Question 1

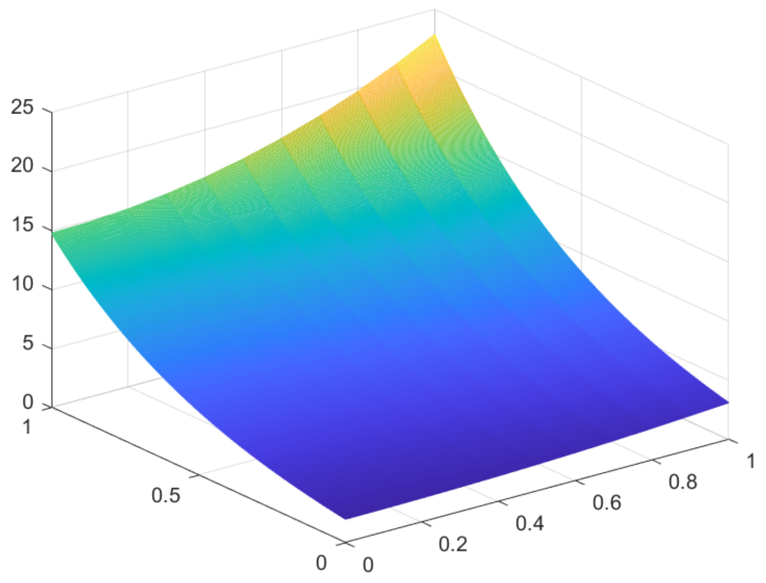
Question 1 Consider the heat equation, for $0 \le x \le 1$, $0 \le t \le 1$,

$$\begin{cases} u_t = 2u_{xx}, \\ u(x,0) = 2cosh(x), \\ u(0,t) = 2e^{2t}, \\ u(1,t) = (e^2+1)e^{2t-1}. \end{cases}$$

This problem has the exact solution:

$$u(x,t) = e^{2t-x} + e^{2t+x}$$

Which looks like this when graphed:



Part 1

Solve the equation using the Forward Difference Method with step sizes h = 0.1 and k = 0.002. Plot the approximate solution $u_{i,j}$ and the error $e_{i,j} = u_{i,j} - u(x_i, t_j)$ using MATLAB mesh command.

My implementation of the forward difference method for the heat equation takes in d (the heat diffusion coefficient, in this case 2), hx and hy (the step sizes in x and y, in this case hx = h = 0.1, hy = k = 0.002), xr and xl (the right and left spatial boundaries, in this case 0 and 1), yb and yt (the bottom and top temporal boundaries, in this case 0 and 1), init (the initial condition, in this case an anonymous function that returns 2cosh(x)), be and te (the bottom and top boundary equations, in this case one anonymous function that returns $2e^{2t}$ and another that returns e^{2t+1} + e^{2t-1}).

It approaches this problem by creating a coefficient matrix (A) that is derived from a system of linear equations with the form:

$$u_{i,j+1} = \sigma * u_{i-1,j} + (1 - 2 * \sigma) * u_{i,j} + \sigma * u_{i+1,j}$$

Which then implies

$$0 = \sigma * u_{i-1,j} + (1 - 2 * \sigma) * u_{i,j} + \sigma * u_{i+1,j} - u_{i,j+1}$$

The latter definition also sets our target value for the equation as 0, which is stored in the value of the b vector corresponding to the equation.

For this problem:

$$\sigma = \frac{D * k}{h^2} = \frac{2 * 0.002}{0.1^2} = 0.4$$

This handles the interior points for all non-initial timesteps, but will also evaluate one time step beyond our problem, so the method has to trim the returned u values to the correct size.

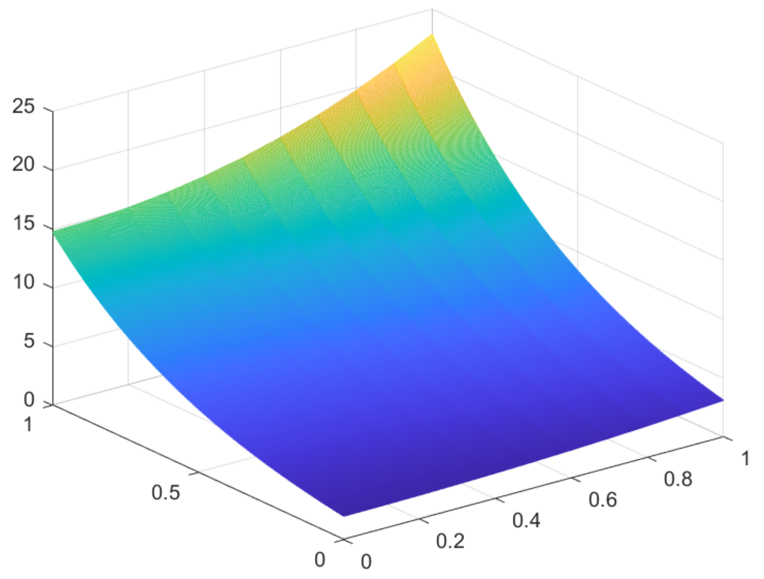
For the boundary and initial points, A equals 1 for that point and b equals the boundary or initial condition, as applicable.

```
% Forward Differences for 2D heat equation script
function [xs, ys, vals] = heatforwarddm(d, hx, hy, xl, xr, yb, yt, init,
be, te)
% calculate the size of our mesh
m = (xr - xl)/hx + 1;
n = (yt - yb)/hy + 1;
mn = m*n;
% get hx^2 in a variable for efficiency
hx2 = hx^2;
% set up our A and b vectors for Au = b
A = zeros(mn, mn);
b = zeros(mn, 1);
% find our xs and ys
xs = xl + (0:m-1)*hx; ys = yb + (0:n-1)*hy;
% calculate our sigma
sigma = d*hy/hx2;
% define our indexer function
index = @(xi, yj) xi+m*(yj-1);
% do the interior points
for i=2:m-1
```

