Math 405 7b: Splitting, IMEX, advection, diffusion and 2D

Splitting methods

Treating different terms/dimensions/waves/etc in a PDE in different ways is a big area. E.g., number of hits on Google for "Operator Splitting" or "Strang Splitting".

Simplest operator splitting: suppose $u_t = A(u) + B(u)$. Say we alternate between stepping $u_t = A(u)$ and $u_t = B(u)$:

$$\tilde{u} = u^n + kA(u^n)$$
$$u^{n+1} = \tilde{u} + kB(\tilde{u})$$

This is first-order accurate even if we use a high-order Runge-Kutta method (instead of FE). *Strang splitting*:

$$\tilde{u} = u^n + k/2\, A(u^n)$$
$$\bar{u} = \tilde{u} + kB(\tilde{u})$$
$$u^{n+1} = \bar{u} + k/2\, A(\bar{u})$$

(this is first-order as written but can be second-order using e.g., an RK2 method.)

IMEX: implicit/explicit methods

Recall Kuramoto–Sivashinsky example: $u_t = -u_{xx} - u_{xxxx} - (u^2/2)_x$ More generally: $u_t = Lu + N(u)$. Here L linear and N nonlinear operators. Simplest idea is "IMEX Euler": forward Euler for N and backward Euler for L. [demo_06_kuramoto_sivashinsky.m]

IMEX with higher-order accuracy

For higher-order accuracy, see:

- [Ascher–Ruuth–Spiteri, APNUM 1997, Implicit-explicit Runge-Kutta methods for time-dependent partial differential equations]
- [Ascher–Ruuth–Wetton, SINUM 1995, Implicit-explicit methods for time-dependent PDEs]

I like the "SBDF" semi-implicit BDF schemes from this last reference, particularly for reaction-diffusion problems. E.g., SBDF-2:

$$u^{n+1} = 4/3u^n - 1/3u^{n-1} + 2k/3Lu^{n+1} + 4k/3N(u^n) - 2k/3N(u^{n-1}).$$

Exponential Time Differencing

ETDRK treats L part exactly (often in the "Fourier domain") and uses Runge–Kutta for N.

[Cox–Matthews, JCP 2002, "Exponential Time Differencing for Stiff Systems"]

[Kassam–Trefethen, SISC 2005, "Fourth-Order Time-Stepping for Stiff PDEs"]

Example: Korteweg–de Vries eqn:

$$u_t + uu_x + u_{xxx} = 0.$$

Note solitons pass through each other with no lasting effect. Numerical computations were crucial in this discovery (e.g., Fornberg–Whitham, 1970's). [demo_07_etd_kdv.m] from Kassam–Trefethen paper. (We will discuss "spectral" spatial discretizations later).

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). You can also implement finite differences using for-loops. [demo_07_heat_mol_matrix.m] [demo_07_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: "YMMV".

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

Advective PDE problems

Wave equation $u_{tt} = \nabla^2 u$; in 1D: $u_{tt} = u_{xx}$. A fully discrete scheme is "the Leap Frog method" [demo_07_wave_leap.m]:

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

(why leap frog? draw stencil)

First-order advection

1D: $u_t + a u_x = 0$.

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

Advection and the wave equation are similar and quite different from diffusion: they are hyperbolic and "information" about the solution travels along characteristics. Here, 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: $u_x \approx \frac{u_j - u_{j-1}}{h}$

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

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). If we don't do this, the method will be unstable.

Specifically: if $a > 0$, use backward difference and if $a < 0$, use forward differences. If $a = 0$, who cares, its multiplied by zero. [demo_07_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_07_variable_adv.m]

Advection in 2D

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

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_07_2d_adv.m]

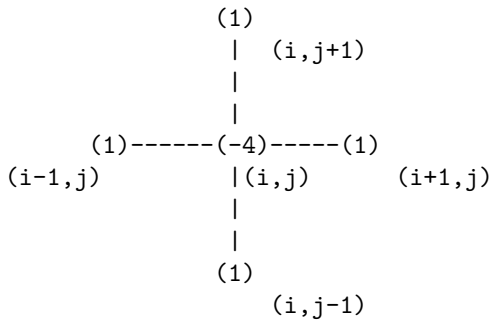
Diffusion in 2D

We can solve diffusion problems in 2D as well. Heat equation becomes $u_t = \nabla^2 u = u_{xx} + u_{yy}$. More generally: $u_t + \nabla \cdot (\kappa \nabla u) = f$. Also need boundary conditions and an initial condition. Solution will be $u(t, x, y)$.

