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. 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 below:

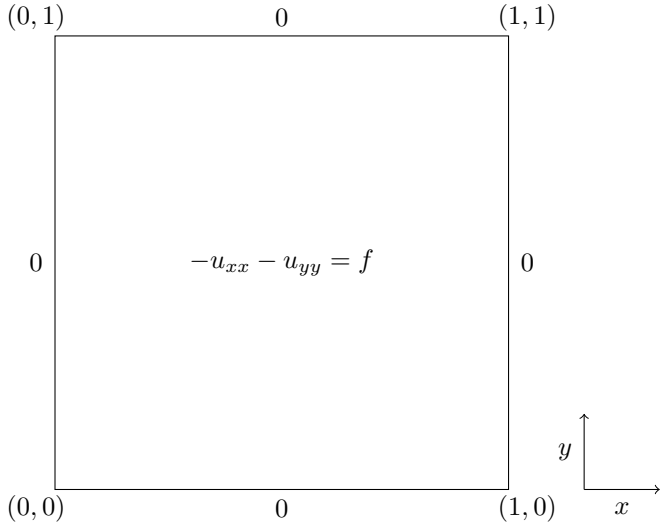

Figure 1: Domain for $-u_{xx} - u_{yy} = f$.

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 below:

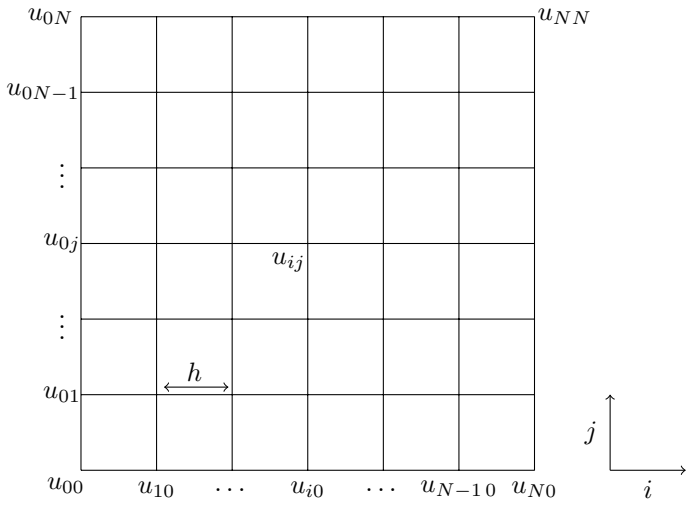

Figure 2: Discretised domain for $-u_{xx} - u_{yy} = f$.

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 iterative and direct) 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:

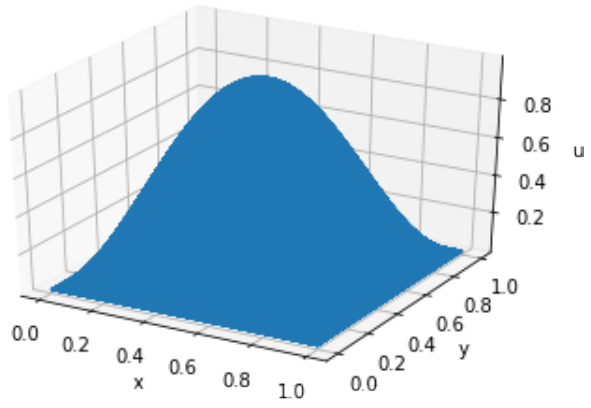

Figure 3: Plot of the solution with $n = 1000$.

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 for the system.
- Time(s): The time taken in seconds for the method to compute the solution to the problem.

- $L_{\max}$: Measures the maximum difference between the actual solution and computed solution for each u, defined as:

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

- L_{squared}: 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: Measures the rate of convergence of a method, which should approach 2 as the step size is increased. Defined as:

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

  where E_i is the error and h_i is the mesh 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:

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

where $\mathcal{F}$ is a vector of dimension n^2, $\mathcal{T} = I \otimes T + T \otimes I$ 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 given below:

[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
10	0.0026031	0.0066868	0.0034125	-
100	0.061922	8.0611×10^{-5}	4.0315×10^{-5}	1.9927
1000	36.681	8.2082×10^{-7}	4.1041×10^{-7}	1.9999
2000	428.06	2.0540×10^{-7}	1.0270×10^{-7}	2.0001

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

n	Time(s)	$\|\|u-u_h\|\|_{L^\infty}$	$\|\|u-u_h\|\|_{L^2}$	eoc
10	0.0076599	0.0066868	0.0034125	-
100	2.0357	8.0611×10^{-5}	4.0315×10^{-5}	1.9927
1000	1449.6	8.2083×10^{-7}	4.1041×10^{-7}	1.9999
2000	8675.8	$2.0561e \times 10^{-7}$	1.0276×10^{-7}	1.9986

Figure 5: Results using the Bartels-Stewart algorithm.

3.1.2 Bartels-Stewart Algorithm

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

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

where A^* denotes the conjugate transpose of A.