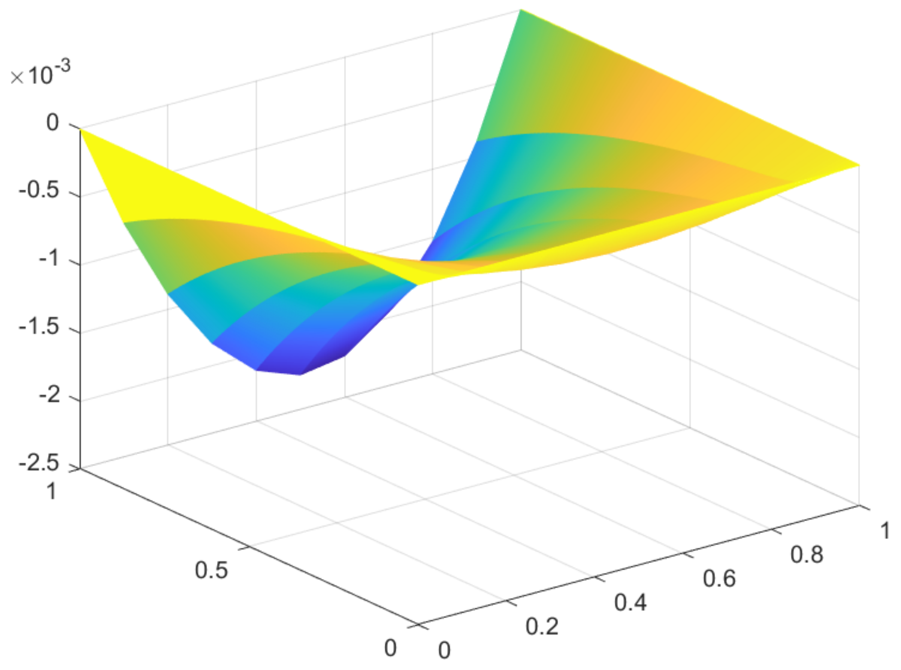
```
    for j=2:n
        A(index(i,j), index(i,j-1)) = 1-2*sigma;
        A(index(i,j), index(i-1, j-1)) = sigma;
        A(index(i,j), index(i+1, j-1)) = sigma;
        A(index(i,j), index(i, j)) = -1;
        b(index(i,j)) = 0;
    end
end
% do the left temporal bound
for i=1:m
    A(index(i,1), index(i,1)) = 1;
    b(index(i,1)) = init(xs(i), ys(1));
end
for j=2:n
    A(index(1,j), index(1,j)) = 1;
    b(index(1,j)) = be(xs(1), ys(j));
    A(index(m,j), index(m,j)) = 1;
    b(index(m,j)) = te(xs(m), ys(j));
end
v = A\b;
vals = reshape(v(1:mn), m, n)';
```

This implementation for the given parameters took 0.110283 seconds and had a mean absolute error of 0.00057874.

The approximate solution is as follows:



And the error is as follows:



As the approximation gets further and further away from the time and space boundaries, the error grows in magnitude, which makes sense, and also stays well conditioned since σ < 0.5.

Part 2

Solve the equation using the Forward Difference Method with step sizes h = 0.1 and k = 0.004. Plot the approximate solution $u_{i,j}$ and the error $e_{i,j} = u_{i,j} - u(x_i, t_j)$ using MATLAB mesh command.

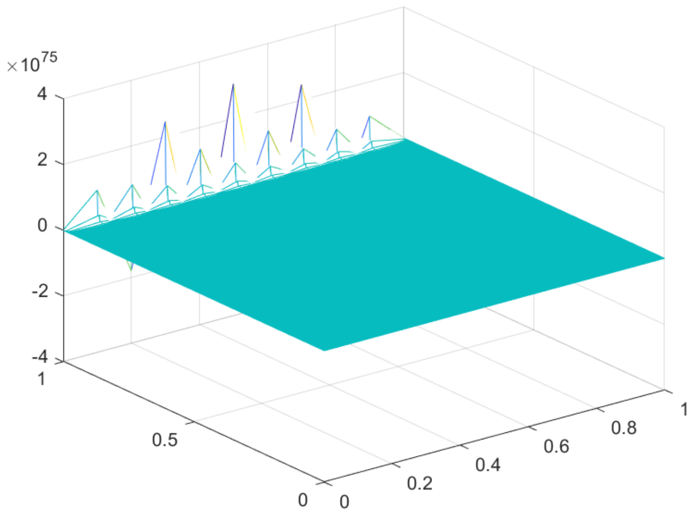
Using the same code as the last part but changing the step sizes to match, the time elapsed was 0.026176 seconds while the average mean error was 1.321710359302625e+73. This combination of step sizes results in a very unstable method.

