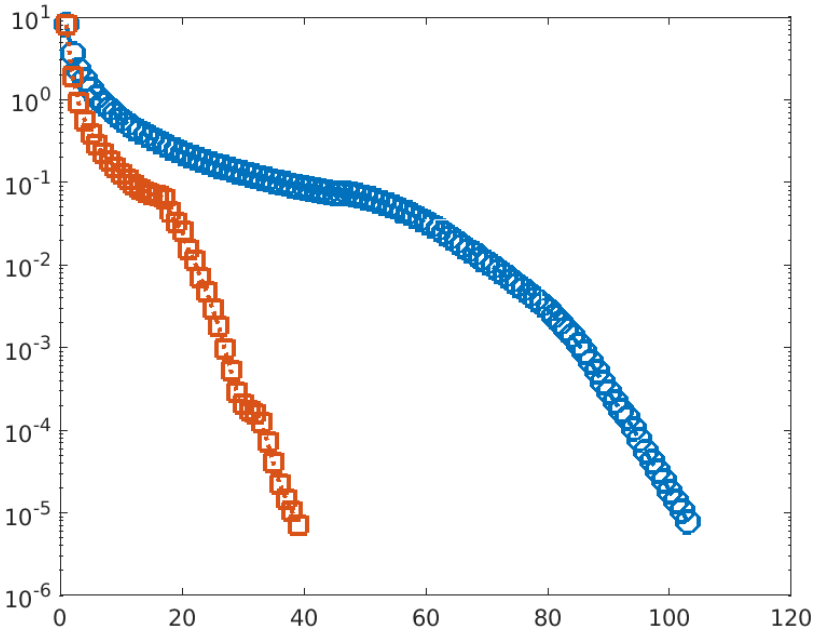
CS319: Scientific Computing (with MATLAB)

Direct and Iterative Solvers

Niall Madden

Week 10: **9am**, and **4pm**, 15 March 2023



This week...

1. 1. Projects (again)
2. 2: Linear Solvers in MATLAB
   - An example problem
3. 3: Direct solvers
4. 4. Iterative Solvers
   - minres
   - Conditioning
5. 5: Classes
   - Encapsulation
   - Implementation
   - `classdef`
6. 6. Example: MyStack
   - Example 1
   - Example 2: palindromes
7. 7. Overloading equality operator

3: Direct solvers

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, it is not easy to record this data in a systematic way. But we will look at the solve times...

3: Direct solvers

DoFs = "degress of freedom" = "number of unknows"

