Numerical Analysis 2 Project

Nick Kallfa

5-9-14

1 Introduction

In this project we set out to solve a reaction-diffusion equation for a function $u(t, x)$. Reaction-diffusion systems arise naturally in areas such as chemistry or chemical engineering where a substance is transformed into another and spreads out over time, but they can also be used to describe non-chemical processes in biology, for example, with population dynamics or predator-prey models. We will be concerned with a chemical process so $u(t, x)$ can be seen to represent the concentration of some chemical at time t and position x. Such systems typically involve two parts: reaction and diffusion. The reaction part describes how chemicals are transformed into each other and the diffusion part explains how the substance smooths out over time. We expect our numerical approximation to exhibit these properties.

2 The Problem

The problem we face is computing a numerical approximation to the solution of a reaction-diffusion equation in one spatial dimension. The general equation for a process of this sort is given by:

$$\frac{\partial}{\partial t} = \frac{\partial}{\partial x}\Big[D\frac{\partial u}{\partial x}\Big] + \Phi(t, x, u)$$

where D is a constant called the diffusion coefficient and $\Phi(t, x, u)$ is a func-

tion that gives us the rate at which the chemical is being produced. To simplify computations we will assume $0 \leq x \leq 1$. We will denote the spatial interval as Ω and set boundary conditions $u(t,0) = u(t,1) = 0$. In addition we will specify initial conditions as $u(0,x) = f(x)$ where $f(x)$ is a given function.

3 Method

In order to compute an approximation to the solution of the reaction-diffusion equation we start by discretizing both space and time. As we will explain in greater detail in the coming subsections we used a second order central difference formula for space and a third order implicit Runge-Kutta method known as Radau IIA for time.

The implicit nature of the Radau IIA method forces us to solve a linear system of equations at each time step. In order to solve the system at each time step we set up an iterative scheme which gives us information needed to compute the future time step. This procedure is repeated over and over again for all time steps in order to compute our numerical approximation.

3.1 Spatial Discretization

For the spatial discretization we broke up Ω into N sub-intervals with spacing $\Delta x = k = \dfrac{1}{N}$ with $x(i) = ik$ for $i = 1, 2, 3, \ldots, N$. Then for each time value, t, we could determine $u_i(t) \approx u\Big(t, x(i)\Big)$.

Recall that for a general function f and step size h, the second order central difference formula is given by:

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$$

This is what we will use in the spatial discretization. For our particular problem what this gives us is the following formula:

$$\frac{du_i}{dt} = \frac{D}{k^2}\Big[u_{i+1} - 2u_i + u_{i-1}\Big] + \Phi(t, x(i), u_i)$$

where, again, D is the diffusion coefficient and $k = \dfrac{1}{N}$ is the spatial step size.

This gives us the ith component of $\dfrac{du_i}{dt}(t)$ for any time t for $i = 1, 2, ..., N$. We can write this in matrix form as:

$$\frac{d\mathbf{u}}{dt} = B_k\mathbf{u} + \mathbf{\Phi}(t, \mathbf{u})$$

This is a nonlinear differential equation of a vector valued function $\mathbf{u}$ with the right hand side $\mathbf{w} = B_k\mathbf{u} + \mathbf{\Phi}(t, \mathbf{u})$. Written component-wise we see that the ith component of $\mathbf{w}$ is given by:

$$w_i = \frac{D}{k^2}\Big[u_{i+1} - 2u_i + u_{i-1}\Big]$$

Also, the ith component of $\mathbf{\Phi}$ is given by $\Phi_i(t, \mathbf{u}) = \Phi(t, x(i), u_i)$. Note that with the boundary conditions $u(t, 0) = u(t, 1) = 0$, $\mathbf{w}$ is a vector of length $N - 1$. Similarly, because of the boundary conditions, the matrix B_k is a $(N - 1) \times (N - 1)$ tridiagonal matrix having the following form:

$$B_k = \begin{bmatrix} \frac{-2D}{k^2} & \frac{D}{k^2} & & & & \\ \frac{D}{k^2} & \frac{-2D}{k^2} & \frac{D}{k^2} & & & \\ & \frac{D}{k^2} & \frac{-2D}{k^2} & \frac{D}{k^2} & & \\ & & \ddots & \ddots & \ddots & \\ & & & \frac{D}{k^2} & \frac{-2D}{k^2} & \frac{D}{k^2} \\ & & & & \frac{D}{k^2} & \frac{-2D}{k^2} \end{bmatrix}$$

3.2 Time Discretization

Now that we have discretized space we also need to discretize time as well. In order to do this we use a third order implicit Runge-Kutta method known as Radau IIA.

