Numerical Solution of Partial Differential Equations

Timothy Holmes

Department of Physics, DePaul University

June 1, 2017

Abstract

This lab was carried out to solve some partial differential equations numerically. Two different methods were used in this lab including, the Jacobi method and Gauss-Shield Method. When the partial differential equations is easy to solve and the inputs are minimal, the Jacobi method and the Gauss-Seidel method have essentially the same efficiency in their computation. However, with a longer computation the Gauss-Seidel method was found to be more efficient than the Jacobi method, with the Gauss-Seidel method have on average a 20% faster computation time than the Jacobi Scheme.

I Introduction

Not all partial differential equations (PDEs) are able to be solved analytically. When it is not possible to solve PDEs on paper, numerical methods are applied to help us understand some of these difficult problems. With the use of Matlab four different codes using two different methods were used for solving PDEs. This includes the Jacobi scheme and the Gauss-Seidel method. Both of these methods allow for a alternative computational approach, with the Gauss-Seidel method being the more efficient method. Using the implicit methods the Laplace equation and the Poisson equation are solved on finite bounds. Using the explicit methods the heat equation is solved on finite bounds.

The first equation that is solved for is the Laplace equation in one dimension. The Laplace equation is one dimension is solved by using the Jacobi scheme. The one dimentional case for the Jacobi method is as follows,

$$u_{xx} = 0. \tag{1}$$

Where the second derivative with respect to space is equal to zero.
To solve this derivative on a grid the points have to be approximated. The notation used is $x = x_i$ and the function of the grid point as $u(x_i) = u_i$ First, by the first derivative and then by the second derivative of u_i. The second derivative u_i'' is given by,

$$u_i'' \approx \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}. \tag{2}$$

Where h is $u_{i+1} - u_i$ and u_i is a grid point. This equation is used to just an approximation. However, it is used to solve the Laplace equation.
When reducing the second derivative of u_i from equation (2), the equation simply becomes

$$u_i = \frac{1}{2}u_{i-1} + u_{i+1} \tag{3}$$

The Laplace equation in two dimensions is given by,

$$u_{xx} + u_{yy} = 0. \tag{4}$$

Where the second derivative of x and the second derivative of y are equal to zero.
The Laplace equation in two dimensions is just an extension of the one dimensional case. The new notation for the to dimensional case is $u_{i,j} = u(x_i, y_i)$. Substituting the notation into equation (4) gives

$$\frac{1}{4}[u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}] \tag{5}$$

Where the for loop can loop over each interval in a nested for loop.

The Poisson equation is modified from equation the Laplace equation. The Poisson equation is different given its boundary conditions that change its location on the grid. The Poisson and boundary conditions are given by

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h_x^2} + \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h_x^2} = \begin{cases} A & (x,y) \neq (x_i, y_j) \\ B & (x,y) = (x_i, y_j) \end{cases} \tag{6}$$

Where A and B are boundary conditions, x any y are initial conditions, and x_i and yi are the coordinates on the grid.
The Heat equation is given by,

$$u_t = \alpha^2 u_{xx}. \tag{7}$$

Where α is some constant that controls the flow of the heat in a material, the change of this constant is not observed in this lab.
The heat equation involves heat, position, and time. When modeling this PDE, the observer is able to tell the given temperature at a certain time and position.

Making this an important equation for engineering and other fields. Substituting in the notation previously used we get

$$u_{i,j+1} = (1-2s)ui, j + s(u_{i+1,j} + u_{i-1,j}). \tag{8}$$

Where

$$s = \frac{\alpha^2 k}{h^2},$$

k is the grid step in time, and h is the grid step in space.

II Method

II.I Laplace Equation and the Jacobi Scheme

The Jacobi scheme in one dimension using equation (1) was the first equation and the first method used in this lab. The Jacobi scheme takes advantage of cycling through variables in the loop.

Using equation (1) a graph was generated to plot U(x) vs. x. When the boundary conditions are u(0) = 10, u(1) = 10, $\Delta x = 0.1$, and tolerance is 0.001. Where Δx is the grid spacing of each point and tolerance allows for the loop to continue running longer, allowing for a more accurate result. Overall, using the tic toc command it took 0.000509 seconds for these specific initial conditions to run. The Matlab code is shown in the appendix and the graph generated is shown below.