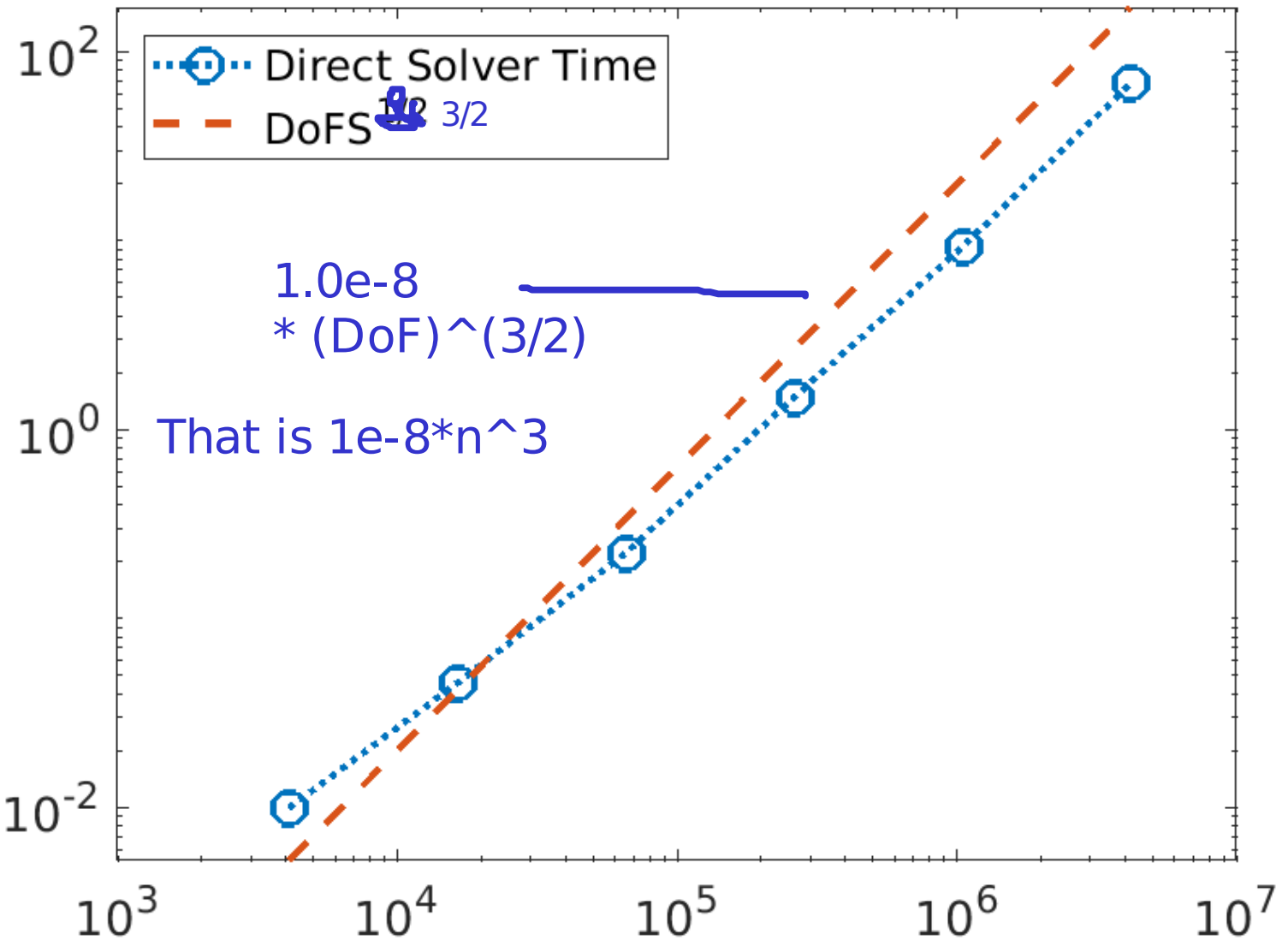
TimeDirectSolver.m

```
k=0;
for n=2.^(6:10)
    k=k+1;
    A = MakeTestProblem(n);
    b = randn(length(A),1);
    DoFs(k)=length(b);
    tic; x=A\b; SolveTime(k)=toc;
    fprintf("n=%4d, DoFs=%8d, Time(s)=%8.3f\n", ...
        n, DoFs(k), SolveTime(k));
end
loglog(DoFs, SolveTime, ':o', DoFs,2e-8*DoFs.^1.5,...
```

This will return the number of seconds elapsed since "tic".

```
n=   64, DoFs=     4096, Time(s)=    0.010
n=  128, DoFs=    16384, Time(s)=    0.046
n=  256, DoFs=    65536, Time(s)=    0.222
n=  512, DoFs=   262144, Time(s)=    1.484
n=1024, DoFs=  1048576, Time(s)=    9.276
n=2048, DoFs=  4194304, Time(s)=   68.213
```

3: Direct solvers



3: Direct solvers

However, these timing are not a completely reliable guide to the effort required to solve these systems of equations.