For this problem:

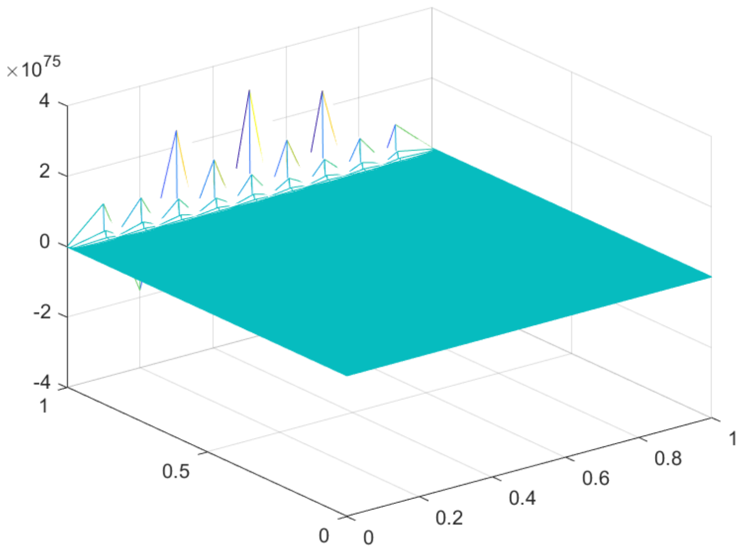
$$\sigma = \frac{D * k}{h^2} = \frac{2 * 0.004}{0.1^2} = 0.8$$

Which means that the system will not be stable, since σ > 0.5.

The mesh of the approximate solution is:



While the error plot is:



On top of the messiness near the beginning, increasing the step size also causes some instability as the method goes on. The magnitude of the error at the final time step is, on average, far larger than on the initial, likely due to how all derivatives of the function increase as x, y, or both increase.

Part 3

Solve the equation using the Backward Difference Method with step sizes h = 0.1 and k = 0.004. Plot the approximate solution ui,j , and the error ei,j = ui,j − u(xi, tj), using MATLAB mesh command.

My implementation of the backward difference method for the heat equation takes in all the same parameters as the code for the forward difference method.

However, the coefficient matrix (A) that is instead derived from a system of linear equations with the form:

$$u_{i,j+1} = -\sigma * u_{i-1,j} + (1 + 2 * \sigma) * u_{i,j} + -\sigma * u_{i+1,j}$$

Which then implies

$$0 = \sigma * -u_{i-1,j} + (1 + 2 * \sigma) * u_{i,j} + \sigma * -u_{i+1,j} - u_{i,j+1}$$

Similarly, the latter definition also sets our target value for the equation as 0, which is stored in the value of the b vector corresponding to the equation.

This handles the interior points for all non-initial timesteps, but will also evaluate one time step beyond our problem, so the method has to trim the returned u values to the correct size.

For the boundary and initial points, A equals 1 for that point and b equals the boundary or initial condition, as applicable.

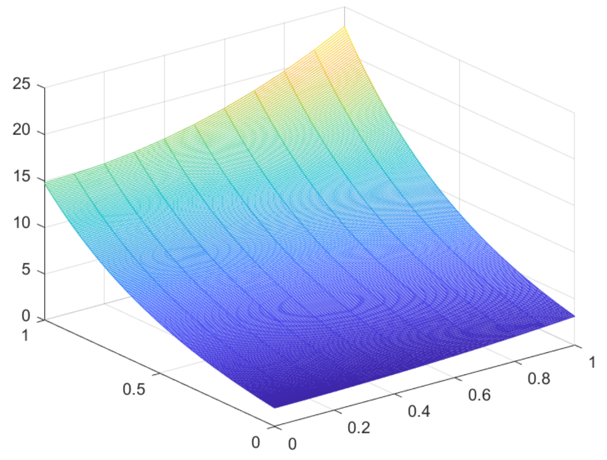
The code is as follows:

```
% Forward Differences for 2D heat equation script
function [xs, ys, vals] = heatbackwarddm(d, hx, hy, xl, xr, yb, yt, init, be, te)
% calculate the size of our mesh
m = (xr - xl)/hx + 1;
n = (yt - yb)/hy + 1;
mn = m*n;
% get hx^2 in a variable for efficiency
hx2 = hx^2;
% set up our A and b vectors for Au = b
A = zeros(mn, mn);
b = zeros(mn, 1);
% find our xs and ys
xs = xl + (0:m-1)*hx; ys = yb + (0:n-1)*hy;
% calculate our sigma
sigma = d*hy/hx2;
% define our indexer function
index = @(xi, yj) xi+m*(yj-1);
% do the interior points
for i=2:m-1
   for j=2:n
       A(index(i,j), index(i,j)) = 1+2*sigma;
       A(index(i,j), index(i-1, j)) = -sigma;
       A(index(i,j), index(i+1, j)) = -sigma;
       A(index(i,j), index(i, j-1)) = -1;
       b(index(i,j)) = 0;
   end
```

