Math 260: Python programming in math

Fall 2020

Sparse matrices (briefly):
Banded systems

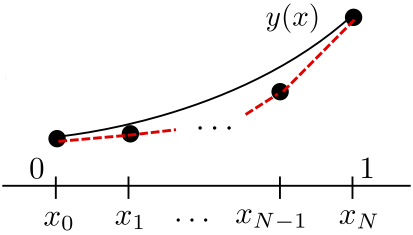
Here's an example of a linear algebra problem (with some ODE context)...

Suppose we want to solve, for $y(x)$, the **boundary value problem**

$$y'' - y = x, \qquad y(0) = 1,\ y(1) = e - 1$$

which has the solution $y(x) = e^x - x$.
Unlike an initial value problem, we can't just 'start' at an endpoint!

An approach is to approximate the function at mesh points x_j...



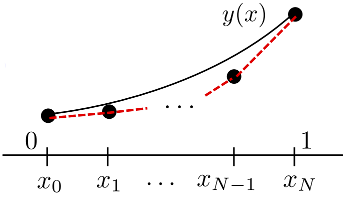
...and use the approximation

$$y''(x) \approx \frac{y(x+h) - 2y(x) + y(x-h)}{h^2}$$

for the second derivative.

Finite differences

$$y'' - y = x,$$
$$y(0) = 1,\ y(1) = e - 1$$



Let $x_j = jh$ be the mesh points ($h = 1/N$). Then, at x_j,

$$\frac{y_{j+1} - 2y_j + y_{j-1}}{h^2} - y_j \approx x_j$$

The formula for our approximation u_j is then

$$u_{j+1} - 2u_j + u_{j-1} - h^2 u_j = h^2 x_j, \quad j = 1, \cdots, N-1$$

for the 'interior' points.
At the endpoints, we impose **boundary conditions**

$$u_0 = 1, \quad u_N = e - 1$$

Finite differences

To summarize, we have the problem/appproximation

$$y'' - y = x,$$
$$y(0) = 1,\ \ y(1) = e - 1$$

For $j = 1, 2, \cdots, N-1,$

$$u_{j+1} - (2 + h^2)u_j + u_{j-1} = h^2 x_j$$

$$u_0 = 1, \quad u_N = e - 1$$

Example: With five mesh points $0, 0.2, \cdots, 1$ we have $h = 0.2$ and

$$u_2 - (2 + h^2)u_1 + 1 = h^2 x_1$$

$$u_3 - (2 + h^2)u_2 + u_1 = h^2 x_2$$

$$e - 1 - (2 + h^2)u_3 + u_2 = h^2 x_3$$

which is the linear system

$$\begin{bmatrix} 2 + h^2 & -1 & 0 \\ -1 & 2+h^2 & -1 \\ 0 & -1 & 2+h^2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = -h^2 \begin{bmatrix} 0.2 \\ 0.4 \\ 0.6 \end{bmatrix} - \begin{bmatrix} 1 \\ 0 \\ (e-1) \end{bmatrix}$$

Finite differences

$$u_{j+1} - (2+h^2)u_j + u_{j-1} = h^2 x_j, \quad j = 1, \cdots, N-1$$

$$u_0 = u_N = 0.$$

In general, the system to solve has the form

$$\begin{bmatrix} 2+h^2 & -1 & 0 & \cdots & 0 \\ -1 & 2+h^2 & -1 & \ddots & \vdots \\ 0 & -1 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & 2+h^2 & -1 \\ 0 & \cdots & 0 & -1 & 2+h^2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{N-2} \\ u_{N-1} \end{bmatrix} = -h^2 \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{N-2} \\ x_{N-1} \end{bmatrix} - \begin{bmatrix} u_0 \\ 0 \\ \vdots \\ 0 \\ u_N \end{bmatrix}$$

- The matrix has three diagonals (around the center), called **tri-diagonal**
- Matrices like this how up often when data relates only to adjacent data
- We can solve using Gaussian elimination!

But GE takes $O(n^3)$ work... but only $\approx 3n$ non-zeros - can we do better?

Finite differences

The answer is yes - we can get $O(n)$ time - extremely fast!

Now forget about the ODE context and just consider trying to solve

$$A\mathbf{x} = \mathbf{b}, \qquad A = \begin{bmatrix} q_1 & r_1 & 0 & \cdots & 0 \\ p_2 & q_2 & r_2 & \ddots & \vdots \\ 0 & p_3 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & q_{n-1} & r_{n-1} \\ 0 & \cdots & 0 & p_n & q_n \end{bmatrix}$$

Let's first look at an example, where we use GE to reduce

$$A = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix}$$

and get the LU factorization ($A = LU$).

