Num Soln of DEs: 08c: PDEs, advection, higher dimensions

Implementation issues

Our demos often use sparse matrices; this saves storage by only storing nonzero entries. Can also be faster to do matrix-vector multiply (for very big problems).

Q: in Matlab, why might k*(L*u) be preferable to k*L*u?

Q: why is h:h:1 the correct grid (rather than 0:h:1) for a periodic computation on [0, 1]?

You can also implement finite differences using for-loops. [demo_08_heat_mol_matrix.m]
[demo_08_heat_loops.m]

Loops can be much slower in Matlab (and other high-level languages), although “JIT-compilers” often speed it up. Personally, I like the abstraction of constructing a discrete operator to approximate my derivatives.

Wave equation with leap frog scheme

\[ u_{tt} = \nabla u \]

or \( u_{tt} = u_{xx} \) in 1D. A fully discrete scheme:

\[
\frac{v_j^{n+1} - 2v_j^n + v_j^{n-1}}{k^2} = \frac{v_{j-1}^n - 2v_j^n + v_{j+1}^n}{h^2}.
\]

[demo_08_wave_leap.m]

Advection

1D: \( u_t + au_x = 0 \).

In 2D:
\( u_t + w \cdot \nabla u = 0 \) where the vector field \( w(x,y) \) is the wind velocity. (or more generally \( u_t + \nabla \cdot (wu) \)).

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 by the vector field \( w(x,y) \).

Upwinding

In 1D, information can flow left or right. We can approximate \( u_x \) with either of two finite difference schemes:

Backward difference: \( u_x \approx \frac{u_{j+1} - u_{j-1}}{2h} \)

Forward difference: \( u_x \approx \frac{u_{j+1} - u_j}{h} \)

For stability, we should select which one to use based which way the wind is blowing: use the one with information from “upwind”/“upstream”. (The fluid-like description is reasonable: commonly used in advection dominated computational fluid dynamics).

Specifically: if \( a > 0 \), use backward difference and if \( a < 0 \), use forward differences. If \( a = 0 \), who cares, its multiplied by zero. [demo_08_advection1d.m]
Variable coefficient advection

\[ u_t + a(x)u_x = 0. \]

Here we upwind at each step. In Matlab, its easy to just compute both and use pointwise “dotstar” multiplication. (In some parallel computing paradigms, this is usually cheaper than “if then” constructs). [demo_08_variable_adv.m]

PDEs in two or more spatial dimensions

Advection

\[ u_t + a(x,y)u_x + b(x,y)u_y = 0. \]

Or as noted above we can write this as:

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

with a vector field \( \vec{w}(x,y) \). [demo_08_2d_adv.m]

Diffusion

\[ u_t + \nabla \cdot (\kappa \nabla u) = f \]

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

Solution: \( u(t,x,y) \)

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

In the method-of-lines approach, we write

\[ u_{xx} + u_{yy} \approx \frac{u_{i-1,j} - 2u_{ij} + u_{i+1,j}}{h^2} + \frac{u_{i,j-1} - 2u_{ij} + u_{i,j+1}}{h^2}. \]

This gives a stencil in space

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

Using forward or backward Euler, accuracy is \( O(k + h^2) \).
And a stability restriction for FE of \( k < h^2/4 \).
Implementation

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

\[ \frac{dU}{dt} = LU. \]

How does this “stretch” work? Need an ordering of the grid points. In Matlab, the following ordering is convenient:

```
[xx, yy] = meshgrid(0:0.5:1, 2:0.5:3.5)
x = xx(:) % stretch
y = yy(:)
```

Also assume zero boundary conditions:

```
  y
 | 0 0 0
 | 3.5+ 0 x4 x8 x12 0
 | 3.0+ 0 x3 x7 x11 0
 | 2.5+ 0 x2 x6 x10 0
 | 2.0+ 0 x1 x5 x9 0
 | 0 0 0 0
+-----------------------+----+----+---------> x
 0 0.5 1
```

Matrix structure Consider the corresponding unknowns \( u_1, \ldots, u_{12} \). The discrete Laplacian:

\[
\begin{bmatrix}
-4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 \\
\end{bmatrix} \begin{bmatrix}
u_1 \\ u_2 \\ u_3 \\ u_4 \\ \vdots \\ u_5 \\ u_6 \\ u_7 \\ u_8 \\ \vdots \\ u_9 \\ u_{10} \\ u_{11} \\ u_{12} \\
\end{bmatrix}
\]

We see that \( L \) has a block structure. We can build this easily with the Kronecker product (\texttt{kron()} in Matlab). \([\text{demo}_08\_heat2d.m]\)

See also \([\text{diff2d}\_\text{matrices.m}] [\text{diff3d}\_\text{matrices.m}]\) in the \texttt{fdmatrices} git repository (see HW4).
Spatially varying diffusion

\[ u_t = \nabla \cdot (k(x)\nabla u) \]

(more generally, diffusion constant might depend on solution)

In one dimension, \( u_t = (k(x)u_x)_x \).

A common approach to discretizing this is to use a forward difference \( D^+_x \) on the \( u_x \) term then evaluate \( k(x) \) at the midpoint: \( k(x_{i+\frac{1}{2}}) \), so that

\[ k(x)u_x \approx k(x_{i+\frac{1}{2}}) \frac{u_{i+1} - u_i}{h} , \]

Then, differentiate the result approximately with a backwards difference \( D^-_x \).

\[ (k(x)u_x)_x \approx D^-_x (k(x_{i+\frac{1}{2}}) D^+_x u) = \frac{k(x_{i+\frac{1}{2}}) \frac{u_{i+1} - u_i}{h} - k(x_{i-\frac{1}{2}}) \frac{u_{i-1} - u_{i-1}}{h}}{h} \]

If \( k \) known only at the grid points, use neighbour averages for midpoint diffusion coefficient:

\[ k(x_{i+\frac{1}{2}}) \approx \frac{1}{2} (k(x_i) + k(x_{i+1})) , \]

(important in more general case where \( k \) depends on \( u \), e.g., \( k(|\nabla u|) \).)

And you can do this in higher-dimensions as well.