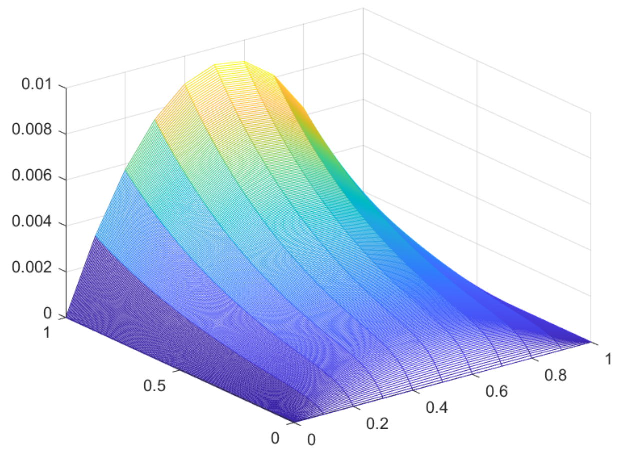
```
end
% do the left temporal bound
for i=1:m
   A(index(i,1), index(i,1)) = 1;
   b(index(i,1)) = init(xs(i), ys(1));
end
for j=2:n
   A(index(1,j), index(1,j)) = 1;
   b(index(1,j)) = be(xs(1), ys(j));
   A(index(m,j), index(m,j)) = 1;
   b(index(m,j)) = te(xs(m), ys(j));
end
v = A\b;
vals = reshape(v(1:mn), m, n)';
```

This implementation with the given parameters took 0.110283 seconds and had an average absolute error of 0.0023875. Due to the unconditional stability of this method, it is well adjusted to doing the larger step sizes. Even if the smaller step size of the forward difference method is higher accuracy, the ability of the backwards difference method to run higher step sizes without too notable of an increase in error makes this algorithm generally better.

The approximate solution looked like this:



And the error looked like this:



The scale on the z axis shows that this error is really small, especially considering the spikes that the forward difference method showed in its error.

Part 4

Solve the equation using the Crank-Nicolson Method with step sizes h = k = 0.1. Plot the approximate solution ui,j , and the error ei,j = ui,j −u(xi, tj), using MATLAB mesh command.

Again, this method used all the same parameters as the forward difference method.

This time, the coefficient matrix (A) that is instead derived from a system of linear equations with the form:

$$-\frac{\sigma}{2} * u_{i-1,j+1} + (1+\sigma) * u_{i,j+1} - \frac{\sigma}{2} * u_{i+1,j+1}$$
$$= \frac{\sigma}{2} * u_{i-1,j} + (1-\sigma) * u_{i,j} + \frac{\sigma}{2} * u_{i+1,j}$$

Which then implies

$$0 = -\frac{\sigma}{2} * u_{i-1,j} + (1-\sigma) * u_{i,j} - \frac{\sigma}{2} * u_{i+1,j}$$
$$+ \frac{\sigma}{2} * u_{i-1,j+1} - (1+\sigma) * u_{i,j+1} + \frac{\sigma}{2} * u_{i+1,j+1}$$

Similarly, the latter definition also sets our target value for the equation as 0, which is stored in the value of the b vector corresponding to the equation.

This handles the interior points for all non-initial timesteps, but will also evaluate one time step beyond our problem, so the method has to trim the returned u values to the correct size.

For the boundary and initial points, A equals 1 for that point and b equals the boundary or initial condition, as applicable.

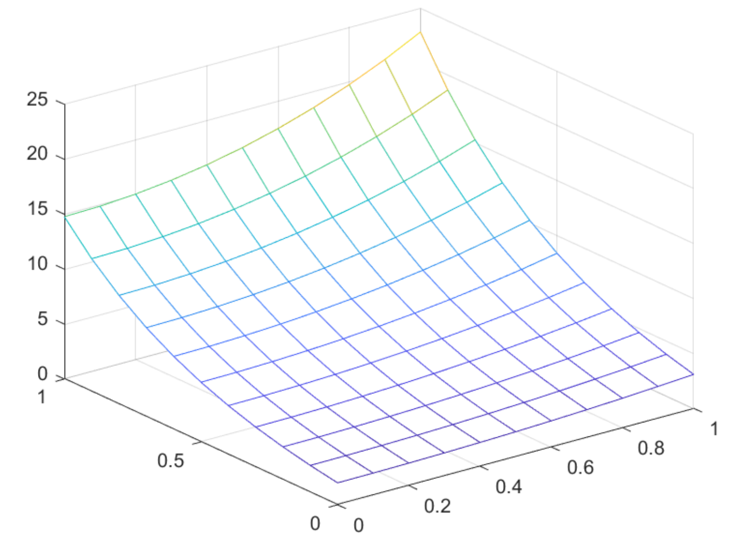
The code is as follows:

```
% Forward Differences for 2D heat equation script
function [xs, ys, vals] = heatcranknicholson(d, hx, hy, xl, xr, yb, yt,
init, be, te)
% calculate the size of our mesh
m = (xr - xl)/hx + 1;
n = (yt - yb)/hy + 1;
mn = m*n;
% get hx^2 in a variable for efficiency
hx2 = hx^2;
% set up our A and b vectors for Au = b
A = zeros(mn, mn);
b = zeros(mn, 1);
% find our xs and ys
xs = xl + (0:m-1)*hx; ys = yb + (0:n-1)*hy;
% calculate our sigma
sigma = d*hy/hx2;
% define our indexer function
index = @(xi, yj) xi+m*(yj-1);
% do the interior points
for i=2:m-1
   for j=2:n
       % this time step
       A(index(i,j), index(i,j-1)) = 1-sigma;
       A(index(i,j), index(i-1,j-1)) = sigma/2;
       A(index(i,j), index(i+1,j-1)) = sigma/2;
       % value of our solution
       b(index(i,j)) = 0;
       A(index(i,j), index(i,j)) = -1-sigma;
       A(index(i,j), index(i-1,j)) = sigma/2;
       A(index(i,j), index(i+1,j)) = sigma/2;
   end
end
% do the left temporal bound, fill all values of x
for i=1:m
   A(index(i,1), index(i,1)) = 1;
   b(index(i,1)) = init(xs(i), ys(1));
end
% handle the top and bottom bounds
for j=2:n
   A(index(1,j), index(1,j)) = 1;
   b(index(1,j)) = be(xs(1), ys(j));
   A(index(m,j), index(m,j)) = 1;
```

