A Brief Primer on the Numerical Solution of Differential Equations with Matlab

(accompanied by sample Python code)

James Blanchard
blanchard@engr.wisc.edu
University of Wisconsin – Madison

December 2023

Contents

Introduction

In this document, I give brief discussions of the most common numerical methods used to solve ordinary differential equations (both initial value and boundary value), parabolic partial differential equations, and elliptic partial differential equations. Most of the discussion centers around Matlab solutions, including some built-in solvers, but in a few cases examples are also provided in Python. For every topic, there is a discussion of how to validate your solutions. I consider these critical, as we are all susceptible to an undeserved trust of our initial attempts at numerical solutions to problems. It takes discipline and creativity to properly test our simulations and I hope to foster both with these examples.

Ordinary Differential Equations: Initial Value Problems

First Order Initial Value Problems

A general, first-order ordinary differential equation (ODE) can be written as:

$$\frac{dy}{dt} = f(t, y)$$

Here the function *f* can be any arbitrary function of *t* and *y*. If *f* is a linear function of *y*, then this differential equation is linear. If *f* is nonlinear in *t*, the differential equation is still linear. A general solution to this equation will always involve an independent, arbitrary constant which is typically determined based on an initial condition. As long as *f* is continuous, then linear equations will always have a unique solution. The situation for nonlinear equations is much less clear.

Numerical solutions to initial value problems typically involve starting at the initial time (typically t=0) and progressing in time using discrete time steps (with *h* defined as the time between steps). The process involves approximating the derivative in terms of these discrete values for *y* and then using this approximation to advance the solution one step at a time. The easiest algorithm for this approach is Euler's method, which assumes that the slope of the solution during a time step depends only on variables at the beginning of the step. [Note that Euler's method has flaws and, for most problems there are better algorithms. However, it does help me illustrate the basic process used to numerically solve ordinary differential equations.] Euler's method leads to a solution algorithm that can be written as

$$\frac{dy}{dt} \approx \frac{y_{i+1} - y_i}{t_{i+1} - t_i} = \frac{y_{i+1} - y_i}{h} = f(t_i, y_i)$$

The key here is that we evaluate the derivative using only the values of t and y at the beginning of the time step. This simplifies to

$$y_{i+1} \approx y_i + h\, f(t_i, y_i)$$

To demonstrate this algorithm, consider the following equation:

$$\frac{dy}{dt} - 2ty = 0$$

$$y(0) = 1$$

We know that the solution to this equation is

$$y = e^{t^2}$$

Which will allow us to compare results from our numerical algorithm to this known solution.

To carry out Euler's method to approximate the solution to the differential equation above, we start at *t=0* with our known initial value: *y(0)=1* and then carry out a solution step-by-step.

For our model equation, the Euler algorithm simplifies to

$$y_{i+1} \approx y_i + h\, f(t_i, y_i) = y_i + h * 2 * t_i * y_i$$

Or

$$y_{i+1} \approx y_i * (1 + 2ht_i) = y_i * (1 + 2h(ih)) = y_i * (1 + 2ih^2)$$

This can be demonstrated in the first few time steps as

y_0=y(0)=1

y_1=y_0*(1+2ht_0)= y_0

y_2=y_1*(1+2ht_1) =y_1*(1+2h^2)

…

By continuing this step-by-step process, we can generate an approximate solution for as long as we please. I will present a graphical solution for this equation, using Euler's method, after taking a look at a second algorithm.

The 4th order Runge-Kutta algorithm is probably the most commonly used algorithm for a typical initial value problem. The algorithm uses a stepwise approach, much like the Euler method, but the time stepping algorithm is somewhat more complicated. Specifically, the algorithm is given as:

$$k_1 = f(t_i, y_i)$$

$$k_2 = f\left(t_i + \frac{h}{2}, y_i + h\frac{k_1}{2}\right)$$

$$k_3 = f\left(t_i + \frac{h}{2}, y_i + h\frac{k_2}{2}\right)$$

$$k_4 = f(t_i + h, y_i + hk_3)$$

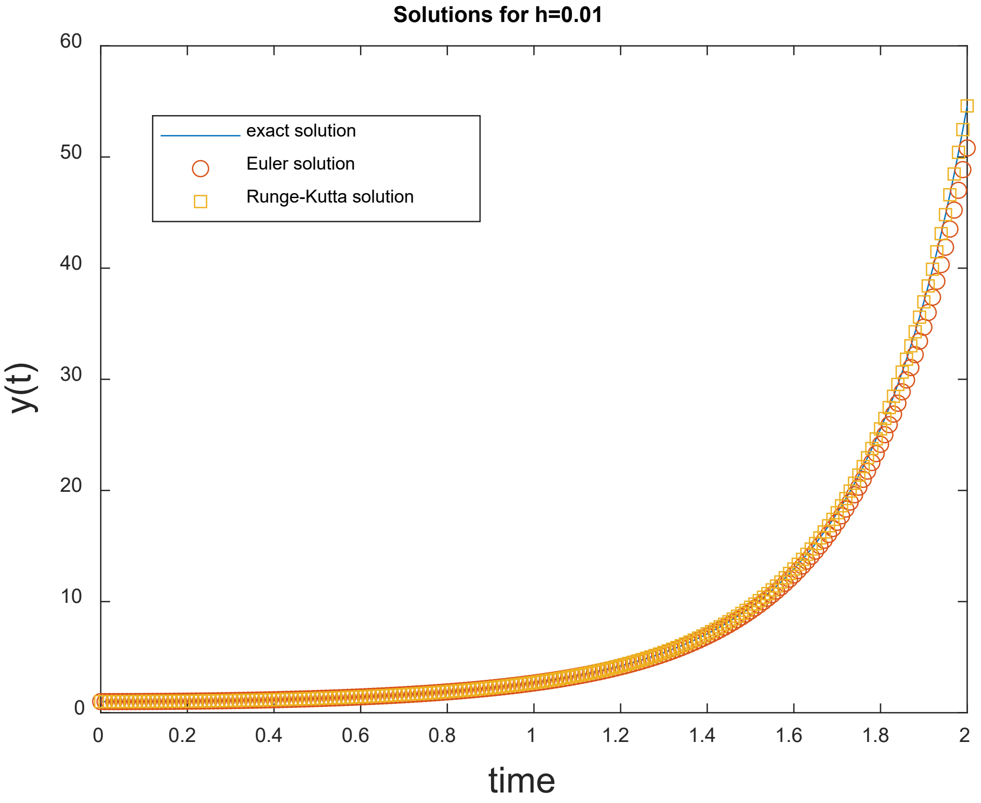
$$y_{i+1} = y_i + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

Though somewhat more complicated than the Euler method, this algorithm still does not require any iteration or solutions of algebraic systems, so it is quite efficient. It also can easily solve quite complex equations. Hence, it is a workhorse.

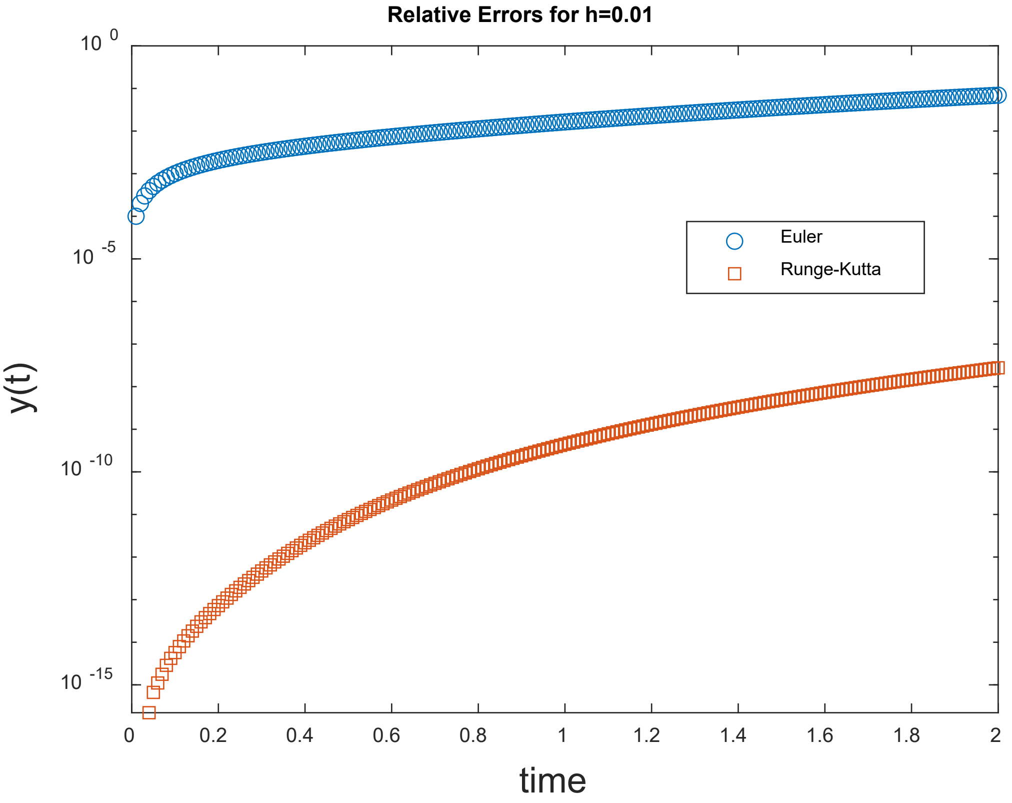
A Matlab implementation of both Euler's method and the 4th order Runge Kutta method for our model problem is provided below.

```
clear
h=0.01; %step size
tend=2; %time at end of simulation
time=0:h:tend;
yinitial=1; %initial value [y(0)]
yeuler(1)=yinitial;
yrk(1)=yinitial;
for i=2:size(time,2)
    yeuler(i)=yeuler(i-1)+h*f(time(i-1),yeuler(i-1));
    yrk(i)=yrk(i-1)+rkstep(time(i-1),yrk(i-1),h);
end
yexact=exp(time.^2);

plot(time,yexact,time,yeuler, 'o',time,yrk,'+')
titlestring=sprintf('Comparison of Numerical Solutions to Exact
Solution for h=%2.1d',h);
xlabel('time','FontSize', 18);ylabel('y(t)','FontSize',
18);title(titlestring)
legend('exact solution','Euler solution','Runge-Kutta solution')

eulererror=abs((yeuler-yexact)./yexact);
rkerror=abs((yrk-yexact)./yexact);

figure, semilogy(time,eulererror, 'o',time,rkerror,'+')
titlestring=sprintf('Comparison of Relative Errors for
h=%2.1d',h);
xlabel('time','FontSize', 18);
ylabel('y(t)','FontSize', 18);
title(titlestring)
legend('Euler','Runge-Kutta')

function z=rkstep(t,y,h)
k1=f(t,y);
k2=f(t+h/2,y+h*k1/2);
k3=f(t+h/2,y+h*k2/2);
k4=f(t+h,y+h*k3);
z=h/6*(k1+2*k2+2*k3+k4);
end

function z=f(t,y)
z=2*y*t;
end
```

The resulting solutions, along with the exact solution, are provided below for a time step of *h=0.01*:

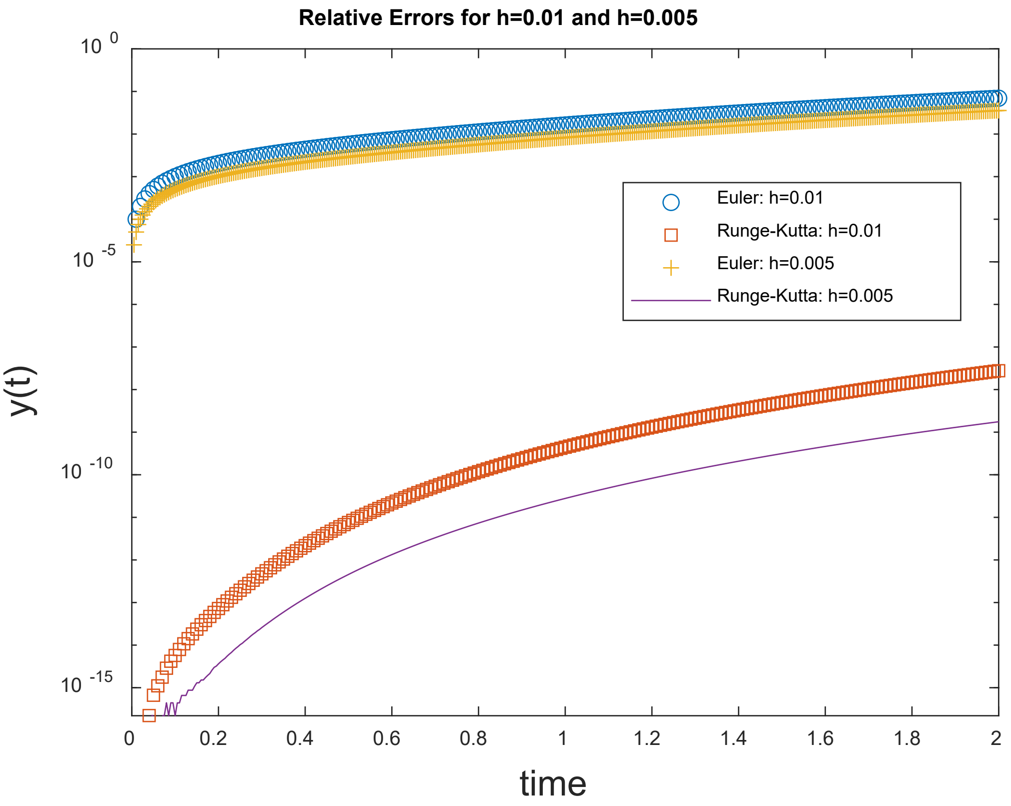


It appears from this plot that both solutions are fairly close to the exact solution, but we can see the accuracy better if we plot the relative error for each:



From this error plot, it is obvious that the Euler error is fairly large (around 10% at the end), while the error for the Runge-Kutta solution is orders of magnitude more accurate (better than 10^{-7} at the end). This demonstrates the value of the Runge-Kutta algorithm.

Now let's halve the time step and see the effect.



These results show that our solutions improve as the time step gets smaller. In the case of Runge-Kutta, halving the time step improved the relative error by a factor of 16. This is consistent with the theoretical scaling of h^4 for the global error[1].

Built-In Function: ode45

Matlab has several built-in functions for solving ordinary differential equations. The most commonly used function is probably ode45, which employs both 4th and 5th order Runge-Kutta algorithms to allow for time step adjustment in order to achieve a desired accuracy. That is, it adjusts the time step as it goes, reducing it when the error exceeds the desired accuracy and increasing it when the error is below the desired accuracy. Hence, we do not have to specify the time step, we merely provide a function defining the differential equation, the desired time interval over which to solve the equation, and the initial value. A solution for our model problem is as follows:

```
clear
tspan=[0 2]; %time interval for solution
yinitial=1; %initial value
[t,y]=ode45(@f, tspan, yinitial);
plot(t,y)
xlabel('time'); ylabel('y');
```

[1] https://lpsa.swarthmore.edu/NumInt/NumIntFourth.html

```
function z=f(t,y)
z=2*y*t;
end
```

This solution uses the default accuracy, which is set to a relative accuracy of 0.001. The code below sets values for both the relative tolerance and the absolute tolerance.

```
opts = odeset('RelTol',1e-6,'AbsTol',1e-6);
[t,y] = ode45(@f, tspan, yinitial, opts);
```

If you want to set a parameter in the function, we can set it in the main code and then pass it into the function as follows:

```
clear
tspan=[0 2]; %time interval for solution
yinitial=1; %initial value

aparam=2; %set a parameter to be used in the function fpar
[t,y] = ode45(@(t,y) fpar(t,y,aparam), tspan, yinitial);

function z=fpar(t,y,aparam)
z=aparam*y*t;
end
```

New Infrastructure in Recent Versions of Matlab

Recent versions of Matlab add a new infrastructure for solving ordinary differential equations. A typical script for solving our model equation is:

```
F = ode;
F.ODEFcn = @(t,y) 2*y*t;
F.InitialValue = 1;
sol = solve(F,0,10);
plot(sol.Time,sol.Solution,"-o")
```

Here F is an "ode" object, F.ODEFCN defines our differential equation, F.InitialValue is y(0) and sol becomes the solution. Calling the solve function with solve(F,0,10) solves the ode object F from t=0 through t=10.

Python Solution

To solve our model equation in Python, use:

```
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
def f(t, y): return 2 * t * y
```

```
sol = solve_ivp(f, [0, 2], [1], rtol = 1e-6)
print(sol.t)
print(sol.y)
plt.plot(sol.t, sol.y.T)
plt.xlabel('time')
plt.ylabel('y(t)')
plt.show()
```

Systems of Initial Value Problems

To solve a system of ordinary differential equations, we can still use the same routines, but our solutions are stored in an array and we must supply a vector of initial values as well as a vector function defining the differential equations. Consider the following system:

$$\frac{dx}{dt} = x - \alpha xy$$

$$\frac{dy}{dt} = -y + \beta xy$$

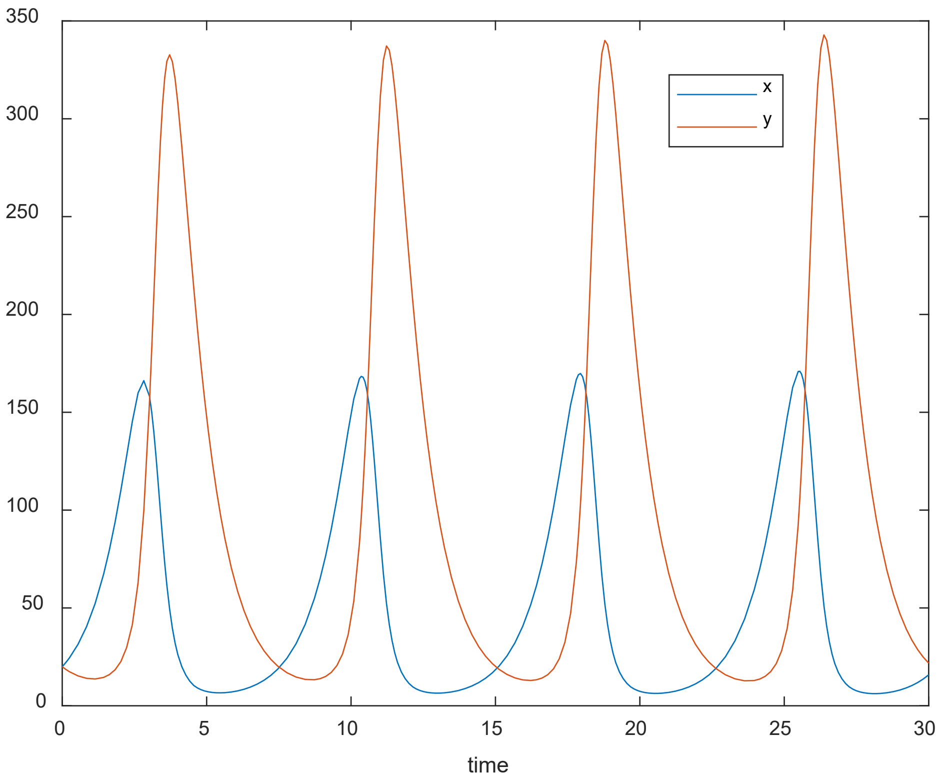
Note that this is an example of a predator-prey problem. Consider x as the population of the prey and y the population of the predator. So the positive x term in the first equation indicates that the prey population would grow exponentially if there were no predators and the negative y term in the second equation indicates that the predators would die off if there were no prey. The cross terms, involving the product xy, indicate that the growth rate of the prey is lower when there are predators around and that the growth rate of the predators increases when there is a lot of prey.