In this case $A = B = T$, $T = T^*$ and $X = U$, 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 given below:

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

n	Time(s)	$\|\|u-u_h\|\|_{L^\infty}$	$\|\|u-u_h\|\|_{L^2}$	eoc
10	0.00051880	0.0066868	0.0034125	-
100	0.027365	8.0611×10^{-5}	4.0315×10^{-5}	1.9927
1000	20.248	8.2083×10^{-7}	4.1041×10^{-7}	1.9999
2000	244.27	2.0558×10^{-7}	1.0277×10^{-7}	1.9988
4000	2886.7	5.0158×10^{-8}	2.4922×10^{-8}	2.0359

Figure 6: Results using SciPy's `scipy.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 Hessenberg-Schur Method

The Hessenberg-Schur method uses a similarity transformation [2] [3] to solve the Sylvester equation $AX + XB = F$, 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 $\tilde{F} = P^{-1}FQ$. The solution is then:

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

In this case, $A = B = T$ and $X = U$, 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. Also $\tilde{F} = P^{-1}FP$. The solution is therefore:

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

Using numpy.linalg.eig

The eigenvalues and eigenvectors can be computed using NumPy's `linalg.eig` function. The results using this are given below:

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

n	Time(s)	$\|\|u-u_h\|\|_{L^\infty}$	$\|\|u-u_h\|\|_{L^2}$	eoc
10	0.023603	0.0066868	0.0034125	-
100	0.049893	8.0611×10^{-5}	4.0315×10^{-5}	1.9927
1000	4.8863	8.2083×10^{-7}	4.1041×10^{-7}	1.9999
2000	26.098	2.0560×10^{-7}	1.0277×10^{-7}	1.9987
4000	185.13	5.0151×10^{-8}	2.4922×10^{-8}	2.0362
8000	1274.5	1.2959×10^{-8}	5.1767×10^{-9}	1.9527

Figure 7: Results using the Hessenberg-Schur method, calculating the eigenvalues and eigenvectors using `numpy.linalg.eig`.

n	1	2	3	4	5	Total
10	0.0057750	0.0045397	0.0010521	0.011951	0.00028515	0.023603
100	0.024929	0.0011380	0.0017610	0.020651	0.0014130	0.049893
1000	2.5253	0.070281	0.21545	1.8795	0.19585	4.8863
2000	15.457	0.0056458	1.5614	7.5858	1.4880	26.098
4000	121.16	0.060127	12.136	39.220	12.557	185.13
8000	922.39	0.04993	102.79	131.24	118.03	1274.5

Figure 8: Timing results for each step of the Hessenberg-Schur method, calculating the eigenvalues and eigenvectors using `numpy.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 given below:

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,

n	Time(s)	$\|\|u-u_h\|\|_{L^\infty}$	$\|\|u-u_h\|\|_{L^2}$	eoc
10	0.0041900	0.0066868	0.0034125	-
100	0.076722	8.0610×10^{-5}	4.0315×10^{-5}	1.9927
1000	7.0524	8.2082×10^{-7}	4.1041×10^{-7}	1.9999
2000	29.164	2.0541×10^{-7}	1.0270×10^{-7}	2.0000
4000	103.42	5.1313×10^{-8}	2.5655×10^{-8}	2.0018
8000	593.77	1.2814×10^{-8}	6.4065×10^{-9}	2.0020

Figure 9: Results using the Hessenberg-Schur method, calculating the eigenvalues and eigenvectors explicity.

n	1	2	3	4	5	Total
10	0.00089598	0.00071597	0.0014319	0.00087380	0.00027227	0.0041900
100	0.047597	0.00061584	0.0015020	0.0.025820	0.0011868	0.076722
1000	4.1079	0.0032451	0.23045	2.4986	0.21222	7.0524
2000	16.395	0.063026	1.4521	9.8474	1.4060	29.164
4000	52.499	0.034972	10.389	28.196	12.305	103.42
8000	238.04	0.12201	113.94	132.86	108.81	593.77

Figure 10: Timing results for each step of the Hessenberg-Schur method, calculating the eigenvalues and eigenvectors explicitly.

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 below:

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.

n	Time(s)	$\|\|u-u_h\|\|_{L^\infty}$	$\|\|u-u_h\|\|_{L^2}$	eoc
10	0.0023940	0.0066868	0.0034124	-
100	0.0048580	8.0611×10^{-5}	4.0315×10^{-5}	1.9927
1000	0.61869	8.2082×10^{-7}	4.1041×10^{-7}	1.9999
2000	2.1425	2.0541×10^{-7}	1.0271×10^{-7}	2.0000
4000	23.065	5.1378×10^{-8}	2.5689×10^{-8}	2.0000
8000	160.09	1.2848×10^{-8}	6.4239×10^{-9}	1.9999

Figure 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 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 below, using a convergence tolerance of 10^{-9}.

3.2.2 Gradient based method

In [4] a gradient based method for solving Sylvester equations is proposed. 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.

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] G. H. Golub, S. Nash, and C. Van Loan. A Hessenberg–Schur method for the problem AX + XB = C. 24:909 – 913, 01 1980.

[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.