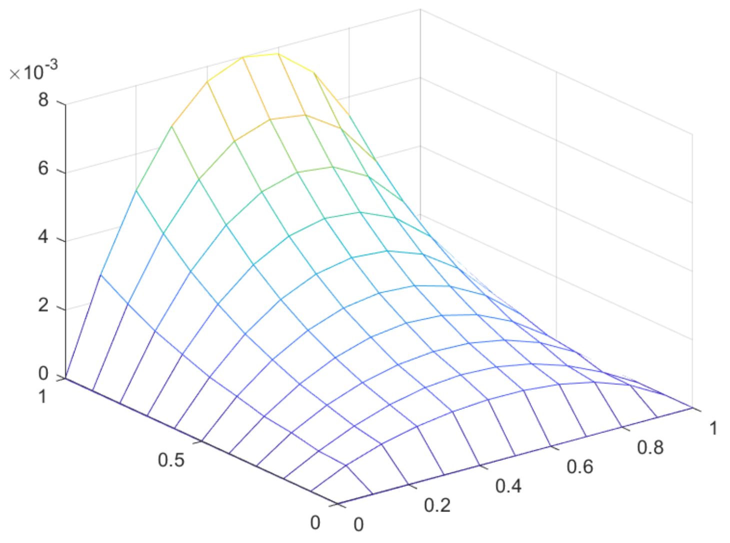
```
    b(index(m,j)) = te(xs(m), ys(j));
end
v = A\b;
vals = reshape(v(1:mn), m, n)';
```

This implementation with these parameters took 0.004083 seconds and had an average absolute error of 0.0020949. Given the large step size, I'm surprised the error is so low.

The approximate solution looked like this:



The error looked like this:



The Crank-Nicolson method has a lot of elements of the backwards difference method, so it is reasonable that the two error plots look very similar.

Question 2

Consider the wave equation, for $-2 \leq x \leq 2$, $0 \leq t \leq 1$,

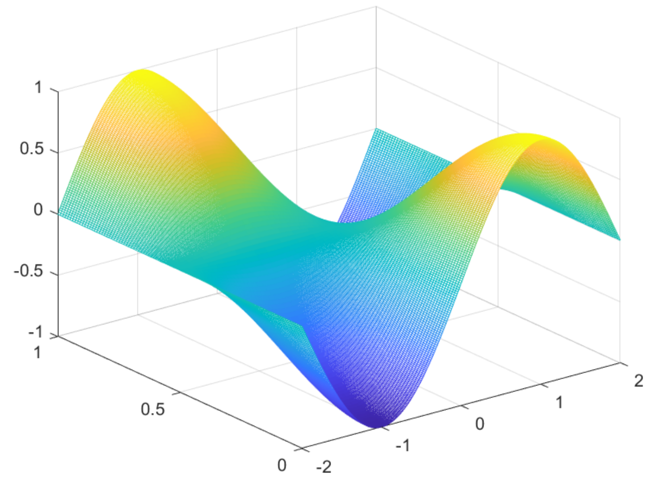
$$\begin{cases} u_{tt} = 4u_{xx}, \\ u(x,0) = sin(\pi x/2), \\ u_t(x,0) = 0, \\ u(-2,t) = 0, \\ u(2,t) = 0. \end{cases}$$

For this question, I created two different methods. The first is an iterative solution while the second is the matrix implementation of that iterative solution. The iterative solution is more intuitive to understand and much faster since it doesn't have to solve a matrix equation, but also is far less accurate since it doesn't have all of the information on the system.

The exact solution is:

$u(x,t) = sin(\frac{\pi}{2} * x) * cos(\pi * t)$

Which looks like this when graphed:



Part 1

Solve the equation using the Finite Difference Method with step sizes h = 0.05 and k = 0.02. Plot the approximate solution ui,j , and the error ei,j = ui,j − u(xi, tj) using MATLAB mesh command.

The sigma for this problem is:

$$\sigma = \frac{c * k}{h} = \frac{2 * 0.02}{0.05} = 0.8$$

So this problem will be CFL stable.

Part 2

Solve the equation using the Finite Difference Method with step sizes h = 0.05 and k = 0.03. Plot the approximate solution ui,j , and the error ei,j = ui,j − u(xi, tj) using MATLAB mesh command.

The sigma for this problem is:

$$\sigma = \frac{c * k}{h} = \frac{2 * 0.02}{0.05} = 1.2$$

So this problem will not be CFL stable.

