Problem 5.6

HW07
ENSC-481
Paul Freihoefer

5.6.1

Given: A pendulum with the following hamiltonian:

$$H = \frac{p^2}{2} - \cos q$$

Determine:

1. Hamilton's Equations of this System
2. Integrate the system over a range of I.C. using the following methods:
   - Implicit Euler
   - AB 2
   - IRK (Given B-table)
3. Provide discussion on some questions.

Solution:

1. Hamilton's Equations:

$$\dot{p} = -\frac{\partial H}{\partial q} + \cancel{\emptyset}^{\,0}$$

No external forces or damping given

$$\dot{q} = \frac{\partial H}{\partial p}$$

$$\dot{p} = -(+\sin q) \qquad \dot{q} = p$$

$$\dot{p} = -\sin q$$

```julia
module ForwardEuler

using LinearAlgebra
export FEuler

function FEuler(f,tf,h,x0)

    #FEuler Equation:
    #x(i+1) = x(i) + h*f(i)

    time = 0:h:tf
    n = length(time)
    p = length(x0)
    x = zeros(n,p)

    #Initial Condition
    x[1,:] .= x0

    for i = 1:n-1
            x[i+1,:] = x[i,:] + h*f(x[i,:], time[i])
    end

    return x, time

end
end
```

```julia
module AdamsBashforth2

using LinearAlgebra
export AB2

function AB2(f,tf, h, x0)

    #AB2 Method (Which is adapted to match where i starts):
    #AB2: x[i+1,:] = x[i,:] + h/2*(3*f(x[i,:],time[i]) - f(x[i-1,:],time[i-1]))

    time = 0:h:tf
    n = length(time)
    p = length(x0)
    x = zeros(n,p)

    #Initial Condition
    x[1,:] .= x0

    #The Adams Bashforth method requires that we know x(i+1) and then we can
    #continue the method without having an implicit equations. We can use the
    #forward euler method to estimate this value and then proceed with the AB2
    #equation.

    #Forward Euler
    fn = f(x[1,:],time[1])
    x[2,:] = x[1,:] + h*fn

    for i = 2:n-1
        fn_m1 = f(x[i-1,:], time[i-1]) #Just rewriting the above for
        fn = f(x[i,:],time[i-1])
        x[i+1,:] = x[i,:] + h/2*(3*fn-fn_m1)
    end

    return x, time
end
end
```

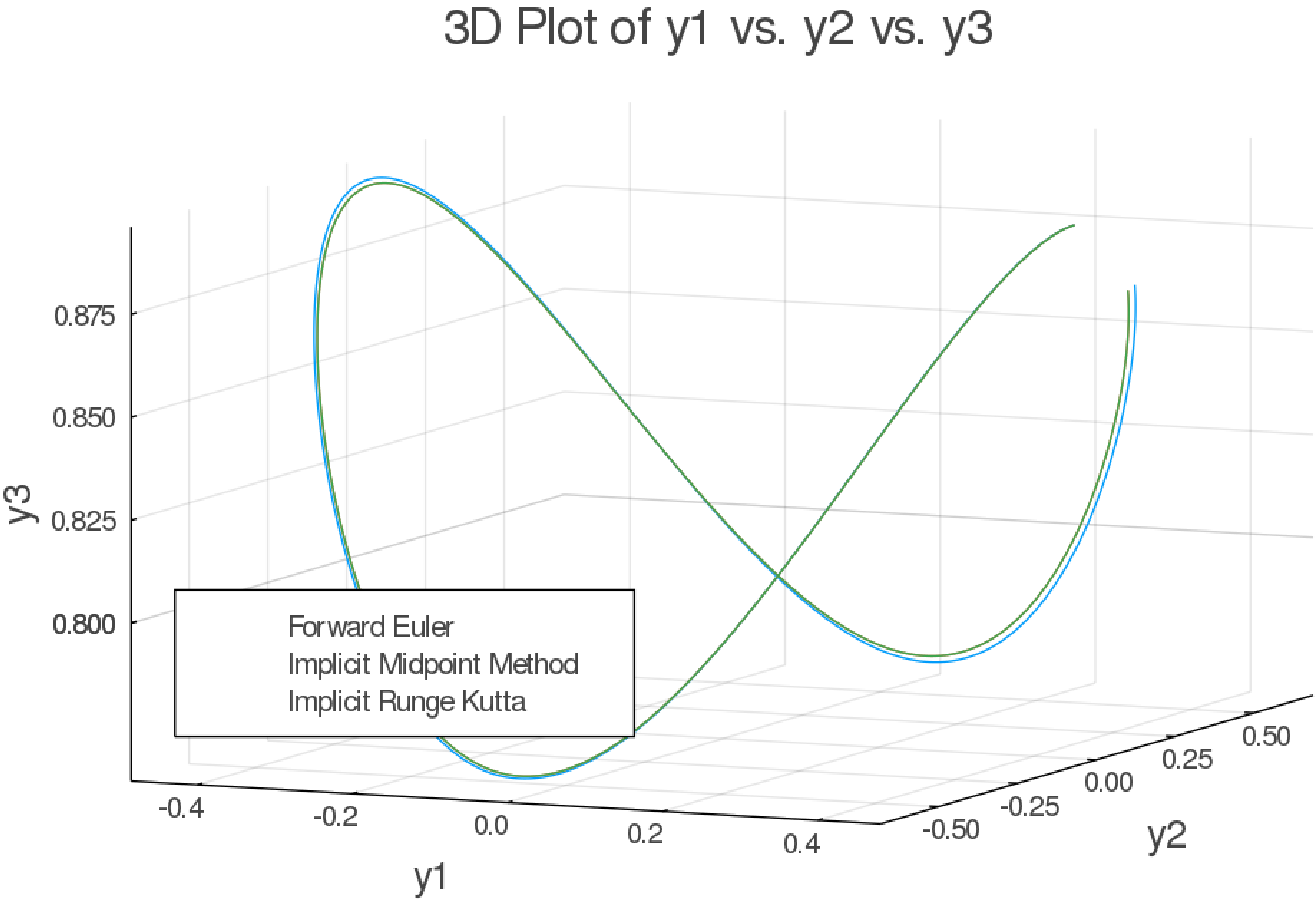
```julia
module ImplicitMidpoint

using LinearAlgebra

export IM

function IM(f,tf,h,x0; tol = 1e-4, iterMax = 200)

    #Implicit Midpoint Equation:
    #x(i+1) = x(i) + h*f(1/2*(x(i)+x(i+1)),t+h/2)

    time = 0:h:tf
    n = length(time)
    p = length(x0)
    x = zeros(n,p)

    #Initial Condition
    x[1,:] .= x0

    for i = 1:n-1
        #Because this is an implicit method we need to use an explicit method to
        #acquire an estimate for the first i+1 term. We will use a fixed point
        #iteration method to solve this: "Forward Euler".
        #x(i+1) = x(i) + h*f(x(i),t(i))

        #This gives us our first x(i+1)
        y = x[i,:] + h*f(x[i,:],time[i,:])

        flag = 0
        iter = 0
        while flag == 0
            iter += 1
            y = x[i,:] + h*f(1/2*(x[i,:]+y), time[i]+h/2)

            #Calculating the residual so we know when it "converges"
            resid = norm(y - x[i,:] - h*f(1/2*(x[i,:]+y), time[i]+h/2))

            if resid <= tol
                flag = 1
            elseif iterMax <= iter
                flag = -1
                error("Error: Method failed to converge")
            end

        x[i+1,:] = y
        end

    end

        return x, time
end

end
```

