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 $i, j = 1,\ldots,N$ 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,

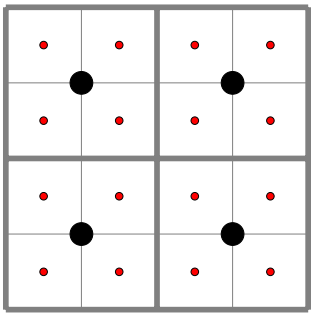
$$\frac{\mathrm{d}}{\mathrm{d}t}\int_\Omega u \mathrm{d}V = 0.$$

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

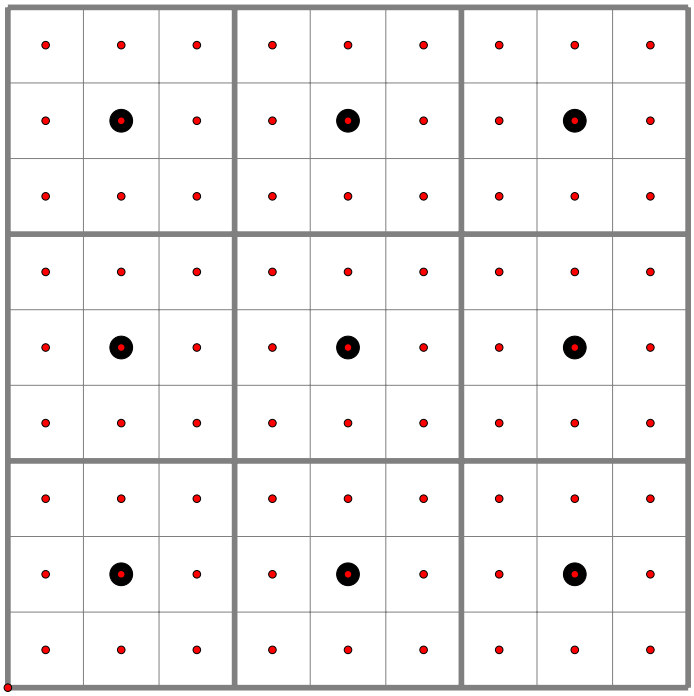
$$\sum_{i,j} u^n_{i,j} = \sum_{i,j} u^0_{i,j}$$

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}	7.0577×10^{-3}	-
3^{-3}	3^{-3}	8.2709×10^{-4}	8.5332
3^{-4}	3^{-4}	9.1142×10^{-5}	9.0747
3^{-5}	3^{-5}	1.0117×10^{-5}	9.0092
3^{-6}	3^{-6}	1.1240×10^{-6}	9.0008
3^{-6}	3^{-6}	1.2501×10^{-7}	8.9913

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	Crank Nicolson time (in seconds)	Crank Nicolson time ratios
3^{-1}	3^{-1}	2.8170×10^{-3}	-	4.217×10^{-3}	-
3^{-2}	3^{-2}	3.3240×10^{-3}	1.1800	4.262×10^{-3}	1.0107
3^{-3}	3^{-3}	7.5500×10^{-3}	2.2714	4.1217×10^{-2}	9.6708
3^{-4}	3^{-4}	7.3320×10^{-2}	9.7113	1.4063	34.1190
3^{-5}	3^{-5}	1.1128	15.1773	85.2092	60.5919
3^{-6}	3^{-6}	38.4695	34.5700	-	-
3^{-7}	3^{-7}	1518.1730	39.4643	-	-

I did not run Crank Nicolson for more than 3^{-5} since it would have taken nearly 2 hours for 3^{-6} and my computer stalled when I tried to run it for 3^{-7}.

(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 $\sum_{i,j} u^n_{i,j} = \sum_{i,j} u^0_{i,j}$ for all n, we want to show $\sum_{i,j} u^*_{i,j} = \sum_{i,j} u^n_{i,j}$ where

$$\left(I - \frac{D\Delta t}{2}L\right)u^* = \left(I + \frac{D\Delta t}{2}L\right)u^n$$

So,

$$\sum_{i,j}\left(I - \frac{D\Delta t}{2}L\right)u^*_{i,j} = \sum_{i,j}\left(I + \frac{D\Delta t}{2}L\right)u^n_{i,j}$$

$$\sum_{i,j} u^*_{i,j} - \sum_{i,j}\frac{D\Delta t}{2}Lu^*_{i,j} = \sum_{i,j} u^n_{i,j} + \sum_{i,j}\frac{D\Delta t}{2}Lu^n_{i,j}$$

