CS319: Scientific Computing (with MATLAB)

Solving Linear Systems (v2)

Niall Madden

Week 9: **9am**, and **4pm**, 08 March 2023

Useful reading:

- Learning MATLAB, Sections 2.5 (brief overview), 7.1 and 7.2
  https://doi-org.nuigalway.idm.oclc.org/10.1137/1.9780898717662
- The MATLAB Guide, Chapters 9 (Linear Algebra) and 15 (Sparse Matrices) of
  https://doi-org.nuigalway.idm.oclc.org/10.1137/1.9781611974669

Reminder of CS319 Assessment

(This is taken from Slide 7 of the notes from Week 1)
The final grade for CS319 is be based on

- **Four lab assignments** (40%). *You've completed 3 of these.*
- a mid-semester open-book test (20%) (Week 7).
- The project (40%)

This module does not have an end-of-semester exam.

This week...

1. Projects
2. 1: About Lab 5
   - Jacobi's method
   - Implementation
3. 2: Sparse Matrices (recall)
4. 3: Sparse matrices in MATLAB
5. 4: Linear Solvers in MATLAB
   - An example problem
6. 5: Direct solvers

Projects

The slides for this section are at https://www.niallmadden.ie/2223-CS319/2223-CS319-Projects.pdf

1: About Lab 5

One of our goals in this module is to investigate ways of solving N simultaneous 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* x *such that*

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

where A *is a* $N \times N$ *matrix, and* b *and* X *are (column) vector with* N *entries.*
We could do this with **Gaussian Elimination** (or LU-factorization, etc). But instead we proposed using *Jacobi's method*, probably the easiest to code.

The idea is to choose an initial "guess" vector, which we denote $\mathrm{x}^{(1)}$.
The we try to compute an improved estimate, $\mathrm{x}^{(2)}$.
And we improve that again, to get $\mathrm{x}^{(3)}$.
Eventually, we have a sequence of estimates

$$\{x^{(1)}, x^{(2)}, x^{(3)}, \ldots, x^{(k)}, \ldots\}$$

If could do this an infinite number of times, then

$$\text{as } k \to \infty, \text{ we get } \mathrm{x}^{(k)} \to \mathrm{X}.$$

But in practice, we just iterate until $\mathrm{x}^{(k)}$ is "close enough" to X.

The algorithm (i.e., the method of "improving" the $x^{(k)}$) comes from the observation that, since (for example)

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

then

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

So we can be optimistic that if $x_2^{(k)}$, $x_2^{(k)}$, ..., $x_N^{(k)}$ are a good estimates for x_2, x_2, ..., x_N, then

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

will be an even better one for x_1.

Applying the same idea to the rest of the equations, we get

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

This can be programmed with two (or so) nested `for` loops, as discussed in Lab 5. But it can also be expressed in a simple way, using matrices and vectors.

Now that we know the method, let us summarise the steps, so as to work out what standard operations on vectors and matrices we need.
We expressed the problem 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) vector with N entries.*
We then derived **Jacobi's method**: choose $\mathrm{x}^{(0)}$ and set

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

where $D = \mathrm{diag}(A)$ and $T = D - A$.
Looking at this we see that the fundamental operations are: **vector addition** and **matrix-vector multiplication**.

2: Sparse Matrices (recall)

- Last week we learned that a matrix is **sparse** of many of its entries are zero.
- For example, and $N \times N$ matrix might have only a fixed number $m \ll N$ non-zeros in each row.
- A matrix that is not sparse is said to be **dense**.
- The word **sparse** is also used to refer to a method of storage that tries to exploit the fact that there are so many zeros. Use a sparse format when
  - The memory required by the sparse format is less then the usual one (which is called **full**).
  - The expense of updating the sparse format is not excessive;
  - Computing a matrix-vector product ("`MatVec`") is faster for a sparse matrix than for full.
- The number of non-zeros in a matrix is usually denoted `NNZ`.
- The most basic sparse storage format is called **triplet** (see notes from Week 8). We also looked at **CCS**: **Compressed Column Storage**.

3: Sparse matrices in MATLAB

We'll finish going through the `CS319_Week08_SparseMatrices.mlx` Live script from Week 8.

4: Linear Solvers in MATLAB

In this section, we study the details of solving linear systems of equations in MATLAB.
Solvers can be classified as one of two types:

1. **Direct solvers**, which preform a fixed number of steps and give back the true answer (or, as close as possible, given that we have finite precision). Gaussian Elimination is the most famous example of this.
2. **Iterative solvers**, which take an initial guess and repeatedly try to improve it, until some tolerance is reached, or a maximum number of iterations is reached. The Jacobi method of Lab 5 is an example.

