```
'''
Created on 14 Oct 2014

@author: bob
'''

import  scipy
import multiprocessing
import function.function as func
import system.waveSystem as wave
import integrators.rungeKutta as rK
from scipy.signal import argrelextrema

UPPERZERO = 0.05

class Worker(object):
    '''
    A class to represent a worker
    '''


    def __init__(self, Ksqr=[1],sigma=[1],g=[1],y0=[0.,1.],n=10,t0=0,tend=
        '''
        Constructor
        '''
        self.Ksqr = Ksqr
        self.sigma = sigma
        self.g = g
        self.n = n
        self.y0 = y0
        self.t0 = t0
        self.tend = tend
        self.h = h
        self.spectrum = scipy.zeros((len(self.Ksqr)*len(self.sigma)*len(se
    def task(self,procnum, return_dict):
        '''
        Defines the task that has to be preformed for a parallel worker.
        '''
        print '%s : started the task '%(self.name)
        teller = 0
        for i in range(len(self.Ksqr)):
            for j in range(len(self.sigma)):
                for k in range(len(self.g)):
                    Ksqr = self.Ksqr[i]
                    sigma = self.sigma[j]
                    g = self.g[k]
                    eigen_nodes = scipy.zeros(self.n)
                    eigen_nodes = self.search(Ksqr,sigma,g,self.n)
```

```
                    a = scipy.append([Ksqr ,sigma ,g] , eigen_nodes,1)
                    self.spectrum[teller,:] = a
                    teller +=1
                    print ('%s : Eigen Nodes for (%i,%i,%i) = %s')%(self.r
        return_dict[procnum] =  self.spectrum
    def taskNonParallel(self):
        '''
        Creates a task with default proces number.
        '''
        manager = multiprocessing.Manager()
        return_dict = manager.dict()
        self.task(0, return_dict)
        print return_dict
        return return_dict[0]

    def search(self,Ksqrnum,sigmanum,gnum,n):
        '''
        A method to search for the first n roots given the specified value
        K sigma and g.
        '''
        raise NotImplementedError

    def endPoint(self,wguess):
    #Create the ode
        funcP = func.P(self.tempKsqrnum,self.tempsigmanum,self.tempgnum,wg
        funcQ = func.Q(self.tempKsqrnum,self.tempsigmanum,self.tempgnum,wg
        vgl = wave.WaveSystem(funcP,funcQ)
        fe = rK.RungeKutta(vgl)
        t_runge,soln_runge = fe.integrate(self.y0, self.t0, self.tend, sel
        # Now we are going to calculate the local minima of the absolute v
        solution_runge = [soln_runge[i][0] for i in range(len(soln_runge))
        return solution_runge[len(solution_runge)-1]

    def zero_point_info(self,Ksqrnum,sigmanum,gnum,wsqrnum):
        '''
        This method will give some information about the zero points of th
        function.
        Output:
            - nb_of_zero: Holds the number of 0 points
            - index_last_min: Holds the index of the last 0 point
            - value_end_point: Holds the value of the endpoint of the equa
        '''
        #Create the ode
        funcP = func.P(Ksqrnum,sigmanum,gnum,wsqrnum)
        funcQ = func.Q(Ksqrnum,sigmanum,gnum,wsqrnum)
        vgl = wave.WaveSystem(funcP,funcQ)
        fe = rK.RungeKutta(vgl)
        t_runge,soln_runge = fe.integrate(self.y0, self.t0, self.tend, sel
```

```
        # Now we are going to calculate the local minima of the absolute v
        solution_runge = [soln_runge[i][0] for i in range(len(soln_runge))
        index_local = argrelextrema(scipy.absolute(solution_runge), scipy.
        # This will in theory give all the points where the data is zero,
        # plus the starting point and possible also the end point.
        # But due to the possible un smoothness of the data some points co
        # appear several times.
        # Seen that we are looking for the first n values of we can safely
        # assume that two 0 points should lie at a minimum distance of say
        # ceil(tend/h) + 1)/20 = N/20
        presision = scipy.ceil(self.tend/self.h)/20
        nb_of_zero = 0
        # Count the number real of local 0 points.
        end_point = solution_runge[len(solution_runge)-1]
        if (len(index_local) == 0):
            return 0,0,end_point,len(solution_runge)
        if index_local[0]>presision:
            nb_of_zero = 1
        for i in range(1,len(index_local)):
            if ((index_local[i]-index_local[i-1])<presision):
                # if this is the case the two points are to close to
                # each other and are the same minima.
                pass
            else:
                # In this case we check the number of actual 0
                if(solution_runge[index_local[i]]< UPPERZERO):
                    nb_of_zero = nb_of_zero + 1
        # nb_of_zero should now be the number of times the function was 0
        index_last_min = index_local[len(index_local)-1]
        return nb_of_zero , index_last_min , end_point , len(solution_rung
```