CS319: Scientific Computing

Linear Equation Solving

Dr Niall Madden

Week 9: Friday March, 2025

Slides and examples: https://www.niallmadden.ie/2425-CS319

0. Outline

Slides and examples:

https://www.niallmadden.ie/2425-CS319

1. Systems of equations

As mentioned in Part 1, probably the most ubiquitous problem in scientific computing is the solution of (large) systems of simultaneous equations.
Here is an example (previously seen) where we want to solve a linear system of 3 equations in 3 unknowns: *find* x_1, x_2, x_3, *such that*

$$\begin{aligned} 3x_1 + 2x_2 + 4x_3 &= 19 \\ x_1 + 2x_2 + 3x_3 &= 14 \\ 5x_1 + x_2 + 6x_3 &= 25 \end{aligned}$$

This can also be expressed as a **matrix-vector equation**; $Ax = b$, where

$$\underbrace{\begin{pmatrix} 3 & 2 & 4 \\ 1 & 2 & 3 \\ 5 & 1 & 6 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}}_{x} = \underbrace{\begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}}_{b}$$

1. Systems of equations

More generally, the linear system of N equations in N unknowns: *find* $x_1, x_2, \ldots, x_N$, *such that*

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

This, as a **matrix-vector equation** is:

$$\begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1N} \\ a_{21} & a_{22} & \ldots & a_{2N} \\ \vdots & & \ddots & \vdots \\ a_{N1} & a_{N2} & \ldots & a_{NN} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{pmatrix}$$

1. Systems of equations

So, eventually, we need to be able to represent **vectors** and **matrices** as classes in our codes.
However, in the short term, we'll make do with just 1D and 2D arrays.

1. Systems of equations

There are several classic approaches for solving linear systems:

1. Gaussian Elimination, often expressed as LU-factorisation
2. Cholesky factorisation (which is LL^T for symmetric positive definite matrices)
3. Stationary Iterative schemes such as **Jacobi's method**, **Gauss-Seidel** and Successive Over Relaxation (SOR);
4. Krylov subspace methods, of which Conjugate Gradients is the best known;
5. Enhancements of the Methods 3 and 4, using preconditioning with, for example, MultiGrid and Incomplete LU-factorisation.
6. But never, *ever* solving $Ax = b$ by computing A^{-1}, unless $N = 2$ or $N = 3$.

Of the approaches listed above, Jacobi's is by far the simplest to implement, and so is the one we will study first.

1. Systems of equations

Some facts about the previously mentioned methods:

- What we call "Gaussian Elimination" was known in Chinese mathematics for at least 2,000 years.
- It was known in European mathematics since the 1500's, but it is believed that the method was so obvious, nobody bother writing it down!
- Eventually, Newton did in his personal notes, later published as *Arithmetica Universalis* in 1707.
- Gauss devised some notation for the symmetric case. But the name "Gaussian Elimination" came from some textbooks published in the 1950s, which (mis)-attributed the method to him.
- The modern version (LU-factorisation) is understood as a matrix-based method, since the work of Tadeusz Banachiewicz in the late 1930s. (Turing and von Neumann were pivotal too).

1. Systems of equations

However, Gaussian Elimination (GE), is an example of what is called a **direct method**. That means, we arrive at a solution after a fixed number of steps, and that the solution is exactly correct (if we can neglect round-off error...)
However, although it is very efficient, GE has many technical difficulties.

- In many applications, the system matrix has very structured, and that can be exploited to great effect. For example, A may be *sparse* (come back next week to learn what that means). But Gaussian Elimination may destroy the sparseness. As a result, it can be very memory intensive.
- It can be *ill-conditioned*, meaning that small errors (e.g., due to the use of floating point numbers) can be greatly magnified.

So we are not going to study it any more in CS319! Instead, we'll consider **iterative methods**

2. Iterative methods

The alternative to **direct methods** are **iterative methods**. These are algorithms which take an initial guess for the solution, and then refine it, again and again, to get better and better approximations of the solution.
That is, they compute a sequence

$$x^{(0)}, x^{(1)}, x^{(2)}, \dots,$$

where, if everything works correctly, $\lim_{n\to\infty} \|x - x^{(n)}\| = 0$.
That is, if we take an infinite number of iterations (and have exact arithmetic) we get the true solution.
However, we don't try to take an infinite number of iterations, for two reasons:

(a) We don't have a infinite amount of (CPU) time at our disposal;

(b) It is not necessary: modelling error or numerical error are not zero.