Before we discuss the particular Runge-Kutta method that we will be using recall that for a differential equation $\frac{dy}{dt} = f(t, y)$ implicit Runge-Kutta methods with s stages have the general form:

$$v_i = f\Big(t_n + c_i h + \textstyle\sum_{j=1}^{s} a_{ij} v_j\Big), \qquad \text{for } i = 1, 2, ..., s$$

with update

$$y_{n+1} = y_n + h \textstyle\sum_{j=1}^{s} b_i v_i$$

The a, b, c terms are given by the Butcher tableau of the corresponding Runge-Kutta Method written as

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \dots & a_{1s} \\ c_2 & a_{21} & a_{22} & \dots & a_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \dots & a_{ss} \\ \hline & b_1 & b_2 & \dots & b_s \end{array} = \begin{array}{c|c} \mathbf{c} & A \\ \hline & \mathbf{b^T} \end{array}$$

For our particular problem, we will be using a third order RK method, Radau IIA. This method has two stages and its Butcher tableau is given below:

$$\begin{array}{c|cc} 1/3 & 5/12 & -1/12 \\ 1 & 3/4 & 1/4 \\ \hline & 3/4 & 1/4 \end{array}$$

So now at each time step we must solve the following set of equations for

$p = 1, 2$:

$$\mathbf{v}_p = B_k\Big[\mathbf{u} + h\sum_{q=1}^{2} a_{pq}\mathbf{v}_q\Big] + \mathbf{\Phi}\Big(t + c_p h, \mathbf{u} + h\sum_{q=1}^{2} a_{pq}\mathbf{v}_q\Big)$$

As we will explain in the next subsection, because of the implicit nature of the method, solving this set of equations is a delicate process. We need to solve a linear system of equations and we develop an iterative scheme for doing so.

3.3 Iterative Scheme

Again, because we are using an implicit method we need to solve a linear system of equations at each time step. This is because in the previous equation we have the unknown $\mathbf{v}_p$ on the left hand side defined in terms of itself. This is what forces us to solve a system of equations. In order to find $\mathbf{v}_p$ for $p = 1, 2$ we set up an iterative scheme given by:

$$\mathbf{v}_p^{r+1} = B_k\Big[\mathbf{u} + h\sum_{q=1}^{2} a_{pq}\mathbf{v}_q^{r+1}\Big] + \mathbf{\Phi}\Big(t + c_p h, \mathbf{u} + h\sum_{q=1}^{2} a_{pq}\mathbf{v}_q^{r}\Big)$$

for $p = 1, 2$ and for $r = 1, 2, 3, ...$

The idea is that given an initial $\mathbf{v}_1^1$ and $\mathbf{v}_2^1$ we can find $\mathbf{v}_1^2$ and $\mathbf{v}_2^2$. Once we have $\mathbf{v}_1^2$ and $\mathbf{v}_2^2$ we can repeat the same procedure and we can continue doing so until we reach a stopping point. This stopping point will be determined once the 2-norm of the difference between the new output, $\mathbf{v}_p^{r+1}$, and the previous output, $\mathbf{v}_p^r$, is below a given threshold. In the actual implementation we set the threshold to be 10^{-10} and the initial $\mathbf{v}_p^1$ we use the zero vector. We begin by rewriting the previous equation by isolating all of the terms with $\mathbf{v}^{r+1}$ on one side giving us

$$\mathbf{v}_p^{r+1} - B_k h\sum_{q=1}^{2} a_{pq}\mathbf{v}_q^{r+1} = B_k\mathbf{u} + \mathbf{\Phi}\Big(t + c_p h, \mathbf{u} + h\sum_{q=1}^{2} a_{pq}\mathbf{v}_q^{r}\Big)$$

Factoring out the $\mathbf{v}^{r+1}$'s on the left hand side and letting $\mathbf{b}_p^r = \mathbf{\Phi}\Big(t + c_p h, \mathbf{u} +$

$h\sum_{q=1}^{2} a_{pq}\mathbf{v}_q^r$) we obtain

$$\Big(I - B_k h\sum_{q=1}^{2} a_{pq}\Big)\mathbf{v}_p^{r+1} = B_k\mathbf{u} + \mathbf{b}_p^r$$

for $p = 1, 2$.

Again, the idea is that given the initial $\mathbf{v}_p^1$ we will know the right hand side $B_k\mathbf{u} + \mathbf{b}_p^1$ allowing us to find $\mathbf{v}_p^2$ on the left hand side of the above equation. Once we have this second set of $\mathbf{v}_p$'s we can use them to determine the new right hand side $B_k\mathbf{u} + \mathbf{b}_p^2$ in order to find $\mathbf{v}_p^3$. We continue this iterative procedure until the 2-norm of the difference between successive $\mathbf{v}_p$'s is below a predetermined threshold. The method we use to solve this linear system is different than most standard methods used and what we are trying to show is that the method we use here is equivalent to the standard method. Once we have set up this system, we will see it can be solved by means of a block LU factorization and subsequent forward/backward substitutions.

To get back to the task at hand, let's drop the r superscripts and look at what $\mathbf{v}_p$ is for both $p = 1$ and $p = 2$. For $p = 1$ we have to solve

$$\Big(I - B_k h\sum_{q=1}^{2} a_{1q}\Big)\mathbf{v}_1 = B_k\mathbf{u} + \mathbf{b}_1$$

which can be written as

$$\Big(I - B_k h(a_{1,1} + a_{1,2})\Big)\mathbf{v}_1 = B_k\mathbf{u} + \mathbf{b}_1$$

Similarly, for $p = 2$ we have

$$\Big(I - B_k h(a_{2,1} + a_{2,2})\Big)\mathbf{v}_2 = B_k\mathbf{u} + \mathbf{b}_2$$

