$ as $\mathcal{D} = \{(x, y) : 0 \leq x \leq 1,\ 0 \leq y \leq 1\}$, and let Ω denote the sample space. We can introduce an unknown coefficient $\varepsilon(\omega)$, where $\omega \in \Omega$ denotes dependence on a random variable and $\varepsilon : \Omega \to \mathbb{R}$ is uniformly distributed over the interval $[-1, 1]$ with $\varepsilon \neq 0$. Let $u(x, y, \varepsilon) = \sin(\pi x)\sin(\pi y) + \varepsilon \sin(3\pi x)\sin(5\pi y)$ be the exact solution of the equation:

$$\begin{aligned} -\varepsilon u_{xx} - \varepsilon u_{yy} &= f \quad \text{on} \quad \mathcal{D} \\ u &= 0 \quad \text{on} \quad \partial\mathcal{D} \end{aligned} \tag{4.1}$$

which gives:

$$\begin{aligned} u_{xx} &= -\pi^2 \sin(\pi x)\sin(\pi y) - 9\pi^2 \varepsilon \sin(3\pi x)\sin(5\pi y) \\ u_{yy} &= -\pi^2 \sin(\pi x)\sin(\pi y) - 25\pi^2 \varepsilon \sin(3\pi x)\sin(5\pi y) \\ \implies f &= 2\pi^2 \varepsilon \sin(\pi x)\sin(\pi y) + 34\pi^2 \varepsilon^2 \sin(3\pi x)\sin(5\pi y) \end{aligned} \tag{4.2}$$

The equation to be solved is therefore:

$$\begin{aligned} -\varepsilon u_{xx} - \varepsilon u_{yy} = 2\pi^2 \varepsilon \sin(\pi x)\sin(\pi y) + 34\pi^2 \varepsilon^2 \sin(3\pi x)\sin(5\pi y) \quad &\text{on} \quad \mathcal{D} \times \Omega \\ u = 0 \quad &\text{on} \quad \partial\mathcal{D} \times \Omega \end{aligned} \tag{4.3}$$

By discretising the domain using the centred finite difference approximations, (4.3) can be formed as a matrix equation:

$$TU + UT = F \tag{4.4}$$

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 problem is significantly harder to solve accurately and efficiently than the problem in Section 3 due to the fact that ε is an unknown, random coefficient.

As ε is unknown, the exact solution to (4.3) cannot be computed and therefore solutions to (4.4) cannot be compared to the true solution. Instead, certain

quantities of interest can be computed using the probability density function $\rho(\omega)$ of ε. Firstly, the expectation $\mathbb{E}[u]$ can be computed as:

$$\mathbb{E}[u] = \int_{\Omega} u(x, y, \varepsilon)\rho(\omega)d\omega \tag{4.5}$$

The variance $\mathrm{Var}[u]$ can also be computed as:

$$\mathrm{Var}[u] = \int_{\Omega} \left(u(x, y, \varepsilon) - \mathbb{E}[u]\right)^2 \rho(\omega)d\omega \tag{4.6}$$

and the standard deviation $\mathrm{Std}[u]$ as:

$$\mathrm{Std}[u] = \sqrt{\mathrm{Var}[u]} \tag{4.7}$$

As ε is uniformly distributed over the interval $[-1, 1]$, it has a probability distribution function:

$$\rho(\omega) = \frac{1}{2} \tag{4.8}$$

The expectation is therefore:

$$\begin{aligned}
\mathbb{E}[u] =& \frac{1}{2}\int_{-1}^{1} \sin(\pi x)\sin(\pi y) + \varepsilon \sin(3\pi x)\sin(5\pi y)d\omega \\
=& \frac{1}{2}\left[\varepsilon \sin(\pi x)\sin(\pi y) + \frac{\varepsilon^2}{2}\sin(3\pi x)\sin(5\pi y)\right]_{-1}^{1} \\
=& \sin(\pi x)\sin(\pi y)
\end{aligned} \tag{4.9}$$

and the variance is:

$$\begin{aligned}
\mathrm{Var}[u] =& \frac{1}{2}\int_{-1}^{1} (\sin(\pi x)\sin(\pi y) + \varepsilon\sin(3\pi x)\sin(5\pi y) - \sin(\pi x)\sin(\pi y))^2 d\omega \\
=& \frac{1}{2}\int_{-1}^{1} \varepsilon^2 \sin^2(3\pi x)\sin^2(5\pi y)d\omega \\
=& \frac{1}{2}\left[\frac{\varepsilon^3}{3}\sin^2(3\pi x)\sin^2(5\pi y)\right]_{-1}^{1} \\
=& \frac{1}{3}\sin^2(3\pi x)\sin^2(5\pi y)
\end{aligned} \tag{4.10}$$

which gives the standard deviation as:

$$\mathrm{Std}[u] = \frac{1}{\sqrt{3}}\sin(3\pi x)\sin(5\pi y) \tag{4.11}$$

Once solutions to (4.4) are computed they can be compared to the known expectation, variance and standard deviation.

4.1.1 Monte Carlo

A simple way of solving (4.4) 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 (4.4) 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 expectation can be calculated as:

$$\mathbb{E}_M[u] = \frac{1}{M}\sum_{i=1}^{M} u_h(x, y, \varepsilon) \tag{4.12}$$

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. The expectation here is denoted as $\mathbb{E}_M[u]$ to indicate that it is the expectation computed from the Monte Carlo method, rather than the known expectation (that is, (4.5)).