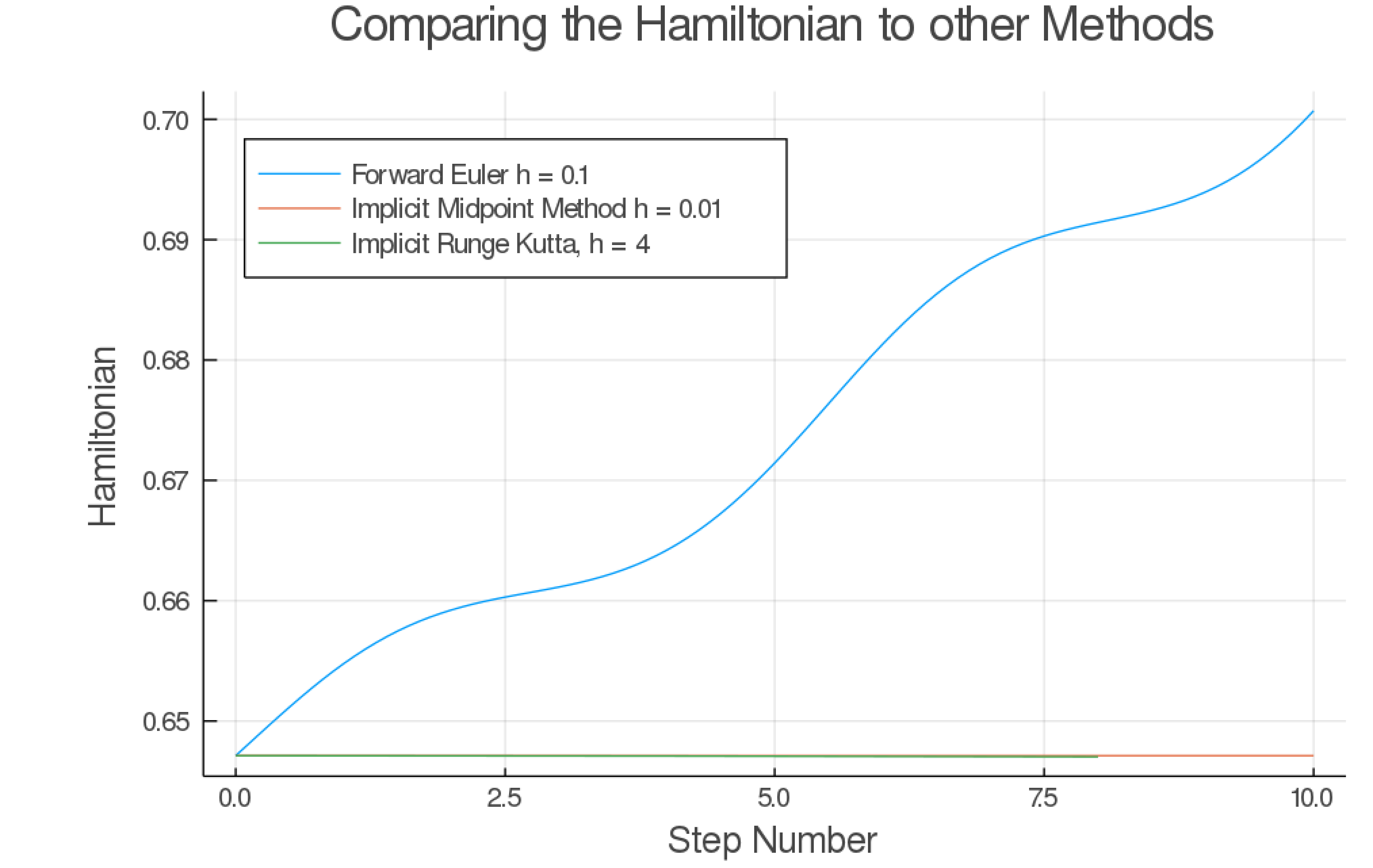
```julia
module ImplicitRungeKutta

using LinearAlgebra
export IRK

function IRK(f,tf,h,x0; tol =1e-4, iterMax = 200)

    #Implicit Runge Kutta Equation:
    #Derived from the butcher table and these conditions
    #x(i+1) = x(i) + h*∑bᵢkᵢ, b is a value from the butcher table and k is:
    #kᵢ = f(x(i)+h*∑aᵢⱼkᵢ,tₙ+c₁*h)
    #Oof... Let's get into that butcher table given in problem 5.6, fig (5.10)

    #From the butcher table
    c1 = 1/2-sqrt(3)/6
    c2 = 1/2+sqrt(3)/6
    a11 = 1/4
    a12 = 1/4-sqrt(3)/6
    a21 = 1/4+sqrt(3)/6
    a22 = 1/4
    b1 = 1/2
    b2 = 1/2

    #Now we can write out k1 and k2
    #k1 = f(x(i)+h*(a11*k1+a12*k2),time(i)+c1*h)
    #k2 = f(x(i)+h*(a21*k1+a22*k2),time(i)+c2*h)

    #Finally our IRK Equation looks like the following:
    #x(i+1) = x(i) + h*(b1*k1+b2*k2), looks so nice in this form...

    time = 0:h:tf
    n = length(time)
    p = length(x0)
    x = zeros(n,p)

    #Initial Condition
    x[1,:] .= x0

    #We need to make a guess for k1/k2 and we have to simaltaneously find them
    #because they are present in both of their respective equations. The easiest
    #way to do this (as in the way I understand how to do this) is using fixed
    #point iteration and getting the two values to converge.
```

```
    #We start with a guess for k1 & k2
    k1 = zeros(p)
    k2 = zeros(p)
    k1N = zeros(p)
    k2N = zeros(p)

    for i = 1:n-1
        iter = 0
        flag = 0
        while flag == 0
            iter += 1
            k1N[:] = f(x[i,:]+h*(a11*k1+a12*k2),time[i]+c1*h)
            k2N[:] = f(x[i,:]+h*(a21*k1+a22*k2),time[i]+c1*h)

            #We need to evaluate two different residuals
            residk1 = norm(k1-k1N)
            residk2 = norm(k2-k2N)
            k1[:] = k1N
            k2[:] = k2N

            if residk1 <= tol && residk2 <= tol
                flag = 1
            elseif iterMax <= iter
                flag = -1
                error("Error: System failed to converge")
            end
        end

        #Now we have values for k1 and k2 that converged at this x(i) we will
        #use these estimates for each consequetive iteration in the hopes that
        #k1 and k2 will converge faster.

        x[i+1,:] = x[i,:] + h*(b1*k1+b2*k2)
    end

    return x, time

end
end
```