If we stack $\mathbf{v}_1$ and $\mathbf{v}_2$ then we obtain the following system:

$$\begin{pmatrix} I - ha_{1,1}B_k & -ha_{1,2}B_k \\ I - ha_{2,1}B_k & -ha_{2,2}B_k \end{pmatrix}\begin{pmatrix}\mathbf{v}_1 \\ \mathbf{v}_2\end{pmatrix} = \begin{pmatrix} B_k\mathbf{u} + \mathbf{b}_1 \\ B_k\mathbf{u} + \mathbf{b}_2\end{pmatrix}$$

which can be written as

$$I - h \begin{pmatrix} a_{1,1}B_k & a_{1,2}B_k \\ a_{2,1}B_k & a_{2,2}B_k \end{pmatrix} \begin{pmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \end{pmatrix} = \begin{pmatrix} B_k\mathbf{u} + \mathbf{b}_1 \\ B_k\mathbf{u} + \mathbf{b}_2 \end{pmatrix}$$

If we use tensor product notation then we have the following

$$(I - A \otimes hB_k) \begin{pmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \end{pmatrix} = \begin{pmatrix} B_k\mathbf{u} + \mathbf{b}_1 \\ B_k\mathbf{u} + \mathbf{b}_2 \end{pmatrix}$$

where I is a $2(N-1) \times 2(N-1)$ identity matrix.

This is the standard method we would use for solving the linear system iteratively by stacking the vector $\mathbf{v}_1$ directly on top of the vector $\mathbf{v}_2$. We do the same thing for the right hand side vectors.

The system we formulate is slightly different, but still equivalent to this standard method. What we do instead is interweave the entries of $\mathbf{v}_1$ and $\mathbf{v}_2$ as follows:

$$\begin{pmatrix} (\mathbf{v}_1)_1 \\ (\mathbf{v}_2)_1 \\ (\mathbf{v}_1)_2 \\ (\mathbf{v}_2)_2 \\ (\mathbf{v}_1)_3 \\ (\mathbf{v}_2)_3 \\ \vdots \\ (\mathbf{v}_1)_{N-1} \\ (\mathbf{v}_2)_{N-1} \end{pmatrix}$$

where $(\mathbf{v}_p)_i$ means the ith entry of the length $(N-1)$ column vector $\mathbf{v}_p$. We interweave the right hand side vectors in a similar fashion. Now since $I_{2(N-1)\times 2(N-1)} = I_2 \otimes I_{(N-1)\times(N-1)} = I_{(N-1)\times(N-1)} \otimes I_2$ and by the way we have interweaved elements of $\mathbf{v}_p$ we can form a block tridiagonal matrix represented by $hB_k \otimes A$. In general, the tensor product between matrices does not commute so $A \otimes hB_k \neq hB_k \otimes A$, but it is permutation equivalent

and what we do with the $\mathbf{v}_p$'s and the right hand side vectors is arrange them accordingly so we can form this equivalent system. Doing so we obtain a linear system of the following form:

$$\begin{pmatrix} \frac{I+2hAD}{k^2} & \frac{-hDA}{k^2} & & & & \\ \frac{-hDA}{k^2} & \frac{I+2hDA}{k^2} & \frac{-hDA}{k^2} & & & \\ & \ddots & \ddots & \ddots & & \\ & & \frac{-hDA}{k^2} & \frac{I+2hDA}{k^2} & \frac{-hDA}{k^2} & \\ & & & \frac{-hDA}{k^2} & \frac{I+2hDA}{k^2} & \end{pmatrix} \begin{pmatrix} (\mathbf{v}_1)_1 \\ (\mathbf{v}_2)_1 \\ (\mathbf{v}_1)_2 \\ (\mathbf{v}_2)_2 \\ (\mathbf{v}_1)_3 \\ (\mathbf{v}_2)_3 \\ \vdots \\ (\mathbf{v}_1)_{N-1} \\ (\mathbf{v}_2)_{N-1} \end{pmatrix}$$

$$= \begin{pmatrix} (B_k\mathbf{u}+\mathbf{b}_1)_1 \\ (B_k\mathbf{u}+\mathbf{b}_2)_1 \\ (B_k\mathbf{u}+\mathbf{b}_1)_2 \\ (B_k\mathbf{u}+\mathbf{b}_2)_2 \\ (B_k\mathbf{u}+\mathbf{b}_1)_3 \\ (B_k\mathbf{u}+\mathbf{b}_2)_3 \\ \vdots \\ (B_k\mathbf{u}+\mathbf{b}_1)_{N-1} \\ (B_k\mathbf{u}+\mathbf{b}_2)_{N-1} \end{pmatrix}$$

This is equivalent to the standard method used by stacking the vectors $\mathbf{v}_p$ on top of each other, but, here, we interweave entries of $\mathbf{v}_1$ and $\mathbf{v}_2$ instead taking advantage of the fact that the tensor product between matrices is commutatively permutation equivalent giving us a block tridiagonal matrix which can be solved by means of a block LU factorization. In practice, we choose to store $\mathbf{v}_p$ not as a single column, but, rather, as a $2 \times N-1$ matrix Ψ arranged so that $\mathbf{v}_1$ is contained in the entire first row of Ψ and $\mathbf{v}_2$ is contained in the entire second row of Ψ. We do this for computational purposes as we are often multiplying Ψ by a 2×2 block matrix.

