```
'''
Created on 29 Oct 2014

@author: bob
'''
import multiprocessing
import scipy
from workers.worker2 import Worker2 as worker2
from workers.workerSimple import workerSimple as workerSimple
from workers.worker_nls import WorkerNLS
import matplotlib.pyplot as plot


if __name__ == '__main__':
    manager = multiprocessing.Manager()
    return_dict = manager.dict()
    jobs = []
    nb_ofLoopsPerProces = 10
    nb_Proces = 8
    Ksqr = [1]
    sigma = scipy.linspace(0.1, 500,  nb_ofLoopsPerProces*nb_Proces).tolis
    g = [-1]
    n = 10
    y0=[0.,1.];
    t0=0
    tend=1
    h=0.01
    for i in range(nb_Proces):
        name = 'Worker %s'%(i+1)
        filename = 'File%s.txt'%(i+1)
        eigenSys = workerSimple(Ksqr,sigma[nb_ofLoopsPerProces*i:nb_ofLoop
        p = multiprocessing.Process(target=eigenSys.task,args=(i+1,return_
        jobs.append(p)
        p.start()
    i = 1;
    for p in jobs:
        p.join()
        print 'Job %s finished, from %s'% (i,len(jobs))
        i+=1
    result = scipy.zeros((nb_ofLoopsPerProces*nb_Proces,n+3))
    for i in range(nb_Proces):
        result[i*nb_ofLoopsPerProces:(i+1)*nb_ofLoopsPerProces,:] =  retur
    fig = plot.figure()
    fig.suptitle('Dispersion Relation', fontsize=18)
    ax = fig.add_subplot(111)
    fig.subplots_adjust(top=0.85)
    ax.set_xlabel('Sigma', fontsize=16)
    ax.set_ylabel('Omega sqr', fontsize=16)
```

```
    ax.tick_params(axis='both', which='major', labelsize=14)
    for i in range(n):
        ax.plot(result[:,1],result[:,3+i])
    plot.savefig('../../plot/dispersionsigma.eps')

    plot.show()
```

```
'''
Created on 14 Oct 2014
@author: bob
'''
class Function(object):
    '''
    A class to represent function objects, these functions
    must be able to be evaluated and to be derived at some point.
    '''
    def __init__(self, delta=0.001):
        '''
        Constructor
        '''
        self._DELTA = delta
        pass
    def evaluate(self,x):
        '''
        A method to evaluate the function
        '''
        raise NotImplementedError
    def derivative(self,x):
        '''
        A method to calculate the derivative of the function
        '''
        return (self.evaluate(x+self._DELTA/2) - self.evaluate(x-self._DEL
class P(Function):
    def __init__(self,Ksqr,sigma,g,wsqr):
        Function.__init__(self)
        self.Ksqr = Ksqr
        self.sigma = sigma
        self.g = g
        self.wsqr = wsqr
    def evaluate(self, x):
        return self.wsqr*(rho0(self.Ksqr,self.sigma,self.g,self.wsqr).eval

class Q(Function):
    def __init__(self,Ksqr,sigma,g,wsqr):
        Function.__init__(self)
        self.Ksqr = Ksqr
        self.sigma = sigma
        self.g = g
        self.wsqr = wsqr
    def evaluate(self, x):
        return self.Ksqr*((rho0(self.Ksqr,self.sigma,self.g,self.wsqr).eva
                           (rho0(self.Ksqr,self.sigma,self.g,self.wsqr).de

class rho0(Function):
    def __init__(self,Ksqr,sigma,g,wsqr):
```

```
            Function.__init__(self)
            self.Ksqr = Ksqr
            self.sigma = sigma
            self.g = g
            self.wsqr = wsqr
        def evaluate(self, x):
            return (1+self.sigma*x)
```

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

@author: bob
"""

class ODESystem(object):

    def __init__(self):
        """
        Constructor for ODE system.  Input can take the form that the
        subclasses desire. This abstract class is meant to create a common
        interface that can be used by Integrator objects.
        """
        raise NotImplementedError

    def f(self,t,y):
        """
        Return the righthand side of the ODE
        """
        raise NotImplementedError

    def y_exact(self,t,y0):
        """
        Returns the exact solution for the ODE at times t,
        starting from y0 at time t=0
        """
        raise NotImplementedError
```