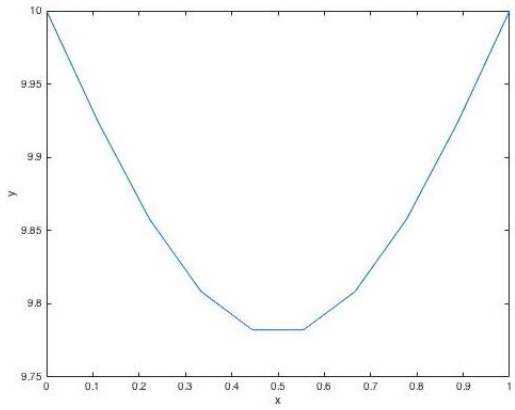
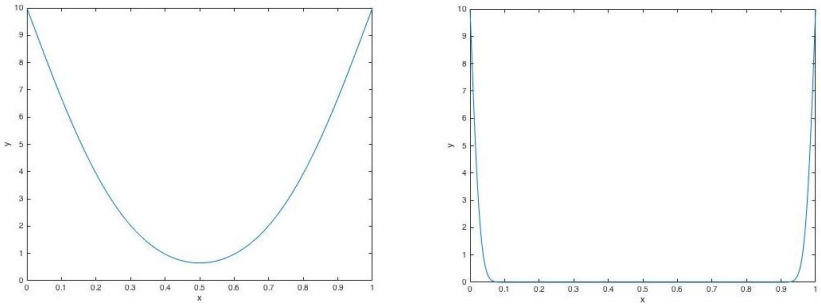


Figure 1: Jacobi 1D with initial conditions when $\Delta x = 0.1$ and the bar is 1 length.

The figure of this graph resembles a parabola and its clear what the boundary conditions u(0) = 10 and u(1) = 10. The reason why the curve is not smooth is because the grid size is very large in perspective to an even smaller number. This graph in application could simulate how heat is transferred trough a one dimensional bar.

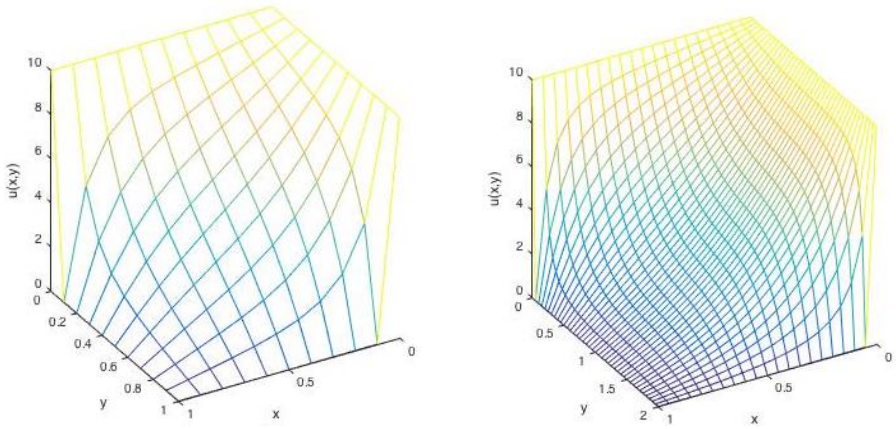
Changing all of the inputs in the code results in different graphs. One of the main differences is when the grid size $\Delta x = 0.0001$ is made significantly smaller. This change made the for loop run through more iterations and as a result the time was 0.004153 seconds, slightly slower than the initial conditions. However, changing the grid size did have it compute more, the graph did generating a smoother paraboloid and also took slightly longer.



(a) Jacobi 1D $\Delta x = 0.01$ (b) Jacobi 1D $\Delta x = 0.001$

Figure 2: (a) Jacobi 1D when $\Delta x = 0.01$ and the bar is 1 length. (b) Jacobi 1D when $\Delta x = 0.001$ and the bar is 1 length.

The Jacobi method was also used for solving the Laplace equation in two dimensions, equation (4). The process for solving this PDE was similar to the one dimensional case, however, in the Matlab code there are now two for loops. One loop that will iterate through all the x's and one loop that will iterate through all of the y's. The initial conditions for this PDE were u(0,y) = 10, u(x,0) = 10, $\Delta x = 0.5$, $\Delta y = 0.5$, and the tolerance = 0.001. These conditions took the code 0.040200 seconds to run. The graph for these initial conditions is shown below.