Both methods have their advantages and disadvantages:

To tests some methods, we need a good test problem. We need the problem to be

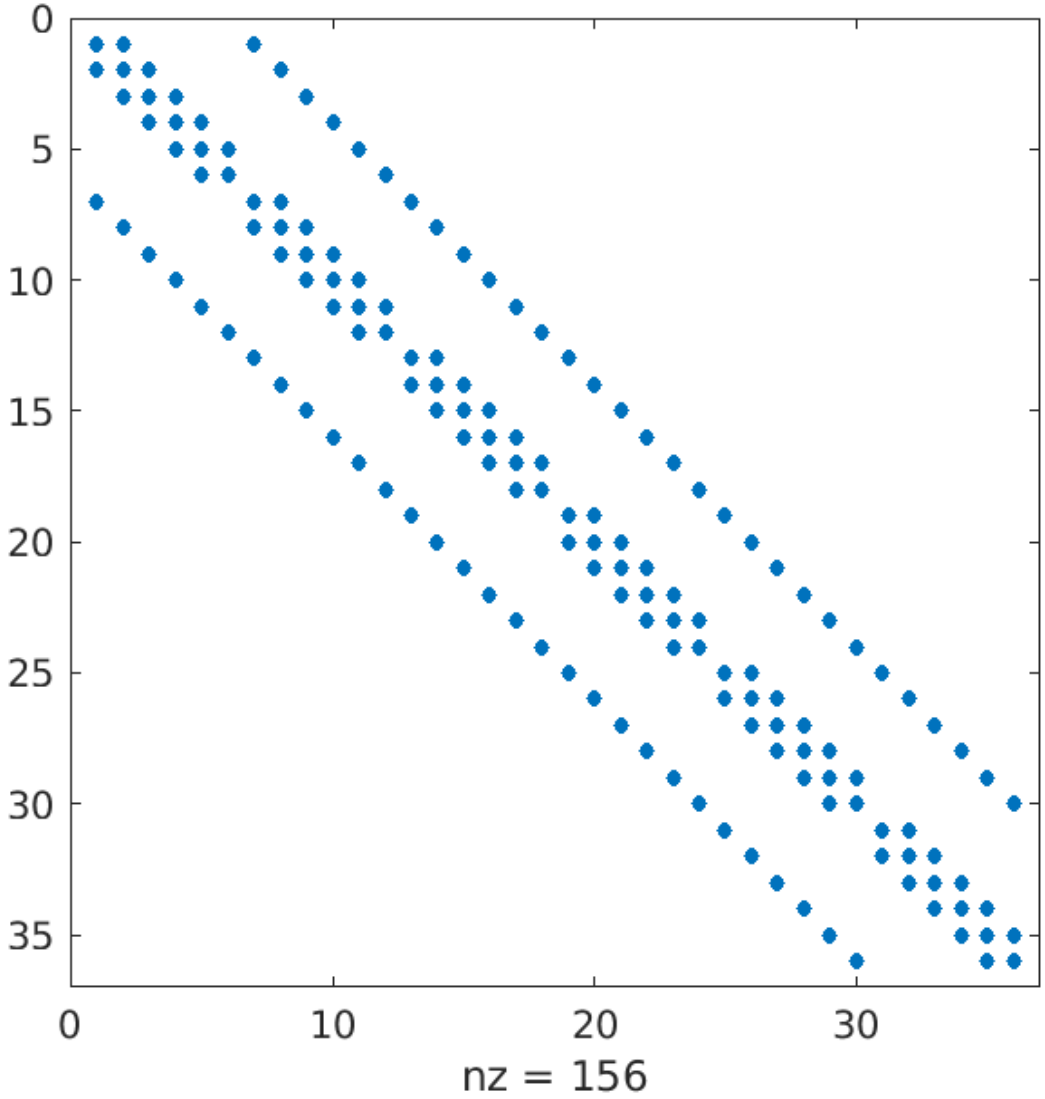
- Scalable: we can make up a version of any size we choose. In particular, we should be able to make it large.
- Not accidentally simple. For example, solving $Ax = b$ can be much easier when A is a tri-diagonal matrix, than when not.
- Sparse: most entries of A are zero.

The matrix we will make will be **banded**, with 9 non-zeros per row.