The *domain* of the problem is much "richer" in 2D than in 1D. Easiest case is a square or rectangle. We can apply centered 2nd-order approx 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 *spatial* stencil of:



A fully discrete scheme with foward Euler:

$$u_{ij}^{n+1} = u_{ij}^n + \frac{k}{h^2}\left(u_{i-1,j}^n + u_{i+1,j}^n + u_{i,j-1}^n u_{i,j+1}^n - 4u_{ij}^n\right).$$

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

And a stability restriction for FE of $k < h^2/4$. (recall 1D: $k < h^2/2$, in general $k < h^2/(2dim)$

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:

$$\frac{dU}{dt} = LU.$$

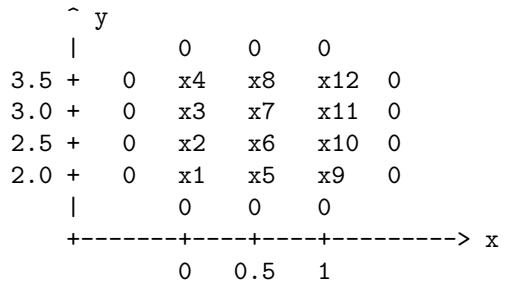
How does this "stretch" work? Mathematically, we are defining an ordering of the set of 2D grid points. In Octave/Matlab, the following ordering is convenient:

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

And we can recover the 2D solution by "reshaping" it back into a matrix:

```
>> reshape(u, size(xx))
```

Example: suppose if we had the rectangle $-0.5 \le x \le 1.5$ and $1.5 \le y \le 4$. Suppose we have zero boundary conditions (so that only interior points appear in our grid). Then this "stretch" ordering is:



Matrix structure The corresponding unknowns (samples of $u(t,x,y)$) are u_1, u_2, …, u_{12}. And the discrete Laplacian is the matrix

$$
\begin{array}{rcl}
 & \left|\begin{array}{cccccccccccc}
-4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
 & & & & & & & & & & & \\
1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 \\
 & & & & & & & & & & & \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4
\end{array}\right| \\
\mathtt{L\ U} = \frac{1}{\mathtt{h\hat{}2}} & \left|\begin{array}{c}
\mathtt{u1} \\ \mathtt{u2} \\ \mathtt{u3} \\ \mathtt{u4} \\ \\ \mathtt{u5} \\ \mathtt{u6} \\ \mathtt{u7} \\ \mathtt{u8} \\ \\ \mathtt{u9} \\ \mathtt{u10} \\ \mathtt{u11} \\ \mathtt{u12}
\end{array}\right| .
\end{array}
$$

We see that L has a block structure. We can build this "easily" (maybe "automatically" is a better word) with the Kronecker product (`kron()` in Octave/Matlab, [demo_07_heat2d.m]). See also `diff2d_matrices.m`, `diff3d_matrices.m` in a future homework.

Caution: `meshgrid` versus `ndgrid` ordering… This is because of the "r, c" convention in linear algebra.

Spatially-varying diffusion $u_t = \nabla \cdot (k(\mathbf{x})\nabla u)$

In one dimension, $u_t = (k(x)u_x)_x$. More generally, diffusion constant k might depend on u too.

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:

$$(k(x)u_x)_x \approx D_-^x\left(k(x_{i+\frac{1}{2}})D_+^x u\right) = \frac{k(x_{i+\frac{1}{2}})\frac{u_{i+1}-u_i}{h} - k(x_{i-\frac{1}{2}})\frac{u_i-u_{i-1}}{h}}{h}.$$

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

$$k(x_{i+\frac{1}{2}}) \approx \frac{1}{2}\big(k(x_i)+k(x_{i+1})\big),$$

(this is important if k depends on u, e.g., $k(|\nabla u|)$.) One can do all this in higher-dimensions as well.