CS319: Scientific Computing (with C++)

Niall Madden (Niall.Madden@NUIGalway.ie)

CS319 Lab 6: Solving Linear Systems I

30 March 2021

Our goal is to improve our understanding of classical iterative solvers for linear systems of equations. We will reimplement the Jacobi solver from Weeks 7 and 8, but without the overloaded operators from the `Vector` and `Matrix` classes, and then implement a Gauss-Siedel solver. However, you don't have to submit any work this week.

In our next (and final) lab, after Easter, we'll return to classes and operator overloading to simplify the Gauss-Siedel solver.

Part 1: Jacobi's Method

In class we discussed Jacobi's method for solving a linear system of equations, and presented an example of how it could be implemented. This implementation is in `02Jacobi.cpp` from `Week08`. It makes use of the code in `Vector08.h`, `Matrix08.h`, `Vector08.cpp`, and `Matrix08.cpp`. To run it, you'll need to create a project with all 5 files. They can be downloaded from https://bitbucket.org/niallmadden/2021-cs319 or http://www.maths.nuigalway.ie/~niall/CS319/Week08/
To verify that you understand how the code works, adapt the `02Jacobi.cpp` program to solve the following problem:

$$6x_1 - 2x_2 + x_3 + x_4 = -7 \tag{1}$$
$$x_1 + 7x_2 - 2x_3 + x_4 = -9 \tag{2}$$
$$-x_1 + 2x_2 + 8x_3 - 2x_4 = 4 \tag{3}$$
$$-x_1 + x_2 + x_3 + 9x_4 = 20 \tag{4}$$

(Remember: mathematically, we usually index vectors and matrices from 1 to N, but in C++ we index them from 0 to $N-1$).

Jacobi's method is designed to solve linear system of N equations in N unknowns: *find* $x_1, x_2, \ldots, x_N$, *such that*

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1N}x_N &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2N}x_N &= b_2 \\ &\vdots \\ a_{N1}x_1 + a_{N2}x_2 + \cdots + a_{NN}x_N &= b_N. \end{aligned}$$

We expressed this as a matrix-vector equation: *Find* $\mathbf{x}$ *such that*

$$A\mathbf{x} = \mathbf{b},$$

where A *is a* $N \times N$ *matrix, and* $\mathbf{b}$ *and* $\mathbf{x}$ *are (column) vector with* N *entries.*

However, a simpler, if less elegant presentation is possible: choose $\mathbf{x}^{(0)}$ and set

$$x_1^{(k+1)} = \frac{1}{a_{11}}(b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - \cdots - a_{1N}x_N^{(k)})$$
$$x_2^{(k+1)} = \frac{1}{a_{22}}(b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - \cdots - a_{2N}x_N^{(k)})$$
$$\vdots$$
$$x_N^{(k+1)} = \frac{1}{a_{NN}}(b_N - a_{N,1}x_1^{(k)} - \cdots - a_{N,N-1}x_{N-1}^{(k)})$$

This can be programmed with two (or so) nested `for` loops, rather than explicit matrix multiplication.
Implement this version of Jacobi's method, using `for` loops, but without the `AddVec` or `MatVec` functions from `02Jacobi.cpp`. Verify that you get the same results as with the original `Jacobi` solver.
(You can, if you choose, use `Vector` and `Matrix` objects to store the data, but it is not essential. Similarly, it is OK to use `AddVec` and `MatVec` to compute the residual, if you wish).

Part 2: Improving Jacobi

Jacobi's method is not particularly efficient. Heuristically, you argue that it could be improved as follows. In Jacobi's method, we compute $x_1^{(k+1)}$ from

$$x_1^{(k+1)} = \frac{1}{a_{11}}(b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - \cdots - a_{1N}x_N^{(k)})$$

We expect that it is a better estimate for x_1 than $x_1^{(k)}$.
Next we compute

$$x_2^{(k+1)} = \frac{1}{a_{22}}(b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - \cdots - a_{2N}x_N^{(k)})$$

However, here we used the "old" value $x_1^{(k)}$ even though we already know the new, improved $x_1^{(k+1)}$. That is, we could use

$$x_2^{(k+1)} = \frac{1}{a_{22}}(b_2 - a_{21}x_1^{(k+1)} - a_{23}x_3^{(k)} - \cdots - a_{2N}x_N^{(k)})$$

More generally, in Jacobi's method we set

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1, j\neq i}^{N} a_{ij} x_j^{(k)}\right).$$

The Gauss-Seidel method uses

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{N} a_{ij} x_j^{(k)}\right).$$

Implement this method as new function called `GaussSeidel`. Verify that it is more efficient than the Jacobi method, in the sense that fewer iterations are required to achieve the same level of accuracy.

Part 3: A test problem

You should now have working code for the Jacobi and Gauss-Seidel methods. Test these functions for problems of at least size $N = 10$. According to the theory, these methods are guaranteed to converge if the system matrix is *diagonally dominant*. This means that, in each row, the magnitude of the diagonally entry, $|a_{ii}|$, must be greater than the sum of the other terms: i.e.,

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|.$$

Keep this in mind when you are choosing a test problem.
Ideally, your programme should

- Generate a random matrix that is diagonally dominant. (Have a look at `03TestOverload.cpp` from Week 8 to see an example of a vector with random entries).
- Choose a simple solution vector $\mathbf{x}$, e.g., $x = (1, 0, 1, 0, 1, \dots)$.
- Set $\mathbf{b} = A\mathbf{x}$
- Use your Gauss-Seidel solver to compute and estimate for $\mathbf{x}$.
- Compute how close this estimate is to the true solution.

Part 4: Exercises

You don't have to submit any work this week. Nonetheless, you should complete the following exercises before next week.

1. Produce a working program that implements the Jacobi and Gauss-Seidel methods.
2. Even though most books present the Gauss-Seidel method as an improvement upon Jacobi's, it is actually the (slightly) older of the two methods (it is from 1826, whereas Jacobi's is from 1845). Jacobi's preferred his slower method, however, because it is easier to parallelise. Explain why this is the case.