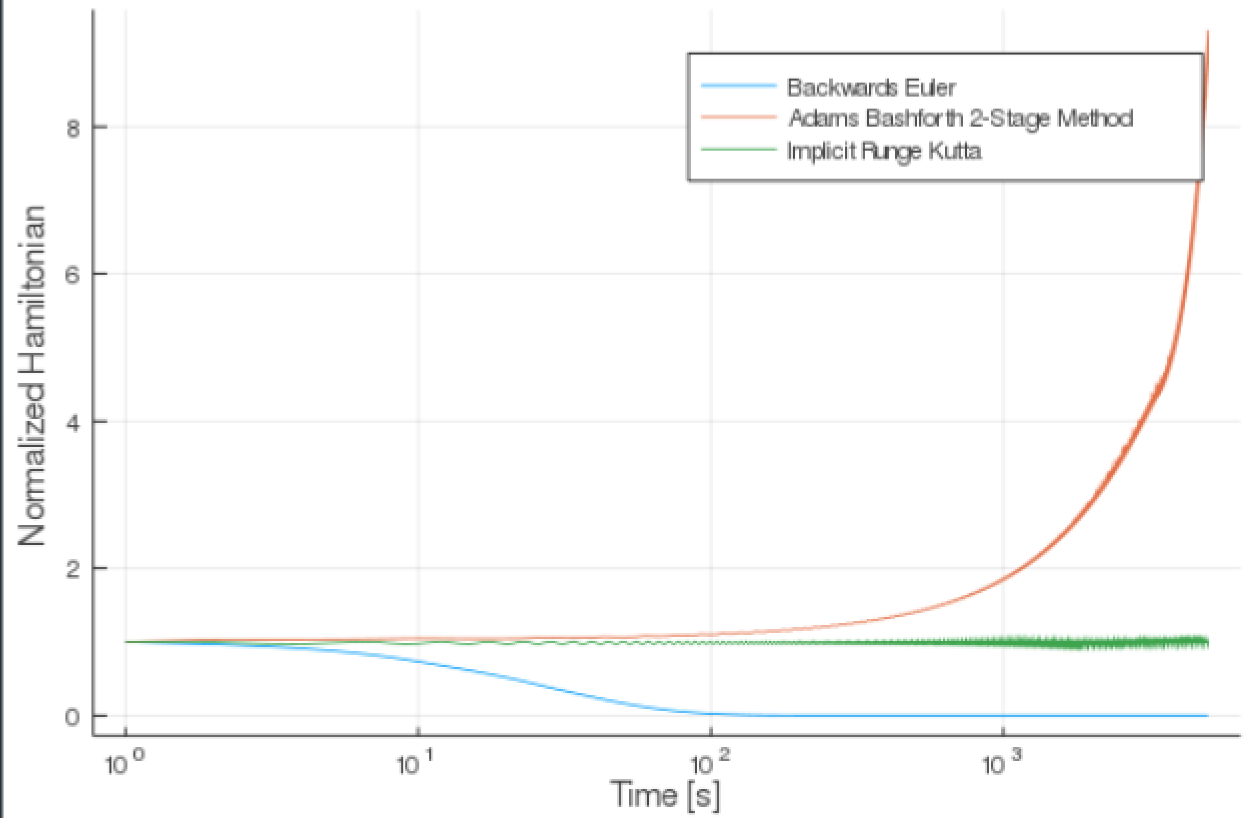
```
##Establishing the Pkg
using Pkg  ✔
Pkg.activate(".")  ✔

using Plots  ✔
using LinearAlgebra  ✔
##Loading our various functions
include("C:/Users/paulf/Documents/GitHub/Computational-Dynamics/ForwardEuler.jl")  Main
using .ForwardEuler  ✔

include("C:/Users/paulf/Documents/GitHub/Computational-Dynamics/IM.jl")  Main.ImplicitM
using .ImplicitMidpoint  ✔

include("C:/Users/paulf/Documents/GitHub/Computational-Dynamics/ImplicitRungeKutta.jl")
using .ImplicitRungeKutta  ✔


##We are now considering a new hamiltonian and with new ODEs
#We are given the following:
i1 = 2  2
i2 = 1  1
i3 = 2/3  0.667…
y0 = [cos(1.1),0,sin(1.1)]  > Vector{Float64} with 3 elements

#We know these are the following ODEs
#y1Dot = a1*y2*y3
#y2Dot = a2*y3*y1
#y3Dot = a3*y1*y2
a1 = (i2-i3)/(i2*i3)  0.500…
a2 = (i3-i1)/(i3*i1)  -1.00…
a3 = (i1-i2)/(i1*i2)  0.500

#Now for our function
f(y,t) = [a1*y[2]*y[3],a2*y[3]*y[1],a3*y[1]*y[2]]  > f
h = 0.01 #Time step  0.0100
tf = 10 #Final time  10
```