Similarly, the variance and standard deviation can also be computed as:

$$\mathrm{Var}_M[u] = \frac{1}{M-1}\sum_{i=1}^{M}(u_h - \mathbb{E}_M[u])^2 \tag{4.13}$$

and:

$$\mathrm{Std}_M[u] = \sqrt{\mathrm{Var}_M[u]} \tag{4.14}$$

Any of these quantites can then be chosen and compared to the known values as a measure of error for the problem. By choosing the mean we can measure the error as:

$$\begin{aligned} \text{Error} &= \left|\left| \mathbb{E}[u] - \mathbb{E}_M[u] \right|\right| \\ &= \left|\left| \mathbb{E}[u] - \frac{1}{M}\sum_{i=1}^{M} u_h(x, y, \varepsilon) \right|\right| \end{aligned} \tag{4.15}$$

As noted in [2], for the error to converge to 0 the mesh size must be decreased as the number of samples is increased. This is because the error depends both on the mesh size and the number of samples, so by changing only one of these factors the method will be converging to the error of the factor that is not being changed, rather than to 0.

Results using the Monte Carlo method, with a single uncertain parameter, are given in Table 14. As can be seen by these results, the error here is much bigger than the PDE without uncertainty.

n	M	Time(s)	Error
125	10		0.098539
250	40		0.14002
500	160		0.042046
1000	640		0.0088454
2000	2560		0.00030675

Table 14: Results using the Monte Carlo method with a single uncertain parameter.

4.1.2 Stochastic Galerkin

The stochastic Galerkin method uses the properties of orthogonal polynomials to produce a functional approximation to the solution of a stochastic PDE (that is, a PDE with some kind of random input). It does this by using a Galerkin projection to "discretise the random dimensions to allow computation" [3].

A set of polynomials $\{\Psi_n(x), n \in \mathbb{N}\}$, where $\Psi_n(x)$ is a polynomial of exact degree n, is orthogonal if it satisfies the orthogonality condition:

$$\int_\Omega \Psi_n(x)\Psi_m(x)w(x)dx = h_n\delta_{nm}, \qquad n, m \in \mathbb{N} \tag{4.16}$$

where Ω represents the support of $\{\Psi_n\}$ (the subset of the domain containing elements not mapped to zero), $w(x)$ is a weight function, h_n are non-zero constants and δ_{nm} is the Kronecker delta ($\delta_{nm} = 1$ for $n = m$ and 0 otherwise).

As noted in [6], certain classes of orthogonal polynomials have the exact same weight function as the probability distribtution of certain random variables. Once the distribution of the variable is known, the solution u can be approximated via a polynomial chaos expansion as:

$$u(x, y, \varepsilon) = \sum_{k=0}^{P} u_k(x, y)\Psi_k(\varepsilon) \tag{4.17}$$

and can then be substituted into the PDE. Here $P+1 = \frac{(K+N)!}{K!N!}$, where N is the number of random variables and K is a convergence parameter to be chosen. Any random quantities can be represented via a Karhunen-Loeve expansion as:

$$\alpha = \bar{\alpha} + \sum_{k=1}^{Q} \sqrt{\lambda_k}\phi_k\varepsilon_k \tag{4.18}$$

for a spatially varying random field α with mean $\bar{\alpha}$, where λ_k and ϕ_k are the eigenvalues and eigenfunctions of the covariance function C_α.

A Galerkin projection is then performed on the PDE by multiplying it by Ψ_k for $k = 0, \ldots, P$, which gives a system of coupled deterministic differential equations which can be discretised and solved via, for example, the finite difference method. The mean and variance are then computed as:

$$\mathbb{E}[u] = u_0 \tag{4.19}$$

$$\mathrm{Var}[u] = \sum_{k=1}^{P} u_k^2 \mathbb{E}[\Psi_k^2] \tag{4.20}$$

and $\mathrm{Std}[u] = \sqrt{\mathrm{Var}[u]}$ as before.

In the case of the equation (4.4), ε is uniformly distrubted over $[-1, 1]$. The uniform distribution has a probability density function which is identical to the weighting function of the Legendre polynomials, which solve the equation:

$$\frac{d}{dx}\left[(1-x^2)\frac{dP_n(x)}{dx}\right] + n(n+1)P_n(x) = 0 \tag{4.21}$$

The SciPy library has a built in function, `special.legendre`, for calculating Legendre polynomials.

4.2 Multiple Parameters

4.2.1 Monte Carlo

4.2.2 Stochastic Galerkin

5 Application

6 Evaluation

6.1 Aims

6.2 Extensions

6.3 Conclusion

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. Bishop and O. E. Strack. A statistical method for verifying mesh convergence in Monte Carlo simulations with application to fragmentation. 88:279 – 306, 10 2011.

[3] P. Constantine. A Primer on Stochastic Galerkin Methods. 03 2007.

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

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

[6] D. Xiu and G. Karniadakis. The Wiener-Askey Polynomial Chaos for Stochastic Differential Equations. *SIAM Journal on Scientific Computing*, 24(2):619–644, 2002.

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