MakeTestProblem.m

```
function [A,b]=MakeTestProblem(n)
A1 = (-spdiags(ones(n,1),-1,n,n)/2 ...
   + spdiags(ones(n,1),0,n,n) ...
   - spdiags(ones(n,1),1,n,n)/4);
A = kron(A1,speye(n)) + kron(speye(n),A1) ;

x = ones(length(A),1);
b = A*x;
```


0
5
10
15
20
25
30
35
0
10
20
30
nz = 156

5: Direct solvers

The simplest way to solve a linear system in MATLAB is with the backslash operator:

```
>> x=A\b
```

To find out what is *really* happening we can turn on some diagnostics:

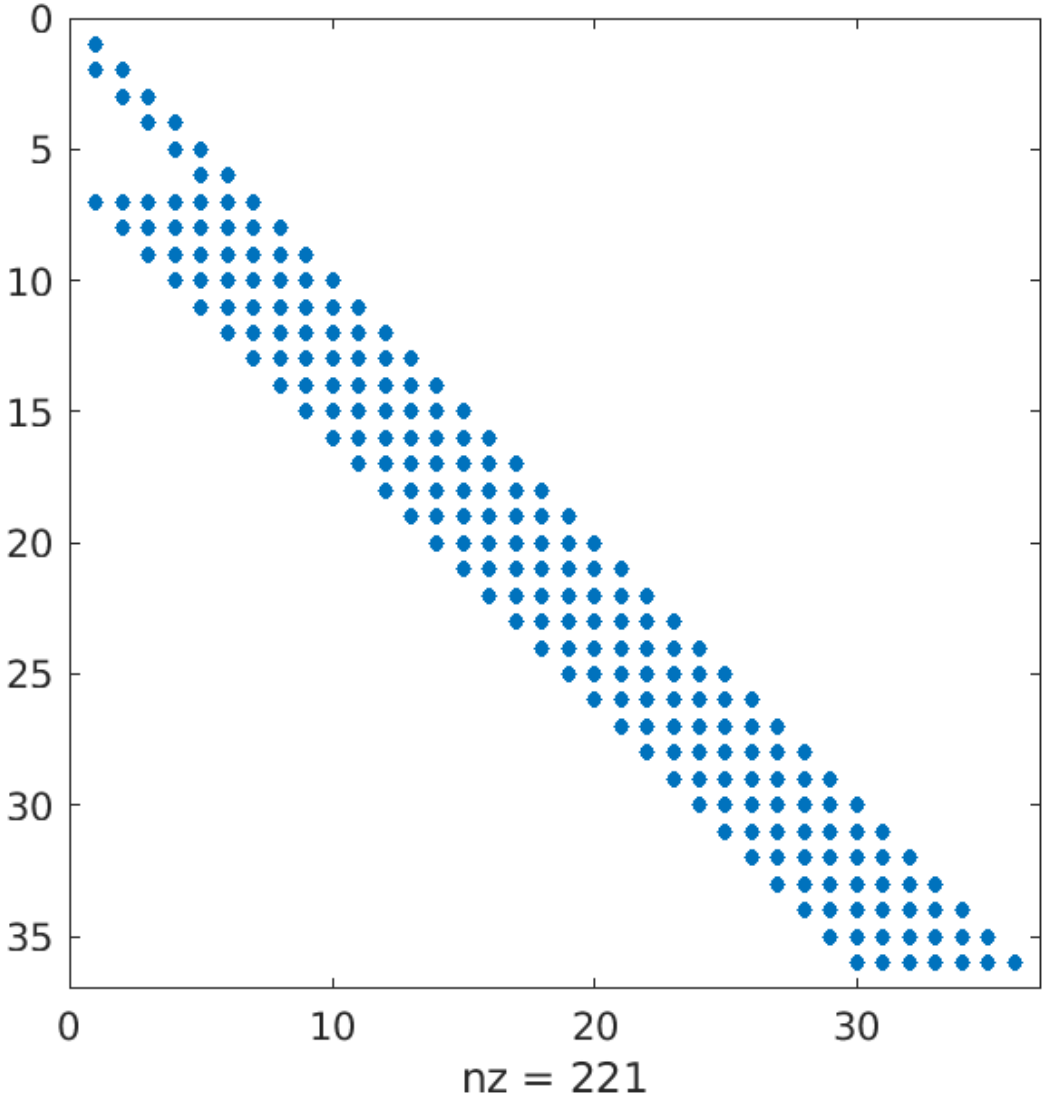
```
>> spparms('spumoni', 2)
>> x=A\b
sp\: bandwidth = 6+1+6.
sp\: is A diagonal? no.
sp\: is band density (0.366197) > bandden (0.500000) to try
     banded solver? no.
sp\: is A triangular? no.
sp\: is A morally triangular? no.
sp\: is A a candidate for Cholesky (symmetric, real
    positive or negative diagonal)? no.
sp\: use Unsymmetric MultiFrontal PACKagewith automatic
    reordering.
```

In that example, we use the (somewhat obscure) function `spparms()` which sets parameters for sparse matrix routines.
The `spumoni` option turns on monitoring.
If we set it to Level 2, we get more output (usually way too much). But with some effort, we can use the data to test the algorithms efficiency.

Unfortunately, time did not allow me to prepare a table showing how the memory demands grows.
But to get a sense of it, we'll learn how $Ax = b$ really works...

5: Direct solvers

5: Direct solvers



5: Direct solvers