CS319: Scientific Computing (with MATLAB)

Lab 5: Solving Linear Systems, Part 1

Week 9, 2023

Niall Madden (Niall.Madden@UniversityOfGalway.ie)

- **What to do:** adapt a MATLAB live script to investigate an iterative method for solving linear systems of equations.
- **What has this got to do with Scientific Computing?** The solution of linear systems is absolutely central to scientific computing.
- **What to upload:** Nothing this week. But we'll develop this more next week.

1: An example

We'll start by choosing a problem to solve:

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

We'll write this as a matrix-vector equation, $Ax = b$, where

$$
A = \begin{bmatrix} 6 & -2 & 1 & 1 \\ 1 & 7 & -2 & 1 \\ -1 & 2 & 8 & -2 \\ -1 & 1 & 1 & 9 \end{bmatrix}, \quad X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4. \end{bmatrix}, \quad \text{and } b = \begin{bmatrix} -7 \\ -9 \\ 4 \\ 20. \end{bmatrix}
$$

Section 1 of the live script, `Lab5_JacobiV01.mlx` defines the matrix A and vector b, and solves for X using the `mldivide` ("backslash") operator. You should find that $X = [-2, -1, 1, 2]$.

2: Jacobi's method: Motivation

For "reasons" (that I will explain in class), the algorithm used by `mldivide` isn't always suitable. One of the simplest alternatives is **Jacobi's method**. It is an example of an **iterative method:** we make some initial guess, and then continually try to improve it until it is "good enough".

We'll start by writing down the general linear system: of N equations in N unknowns: *find* $x_1, x_2, \ldots, x_N$, *such that*

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

We expressed this as a matrix-vector equation: *Find* X *such that*

$$AX = \mathrm{b}, \tag{3}$$

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

2: Jacobi's method: Motivation

We can motivate Jacobi's method as follows. Suppose I happen to know that values of x_2, x_3, ..., x_N, but not x_1. Then we could compute x_1 using, for example, the first equation in (2)

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1N}x_N = b_1$$

rearranged as

$$x_1 = \frac{1}{a_{11}}(b_1 - a_{12}x_2 - a_{13}x_3 - \cdots - a_{1N}x_N) \tag{4}$$

The idea is that if we have **estimates** for x_2, x_3, ..., x_N, then we can use (4) to get an estimate for x_1.

Similarly, we can use the second equation in (2), rearranged as

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

to get an **improved** estimate for x_2.

The process can be repeated for x_3, x_4, ..., x_N. This is implemented in Section 2 of the live script.

3: Jacobi's method: in general

In its general form, the method can be stated as follows:

- Choose any vector $\mathrm{x}^{(1)}$.
- For $k = 2, 3, \dots,$ set

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

3: Jacobi's method: in general

This can be written more succinctly as: *for* $k = 2, 3, \ldots,$ *set...*

$$(5) \quad x_i^{(k)} = \big(b_i - \sum_{j=1:(i-1),(i+1:N)} a_{ij} x_j^{(k-1)}\big)/a_{ii}, \qquad i = 1, \ldots, N.$$

This is implemented in the live script, using three nested `for` loops:

- The outer loop iterates over $k = 2:6$. Solution vectors are stored in a cell array.
- The middle loop iterates over $i = 1:N$. Solution vectors are stored in a cell array.
- The inner loop computes the sum over j.

4: Exercises

Q1. The first improvement you will make to the live script is to test it for a larger problem, e.g., with $N = 10$. It transpires that the Jacobi's method will work exactly when A 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.,

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

Modify Section 1 of the Live Script so that it constructs a random matrix that has this property. *Hint: if you use the* `rand()` *function to create the matrix A, then all then entries of A will be between 0 and 1. So the off-diagonal terms can't sum to more than $N - 1$. Setting the diagonal entries to sufficiently large will ensure that A is diaogonally dominant.*

4: Exercises

Q2. The choose a solution vector e.g., $X = (1, 0, 1, 0, 1, \dots)$, and set $b = AX$ to get the right-hand side.
Verify that Jacobi's method appears to converge. That is: $\|X - x^{(k)}\|$ decreases as k increases.

Q3. The iteration version of Jacobi's method, given in Section 3, uses a `for` loop to compute the first 6 iterations. Rewrite this using a *`while`* loop, so that it will continue to iterate until some user-chosen tolerance is reached, or until some specific maximum number of iterations is exceeded. *Hint: you did something like this in Lab 2.*

4: Exercises

Q4. Jacobi's method can be expressed even more efficiently, in a matrix format.

- Let D be the diagonal matrix whose entries are the diagonal entries of A. That is, $d_{ii} = a_{ii}$ for $i = 1, 2, \ldots N$. In MATLAB this can be done in one, slightly obscure, line: `D = diag(diag(A))`. Review the notes from Week 5 to see why this works.
- Set $T = D - A$.
- Now **Jacobi's method** can be written as: choose $\mathrm{x}^{(0)}$ and set

$$x^{(k)} = D^{-1}(b + Tx^{(k-1)}). \quad (6)$$

This eliminates two for-loops from the implementation, and makes the code shorted and easier to read.

(a) Can you convince yourself that (6) is the same as (5)?

(b) Since D is diagonal, it is actually OK to compute its inverse using the `inv()` function. But can you construct D^{-1} without even constructing D?

4: Exercises

Q5. Next week, we will be interested in how quickly Jacobi's method converges, and what influences convergence. Modify the code to record the error for each k in an array, and plot it. Try plotting using the `semilogy()` function. If it works, you should see a straight line. Can you explain why?
We'll also implement a similar scheme: the Gauss-Seidel method. Read up about it.