(a) Jacobi 2D with initial conditions.

(b) Jacobie 2D $\Delta x = \Delta y = 0.001$.

Figure 3: (a) The initial values for the Jacobi 2D. (b) The Jacobi 2D with rectangular boundaries.

The Laplace equation for the Jacobi Scheme is on a standard square shown in Figure 3 (a). This plot was plotted with the initial conditions given for the experiment. This also includes the boundary conditions $u(x, y) = 10$ displayed on the z axis, while the x and y axis are 1 by 1 forming a square. The specific grid size for x and y was 0.1. The tic toc command found that it took 0.002068 seconds seconds to run this code with the initial conditions. In figure 3 (b) the boundary conditions for x and y have changed. It is now plotted on a rectangular plane. The grid size of the Laplace equation was also changed to 0.001. If $\Delta x \neq \Delta y$ then the grid size will come out as rectangles just like the boundary conditions. This alteration also changes the numerical scheme and will result in a different graph. This alteration drastically changes the outcome of the graph and makes it indistinguishable from the initial conditions from Figure 3 (a). Matlab requires more computational power when the grid size is this large since the program is iterating through more values. This also means that it takes the program longer to plot the graph, the tic toc command can confirm this since it took Matlab 2.654223 seconds to run with the grid size this large. This statement is true for all conditions that require more processing power.

Essentially everything that can be changed in the Jacobi 2D include the boundary conditions. In detail where the x-axis starts or x_0 and where x will end. Also the y-axis starts or y_0 and where y will end. Finally, when $u(x, y)$ starts and ends is also very important. An example of an alteration is shown below.

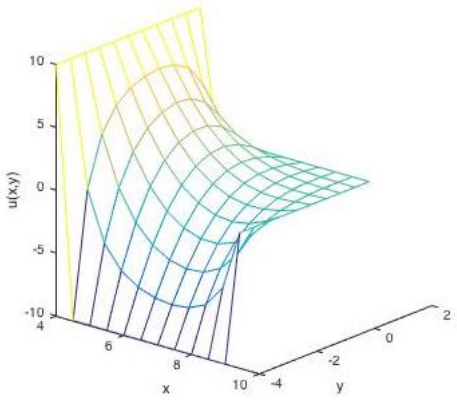


Figure 4: The Jacobi 2D with adjusted values.

By altering and using different values for the Laplace equation, different results from the normal are able to be discovered. There are many different ways of altering this equation and it is not obvious how big of a difference boundary conditions control the outcome.

II.II Poisson Equation and the Gauss-Seidel Method

The Gauss-Seidel method calculates the Laplace equation (4) in this experiment. However, the Gauss-Seidel method is different from the Jacobi Scheme when it comes to how it is calculated. When the code is run and is going through the for loop, the value is now updated and stored. This keeps track of the calculation and is a more efficient way of coding.

The process and initial conditions are exactly the same as they were with the Jacobi Scheme. However, the computational methods are incredibly different. With the same conditions and the same equation the time for the Gauss-Seidel method was 0.000408 seconds. Compared to the Jacobi Scheme that took 0.002068 seconds. This is only a 20% increase in speed and at these speeds it might not seem like much. But with a larger grid and a longer computation time this is significant, especially when it is important to get an accurate graph in a shot period of time.

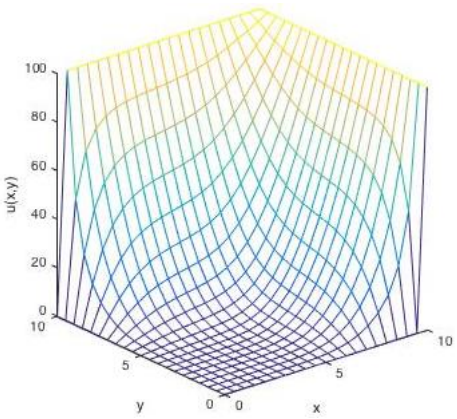
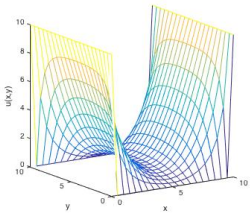
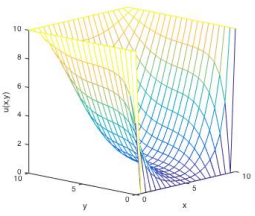


Figure 5: The Gauss-Seidel method solving the Laplace equation with initial conditions.