```
'''
Created on 29 Oct 2014

@author: bob
'''
import multiprocessing
import scipy
from workers.worker2 import Worker2 as worker2
from workers.workerSimple import workerSimple as workerSimple
from workers.worker_nls import WorkerNLS
import matplotlib.pyplot as plot
from test import plottest


if __name__ == '__main__':
    manager = multiprocessing.Manager()
    return_dict = manager.dict()
    jobs = []
    nb_ofLoopsPerProces = 5
    nb_Proces = 8
    Ksqr = scipy.linspace(1, 500,  nb_ofLoopsPerProces*nb_Proces).tolist()
    sigma = scipy.linspace(1, 1.5, 1).tolist()
    g = scipy.linspace(-1, -1.5, 1).tolist()
    n = 10
    y0=[0.,1.];
    t0=0
    tend=1
    h=0.01
    for i in range(nb_Proces):
        name = 'Worker %s'%(i+1)
        filename = 'File%s.txt'%(i+1)
        eigenSys = workerSimple(Ksqr[nb_ofLoopsPerProces*i:nb_ofLoopsPerPr
        p = multiprocessing.Process(target=eigenSys.task,args=(i+1,return_
        jobs.append(p)
        p.start()
    i = 1;
    for p in jobs:
        p.join()
        print 'Job %s finished, from %s'% (i,len(jobs))
        i+=1
    result = scipy.zeros((nb_ofLoopsPerProces*nb_Proces,n+3))
    print return_dict
    for i in range(nb_Proces):
        result[i*nb_ofLoopsPerProces:(i+1)*nb_ofLoopsPerProces,:] =  retur
    ax1 = plot.subplot2grid((1,2), (0,0), colspan=1)
    ax7 = plot.subplot2grid((1,2), (0, 1), rowspan=1,colspan=1)

    ax1.plot(result[:,0],result[:,3])
```

```
    ax1.plot(result[:,0],result[:,4])
    ax1.plot(result[:,0],result[:,5])
    ax1.plot(result[:,0],result[:,6])
    ax1.plot(result[:,0],result[:,7])
    ax1.plot(result[:,0],result[:,8])
    ax1.plot(result[:,0],result[:,9])
    ax1.plot(result[:,0],result[:,10])
    ax1.plot(result[:,0],result[:,11])
    ax1.plot(result[:,0],result[:,12])
    plottest.PlotWave(data=result,fig = ax7)
    plot.savefig('../../plot/mainResult.eps')
    plot.show()
```

```python
'''
Created on 29 Oct 2014

@author: bob
'''

import scipy
from scipy.integrate import odeint
import function.function as func
import system.waveSystem as wave



class PlotWave(object):
    '''
    A class to plot the curves for the given parameters.
    '''
    def __init__(self, data,fig=None):
        self.fig = fig
        for i in range(len(data[:,0])):
            self.plotRow(data[i,:])
    def f(self,y,t):
        return self.vgl.f(t, y)
    def plotRow(self,row):
        y0 = [0.,1.]
        t  = scipy.linspace(0, 1., 1000)
        Solution = row[3:]
        Ksqr = row[0]
        sigma = row[1]
        g = row[2]
        for i in Solution:
            funcP = func.P(Ksqr,sigma,g,i)
            funcQ = func.Q(Ksqr,sigma,g,i)
            # create the ODE
            self.vgl = wave.WaveSystem(funcP,funcQ)
            # solve the DEs
            soln = odeint(self.f, y0, t)
            S =soln[:, 0]
            # Normalising the solution
            MAX = max(S[:])
            S = scipy.multiply(1/MAX,S)
            self.fig.plot(t,S)
```

```
# -*- 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
```

```
'''
Created on 14 Oct 2014

@author: bob
'''
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
import function.function as func
import system.waveSystem as wave
import integrators.rungeKutta as rK
from math import ceil
from scipy.io.matlab.mio5_utils import scipy

def f(y,t):
    return vgl.f(t, y)

wsqr = 0.1
sigma = 1.
Ksqr = 1.
g = 1.