But since L is a matrix of the following form

$$L = \begin{bmatrix}
-1 & 1 & 0 & 0 & \dots & 0\\
1 & -2 & 1 & 0 & \dots & 0\\
0 & 1 & -2 & \ddots & \ddots & \vdots\\
0 & 0 & \ddots & \ddots & 1 & 0\\
\vdots & \vdots & \ddots & 1 & -2 & 1\\
0 & 0 & \dots & 0 & 1 & -1
\end{bmatrix}$$

then for a column $v = (v_i)_{i=1}^N$ of any matrix u,

$$Lv = \begin{bmatrix} -v_1 + v_2 \\ v_1 - 2v_2 + v_3 \\ v_2 - 2v_3 + v_4 \\ \vdots \\ v_{N-2} - 2v_{N-1} + v_N \\ v_{N-1} - v_N \end{bmatrix}$$

So,

$$\sum_{i=1}^{N} (Lv)_i = 0$$

Thus, for any matrix u,

$$\sum_{i,j} Lu = \sum_j \sum_i (Lu_j)_i = \sum_j 0 = 0$$

Thus,

$$\sum_{i,j} u^*_{i,j} - \sum_{i,j} \frac{D\Delta t}{2} Lu^*_{i,j} = \sum_{i,j} u_{i,j} + \sum_{i,j} \frac{D\Delta t}{2} Lu_{i,j}$$
$$\implies \sum_{i,j} u^*_{i,j} = \sum_{i,j} u_{i,j}$$

Since $\left(I - \frac{D\Delta t}{2} L\right) u^{n+1} = \left(I + \frac{D\Delta t}{2} L\right) u^*$, then $\sum_{i,j} u^{n+1}_{i,j} = \sum_{i,j} u^*_{i,j}$. So,

$$\sum_{i,j} u^{n+1}_{i,j} = \sum_{i,j} u^n_{i,j}$$

By induction,

$$\sum_{i,j} u^n_{i,j} = \sum_{i,j} u^0_{i,j}$$

for all n.

Numerically, we can show this by finding, for any grid size, $\sum_{i,j} u^0_{i,j}$ and $\sum_{i,j} u^{N_t}_{i,j}$, and comparing them using absolute value $\left|\sum_{i,j} u^0_{i,j} - \sum_{i,j} u^{N_t}_{i,j}\right|$: The results are similar at every time step n.

Δx	$\sum_{i,j} u^0_{i,j}$	$\sum_{i,j} u^{N_t}_{i,j}$	$\left\|\sum_{i,j} u^0_{i,j} - \sum_{i,j} u^{N_t}_{i,j}\right\|$
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}
3^{-7}	1310998.0170	1310998.0171	4.6604×10^{-5}

Problem 2

The FitzHugh-Nagumo equations

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

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

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

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) To solve

$$\frac{\partial}{\partial t}\begin{bmatrix} v \\ w \end{bmatrix} = A\left(\begin{bmatrix} v \\ w \end{bmatrix}\right) + B\left(\begin{bmatrix} v \\ w \end{bmatrix}\right).$$

where

$$A\left(\begin{bmatrix} v \\ w \end{bmatrix}\right) = \begin{bmatrix} D\nabla^2 v \\ 0 \end{bmatrix} \quad \text{and} \quad B\left(\begin{bmatrix} v \\ w \end{bmatrix}\right) = \begin{bmatrix} (a-v)(v-1) - w + I \\ \varepsilon\big(v - \gamma w\big) \end{bmatrix},$$

I implemented a fractional stepping method over a timestep Δt. First I solved $Au^* = u^n$ for u^* using the same Peaceman-Rachford method used in Problem 1 (since the first component of A is diffusion), except over a timestep of $\frac{\Delta t}{2}$. Then I implemented a second-order Runge-Kutta technique to solve $Bu^{**} = u^*$ over a timestep of Δt. Then I set u^{n+1} as the solution to $Au^{n+1} = u^{**}$, where I again used the Peaceman-Rachford method for time step $\frac{\Delta t}{2}$. Below is all the code needed to solve this problem.