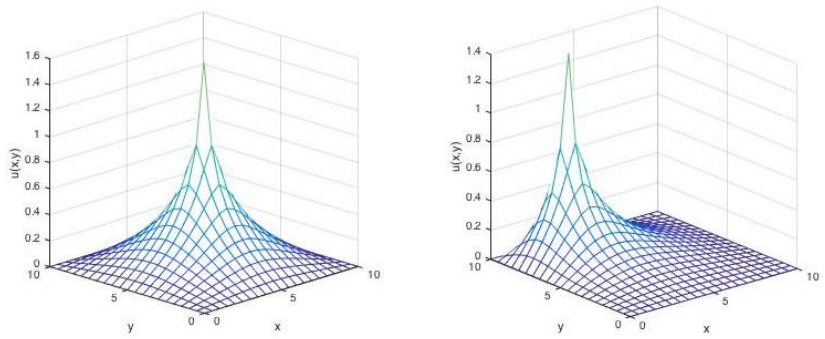
(a) Gauss-Seidel with adjusted conditions.



(b) Gauss-Seidel with adjusted conditions.

Figure 6: (a) The Gauss-Seidel method solving the Laplace equation with adjusted values. (b) The Gauss-Seidel method solving the Laplace equation with adjusted values.

The Poisson equation uses the Gauss-Seidel method for its computation. The Poisson equation is given by equation (6), this is again the Laplace equation given by equation (4). The difference is that the Poisson equation is equal to a piecewise function unlike the Laplace equation that is equal to 0. Two example of this equation is given below.



(a) Poisson equation with initial conditions.

(b) Poisson equation with adjusted grid.

Figure 7: (a) The normal Poisson equation with the correct initial conditions. (b) The poisson equation when the piecewise bounds are changed.

The Poisson equation when given the initial condition is given in Figure 7 (a). The values are $\Delta x = \Delta y = 0.1$ and the boundary conditions for x,y are 0 to 10. The piecewise function for the initial condition centers the spike at the coordinates (5,5) on the x and y plane. The math for this is given in equation (6). The coordinates can be changed to whatever the x or y input is. Thus, the new center of the spike will be whatever the inputs for the x and y, the example of this is in Figure 7 (b).

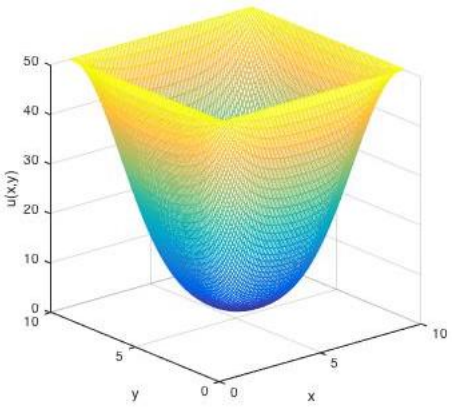


Figure 8: Poisson equation with adjusted conditions.

II.III The Heat Equation

The heat equation is given by equation (7) while equation (8) is the notation used for for the Matlab code. Every experiment in every code has lead up to this one. In the heat equation a real application for PDEs are observed. The boundaries are now set to include positions, time, and temperate. Where the x-axis includes the time, the y-axis includes the position, and the z-axis includes the temperature. The method to generate this graph is the same as all previous PDEs. An example of the heat equation with initial conditions is given below.