To solve this system in Matlab, you can do the following:

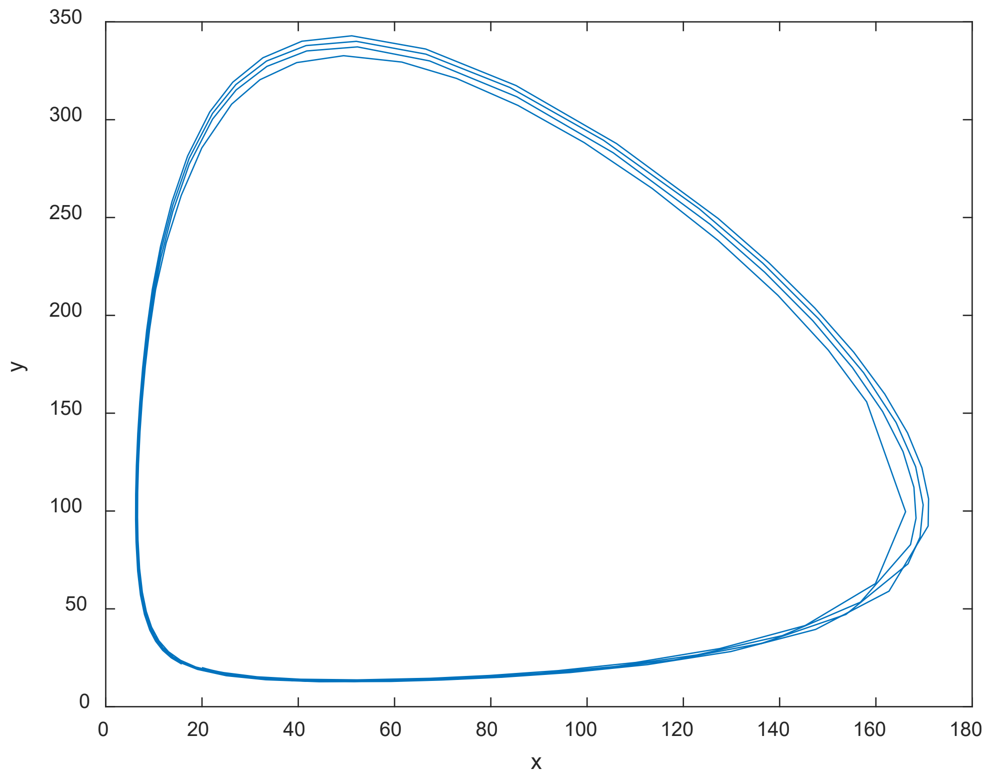
```
F = ode;
F.ODEFcn = @odefunc;
F.InitialValue = [20; 20];
sol = solve(F,0,30);
plot(sol.Time,sol.Solution)
xlabel('time');
legend('x','y')

function growthrates=odefunc(t,z)
x=z(1);
y=z(2);
alpha=0.01; beta=0.02;
growthrates=[x-alpha*x*y; -y+beta*x*y];
end
```

Here I’ve made the arbitrary decision to have x be variable 1 and y be variable 2. Hence, the first entry in the InitialValue vector is the initial value for x and the second entry is the initial value for y. Similarly, the first term in “growthrates” is dx/dt and the second term is dy/dt. You could switch these, and have y be first, but everything has to be changed consistently.

The solution provided by this code can be visualized as:



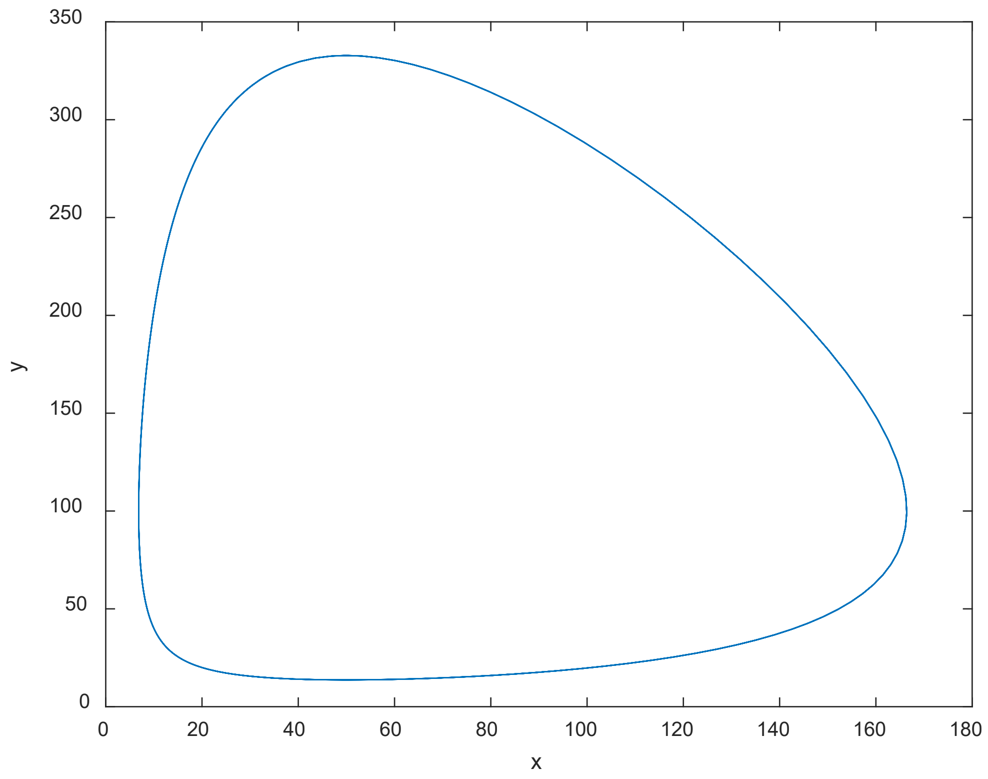
Notice how the prey first increases, as the predator population is low, but then the predator population rises as the food supply rises. This then causes a decline in the prey population, creating cyclic behavior. This can also be visualized as a phase portrait, where we plot x vs y:



This should just be one curve, but the error in the solution prevents this. We can do better by asking for a tighter tolerance. For example, the following code

```
F = ode;
F.ODEFcn = @odefunc;
F.InitialValue = [20; 20];
F.RelativeTolerance=1e-6
sol = solve(F,0,30);
plot(sol.Time,sol.Solution)
xlabel('time');
legend('x','y')
figure,plot(sol.Solution(1,:), sol.Solution(2,:))
xlabel('x'); ylabel('y');

function growthrates=odefunc(t,z)
x=z(1);
y=z(2);
alpha=0.01; beta=0.02;
growthrates=[x-alpha*x*y; -y+beta*x*y];
end
```

will produce a phase portrait with a single curve.



We can use the same approach to solve a larger system, such as the Lorenz Equations, which solve three equations:

$$\frac{dx}{dt} = \sigma(y - x)$$

$$\frac{dy}{dt} = x(\rho - z) - y$$

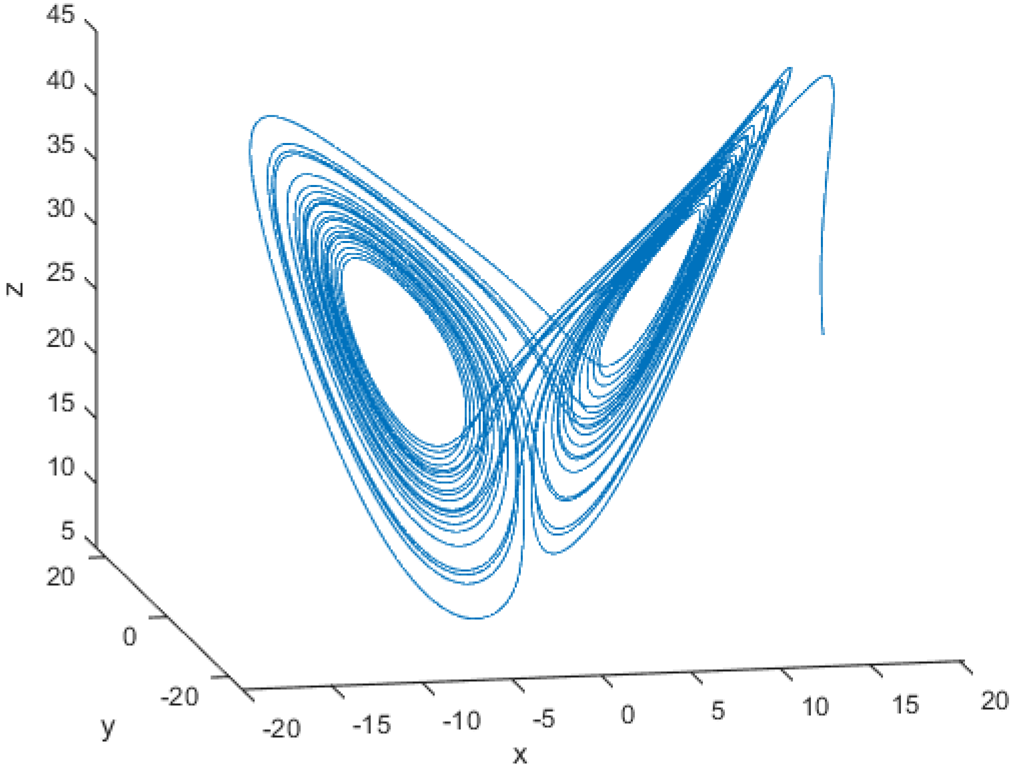
$$\frac{dz}{dt} = xy - \beta z$$

The following code will solve the equations

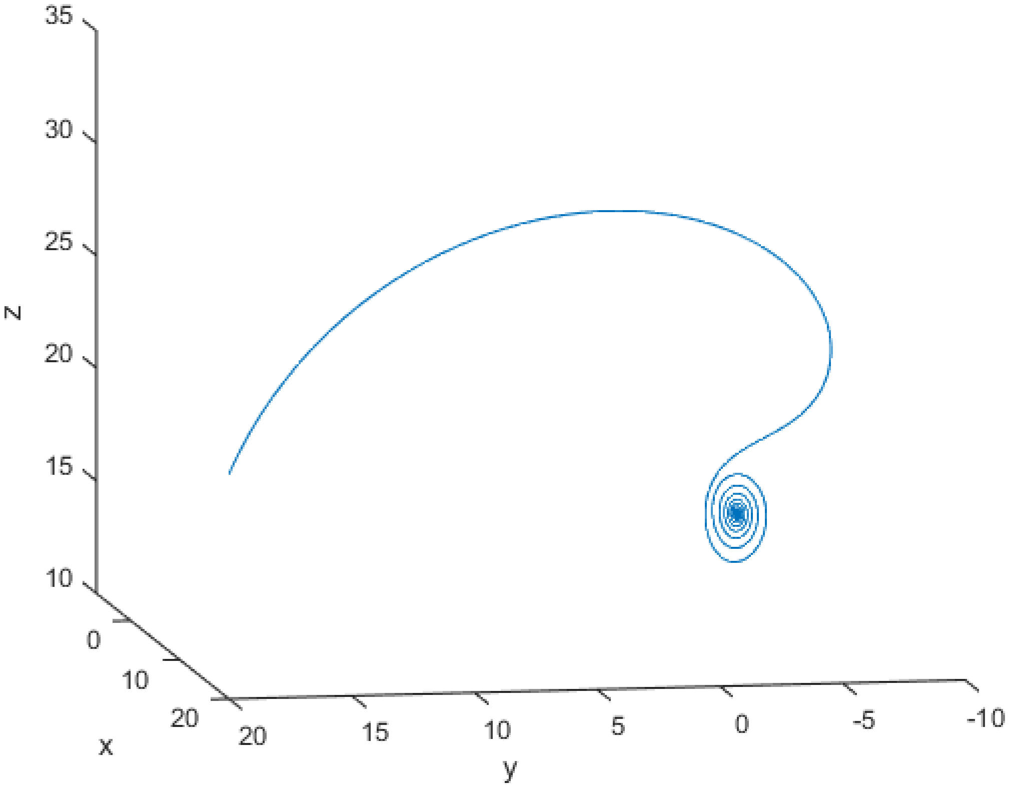
```
F = ode;
F.ODEFcn = @odefunc;
F.InitialValue = [20; 20; 20];
F.RelativeTolerance=1e-6
sol = solve(F,0,30);
figure,plot3(sol.Solution(1,:), sol.Solution(2,:),
sol.Solution(3,:))
xlabel('x'); ylabel('y'); zlabel('z');
```

```
function growthrates=odefunc(t,res)
x=res(1);
y=res(2);
z=res(3);
sigma=10; beta=8/3; rho=28;
growthrates=[sigma*(y-x); x*(rho-z)-y; x*y-beta*z];
end
```

and the resulting phase portrait is now three-dimensional and becomes



Note that this system demonstrates chaos. That is, it has two equilibrium points, represented by the two neighboring spirals in the phase portrait, but the solution is unable to spiral in to either equilibrium. It continuously bounces from one to the other. If we reduce ρ to 14, the system is no longer chaotic and the solution settles in on one of the equilibrium points, depending on the initial conditions.



35
30
25
z
20
15
10
0
10
20
x
20
15
10
5
0
-5
-10
y

Higher Order Initial Value Problems

We can use these same tools to solve higher order problems. The typical approach is to break the equation into a system of first order equations and then solve the system as we did in the lasts section. Consider the equation:

$$\frac{d^2y}{dt^2} + 2y = 0$$

$$y(0) = 0$$

$$\frac{dy}{dt}(0) = 1$$

To break this into a system of two first order equations, we begin by defining a second variable, *z*, such that

$$\frac{dy}{dt} = z$$

Given this definition, the second derivative of y will be equal to the first derivative of z. Substituting this into our equation we find

$$\frac{d^2y}{dt^2} = \frac{dz}{dt} = -2y$$

Our second order equation thus becomes:

$$\frac{dy}{dt} = z$$

$$\frac{dz}{dt} = -2y$$

$$y(0) = 0$$

$$z(0) = 1$$

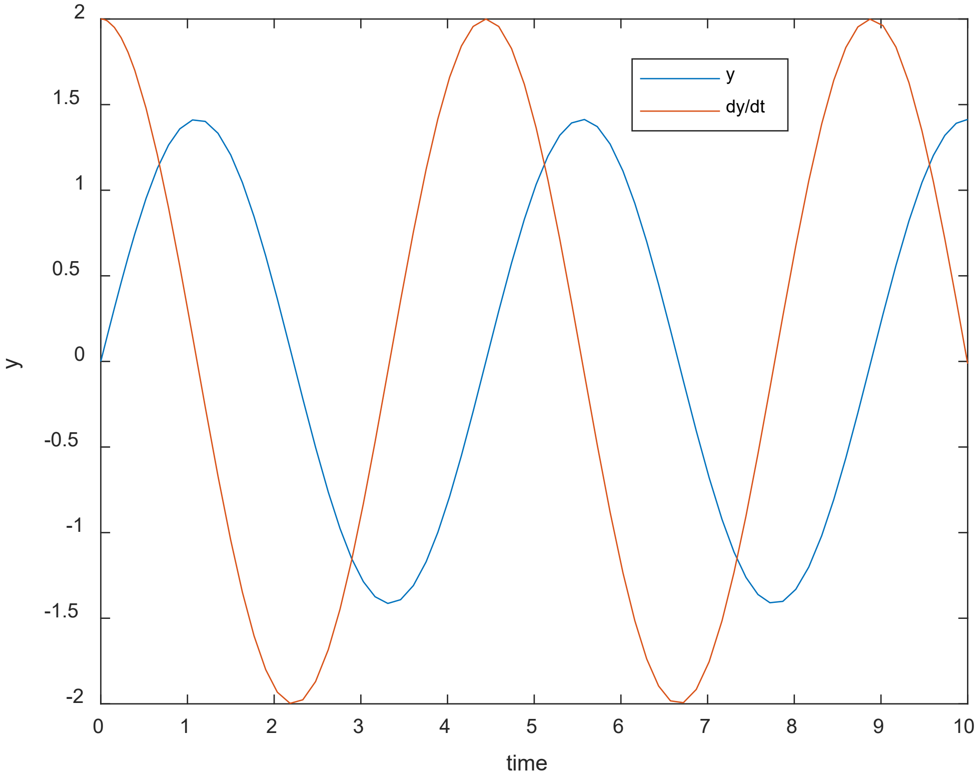
A Matlab solution for this system, again using ode45, can be written as follows:

```
clear
tspan=[0 10]; %time interval for solution
yinitial=[0; 2]; %initial values for [y z]

[t,y] = ode45(@f, tspan, yinitial);
yanswer=y(:,1);
zanswer=y(:,2);

plot(t,yanswer,t,zanswer)
xlabel('time'); ylabel('y');
legend('y', 'dy/dt');

function freturn=f(t,y)
ysolution=y(1);
```

```
zsolution=y(2);
freturn=[zsolution; -2*ysolution]; %This gives [dy/dt dz/dt]
end
```

Note that the result is an array with one column for the solution for *y* and the other for the solution for *z*. I have arbitrarily chosen to have the first column represent *y* and the second represent *z*.