# create the ODE
vgl = wave.WaveSystem(func.P(Ksqr,sigma,g,wsqr),func.Q(Ksqr,sigma,g,wsqr))

# initial condition
y0 = [0.,1.]
t0 = 0
tend = 1.
h = 0.01
NbSteps = ceil((tend-t0)/h)
t_scipy  = np.linspace(t0, tend, NbSteps+1)

# solve the ODE using the integrated solver
soln_scipy = odeint(f, y0, t_scipy)
solution_scipy = soln_scipy[:, 0]


# solve the ODE using the self written runge kutta integrator
fe = rK.RungeKutta(vgl)
t_runge,soln_runge = fe.integrate(y0,t0,tend,h)
solution_runge = [soln_runge[i][0] for i in range(len(soln_runge))]

# plot results
plt.subplot(211)
plt.plot(t_runge,solution_runge)
plt.plot(t_scipy,solution_scipy)
```

```

# plot the error of the scipy method and the self implemented method
error = [scipy.absolute(solution_runge[i]-solution_scipy[i]) for i in rang
plt.subplot(212)
plt.plot(t_runge,error)


plt.show()
```

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

@author: bob

A class to visually check the solutions of the differential equations.

"""
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
import system.waveSystem as wave
import function.function as func

# A dummy function for the runge kutta solver.
def f(y,t):
    return vgl.f(t, y)
# Values for wsqr
Solution = [ 0.1 ]
# Parameters of the differential equations
sigma = 1
Ksqr = 1
g = -1
# initial condition
y0 = [0.,1.]
t  = np.linspace(0, 1., 1000)
# Start the calculation of the ode for the differert values of wsqr
for i in Solution:
    funcP = func.P(Ksqr,sigma,g,i)
    funcQ = func.Q(Ksqr,sigma,g,i)
    # create the ODE
    vgl = wave.WaveSystem(funcP,funcQ)
    # solve the DEs
    soln = odeint(f, y0, t)
    S =soln[:, 0]
    plt.plot(t,S)
# plot results
plt.plot(t,S)
plt.show()
```

```
'''
Created on 13 Oct 2014

@author: bob
'''
import ode_system
class WaveSystem(ode_system.ODESystem):

    def __init__(self,P=None,Q=None):
        """
        Creates an object to represent a differential equation of the form
            d/dx[p(x,w) d/dx e(x)] - q(x,w) e(x) = 0
        This equation is internally converted to a system of first order c
            x'_1 = x_2
            x'_2 = (q(x,w) x_1  - p(x,w) x_2)/(p(x,w)')

        Input:
            p -- an object of type function to represent p(x,w) in the equ
            q -- an object of type function to represent q(x,w) in the equ
        Output:
            an object of the class WaveSystem

        """
        self.P = P
        self.Q = Q

    def f(self,x,y):
        """
        Return the righthand side of the ODE
        """
        P = self.P
        Q = self.Q
        dy_1 = y[1]/P.evaluate(x)
        dy_2 = Q.evaluate(x)*y[0]
        return [dy_1, dy_2]
```

```
'''
Created on 08 Nov 2014

@author: bob
'''
from workers.worker import Worker as worker
import scipy
from scipy.optimize import anderson
#from scipy.optimize import newton_krylov

GUESS_W = 1

class WorkerNLS(worker):
    '''
    A class to represent a worker to find the eigenmodes as
    a function of K^2, sigma and g
    '''
    def __init__(self, Ksqr=[1],sigma=[1],g=[1],y0=[0.,1.],n=3,t0=0,tend=1
        '''
        The constructor to set up the right parameters and to create
        the ode's
        '''
        super(WorkerNLS, self).__init__(Ksqr, sigma, g, y0, n, t0, tend, h
        self.f = open(filename, 'w')
        self.filename = filename
        self.name = name
    def search(self,Ksqrnum,sigmanum,gnum,n):
        '''
        Search for the first n roots.
        '''
        guess = GUESS_W
        self.tempKsqrnum = Ksqrnum
        self.tempsigmanum = sigmanum
        self.tempgnum = gnum
        if n ==1:
            orde = 1
            while(orde!=0):
                oneOnGuess = 1./guess
                print 'Root guess: %s'%(1./oneOnGuess)
                root = (1./scipy.absolute(anderson(self.aid_f, oneOnGuess,
                info = self.zero_point_info(Ksqrnum,sigmanum,gnum,root)
                orde = info[0]
                print 'Root and order: %s , %s'%(root,orde)
                guess = root*(orde + 1)
        else:
            pass