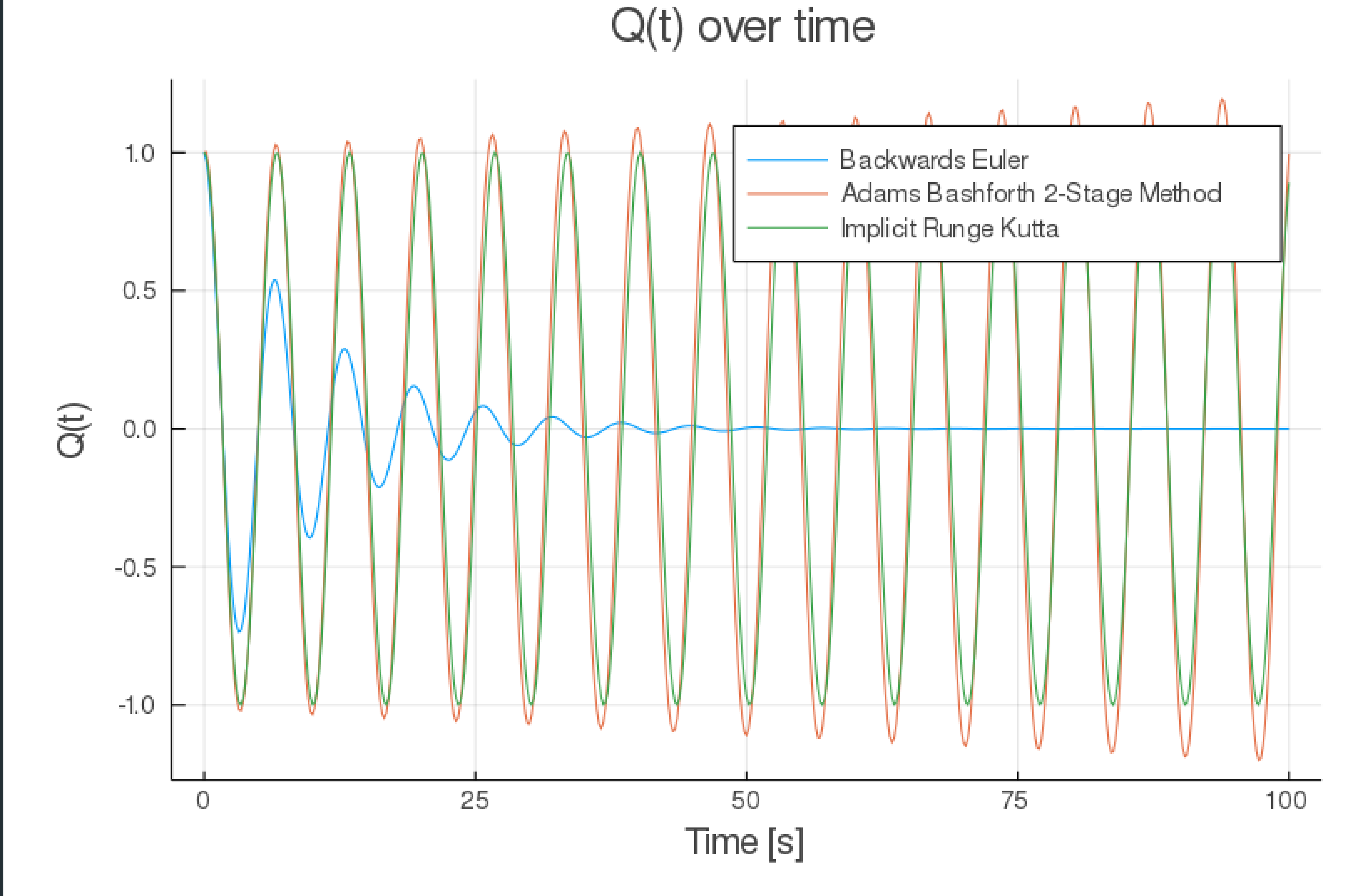
```julia

#Now for our function
f(y,t) = [a1*y[2]*y[3],a2*y[3]*y[1],a3*y[1]*y[2]]   > f
h = 0.01 #Time step   0.0100
tf = 10 #Final time   10
##Gathering our solutions from each method
#Forward Euler
xFE, tFE = FEuler(f,tf,0.1,y0)   (> 101×3 Array{Float64,2}:, 0.0:0.1:10.0)
#Implicit Midpoint Method
xIM, tIM = IM(f,tf,h,y0)   (> 1001×3 Array{Float64,2}:, 0.0:0.01:10.0)
#Implicit Runge Kutta
xIRK, tIRK = IRK(f,tf,4,y0)   (> 3×3 Array{Float64,2}:, 0:4:8)
##Hamiltonian
hFE = 1/2 .*(xFE[:,1].^2 ./i1 + xFE[:,2].^2 ./i2 + xFE[:,3].^2 ./i3)   > Vector{Float64} with 101 elements
hIM = 1/2 .*(xIM[:,1].^2 ./i1 + xIM[:,2].^2 ./i2 + xIM[:,3].^2 ./i3)   > Vector{Float64} with 1001 elements
hIRK = 1/2 .*(xIRK[:,1].^2 ./i1 + xIRK[:,2].^2 ./i2 + xIRK[:,3].^2 ./i3)   > Vector{Float64} with 3 elements
##Plotting Hamiltonians
plot(tFE, hFE, label = "Forward Euler h = 0.1", xlabel = "Step Number", ylabel = "Hamiltonian", title = "Compa
plot!(tIM, hIM, label = "Implicit Midpoint Method h = 0.01", legend =:topleft)   Plot{Plots.GRBackend() n=2}
plot!(tIRK, hIRK, label = "Implicit Runge Kutta, h = 4")   Plot{Plots.GRBackend() n=3}
#It looks like the Forward Euler method doesn't remain on H and start to grow
#over time. In the 3D representation it looks like it matches the correct
#trajectory but is obviously inaccurate enough to start to diverge from H.
#Regardless of how much we alter the conditions of IRK it stays incredibly
#accurate and well-behaved. Making it an incredibly reliable method at high step
#sizes and long time intervals. Just a very costly one.


##Plotting our Solutions
plot(xFE[:,1],xFE[:,2],xFE[:,3], label = "Forward Euler", legend=:bottomleft, xlabel="y1", ylabel = "y2", zlab
plot!(xIM[:,1],xIM[:,2],xIM[:,3], label = "Implicit Midpoint Method")   Plot{Plots.GRBackend() n=2}
plot!(xIRK[:,1],xIRK[:,2],xIRK[:,3], label = "Implicit Runge Kutta")   Plot{Plots.GRBackend() n=3}
```