A plot of the solutions is:



Python Solution for Second Order Equation

Python code to solve this problem is:

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

def model(r,t):
    y=r[0]
    z=r[1]
    dydt=z
    dzdt=-2*y
    drdt=[dydt, dzdt]
    return drdt
```

```
r0=[0,2]
t = np.linspace(0,10,200)
r = odeint(model,r0,t)
plt.plot(t,r[:,0])
plt.xlabel('time')
plt.ylabel('y(t)')
plt.show()
```

Validation of Initial Value Problems

To ensure that we obtain accurate numerical solutions to these problems, it is critical that we always maintain an appropriate skepticism about the results. There are many ways for error to be introduced to a numerical analysis, so constant vigilance is required. Some of the most common error sources are as follows:

- Problems with units
- Coding mistakes
- Errors in the model itself
- Errors in the input data, such as material properties (garbage in, garbage out)
- Incorrect use of a built-in algorithm (such as `ode45`)
- Insufficient choice of numerical parameters (such as the time step in initial value problems, or using too few terms in a series approximation)

To combat these possible errors, the best approach is to have multiple strategies for ensuring the validity of your solutions. Some useful approaches include:

- When you are new to a technique, test it on problems where the solution is known. Start by picking some easy problems for which the solution is known and make sure you can match the known results with your numerical solution.
- The most useful approach is something I think of as a global approach. That is, you should have some idea of the right answer prior to attempting the numerical solution.
- Always make sure your solution satisfies the initial conditions. This is generally easily checked with a plot of the solution.
- Convergence study: Change the tolerance (or shrink the time step) and make sure the results don't change
- Benchmark against known solutions
- Approximate full problem with something we can solve analytically
- Test experimentally

Let's consider these with an example. Consider the cooling of a cup of coffee. Let's try to calculate the temperature of a cup of coffee that was originally at a temperature of 90 C (around 195F). How long will it take to cool to 60 C? If we assume that the entire volume of coffee has a uniform temperature (an assumption we will test when we discuss parabolic partial differential equations), then this can be modeled as an ordinary differential equation. We will assume all the heat loss is from the liquid surface at the top of the mug (which is a pretty good assumption if we have a good quality mug with good thermal insulation).

Assumptions:

- The heat transfer from the liquid to and from the mug itself is negligible
- The liquid has uniform temperature at all times. That is, there is no temperature gradient in the liquid.
- There is no air blowing over the liquid surface. This would accelerate the convective cooling, changing it from free convection to forced convection. It also would accelerate the mass transfer.
- We will use the properties of water to model coffee.
- The liquid surface is circular
- The liquid properties are independent of temperature

Heat is lost from our cup via radiation, mass transfer (evaporation), and convection from the liquid surface (which has a surface area which we will call *A*)[2]. The governing equation for the temperature becomes:

$$mc_p \frac{dT}{dt} = -\epsilon\sigma(T - T_a)^4 - hA(T - T_a) - W\Delta H_v$$

Where *m* is the mass of the liquid in the cup, c_p is the specific heat of the liquid, *T* is its temperature, ε is the emissivity of the surface, σ is the Stefan-Boltzman constant ($5.67\text{x}10^{-8}$ W/m^2-K^4), T_a is the air temperature, *h* is the heat transfer coefficient for free convection from the surface, W is the evaporative mass flux at the surface, and ΔH_v is the heat of vaporization of the liquid per kg of liquid.

The governing equation for the mass left in the mug (accounting for evaporation) is

$$\frac{dm}{dt} = -W$$

The free convection heat transfer coefficient can be approximated by

$$h = 1.31\left(\frac{T - T_a}{D}\right)^{1/4}$$

where D is the diameter of the liquid surface (assumed to be circular).

Since we are losing liquid due to evaporation, the liquid mass will be decreasing over time and we need a model for the rate at which this mass is lost. We will use

$$W = \frac{h\aleph A}{\aleph_{air} c_p^{air} F}[P_v(T) - P_v(T_a)]$$

where א represents the molecular weight, P_v is the vapor pressure, and

[2] Jean Stephane Condoret, "Teaching Transport Phenomena Around a Cup of Coffee," https://oatao.univ-toulouse.fr/1455/

$$F = \frac{\left(P_T - 0.5P_v(T_a)\right) - \left(P_T - P_v(T_a)\right)}{ln\left[\frac{P_T - 0.5P_v(T_a)}{P_T - P_v(T_a)}\right]}$$

where P_T is the total pressure at the surface of the liquid.

For the vapor pressure, we will use

$$\mathrm{P_v} = 10^5 exp\left[\frac{-40700}{8.31}\left(\frac{1}{T} - \frac{1}{373}\right)\right]$$

where the pressure is in Pa and the temperature is in C.

The time rate of change of the mass of the liquid in the mug comes from the evaporation model. Hence,

$$W = -\frac{h\aleph A}{\aleph_{air} c_p^{air} F}\left[P_v(T) - P_v(T_a)\right]$$

Code for the numerical solution of this problem is provided below. Here I assumed the following for inputs:

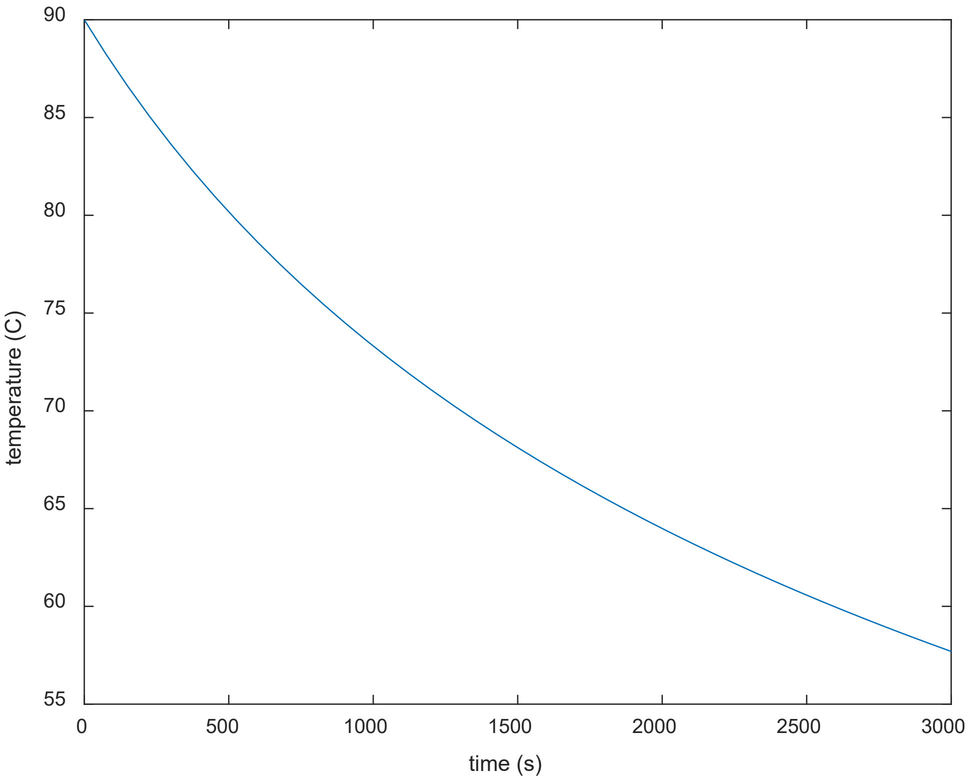
- Liquid surface diameter = 5 cm
- Initial liquid mass = 0.3 kg
- Emissivity of liquid = 0.9
- Air temperature = 20 C
- Molecular weight for water = 18 g/mol
- Molecular weight for air = 29 g/mol
- Ambient air pressure = 0^5 Pa
- Specific heat of liquid = 4,000 J/kg-K
- Specific heat of air = 1,000 J/kg-K
- Heat of vaporization of liquid is $4.2\text{x}10^6$ J/kg

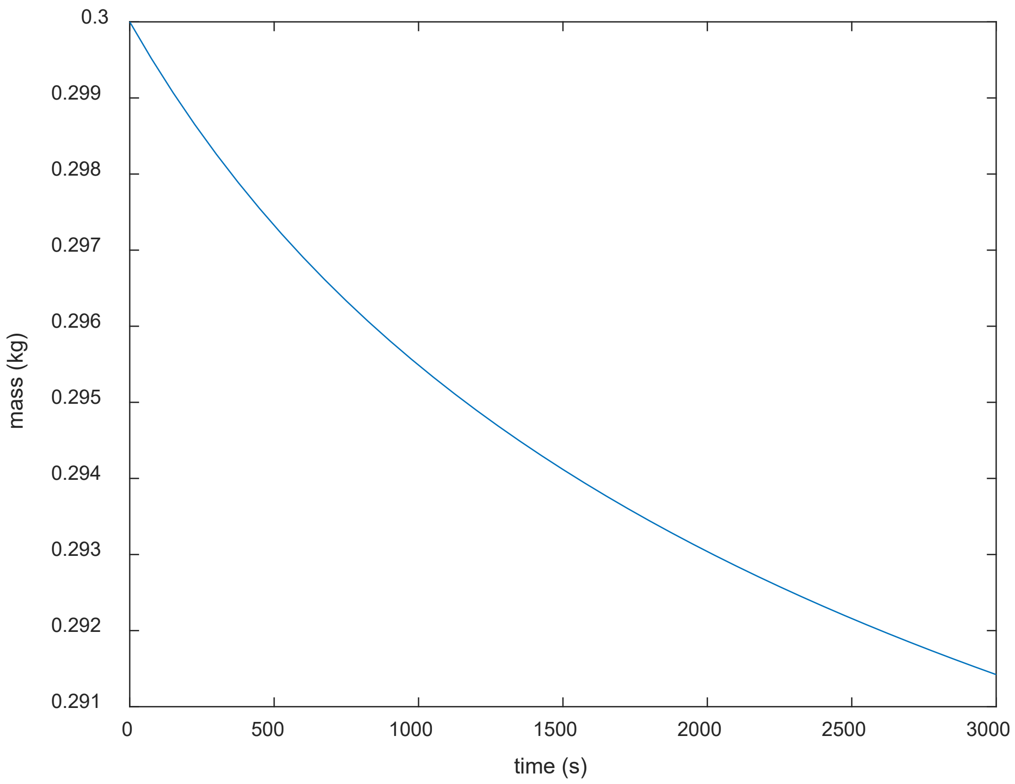
The code for this simulation is as follows:

```
clear variables
F = ode;
F.ODEFcn = @odefunc;
F.InitialValue = [90+273; 0.3];
F.RelativeTolerance=1e-6;
sol = solve(F,0,3000);
figure,plot(sol.Time,sol.Solution(1,:)-273)
xlabel('time (s)'); ylabel('temperature (C)');

figure,plot(sol.Time, sol.Solution(2,:))
xlabel('time (s)'); ylabel('mass (kg)');

function eqnfuncs=odefunc(t,res)
T=res(1);
mass=res(2);
```

```
D=0.05; A=pi*D^2/4;
emiss=0.9; stef=5.67e-8;
Ta=20+273;
cpliquid=4000; cpair=1000;
deltaH=4.2e6;
MWair=29; MWwater=18;
pt=1e5; %Pa

h=1.31*((T-Ta)/D)^(1/4);

c1=pt-0.5*pv(Ta);
c2=pt-pv(Ta);
F=(c1-c2)/log(c1/c2);
W=h*MWwater/MWair/cpair/F*A*(pv(T)-pv(Ta));

eqnfuncs=[(-emiss*stef*A*(T-Ta).^4-h*A*(T-Ta)-
W*deltaH)/mass/cpliquid; -W];
end

function p=pv(T)
p=1e5*exp(-40700/8.31*(1/T-1/373));
end
```

The temperature and mass plots from this simulation are provided below:





So, to answer our original question, this simulation indicates that it takes about 2,600 seconds for our coffee to cool to 60 C. Note that the simulation predicts that about 3% of our coffee evaporated in 3,000 seconds.

Now let's take some time to check our solution, using the strategies described above.

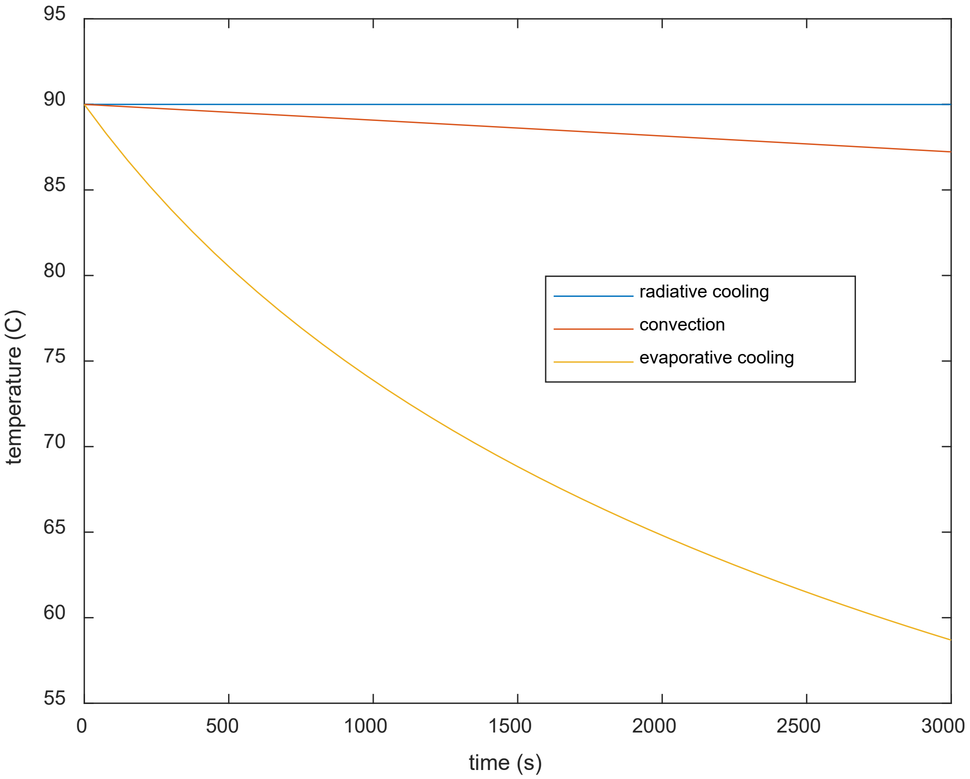
First, let's use some common sense. We all have some experience with hot liquids in a ceramic mug. My guess was that it would take about 30 minutes for this cooling to occur. Since the simulation predicted 45 minutes, I'd say we probably haven't made any egregious errors.

Next, we are using many correlations here and we could easily have made an error in implementing them. It is also possible that our source from which we retrieved the correlations had made an error. Hence, here is a table of some of the relevant parameters at the end of our simulation:

Parameter	Value	Units
Vapor pressure	2773	Pa
Heat transfer coefficient	6.86	W/m^2-K
Logarithmic mean of partial pressure of air	97,900	Pa

Based on my experience with these parameters, these values seem about right, so that is good indication that we haven't made errors in the correlations.

Next, we can run separate simulations for evaporative cooling, radiative cooling, and free convection.



We see that all of the cooling mechanisms, working alone, do reduce the coffee temperature and the evaporative cooling dominates the other two. This is all consistent with expectations.

Next, we can check our analysis parameters. In this case, this means varying the tolerance for the adaptive Runge-Kutta algorithm. The code above sets the Relative Tolerance to a value of 10^{-6}. If we reduce the Tolerance by a factor of 100, the temperature at the end of the simulation changes by 2 parts in 10^{8}. That's definitely a variation we can live with, given the uncertainty in our assumptions and correlations, indicating that our initial setting was adequate.

We can simplify our original differential equation to allow for an analytical solution. This will then allow determination of the error in our simulation results, at least for the simplified problem. For example, we can solve the problem exactly for the case of convection if we assume a constant heat transfer coefficient of 7 MW/m^{2}.

$$mc_p \frac{dT}{dt} = -hA(T - T_a)$$

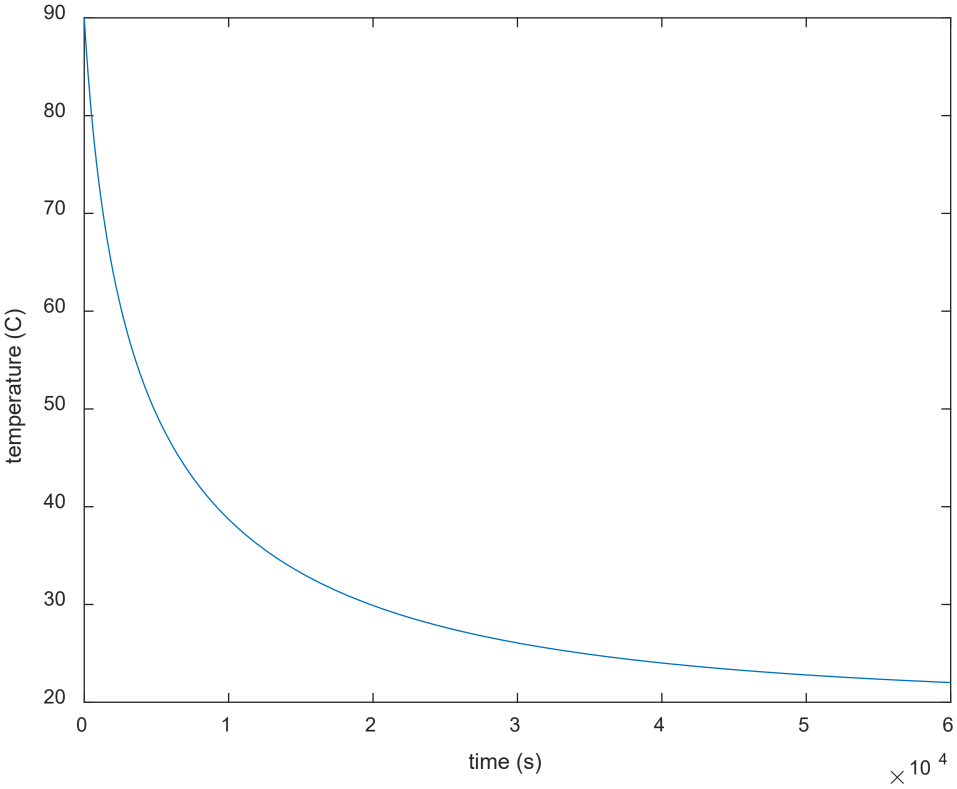
Unlike before, the mass will be constant in this case. The solution is:

$$T = T_a + (T_0 - T_a)\exp{(-zt)}$$

$$z = \frac{hA}{mc_p}$$

When I run the code above with no evaporation and no radiation, my results agree with this analytical solution with a relative error of 3x10^{-4} at 1,000 seconds.

It often is also helpful to check the steady state solutions in problems such as this. In our case, the steady state solution is quite obvious, as the coffee will asymptotically cool to the air temperature, which is 20 C. We can run the simulation for a longer time and, as shown below, we do head towards the correct steady state solution.



Finally, a small experiment would go a long way in validating our simulation. We could get a well-insulated mug, pour in some water that is at 90 C, and then measure the surface temperature as it cools. We could even measure the volume of water before and after to check whether our evaporation model is reasonable. This isn't always possible, but is in many cases the best way to validate a model.

Incorporating Events into Solution: `ode45`

One final note. An interesting aspect of this problem is that our original question dealt with the time that our coffee reaches 60 C, but because we only calculate the temperatures at discrete time steps, we will not get a precise point at which we reach 60 degrees. One approach would be to interpolate between the last point above 60 and the first point below 60, but there is another way. The new infrastructure for solving these types of problems allows for what they call event detection.

The approach is to create an event function that describes what events we are searching for and how we want them handled. Then we call the ode solver with a few extra parameters to get the event information. Our code for this is as follows:

```
clear variables
E = odeEvent;
E.EventFcn=@mugEventsFcn;
E.Response="stop";

F = ode;
F.ODEFcn = @odefunc;
F.InitialValue = [90+273; 0.3];
F.RelativeTolerance=1e-6;
F.EventDefinition=E;
sol = solve(F,0,3000)
figure,plot(sol.Time,sol.Solution(1,:)-273)
xlabel('time (s)'); ylabel('temperature (C)');

function eqnfuncs=odefunc(t,res)
T=res(1);
mass=res(2);

D=0.05; A=pi*D^2/4;
emiss=0.9; stef=5.67e-8;
Ta=20+273;
cpliquid=4000; cpair=1000;
deltaH=4.2e6;
MWair=29; MWwater=18;
pt=1e5; %Pa

h=1.31*((T-Ta)/D)^(1/4);

c1=pt-0.5*pv(Ta);
c2=pt-pv(Ta);
F=(c1-c2)/log(c1/c2);
W=h*MWwater/MWair/cpair/F*A*(pv(T)-pv(Ta));

eqnfuncs=[(-emiss*stef*A*(T-Ta).^4-h*A*(T-Ta)-
W*deltaH)/mass/cpliquid; -W];
end

function p=pv(T)
p=1e5*exp(-40700/8.31*(1/T-1/373));
end

function [position,isterminal,direction] = mugEventsFcn(t,y)
  position = y(1)-(60+273);
  isterminal = 1;
```