Part 1 - Iterative Solution

The iterative solution repeatedly does the following formula for j = 1, 2, …, n:

$$U_{j+1} = AU_j - U_{j-1} + \sigma * S_j$$

U_j is our values of U at time step j, and is a vector that is m long. A is a coefficient matrix derived from the finite difference method that is a tridiagonal matrix of 2 - 2σ along the main diagonal, and σ along the diagonals just above and below that. σ, in this case is c*k/h = 0.8. S_j holds the boundary conditions (S_j^0 is the left boundary condition and S_j^m is the right boundary condition, all other spots are zero) and is reevaluated at each time step.

However, this also needs to be evaluated for t = 0 to determine our first move, which requires some moving around and taking advantage of the fact that:

$$u_t(x, 0) = g(x)$$

This fact can be used to approximate U_{-1} since:

$$u_t(x, 0)\frac{u(x, 1) - u(x, -1)}{2 * k} = g(x)$$

Which then implies:

$$u(x, -1) = u(x, 1) - 2 * k * g(x)$$

We can use this to define:

$$U_1 = \frac{AU_0 + 2 * k * g(x) + \sigma * S_0}{2}$$

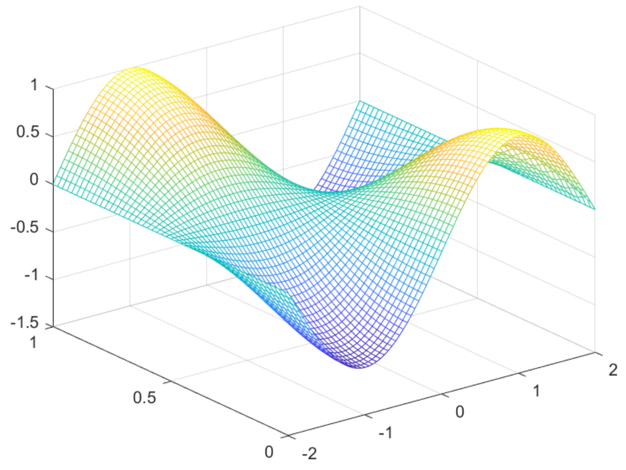
```matlab
function [xs, ys, us] = iterwavefdm(c, hx, hy, xl, xr, yb, yt, init, ut,
le, re)
% calculate the size of our mesh
m = round((xr - xl)/hx) + 1;
n = round((yt - yb)/hy) + 1;
if m <= 3
   disp("m is too small! Try smaller distance step size")
   return
end
if n < 3
   disp("n is too small! Try small time step size")
   return
end
% set up our current A and s matrices
A = zeros(m-2, m-2);
s = zeros(m-2,1);
% set up our us matrix
% Us(i, j) holds ui at time tj for i=1,...,m-1
% each column is a us vector (will be transposed)
us = zeros(m, n+1);
% find our xs and ys
xs = xl + (0:m-1)*hx; ys = yb + (0:n-1)*hy;
% u is our us at the current timestep, starting with initial condition
us(:,1) = init(xs)';
u = us(2:m-1, 1);
% calculate our sigma and sigma squared
sigma = c*hy/hx;
sigma2 = sigma^2;
% set up our A matrix since it doesn't change
% handle our left values
A(1, 1) = 2-2*sigma2;
A(1, 2) = sigma2;
for i=2:m-3
   % handle inner values
   A(i, i) = 2-2*sigma2;
   A(i, i-1) = sigma2;
   A(i, i+1) = sigma2;
end
% handle the right values
A(m-2,m-2) = 2 - 2*sigma2;
A(m-2,m-3) = sigma2;
% handle our initial timestep (j0)
s(1) = sigma2 * init(xs(1), ys(1));
```

```
s(m-2) = sigma2 * init(xs(m-2), ys(1));
% create Uj1
u = (A*u - 2*hy*ut(xs, ys(1)) + sigma2 * s)/2;
us(2:m-1, 2) = u;
us(1, 2) = le(xs(1), ys(2));
us(m, 2) = re(xs(m), ys(2));
for j=3:n
   s(1) = us(1, j-1);
   s(m-2) = us(m, j-1);
   u(1:m-2) = A*us(2:m-1, j-1) - us(2:m-1, j-2) + sigma2 * s;
   us(2:m-1, j) = u(1:m-2);
   us(1, j) = le(xs(1), ys(j));
   us(m, j) = re(xs(m), ys(j));
end
us = us(:, 1:n)';
% vals = reshape(v(1:mn), m, n)';
```

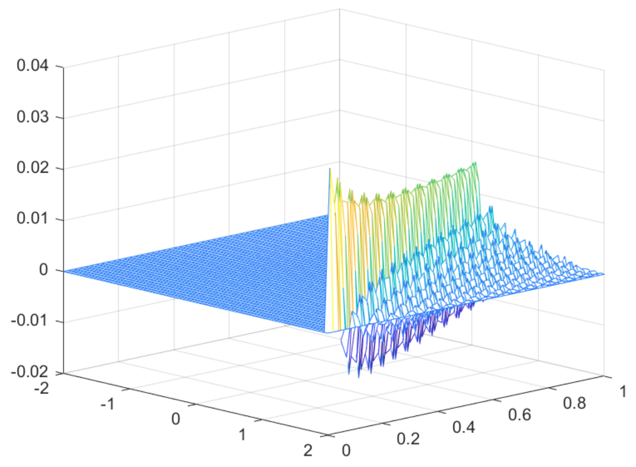
To be able to store all the u values, there was some trickery with extracting the inner m-2 elements for a given time step since the formula only holds for the interior points, but it worked well.