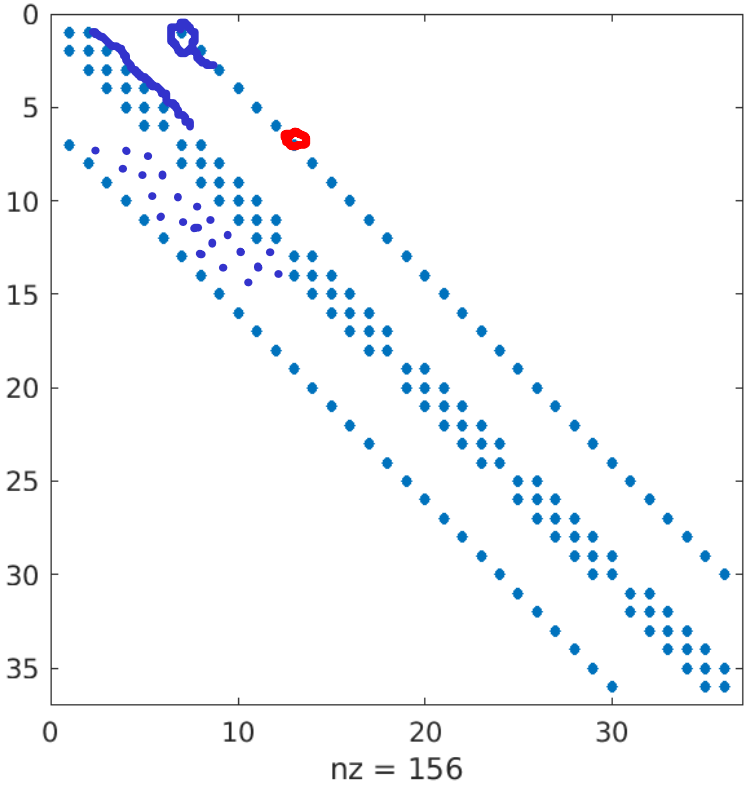
In particular, they were computed on a laptop with 8 cores. As the system gets larger, more and more of these are used by the solver (for smaller systems, it is not worth using more than one core).

Nonetheless, for this (very typical) problem, we estimate that the time is proportional to n^3. Here is why that is correct...

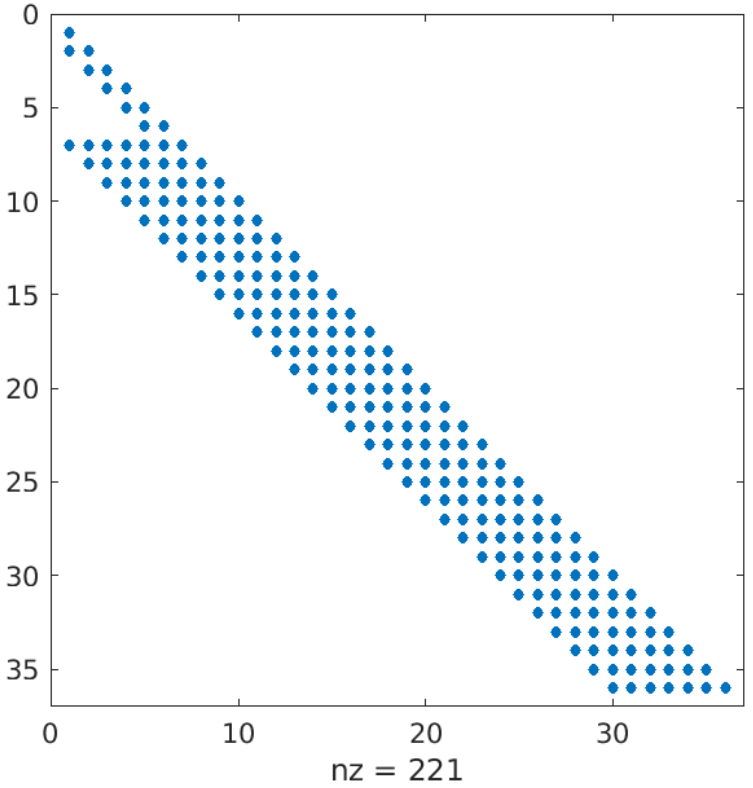
- We solve a system with n^2 rows;
- Each row has a band-width of n;
- When we apply Gaussian Elimination to the system, we "fill-in" the bands. That is, everything between the diagonals and the bands becomes (potentially) non-zero.