2. Iterative methods

In a sense, such iterative methods are similar to Newton's method we worked on in Labs 3+4:

- We take an initial guess;
- We repeatedly refine that;
- We stop when some measure of the error is small enough.

However, there are some differences too: in particular, Newton's method may fail if the initial guess is not sufficiently close to the actual solution. The methods we'll study will converge for *any* initial guess (though they may take longer if the initial guess is not particularly good).
However, we can easily evaluate how good an estimate for the solution is, by computing the **residual**: if $x^{(k)}$ is an estimate for the solution to $Ax = b$, the *residual* is $\|b - Ax^{(k)}\|$.

3. Jacobi's Method

Jacobi's Method is an example of an *iterative method*. It can be derived as follows (see notes on board).

3. Jacobi's Method

In summary, **Jacobi's method** is: *choose* $\mathrm{x}^{(0)}$ *and set*

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

This can be programmed with two (or so) nested `for` loops.

3. Jacobi's Method

An implementation is given in Lab 7: https://www.niallmadden.ie/2425-CS319/lab7/Jacobi-Lab7.cpp
It works as follows:

- The two-dimensional array `A` stores the coefficients for the left-hand side.
- The one-dimensional arrays `x` and `b` stores the true solution and left-hand side, respectively.
- The one-dimensional arrays `xk` and `xk1` represent the vectors $x^{(k)}$ and $x^{(k+1)}$.

3. Jacobi's Method

- It sets `A` and `b` to represent the problem

$$9x_1 + 3x_2 + 3x_3 = 15 \tag{1}$$
$$3x_1 + 9x_2 + 3x_3 = 15 \tag{2}$$
$$3x_1 + 3x_2 + 9x_3 = 15 \tag{3}$$

  The true solution is $x_1 = x_2 = x_3 = 1$.

- Five iterations of the Jacobi method are taken.
- The estimated solution after five iterations is outputted.

3. Jacobi's Method

In that lab you are asked to make various improvements...

4. Gauss-Seidel Method

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

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

4. Gauss-Seidel Method

See notes on board

4. Gauss-Seidel Method

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

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

In Lab 7, you are asked to implement this method, and 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.

More history

The historical links between the Jacobi and GS methods are a little vague, but a few things are known:

1. GS converges more efficiently, and also is actually easier to code. And requires less memory... (why?)
2. Both methods are very **tolerant of errors**. That is, if you are implementing one of the methods, and you make a arithmetic mistake, the method will still, eventually, yield a sensible solution (unlike GE...)
3. GS predates Jacobi, and, in some settings, Jacobi is considered an improvement upon GS.

Point 2+3 may be clearer if you recall that these methods were designed to be implemented by human "computers", often working in parallel...

5. Matrix-based Jacobi

To recap, we are solving a linear system of N equations in N unknowns: *find* $x_1, x_2, \dots, x_N$, *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 express this as a matrix-vector equation: *Find* x *such that*

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

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

5. Matrix-based Jacobi

Then **Jacobi's method** is: choose $\mathrm{x}^{(0)}$ and set

$$\begin{aligned}
x_1^{(k+1)} &= \frac{1}{a_{11}}(b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - \cdots - a_{1N}x_N^{(k)}) \\
x_2^{(k+1)} &= \frac{1}{a_{22}}(b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - \cdots - a_{2N}x_N^{(k)}) \\
&\vdots \\
x_N^{(k+1)} &= \frac{1}{a_{NN}}(b_N - a_{N1}x_1^{(k)} - \cdots - a_{N,N-1}x_{N-1}^{(k)})
\end{aligned}$$

There is also a matrix-version of this iteration. We set D and T to be the matrices

$$d_{ij} = \begin{cases} a_{ii} & i = j \\ 0 & \text{otherwise.} \end{cases} \qquad t_{ij} = \begin{cases} 0 & i = j \\ -a_{ij} & \text{otherwise.} \end{cases}$$

5. Matrix-based Jacobi

So $A = D - T$. Then *Jacobi's method* can be written neatly in matrix form:

$$x^{(k+1)} = D^{-1}(b + Tx^{(k)}). \tag{4}$$

. . . .