For this set of parameters for this problem, the elapsed time was 0.002644 seconds and the mean error was 0.0012254.

The approximate solution was:



And the error was:



There was this weird set of waves for the left side of the error plot, which is a result of a small bump on the approximate solution. However, this error is very small and decreases as it nears the edge.

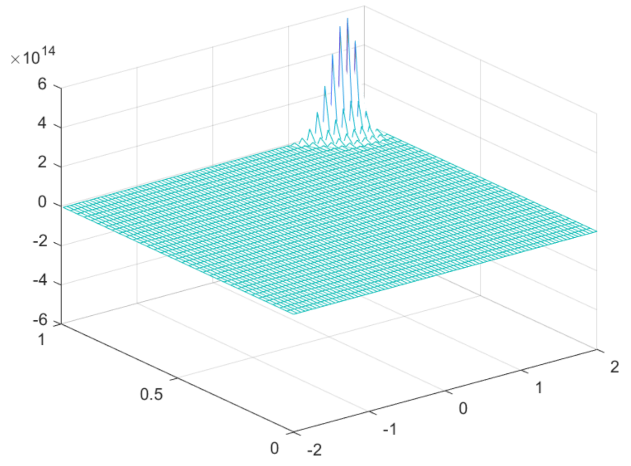
Part 2 - Iterative Solution

Due to the increased step size in k, the new sigma is 1.2, which means that this method is not stable anymore.

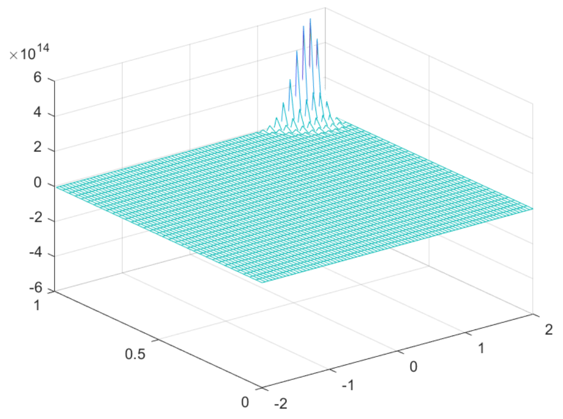
As such, low accuracy is very expected. The code for this part is the same as the first part, but the step size was changed.

The time to run this method was 0.005958 seconds and the average absolute error was 2,636,759,906,907.694, or just over 2.6 trillion.

The approximate solution looked like this:



And the error looked like this:



I am far more surprised that it was only one region of instability as opposed to a whole mess, but obviously the error got up there towards the end, about up to 6 x 10^{14}.

Part 1 - Matrix Solution

The matrix solution directly implements the math from the iterative solution, but differs from the finite difference approaches to the heat equation because we need to manually set up our first step as well as our initial condition. We can do this by taking the first step of the iterative solution, multiplying both sides by two, and rearranging to get:

$$2 * k * g(x) = AU_0 - 2 * U_1 + \sigma * S_0$$

This also means that our first value of b is 2 * k * g(x).

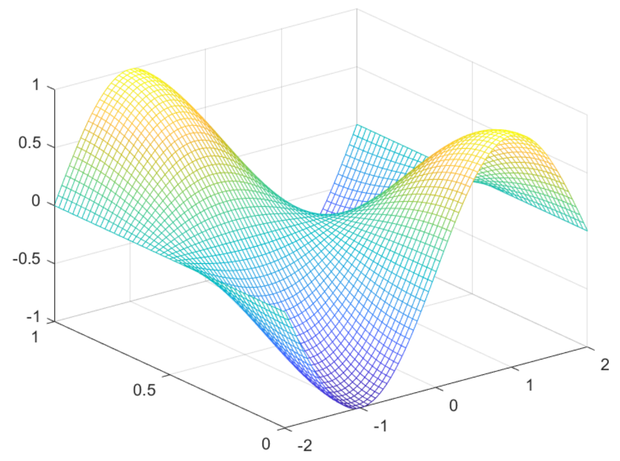
Once the interior points of A are set up, we can set up our boundary points and then solve.

```
% Finite Differences for 2D heat equation script
function [xs, ys, vals] = matrixwavefdm(c, hx, hy, xl, xr, yb, yt, init,
ut, le, re)
% calculate the size of our mesh
m = round((xr - xl)/hx) + 1;
n = round((yt - yb)/hy) + 1;
mn = m*n;
% set up our A and b vectors for Au = b
A = zeros(mn, mn);
b = zeros(mn, 1);
% find our xs and ys
xs = xl + (0:m-1)*hx; ys = yb + (0:n-1)*hy;
% calculate our sigma and sigma squared
sigma = c*hy/hx;
sigma2 = sigma^2;
```