```

```
        return [root]
    def aid_f(self,oneonguess):
        return self.endPoint(scipy.absolute(1/oneonguess))

    def _foundAll(self,Nroots):
        for root in Nroots:
            if (root==0):
                return False
        return True
    def _nextGuess(self,Nroots,guess,previous=None):
        '''
        A method to do an educated guess for the next omega value
        '''
        # Find index of lowest 0
        index = 0
        for root in Nroots:
            if (root==0):
                break
            index = index +1
        print '-'*40
        print previous
        print index
        print '-'*40
        if (previous!=0 and previous==index):
            return (guess/2,index)
        newguess = 0
        count = 1
        nonzero = 0
        for root in Nroots:
            newguess = newguess + root/count
            count = 1 + count
            if (root!=0):
                nonzero = nonzero +1
        if nonzero!=0:
            newguess  = (newguess/nonzero)/(index+1)
            return (newguess,index)
        else:
            return (guess/2,index)
```

```
'''
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.n
        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
```

```
'''
Created on 27 Oct 2014

@author: bob
'''

from workers.worker import Worker as worker
import scipy
from scipy.signal import argrelextrema



# Global parameters to tweak the algorithm
DIST = 1
GUESS_W = 10
UPPERZERO = 0.05
PRECISION = 400



class Worker2(worker):
    '''
    A class to represent a worker to find the eigenmodes as
    a function of K^2, sigma and g
    '''
    def __init__(self, Ksqr=[1],sigma=[1],g=[1],y0=[0.,1.],n=3,t0=0,tend=1
        '''
        The constructor to set up the right parameters and to create
        the ode's
        '''
        super(Worker2, self).__init__(Ksqr, sigma, g, y0, n, t0, tend, h)
        self.f = open(filename, 'w')
        self.filename = filename
        self.name = name

    def search(self,Ksqrnum,sigmanum,gnum,n):
        '''
        A method to search for the first n eigen modes, the hard way.
        This method will first guess the value of w. And based on the numb
        modes it will find the value for tune thise guess.
        If it finds a good value for w it will search for the neigbour val
        for w who are eigenmodes.
        '''
        self.tempKsqrnum = Ksqrnum
        self.tempsigmanum = sigmanum
        self.tempgnum = gnum
        N = 100
        #print w
```

```
        fx = scipy.zeros(N)
        nb_of_eigen = 0
        count = 0
        solutionW = []
        while nb_of_eigen < n:
            count = count + 1
            w = scipy.logspace(0.1*count,10*count,N,base=0.5)
            count = count+1
            for x in xrange(N):
                fx[x] = self.endPoint(w[x])
            index_local = argrelextrema(scipy.absolute(fx), scipy.less)[0]
            solutW = scipy.zeros(len(index_local))
            #matplotlib.pyplot.show(block=False)
            for i in xrange(len(index_local)):
                solutW[i] = w[index_local[i]]
            nb_of_eigen = len(index_local) + nb_of_eigen
            solutionW = scipy.append(solutionW, solutW, 1)
            #matplotlib.pyplot.draw()
        #matplotlib.pyplot.show(block=False)
        return solutW[0:n]

    def getAnswer(self):
        if (self.spectrum==None):
            raise RuntimeError('No spectrum calculated')
        return self.spectrum
```

```
'''
Created on 08 Nov 2014

@author: bob
'''


'''
Created on 08 Nov 2014

@author: bob
'''
from workers.worker import Worker as worker
import scipy

GUESS_W = 10
MAX_TOL = 0.000001

class workerSimple(worker):
    '''
    A class to represent a worker to find the eigenmodes as
    a function of K^2, sigma and g
    '''
    def __init__(self, Ksqr=[1],sigma=[1],g=[1],y0=[0.,1.],n=3,t0=0,tend=1
        '''
        The constructor to set up the right parameters and to create
        the ode's
        '''
        super(workerSimple, self).__init__(Ksqr, sigma, g, y0, n, t0, tenc
        self.f = open(filename, 'w')
        self.filename = filename
        self.name = name
    def search(self,Ksqrnum,sigmanum,gnum,n):
        guess = GUESS_W/0.9