```
    direction = 0;
  end
```

The function defining our differential equation does not change. The key is we add an event function, seen at the bottom of the script above. The “position” variable defines a function that will be zero when our event is reached. The “isterminal” variable says to stop the execution when the event is reached. There are other “isterminal” options available. Everything else is pretty straightforward.

Once we have this event code prepared, we can run the script and query sol.Time(end) and we find that the coffee reaches 60 C at 2594 seconds, which is more precise than our previous estimate which resulted from a glance at the plot.

Ordinary Differential Equations: Boundary Value Problems

Boundary value problems are very similar to the previously discussed initial value problems, except rather than applying all known conditions at one point, you apply them at two or more points. Typically these problems involve spatial variations in the dependent variables, rather than time, so common applications involve heat conduction, fluid flow, beam deflection, etc. As a model, consider

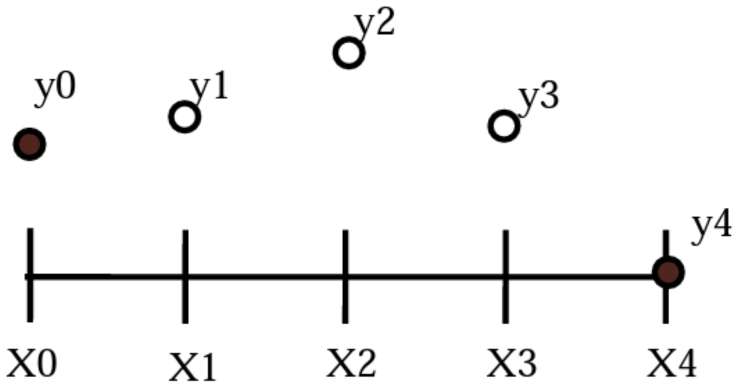
$$\frac{d^2y}{dx^2} + y = 1$$

$$y(0) = 1$$

$$y\left(\frac{\pi}{2}\right) = 0$$

Finite Difference Approach

One common approach for solving problems such as this is the finite difference method. In this case you divide the region 0<x<π/2 into small segments and approximate the differential equation as a series of algebraic equations. For example, if we divide our region into four pieces, it might look something like this



The boundary points are already filled because those values are given by the boundary conditions. Therefore, we have three unknowns and need three equations to solve the system. Hence, we write our approximation for the differential equation at each of the internal points and that gives us what we need.

To convert the differential equation to an approximate algebraic equation, we can rewrite the second derivative term at point *x_i* as

$$\frac{d^2y}{dx^2} \approx \frac{y_{i-1} - 2y_i + y_{i+1}}{s^2}$$

where *s* is the distance between consecutive *x* values. Using this approximation and substituting it into the differential equation yields

$$\frac{y_{i-1} - 2y_i + y_{i+1}}{s^2} + y_i \approx 1$$

or

$$y_{i-1} - (2 - s^2)y_i + y_{i+1} \approx s^2$$

If we write this equation at points 1, 2 and 3, we get

$$y_0 - (2 - s^2)y_1 + y_2 \approx s^2$$

$$y_1 - (2 - s^2)y_2 + y_3 \approx s^2$$

$$y_2 - (2 - s^2)y_3 + y_4 \approx s^2$$

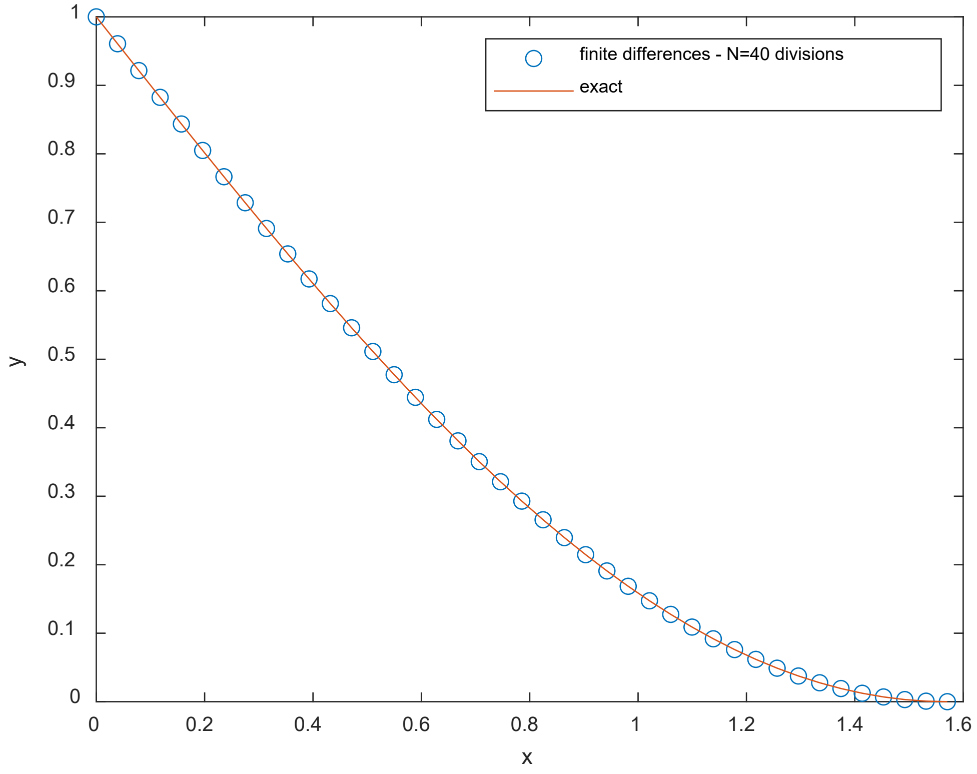
If we substitute the known values for y0 and y4, then the system is easily solved for the internal values of y. We can write this system in matrix form as

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & -(2-s^2) & 1 & 0 & 0 \\ 0 & 1 & -(2-s^2) & 1 & 0 \\ 0 & 0 & 1 & -(2-s^2) & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \end{Bmatrix} = \begin{Bmatrix} 1 \\ s^2 \\ s^2 \\ s^2 \\ 0 \end{Bmatrix}$$

This is easily solved. Note that the first row here defines the boundary condition at x=0, the last row defines the boundary condition at x= π/2, and the rest of the matrix can be thought of as a tridiagonal matrix with constant terms. Hence, this is easily generalized to any number of mesh points by adjusting the sizes of the matrix and vectors and modifying *s* accordingly. A generalized code for solving this problem is as follows:

```
N=40; %number of divisions on interval
s=pi/2/N; %mesh spacing
v=ones(N,1);
A=-(2-s^2)*eye(N+1)+diag(v,1)+diag(v,-1);
A(1,2)=0; A(N+1,N)=0;
A(1,1)=1; A(N+1,N+1)=1;
b=s^2*ones(N+1,1);
b(1)=1; %boundary condition at x=0
b(N+1)=0; %boundary condition at x=pi/2
x=linspace(0,pi/2,N+1); %fill vector with x values
yexact=1-sin(x);
y=A\b; %solve the system
plot(x,y,'o',x,yexact)
xlabel('x'); ylabel('y');
legend('finite differences - N=40 divisions', 'exact')
```

The results for 40 divisions are as follows:



Built-In Function: `bvp4c`

Matlab has a built-in routine called `bvp4c` that can be used to solve these types of problems. An advantage for this routine is that it will solve nonlinear equations, whereas the matrix approach provided previously would have to be modified to handle nonlinearity. To solve our model equation using `bvp4c` we first must break the second order equation into two first order equations:

$$\frac{dy}{dx} = z$$

$$\frac{dz}{dx} = \frac{d^2y}{dx^2} = 1 - y$$

$$y(0) = 1$$

$$y\left(\frac{\pi}{2}\right) = 0$$

The code to solve this system is as follows:

```
clear
xlow=0; xhigh=pi/2; %set boundary locations
xint=linspace(xlow, xhigh, 41); %determine x locations in mesh
solinit= bvpinit(xint,[1 -1]); %set initial guesses for y and z
sol = bvp4c(@bvp4ode, @bvp4bc, solinit); %solve the system
yexact=1-sin(xint); %exact solution for comparison
Sxint= deval(sol, xint); %evaluate solution at x values
plot(xint, Sxint(1,:),'o',xint, yexact)
```

```
xlabel('x'); ylabel('y');
legend('bvp4c solution', 'exact')

function dydx= bvp4ode(x,w)
dydx= [w(2) 1-w(1)]; %define differential equations
end

function res = bvp4bc(wa,wb)
res = [wa(1)-1 wb(1)]; %define boundary conditions
end
```

In this code I assume that w(1) represents y and w(2) represents z. The `bvpinit` call sets the initial values for *y* and *z*. This is necessary because the `bvp4c` function is iterative and requires a guess to start the iterations. Some equations will converge with a wide range of guesses, while others require a fairly accurate guess for convergence. In this call, I use the parameter [1, -1], indicating that I guess y=1 and z=-1 for all values of x. You can provide full vectors for these guesses if needed.

The bvp4ode function defines our differential equations. The `bvp4bc` defines our two boundary conditions by setting up two vector elements that will both be 0 when the solution is found. Here wa is the solution vector at x=0 and wb represents the solution at x=π/2. Hence wa(1) represents y(0) and wb(1) represents y(π/2). If, for example, we needed to use the first derivative of y at x=(π/2), we would use wb(2).

Python Solution

To solve our model equation in Python, use:

```
import math
import numpy as np
import matplotlib.pyplot as plt
len=math.pi/2
numpts=41
h=len/(numpts-1)
diagv=np.ones(numpts)*(-1)*(2-h**2)
offvec=np.ones(numpts-1)
A=np.diag(diagv)+np.diag(offvec,1)+np.diag(offvec,-1)
A[0,0]=1
A[0,1]=0
A[numpts-1,numpts-1]=1
A[numpts-1,numpts-2]=0
B=np.ones(numpts)*h**2
B[0]=1
B[numpts-1]=0
y=np.linalg.solve(A,B)
x=np.linspace(0,len,numpts)
plt.plot(x,y)
```

There also is a built-in python routine called `solve_bvp` that we can use to solve boundary value problems. The solution for our model problem is

```
import numpy as np
import math
from scipy.integrate import solve_bvp
import matplotlib.pyplot as plt
def fun(x, y):
    return np.vstack((y[1], 1-y[0]))
def bc(ya, yb):
    return np.array([ya[0]-1, yb[0]])
n = 41
x = np.linspace(0, math.pi/2, n)
y = np.ones((2, x.size))
sol = solve_bvp(fun, bc, x, y, tol=1e-4)
plt.plot(sol.y[0])
plt.show()
```

Problems with Boundary Conditions that Depend on First Derivative (Neumann)

Some equations will involve the first derivative in the boundary conditions. This can be handled in the same way as the equation discussed above, but we have to find a way to address the boundary condition.

Consider the problem of heat conduction in a cooling fin, governed by

$$\frac{d^2T}{dx^2} - \frac{hC}{kA}\left(T - T_f\right) = 0$$

With boundary conditions

$$T(0) = T_0$$

$$\left.\frac{dT}{dx}\right|_{x=L} = 0$$

Here *T* is the temperature in the cooling fin, *h* is the heat transfer coefficient between the fin surface and the coolant, *C* is the circumference of the fin, *k* is the thermal conductivity of the fin, *A* is the cross-sectional area, T_f is the cooling fluid temperature, and *L* is the length of the fin. The second boundary condition assumes that there is no heat lost from the end of the fin.

We begin by deriving our approximate (algebraic) equation for the differential equation as follows:

$$\frac{T_{i-1} - 2T_i + T_{i+1}}{s^2} - \beta\left(T_i - T_f\right) = 0$$

$$\beta = \frac{hC}{kA}$$

Or

$$T_{i-1} - 2T_i + T_{i+1} - \beta s^2 \left(T_i - T_f\right) = 0$$

$$T_{i-1} - T_i(2 + \beta s^2) + T_{i+1} = -\beta s^2 T_f$$

To treat the boundary condition at x=L we can approximate the first derivative at x=L as

$$\left.\frac{dT}{dx}\right|_{x=L} \approx \frac{T_{N+1} - T_{N-1}}{2s}$$

Here I assume that T_N is the temperature at the last point, such that T_{N+1} is actually outside our region of interest. However, this approach provides better accuracy than if we just define the first derivative in terms of the last two mesh points. Using this approximation for our boundary condition of zero slope gives the following relationship: that $T_{N-1}=T_{N+1}$.

This result can be substituted into our finite difference approximation of the differential equation for *i=N*, giving

$$T_{N-1} - T_N(2 + \beta s^2) + T_{N+1} = -\beta s^2 T_f$$

or

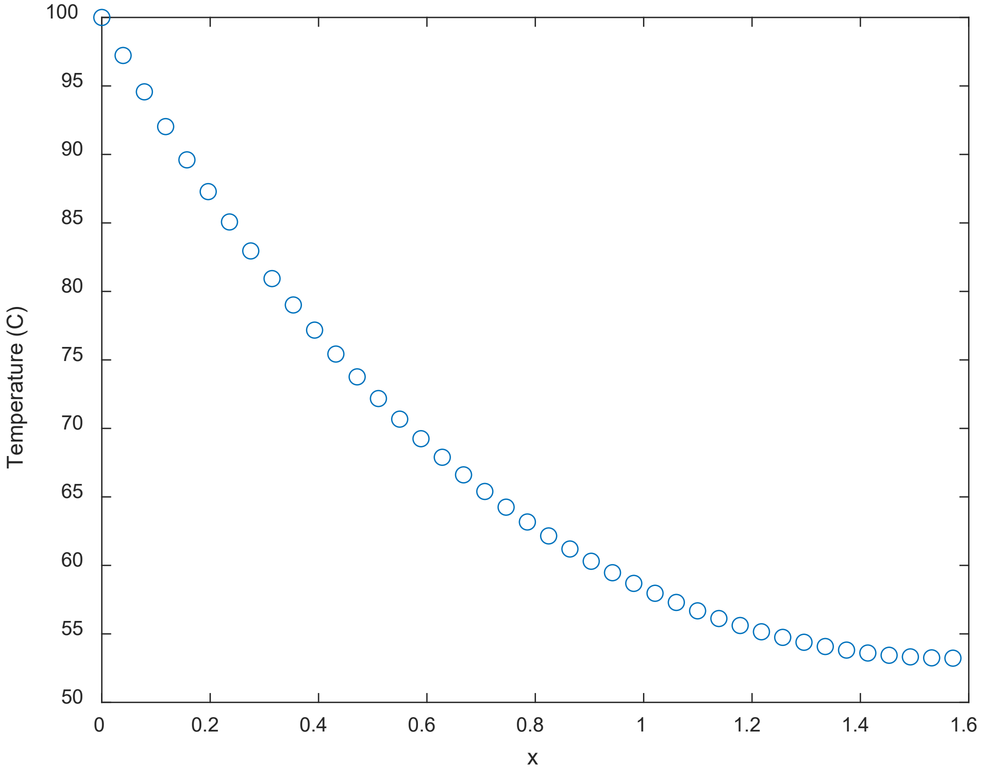
$$2T_{N-1} - T_N(2 + \beta s^2) = -\beta s^2 T_f$$

This last step eliminated any reference to points outside of our region of interest. Now we can solve our system by writing this equation for the last point and using our conventional equations for the other points. The code for this is as follows:

```
clear variables
pinradius=0.003; %meters
C=2*pi*pinradius; %circumference
A=pi*pinradius^2; %cross sectional area
h=1e3; %W/m^2-K heat transfer coefficient
k=100; %W/m=K thermal conductivity
Tbase=100; %centrigrade
Tfluid=25; %centigrade
pinlength=0.02; %meters
N=40; %number of divisions on interval
s=pinlength/N; %mesh spacing
beta=h*C/k/A;
v=ones(N,1);
A=-(2+beta*s^2)*eye(N+1)+diag(v,1)+diag(v,-1);
A(1,2)=0; A(N+1,N)=2;
A(1,1)=1;
b=-beta*s^2*Tfluid*ones(N+1,1);
b(1)=Tbase; %boundary condition at x=0
x=linspace(0,pi/2,N+1); %fill vector with x values
```

```
y=A\b; %solve the system
plot(x,y,'o')
xlabel('x'); ylabel('Temperature (C)');
```

The figure resulting from this code is



Neumann Boundary Conditions with bvp4c

We can also use bvp4c to solve this problem. The code is as follows:

```
clear variables
global beta Tfluid Tbase
pinradius=0.003; %meters
C=2*pi*pinradius; %circumference
A=pi*pinradius^2; %cross sectional area
h=1e3; %W/m^2-K heat transfer coefficient
k=100; %W/m=K thermal conductivity
Tbase=100; %centrigrade
Tfluid=25; %centigrade
pinlength=0.02; %meters

beta=h*C/k/A;

N=41; %number of mesh points
xlow=0; xhigh=pinlength; %set boundary locations
xint=linspace(xlow, xhigh, N); %determine x locations in mesh
```

```
solinit= bvpinit(xint,[Tbase -(Tbase-Tfluid)/pinlength]); %set
initial guesses for temperatureand gradient
sol = bvp4c(@bvp4ode, @bvp4bc, solinit); %solve the system
Sxint= deval(sol, xint); %evaluate solution at x values
plot(xint, Sxint(1,:),'o')
xlabel('x'); ylabel('Temperature (C)');

function dydx= bvp4ode(x,w)
global beta Tfluid Tbase
dydx= [w(2) beta*(w(1)-Tfluid)]; %define differential equations
end

