MECH511 Programming Assignment 2
Krylov Methods

Nick Earle

April 8, 2019

This assignment looked at the efficiency of different types of linear system solvers using the problem from programming assignment 3 of Mech 510. That is solving the 2D energy equation over a domain of $(x, y) = (0 : 40, 0 : 10)$. The following parameters we used:

$$
\begin{aligned}
u(x,y) &= 6\bar{u}y(1-y) & \bar{u} &= 3m/s \\
v(x,y) &= 0 & \Delta t &= 0.25 \\
T(y=0) &= 0 & Re &= 25 \\
T(y=10) &= 1 & Pr &= 0.7 \\
T(x=0) &= y & Ec &= 0.1 \\
\left.\frac{\partial T}{\partial n}\right|_{(x=40)} &= 0
\end{aligned}
$$

1 Direct Solvers

First we compare Direct Solvers, for this I chose two solvers that can be used in any situation, LU factorisation and LU factorisation with partial pivoting, as well as Cholesky factorisation, which can only be used if our LHS matrix is symmetric positive definite (SPD). Both LU and LUP have a computational cost of $\frac{2}{3}n^3$ while the Cholesky method, when applicable, has a cost of $\frac{1}{3}n^3$, about twice as fast. For our full LHS matrix, including boundary conditions, the matrix is not SPD, so unfortunately Cholesky can't be used. In most cases it is also diagonally dominant, which means that there will be no row pivoting, which by adding the row check makes LUP slightly slower than LU. For those reasons basic LU will be our method of choice, but for comparison, removing the boundary conditions and by using winds that are constant in their respective direction, we do get a SPD matrix, so Cholesky can be compared. The solve times are shown below for varying mesh sizes[1] (Table 1).

[1]Note: The cases with no BCs were run on Ubuntu 18.04 with an 8 core Intel Core i7-4790 @ 3.6 GHz and 15.5 GB of memory. All other cases were run on Windows 10 Pro with a 4 core Intel Core i5-6200U @ 2.8 GHz and 7.85 GB of memory.

Table 1: Direct Solver solve times

Method	20×10	40×20	80×40
LU	$0.1560s$	$5.9320s$	$312.026s$
LU (no BC)	$0.041187s$	$2.10224s$	$134.291s$
LUP (no BC)	$0.043156s$	$2.19786s$	$134.896s$
Cholesky (no BC)	$0.018547s$	$0.766829s$	$47.4729s$

Looking at the solve times for each method, I was surprised generally by how slow they were. I assumed they would be faster, but the slowness may be due to my (naive) homemade functions used for the solvers, and a case of poor memory management (i.e. vectors vs. arrays, I have recently installed a new library, 'xtensor', which promises to be almost as fast as standard arrays but with the functionality of vectors, unfortunately I didn't have time to implement. Next time:). As for the expectations between mesh sizes, yes, the schemes do come quite close to my expectations. For each increase the change is between 50-60 times which is close to the theoretical increase of $4^3 = 64$ times. I was surprised though, that Cholesky was consistently much more than two times faster than LU, although it probably has something to do with my implementation.

2 Approximate Factorisation

Next, to compare approximate factorisation with our full direct solvers we look at the block Thomas algorithm as was implemented in programming assignment 3 of MECH510. The Thomas algorithm has a theoretical cost of $\mathcal{O}(n)$ operations compared to the $\mathcal{O}(n^3)$ of the previous schemes, but does lack somewhat in accuracy compared to the direct schemes which provide an "exact" solution. The execution time and L_2-norm of the errors are found below (Table 2). With smaller mesh sizes the process was so quick that it was difficult to get a reliable execution time for a single solve. It looks like we are increasing around $\mathcal{O}(n)$ with larger matrices, but I ran out of memory for 320×160. I then ran it to steady state as well to try to see the how it changes with n. As we can deduce from the times below, when solving to steady state the time does increase at a somewhat linear rate. We can also see that as we increase the size of the mesh, the approximate factorisation loses some accuracy with the L_2-norm steadily increasing.

Table 2: Approximate Factorisation solve times

Mesh	20×10	40×20	80×40	160×80
Single solve	$0.001998s$	$0.002002s$	$0.002999s$	0.005999
L_2-norm	0.0782852	0.106050	1.8030	–
(SS) Iterations	262	298	262	167
(SS) Time	$6.73368s$	$8.13489s$	$10.1133s$	14.9963

3 Krylov Methods - GMRES

Next up we look at the GMRES scheme with no preconditioning. Running the solver using 20, 30, and 40 vectors for the iteration, the L_2-norm of the error and time are tabulated below (Table 3). Looking at the results, for the meshes of 20×10 and 40×20, GMRES comes much closer to a solution using 40 vectors, which shows in the norm or the error. For the 20×10, GMRES reaches a convergence of 10^{-9} in 41 iterations, while the 40×20 mesh took just 129 iterations, but through 360 iterations ($\approx$ 4min) on the 80×40 mesh, the residual was still over 0.5. This shows that for smaller systems GMRES is much quicker than any direct method and much more accurate than approximate factorisation. For larger systems however, some preconditioning or restarting may be required.

Table 3: GMRES Time and Accuracy

Mesh	Vectors	L_2-norm	Time (s)
20×10	20	$7.23399e-06$	0.07346
	30	$2.32636e-08$	0.10965
	40	$6.97171e-11$	0.14369
40×20	20	0.0114612	0.41914
	30	0.00178807	0.61222
	40	0.00032642	0.85685
80×40	20	1.85626	5.01985
	30	1.85974	7.33893
	40	1.84872	9.72577

4 GMRES with preconditioning

Now to look at GMRES with preconditioning. As a simple preconditioner we will use the tridiagonal of the full matrix and invert that. Unfortunately, my inversion algorithm used to invert the decomposed L and R matrices of the tridiagonal is dreadfully slow, so I was unable (to wait long enough) to solve on the larger mesh, so the 40×20 mesh was used to show some of the effects. Table 4, below, shows the L_2-norm and solve times for the GMRES routine as above using an LR preconditioner. We can see from the data below, that early on in the GMRES solve, the preconditioner does nothing more than increase the execution time. But as we start to increase the number of vectors, the preconditioner does help GMRES to converge faster. Comparing just these three cases to those above it is hard to conclude, but running until a convergence of 10^{-9}, GMRES without preconditioning takes 129 iterations, while preconditioning does bring that down to 121 iterations. While this isn't a huge amount, I expect it to really make a difference on larger meshes (the 80×40 case?) and especially for unbounded matrices (we saw this a bit in MATH521).

Table 4: Preconditioning Error and Execution Time

Vectors	L_2-norm	Time (s)
20	0.0124917	0.496139
30	0.00179775	0.750769
40	0.000284051	1.012650

Looking at the results above, it is very clear that iterative solving methods provide huge advantages compared to the direct methods and an accuracy advantage compared to approximate factorisation. I am pretty interested now to see what other preconditioners can be used in different situations to speed up GMRES and other schemes. Trying the easiest one I know, Jacobi preconditioning, it did absolutely nothing, but I'm sure there are some that provide huge savings, and if I could get my inversion scheme to work a bit faster, I could maybe implement. That being said I am going to try to improve some of this code my managing memory a bit better, and also switching away from vectors of vectors of vectors etc. I also tried to implement the restarted GMRES to see if that would cut down on time and memory usage, but couldn't get it to converge before running out of time, even though I am submitting late, but stay tuned.