3.4 Linear System

So what we have obtained by stacking and interweaving the components of $\mathbf{v}_p$ in a column vector is a block tridiagonal matrix with the following form:

$$\begin{pmatrix} \alpha_1 & \beta_1 & & & & \\ \rho_1 & \alpha_2 & \beta_2 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \rho_{N-3} & \alpha_{N-2} & \beta_{N-2} & \\ & & & \rho_{N-2} & \alpha_{N-1} & \end{pmatrix}$$

where $\alpha_i = \dfrac{I + 2hAD}{k^2}$ and $\beta_i = \rho_i = \dfrac{-hDA}{k^2}$ are 2×2 block matrices with A being a 2×2 matrix given from the Butcher tableau of the Radau IIA method. Computationally, it's important to take advantage of the structure of this block matrix. Note that since α_i is the same for all i and $\beta_i = \rho_i$ for all i then we need only store those particular 2×2 matrices instead of a $2N \times 2N$ matrix. This is a significant reduction in both computing time and memory usage.

Now what we can do is implement a block tridiagonal LU factorization to obtain

$$\begin{pmatrix} \alpha_1 & \beta_1 & & & & \\ \rho_1 & \alpha_2 & \beta_2 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \rho_{N-3} & \alpha_{N-2} & \beta_{N-2} & \\ & & & \rho_{N-2} & \alpha_{N-1} & \end{pmatrix}$$

$$= \begin{pmatrix} L_1 & & & & \\ M_1 & L_2 & & & \\ & \ddots & \ddots & & \\ & & M_{N-3} & L_{N-2} & \\ & & & M_{N-2} & L_{N-1} \end{pmatrix} \begin{pmatrix} U_1 & V_1 & & & \\ & U_2 & V_2 & & \\ & & \ddots & \ddots & \\ & & & U_{N-2} & V_{N-2} \\ & & & & U_{N-1} \end{pmatrix}$$

which yields the equations

$$\alpha_1 = L_1 U_1$$
$$\beta_1 = L_1 V_1$$
$$\rho_1 = M_1 U_1$$
$$\alpha_2 = M_1 V_1 + L_2 U_2$$
$$\beta_2 = L_2 V_2$$
$$\rho_2 = M_2 U_2$$
$$\alpha_3 = M_2 V_2 + L_3 U_3$$

etc...

In order to implement the LU factorization we can begin by first finding L_1 and U_1. We can do this since we already know α_1 so we merely need to do an LU factorization of a 2×2 matrix. In the actual implementation we used matlab's built-in function lu to do this. Once we have L_1 we can use β_1 to find V_1. Finally, since we have U_1 we can use ρ_1 to find M_1. After we have found the initial L_1, M_1, U_1, V_1 we can then run a loop to find all the remaining L_i, M_i, U_i, V_i for $i = 2...N-1$ and then do forward and backward substitution to find $\mathbf{v}_p$ for $p = 1, 2$.

4 Testing & Implementation

4.1 Testing the Block LU Factorization

In order to ensure that our code was running correctly we performed tests on each sub-routine. For the block LU factorization it would suffice to check that it worked for a more manageable block tridiagonal matrix of smaller size. Given the matrix blocks along the diagonals we could construct the actual tridiagonal matrix. We could then use Matlab to make a random vector of appropriate length and the matrix-vector product between these two would give us the right hand side vector which we could input into our program along with the block matrices. This was the idea we used in order to test the routine.

In practice, we constructed a zero matrix of size 10×10 and then ran some loops to place the 2×2 matrices α_i along the main diagonal and the 2×2 matrices β_i along the super-diagonal and sub-diagonal. Let's denote this matrix by M. We then used Matlab to create a random column vector which we will denote by V of length 10. Computing the matrix-vector product $M \cdot V = RHS$ gives us a column vector of length 10 which we will denote by RHS. Now we have RHS and the block matrices α_i, β_i which we could input into our program for the block LU factorization.

If the code was working properly, we would expect the output to be a column vector very close to V. In other words, we expect the 2-norm between V and the output from our program to be very small. Indeed the norm was small as we found it to be 1.0336e-15. This provided us with some reassurance that the code was working as intended.

4.2 Testing the Iterative Solver

The iterative solver works by running through the block LU factorization multiple times with each iteration using a different right hand side vector. We first begin the iterative solver by initializing a vector $\mathbf{v}_p$ for $p = 1, 2$. In practice, we simply used a zero vector for the initial $\mathbf{v}_p$. We can then use this vector to form the right hand side of the differential equation and begin the iteration. The output from the first iteration will be used to update

a new right hand side which we could then input back into the block LU factorization. This process would continue until the 2-norm of the difference between successive outputs is below a determined threshold which we set to 10^{-10}. Once we had a vector $\mathbf{v}_p$ for which this happened, we could use this to then update our solution for the next time step.

To ensure the iterative solver was working correctly we would expect the 2-norm between successive $\mathbf{v}_p$'s to decrease in each iteration. If it did not decrease then, clearly, something must be wrong with the code. Using the zero vector as the initial $\mathbf{v}_p$ we found that it took 7 iterations through the process and at each iteration the 2-norm was decreasing. We provide a table of error values per iteration in figure 1.