function res = bvp4bc(wa,wb)
global beta Tfluid Tbase
res = [wa(1)-Tbase wb(2)]; %define boundary conditions
end
```

Validation of Boundary Value Problems

As a reminder, the list of strategies that we used for initial value problems was

- When you are new to a technique, test it on problems where the solution is known. Start by picking some easy problems for which the solution is known and make sure you can match the known results with your numerical solution.
- The most useful approach is something I think of as a global approach. That is, you should have some idea of the right answer prior to attempting the numerical solution.
- Always make sure your solution satisfies the boundary conditions. This is generally easily checked with a plot of the solution.
- Convergence study
- Benchmark against known solutions
- Approximate full problem with something we can solve analytically
- Test experimentally

These still apply for boundary value problems. As a model, consider the equations:

$$\varepsilon\frac{d^2y}{dx^2}+(1+\varepsilon)\frac{dy}{dx}+y=0$$

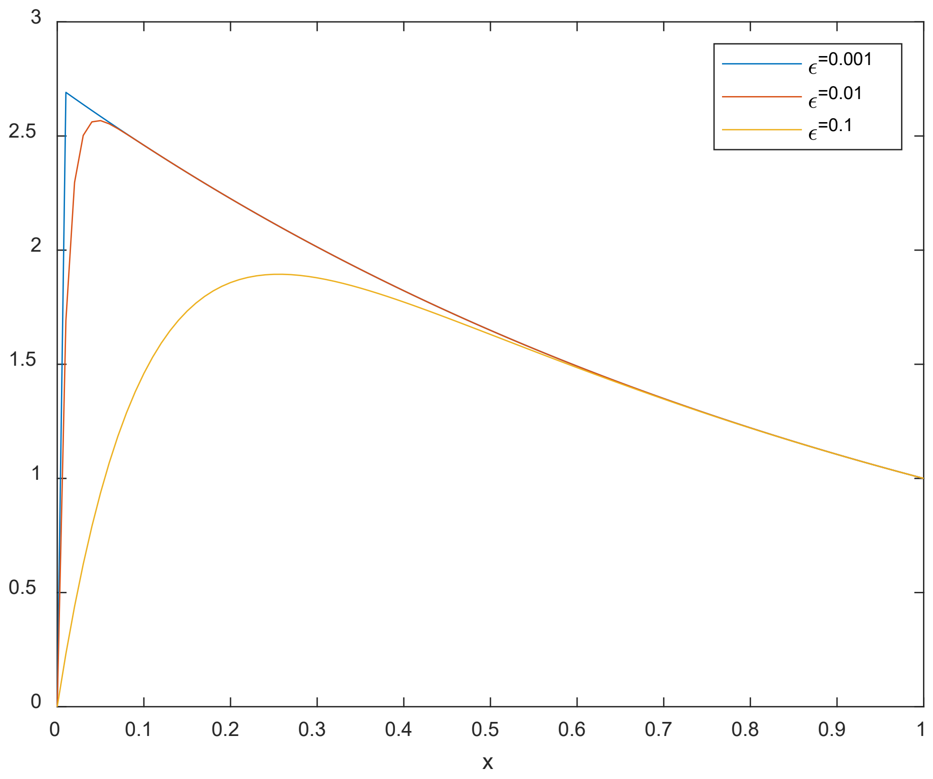
where ε is small relative to 1. The boundary conditions are y(0)=0 and y(1)=1. What we would like to know is the value of y at x=0.5. In this case, we don't have any connection to the real world, so our intuition is somewhat limited, as is our opportunity for testing. In this case, we will focus on a convergence study and the use of approximate solutions. This equation actually has a known analytical solution, so we can use that to study the effect of mesh spacing on the convergence of the relative error. The exact solution is

$$y=\frac{e^{-x}-e^{-x/\varepsilon}}{e^{-1}-e^{-1/\varepsilon}}$$

and the value of y at x=0.5 is 1.6306. Now we will carry out a set of runs for different values of ε and different mesh spacings to tabulate the effects.

Values of ε	Number of Divsions Used	Relative Error
0.1	20	4e-6
	40	3e-7
	80	2e-8
	160	6e-10
0.01	20	1e-6
	40	2e-7
	80	7e-10
	160	2e-11

Now let's consider some approximate solutions. In this case, it is instructive to plot the solution for different values of ε to develop an understanding of its character.



Here we see that as ε decreases, there is a boundary layer forming near the origin.

In this case, the ε parameter, which is stated to be small, provides an opportunity for perturbation theory. In this case, we seek a solution of the form:

$$y \approx y_0(x) + y_1(x)\epsilon + \cdots + y_n(x)\varepsilon^n$$

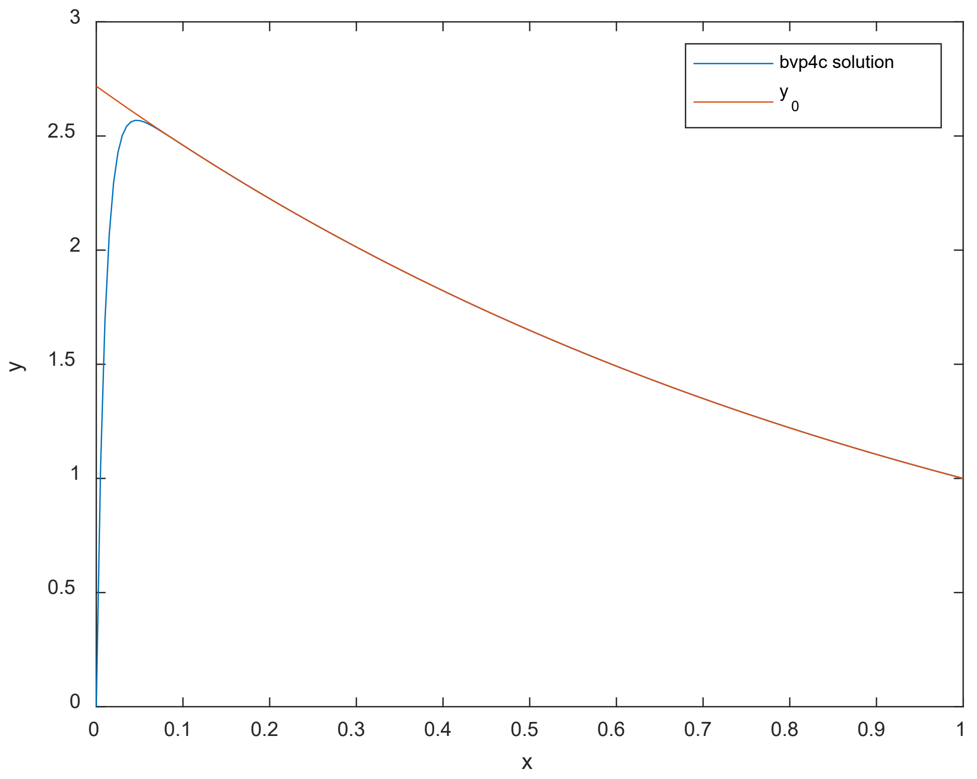
Substituting this into the original equation and equating terms with powers of ε, we obtain

$$\frac{dy_0}{dx} + y_0 = 0$$

This should provide a solution far away from the boundary layer, so we can look at the solution to this equation and make sure it matches the far field solution. Hence we can solve this first order equation and use the boundary condition at x=1 to form the far-field solution. The result is

$$y_0(x) = e^{1-x}$$

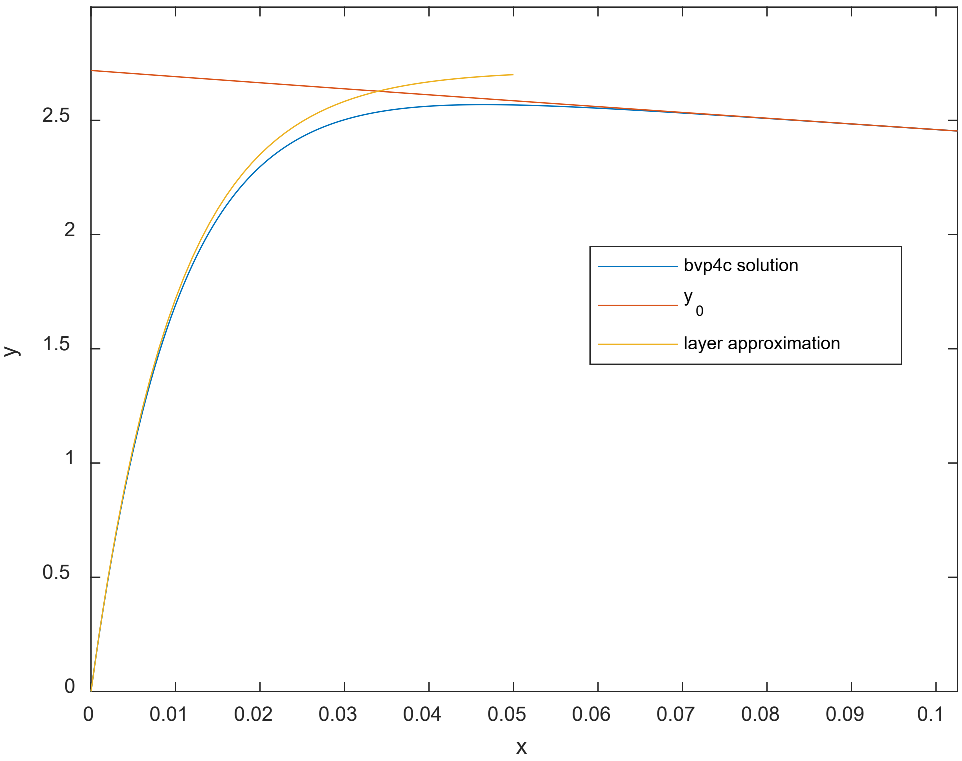
The results are as follows (for ε=0.1).



It is apparent that this far-field solution is matching well with the numerical solution of the original equation, so this gives us faith that our numerical solution is working. We can take this perturbation theory further and find a solution for the boundary layer itself. In this case, it can be shown that for small values of ε and for values of x near the origin, the solution can be approximated as:

$$y(x) \approx e\left(1 - e^{-x/\varepsilon}\right)$$

We can put this layer approximation together with the previously derived far-field solution and get an approximation that does a reasonable job for all x. The results are as follows (zooming in near the origin):



Again, the approximate solution helps to validate the numerical solution.

There won't always be a perturbation solution available to us for validation, but there often are other approximate solutions available, for example by removing certain nonlinearities that prevent an analytical solution. The key to the entire process is to always retain a skepticism about your numerical solutions.

Partial Differential Equations: Parabolic

Parabolic partial differential equations are typically time-dependent problems in one spatial dimension. More formally, we can define a general linear partial differential equation as follows:

$$A\frac{\partial^2 u}{\partial x^2} + B\frac{\partial^2 u}{\partial x \partial y} + C\frac{\partial^2 u}{\partial y^2} + D\frac{\partial u}{\partial x} + E\frac{\partial u}{\partial y} + Fu + G = 0$$

where the capital letters represent functions of x and y. In this form, a parabolic partial differential equation results when

$$B^2 - 4AC = 0$$

As a model, consider the parabolic partial differential equation:

$$\frac{\partial u(x,t)}{dt} = \frac{\partial^2 u(x,t)}{\partial x^2}$$

on the region

$$0 < x < 1$$

with initial condition:

$$u(x,0) = \varphi\,(x)$$

and boundary conditions:

$$u(0,t) = u(1,t) = 0$$

The analytical solution to this equation is:

$$u(x,t) = \sum_{n=1}^{\propto} A_n exp[-(n\pi)^2 t] sin(n\pi x)$$

$$A_n = 2\int_0^1 \varphi(x)\sin(n\pi x)\,dx$$

If we choose our initial conditions to be

$$\varphi(x) = \sin\,(\pi x)$$

then we obtain

$$A_n = 2\int_0^1 \sin\,(\pi x)\sin(n\pi x)\,dx = \begin{cases} 1 & n = 1 \\ 0 & n > 1 \end{cases}$$

which leaves us with

$$u(x,t) = exp(-\pi^2 t)\; sin(\pi x)$$

Finite Difference Methods: Explicit

To solve this equation numerically, we will again employ the finite difference method. We will divide time and space in such a way that the time steps and spatial divisions are uniform. We will refer to the values of *u* on this grade as $u_{i,j}$ where *i* refers to a spatial point on the mesh and *j* refers to a temporal point. Give this convention, we can approximate our partial differential equation at point *i,j* as:

$$\frac{u_{i,j+1} - u_{i,j}}{w} = \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{s^2}$$

where *w* represents the time between consecutive time steps and *s* represents the distance between consecutive spatial steps. By writing the spatial term at time *j*, we have created an explicit algorithm. For this approach, it is quite simple to derive a stepping algorithm as follows:

$$u_{i,j+1} = u_{i,j} + \frac{w}{s^2}\left(u_{i-1,j} - 2u_{i,j} + u_{i+1,j}\right)$$

With this approach, we can begin at the beginning of time (*t=0*) with our initial conditions and then write the results at the first time step (*j=1*) in terms of the initial values. Once that is complete, we can move on to the second time step and calculate all the values in terms of the results for the first step. We can continue this indefinitely without any iteration or solution of a linear system of equations. However, this simplicity comes at a cost since, as I'll discuss later, because it is numerically unstable for certain combinations of *w* and *s*.

Finite Difference Methods: Implicit

To avoid this instability, we can employ an implicit algorithm, which writes the spatial term at time *j+1* as follows:

$$\frac{u_{i,j+1} - u_{i,j}}{w} = \frac{u_{i-1,j+1} - 2u_{i,j+1} + u_{i+1,j+1}}{s^2}$$

Now we collect terms at common time steps, according to

$$-\frac{w}{s^2}u_{i-1,j+1} + \left(1 + 2\frac{w}{s^2}\right)u_{i,j+1} - \frac{w}{s^2}u_{i+1,j+1} = u_{i,j}$$

The solution approach here is to write a system of algebraic equations for the results to the end of the first time step in terms of the known initial values and then solve that system for those values at the end of the first step. Then we move on from time step to time step, solving a linear system of equations each time before moving on. This is more work than the explicit method, but it has the advantage of being stable for all combinations of *w* and *s*.

Sample code, employing both the explicit and implicit approaches is as follows:

```
clear variables
%explicit parabolic PDE
%
length=1;
ndiv=10; %number of spatial divisions
h=length/ndiv; %mesh spacing
alpha=0.25;
dt=alpha*h^2; %s
```

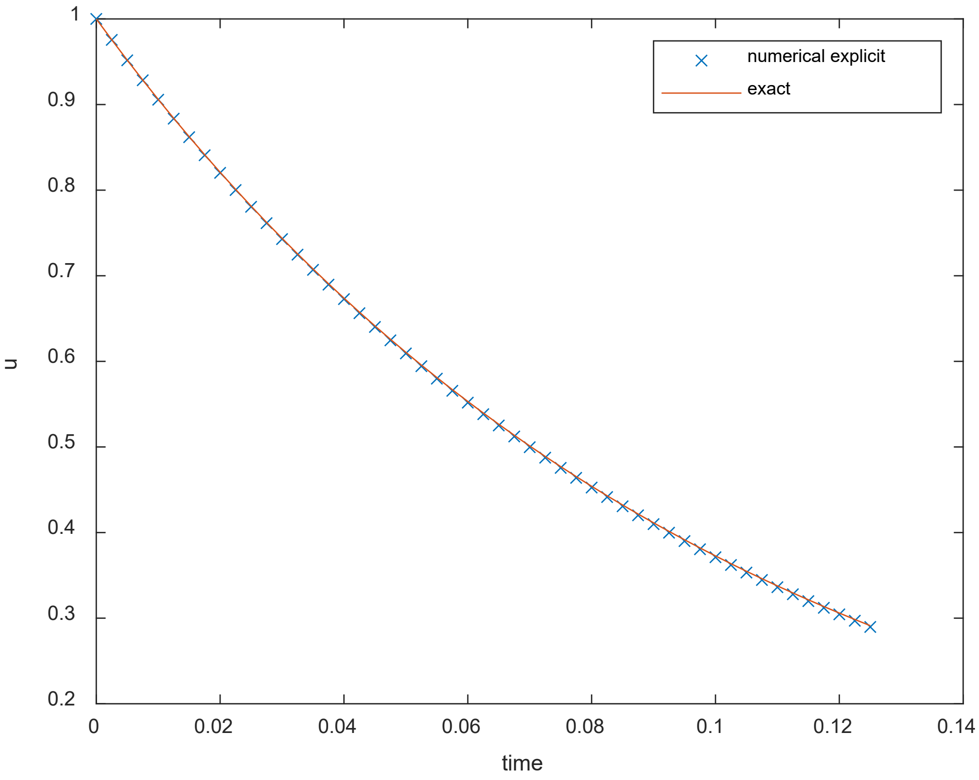
```
xvals=linspace(0,length,ndiv+1);
xvalshires=linspace(0,length,101);
%
ntimes=50; %number of time steps to do
time=zeros(1,ntimes+1);
uinitial=sin(pi*xvals);
uinitialhires=sin(pi*xvalshires);

uboundary=0;
u=zeros(ndiv+1,1);
umid=zeros(1,ntimes+1);
umidimplicit=zeros(1,ntimes+1);
uold=uinitial; %fill temperature vector with zeros
umid(1)=uinitial(ndiv/2+1);

for i=1:ntimes
    time(i+1)=i*dt;
    u(1)=uboundary;
    u(ndiv+1)=uboundary;
    for j=2:ndiv
        u(j)=(1-2*alpha)*uold(j)+alpha*(uold(j-1)+uold(j+1));
    end
    uold=u;
    umid(i+1)=u(ndiv/2+1);
end
ufinaldist=u;

umidexact=exp(-pi^2*time);

figure,plot(time,umid,'x',time,umidexact)
xlabel('time'); ylabel('u');
legend('numerical explicit','exact')

uexactdist=exp(-pi^2*time(end))*sin(pi*xvalshires);

figure,plot(xvals,ufinaldist,'x',xvalshires,uexactdist,xvalshires
,uinitialhires)
xlabel('x'); ylabel('u');
legend('final: explicit', 'final exact','initial condition')

%start implicit solution
umidimplicit(1)=uinitial(ndiv/2+1); %initial centerline
temperature
u=uinitial'; %fill temperature vector with zeros
a=(1+2*alpha)*eye(ndiv+1)-alpha*diag(ones(ndiv,1)',1)-
alpha*diag(ones(ndiv,1)',-1);
a(1,1)=1; a(1,2)=0; a(ndiv+1,ndiv+1)=1; a(ndiv+1,ndiv)=0;
for i=1:ntimes
```

```
    b=u;
    b(1)=0; b(ndiv+1)=0;
    u=a\b;
    umidimplicit(i+1)=u(ndiv/2+1);
end
figure, plot(time,umidimplicit,'o',time,umid,'x',time,umidexact)
xlabel('time'); ylabel('u');
legend('implicit','explicit','exact')

