CS450: Project Progress Report

Haroun Habeeb **hhabeeb2@illinois.edu**

Summary:

I understand ODEs, IVPs and PDEs, and know techniques to solve them. Of my deliverables, I have finished the two exercise problems. Their solutions are presented here. I have also written up code for the three computer problems. They work as expected. However, they are not yet optimized. I also need to test my implementation on representative matrices.

I am using the second edition of the course textbook. I opted for extra homework. I opted to do problems exercise 11.1, exercise 11.7, computer problem 11.4, computer problem 11.12, computer problem 11.13.
For each question I have written down the problem (roughly).

All implementations were done in python. Numpy and scipy were used extensively.

The aim of the homework was to familiarize me with PDEs and solving them.

Exercise 11.1

Suppose you're given a general-purpose subroutine for solving IVPs for system of n first order-ODEs $y' = f(t, y)$ and it is the only available subroutine. For each type of problem in parts (a)-(c), answer each of the following questions:

1. What is the function f?
2. How would you obtain initial conditions?
3. What special properties could be used to decide on a solver?

(a) To compute $\int_a^b g(t)dt$

If we use $y(t) = \int_a^t g(x)dx$, then the required integral is $y(b)$. Hence, $f(t, y) = \frac{d\int_a^t g(x)dx}{dt} = g(t)$

The initial value can be evaluated at $t = a$. $y(a) = 0$ since the upper and lower limits of the integral in $y(a)$ are equal.

Any solver that can solve $y' = g(t)$ can solve our ODE. However, unless we know g, we cannot discuss special properties of the ODE.

(b) To solve $y'' = y^2 + t \; \forall \; 0 \leq t \leq 1$ given $y(0) = 0, y(1) = 1$
We need to reduce a higher order differential equation to a first order differential equation.

To do so, we'd define two variables – say a, b such that

$$\begin{bmatrix} a' \\ b' \end{bmatrix} = \begin{bmatrix} b \\ a^2 + t \end{bmatrix}$$

Where $a = y$ and $b = y'$ essentially. Notice that the line above is a linear ODE and we have a subroutine for that.

Formatted: Font: Bold

Formatted: Normal, No bullets or numbering

Formatted: Font:

Since $a = y$, we know $a(0) = 0$ and $a(1) = 1$.However, b is trickier. We know that $b'(0) = 0$ and $b'(1) = 2$. We would need to guess values for $b(0), b(1)$ until the system becomes satisfiable. This is essentially the shooting method of solving BVPs.

As such, it's a normal multi-dimensional IVP with coupled variables. There are no other special properties that would help an ODE solver. The jacobian is a simple $\begin{bmatrix} 0 & 1 \\ 2a & 0 \end{bmatrix}$ which is well conditioned if a is reasonable.

Alternatively, we can also discretize along t and solve using the colocation method. We could also use a set of basis function – the choice of basis functions would produce different kinds of solutions.

(c) Solve heat equation $u_t = cu_{xx}\ \forall x \in [0,1]$ given $u(0,x) = g(x), u(t,0) = u(t,1) = 0$
The heat equation is a parabolic 2nd order PDE.

Since we need to essentially remove one variable to make it an ODE, we could discretize either along x or along t. Lets say that we discretize along x. Hence, we have an ODE at each x_i where $x_i = \frac{i}{n}$ for $i \in \{0,1,2 \dots n\}$. Let $v_i(t) = u(t, x_i)$

However, we can no longer differentiate along x. We must instead use a finite difference method.

We replace $u_{xx}(t,x)$ by $\frac{1}{\frac{1}{n^2}}(u(t,x_{i+1}) + u(t,x_{i-1}) - 2u(t,x_i))$, or any other equivalent for $u_{xx}(t,x)$.

Hence the system of ODEs is:

$$\boldsymbol{v}'(t) = cn^2 M\boldsymbol{v}(t)$$

Hence, the function f is $cn^2 M\boldsymbol{v}$(t)

Where M is a triadiagonal matrix with $M_{i,i} = -2, M_{i+1,i} = M_{i,i+1} = 1$.

The boundary conditions become $v_i(0) = g(x_i)$ and $v_0(t) = v_n(t) = 0$

The jacobian of the system is ill conditioned because of the cn^2 factor. Since the jacobian is ill conditioned, our ODE system solver would have to be capable of handling such systems.

