UC Davis, Numerical Solutions of Differential Equations (MAT 228B), Winter 2017

Homework #3

Sam Fleischer

March 3, 2017

Problem 1

Consider

$$u_t = 0.1\nabla^2 u \text{ on } \Omega = (0,1)\times(0,1)$$
$$\frac{\partial u}{\partial \vec{n}} = 0 \text{ on } \partial\Omega$$
$$u(x,y,0) = \exp\Big(-10\Big((x-0.3)^2 + \big(y-0.4\big)^2\Big)\Big).$$

Write a program to solve this PDE using the Peaceman-Rachford ADI scheme on a cell-centered grid. Use a direct solver for the tridiagonal systems. In a cell-centered discretization the solution is stored at the grid points $(x_i, y_i) = \big(\Delta x(i-0.5), \Delta x\big(j-0.5\big)\big)$ for $i, j = 1, \dots, N$ and $\partial_x = \frac{1}{N}$. This discretization is natural for handling Neumann boundary conditions, and it is often used to discretize conservation laws. At the grid points adjacent to the boundary, the one-dimensional discrete Laplacian for homogeneous Neumann boundary conditions is

$$u_{xx}(x_1) \approx \frac{-u_1 + u_2}{\Delta x^2}.$$

(a) Perform a refinement study to show that your numerical solution is second-order accurate in space and time (refine time and space simultaneously using $\Delta t = \Delta x$) at time $t = 1$.

(b) Time your code for different grid sizes. Show how the computational time scales with the grid size. Compare your timing results with those from the previous homework assignment for Crank-Nicolson.

(c) Show that the spatial integral of the solution to the PDE does not change in time. That is,

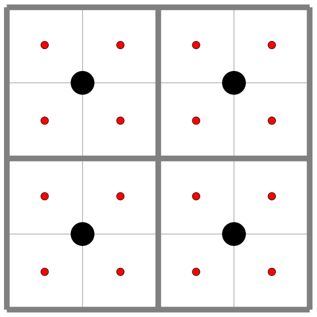
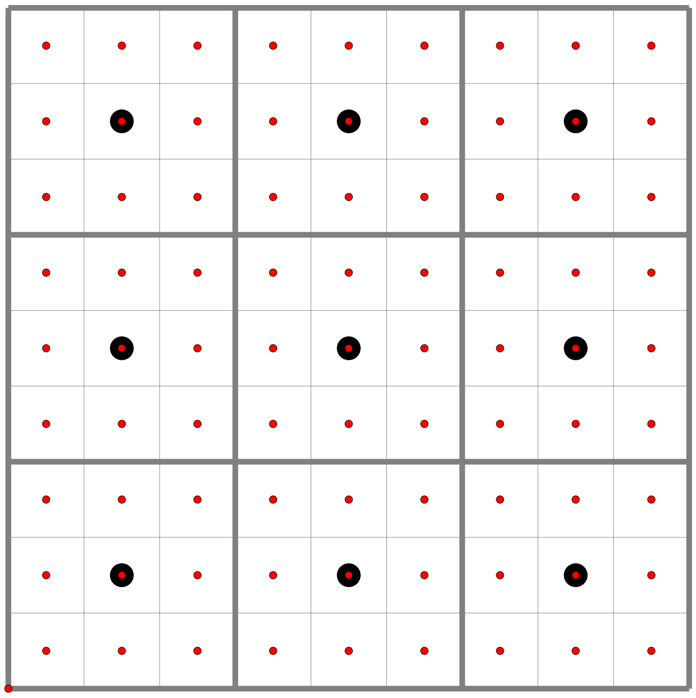
$$\frac{\mathrm{d}}{\mathrm{d}t}\int_\Omega u \,\mathrm{d}V = 0.$$

(d) Show that the solution to the discrete equations satisfies the discrete conservation property

$$\sum_{i,j} u_{i,j}^n = \sum_{i,j} u_{i,j}^0$$

for all n. Demonstrate this property with your code.

(a) When we use methods which don't utilize cell-centered grids, it is easiest to do a refinement study using a grid spacing of 2^n since every other point in the finer mesh is in the same location as a point from the coarser mesh. In cell-centered grids, however, halving Δx moves where we are sampling our functions, so it is easiest to do a refinement study using a grid spacing of 3^n since every third point in the finer mesh is in the same location as a point from the coarser mesh. This is displayed below where large black points are the locations of coarse sampling and small red circles are the locations of fine sampling.


vs.