expliciterror=(umid-umidexact)./umidexact;
impliciterror=(umidimplicit-umidexact)./umidexact;
figure,semilogy(time,abs(impliciterror),'x',time,abs(expliciterro
r),'o')
xlabel('time'); ylabel('relative error');
legend('implicit','explicit')
```

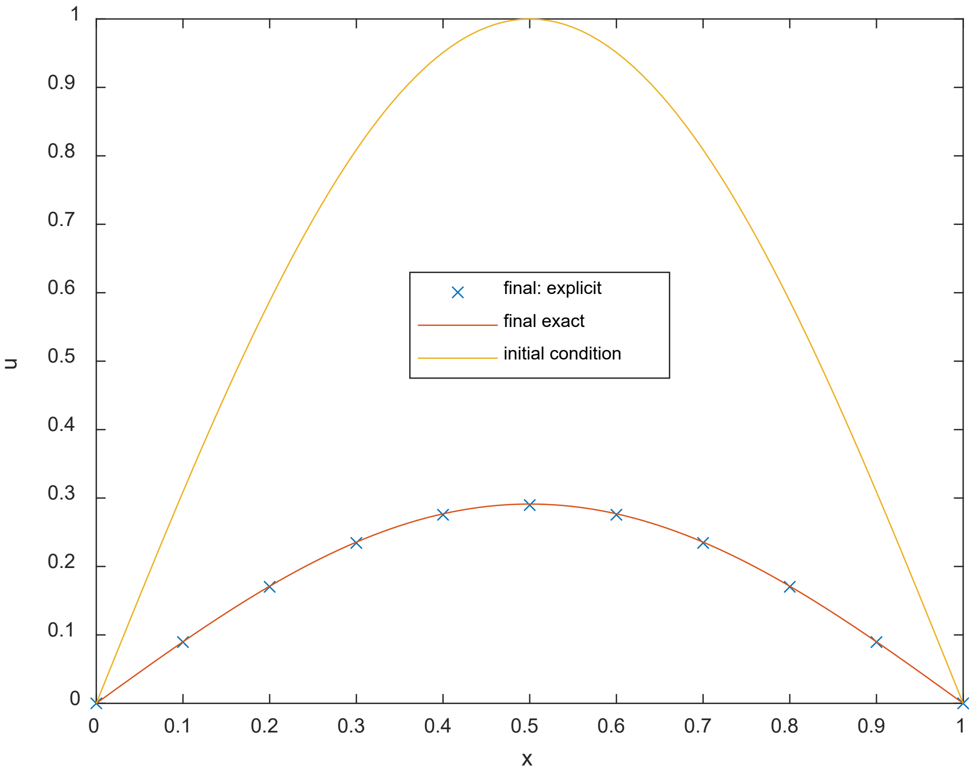
This script uses 10 spatial divisions and 50 time steps over a total time of 0.125 seconds, resulting in

$$\frac{w}{s^2} = 0.25$$

The results for the centerline temperature as a function of time, using the explicit algorithm, compared to the exact solution, are as follows:



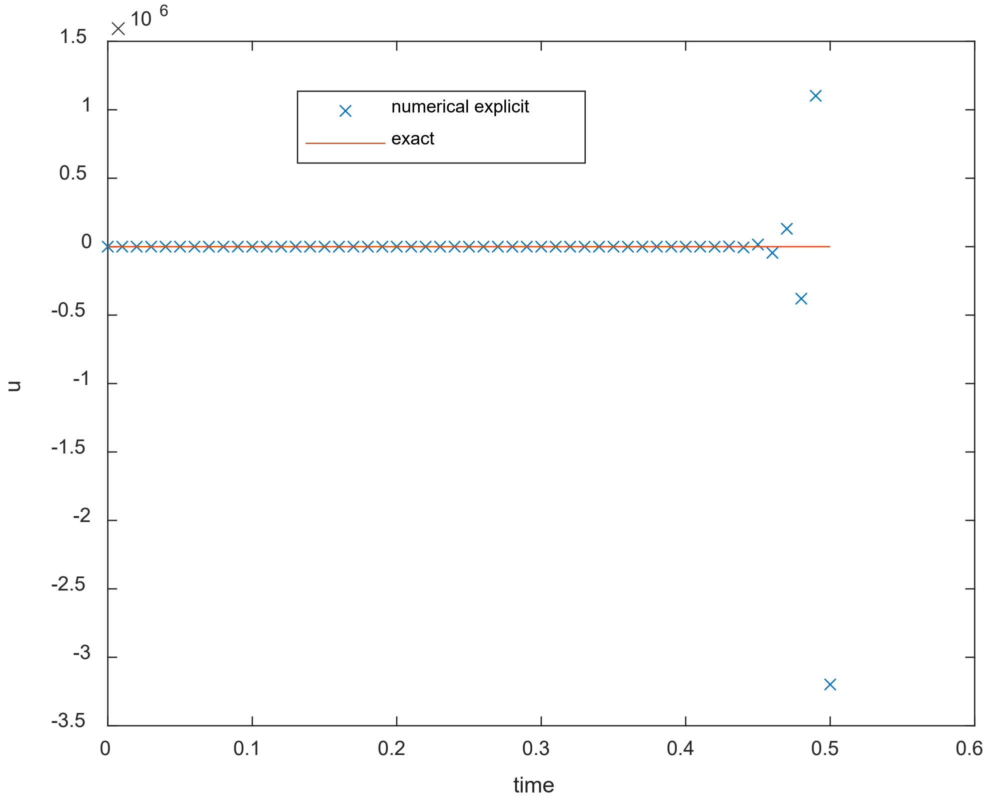
For the same case, we can also visualize the final spatial distribution of *u* as a function of x and compare the numerical and exact results.



To demonstrate the susceptibility of the explicit solution to instability, we can increase the time step by a factor of 4, such that

$$\frac{w}{s^2} = 1$$

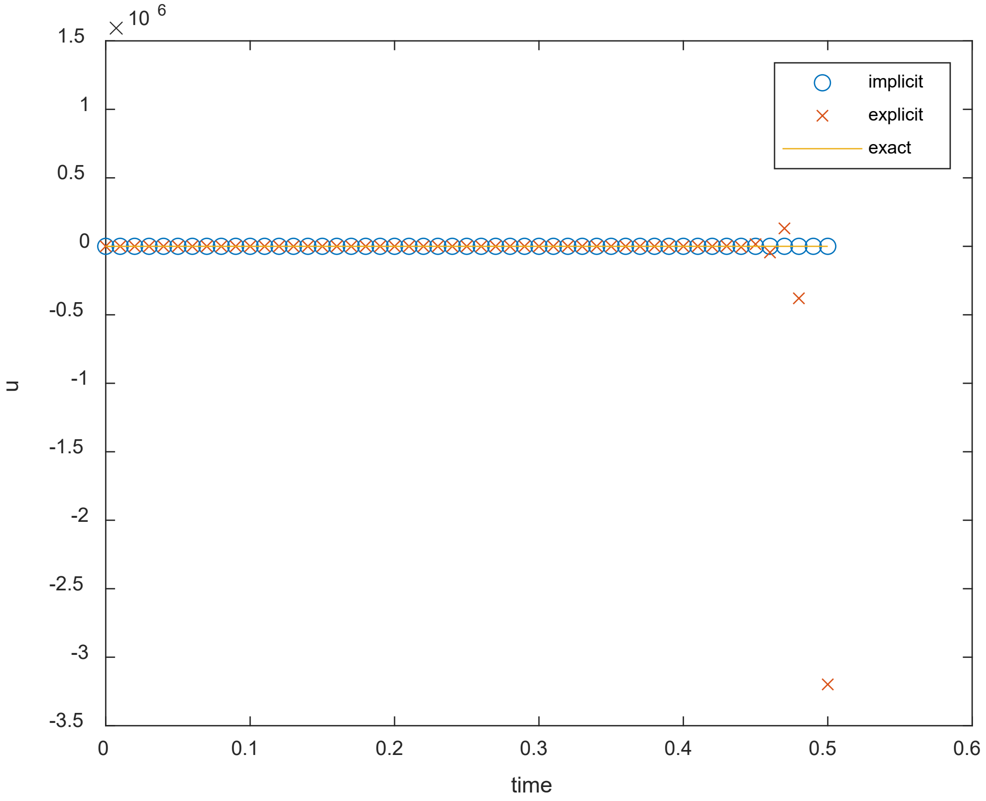
In this case, the plot of the central value of *u* as a function of time reveals the instability:



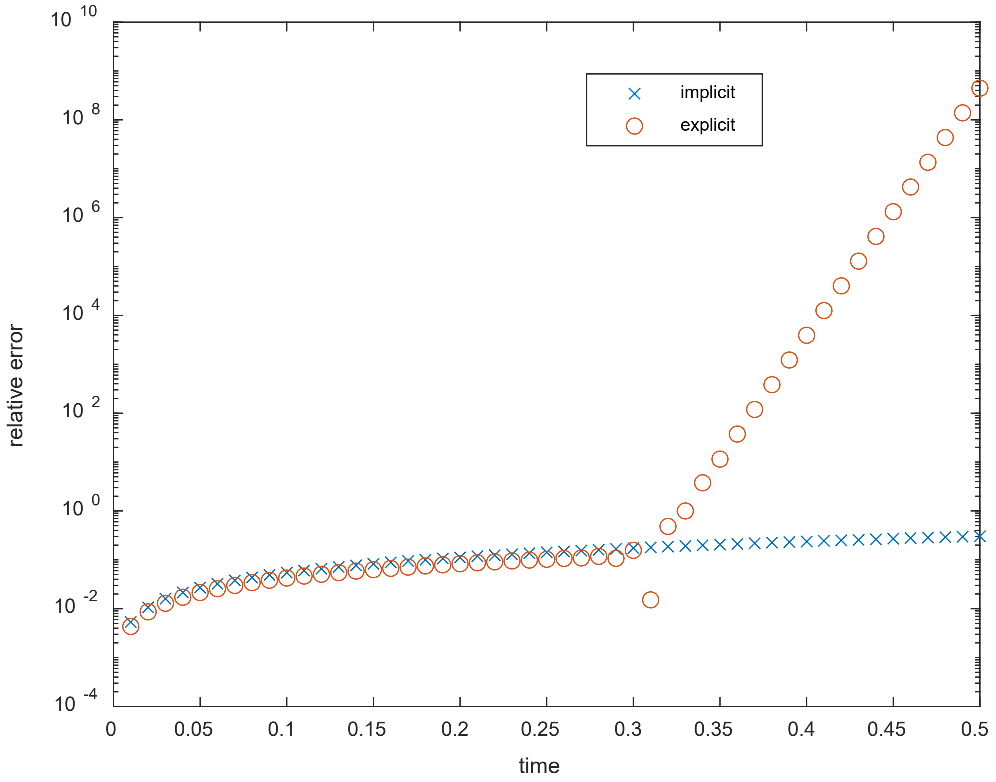
It is evident here that employing the explicit solution, which might be advantageous in some circumstances, requires the analyst to be vigilant about choosing appropriate values for the time and spatial step. To this end, it can be shown that the explicit solution will be stable when

$$\frac{w}{s^2} < 0.5$$

To demonstrate the innate stability of the implicit algorithm, we can compare the implicit and explicit algorithm results for the central value of *u*.



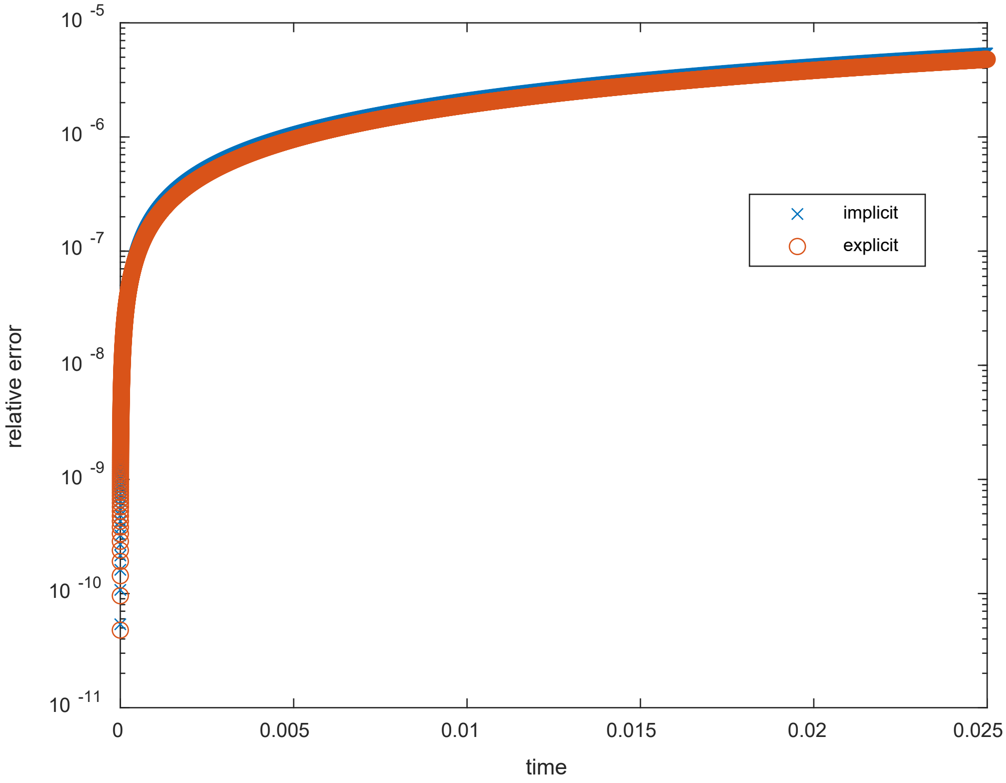
This is perhaps better visualized with a semilog plot of the relative error for the two numerical solutions:



In general, we would use more spatial and temporal points in a solution such as this. As an example, consider the problem solved above with 200 spatial divisions and 100,000 time steps over a total time of 0.025 seconds. This gives us

$$\frac{w}{s^2} = 0.01$$

The resulting relative accuracy plot becomes



You can see that the relative accuracy has improved by several orders of magnitude.

Built-In Function: `pdepe`

Matlab has a built-in function called `pdepe` for solving parabolic partial differential equations. It solves partial differential equations of the form:

$$c\left(x,t,u,\frac{\partial u}{\partial x}\right)\frac{\partial u}{\partial t} = x^{-m}\frac{\partial}{\partial x}\left[x^m f\left(x,t,u,\frac{\partial u}{\partial x}\right)\right] + s\left(x,t,u,\frac{\partial u}{\partial x}\right)$$

The parameter *m* is used to allow for different coordinate systems. That is, *m*=0, for cartesian symmetry, 1 for cylindrical symmetry, and 2 for spherical symmetry. For our model problem, we can use the following:

$$m = 0$$

$$c\left(x,t,u,\frac{\partial u}{\partial x}\right) = 1$$

$$f\left(x,t,u,\frac{\partial u}{\partial x}\right) = \frac{\partial u}{\partial x}$$

$$s\left(x,t,u,\frac{\partial u}{\partial x}\right) = 0$$

We use a user-defined function to define the initial distribution for *u*. The boundary conditions are set in a separate function, using the following algorithm:

$$p(x,t,u) + q(x,t)f\left(x,t,u,\frac{\partial u}{\partial x}\right) = 0$$

We set values for *pl* and *ql* to define the left boundary condition and *pr* and *qr* to define the right boundary condition. Since our function for *f* is just the gradient of *u*, then *ql* and *qr* will be zero when our boundary conditions do not involve the gradient (which is the case for our model problem) and non-zero when the boundary conditions do involve the gradient. For our model problem, we thus have

$$pl = ul$$

$$ql = 0$$

$$pr = ur$$

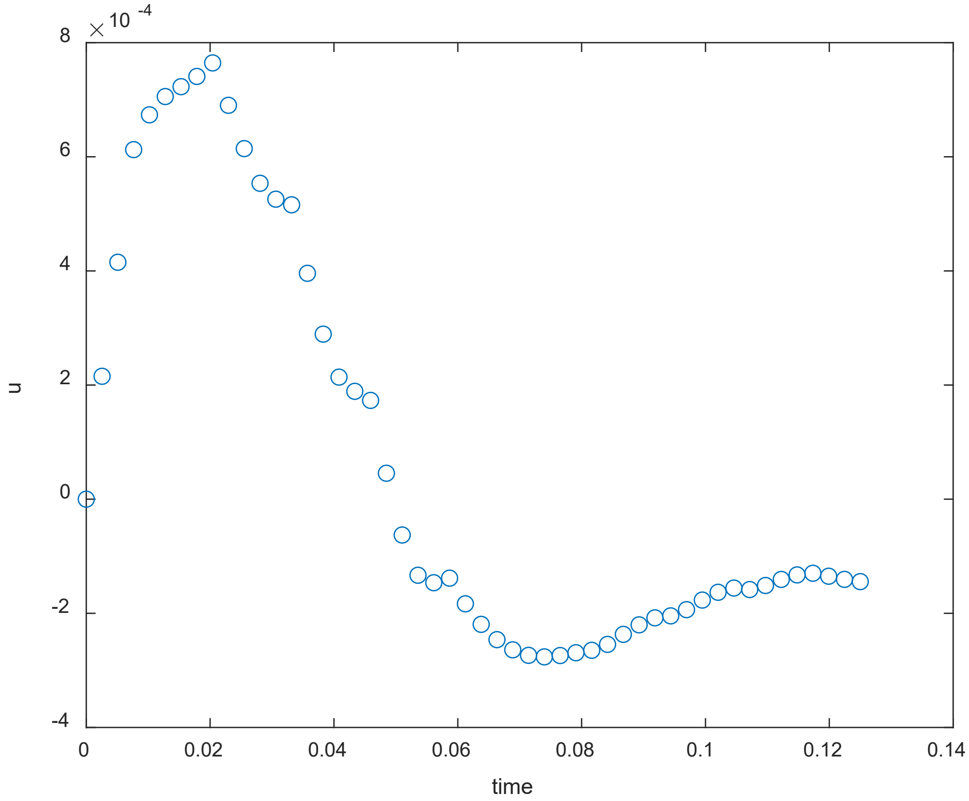
$$qr = 0$$

Sample code for our model equation is as follows:

```
length=1;
tend=0.125;

m = 0;
x = linspace(0,length,201);
t = linspace(0,tend,50);

sol = pdepe(m,@pdex1pde,@pdex1ic,@pdex1bc,x,t);
u = sol(:,:,1);

figure, plot(x,u(end,:))
title(strcat('Solution at t = ', num2str(tend)))
xlabel('x')
ylabel('u')

figure, plot(t,u(:,101))
xlabel('t')
ylabel('u')

uexact=exp(-pi^2*t);
figure,plot(t,uexact)
relerror=(u(:,101)'-uexact)./uexact;
figure, plot(t,relerror,'o')
xlabel('time'); ylabel('u');

function [c,f,s] = pdex1pde(x,t,u,DuDx)
c=1;
f = DuDx;
s = 0;
```

```
end

function u0 = pdex1ic(x)
u0 = sin(pi*x);
end

function [pl,ql,pr,qr] = pdex1bc(xl,ul,xr,ur,t)
pl = ul;
ql = 0;
pr = ur;
qr = 0;
end
```

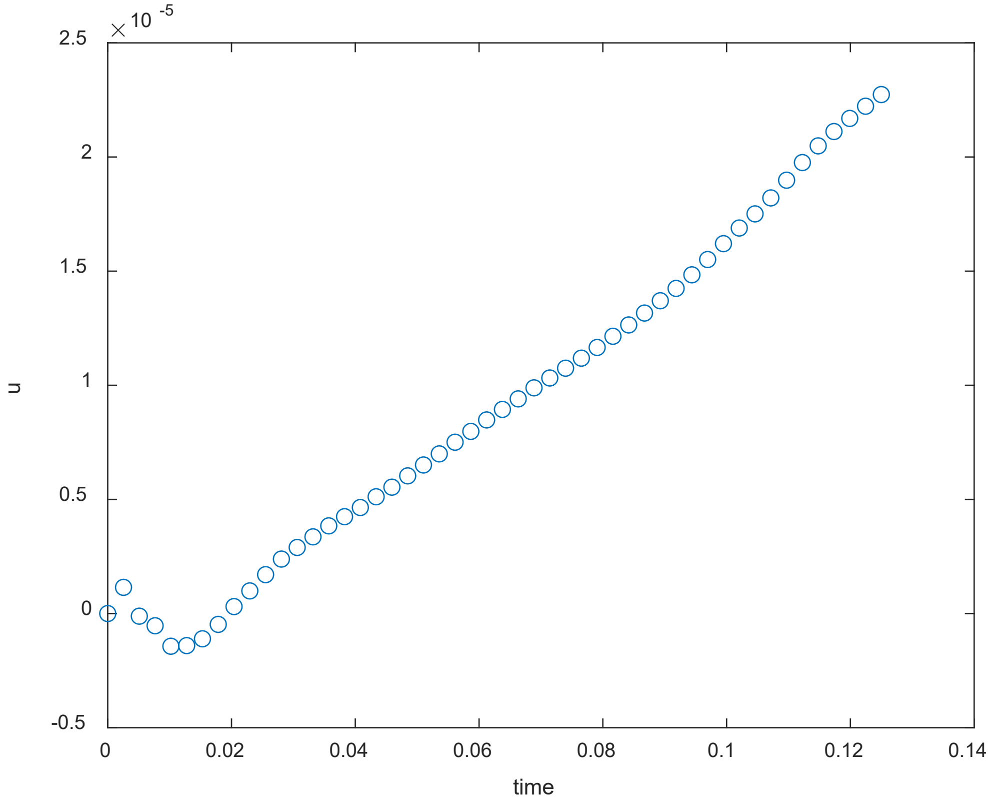
A plot of the relative error resulting from this code is



To improve the accuracy, we can set one or both of the *RelTol* and *AbsTol* parameters with a command such as:

```
options = odeset('RelTol',1e-6,'AbsTol',1e-6);
sol = pdepe(m,@pdex1pde,@pdex1ic,@pdex1bc,x,t,options);
```

which gives, for our model problem,



Python Solution

A python solution to the problem is as follows:

```
import math
import numpy as np
import matplotlib.pyplot as plt
n=10
imid=round(n/2)
L=1
h=L/n
alpha=0.25
dt=alpha*h**2
ntimes=50
uright=0
uleft=0
alpha=dt/h**2
diagv=np.ones(n+1)*(1+2*alpha)
offvec=np.ones(n)*(-1)*alpha
mymat=np.diag(diagv)+np.diag(offvec,1)+np.diag(offvec,-1)
mymat[0,0]=1
mymat[0,1]=0
mymat[n,n]=1
mymat[n,n-1]=0
mytime=np.zeros(ntimes)
```

