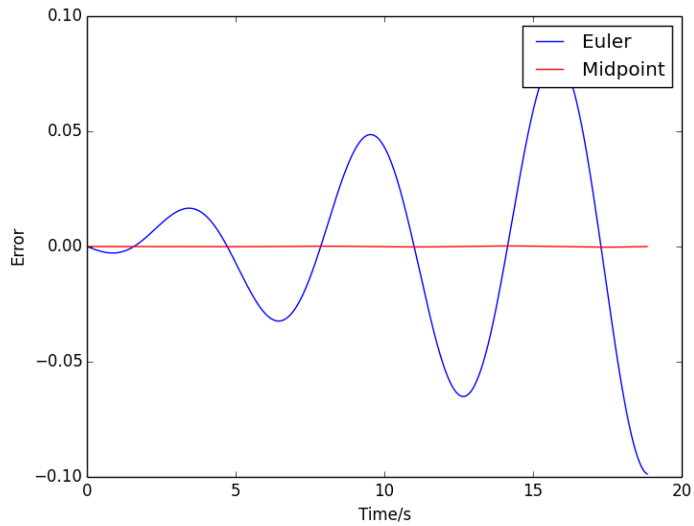
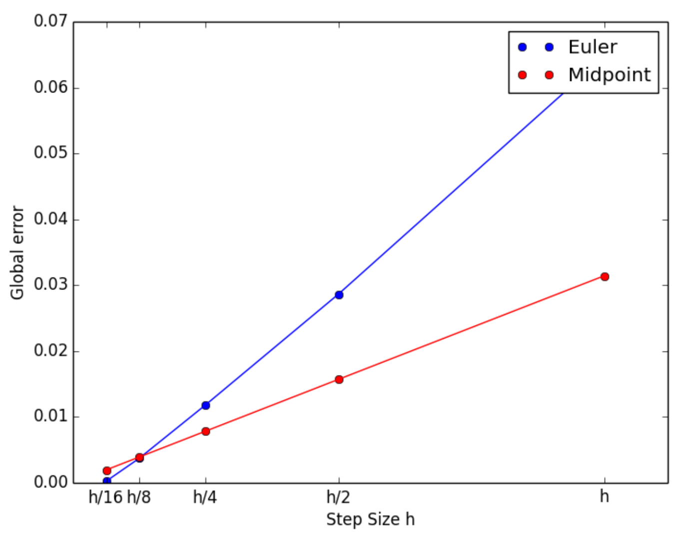
Ph22.1 Return of the ODEs: Higher-order methods

Lim Soon Wei Daniel

2. We consider the system $y'' + y = 0$ and solve the ODE over three cycles using both the explicit Euler method and the midpoint method. The residues are plotted below for step size 0.01.



3. We integrate the total error using both methods and compare its dependence on the step size. The initial step size here is $h = 0.01$.



5. We define the following variables:

$$\begin{aligned} x1 &= x \\ x2 &= y \\ x3 &= \dot{x} \\ x4 &= \dot{y} \end{aligned}$$

Hence we have the following system of equations:

$$\dot{x1} = x3$$
$$\dot{x2} = x4$$
$$\dot{x3} = \frac{-x1}{(x1^2 + x2^2)^{3/2}}$$
$$\dot{x4} = \frac{-x2}{(x1^2 + x^2)^{3/2}}$$

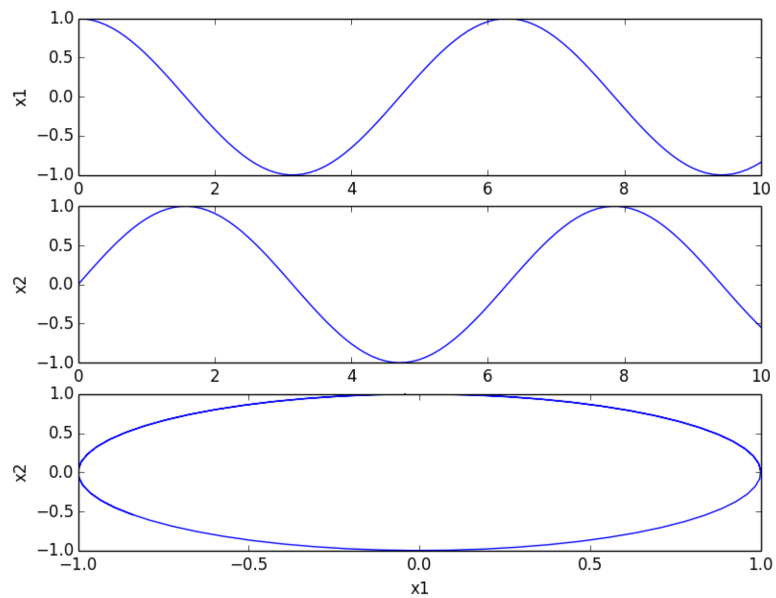
Defining the initial conditions:

$$x1(0) = 1$$
$$x2(0) = 0$$
$$x3(0) = 0$$
$$x4(0) = 1$$

we satisfy the requirement that $\frac{v^2}{R} = \frac{1}{R^2}$ for the circular orbit. The following plots are evaluated using a stepsize of 0.1s and with 100 steps.