3: Direct solvers

Before row reduction



After row reduction



Every "dot" (i.e., nonzero) on the right must be computed.
There are n of them on each row, and there are n^2 rows.
So, there is a total of n^3 = (DoFs)^(3/2) computations to be completed.

3: Direct solvers

Under the hood, the algorithm does its best to minimise this fill-in (using some very interesting methods from graph theory). However, although this can often reduce the computational time by a significant factor, the rate of growth remains the same.

However, there is an even bigger problem...

Suppose $n = 1000$. So A has 1,000,000 rows, and each row as (up to) 5 non-zero entries. That will take about 15Mb to store.

But when we apply Gaussian Elimination, we will need to store n non-zeros per row: so about 15Gb.

These calculations are very rough, but the conclusion is correct: such methods have significant limitations when it comes to solving large problems.

4. Iterative Solvers

For large scale sparse problems, direct solvers are of little use. The alternative is to use iterative solvers. The very simplest are the Jacobi and Gauss-Seidel methods from Labs 5 and 6, but there are methods that are much more modern and efficient.

How they work is beyond the scope of this module (but roughly, think about the effectiveness of Newton's method compare the bisection method).

For this example, we'll use the "Minimum Residual" method, (`minres`, for short).

It is not hard to code: have a look at the Wiki Page: `https://en.wikipedia.org/wiki/Minimal_residual_method`.

However, `minres()` in only designed for symmetric matrices. In `MakeSymTestProblem.m` you'll find code that builds a **symmetric** banded matrix.

Like all iterative methods, `minres` computes a sequence of approximations to the solution of $Ax = b$.

You can apply it, in MATLAB, using `x = minres(A,b)`.

But when using any of these iterative method, you have to choose:

- A convergence tolerance, `tol`
- Maximum number of iterations, `maxit`

At each iteration (hopefully) the quality of the approximation improves. That is, if the kth iteration is $x(k)$ then we "hope" that $\|x - x^{(k)}\| < \|x - x^{(k-1)}\|$. (This inequality should be strict!)

In practical settings, we don't know x, so we can't just iterate until $\|x - x^{(k)}\| \leq tol$. Instead we use the residual as a proxy for the error.

That is, if in fact $x^{(k)} = x$, then $Ax^{(k)} = b$. Put another way, $\|b - Ax^{(k)}\| = 0$. That is unlikely to happen unless k is very small. So instead we stop when the **residual** $\|b - Ax^{(k)}\|$ is less than `tol`.

However, sometimes the method can be very slow, so we also need to specify the maximum value of k. This is `maxit`.

So now we know that we should choose values of `tol` and `maxits`, and then call `x = minres(A, b, tol, maxit);`

However, since convergence is precarious, we should also take some diagnostic information from `minres`.

That is we call

```
[x,flag,res, its, resvec] = minres(A,b, tol, maxits);
```

- `x` is the solution, i.e., $x^{(k)}$.
- `flag` is an integer that gives reasons why if failed (if it did). If `flag==0` the method "converged".
- `res` is $\|b - Ax^{(k)}\|$ for the final k.
- `its` is the number of iterations computed.
- `resvec` is a vector for all iterations, except the last one.