Exercise 11.7

Prove that the SOR method diverges if w doesn't lie in $(0,2)$

The successive over relaxation method is used to iteratively solve linear systems such as $Ax = b$. From the book, we know that the iteration is

$$x^{(k+1)} = (D + wL)^{-1}\big((1-w)D - wU\big)x^{(k)} + w(D + wL)^{-1}b$$

Following the notation in the text book, $M = \frac{1}{w}D + L$ and $N = \left(\frac{1}{w} - 1\right)D - U$

$$x^{(k+1)} = (D + wL)^{-1}(D + wL - wD - wU - wL)x^{(k)} + w(D + wL)^{-1}b$$

$$\Rightarrow x^{(k+1)} = (I - w(D + wL)^{-1}A)x^{(k)} + w(D + wL)^{-1}b$$

$$\Rightarrow x^{(k+1)} - x^{(k)} = w(D + wL)^{-1}(b - Ax^{(k)})$$

Let $e^{(k)} = b - Ax^{(k)}$.

Therefore, the equation above becomes

$$-A^{-1}\big(e^{(k+1)} - e^{(k)}\big) = w(D + wL)^{-1}e^{(k)}$$

$$\Rightarrow e^{(k+1)} = (I - wA(D + wL)^{-1})e^{(k)}$$

The iterations converge if $||I - wA(D + wL)^{-1}|| < 1$

However, I believe it is unclear how to solve something like this. An alternative method is to say that the spectral radius of $G = (D + wL)^{-1}\big((1-w)D - wU\big)$ is less than 1. This is a statement made in the book. Note that I defined G in the line above.

What we need is a bound on λ_1 in terms of w. One way to do this is to say:

$$\Pi_i \lambda_i = \det(G) \leq \lambda_1^n = \rho^n$$

Where n is the dimensionality of G.

$$\det(G) = \det(D + wL)^{-1}\det((1-w)D - wU)$$

We know that the determinant of a triangular matrix is just the product of the diagonal entries.

Hence,

$$\det(G) = \det(D)^{-1}\det\big((1-w)D\big) = (1-w)^n$$

Using this, we see that $(1-w)^n \leq \rho^n$.

$$\Rightarrow \rho \geq |1 - w|$$

If $w \notin (0,2)$, then $|1 - w| \geq 1$ and hence $\rho \geq 1$ which means that the algorithm does NOT converge, or equivalently, diverges.

Computer Problem 11.4

Use method of lines and any ODE solve of your choice to solve

$$u_t = -u_x$$

$$x \in [0,1]$$

$$t \geq 0$$

initial conditions:

$$u(0,x) = 0$$

boundary conditions:

$$u(t,0) = 1$$

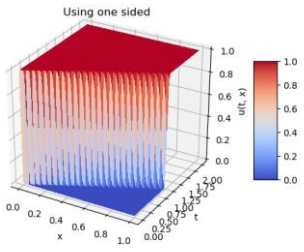
Over $t \in [0,2]$, plot $u(t,x)$ vs t,x (3D surface plot). Try both one sided and centered finite difference for spatial discretization. The actual solution is a step function of height 1 moving toward positive X with velocity 1.

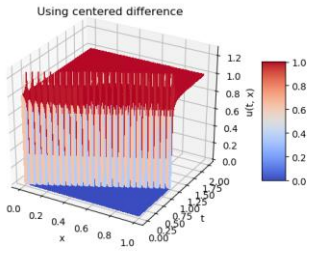
Does either scheme get close? Describe the difference between computed solutions. Which solution is smoother? Which is more accurate?

Answer

The code is in the file cp.11.4.hhabeeb2.py

Both schemes get close if we're careful about the derivatives at the boundary conditions.





Since centered difference is non-zero at two time steps for each point, it looks less smooth than the one using a one-sided derivative.

This result is different from the general expectation because the derivatives are 0 for most of the time. Hence, only the points at (t,x) at which derivative is non-zero is relevant to us. In this case, where the actual (analytical) derivative is a dirac delta, the one-sided derivative works better.

Code was implemented in python. The associated file is cp.11.4.hhabeeb2.py. It can be run simply as python cp.11.4.hhabeeb2.py [nx=1000, nt=1000]. It uses scipy's odeint function to solve a system of linear ODEs.

