```python
# -*- coding: utf-8 -*-
"""
Created on Tue Mar 19 15:56:27 2019
HW5b
@author: Ryan Dalby
"""

import math
import numpy as np
import matplotlib.pyplot as plt
from cooling_fin import cooling_fin
import scipy.integrate as integrate

import pandas as pd

#table = pd.DataFrame(columns = ["n", "y1n", "y2n", "k11", "k12", "k21", "k22", "k31", "k32", "k41"

def runge_kutta_four(func, yInitial, xInitial, xFinal, numSteps):
    """
    Given a function(dy[]/dx = func(x,y[])) and yInitial[] values at an xInitial will find
    y[] at xFinal with a given number of steps between xInitial and xFinal
    """
    h = abs(xFinal - xInitial) / numSteps
    x = xInitial
    y = yInitial
#    row = np.empty(13)
    for i in range(numSteps):
#        print(x,y)
#        row[0] = i
#        row[1] = y[0]
#        row[2] = y[1]
        k1 = func(x, y)
#        row[3] = k1[0]
#        row[4] = k1[1]
        k2 = func(x + h/2, y + k1*h/2)
#        row[5] = k2[0]
#        row[6] = k2[1]
        k3 = func(x + h/2, y + k2*h/2)
#        row[7] = k3[0]
#        row[8] = k3[1]
        k4 = func(x + h, y + k3*h)
#        row[9] = k4[0]
#        row[10] = k4[1]
        y = y + (k1 + 2*k2 + 2*k3 + k4)/6 * h
        x = x + h
#        row[11] = y[0]
#        row[12] = y[1]
#        table.loc[i] = row
#    print(x,y)

    return y


##Define test ODE
#f = lambda t,y: 2 * (325/850) * (math.sin(t)**2) - (200*(1+y)**(3/2))/850
#print(rungeKutta4(f, 2, 0, 10, 50))
```

```
#w = integrate.solve_ivp(f, (0.0,10.0), np.array([2]))
#print(w)


def shootingMethodForCooling_fin(func, firstx, firstT, secondx, secondT, numberOfNodes, beta1, beta
    """
    From cooling_fin(passed in as func) which represents two linear ODEs say dT/dx = f(z) and dz/dx
    and two boundary conditions (x1, T1) and (x2, T2), and number of nodes(essentially how many ste
    and beta1 and beta2 guesses for a IVP that will satisfy a BVP
    Will return the z(firstx) that allows problem to be IVP and yet satisfies BVP
    """
    xInitial = firstx #initial x value
    xFinal = secondx #x value we will integrate to

    z1Initial = beta1 #z = dT/dx; first guess for our IVP; z(firstx) = beta1
    z2Initial = beta2 #z = dT/dx; second guess for our IVP; z(firstx) = beta2

    Tb = firstT #K temperature at beginning
    Te = secondT #K temperature at end

    T1Final,_ = runge_kutta_four(func, np.array([Tb, z1Initial]), xInitial, xFinal, numberOfNodes-1
#    print(table.to_string())
#    print()
    T2Final,_ = runge_kutta_four(func, np.array([Tb, z2Initial]), xInitial, xFinal, numberOfNodes-1
#    print(table.to_string())

#    print(T1Final, T2Final)
    zInitial = z1Initial + ((z2Initial - z1Initial) / (T2Final - T1Final)) * (Te - T1Final) #Now we

#    print(runge_kutta_four(func, np.array([Tb, zInitial]), xInitial, xFinal, numberOfNodes)) #Test

    #Now that we have zInitial just solve like IVP and can change xFinal to get temperature at diff
    #Now use this zInital value in another script to get an array of Ts at a given array of xs (cal
    return zInitial


def find_temperature_profile_shooting(firstx, firstT, secondx, secondT, numberOfNodes, shouldPlot =
    """
    From cooling_fin which represents two linear ODEs say dT/dx = f(z) and dz/dx = f(T)
    and two boundary conditions (x1, T1) and (x2, T2), and number of nodes(essentially how many ste
    and an xDesired where the temperature will be calcuated at
    Will display a plot of T vs x(if indicated to do so) and return np arrays of the x values and t
    This is essentially a driver for shootingMethodForCooling_fin
    """
    d = 0.1 #m
    h = 20 #W/m^2*K
    k = 200 #W/m * K
    Too = 300 #K surrounding temperature, time at t = infinity
    func = lambda x,y: cooling_fin(x, y, d, h, k, Too) #Cooling fin returns [dT/dx, dz/dx] given [x

    
    zInitialAtfirstx = shootingMethodForCooling_fin(func, 0, 600, 2.0, 350, numberOfNodes, 0, 4) #b
    xVals = np.linspace(firstx, secondx, numberOfNodes)
    TVals = []
    solverIterations = 30 #how many iterations our solver will take per value it is solving for
    for xVal in xVals:
```

```python
        TVal, _= runge_kutta_four(func, [firstT, zInitialAtfirstx], firstx, xVal, solverIterations)
        TVals.append(TVal)
#     print(table.to_string())
    if(shouldPlot):
        fig, ax = plt.subplots()
        ax.plot(xVals, TVals)
        ax.set_title("Shooting Method temperature vs x solution")
        ax.set_xlabel("x(m)")
        ax.set_ylabel("Temperature(K)")

    return xVals, TVals

shootingx, shootingT = find_temperature_profile_shooting(0, 600, 2.0, 350, 51, shouldPlot=True)
```