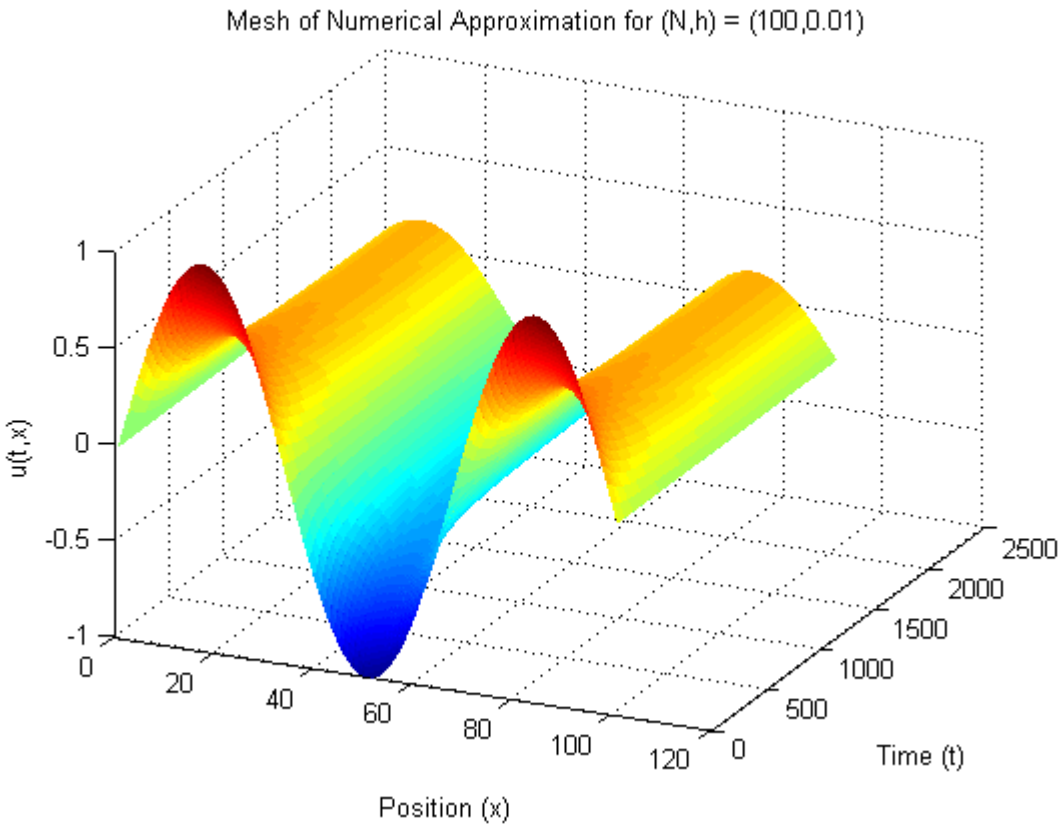
Figure 1:

Iteration Count	1	2	3	4	5	6	7
Error Norm	6.76786257334828	0.083949849189923 8	0.000707248265720 100	4.33425531336423e-06	1.75297438355997e-08	1.09701768185529e-10	1.64086457446198e-12

4.3 Results from the Test Problem

Now we arrive at the part where we actually get to show our results from the program. For the test problem, we set the diffusion coefficient $D = 10^{-2}$ and the reaction function $\Phi = u(1-u^2)$. Our initial conditions were $u(0,x) = \sin\left(\frac{3\pi i}{N}\right)$ for $i = 1, ...N-1$. Recall, the spatial interval was $\Omega = [0,1]$ with boundary conditions $u(t,0) = u(t,1) = 0$. We used this to test four different meshes with different spatial and time steps. The four cases were $(N,h) = (100, 0.01), (200, 0.01), (100, 0.005), (200, 0.005)$ over a time interval of $[0, 20]$. We see a plot of the numerical approximation for $N = (100, 0.01)$ in Figure 2.

Figure 2:

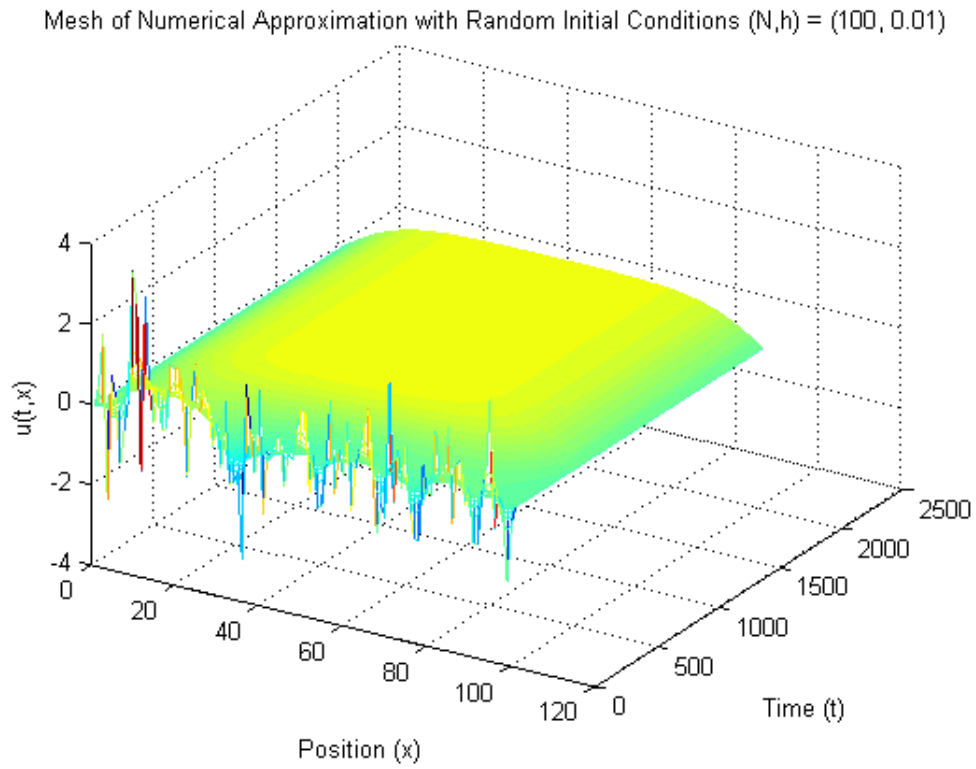


Now this is a numerical approximation to a reaction-diffusion equation so we expect to see diffusion of the solution over time. We can see how it's initially defined at the start of time and as time increases the solution does seem to smooth out some. The large initial oscillations as given from the initial function seems to damp out as time increases, but it does not completely damp out as we still notice the solution oscillating at the terminal point in time. We also ran the same computations for $(N, h) = (200, 0.01), (100, 0.005), (200, 0.005)$, but we omit a plot of the solution because it is nearly identical to the plot given in figure 2.

In order to determine if this is indeed a diffusive process we could use initial conditions composed of random positive and negative entries. This would give us initial conditions that are choppy and sharply peaked. If the

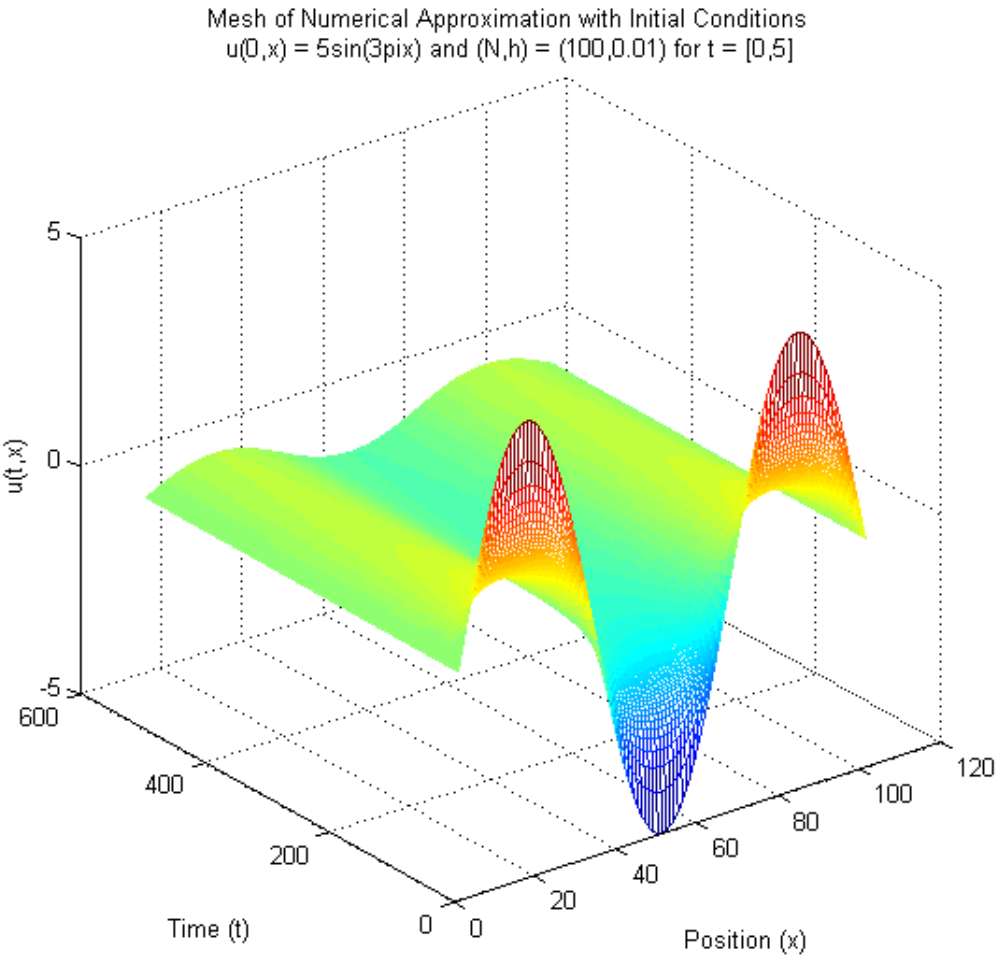
code was working correctly, then we would expect that these sharp peaks would smooth out over time. The results of this test with random initial conditions with $(N, h) = (100, 0.01)$ are given below in figure 3.

Figure 3:



As expected we see the sharply peaked initial conditions smooth out almost immediately. This is much more characteristic of a diffusive process with the solution spreading out and becoming more uniform over time. One thing that we did find surprising was our results of the solution if we changed the amplitude of the initial condition function. For example, if we changed $u(0, x) = \sin\left(\frac{3\pi i}{N}\right)$ to $u(0, x) = 5\sin\left(\frac{3\pi i}{N}\right)$ then we would get the following solution mesh in figure 4.