Finite differences

Here entries of L are noted in red (in the zeroed entries).

$$A = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix} \implies A = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -0.5 & 2.5 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix}$$

(Zero out $(2,1)$ entry using $R_2 \leftarrow R_2 + 0.5R_1$).
From here, we use 'lazy' notation: X denotes a value we *could* compute.

$$\begin{bmatrix} 2 & -1 & 0 & 0 \\ -0.5 & 2.5 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix} \implies \begin{bmatrix} 2 & -1 & 0 & 0 \\ -0.5 & 2.5 & -1 & 0 \\ 0 & X & X & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix}$$

(Zero out $(3,2)$ entry using $R_3 \leftarrow R_3 + (1/2.5)R_2$.

$$\begin{bmatrix} 2 & -1 & 0 & 0 \\ -0.5 & 2.5 & -1 & 0 \\ 0 & X & X & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix} \implies \begin{bmatrix} 2 & -1 & 0 & 0 \\ -0.5 & 2.5 & -1 & 0 \\ 0 & X & X & -1 \\ 0 & 0 & X & X \end{bmatrix}$$

Done! Notice the mostly-zero structure has greatly simplified things...

Tridiagonal matrices

Thus we have found that the result looks like (X being some numbers)

$$A = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix} \implies \begin{bmatrix} 2 & -1 & 0 & 0 \\ \color{red}{X} & X & -1 & 0 \\ 0 & \color{red}{X} & X & -1 \\ 0 & 0 & \color{red}{X} & X \end{bmatrix}$$

$$\implies A = LU \text{ where } L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \color{red}{X} & 1 & 0 & 0 \\ 0 & \color{red}{X} & 1 & 0 \\ 0 & 0 & \color{red}{X} & 1 \end{bmatrix}, \qquad U = \begin{bmatrix} 2 & -1 & 0 & 0 \\ 0 & X & -1 & 0 \\ 0 & 0 & X & -1 \\ 0 & 0 & 0 & X \end{bmatrix}$$

This process generalizes to the $N \times n$ tri-diagonal matrix, where:

- We only need to zero out one entry below the diagonal for each column
- The upper-diagonal never changes
- Both L and U have one diagonal other than the center ('bi-diagonal')

Tridiagonal matrices

Now let's derive an efficient Gaussian elimination for a tridiagonal matrix:

$$\begin{bmatrix} q_1 & r_1 & 0 & \cdots & 0 \\ p_2 & q_2 & r_2 & \ddots & \vdots \\ 0 & p_3 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & q_{n-1} & r_{n-1} \\ 0 & \cdots & 0 & p_n & q_n \end{bmatrix} \implies \begin{bmatrix} d_1 & r_1 & 0 & \cdots & 0 \\ \ell_2 & d_2 & r_2 & \ddots & \vdots \\ 0 & \ell_3 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & d_{n-1} & r_{n-1} \\ 0 & \cdots & 0 & \ell_n & d_n \end{bmatrix}$$

We want to find the ℓ's and d's.
First, $d_1 = q_1$ trivially. Then the first step of GE gives

$$\ell_2 = \frac{p_2}{d_1}, \quad d_2 = q_2 - \ell_2 r_1, \qquad \text{(multiplier: } \ell_2)$$

Then for the next step after that (and so on),

$$\ell_3 = \frac{p_3}{d_2}, \quad d_3 = q_3 - \ell_3 r_2,$$

$$\ell_j = \frac{p_j}{d_{j-1}}, \quad d_j = q_j - \ell_j r_{j-1}, \quad j = 2, 3 \cdots, n.$$

Thus we can solve for variables in the order

$$\ell_2 \to d_2 \ \to \ \ell_3 \to d_3 \ \to \cdots \ell_n \to d_n.$$

Tridiagonal matrices

Finally, to solve $Ax = b$ we solve

$$Ly = b, \qquad Ux = y.$$

Both solves are quite fast - forward/back substitution also simplify!

Forward solve: we have

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ \ell_2 & 1 & \ddots & \vdots \\ \vdots & \ddots & 1 & 0 \\ 0 & \cdots & \ell_n & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \implies y_j + \ell_j y_{j-1} = b_j$$

so y is given by

$$y_1 = b_1, \quad y_j = b_j - \ell_j y_{j-1}, \quad j = 2, \cdots, n.$$

Finally, to solve $Ax = b$ we solve

$$Ly = b, \qquad Ux = y.$$

Both solves are quite fast - forward/back substitution also simplify!