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

```
1  def get_Lap_1D(N,dx):
2      # Generate the 1D Laplacian with 1/dx^2 on the
3      # super- and sub-diagonals and -2/dx^2 on the diagonal
4      off_diag = np.ones(N)
5      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:

```
def make_PR_step_method(N,dx,dt,transport_coef):
    # Make a method to run one step of Peaceman-Rachford.
    # Get the Laplacian and Identity matrices
    L = get_Lap_1D(N,dx)
    I = sp.identity(N)
    # Form the RHS (I + b*dt/2*L) and LHS (I - b*dt/2*L) matrices
    # The LHS should be sparse since we are directly inverting it.
    right_mat = I + (transport_coef*dt/2)*L
    left_mat = sp.csc_matrix(I - (transport_coef*dt/2)*L)

    def PR_step(u):
        # Take input u^n and return u^{n+1} after diffusing
        # in the x direction...
        RHS_half = right_mat.dot(u)
        u_half = sp.linalg.spsolve(left_mat, RHS_half)

        # ... and diffusing in the y direction.
        RHS = right_mat.dot(np.transpose(u_half))
        u_next = np.transpose(sp.linalg.spsolve(left_mat, RHS))

        return u_next
    # Return the function PR_step.  right_mat and left_mat
    # are constant for each PR_step function returned.
    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):
    # Make a method to run one step of Runge-Kutta 2.

    def RK2_step(v,w):
        # Get the left side gradient
        k1 = f_v(v,w)
        l1 = f_w(v,w)
        # Get the right side gradient
        k2 = f_v(v+dt*k1,w+dt*l1)
        l2 = f_w(v+dt*k1,w+dt*l1)
        # Average the gradients
        k = (k1+k2)/2
        l = (l1+l2)/2
        # Use the average gradient to get the next point
        v_next = v + dt*k
        w_next = w + dt*l

```

```
18          return (v_next, w_next)
19      # Return a customized RK2_step function.  f_v and f_w
20      # are constant for each RK2_step function returned.
21      return RK2_step
```

Note that the function `make_RK2_step_method` returns the function `RK2_step`. This is so that I don't need to pass the model functions (given by the operator B above and by `f_v` and `f_w` in the code), to the Runge-Kutta method every time I call it. To construct the 2-variable vector functions given by the operator B, I used the following code:

```
1  def make_RHS_ODEs(a,I,gamma,epsilon):
2      # Form functions given parameters a, I, gamma, and epsilon.
3      def f_v(v,w):
4          return (a - v)*(v - 1)*v - w + I
5
6      def f_w(v,w):
7          return epsilon*(v - gamma*w)
8      # Return these functions which can be passed into
9      # make_RK2_step_method().
10     return (f_v,f_w)
```

Next I defined my initial condition functions:

```
1  def initial_condition_b(x,y):
2      return (np.exp(-100*(x**2 + y**2)), 0*x)
3
4  def initial_condition_c(x,y):
5      return (1 - 2*x, 0.05*y)
```

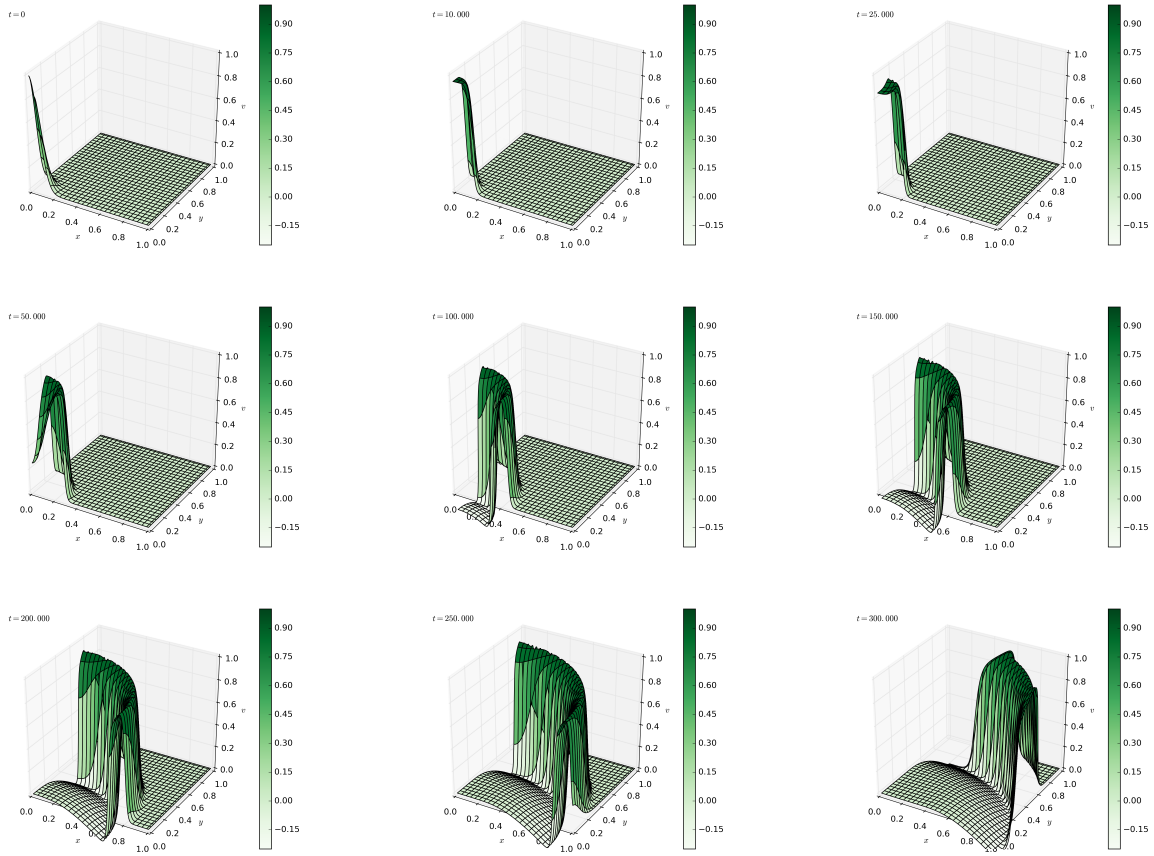
Finally, here is the code to bring it all together to solve a problem:

```
1  # Set dx and dt.
2  dx = 3**(-4)
3  final_time = 300
4  dt = 1
5  Nx = int(round(1/dx))
6  Nt = int(round(final_time/dt))
7
8  # Set parameters
9  D = 5*(10**(-5))
10 a = 0.1
11 gamma = 2
12 epsilon = 0.005
13 I = 0
14
15 # Define the Peaceman-Rachford step function using given parameters.
16 PR_step = make_PR_step_method(Nx,dx,dt/2,D)
17
18 # Define the vector functions for the RK2 step
19 (f_v,f_w) = make_RHS_ODEs(a,I,gamma,epsilon)
20 # Define the RK2 step using those functions.
21 RK2_step = make_RK2_step_method(Nx,dx,dt,f_v,f_w)
22
23 # Define x and y mesh points.
24 xaxis = np.linspace(0,1-dx,Nx) + dx/2
25 yaxis = np.linspace(0,1-dx,Nx) + dx/2
```