Computer Problem 11.12

~~I am using the second edition of the course textbook. I opted for extra homework. I opted to do problems exercise 11.1, exercise 11.7, computer problem 11.4, computer problem 11.12, computer problem 11.13.~~

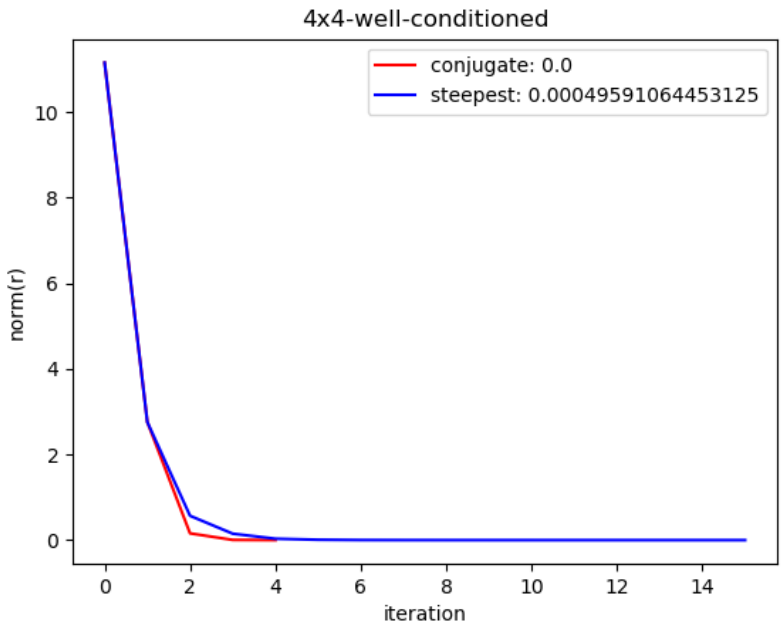
Implement steepest descent and conjugate gradient methods for solving symmetric positive definite linear systems. Compare their performance, both in rate of convergence and in total time. Use a representative sample of test problems – well conditioned and ill-conditioned too.

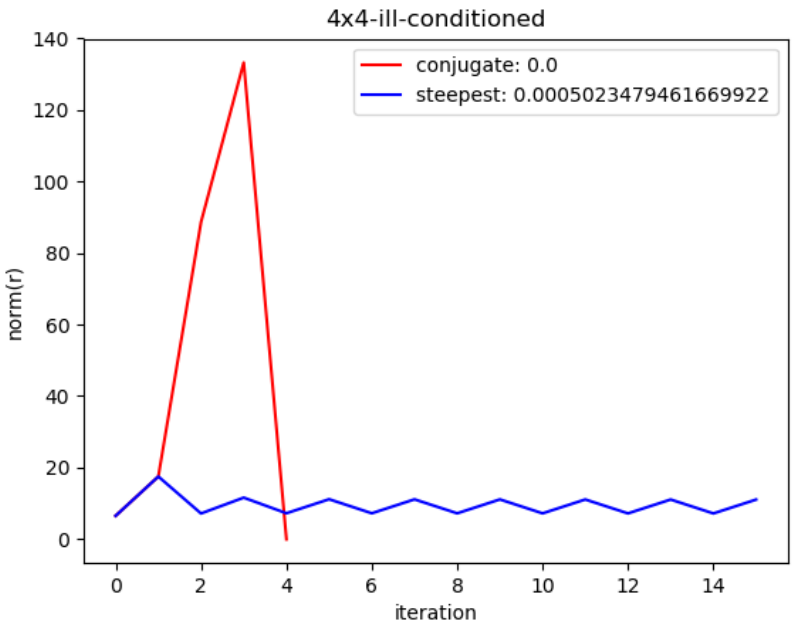
How does the rate of convergence compare with theoretical convergence rate?

Answer

The code is in the file cp.11.12.hhabeeb2.py

I have not yet conducted extensive testing with large matrices. However, the effect of ill conditioning is clear from the following two plots





The numbers in the legends are the time taken. Since the matrix is small, its instantaneous. We see that conjugate gradient increases error significantly but then fixes itself shortly. Steepest descent on the other hand, oscillates in a bounded fashion. Since it is essentially optimizing a quadratic function, conjugate descent converges in n iterations. In this case, 4. However, steepest descent is only $O(n^{-1})$ convergence if the matrix is well conditioned. Hence it is significantly slower.

Computer Problem 11.13

Implement the gauss siedel method for solving $n \times n$ systems $Ax = b$ where A is a matrix resulting from a finite difference approximation to the one dimensional Laplace equation on an interval which BVs of 0. Thus A is a tridiagonal matrix with 2s on the diagonal and -1s off diagonal. x represents the solution at interior mesh points. Take $b = 0$ so that the solution is $x = 0$. As the starting guess, take

$$x_j = \sin\left(\frac{jk\pi}{n+1}\right), \quad j = 1, \dots, n$$

For any value of k, the starting point is a discrete sample of a sine wave with frequency dependent on k. Observing results of gauss-siedel iterations for various values of k will tell us about the relative speeds at which components of error of various frequencies are damped out.