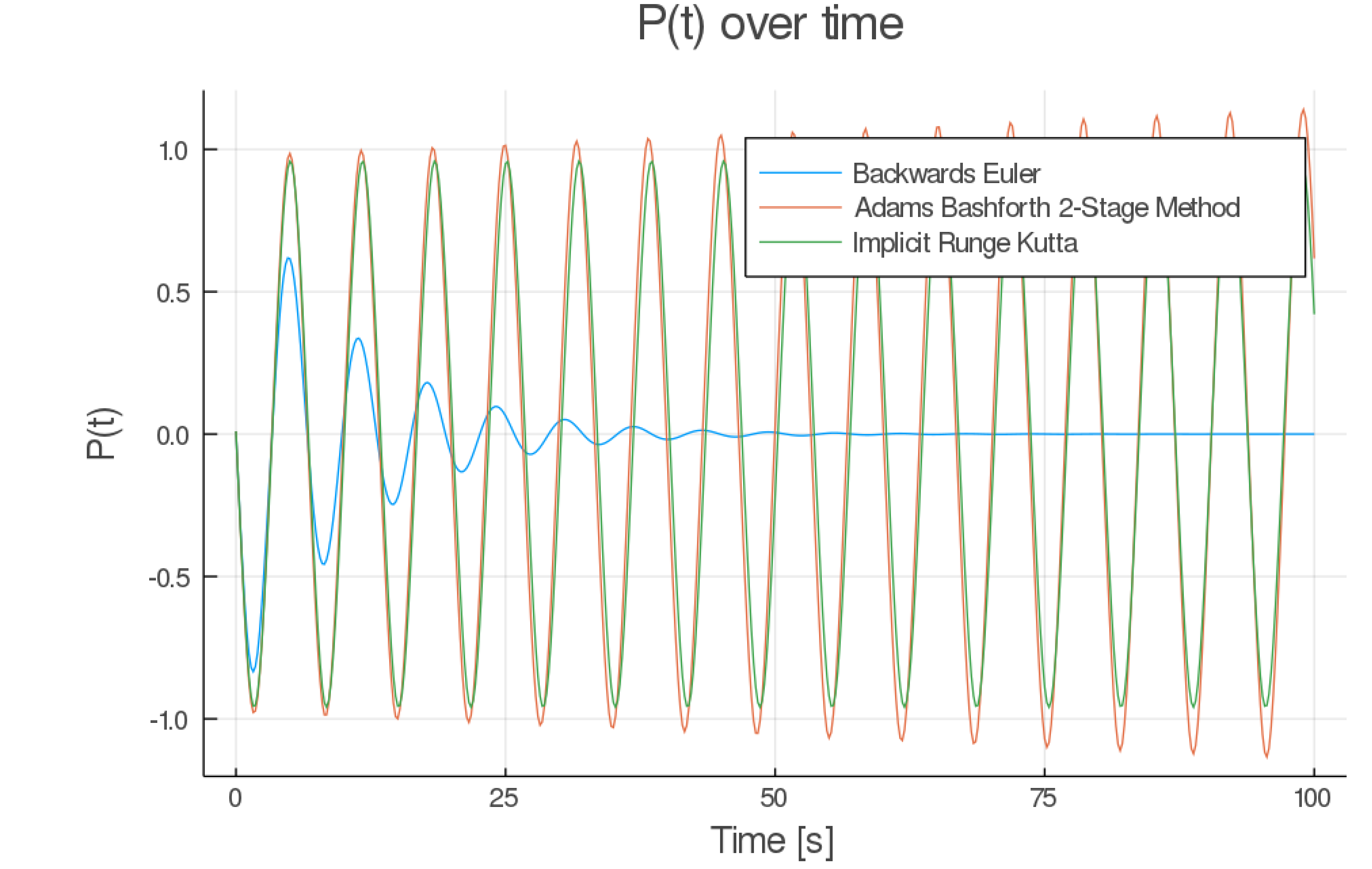

3D Plot of y1 vs. y2 vs. y3
0.875
0.850
0.825
0.800
y3
Forward Euler
Implicit Midpoint Method
Implicit Runge Kutta
-0.4
-0.2
0.0
0.2
0.4
y1
-0.50
-0.25
0.00
0.25
0.50
y2



Comparing the Hamiltonian to other Methods
Forward Euler h = 0.1
Implicit Midpoint Method h = 0.01
Implicit Runge Kutta, h = 4
Hamiltonian
0.70
0.69
0.68
0.67
0.66
0.65
0.0
2.5
5.0
7.5
10.0
Step Number


Backwards Euler
Adams Bashforth 2-Stage Method
Implicit Runge Kutta
Normalized Hamiltonian
Time [s]
0
2
4
6
8
10^0
10^1
10^2
10^3


Q(t) over time
Backwards Euler
Adams Bashforth 2-Stage Method
Implicit Runge Kutta
1.0
0.5
0.0
-0.5
-1.0
Q(t)
0
25
50
75
100
Time [s]