Backward solve: Similarly,

$$\begin{bmatrix} d_1 & r_1 & \cdots & 0 \\ 0 & d_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & r_{n-1} \\ 0 & \cdots & 0 & d_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \implies d_j x_j + r_j x_{j+1} = y_j$$

so we can solve for $\mathbf{x}$ by

$$x_n = y_n/d_n, \quad x_j = \frac{y_j - r_j x_{j+1}}{d_j}, \quad j = n-1, n-2, \cdots, 1$$

Tridiagonal matrices

In summary, we have an efficient Gaussian elimination for solving $Ax = b$ where

$$A = \begin{bmatrix} q_1 & r_1 & 0 & \cdots & 0 \\ p_2 & q_2 & r_2 & \ddots & \vdots \\ 0 & p_3 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & q_{n-1} & r_{n-1} \\ 0 & \cdots & 0 & p_n & q_n \end{bmatrix} \Longrightarrow \begin{bmatrix} d_1 & r_1 & 0 & \cdots & 0 \\ \ell_2 & d_2 & r_2 & \ddots & \vdots \\ 0 & \ell_3 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & d_{n-1} & r_{n-1} \\ 0 & \cdots & 0 & \ell_n & d_n \end{bmatrix}$$

This method is sometimes called the **Thomas algorithm**.

- (**initialize**) Set $d_1 = q_1$ and $y_1 = b_1$.
- (**LU and fwd. solve**) Then for $j = 2, \cdots, n$:

$$\ell_j = p_j / d_{j-1}, \quad d_j = q_j - \ell_j r_{j-1}$$

$$y_j = b_j - \ell_j y_{j-1}.$$

- (**Back solve**) Finally set $x_n = y_n / d_n$ and for $j = n-1, n-2, \cdots, 1$:

$$x_j = (y_j - r_j x_{j+1}) / d_j.$$

Note that you can do the $Ux = y$ solve in parallel with the LU.

Tridiagonal matrices

A **tridiagonal matrix** should be stored in **banded form**:

$$A = \begin{bmatrix} q_1 & r_1 & 0 & \cdots & 0 \\ p_2 & q_2 & r_2 & \ddots & \vdots \\ 0 & p_3 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & q_{n-1} & r_{n-1} \\ 0 & \cdots & 0 & p_n & q_n \end{bmatrix} \text{ is stored as } \begin{bmatrix} 0 & q_1 & r_1 \\ p_2 & q_2 & r_2 \\ \vdots & \vdots & \vdots \\ p_{N-1} & q_{N-1} & r_{N-1} \\ p_{N-1} & q_{N-1} & 0 \end{bmatrix}$$

Pay attention to:

- The zeros - not part of the data (correct code should never read them!)
- Conventions may differ on the unused zeros (**'padding'**)
- Python needs indexing from zero here (and for the algorithm...) (Also indices have to be shifted down by one for python...)

We store only $3n$ numbers - much more feasible than n^2.

Linear algebra in numpy

Features:

- Most linear algebra stuff is in `scipy.linalg`
- Basic features are in `numpy` itself

Slices and such:

- `a.shape` is a tuple of A's dimensions
- Slices in numpy create 'views' to the array
  they are **references** to that data
- Slices can be used to get blocks of a matrix...

Useful 'constructors':

- `arr.tolist()` returns a 'list version' of `arr`
- 'Copy constructor' `b = np.array(a)` (makes a new array `b`!)
- `np.zeros(shape)`: zero matrix
- `np.ones`: matrix of ones
- `np.eye(n)`: identity matrix

Linear algebra in numpy

See example code for the finite difference method. We solve

$$y'' - q(x)y = f(x), \quad y(0) = y_a, \quad y(b) = y_b$$

by solving the linear system In general, the system to solve has the form

$$\begin{bmatrix} 2+h^2 & -1 & 0 & \cdots & 0 \\ -1 & 2+h^2 & -1 & \ddots & \vdots \\ 0 & -1 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & 2+h^2 & -1 \\ 0 & \cdots & 0 & -1 & 2+h^2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{N-2} \\ u_{N-1} \end{bmatrix} = -h^2 \begin{bmatrix} h^2 x_1 - u_0 \\ h^2 x_2 \\ \vdots \\ h^2 x_{N-2} \\ h^2 x_{N-1} - u_N \end{bmatrix} \qquad \text{(FD)}$$

breaking up into the following functions:

a) `build_fd` that creates A (as an array `bands`) and `rhs` as in (FD)

b) `trisolve(bands, rhs)`: solves $Ax =$ `rhs`, with A tri-diagonal

- A `solve` 'main' function that:
  - gets the $Ax =$ `rhs` system from (a)...
  - then solves it using (b),
  - and finally,it adds back in the endpoints u_0, u_N and plots.