Code:

```
rungeODE(['x3','x4','-x1*pow(x1**2+x2**2,-3/2.0)','-x2*pow(x1**2+x2**2,-3/2.0)'],[1,0,0,1],0,0.1,100)
```



The plot of y against x (bottom plot) is clearly a circle of radius unity.

Appendix: Question 2 code

```
"""
This function numerically solves the
simple harmonic oscilator equation
using k/m=1. The input is the step
size h, and the methods used are the explicit
Euler method and midpoint method.
h = Starting Step size (time)
The number of subintervals is chosen
so that three cycles are plotted. The error at each
step is then plotted for each method
on the same axis.
"""
from math import *
from numpy import *
from pylab import *
import sys
def eulerrun(h):
    #main loop for Explicit Euler Method
    #initial conditions x=1(max), v=0
    x = 1
    v = 0
    xlist = [1]
    vlist = [0]
    h = min(h,0.1)
    N = int((3*2*pi)/h)
    det = 1 + h**2
    for n in range(N):
        x1 = x + h*v
        v1 = v - h*x
        xlist = xlist + [x1]
        vlist = vlist + [v1]
        x = x1
        v = v1
    t = [h*n for n in range(N+1)]
    # analytic solution is just cos(t)
    anx = [cos(t[i]) for i in range(N+1)]
    error = [anx[i]-xlist[i] for i in range(N+1)]
    return error
def midpointrun(h):
    #main loop for Midpoint Method
    #initial conditions x=1(max), v=0
    x = 1
    v = 0
    xlist = [1]
    vlist = [0]
    h = min(h,0.1)
    N = int((3*2*pi)/h)
    for n in range(N):
        xmid = x + (h/2)*v
        vmid = v - (h/2)*x
        x1 = x + h*vmid
        v1 = v - h*xmid
        xlist = xlist + [x1]
        vlist = vlist + [v1]
        x = x1
        v = v1
    t = [h*n for n in range(N+1)]
    # analytic solution is just cos(t)
    anx = [cos(t[i]) for i in range(N+1)]
```

```
    error = [anx[i]-xlist[i] for i in range(N+1)]
    return error
if len(sys.argv)==2:
    [name,h] = sys.argv
    h=abs(float(h))
    N = int((3*2*pi)/h)
    t = [h*n for n in range(N+1)]
    listeuler = eulerrun(h)
    listmidpoint = midpointrun(h)
    eulerplot = plot(t,listeuler,'b',label='Euler')
    midpointplot = plot(t,listmidpoint,'r',label='Midpoint')
    xlabel('Time/s')
    ylabel('Error')
    legend()
    show()
else:
    print 'Incorrect number of arguments'
```

Appendix: Question 3 code

```
"""
This function numerically solves the
simple harmonic oscilator equation
using k/m=1. The input is the start step
size h, and the methods used are the explicit
Euler method and midpoint method.
h = Starting Step size (time)
The number of subintervals is chosen
so that three cycles are plotted.
The evaluation is performed for step-sizes
h, h/2, h/4, h/8 and h/16. The error
(given by x analytic - x for each step)
is integrated over the three cycles to
give the global error. The error at each
step-size is then plotted for each method
on the same axis.
"""
from math import *
from numpy import *
from pylab import *
import sys
def eulerrun(h):
    #main loop for Explicit Euler Method
    #initial conditions x=1(max), v=0
    x = 1
    v = 0
    xlist = [1]
    vlist = [0]
    h = min(h,0.1)
    N = int((3*2*pi)/h)
    det = 1 + h**2
    for n in range(N):
        x1 = x + h*v
        v1 = v - h*x
        xlist = xlist + [x1]
        vlist = vlist + [v1]
        x = x1
        v = v1
    t = [h*n for n in range(N+1)]
    # analytic solution is just cos(t)
    anx = [cos(t[i]) for i in range(N+1)]
    globalerror = abs(sum([anx[i]-xlist[i] for i in range(N+1)]))
    return globalerror
def midpointrun(h):
    #main loop for Midpoint Method
    #initial conditions x=1(max), v=0
    x = 1
    v = 0
    xlist = [1]
    vlist = [0]
    h = min(h,0.1)
    N = int((3*2*pi)/h)
    for n in range(N):
        xmid = x + (h/2)*v
        vmid = v - (h/2)*x
        x1 = x + h*vmid
        v1 = v - h*xmid
        xlist = xlist + [x1]
        vlist = vlist + [v1]
```

```
        x = x1
        v = v1
    t = [h*n for n in range(N+1)]
    # analytic solution is just cos(t)
    anx = [cos(t[i]) for i in range(N+1)]
    globalerror = abs(sum([anx[i]-xlist[i] for i in range(N+1)]))
    return globalerror
if len(sys.argv)==2:
    [name,h] = sys.argv
    h=abs(float(h))
    listeulerh = [eulerrun(h)]
    listmidpointh = [midpointrun(h)]
    for i in range(4):
        h = h/2
        listeulerh = listeulerh + [eulerrun(h)]
        listmidpointh = listmidpointh + [midpointrun(h)]
    eulerplot = plot([h,h/2,h/4,h/8,h/16],listeulerh,'bo',label='Euler')
    plot([h,h/2,h/4,h/8,h/16],listeulerh,'b-')
    midpointplot = plot([h,h/2,h/4,h/8,h/16],listmidpointh,'ro',label='Midpoint')
    plot([h,h/2,h/4,h/8,h/16],listmidpointh,'r-')
    xticks([h,h/2,h/4,h/8,h/16], ['h','h/2','h/4','h/8','h/16'])
    xlabel('Step Size h')
    ylabel('Global error')
    legend()
    show()
else:
    print 'Incorrect number of arguments'
```