The refinement study was done by refining the mesh $\Delta x \to \frac{1}{3}\Delta x$, performing the calculations, restricting the solution on the fine mesh by ignoring fine mesh points off of coarse mesh points, and finding the norm of the difference between the coarse solution and the restricted fine solution. The results are below. Note $\|R(u_{\text{fine}}) - u_{\text{coarse}}\|$ is defined as

$$\|R(u_{\text{fine}}) - u_{\text{coarse}}\| = \max_{i,j}\left|R(u_{\text{fine}})_{i,j} - (u_{\text{coarse}})_{i,j}\right|$$

where R is the restriction operator, $R : \mathbb{R}^{3^{i+1}\times 3^{i+1}} \to \mathbb{R}^{3^i \times 3^i}$:

$$R(u_{\text{fine}})_{i,j} = (u_{\text{fine}})_{3i+1,3j+1}$$

Δx	Δt	$\|u_{\text{fine}} - u_{\text{coarse}}\|$	$\frac{\|u_{\text{fine}} - u_{\text{finer}}\|}{\|u_{\text{coarse}} - u_{\text{fine}}\|}$
3^{-1}	3^{-1}	-	-
3^{-2}	3^{-2}	7.0578×10^{-3}	-
3^{-3}	3^{-3}	8.2709×10^{-4}	8.5332
3^{-4}	3^{-4}	9.1141×10^{-5}	9.0747
3^{-5}	3^{-5}	1.0117×10^{-5}	9.0092
3^{-6}	3^{-6}	1.1240×10^{-6}	9.0008

We can see that as we refine our grid and timestep by a factor of 3, our error ratios approach $3^2 = 9$, which is evidence that our numerical scheme is 2nd order accurate.

(b) Here are my results of timing for various grid sizes:

Δx	Δt	time (in seconds)	time ratios
3^{-1}	3^{-1}	4.6710×10^{-3}	-
3^{-2}	3^{-2}	5.6630×10^{-3}	1.2124
3^{-3}	3^{-3}	7.8460×10^{-3}	1.3855
3^{-4}	3^{-4}	7.7185×10^{-2}	9.8375
3^{-5}	3^{-5}	1.1787	15.2706
3^{-6}	3^{-6}	40.6882	34.5207

(c)

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}t}\int_\Omega u\,\mathrm{d}V &= \int_\Omega \frac{\mathrm{d}}{\mathrm{d}t}u\,\mathrm{d}V && \text{by the Lebesgue Dominated Convergence Theorem}\\
&= \int_\Omega u_t\,\mathrm{d}V && \text{by the definition of } \frac{\mathrm{d}}{\mathrm{d}t}\mathrm{u}\\
&= \int_\Omega 0.1\nabla^2 u\,\mathrm{d}V && \text{by the given PDE}\\
&= 0.1\int_\Omega \nabla\cdot\nabla u\,\mathrm{d}V && \text{by the definition of } \nabla^2\\
&= 0.1\int_{\partial\Omega} \nabla u\cdot n\,\mathrm{d}S && \text{by the flux-divergence theorem}\\
&= 0.1\int_{\partial\Omega} 0\,\mathrm{d}S && \text{by the boundary condition}\\
&= 0
\end{aligned}$$

(d) To show $\sum_{i,j} u_{i,j}^n = \sum_{i,j} u_{i,j}^0$ for all n, consider the following:

Numerically, we can show this by finding, for any grid size, $\sum_{i,j} u_{i,j}^0$ and $\sum_{i,j} u_{i,j}^{N_t}$, and comparing them using absolute value $\left|\sum_{i,j} u_{i,j}^0 - \sum_{i,j} u_{i,j}^{N_t}\right|$:

Δx	$\sum_{i,j} u_{i,j}^0$	$\sum_{i,j} u_{i,j}^{N_t}$	$\left\|\sum_{i,j} u_{i,j}^0 - \sum_{i,j} u_{i,j}^{N_t}\right\|$
3^{-1}	2.5644	2.5644	0.0
3^{-2}	22.3003	22.3003	4.9738×10^{-14}
3^{-3}	199.9151	199.9151	0.0
3^{-4}	1798.4493	1798.4493	1.1141×10^{-11}
3^{-5}	16185.2578	16185.2578	1.3439×10^{-08}
3^{-6}	145666.5337	145666.5337	9.6631×10^{-07}

Problem 2

The FitzHugh-Nagumo equations

$$\frac{\partial v}{\partial t} = D\nabla^2 v + (a-v)(v-1)v - w + I$$
$$\frac{\partial w}{\partial t} = \varepsilon\left(v - \gamma w\right)$$

are used in electrophysiology to model the cross membrane electrical potential (voltage) in cardiac tissue and in neurons. Assuming that the spatial coupling is local and passive results in the term which looks liek the diffusion of voltage. The state variables are the voltage v and the recovery variable w.

(a) Write a program to solve the FitzHugh-Nagumo equations on the unit square with homogeneous Neumann boundary conditions for v (meaning electrically insulated). Use a fractional step method to handle the diffusion and reactions separately. Use an ADI method for the diffusion solve. Describe what ODE solver you used for the reactions and what fractional stepping you chose.

(b) Use the following parameters $a = 0.1$, $\gamma = 2$, $\varepsilon = 0.005$, $I = 0$, $D = 5\cdot 10^{-5}$, and initial conditions

$$v(x,y,0) = \exp\left(-100\left(x^2+y^2\right)\right)$$
$$w(x,y,0) = 0.0.$$

Note that $v = 0$, $w = 0$ is a stable steady state of the system. Call this the rest state. For these initial conditions the voltage has been raised above rest in the bottom corner of the domain. Generate a numerical solution up to time $t = 300$. Visualize the voltage and describe the solution. Pick space and time steps to resolve the spatiotemporal dynamics of the solution you see. Discuss what grid size and time step you used and why.

(c) Use the same parameters from part (b), but use the initial conditions

$$v(x,y,0) = 1 - 2x$$
$$w(x,y,0) = 0.05y,$$

and run the simulation until time $t = 600$. Show the voltage at several points in time (pseudocolor plot, or contour plot, or surface plot $z = v(x,y,t)$) and describe the solution.

(a) Here is my Python code to generate a 1D Laplacian:

```python
def get_Lap_1D(N,dx):
    off_diag = np.ones(N)
    diag = (-2)*np.ones(N)
    diag[0] = -1
    diag[-1] = -1
    A = np.vstack((off_diag,diag,off_diag))/(dx**2)
    L = sp.dia_matrix((A,[-1,0,1]),shape=(N,N))
    return L
```

where `N` is the number of grid points in one direction and `dx` is the grid spacing. Note that I imported the package `numpy` as `np` and `scipy.sparse` as `sp`. Here is my code to generate a Peaceman-Rachford step function:

```python
def make_PR_step_method(N,dx,dt,transport_coef):

    L = get_Lap_1D(N,dx)
    I = sp.identity(N)
    right_mat = I + (transport_coef*dt/2)*L
    left_mat = sp.csc_matrix(I - (transport_coef*dt/2)*L)

    def PR_step(u):
        RHS_half = right_mat.dot(u)
        u_half = sp.linalg.spsolve(left_mat, RHS_half)

        RHS = right_mat.dot(np.transpose(u_half))
        u_np1 = np.transpose(sp.linalg.spsolve(left_mat, RHS))

        return u_np1
    return PR_step
```

Note that the function `make_PR_step_method` returns the function `PR_step`. I implemented it this way so that I wouldn't need to pass in `right_mat` and `left_mat` every time I called `PR_step`. Here is my code to generate a Runge-Kutta-2 method:

```
def make_RK2_step_method(N,dx,dt,f_v,f_w):

    def RK2_step(v,w):
        k1 = f_v(v,w)
        l1 = f_w(v,w)

        k2 = f_v(v+dt*k1,w+dt*l1)
        l2 = f_w(v+dt*k1,w+dt*l1)

        k = (k1+k2)/2
        l = (l1+l2)/2
        v_new = v + dt*k
        w_new = w + dt*l
        return (v_new, w_new)

    return RK2_step
```