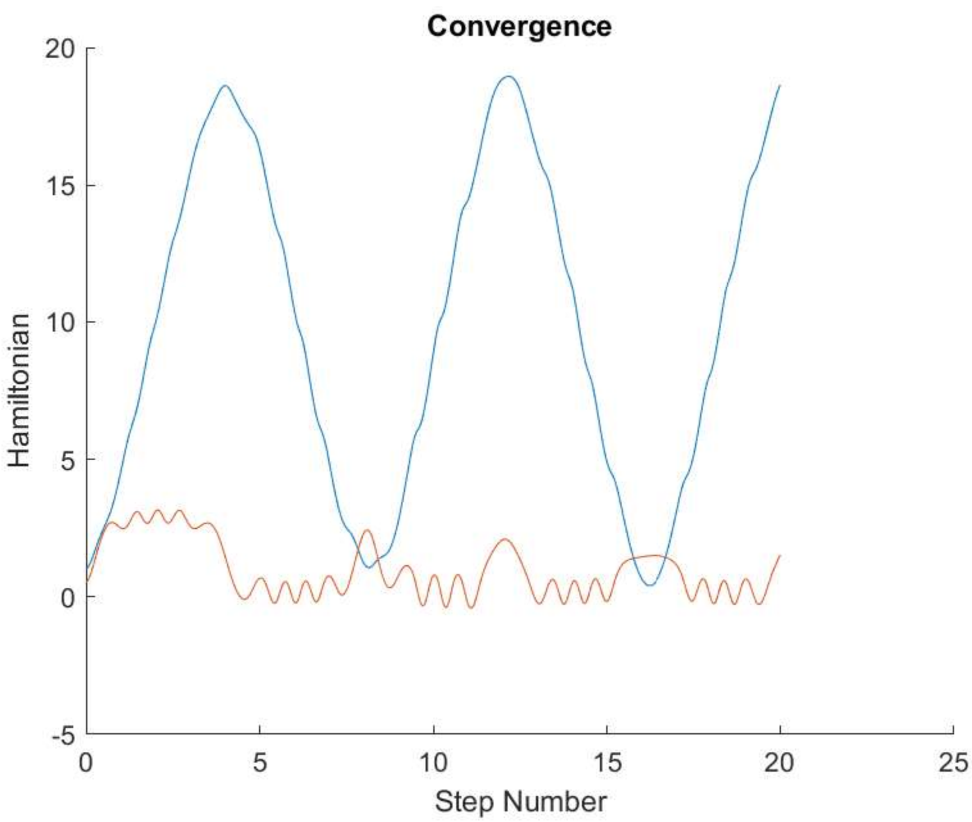
P(t) over time
Backwards Euler
Adams Bashforth 2-Stage Method
Implicit Runge Kutta
1.0
0.5
0.0
-0.5
-1.0
P(t)
0
25
50
75
100
Time [s]

```julia
##Problem 5_6_1 Overview
#Plan: We need to construct three different methods to integrate the following
#equations:
#pDot = -sin(q), f'(t,q) = f(t,q) Another way to imagine these equations
#qDot = p, f'(t,p) = f(t,p)
#Using any of these first order integrators will give us solutions to our
#pendulum's motion. It's also important to remember that q represents the angle
#of the pendulum at any given moment and p is the non-dimensionalized momentum.
##Establishing our packages
using Pkg ✓
Pkg.activate(".") ✓
#Pkg.add("BenchmarkTools")

using Plots ✓
using LinearAlgebra ✓
using BenchmarkTools ✓
##Loading our various functions
include("C:/Users/paulf/Documents/GitHub/Computational-Dynamics/BackwardEuler.jl")   Main.BackwardEuler
using .BackwardEuler ✓

include("C:/Users/paulf/Documents/GitHub/Computational-Dynamics/AdamsBashforth2.jl")   Main.AdamsBashforth
using .AdamsBashforth2 ✓

include("C:/Users/paulf/Documents/GitHub/Computational-Dynamics/ImplicitRungeKutta.jl")   Main.ImplicitRung
using .ImplicitRungeKutta ✓
##Setting up the Problem
f(x,t) = [x[2],-sin(x[1])]   > f
x0 = [1,0.01] #Initial guess   > Vector{Float64} with 2 elements
h = 0.2 #Time step   0.200
tf = 100 #Final time   100
##Gathering our solutions from each method
#Backwards Euler
xBE, tBE = BEuler(f,tf,h,x0)   (> 501×2 Array{Float64,2}:, 0.0:0.2:100.0)
#Adams Bashforth 2-Stage Method
xAB2, tAB2 = AB2(f,tf,h,x0)   (> 501×2 Array{Float64,2}:, 0.0:0.2:100.0)
#Implicit Runge Kutta
xIRK, tIRK = IRK(f,tf,h,x0)   (> 501×2 Array{Float64,2}:, 0.0:0.2:100.0)
##Comparing the solutions to q
plot(tBE, xBE[:,1], label="Backwards Euler", xlabel = "Time [s]", ylabel = "Q(t)", title="Q(t) over time")
plot!(tAB2, xAB2[:,1], label = "Adams Bashforth 2-Stage Method")   Plot{Plots.GRBackend() n=2}
```

```
plot!(tIRK,xIRK[:,1], label = "Implicit Runge Kutta")  Plot{Plots.GRBackend() n=3}

##Comparing the solutions to p
plot(tBE, xBE[:,2], label="Backwards Euler", xlabel = "Time [s]", ylabel = "P(t)", title="P(t) over time")  P
plot!(tAB2, xAB2[:,2], label = "Adams Bashforth 2-Stage Method")  Plot{Plots.GRBackend() n=2}
plot!(tIRK,xIRK[:,2], label = "Implicit Runge Kutta")  Plot{Plots.GRBackend() n=3}

##Hamiltonian
#h = 0.2, tf = 1000s
nHBE = (xBE[:,2].^2/2 - cos.(xBE[:,1]).+1)/(x0[2]^2/2-cos(x0[1])+1)  > Vector{Float64} with 501 elements
#h = 0.2, tf = 1000s
nHAB2 = (xAB2[:,2].^2 ./2-cos.(xAB2[:,1]).+1)./(x0[2]^2/2-cos(x0[1])+1)  > Vector{Float64} with 5001 elements
#h = 2, tf = 10000s
nHIRK = (xIRK[:,2].^2 ./2-cos.(xIRK[:,1]).+1)./(x0[2]^2/2-cos(x0[1])+1)  > Vector{Float64} with 5001 elements

plot(nHBE, label="Backwards Euler", xlabel = "Step Number", ylabel = "Normalized Hamiltonian", xscale = :log)
plot!(nHAB2, label = "Adams Bashforth 2-Stage Method")  Plot{Plots.GRBackend() n=2}
plot!(nHIRK, label = "Implicit Runge Kutta")  Plot{Plots.GRBackend() n=3}
##Answering Part A if part 3:
#From the energy plot we derive we can see that over a long period of
#time the backwards euler method starts to fall apart and is completely
#off the path. Our Adams Bashforth 2-Stage method also eventually starts
#to break away but not as quickly as the BEuler method does. Finally,
#and the most interesting thing to note is that our IRK method stays
#within a stable energy range even when put under intense conditions
#(high step size and large final time). The method oscillates and stays
#within the correct range of energy for a lot of steps. Obviously this
#method just kicks way more ass than the other two. It's like going to
#Whole foods over Cash and Carry.
```

```
##Benchmarking Each Method
#Backwards Euler (evaluated for P with tf = 100s)
#Mean time = 7.400 ms
#AB2
#Mean Time = 4.263 ms
#IRk
#Mean Time = 18.998 ms
#At tf = 25s
#BEuler: 1.785 ms
#AB2: 1.059 ms
#IRK: 4.701 ms
#At tf = 10s
#BEuler: 721.662 μs
#AB2: 422.611 μs
#IRK: 1.906 ms
##Answering Part B of 5.6.1.3:
#At varying final times we can see how each method does compared to one
#another. Our most complex method (IRK) is clearly the most expensive and
#always takes considerably longer than the other two. However, we see
#that Adam Bashforths 2-Stage Method is very quick for how complex it is.
#Although we do need to consider the fact that our Backwards Euler uses
#a while loop to make sure that the first estimate converges. This
#probably adds onto the time and means that our AB2 and BEuler method are
#probably quite similar in length. In general it is probably safe to say
#that as long as you are not adapting the time step for a problem less
#complex methods will take less time to compute compared to more complex
#methods over the same range of times. If we are considering an adaptive
#time step then we might see "slower" methods that are complex become
#quite fast because they are more accurate with less iterations.
##Answering Part C:
#As discussed in Part A we can see that the length in time or number of
#step sizes does not cause the IRK method to break away from our
#Hamiltonian. In fact, the method stays well behaved even under extreme
#conditions. The real downside though is the computing cost for an IRK
#method. It is incredibly expensive to run which makes it very hard to
#justify compared to Euler's Methods with small step sizes. Which can be
#just as accurate as IRK but cost a fraction of the computing power.
```

```julia
module BackwardEuler

using LinearAlgebra
export BEuler

function BEuler(f,tf,h,x0; tol = 1e-4, iterMax = 300)

    #BEuler Equation:
    #x(i+1) = x(i) + h*f(i+1)

    time = 0:h:tf
    n = length(time)
    p = length(x0)
    x = zeros(n,p)

    #Initial Condition
    x[1,:] .= x0

    for i = 1:n-1
        #Because this is an implicit method we need to use an explicit method to
        #acquire an estimate for the first i+1 term. We will use a fixed point
        #iteration method to solve this: "Forward Euler".
        #x(i+1) = x(i) + h*f(x(i),t(i))

        #This gives us our first x(i+1) which we can now use in the Backward
        #Euler method.
        y = x[i,:] + h*f(x[i,:],time[i,:])

        flag = 0
        iter = 0
        while flag == 0
            iter += 1
            y = x[i,:] + h*f(y, time[i+1])

            #Calculating the residual so we know when it "converges"
            resid = norm(y - x[i,:] - h*f(y, time[i+1]))

            if resid <= tol
                flag = 1
            elseif iterMax <= iter
                flag = -1
                error("Error: Method failed to converge")
            end

        x[i+1,:] = y
        end

    end

        return x, time

end
end
```