Figure 4:

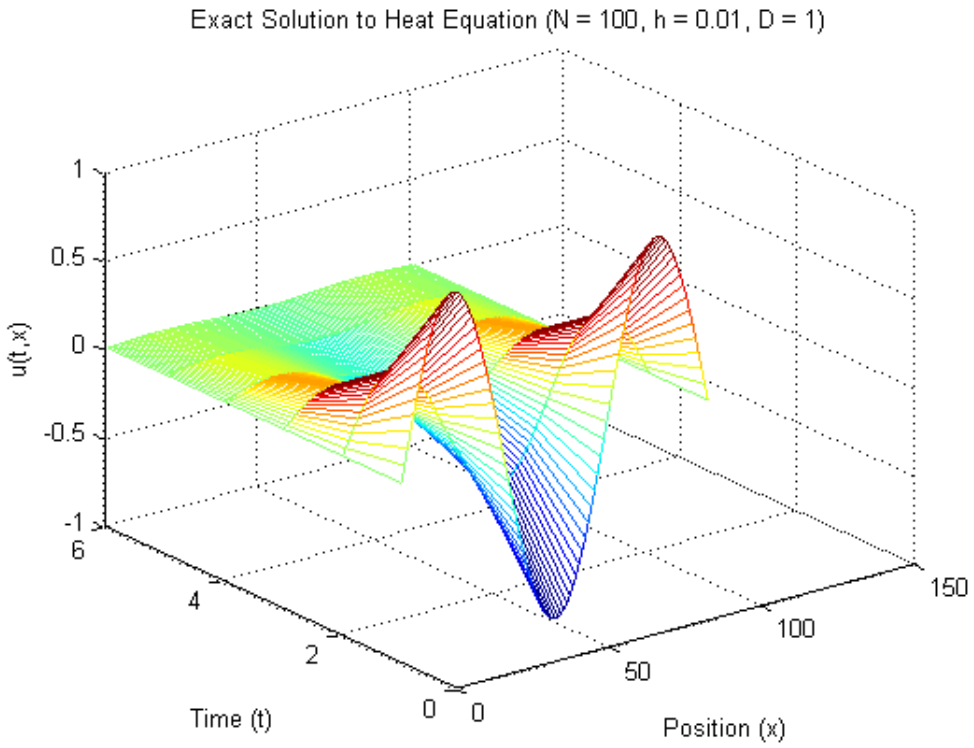


Recall that in figure 2 the oscillations from the initial conditions damped out, but they did not damp out completely. Even at terminal time $t = 20$ we still had oscillations, yet if we increase the amplitude of the initial condition function we obtain a solution that has smaller and smaller oscillations as time increases. We found this to be a surprising result as we would not have expected this to happen.

4.4 Comparing to the Heat Equation

We would like to provide evidence that our code is working correctly. This can be difficult to do because we do not have an analytical solution to our original non-linear differential equation given on p. 3. However, we could check our code by setting the nonlinear reaction term $\Phi = 0$. Doing so would give us the heat equation with initial condition $u(0, x) = \sin(3\pi x)$. Now this we can determine the exact solution to analytically and it is given by $u(t, x) = \sin(3\pi x) \exp(-D(3\pi)^2 t)$. Figure 5A shows the plot of the exact solution to the heat equation and figure 5B gives our numerical approximation to the exact solution over Ω and $0 \leq t \leq 0.05$.

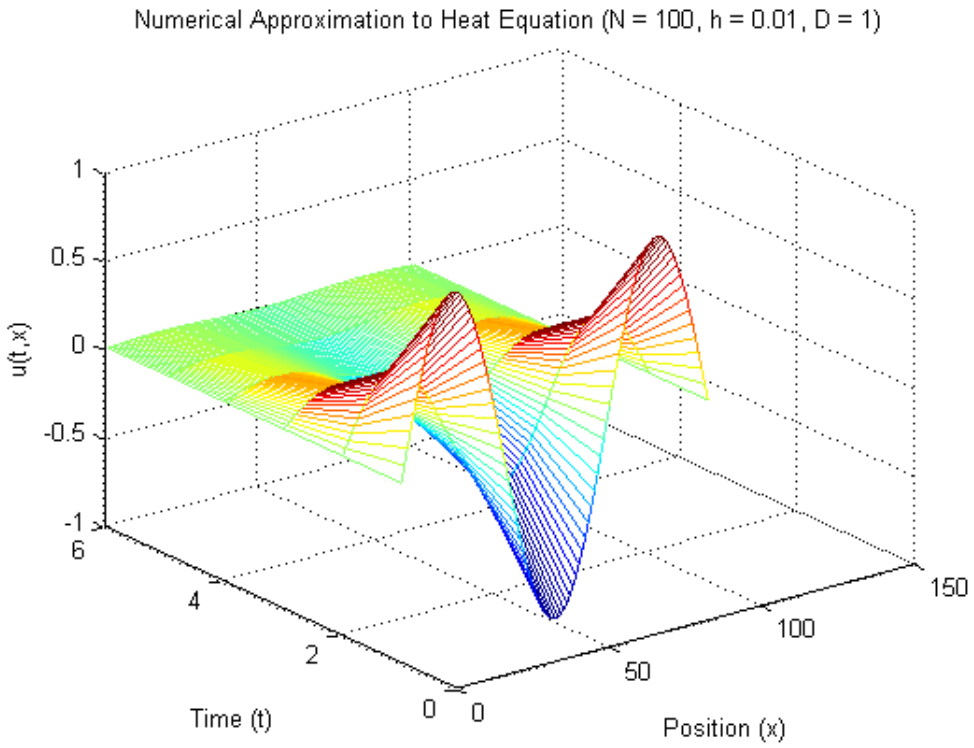
Figure 5A:



As expected, we see the exact solution beginning to decay since for large time values, the exponential term in the exact solution will come to

dominate the solution. This type of behavior is what we would expect to see in our numerical approximation given in figure 5B.