The significance for us is:

- We should implement C++ classes for vectors and matrices;
- We need to "overload" the assignment operator `=` for `Vector`s.
- We should overload the `+` and `-` operators for `Vector` addition and subtraction.
- We need to overload the `Matrix`-`Vector` multiplication operator.

5. Matrix-based Jacobi

If we do all that, we can implement the Jacobi method "just"[1] a few lines:

```
while ( (Rnorm > TOL) && (count<max_its))
{
  count++;
  xk1 = Dinv*(b+T*xk); // set x = inverse(D)*(b+T*x)
  R = A*xk-b;
  Rnorm = R.norm();
  xk = xk1;
}
```

Of course, this depends on already having defined the matrices `T` and `Dinv` first.

[1] Investing time and energy in developing data structures and methods which then allow us to succinctly implement algorithms is a touch-stone of scientific computing.

6. Triangular systems

Next we want to consider a matrix-based GS method. First we consider triangular systems.
Some systems of equations are very easier to solve than others.
Suppose the system is $L\mathrm{x} = \mathrm{b}$, but L is a lower triangular matrix.
The associated system of equations looks like this:

$$\begin{aligned}
l_{11}x_1 \qquad\qquad\qquad\qquad &= b_1\\
l_{21}x_1 + l_{22}x_2 \qquad\qquad\qquad &= b_2\\
l_{31}x_1 + l_{32}x_2 + l_{33}x_3 \qquad &= b_2\\
&\vdots\\
l_{N1}x_1 + l_{N2}x_2 + \cdots + l_{NN}x_N &= b_N.
\end{aligned}$$

To solve this,

- first set $x_1 = b_1/l_{11}$.

6. Triangular systems

- Now substitute this into the second equation to get $x_2 = (b_2 - l_{21}x_1)/l_{22}$.
- Next we use $x_3 = (b_3 - l_{31}x_1 - l_{32}x_2)/l_{33}$,
- and so on.

In fact, this is quite like Jacobi's method, except we don't have to iterate...
Suppose we code an implementation for a (lower) triangular matrix, `L`. Then, since `L*x= b`, it is reasonable to write `x=b/L`, where the "forward slash" operator is implementing back-substitution as just described.

7. Matrix-based Gauss-Seidel

Recall that the **Gauss-Seidel method** is choose $x^{(0)}$ and set

$$\begin{aligned}
x_1^{(k+1)} &= \frac{1}{a_{11}}(b_1 - a_{12}x_2^{(k+1)} - a_{13}x_3^{(k)} - \cdots - a_{1N}x_N^{(k)}) \\
x_2^{(k+1)} &= \frac{1}{a_{22}}(b_2 - a_{21}x_1^{(k+1)} - a_{23}x_3^{(k+1)} - \cdots - a_{2N}x_N^{(k)}) \\
&\vdots \\
x_N^{(k+1)} &= \frac{1}{a_{NN}}(b_N - a_{N1}x_1^{(k+1)} - \cdots - a_{N,N-1}x_{N-1}^{(k+1)})
\end{aligned}$$

In the same way as we did for Jacobi's method, we can write this in a succinct matrix-vector form: we set L and U to be the matrices

$$l_{ij} = \begin{cases} a_{ij} & i \geq j \\ 0 & \text{otherwise.} \end{cases} \qquad u_{ij} = \begin{cases} 0 & i \geq j \\ -a_{ij} & \text{otherwise.} \end{cases}$$

7. Matrix-based Gauss-Seidel

So $A = L - U$. Then the *Gauss-Seidel method* can be written as

$$Lx^{(k+1)} = b + Ux^{(k)}. \qquad (5)$$

Note that this involves solving a linear system where L is the coefficient matrix.

8. Some further details

- Writing $A = L - U$ is called a **regular splitting** of A.
- **Important** For either Jacobi or GS to converge, it is sufficient (but not necessary) for A to be "SDD": strictly diagonally dominant (meaning...)
- Being "SDD" just ensures that the spectral radius of the iteration matrix a is less than 1.