Appendix: Question 4 code

```python
from math import *
from numpy import *
from pylab import *
from MyVector import *
"""
Runge-Kutta ODE Solver (maximum 10 variables!)
Input:
func = List of N strings, each representing the first derivative
    of the ith variable in terms of the variables x1...xN
init = List of N floats, initial conditions for x1 ... xN
t0 = Initial time (float)
h = Step size (seconds)
steps = Number of steps to evaluate
Output:
CSV file of t,x1,...,xN values for each step
"""
def rungekutta(vecold,t,h,func):
    # algorithm routine
    """
    Runge-Kutta step function
    vecold = N-vector of variable values [x1,x2,...,xN] at time t
    t = initial time (float number)
    h = Time step size
    func = List of N strings that for the function of derivatives
    dxi/dt from independent variable t in terms of variables x1, x2 ... xN.
    Output = vecnew, N-list of parameter values at t+h.
    """
    if len(vecold) != len(func):
        return 'Length Error: Variable vector length: %d, Derivative function \
length: %d' %(len(vecold),len(func))
    N = len(vecold)
    t = float(t)
    h = float(h)
    vecold = [float(vecold[i]) for i in range(N)]
    prek1 = funceval(func,vecold,t)
    k1 = [h*prek1[i] for i in range(N)]
    prek2 = funceval(func,[vecold[i]+k1[i]/2.0 for i in range(N)],t+h/2.0)
    k2 = [h*prek2[i] for i in range(N)]
    prek3 = funceval(func,[vecold[i]+k2[i]/2.0 for i in range(N)],t+h/2.0)
    k3 = [h*prek3[i] for i in range(N)]
    prek4 = funceval(func,[vecold[i]+k3[i] for i in range(N)],t+h)
    k4 = [h*prek4[i] for i in range(N)]
    vecnew = [vecold[i]+k1[i]/6.0+k2[i]/3.0+k3[i]/3.0+k4[i]/6.0 for i in range(N)]
    return vecnew
def funceval(func,values,t):
    """
    Evaluates the vector-valued function func in terms
    of the variables x1, x2, ..... , xN at values determined by
    the N-list values and at time t.
    """
    if len(func) != len(values):
        return 'Length Error: Function vector length: %d, \
Values vector length: %d' %(len(func),len(values))
    if len(values) > 10:
        return 'Too many variables'
    N = len(func)
    t = float(t)
    values = [float(values[i]) for i in range(N)]
    values = values + [0 for i in range(10-N)]
```

```
    values = values + [t]
    outlist = []
    for i in range(N):
        function = lambda x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,t:eval(func[i])
        out = function(*values)
        outlist = outlist + [out]
    return outlist
def rungeODE(func,init,t0,h,steps):
    # driver routine
    steps = abs(int(steps))
    t = float(t0)
    h = float(h)
    if len(func) != len(init):
        return 'Length error: Function vector length: %d, \
Initial condition value length: %d' %(len(func),len(init))
    if len(func) > 10:
        return 'Too many variables'
    N = len(func)
    init = [float(init[i]) for i in range(N)]
    savefile = open('RungeKuttaoutput.csv','w')
    savefile.write('t,x1,x2,x3,x4,x5,x6,x7,x8,x9,x10\n')
    init = init + [0 for i in range(10-N)]
    func = func + ['0' for i in range(10-N)]
    # highly inefficient print method while I find a way to unpack in Python 2
    savefile.write('%f,'%t)
    for i in range(9):
        savefile.write('%f,'%init[i])
    savefile.write('%f\n'%init[9])
    xlist = [init[0]]
    ylist = [init[1]]
    for i in range(steps):
        output = rungekutta(init,t,h,func)
        t = t + h
        xlist = xlist + [output[0]]
        ylist = ylist + [output[1]]
        savefile.write('%f,'%t)
        for i in range(9):
            savefile.write('%f,'%init[i])
        savefile.write('%f\n'%init[9])
        init = output
    savefile.close()
    tlist = [float(t0)+i*h for i in range(steps+1)]
    subplot(311)
    plot(tlist,xlist)
    xlabel('Time/s')
    ylabel('x1')
    subplot(312)
    plot(tlist,ylist)
    xlabel('Time/s')
    ylabel('x2')
    subplot(313)
    plot(xlist,ylist)
    xlabel('x1')
    ylabel('x2')
    show()
```