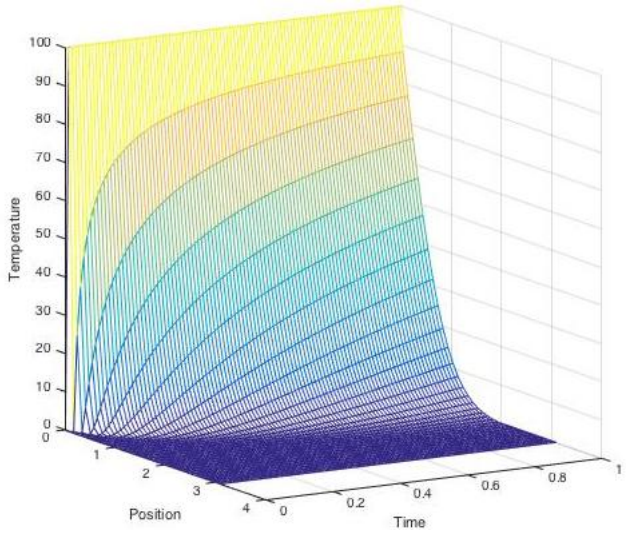


Figure 9: Heat equation with initial conditions.

In figure 9 we see that when time is zero so is position and temperature is the hottest. Due to entropy the energy (in heat) spreads out with positions and time.

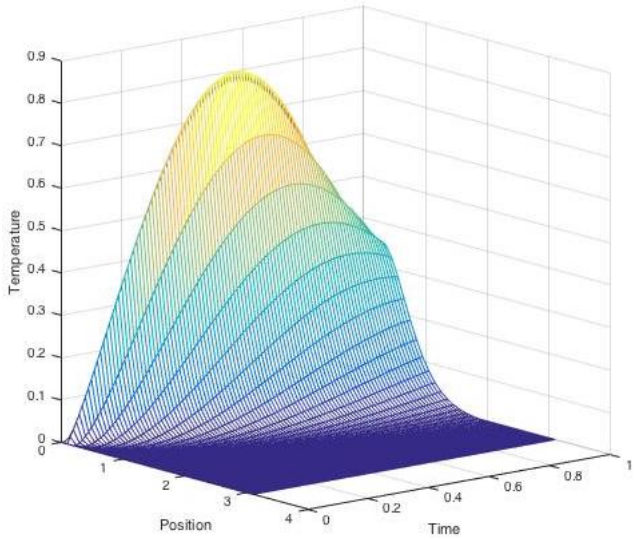


Figure 10: Heat equation with Sin function.

Figure 10 has the same concept as figure 9. However, this figure takes on a different shape because because it takes on a different function.

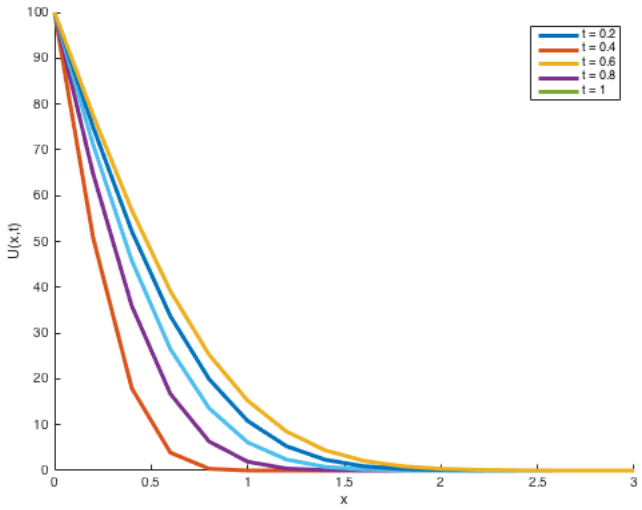


Figure 11: Heat equation table times.

III Conclusion

In this lab we were able to solve all PDEs numerically with accurate results using the implicate and explicit methods. The Laplace equation was solved in the one dimensional case as well as the two dimensional case with the use of the Jacobi Scheme. The Laplace equation was also solved for with the Gauss-Seidel method, in which case it was found the the Gauss-Seidel method is the more efficient method of the two. Changing the initial inputs of each of the codes shows how complex PDEs actually are. This also demonstrated the computational power of Matlab and the methods used when increasing the number of iterations going through the for loop. More importantly, how accurate the results come out to be, in this case, the Gauss-Seidel method had always demonstrated that it was the dominate method when running the codes. When it came to the Poisson equation the Gauss-Seidel method was used due to its computational ability. The same speculation was done with the Poisson equation as with the Laplace equation. In this case the grid size and boundaries were modified for accuracy, just like the Laplace Equation. The Poisson equation then generated a spike and could be manipulated in different ways depending on; the signs, boundaries, grid size, and position variables. The last equation to be solved for was the heat equation. Where again the initial parameters of this code were observed and changed to understand what was happening with the equation. Thus, what was observed in this lab were three different equations each with their own unique characteristics and two different methods of solving them, the implicit Jacobi Scheme and explicit Gauss-Seidel method.