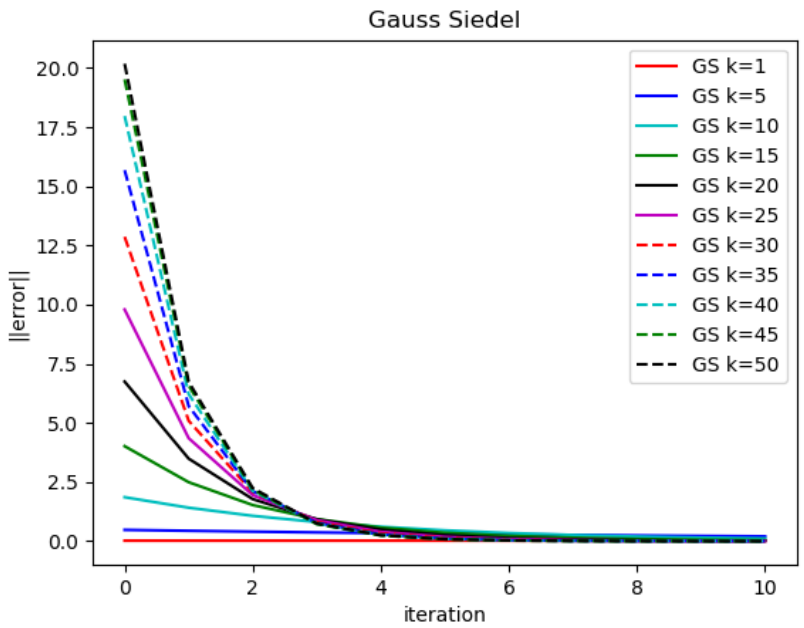
With $n = 50$ perform the experiment for $k = 1,5,10, \dots,25$. For each value of k, make a plot of the solution x at the starting value and for each iteration of the first 10 iterations. For which values of k does the error damp out slowly/rapidly?

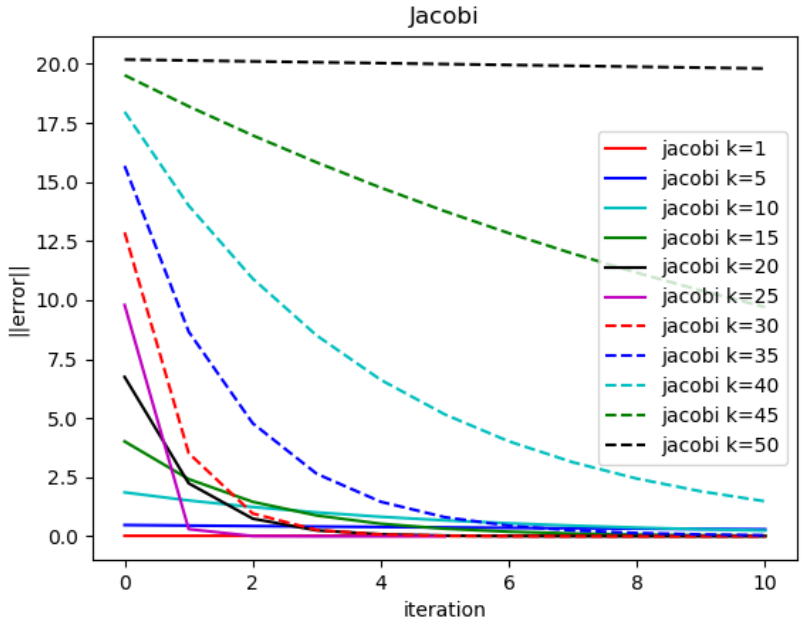
Repeat for jacobi iteration. How does error behave compared to gauss siedel? Is Gauss siedel smoother?

Answer

The code is in the file cp.11.13.hhabeeb2.py

Since plotting x vs j vs $iteration$ vs k is too cumbersome. Hence, to reach similar conclusions, we plot $||x||$ vs k vs $iteration$ for both methods.





It looks as though the lots for $k > \frac{n}{2}$ don't exactly behave like the ones with $k < \frac{n}{2}$. I suspect this is because of numerical instability and requires further investigation.

However, as expected, gauss-siedel converges better than jacobi. The error in jacobi decreases as rapidly, but is prone to larger instability. As observed by the plot for $k = 50$.

The next step would involve fixing the instability, which is likely in the matrix-vector multiplication.

Gauss Siedel behaves more similarly for various k than jacobi. Hence, it is "smoother", as section 11.5.7 puts it. It reduces components of error with different frequencies at approximately the same rate.