Convergence
Hamiltonian
Step Number
20
15
10
5
0
-5
0
5
10
15
20
25

```
clear,
clc
syms m y_dot theta_dot L k k_t J_G theta y g t y_ddot
%theta_dot = diff(theta, t)
%y_dot = diff(y,t)

K_r = 1/2*J_G*theta_dot^2
```

K_r =

$$\frac{J_G\,\dot{\theta}^2}{2}$$

```
K_l = 1/2*m*(y_dot^2 + (theta_dot*L)^2 + 2*y_dot*theta_dot*L*cos(theta))
```

K_l =

$$\frac{m\,\left(L^2\,\dot{\theta}^2+2\cos(\theta)\,L\,\dot{\theta}\,\dot{y}+\dot{y}^2\right)}{2}$$

```
U_s = 1/2*k*y^2 +1/2*k_t*theta^2
```

U_s =

$$\frac{k_t\,\theta^2}{2}+\frac{k\,y^2}{2}$$

```
U_g = -m*g*(y+L*sin(theta))
```

U_g = $-g\,m\,\left(y+L\sin(\theta)\right)$

```
K= K_r + K_l;
U = U_s+U_g;
Lag = K-U
```

Lag =

$$\frac{m\,\left(L^2\,\dot{\theta}^2+2\cos(\theta)\,L\,\dot{\theta}\,\dot{y}+\dot{y}^2\right)}{2}+\frac{J_G\,\dot{\theta}^2}{2}-\frac{k_t\,\theta^2}{2}-\frac{k\,y^2}{2}+g\,m\,\left(y+L\sin(\theta)\right)$$

```
%Eqn1 = diff(Lag,y) == diff(Lag,y_dot)%make dots -> double dots
%subs(Eqn1, y_dot, y_dot(t))
%Eqn2 = diff(Lag,theta) ==diff(Lag,theta_dot)

H = K+U
```

H =

$$\frac{m\,\left(L^2\,\dot{\theta}^2+2\cos(\theta)\,L\,\dot{\theta}\,\dot{y}+\dot{y}^2\right)}{2}+\frac{J_G\,\dot{\theta}^2}{2}+\frac{k_t\,\theta^2}{2}+\frac{k\,y^2}{2}-g\,m\,\left(y+L\sin(\theta)\right)$$

```
Ptheta_dot = -diff(H,theta)
```

Ptheta_dot = $L\,g\,m\cos(\theta)-k_t\,\theta+L\,m\,\dot{\theta}\,\dot{y}\sin(\theta)$

```
Pydot = -diff(H,y)
```

Pydot = $g\,m-k\,y$

```
syms P_y P_theta
H = ((m*L^2*(P_theta/J_G)^2 + 2*cos(theta)*L*P_theta/J_G*P_y + (P_y/m)^2))/2 + (P_theta^2)/(2*J_G) + (k_t*theta^sym(2))/2 + (k*y^2)/2 - g*m
diff(H,P_y)
```