IV References

References

[1] Jesus Pando and C Goedde, Physics 301 course Notes, (2017)

[2] MATLAB 2015b, The Mathworks, INC, Natick, Massachusetts, USA

Appendix

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%           Timothy Holmes
%           Jacobi
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [uu] = jacobi(A,B,dx,tol)
%A is inital
%B is final
```

```
%N is number of grids and dx is size
%10,10,0.1,0.0001
N = (1/dx);

u = zeros(N,1) ;
uu = ones(N,1) ;

u(1)=A ; uu(1)=A ;
u(N)=B ; uu(N)=B ;

sigma = tol + 1 ;
sigmaP = tol + 1 ;


while (sigma > tol) && (sigmaP > tol)

    for i = 2:(N-1)

    uu(i) =( u(i-1) + u(i+1) )/2 ;

    end

    sigma = abs( norm(uu - u) ) ;
    sigmaP = sigma./norm(uu) ;

    u = uu ;

end

plot(uu) ;
fspec = 'the values are %g \n' ;
fprintf(fspec,uu) ;

end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%         Timothy Holmes
%         Jacobi2D
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [uu] = jacobi2D(x0,y0,A,B,dx,dy,tol)
%A is inital
%B is final
%N is number of grids and dx is size
%0,0,10,10,10,10,0.001
M = dx;
N = dy;
```

```
12 u = zeros(M,N) ;
13 uu = ones(M,N) ;
14
15 u(1,:) = A; uu(1,:) = A;
16 u(:,1) = B; uu(:,1) = B;
17
18 sigma = tol + 1 ;
19 sigmaP = tol + 1 ;
20
21 tic
22 while (sigma > tol) && (sigmaP > tol)
23
24     for i = 2:(N-1)
25
26         for j = 2:(N-1)
27
28         uu(i,j) = ( u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,
               j+1))/4 ;
29
30         end
31
32     end
33
34     sigma = abs( norm(uu - u) ) ;
35     sigmaP = sigma./norm(uu) ;
36
37     u = uu ;
38
39 end
40 toc
41 hold on
42 [x,y] = meshgrid(x0:(1):x0+M-1,y0:(1):y0+N-1);
43 surf(x,y,u);
44 mesh(x,y,u);
45 xlabel('x');
46 ylabel('y');
47 zlabel('u(x,y)');
48 view(150,30)          % set viewpoint
49 camzoom(0.9)          % zoom into scene
50 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%Record
       %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
51 viobj = VideoWriter('jacobi2D.avi');
52 open(viobj);
53 for k = 1:250
54     camorbit(1,0,'data',[0 0 1])
55     drawnow
```

```
    currFrame = getframe;
    writeVideo(viobj,currFrame);
end
close(viobj);
hold off
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%          Timothy Holmes
%          Gauss Seidel Equation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [u] = gaussSeidel(u0x,ufx,x0,xf,u0y,ufy,y0,yf,dx,dy,tol)
%0,100,0,10,0,100,0,10,0.5,0.5,0.001
M = ((xf - x0)/dx) + 1;
N = ((yf - y0)/dy) + 1;
u = zeros(M,N) ;

u(1,:) = u0y; u(N,:) = ufy;
u(:,1) = u0x; u(:,M) = ufx;

sigma = tol + 1 ;
%%%%%%%%%%%%%%%%%%%%%%%%Iterations%%%%%%%%%%%%%%%%%%%%%%%%
tic
while (sigma > tol)

    for i = 2:(N-1)

        for j = 2:(N-1)

            uu = u(i,j);
            u(i,j) = ( u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1))/4 ;

        end

    end

    sigma = abs((u(i,j)-uu)/((u(i,j)+uu)/2));

