```
# -*- coding: utf-8 -*-
"""
Created on Wed Oct  8 17:17:48 2014

@author: bob
"""

import scipy

class RungeKutta(object):

    def __init__(self,ode,A=None,b=None,c=None):
        """
        Initializes a runge-kutta time integration object for the system o
        ODEs specified by the object ode
        Input:
            ode -- an object of the class ODESystem that contain the ODE t
            integrated
        Output:
            an object of the class RungeKutta that can integrate
            the ODE specified in the object ode

        #If one of the parameters is left None all the parameters A,b,c wi
        #set to the default runge-kutta 4th order sceme (RG4)
               0
        1/2    1/2
        1/2    0    1/2
        1      0    0    1
        1/6    1/3    1/3    1/6

        """
        if (A==None) or (b==None)  or (c==None) :
            A = [[0 * j *i for i in range(4)] for  j in range(4)]
            A[1][0] = 1.0/2
            A[2][1] = 1.0/2
            A[3][2] = 1.0
            c = [0 * i for i in range(4)]
            c[1] = 1.0/2
            c[2] = 1.0/2
            c[3] = 1.0
            b = [0 * i for i in range(4)]
            b[0] = 1.0/6
            b[1] = 1.0/3
            b[2] = 1.0/3
            b[3] = 1.0/6
        self.A = A
        self.b = b
        self.c = c
```

```
        self.ode = ode

    def step(self,tn,yn,h):
        """
        takes a single time step using the runge-kutta method
            y_(n+1) = y_n + sum(b_i*k_i)
        Input:
            tn -- current time
            yn -- state at time tn
            h - size of time step
        Output:
            y -- state at time t0+h
        """
        k = self.kValues(tn,yn,h)
        lincombinatie = scipy.zeros(len(yn))
        for i in range(len(self.b)):
            lincombinatie = scipy.add(scipy.multiply(k[i],self.b[i]*h), li
        return yn + lincombinatie
    def  kValues(self,tn,yn,h):
        #Initialise an empty vector k of the same length as b and init tne
        A = self.A
        b = self.b
        c = self.c
        k = [ [0. * i *j for j in range(len(yn))]  for i in range(len(b))]
        tnew = 0
        ynew = scipy.zeros(len(yn))
        lincombinatie = ynew
        for i in range(len(b)):
            tnew = tn + c[i]*h
            ynew = scipy.zeros(len(yn))
            lincombinatie = scipy.zeros(len(yn))
            for j in range(i):
                prod = scipy.multiply(A[i][j]*h,k[j])
                lincombinatie = scipy.add(lincombinatie,prod)
            ynew = scipy.add(yn,lincombinatie)
            k[i] = scipy.multiply(1,self.ode.f(tnew,ynew))
            #k[i] = scipy.multiply(h,self.ode.f(tnew,ynew))
        return k
    def scalarProductArray(self,sc,ar):
        return [x*sc for x in ar]
    def sumOfArray(self,ar1,ar2):
        if len(ar1) != len(ar2):
            raise AttributeError
        return [ar1[i]+ar2[i] for i in range(len(ar1))]

    def integrate(self,y0,t0,tend,h):
        """
        Integrates using forward Euler time steps
```

```
        Input:
            t0 -- initial time
            y0 -- initial condition at time t0
            tend -- time horizon of time integration
            Dt -- size of time step
        """
        # obtain the number of time steps
        N = int(scipy.ceil(tend/h))
        # create a vector of time instances
        t = scipy.arange(t0,N*h+h/2.,h)
        # obtain the number of equations
        D = scipy.size(y0)
        # create the matrix that will contain the solutions
        y = [[0*i*j for i in range(D)] for j in range(N+1)]
        # set the initial condition
        y[0]=y0
        # perform N time steps
        for n in range(N):
            y[n+1]=self.step(t[n],y[n],h)
        return t,y
```