Figure 5B:



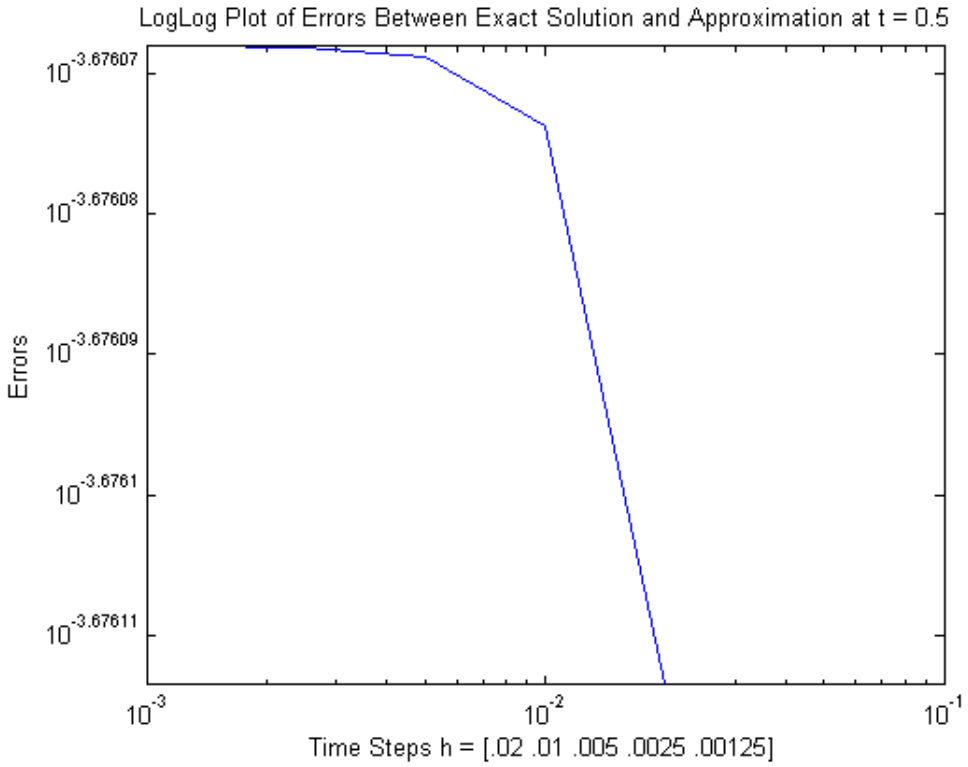
Indeed we note that the exact solution and numerical approximation look nearly identical from these plots. The approximation seems to be smoothing out over time as we would expect given the behavior of the exact solution.

We would like to take a closer look at this though as a simple plot of the two solutions can be misleading. In order to check that the exact solution and our approximation are close we could compute the infinity norm of the difference between the exact solution and approximation at the terminal time. What we give below is the error between the exact solution and our approximation for various values of h keeping N fixed as 100, D as 10^{-2} and $0 \leq t \leq 0.5$. The table in figure 6A gives us the errors for various values of h and a loglog plot of this data is given in figure 6B.

Figure 6A:

Time Step h	.02	.01	.005	.0025	.00125
Errors	0.00021080772 6017559	0.00021082698 7320778	0.00021082940 1050328	0.00021082970 3148008	0.00021082974 0934226

Figure 6B:



5 Conclusion

In conclusion, we would like to make some final remarks about the program. We see from the loglog plot in figure 6B that the convergence seems to agree

with the order of the numerical method used. Radau IIA is a third order method and the loglog plot shows that we have order 3 convergence.

One thing we could do in the future is try to make the code more efficient For the test problem with $\Phi = u(1-u^2)$ and $N = 100$, $h = 0.01$ we found that on a home laptop it took slightly over 68 seconds to compute our numerical approximation. Making a finer mesh by setting $N = 200$ and $h = 0.005$ only took us even longer to compute the approximation with the amount of computing time being just under 311 seconds. There could be some potential room for improvement in the code to make it run quickly and more efficiently.

Finally, we note how we took advantage of the special structure of the block tridiagonal matrix. Each block is a 2×2 matrix and if we actually constructed the matrix out of the 2×2 blocks then we would have a $2(N-1) \times 2(N-1)$ matrix if we ignore the boundary conditions. If we were to store it as a large sparse matrix then for all the non-zero entries we would need to store $N-1$ blocks along the main diagonal and $N-2$ blocks along the sub-diagonal and super-diagonal. So we would need to store a total of $3N-5$ blocks. Now each block is a 2×2 matrix so there are 4 elements in each block so we would store $4(3N-5)$ non-zero entries. In addition to storing the value, we would also need to store both the row number and column number so we would have to actually store $12(3N-5)$ elements if we used a large sparse matrix.

Luckily, we do not have to do that since the block matrices are all the same along the main diagonal and the block matrices along the sub-diagonal and super-diagonal are also the same. Thus, we only need to store two 2×2 blocks giving us only 8 stored digits to do the LU factorization. As we alluded to earlier, this is a significant reduction in memory usage and computational time.