end
toc
%%%%%%%%%%%%%%%%%%%%%%%%graph%%%%%%%%%%%%%%%%%%%%%%%%%
hold on
```

```
figure(1)
[x,y] = meshgrid(x0:dx:xf,y0:dy:yf);
surf(x,y,u);
mesh(x,y,u);
xlabel('x');
ylabel('y');
zlabel('u(x,y)');
%%%%%%%%%%%%%%%%%%%%%%%%%View
    %%%%%%%%%%%%%%%%%%%%%%%%%
view(-40,30)
camzoom(0.9)
%%%%%%%%%%%%%%%%%%%%%%%%%Record
    %%%%%%%%%%%%%%%%%%%%%%%%%
viobj = VideoWriter('gaussSiedel.avi');
open(viobj);
for k = 1:500
    camorbit(1,0,'data',[0 0 1])
    drawnow
    currFrame = getframe;
    writeVideo(viobj,currFrame);
end
close(viobj);
hold off
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%         Timothy Holmes
%         Poisson Equation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [u] = poisson(u0x,ufx,x0,xf,u0y,ufy,y0,yf,dx,dy,
    tol)
%A is inital
%B is final
%N is number of grids and dx is size
%0,0,0,10,0,0,0,10,0.5,0.5,0.001
M = ((xf - x0)/dx) + 1;
N = ((yf - y0)/dy) + 1;
u = zeros(M,N) ;

u(1,:) = u0y; u(N,:) = ufy;
u(:,1) = u0x; u(:,M) = ufx;

sigma = tol + 1 ;

tic
%%%%%%%%%%%%%%%%%%%%%%%%%Iterations
```

```
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
while (sigma > tol)

    for i = 2:(N-1)

        for j = 2:(N-1)
            uu=u(i,j);

            if(j*dx-dx == 5 && i*dy-dy == 5)

                u(i,j)=(u(i+1,j)+u(i-1,j)+u(i,j+1)+u(i,j
                    -1)+10*dx^2)/4;

            else

                u(i,j)=(u(i+1,j)+u(i-1,j)+u(i,j+1)+u(i,j
                    -1))/4;

            end

                sigma = abs((u(i,j)-uu)/((u(i,j)+uu)/2));

        end

    end

end
toc
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%graph
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
[x,y] = meshgrid(x0:dx:xf,y0:dy:yf);

surf(x,y,u);
mesh(x,y,u);
xlabel('x');
ylabel('y');
zlabel('u(x,y)');
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%View
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
camzoom(0.9)
view(-40,20)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%Record
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
viobj = VideoWriter('poisson.avi');
open(viobj);
for k = 1:500
```

```
    camorbit(1,0,'data',[0 0 1])
    drawnow
    currFrame = getframe;
    writeVideo(viobj,currFrame);
end
close(viobj);
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%          Timothy Holmes
%          Heat Equation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function heatEquation(x0,xf,t0,tf,h,k,a)
%0, pi, 0, 1, 0.1,0.01,0.5
xl = int16(((xf - x0)/(h))+1);
tl = int16(((tf - t0)/(k))+1);
u = zeros(xl,tl);

u(1,:) = 100;
u(:,1) = 0;

s = ((a^2)*k)/(h^2);%Constants

%%%%%%%%%%%%%%%%%%%%%%%%%%%%Iterations %%%%%%%%%%%%%%%%%%%%%%%%%%%
for j = 2:1:(tl - 1)
    for i = 2:1:(xl - 1)
        u(i,j) = ((1-2*s)*u(i,j-1)) + (s*(u(i+1,j-1)+u(i-1,j-1)));
    end
end

u = u';
%%%%%%%%%%%%%%%%%%%%%%%%%%%%graph %%%%%%%%%%%%%%%%%%%%%%%%%%%%
hold on
figure(1)
[x,y] = meshgrid(x0:h:xf,t0:k:tf);
surf(x,y,u);
mesh(x,y,u);
xlabel('Position');
ylabel('Time');
zlabel('Temperature');
%%%%%%%%%%%%%%%%%%%%%%%%%%%%View %%%%%%%%%%%%%%%%%%%%%%%%%%%%
view(300,20)
```

```
35 camzoom(1.2)
36 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%Record
       %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
37 viobj = VideoWriter('heatEquation.avi');
38 open(viobj);
39 for k = 1:250
40     camorbit(1,0,'data',[0 0 1])
41     drawnow
42     currFrame = getframe;
43     writeVideo(viobj,currFrame);
44 end
45 close(viobj);
46 hold off
47 end
```