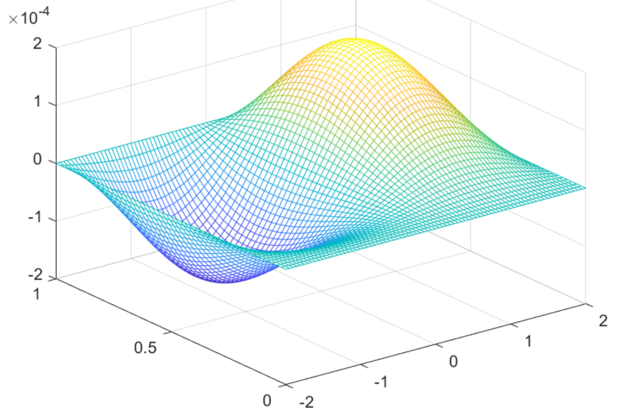
```
% define our indexer function
index = @(xi, yj) xi+m*(yj-1);
% do the interior points all non-initial timesteps
for i=2:m-1
   % handle the first step
   A(index(i,2), index(i, 1)) = 2-2*sigma2;
   A(index(i,2), index(i-1, 1)) = sigma2;
   A(index(i,2), index(i+1, 1)) = sigma2;
   A(index(i,2), index(i, 2)) = -2;
   b(index(i,2)) = 2*hy*ut(xs(i),ys(2));
   for j=3:n
       A(index(i,j), index(i,j-1)) = 2-2*sigma2;
       A(index(i,j), index(i-1, j-1)) = sigma2;
       A(index(i,j), index(i+1, j-1)) = sigma2;
       A(index(i,j), index(i, j-2)) = -1;
       A(index(i,j), index(i, j)) = -1;
       b(index(i,j)) = 0;
   end
end
% fill in the initial condition
for i=1:m
   A(index(i,1), index(i,1)) = 1;
   b(index(i,1)) = init(xs(i), ys(1));
end
% do the left and right for all values of j
for j=2:n
   A(index(1,j), index(1,j)) = 1;
   b(index(1,j)) = le(xs(m), ys(j));
   A(index(m,j), index(m,j)) = 1;
   b(index(m,j)) = re(xs(m), ys(j));
end
v = A\b;
vals = reshape(v(1:mn), m, n)';
```

For this set of parameters for this problem, the elapsed time was 0.116592 seconds seconds and the mean error was 5.7015e-05. This means that this method was about 40x slower than the iterative method, but also about 20x more accurate than the iterative method.

The approximate solution was:



The error was:



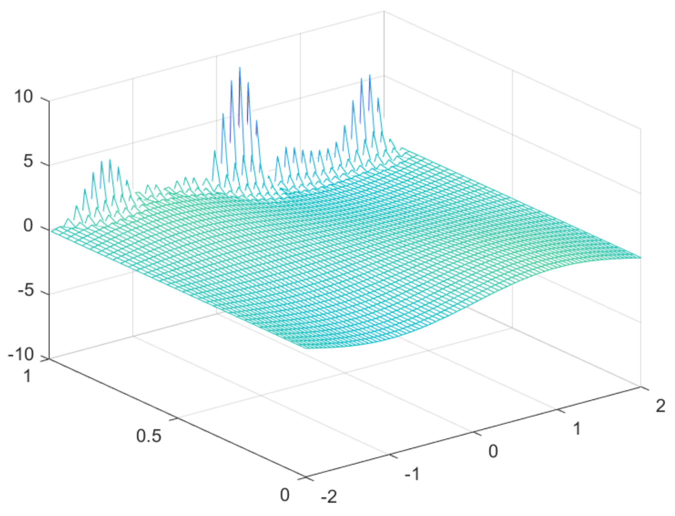
Which is a very nice shape and also proportionate to the time derivative of the exact solution.

Part 2 - Matrix Solution

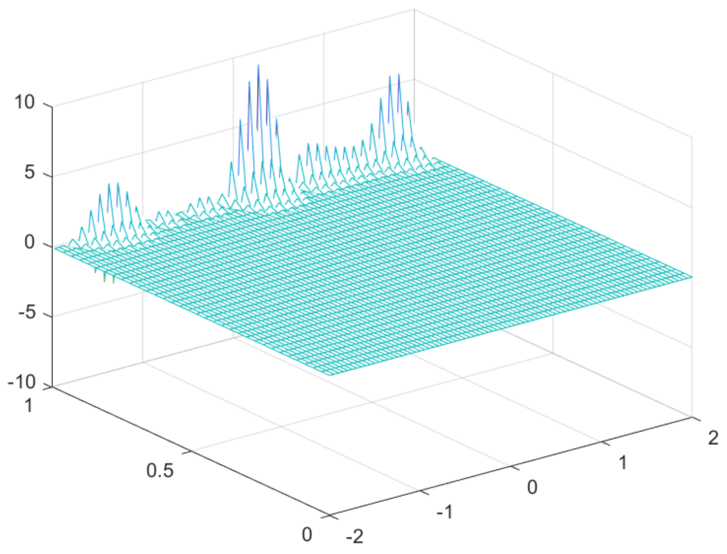
Due to the increased step size in k, the new sigma is 1.2, which means the problem is not CFL stable. However, this approach is not as affected by this condition, though it does still show some unstable behavior.

The time to run this method was 0.026722 seconds and the average absolute error was 0.10411. While this approach may have taken about 25 times as long, the error is smaller by about 14 orders of magnitude. This matrix approach is far more accurate, even for large step sizes, but is not free from a lack of stability.

The approximate solution looked like this:



And the error looked like this:



For the regions farther away from the upper time bound, the approximate solution shows the expected pattern, but unfortunately that breaks in the last few time steps.