```
myu=np.zeros(ntimes)
xvals=np.linspace(0,L,n+1)
ures=np.sin(np.pi*xvals)
myu[0]=ures[imid]
time=dt
for i in range(0, ntimes):
    b=ures
    b[0]=uleft
    b[n]=uright
    uout=np.linalg.solve(mymat,b)
    ures=uout
    time=time+dt
    mytime[i]=time
    myu[i]=uout[imid]

plt.plot(mytime, myu)
```

Validation of Parabolic Partial Differential Equations

In validating partial differential equations, we can still use the approaches discussed for ordinary differential equations:

- When you are new to a technique, test it on problems where the solution is known. Start by picking some easy problems for which the solution is known and make sure you can match the known results with your numerical solution.
- The most useful approach is something I think of as a global approach. That is, you should have some idea of the right answer prior to attempting the numerical solution.
- Always make sure your solution satisfies the initial conditions. This is generally easily checked with a plot of the solution.
- Convergence study
- Benchmark against known solutions
- Approximate full problem with something we can solve analytically
- Test experimentally

However, we have one critical new step which is to employ the use of steady state solutions. Quite often we can calculate the steady state solution of a parabolic partial differential equation analytically, which allows us to test our transient solution by checking the long term behavior against our analytical steady state solution.

To demonstrate this, let’s revisit the coffee mug cooling from before. In modeling the coffee mug, we assumed that any temperature gradients within the coffee were small. We can test this assumption using a parabolic PDE. In doing so, we will make some assumptions to simplify the model. Most importantly, properly modeling the problem would require that we consider evaporation of the coffee. However, doing so would require that we consider the impact of the moving boundary, which recedes as the coffee evaporates. There are models for this, but they are beyond the scope of this document.

Hence, we will model the coffee volume as fixed, but will attempt to include the cooling effect of the evaporation through an equivalent heat transfer coefficient at the surface. This model would be deficient if our goal was to properly model the cooling of coffee in a mug, but is probably adequate if our only goal is to estimate the impact of temperature gradients within the mug.

Using the assumptions described above, and assuming that the lateral gradients are negligible, we arrive at the following parabolic PDE for our mug:

$$\rho c_p \frac{dT}{dt} = k \frac{d^2 T}{dx^2}$$

with boundary conditions

$$-k \frac{dT}{dx}(0,t) = h[T(0,t) - T_a]$$

$$\frac{dT}{dx}(D,t) = 0$$

and initial condition

$$T(x,0) = T_0$$

Here ρ is the coffee density, c_p is the specific heat of the coffee, k is the thermal conductivity of the coffee, h is the equivalent heat transfer coefficient accounting for radiation, convection, and evaporation, D is the depth of the coffee in the mug, and T_0 is the initial coffee temperature, which is assumed to be uniform. To be consistent with our previous solution, we will take T_0 to be 90 C, T_a to be 20 C, and c_p to be 4,000 J/kg-K. We will assume the density of the coffee is 1,000 kg/m^3 and, to be consistent with the earlier problem, the depth of the coffee is 0.15 m. The thermal conductivity of the coffee will be taken to be 0.6 W/m-K.

Next, we must decide what heat transfer coefficient would best model our coffee. According to Condoret[3], the global transfer coefficient is approximately 13 W/m^2-K.

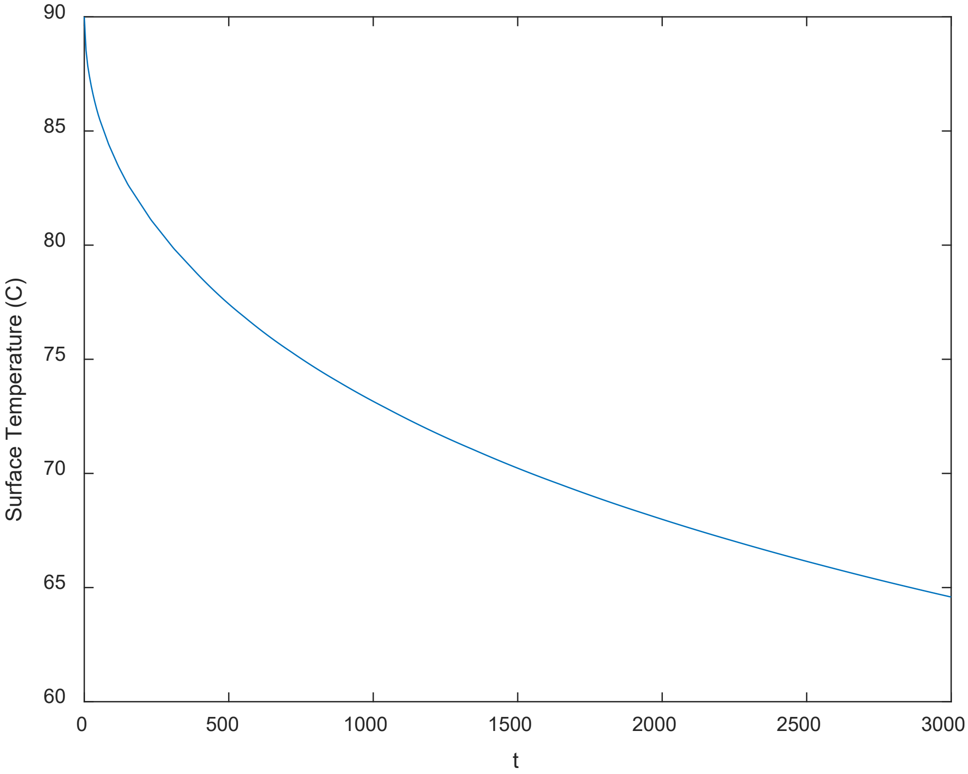
The code for this simulation is (using `pdepe`)

```
clear variables
length=0.15;
tend=3000;

m = 0;
x = linspace(0,length,201);
t = linspace(0,tend,500);

sol = pdepe(m,@pdex1pde,@pdex1ic,@pdex1bc,x,t);
u = sol(:,:,1);
```

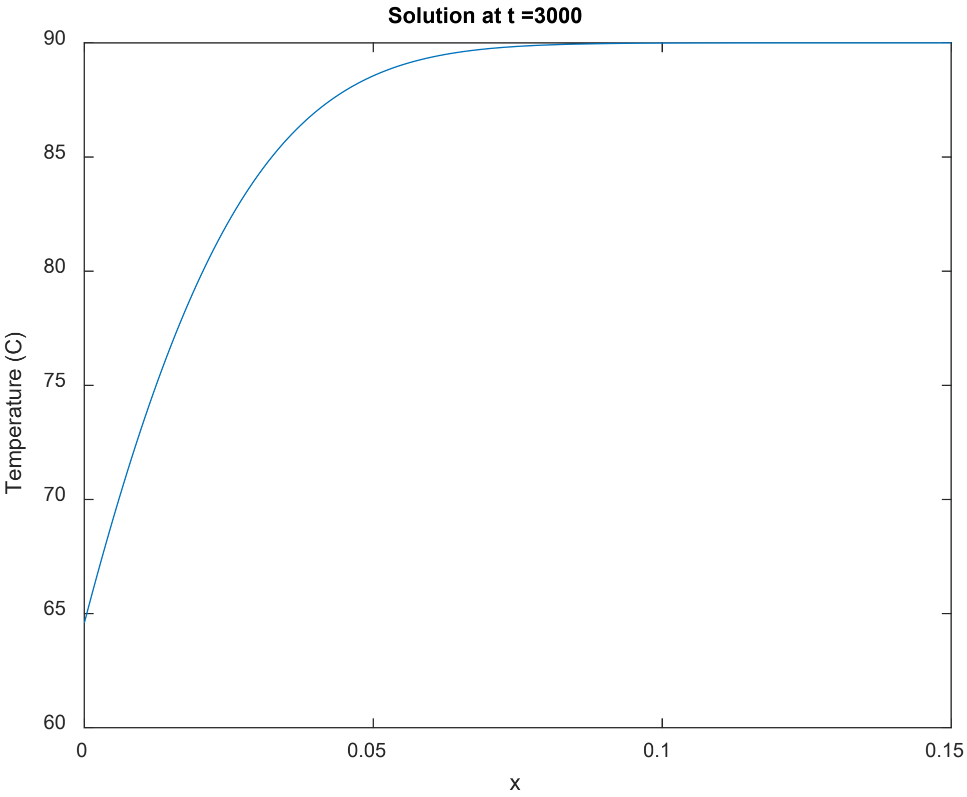
[3] Jean Stephane Condoret, "Teaching Transport Phenomena Around a Cup of Coffee," https://oatao.univ-toulouse.fr/1455/

```
figure, plot(x,u(end,:)-273)
title(strcat('Solution at t = ', num2str(tend)))
xlabel('x')
ylabel('Temperature (C)')

figure, plot(t,u(:,1)-273)
xlabel('t')
ylabel('Surface Temperature (C)')


function [c,f,s] = pdex1pde(x,t,u,DuDx)
k=0.6; %W/m-K
rho=1000; %kg/m3
cp=4000; %J/kg-K
c=rho*cp;
f = k*DuDx;
s = 0;
end

function u0 = pdex1ic(x)
u0 = 90+273; %uniform, Kelvin
end

function [pl,ql,pr,qr] = pdex1bc(xl,ul,xr,ur,t)
Ta=20+273; %K
h=13; %W/m2-K
pl = -h*(ul-Ta);
ql = 1;
pr = 0;
qr = 1;
end
```

The time dependence of the surface temperature is



And the spatial variation of the temperature after 3,000 seconds is



These results invite a few comments:

- Note that I ran the simulation using a variety of different time steps and mesh spacings to verify that my initial choices were adequate.
- These results indicate that the surface temperature after 3,000 seconds is roughly 65 C. Our previous model ignoring temperature gradients, predicted slightly less than 60 C after 3,000 seconds. Given that our model of the evaporative cooling here is quite crude, and that we are ignoring mass loss here, this agreement indicates that we probably haven't made any egregious errors in our PDE simulation. However, it also indicates that we should make an effort to include a better evaporation model here.
- The second figure above indicates that there is a substantial temperature gradient within our mug after 3,000 seconds, so our assumptions in our earlier lumped model were not valid. To properly model this problem, we should probably use a parabolic pde with proper consideration of evaporation. This forces us to address moving boundaries, which, as stated before, is beyond the scope of this document. A discussion of simulations with moving boundaries (for the case of melting of a solid can be found here): https://theses.gla.ac.uk/75461/1/13832024.pdf [4]

[4] Mohamed Zerroukat, "Numerical Computation of Moving Boundary Phenomena," Master's Thesis, University of Glasgow, 1993.

Partial Differential Equations: Elliptic

Returning to our previously discussed form for a general linear partial differential equation

$$A\frac{\partial^2 u}{\partial x^2} + B\frac{\partial^2 u}{\partial x \partial y} + C\frac{\partial^2 u}{\partial y^2} + D\frac{\partial u}{\partial x} + E\frac{\partial u}{\partial y} + Fu + G = 0$$

the equation is elliptic if

$$B^2 - 4AC < 0$$

These equations can typically be thought of as boundary value problems in two dimensions. Common examples are steady-state conduction or diffusion in two dimensions.

As a model, consider a square region with sides of width 1 and a dependent variable *u* governed by

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + 1 = 0$$

We will solve this equation assuming *u=0* on all boundaries. The exact solution for the maximum value of *u* (which occurs at the center of the region) is

$$u_{max} = \frac{1}{8} + \frac{1}{2}\sum_{k=1}^{\infty} \frac{(-1)^k}{\theta_k^3 \cosh \theta_k}$$

$$\theta_k = \frac{\pi}{2}(2k-1)$$

Finite Difference Approach

To solve this equation numerically, we again employ the finite difference method. Here we will assume we divide our square region into a mesh with a uniform grid, such that the grid spacing in both directions is equal. Also, we will use the convention that a value of *u* on a grid point is labeled as $u_{i,j}$ where *i* refers to the location in the *x* direction and *j* refers to the location in the *y* direction. Using these assumptions, our finite difference approximation for the equation can be written as

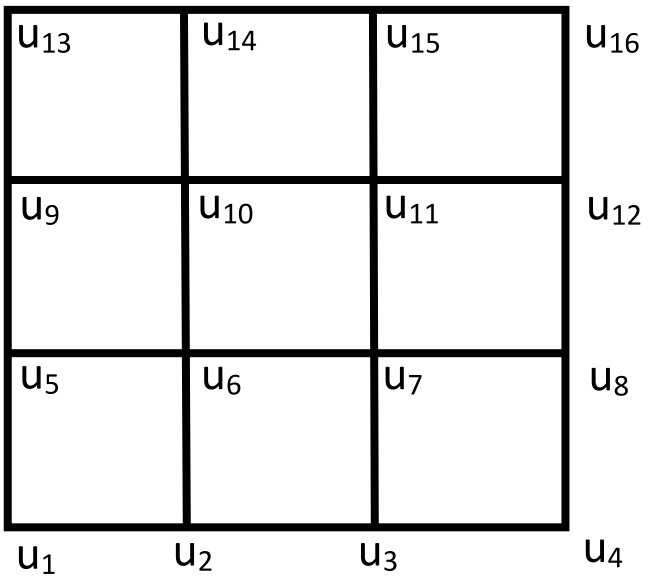
$$\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{s^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{s^2} + 1 = 0$$

or

$$u_{i-1,j} - 4u_{i,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} = -s^2$$

Where *s* is the mesh spacing. We write this equation at all the interior mesh points and solve the resulting algebraic system for the internal values of *u*. For coding purposes, it is convenient to number the values for *u* sequentially, so that the lower left corner is u_1 and subsequent values increment moving left to right and then bottom to top.

The hardest part of implementing this algorithm is working out the non-zero values of the matrix. If we use three divisions in each direction, then we have 4 mesh points in each direction, resulting in 16 unknown values of *u* (assuming that we consider the boundary values as unknowns). The mesh for three divisions is pictured below, with the numbering for the *u* values displayed adjacent to the mesh points.


u_{13}
u_{14}
u_{15}
u_{16}
u_9
u_{10}
u_{11}
u_{12}
u_5
u_6
u_7
u_8
u_1
u_2
u_3
u_4

Given that we have 16 unknowns (counting the boundary points), the *A* matrix from the code will be a 16x16 array. For the internal mesh points, our algorithm indicates that the value of *u* at each point will depend on the four nearest mesh points. If we number the equations from left to right and bottom to top, then the *A* matrix must address our 16 variables in order. The *A* matrix for our model equation becomes

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	1	-4	1	0	0	1	0	0	0	0	0	0
0	0	1	0	0	1	-4	1	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	1	-4	1	0	0	1	0	0
0	0	0	0	0	0	1	0	0	1	-4	1	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Every row here that contains the value -4 represents our finite difference equation being written at one of the 4 internal mesh points. The rest of the rows represent boundary points. For the internal mesh points, the -4 values are on the diagonal and each has an adjoining value of 1 on the off-diagonals. These points represent the dependence on the mesh points to the left and right of the point where the equation is written. So, for example, u_6 will depend on u_5 and u_7, giving us a -4 on the diagonal and values of 1 on the off-diagonals. The rows representing the internal mesh points also have two more non-zero terms, separated from the diagonal by 4 spots. These represent the dependence on the mesh points above and below. So, returning to mesh point 6, we have u_6 depending on u_2 and u_{10}, as shown in the mesh figure above. So we can see that row 6 of the *A* matrix has a -4 in column six and values of -1 in columns 2, 5, 7, and 10. The gap of 4 spots will depend on how many mesh points are in each row, so for a 5x5 mesh, this gap will be 5.

We can generalize the approach to the creation of the A matrix, with respect to the difference equations at the internal mesh points, as follows. Consider a square region divided up into N divisions in each direction, leading to N+1 mesh points in each direction and a total number of variables of $(N+1)^2$. If point *i* represents an internal mesh point, then row *i* will have a -4 in column *i* and -1 in columns *i-N-1*, *i-1*, *i+1*, and *i+N+1*.

This description will change if we change the original partial differential equation, but the locations of the non-zero terms will not change. The discussion is easily generalized to a rectangular region, with a different number of mesh points in each direction.

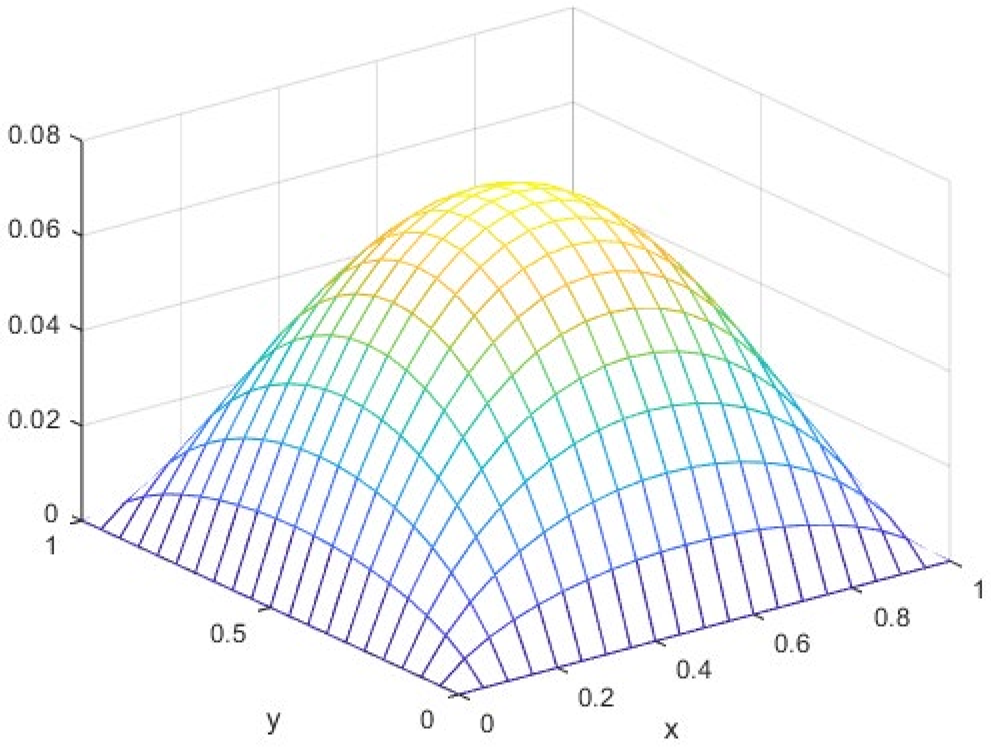
The rows representing the boundary points are somewhat easier to address. If a particular row in *A* represents a boundary point, we put a value of 1 on the diagonal in that row and leave everything else 0. Then we can place the desired boundary value of *u* in the *b* vector and we'll get the desired outcome.

The resulting code is as follows:

```
clear variables
numterms=10;
kvec=1:numterms;
lam_k=1/2*(2*kvec-1)*pi;
Tmax_a=(1/8+1/2*sum((-1).^kvec./(lam_k.^3.*cosh(lam_k))))

width=1;
ndiv=20;
npts=ndiv+1;
s=width/ndiv;
x=(0:ndiv)*s; y=x;
A=zeros(npts^2);b=zeros(npts^2,1);
for i=2:npts-1
    for j=2:npts-1
        eqnum=i+(j-1)*npts;
        up=eqnum+npts;
        down=eqnum-npts;
        left=eqnum-1;
        right=eqnum+1;
        A(eqnum,eqnum)=-4;
        A(eqnum,up)=1;
        A(eqnum,down)=1;
        A(eqnum,left)=1;
        A(eqnum,right)=1;
        b(eqnum)=-s^2;
    end
end
for i=1:npts % bottom and top boundary points as well as diagonal
of A
    A(i,i)=1;
    b(i)=0;
    A(i+(npts-1)*npts,i+(npts-1)*npts)=1;
    b(i+(npts-1)*npts)=0;
end
for j=2:npts-1 % left and right boundary points
    A(1+(j-1)*npts,1+(j-1)*npts)=1;
    b(1+(j-1)*npts)=0;
    A(npts+(j-1)*npts,npts+(j-1)*npts)=1;
    b(npts+(j-1)*npts)=0;
end
v=A\b; % solve for solution in v labeling
u=reshape(v(1:npts^2),npts,npts);  %translate from v to u
```

```
mesh(x,y,u')
xlabel('x');ylabel('y');
umax_direct=max(max(u))
```

This produces a plot of *u* as follows



The peak analytical value with 10 terms in the series is 0.0737 and the peak numerical value with 20 division in each direction is 0.0735. This is a relative error of 0.002. If we increase to 40 divisions, the relative error drops to 5x10^{-4}.

One issue with this approach is the size of the matrices we are trying to employ. For the 20x20 division mesh, our matrix is 441x441. Hence, with a typical computer we will run out of memory at some point. The computation also slows considerably as the matrices grow. To combat these issues with matrix size, we can invoke Matlab's sparse matrix features. Here we only define the non-zero terms in our matrix. Sample code for this model problem is as follows:

```
clear all
width=1;
ndiv=200;
npts=ndiv+1;
s=width/ndiv;
x=(0:ndiv)*s; y=x; % set mesh values
A=sparse(npts^2);b=zeros(npts^2,1);
for i=2:npts-1 % interior points
```