ans =

$$\frac{P_y}{m^2}+\frac{L\,P_\theta\cos(\theta)}{J_G}$$

```
diff(H,P_theta)
```

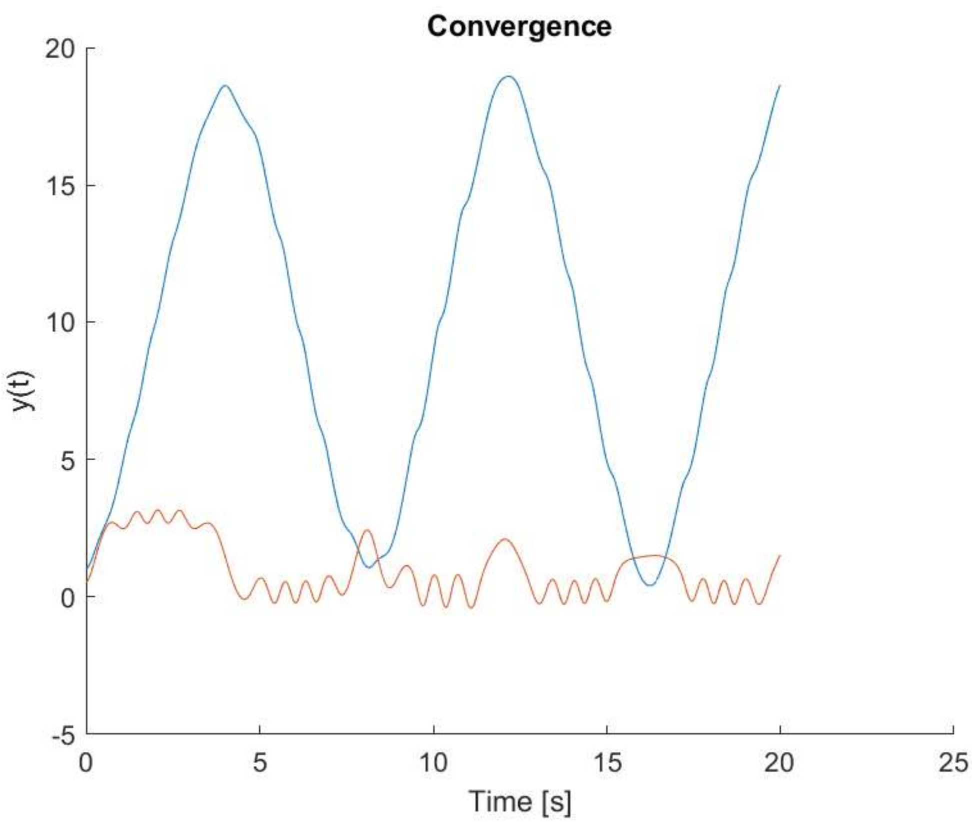
ans =

$$\frac{P_\theta}{J_G}+\frac{L^2\,P_\theta\,m}{{J_G}^2}+\frac{L\,P_y\cos(\theta)}{J_G}$$

```
%% Section 5.7.2
clear
clc
%%
%Define System Constants:
m = 1; %kg
J_G = 1; %kg*m^2
L = 1;%m
g = 9.81;%m/s^2
k = 1; %N*m
k_t = 1; %N*m
%%
%Define Solver Values:
tspan = [0, 20]; %s
y0 = [1,pi/6,1,0.01];% [y0, theta0 [rad], Py, Ptheta]
tol = 10^(-2);

%EoM:
%z_dot = [dy, dtheta, py, ptheta]
f = @(t,y) [y(3)./(m^2) + L.*y(4)./J_G.*cos(y(2)),
            y(4)./J_G + m.*L.*L.*y(4)./J_G.^2 + L.*y(3).*cos(y(2))./J_G,
            m.*g-k.*y(1),
            m.*g.*L.*cos(y(2)) - k_t.*y(2) + m.*L.*y(4)./J_G.*y(3)./m.*sin(y(2))]'
tic
S = myOde23(f,tspan,y0,tol);
toc
% S.y = [y,theta,py,ptheta]
hold on
plot(S.t,S.y, 'o')
xlabel('Step Number')
ylabel('Hamiltonian')
title('Convergence')
%% Perform A Convergence Study
n =3; %Exponent for the smallest tolerance (Ex: n = 3 -> tol = 10^-3)
for i = 1:n
    tol = 10^(-i);
    C{i} = myOde23(f,tspan,y0,tol); %C for cell
    y = C{i}.y;
    t = C{i}.t;
    hold on
    plot(t,y(:,1),t,y(:,2))
    xlabel('Step Number')
    ylabel('Hamiltonian')
    title('Convergence')
    %labels{i} = sprintf('tol = 10^%.f',i);
end
%legend(labels)
hold off
```

```
function S = myOde23(ODEFUN, TSPAN, Y0,TOL)

close all

tspan = TSPAN;  % set the interval of t
y(1,:) = Y0;    % set the intial value for y
t(1) = tspan(1);

if ~exist('TOL','var')
    tol = 10^(-8); %Default tolerance value
else
    tol = TOL;
end

%{
(This bit of the code doesn't work.)
if ~exist('ODEFUN','var')
    %f =@(t,y)[sin(y(1)), y(2)] ; %insert function to be solved,
else
    f = ODEFUN;
end
%}

f = ODEFUN;


h = 0.5*tol; % set the step size


a = [0,   0,      0;
    0.5, 0.75,   0;
    0,   0.75,   0;
    2/9, 1/3, 4/9];

b = [2/9, 1/3, 4/9, 0];
b2= [7/24, 1/4, 1/3, 1/8];
c = [0, 1/2,   3/4, 1];


i = 1;
while t(i) < tspan(2)

    k1 = f(t(i),y(i,:));
    k2 = f(t(i)+c(2).*h(i),y(i,:)+a(2,1).*k1.*h(i));
    k3 = f(t(i)+c(3).*h(i),y(i,:)+a(3,1).*k1.*h(i)+a(3,2).*k1.*h(i));
   y(i+1,:) = y(i,:) + h(i).*(b(1).*k1 + b(2).*k2 + b(3).*k3);
    k4 = f(t(i)+c(4).*h(i),y(i,:)+a(4,1).*k1.*h(i)+a(4,2).*k1.*h(i)+a(4,3).*k3*h(i));
   z(i+1,:) = y(i,:) + h(i).*(b2(1).*k1 + b2(2).*k2 + b2(3).*k3 + b2(4).*k4);

    e(i+1,:) = abs(y(i+1,:)-z(i+1,:));
```

```
    if max(e(i+1,:)) > tol
        h(i) = 0.5*h(i);
        t(i+1) = t(i)+h(i);
    elseif max(e(i+1,:)) <= tol
        h(i+1) = 0.9*h(i)*min(max(tol./max(e(i+1,:)),0.3),2);
        t(i+1) = t(i)+h(i);
        i = i+1;
    else
        fprintf('\nERROR\n\n')
        break
    end
end

S.t = t;
S.y = y;
S.e = e;
S.z = z;
S.h = h;
end
```


Convergence
20
15
10
5
0
-5
y(t)
0
5
10
15
20
25
Time [s]