Lecture 7: PDEs in higher dimensions

Advection

1D: $u_t + au_x = 0$.

In 2D:

$$u_t + a(x,y)u_x + b(x,y)u_y = 0.$$

Or more generally, we can write this as:

$$u_t + \nabla \cdot (\vec{w}u) = 0,$$

with a vector field $\vec{w}(x,y)$.

[m25_2d_adv.m]

Advection and the wave equation are quite different from diffusion: they are hyperbolic and "information" about the solution travels along characteristics. These are the lines traced out my the vector field $w(x,y)$.

The numerics are a bit different too: this code uses "upwinding" finite differences which are appropriate for advection-dominited problems, but we haven't talked about them in this course.

But we do want to look more carefully about constructing the matrices in this code...

Heat equation

$$u_t = \nabla^2 u = u_{xx} + u_{yy}$$

on a square or rectangle. We can apply centered 2nd-order approximation to each derivative.

In the method of lines approach, we write

$$u_{xx} + u_{yy} \approx \frac{v^n_{i-1,j} - 2v^n_{ij} + v^n_{i+1,j}}{h^2} + \frac{v^n_{i,j-1} - 2v^n_{ij} + v^n_{i,j+1}}{h^2}.$$

This gives a stencil in *space* (then still need to deal with time).

```
           (1)
            | (i,j+1)
            |
            |        (i+1,j)
(1)------(-4)-----(1)
            |(i,j)
            |
            |
           (1)
```

Using forward or backward Euler, accuracy is $O(k + h^2)$.

And a stability restriction for FE of $k < h^2/4$.

In principle, our “finite difference Laplacian” maps a matrix of 2D grid data to another such, and is thus a “4D tensor”. However, in practice we stretch out 2D to 1D, so that the tensor becomes a matrix:

$$\frac{v}{dt} = Lv.$$

How does this “stretch” work? It defines an ordering of the grid points. In Matlab: meshgrid() and (:), see later.

Matrix structure

Let’s look at the structure of L. We choose an ordering for the grid points (why this one? see below) and assume zero boundary conditions:

```
^ y
|       0     0     0
|
|  0    x4    x8    x12   0
|
|  0    x3    x7    x11   0
|
|  0    x2    x6    x10   0
|
|  0    x1    x5    x9    0
|
|       0     0     0
+--------------------------> x
```

with corresponding unknowns $v_1, \ldots, v_{12}$. The discrete Laplacian now looks like this:

```
             |-4  1  0  0    1  0  0  0    0  0  0  0|   | v1  |
             | 1 -4  1  0    0  1  0  0    0  0  0  0|   | v2  |
             | 0  1 -4  1    0  0  1  0    0  0  0  0|   | v3  |
             | 0  0  1 -4    0  0  0  1    0  0  0  0|   | v4  |
             |                                       |   |     |
          1  | 1  0  0  0   -4  1  0  0    1  0  0  0|   | v5  |
 Lv  =   --- | 0  1  0  0    1 -4  1  0    0  1  0  0|   | v6  | .
         h^2 | 0  0  1  0    0  1 -4  1    0  0  1  0|   | v7  |
             | 0  0  0  1    0  0  1 -4    0  0  0  1|   | v8  |
             |                                       |   |     |
             | 0  0  0  0    1  0  0  0   -4  1  0  0|   | v9  |
             | 0  0  0  0    0  1  0  0    1 -4  1  0|   | v10 |
             | 0  0  0  0    0  0  1  0    0  1 -4  1|   | v11 |
             | 0  0  0  0    0  0  0  1    0  0  1 -4|   | v12 |
```

We see that L has a block structure. We can build this easily in Matlab…

Look at [m26_heat2d.m] code… seems to be the kron() command, which seems a bit of “Black Magic”, look at this further…

Matlab: spy command.

Kronecker product

Look at what the kron(I, A) command in Matlab does to a matrix A and an identity matrix I.

One way to explore (and debug) this is to use different h and N in the x and y directions, and on a rectangle instead of a square:

[m27_heat2d_rectangle.m]

Note that meshgrid(), x(:), and reshape(...) all define an ordering of the variables.

We can use a problem where we know the exact solution:

[m27_heat2d_rectangle_error.m]

Then we can do a convergence study, doubling Nx and Ny and making sure the error is roughly second order in h.

Eigenvalue problems

$$-\nabla^2 u = \lambda u.$$

Find eigenfunction u and eigenvalue λ.

In [m28_ellipse_eigen.m] we find a matrix that approximates the operator. Then use sparse matrix methods (**eigs()**).

Geometry Rectangles, cubes etc straightforward. Curved boundaries etc are traditionally harder, particularly to higher-order of accuracy.

[m28_ellipse_eigen.m] For interest: its quite interesting how this works: choses a subset of a regular uniform grid, labels the unknowns within that subset and builds a discrete Laplacian. But note only first-order accurate b/c of treatment of boundary conditions. More on curved geometry coming later in the course.

Non constant diffusion

You will probably look at this in the image processing part of the course.

Elliptic, time-independent problems

Poisson problem:

$$\nabla^2 u = f$$

+ boundary conditions.

Can be the steady state of a heat equation $u_t = \nabla^2 u - f$ (when BCs and f are independent of time) or of interest directly.

We can approximate this using linear algebra with finite difference methods. Approximate the function u with a discrete values on a grid x_j. Then approximate the differential operators with matrices (as we've seen in the method-of-lines approach for time-dependent problems).

For example

$$Lv = f.$$

Software

Here are some routines that construct the various matrices we've discussed, including various boundary conditions. Debugging the order of inputs to the kron() commands is not fun so we've done it for you.

github.com/cbm755/cp_matrices/blob/master/cp_matrices/diff_matrices1d.m

github.com/cbm755/cp_matrices/blob/master/cp_matrices/bulk2d_matrices.m

github.com/cbm755/cp_matrices/blob/master/cp_matrices/bulk3d_matrices.m

Disclaimer: Although care has been taken to remove bugs, its unfortunately possible that some small bugs still remain :-(

We use "unit tests" for this software, you can see them here if you are interested:

github.com/cbm755/cp_matrices/tree/master/cp_matrices/tests_unit