Example

Suppose we take $n = 16$ and make the symmetric test problem w. Then setting $tol = 10^{-4}$ we'll find that `minres()` needs 22 iterations. Each iteration is quick, but that is still suboptimal: if we double n then the number of iterations doubles too.

tol=1e-5

TimeIterativeSolver.m

```
Minres  n=   16, DoFs=     256, Time(s)=   0.003, Its=  26
Minres  n=   32, DoFs=    1024, Time(s)=   0.001, Its=  53
Minres  n=   64, DoFs=    4096, Time(s)=   0.007, Its= 102
Minres  n=  128, DoFs=   16384, Time(s)=   0.058, Its= 198
Minres  n=  256, DoFs=   65536, Time(s)=   0.323, Its= 382
Minres  n=  512, DoFs=  262144, Time(s)=   3.031, Its= 736
Minres  n=1024, DoFs= 1048576, Time(s)=  19.449, Its=1000
```

The method did not converge.

At a glance, that seems like the method is not very good, but in fact we we just need to understand:

- What determines the convergence;
- How it can be improved.

The speed of methods like `minres` depends on the **condition number** of the matrix, which is defined as

$$\kappa(A) := \|A\|\|A^{-1}\|.$$

In practice, never compute this, but we can estimate it. Use "conest()"

Then (roughly) at each iteration the residual changes by a factor or

$$\sqrt{\frac{\kappa(A)-1}{\kappa(A)+1}}$$

E.g., $\frac{999}{1001} = 0.9999$

Certainly, this factor is less than one (which is good), but not by much...

```
Minres  n=   16, DoFs=     256, Time(s)=   0.003, Its=  26
Minres  n=   32, DoFs=    1024, Time(s)=   0.001, Its=  53
Minres  n=   64, DoFs=    4096, Time(s)=   0.007, Its= 102
Minres  n=  128, DoFs=   16384, Time(s)=   0.058, Its= 198
Minres  n=  256, DoFs=   65536, Time(s)=   0.323, Its= 382
Minres  n=  512, DoFs=  262144, Time(s)=   3.031, Its= 736
Minres  n=1024, DoFs= 1048576, Time(s)=  19.449, Its=1000
```

There is a strategy, called **preconditioning** that tries to improve this situation.

Unfortunately, it is beyond the scope of what we can do today but, roughly...

- Find a matrix M that approximates A, but for which M^{-1} is easily computed.
- Also need that $\kappa(M^{-1}A) < \kappa(A)$.
- Instead of solving $Ax = b$ we solve $M^{-1}Ax = M^{-1}b$.

The simplest method use to take `M=diag(diag(A))`. (However, that is so easy, minres automatically does it).

The second simplest is called **Incomplete LU factorisation**

In the algorithm, we only ever have to compute matrix vector products. So "solving" inv(M)Ax = b, really just involves solving My=d .