```
        for j=2:npts-1
            A(i+(j-1)*npts,i-1+(j-1)*npts)=1;
            A(i+(j-1)*npts,i+1+(j-1)*npts)=1;
            A(i+(j-1)*npts,i+(j-1)*npts)=-4;
            A(i+(j-1)*npts,i+(j-2)*npts)=1;
            A(i+(j-1)*npts,i+j*npts)=1;
            b(i+(j-1)*npts)=-s^2;
        end
    end
    for i=1:npts % bottom and top boundary points as well as diagonal
    of A
        A(i,i)=1;
        b(i)=0;
        A(i+(npts-1)*npts,i+(npts-1)*npts)=1;
        b(i+(npts-1)*npts)=0;
    end
    for j=2:npts-1 % left and right boundary points
        A(1+(j-1)*npts,1+(j-1)*npts)=1;
        b(1+(j-1)*npts)=0;
        A(npts+(j-1)*npts,npts+(j-1)*npts)=1;
        b(npts+(j-1)*npts)=0;
    end
    v=A\b; % solve for solution in v labeling
    u=reshape(v(1:npts^2),npts,npts);  %translate from v to u
    mesh(x,y,u')
    xlabel('x');ylabel('y');
    umax_direct=max(max(u))
```

On my personal desktop computer, this script runs in about 10 seconds using 200 divisions in each direction, while the previous script took more than 250 seconds. Both gave the same results, with a relative error of about $2x10^{-5}$.

Iterative Approach

An alternative to this approach is an iterative algorithm. This has the advantage of permitting solution of nonlinear equations. The basic idea is to make a guess for the solution and then use the finite difference approximation to provide an iterative algorithm that, hopefully, converges to an adequate solution. For linear equations, this approach is reliable, but somewhat slow. For nonlinear equations, convergence often relies on a high quality guess to ensure convergence.

To develop an iteration algorithm, we begin with the same finite difference approach mentioned above, solving for $u_{i,j}$ in terms of the values at the neighboring terms. We then start in one corner of our region and progressively update the solution as we march through the mesh points. We continue this process until we have reached a solution which sufficiently stable.

For our model problem, the algorithm is as follows:

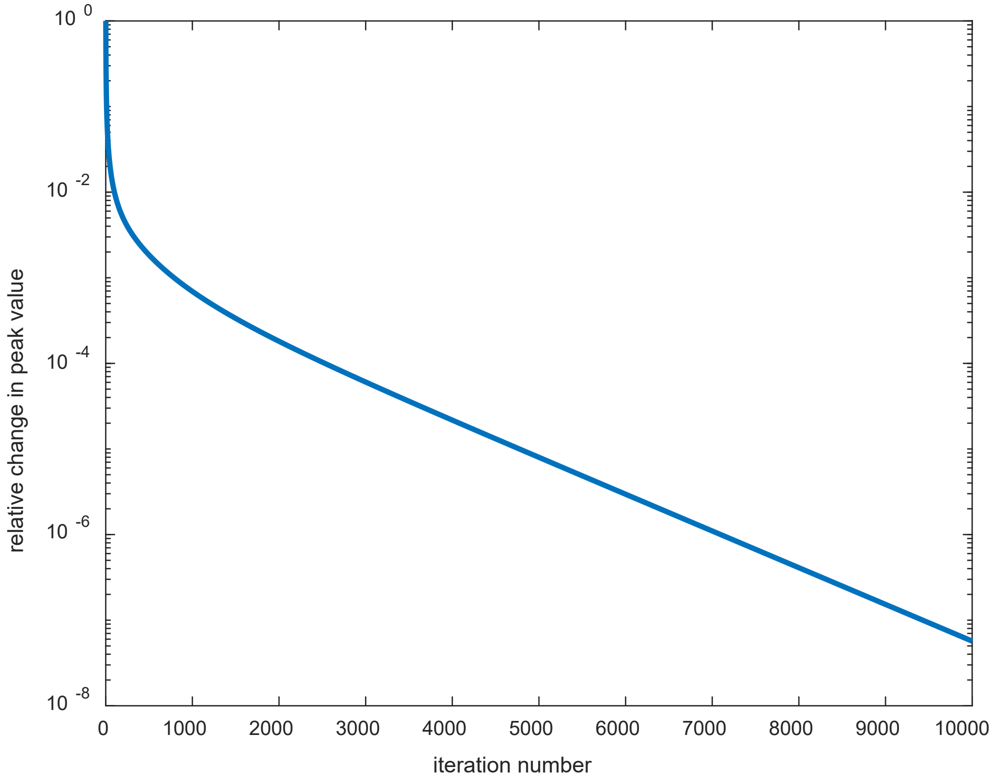
$$u_{i,j} = \frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} + s^2}{4}$$

Sample code for this approach is as follows:

clear variables

```
clear variables

numterms=10;
kvec=1:numterms;
lam_k=1/2*(2*kvec-1)*pi;
umax_analytical=(1/8+1/2*sum((-
1).^kvec./(lam_k.^3.*cosh(lam_k))))

width=1;
%set number of divisions in both directions
ndiv=100; npts=ndiv+1;
centeri=round(ndiv/2+1); centerj=centeri;
%calculate width of divisions in both directions
s=width/ndiv;
x=linspace(0,width,npts); y=x;
u=zeros(npts,npts,'double');
wpeakold=0;
numiters=10000;
relativechange=zeros(numiters,1);
for k=1:numiters
    for i=2:npts-1
        for j=2:npts-1
            u(i,j)=(u(i,j+1)+u(i,j-1)+u(i-1,j)+u(i+1,j)+s^2)/4;
        end
    end
    %track relative change in peak value of w to track
convergence
    newwpeak=u(centeri,centerj);
    relativechange(k)=(abs(wpeakold-newwpeak)/newwpeak);
    wpeakold=newwpeak;
end
Maximum_value_iteration=newwpeak
relerror=(umax_analytical-
Maximum_value_iteration)./umax_analytical
semilogy(1:numiters,relativechange, 'LineWidth',2)
xlabel('iteration number'); ylabel('relative change in peak
value');
set(gca,'FontSize',10)
figure,mesh(x,y,u')
xlabel('X'); ylabel('Y');
figure,contour(x,y,u','ShowText','on');
xlabel('X');ylabel('Y')
```

For an initial guess of u=0 at all internal mesh points, this code obtains a relative error of about 10^{-4} for 10,000 iterations. The convergence is indicated by the plot below.



The code will converge faster if we use successive overrelaxation, which increases the change from step to step by a constant. In other words, whereas our previous iteration algorithm can be thought of as

$$u_{i,j}^{new} = u_{i,j} + \left(u_{i,j}^{new} - u_{i,j}\right)$$

Now we replace this by

$$u_{i,j}^{new} = u_{i,j} + L\left(u_{i,j}^{new} - u_{i,j}\right)$$

where L is a constant. This equation can be written as

$$u_{i,j}^{new} = (1-L)u_{i,j} + Lu_{i,j}^{new}$$

But we have the new values from our previous iteration algorithm, so we can write

$$u_{i,j}^{new} = (1-L)u_{i,j} + L\left(\frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} + s^2}{4}\right)$$

The code is as follows:

```
clear variables
```

```
numterms=10;
kvec=1:numterms;
lam_k=1/2*(2*kvec-1)*pi;
umax_analytical=(1/8+1/2*sum((-
1).^kvec./(lam_k.^3.*cosh(lam_k))))

width=1;
ndiv=100; npts=ndiv+1;
centeri=round(ndiv/2+1); centerj=centeri;
s=width/ndiv;
x=linspace(0,width,npts); y=x;
L=1.5;      %overrelaxation value
u=zeros(npts,npts,'double');
upeakold=0;
numiters=4000;
relativechange=zeros(numiters,1);
for k=1:numiters
    for i=2:npts-1
        for j=2:npts-1
            u(i,j)=L*(u(i,j+1)+u(i,j-1)+u(i-
1,j)+u(i+1,j)+s^2)/4+(1-L)*u(i,j);
        end
    end
    %track relative change in peak value of w to track
convergence
    newwpeak=u(centeri,centerj);
    relativechange(k)=(abs(upeakold-newwpeak)/newwpeak);
    upeakold=newwpeak;
end
Maximum_value_overrelaxation=newwpeak
relerror=(umax_analytical-
Maximum_value_overrelaxation)./umax_analytical
semilogy(1:numiters,relativechange, 'LineWidth',2)
xlabel('iteration number'); ylabel('relative change in peak
value');
set(gca,'FontSize',10)
figure,mesh(x,y,u')
xlabel('X'); ylabel('Y');
figure,contour(x,y,u','ShowText','on');
xlabel('X');ylabel('Y')
```

For L=1.5, this code gives a relative error comparable to the script above with only about 3,400 iterations.

Python Solution

A python solution for a linear problem is as follows:

```
import math
import numpy as np
import matplotlib.pyplot as plt
numterms=10
umax_sum=0
for k in range(1, numterms):
    theta_k=(2*k-1)*math.pi/2
    umax_sum=umax_sum+((-1)**k/(theta_k**3*math.cosh(theta_k)))
umax_analytical=(1/8+umax_sum/2)
print('The maximum value of u (analytical) is ', umax_analytical)

width=1
ndiv=20
n=ndiv+1
h=width/ndiv
x=np.linspace(0,width,n)
y=np.linspace(0,width,n)
A=np.zeros((n*n,n*n))
b=np.zeros(n*n)
for i in range(1,n-1):
    for j in range(1,n-1):
        A[i+j*n,i-1+j*n]=1
        A[i+j*n,i+1+j*n]=1
        A[i+j*n,i+j*n]=-4
        A[i+j*n,i+(j-1)*n]=1
        A[i+j*n,i+(j+1)*n]=1
        b[i+j*n]=-h**2
for i in range(0,n):
    A[i,i]=1
    b[i]=0
    A[i+n*(n-1),i+n*(n-1)]=1
    b[i+n*(n-1)]=0
for j in range(1,n-1):
    A[j*n,j*n]=1
    b[j*n]=0
    A[n-1+j*n,n-1+j*n]=1
    b[n-1+j*n]=0
v=np.linalg.solve(A,b)
print('The maximum value of u (numerical) is ',max(v))
w=v.reshape(n,n)
plt.contour(x,y,w)
```

Validation of Elliptic Partial Differential Equations

In validating elliptic partial differential equations, we can still use the approaches discussed for parabolic partial differential equations:

- When you are new to a technique, test it on problems where the solution is known. Start by picking some easy problems for which the solution is known and make sure you can match the known results with your numerical solution.
- The most useful approach is something I think of as a global approach. That is, you should have some idea of the right answer prior to attempting the numerical solution.
- Always make sure your solution satisfies the initial conditions. This is generally easily checked with a plot of the solution.
- Convergence study
- Benchmark against known solutions
- Approximate full problem with something we can solve analytically
- Test experimentally

We’ve already solved some problems with known solutions and carried out some convergence tests. I’ll leave it to the reader to carry out further validation for their problems.

Appendix 1 Using Matlab

I make no attempt to describe all aspects of Matlab in this document. If you have never used Matlab, you might want to try another source first. For example, you could try the following online tutorial:

https://www.mathworks.com/support/learn-with-matlab-tutorials.html

There are many others.

Below I describe a few idiosyncrasies of Matlab. These might be helpful as you try to develop your Matlab skills.

Scripting in Matlab

Scripts are files, carrying a .m file name extension and often referred to as m-files, that allow us to store Matlab commands in a text file before executing them. To demonstrate how to use m-files, I'll step through a typical Matlab problem. Suppose we want to find the roots of the function f(x)=cos(x). In other words, we're looking for values of x such that cos(x)=0. First we plot the function to get an idea of where the solutions are. To do this, we can take the following steps.

First, we start Matlab. This will open up what Matlab calls the Command Window. Now we can begin typing in commands. Type the following:

```
x=0:0.1:10;
y=cos(x);
plot(x,y)
```

The first command makes a list of x values from 0 to 10 with an increment of 0.1 between each value. This gives us 101 values. The semicolon at the end tells Matlab to not list the values as it calculates them. The second command evaluates a value for cos(x) for each of the elements in the x list. So now we have 101 y-values. Then we plot y vs. x.

We're looking for roots of cos(x). These are x-locations where cos(x) is equal to 0. So in this case, we can see from the plot, that the function goes through 0 at about x=1.7, 4.5, and 8. Now we can use a Matlab function (fzero) to find the roots more accurately. Just type:

```
fzero(@cos,1.7)
```

We get 1.5708, which is the correct answer (it happens to be π/2). Now we can get the other two with:

```
fzero(@cos,4.5)
fzero(@cos,8)
```

These give us x=4.7124 and 7.8540, respectively.

Now, for preserving this work for the future, it's easier to put these commands into a script, which Matlab calls an m-file. To do this, go to the Matlab Command window and click on File/New and then choose M-File. This will bring up a text editor, which Matlab calls the Matlab Editor. In this file, just type or paste the following commands:

```
x=0:0.1:10;
y=cos(x);
plot(x,y)
fzero(@cos,1.7)
fzero(@cos,4.5)
fzero(@cos,8)
```

and then save the file as testfile.m.

To execute the commands, return to the Command window and type the name of the file (without the .m extension). So in this case we just type:

```
testfile
```

This will run the commands in testfile.m and give the results in the Command window. This is how m-files work. They are simply text files containing a series of Matlab commands.

The other kind of m-file that we are interested in defines user-defined functions. For instance, if we want to find the roots of f(x)=3sin(x)-2, we must create an m-file which defines this function. To do this, go to the Editor window and click on File/New. In this new window type:

```
function f=testfunc(x)
f=3*sin(x)-2;
```

Now save this file as testfunc.m. Notice how the name of the m-file must match the string in the first line of the function. Now we can plot this function and find a root using:

```
x=0:0.1:10;
y=testfunc(x);
plot(x,y)
fzero(@testfunc,1)
```

This finds a root of testfunc near x=1, and the value it returns is x=0.7297.

Matlab Arithmetic and Operators

The final Matlab topic deals with some peculiarities in the way Matlab does arithmetic. If I were to plot f(x)=x2 in Matlab, I might try the following:

```
x=0:0.1:1;
y=x^2;
```

```
??? Error using ==> ^ Matrix must be square.
```

This is a problem. The same thing happens if I try to multiply x by itself, though the error message is slightly different.

```
y=x*x;
```

```
??? Error using ==> * Inner matrix dimensions must agree.
```

As you can see, I get these strange errors about matrices. What's happening is that Matlab is taking the vector x and trying to square it. Due to the nature of vector arithmetic, vectors cannot be multiplied by themselves, so an error results. What we want to do here is square each element of the vector individually. To do this, you use the operators .^ and .*, as shown here:

```
y=x .* x
y =
  Columns 1 through 7
         0    0.0100    0.0400    0.0900    0.1600    0.2500    0.3600
  Columns 8 through 11
    0.4900    0.6400    0.8100    1.0000

y=x .^ 2
y =
  Columns 1 through 7
         0    0.0100    0.0400    0.0900    0.1600    0.2500    0.3600
  Columns 8 through 11
    0.4900    0.6400    0.8100    1.0000
```

The same thing happens with division:

```
z=1:0.1:2
z =
  Columns 1 through 7
    1.0000    1.1000    1.2000    1.3000    1.4000    1.5000    1.6000
  Columns 8 through 11
    1.7000    1.8000    1.9000    2.0000
x/z
ans =
    0.3617
```

This does matrix division and yields a single number, which is not expected. If you want to divide element by element, use the ./ operator:

```
x ./ z

ans =
  Columns 1 through 7
         0    0.0909    0.1667    0.2308    0.2857    0.3333    0.3750
  Columns 8 through 11
    0.4118    0.4444    0.4737    0.5000
```

This gives the first element of x divided by the first element of z, the second element of x divided by the second element of z, etc. The rule of thumb is that if you aren't expecting to do any vector or matrix math, you can never go wrong using .*, ./, or .^ instead of *, /, and ^.

User-Defined Functions

Matlab scripts are called m-files because they have a .m file extension. User-defined functions are saved as m-files, but they differ from the traditional Matlab script. A typical m-file is a sequence of Matlab commands that is executed, one command at a time, by the Matlab processor. A function is structured differently, and all have the same form

function [output] =function_name(input)

For example, we can create a function to evaluate x+2y as follows:

```
     function result=f(x,y)
        result=x+2*y
```

This function would be saved as the file f.m, so that the filename matches the function name in the file. The result variable appears in the first line and then must be defined somewhere within the function. The value of result when the function terminates is then returned to the calling routine. The result can actually consist of a list of values, rather than one value, be we won’t address that here.

The function listed above is just one line long, but these functions can be as long as you please. For example, a function that would add up the cubes of the integers 1 through N might look like this:

```
function myval=sumofcubes(N)
    ans = 0;
    for i =  1:N
         ans = ans + i ^ 3;
    end
    myval = ans;
end
```

To reduce the number of files produced, functions can be combined into a single m-file. However, these can only be called from within that file. As an example, consider the following function:

```
function y=f(x)
    z=g(x);
    y=x+x*z;
end
```

```
function g(x)
    g=sin(x)
end
```

If this is saved in a file called f.m, then we can evaluate it using something like f(3) from within the Matlab command window. However, we would not be able to evaluate g(3) from the command window because the function g(x) is only callable from the f(x) function in the same file. In this file, g(x) is referred to as a subfunction and such functions can only be called from within the same m-file.

The examples in this document will all use this embedded approach. That is, all examples can be run by pasting the code into a single m-file and then executing that file.

Anonymous functions

For simple, one-line functions, there is no need to create an m-file. For example, we can do the following:

```
g=@(x) cos(x)+cos(1.1*x)
x=0:0.01:100;
y=g(x);
plot(x,y)
```

The first line here defines a function g(x). The other lines use that function to generate a plot. If you need an additional parameter in the function, you can incorporate it as follows:

```
x=0:0.01:100;
delta=1.05
gg=@(x, delta) cos(x)+cos(delta*x)
y=gg(x, delta);
plot(x,y)
```

Appendix 2 Using Python

Python is a general purpose programming language that is used widely. It is an interpreted language, which means that the code you write is parsed and executed line by line. It is designed for rapid development.

For our purposes (problem solving), several companion libraries are available to assist with computation. A list of math functions available in Python can be found here

https://docs.python.org/3/library/math.html

In this document we will make frequent reference to the numpy library (http://www.numpy.org/), which is "the fundamental package for scientific computing with Python." This library contains functions for linear algebra, random sampling, statistics, FFTs, and many others.

We'll also use Scipy (https://www.scipy.org/), a library for scientific computing. It includes functions for symbolic math, plotting, quadrature, signal processing, optimization, and special functions.

To make use of these tools, you will need a developer environment. I use anaconda (https://www.anaconda.com/), which is a platform for using R and Python for data science applications. Within Anaconda, there is a tool called Spyder, which is an integrated development environment for Python. This is the tool I use for all of my Python work. Once I get the Anaconda Navigator started, I usually go to Spyder for writing code. In Spyder, you can enter code into the Editor and then click the run button near the top of the window to execute the code. Results will show up in the console on the lower right.

I make no attempt to describe all aspects of Python in this booklet. If you have never used Python, you might want to try another source first. For example, you could try the following online tutorials.

https://docs.python.org/3/tutorial/

https://www.w3schools.com/python/

https://www.tutorialspoint.com/python/index.htm

https://www.learnpython.org/

http://thepythonguru.com/

There are many others.

User-Defined Functions

Python allows you to create your own functions. These can be handy ways to avoid having to frequently repeat code that you use often. They can also allow you to organize your code in a more readable, and therefore maintainable, form. The standard structure of a user-defined function is as follows:

```
def nameOfFunc (arg1, arg2, arg3 ...):
    statement
    statement
    statement
    ....
```

```
   return;
```

Note that the first line (the declaration) ends with a colon and the other lines are indented. Both are critical. The “return” at the end is not required.

Here is sample code for a user-defined function, along with code to call the function.

```
def myNumFunc(x):
      A=1
      B=2
      z=A+B*x
      return z
y=myNumFunc(25.3)
print(y)
```

You can add default arguments as follows:

```
def myNumFunc(x=25.3):
      A=1
      B=2
      z=A+B*x
      return z
y1=myNumFunc()
y2=myNumFunc(2)
print(y1)
print(y2)
```

In the latter case, the first call uses the default value of x=25.3, while the second uses x=2.