```
26 x,y = np.meshgrid(xaxis,yaxis)
27
28 # Get initial condition
29 (v_0, w_0) = initial_condition_b(x,y)
30 # Un-comment for part c
31 # (v_0, w_0) = initial_condition_c(x,y)
32
33 for t in xrange(1,Nt+1):
34     # Do a half-step of Peaceman-Rachford
35     v_star = PR_step(v_0)
36     w_star = w_0 + 0
37
38     # Do a full step of Runge-Kutta 2
39     (v_double_star, w_double_star) = RK2_step(v_star, w_star)
40
41     # Do another half-step of Peaceman-Rachford
42     v_1 = PR_step(v_double_star)
43     w_1 = w_double_star + 0
44
45     # Redefine v_0 and w_0 to do the next step
46     v_0 = v_1 + 0
47     w_0 = w_1 + 0
```

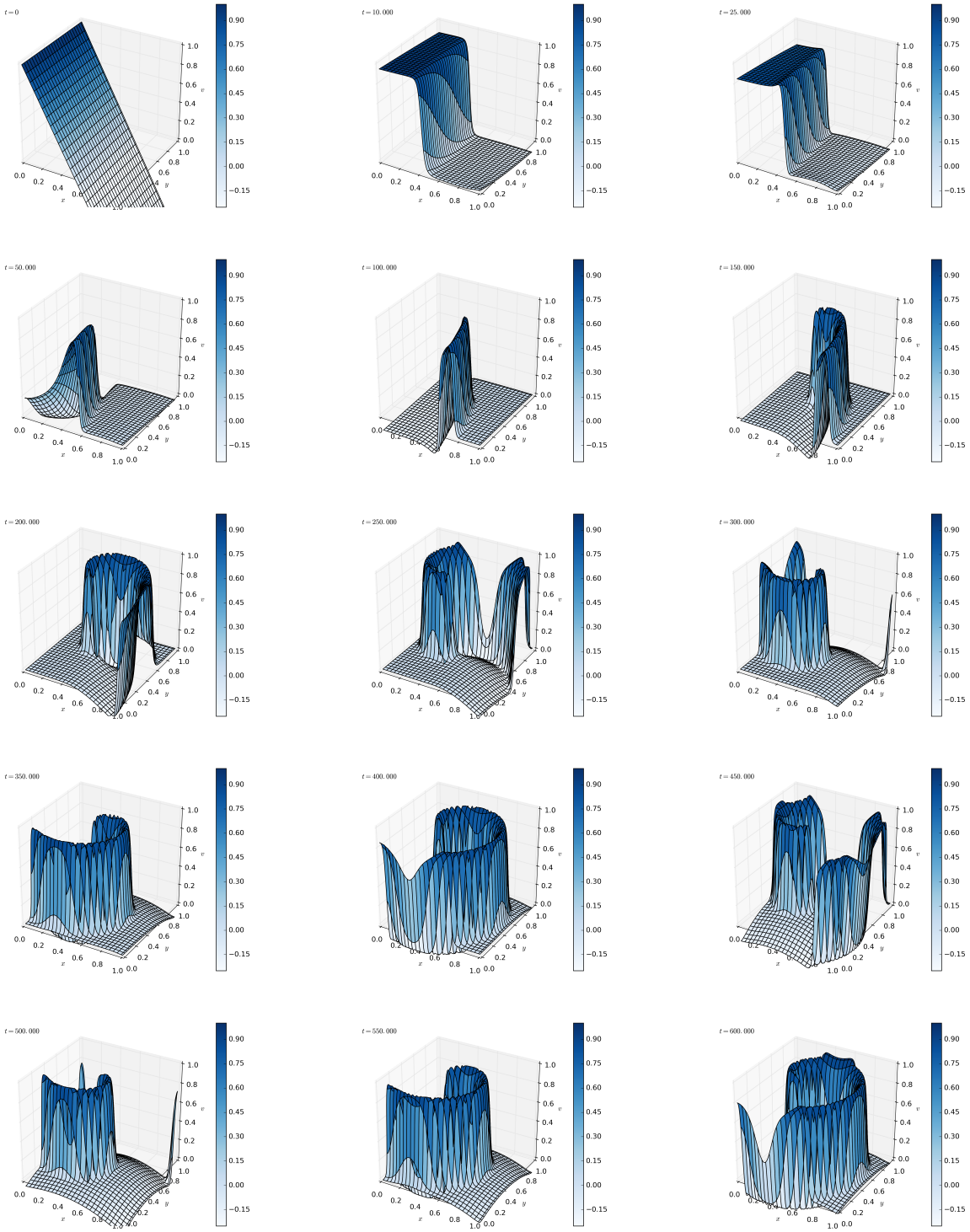
For homework presentation purposes, I've ommited the plot commands.

(b) Since the wave's minimum and maximum are at a distance of about 0.1, I chose $\Delta x = 2^{-7} = \frac{1}{128}$ so that there are over 12 grid points in every 0.1 space interval. Since the wave nearly completely finishes travelling diagonally across a 1×1 grid by $t = 300$, I chose $\Delta t = 1$ to temporally resolve the dynamics. Here are the results for $t = 0$, 10, 25, $50 + 100k$, and $100 + 100k$ for $k = 0, 1, 2$.



Notice this wave travels in the direction where voltage $v = 0$ since after a particular cell is activated there is a refractory period.

(c) Since the parameters are the same as in part (b), I chose $\Delta x = 2^{-7}$ and $\Delta t = 1$. Here are the results for $t = 0$, 10, 25, $50 + 100k$, and $100 + 100k$ for $k = 0, 1, \ldots, 5$.



Since $w(x, y, 0) \not\equiv 0$, some cells need less time to recover than others. This results in cyclic dynamics both spatially and temporally. The wave travels in whatever direction has $v = 0$ and where cells have recovered quickly. This happens to be in a circular pattern.