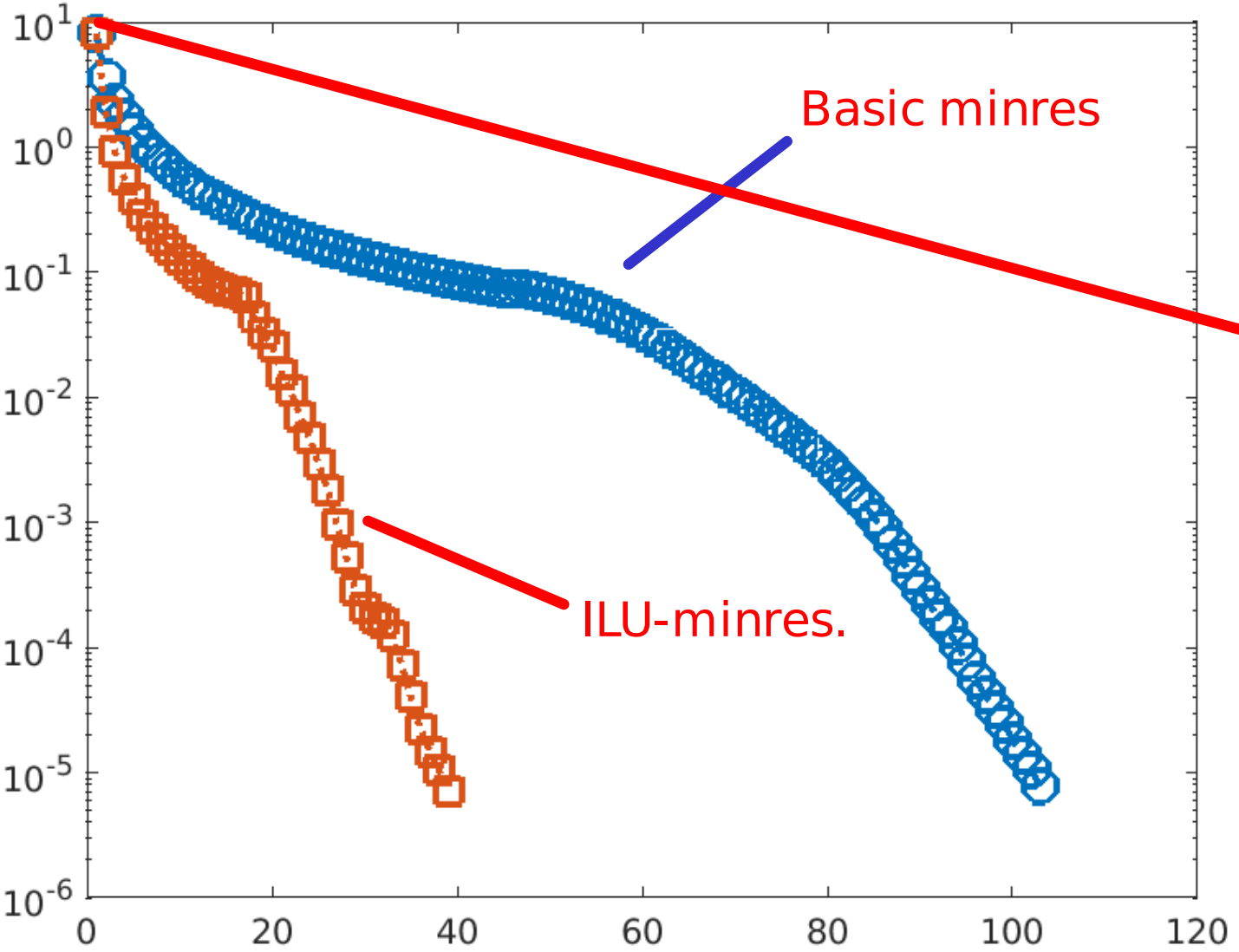
Min res: No preconditioning

```
n=   16, DoFs=     256, Time(s)=    0.003, Its=  26
n=   32, DoFs=    1024, Time(s)=    0.001, Its=  53
n=   64, DoFs=    4096, Time(s)=    0.007, Its= 102
n=  128, DoFs=   16384, Time(s)=    0.058, Its= 198
n=  256, DoFs=   65536, Time(s)=    0.323, Its= 382
n=  512, DoFs=  262144, Time(s)=    3.031, Its= 736
n=1024, DoFs= 1048576, Time(s)=   19.449, Its=1000
```

Min res with ILU

```
n=   16, DoFs=     256, Time(s)=    0.055, Its=  14
n=   32, DoFs=    1024, Time(s)=    0.008, Its=  24
n=   64, DoFs=    4096, Time(s)=    0.015, Its=  38
n=  128, DoFs=   16384, Time(s)=    0.059, Its=  67
n=  256, DoFs=   65536, Time(s)=    0.307, Its= 115
n=  512, DoFs=  262144, Time(s)=    2.514, Its= 220
n=1024, DoFs= 1048576, Time(s)=   19.601, Its= 420
```

Plots of the residuals



But what is ILU???

To date, the programming we have done in MATLAB has been *procedural*: it has been driven by algorithms and involved on very simple data structures.

Encapsulation

Idea: create a single entity in a program that combines data with the program code (i.e., functions) that manipulate that data.
In MATLAB, a description/definition of such entities is called a **class**, and an instance of such an entity is called an **object**.

Finished here.