        self.tempKsqrnum = Ksqrnum
        self.tempsigmanum = sigmanum
        self.tempgnum = gnum
        endP = self.endPoint(guess)
        while endP<0:
            guess = guess*10
            endP = self.endPoint(guess)
        guesses = scipy.zeros(n)
        positiveGuess = guess
        for i in range(n):
            teken = (-1)**i
            endP = self.endPoint(guess)
            shrinksize = 0.9
            overCount = 1
```

```
            while scipy.absolute(endP)>MAX_TOL:
                previous = guess
                guess = previous * shrinksize
                endP = self.endPoint(guess)

                if (scipy.absolute(endP)<MAX_TOL):
                    break
                if (teken*endP<0):
                    overCount = overCount +1
                    shrinksize = shrinksize + 9./(10**overCount)
                    guess = positiveGuess
                    guess = guess/shrinksize
                elif (teken*endP>0):
                    positiveGuess = guess
            guesses[i] = guess
            while (scipy.absolute(endP)<MAX_TOL ):
                guess = positiveGuess
                guess = guess/shrinksize
                endP = self.endPoint(guess)
                pass
        return guesses
```

```python
'''
Created on 15 Oct 2014

@author: bob

This task tests the worker class in more detail.
'''

from workers.worker2 import Worker2 as worker2
import scipy
from matplotlib import pyplot as plot

if __name__ == '__main__':
    Ksqr = scipy.linspace(0.5, 1.5, 1).tolist()
    sigma = scipy.linspace(0.5, 1.5,10).tolist()
    g = scipy.linspace(0.5, 1.5, 1).tolist()
    n = 3
    y0=[0.,1.];
    t0=0
    tend=1
    h=0.01
    eigenSys = worker2(Ksqr,sigma,g,y0,n,t0,tend,h)
    data = eigenSys.taskNonParallel()
    print data
    plot.plot(data[:,1],data[:,3])
    plot.show()
```

```python
'''
Created on 15 Oct 2014

@author: bob


A test file for the worker class.

'''

from workers.workerSimple import workerSimple as workerSimple
import system.waveSystem as wave
import function.function as func
import integrators.rungeKutta as rK
import matplotlib.pyplot as plt

Ksqr = [1]
sigma = [1]
g = [1]
n = 10
y0 = 0
t0 = 1
tend = 1
h = 0.01

#wsqrnum = 0.3-0.285
wsqrnum = 0.015*(1+23./50)
wsqrnum = wsqrnum*(1+7./50)
wsqrnum = 0.001

class rho0(func.Function):
    def __init__(self,Ksqr,sigma,g,wsqr):
        func.Function.__init__(self)
        self.Ksqr = Ksqr
        self.sigma = sigma
        self.g = g
        self.wsqr = wsqr
    def evaluate(self, x):
        return (1+self.sigma*x)
# Create the two objects to represent the functions P and Q
class P(func.Function):
    def __init__(self,Ksqr,sigma,g,wsqr):
        func.Function.__init__(self)
        self.Ksqr = Ksqr
        self.sigma = sigma
        self.g = g
        self.wsqr = wsqr
    def evaluate(self, x):
```

```
        return self.wsqr*rho0(self.Ksqr,self.sigma,self.g,self.wsqr).evalu
class Q(func.Function):
    def __init__(self,Ksqr,sigma,g,wsqr):
        func.Function.__init__(self)
        self.Ksqr = Ksqr
        self.sigma = sigma
        self.g = g
        self.wsqr = wsqr
    def evaluate(self, x):
        return -self.Ksqr*(rho0(self.Ksqr,self.sigma,self.g,self.wsqr).eva
                           rho0(self.Ksqr,self.sigma,self.g,self.wsqr).der


def plot_ode(Ksqr, sigma, g, wsqr):
    funcP = P(Ksqr,sigma,g,wsqr)
    funcQ = Q(Ksqr,sigma,g,wsqr)
    vgl = wave.WaveSystem(funcP,funcQ)
    fe = rK.RungeKutta(vgl)
    t_runge,soln_runge = fe.integrate(y0,t0,tend,h)
    solution_runge = [soln_runge[i][0] for i in range(len(soln_runge))]
    plt.plot(t_runge,solution_runge)
    plt.show()
    pass


def nbOfZerosnumber_of_zeros():
    eigenSys = workerSimple(Ksqr,sigma,g,y0,n,t0,tend,h)
    print eigenSys.zero_point_info(Ksqrnum=1, sigmanum=1, gnum=1, wsqrnum=

def find_eigen_mode():
    w = wsqrnum
    eigenSys = workerSimple(Ksqr,sigma,g,y0,n,t0,tend,h)
    nb_of_zero , index_last_min , end_point , length_Set = eigenSys.zero_p
    newW = w
    while (nb_of_zero==0):
        newW = (newW+0.0)/2
        nb_of_zero , index_last_min , end_point , length_Set = eigenSys.ze
    print nb_of_zero , index_last_min , end_point , length_Set

    print newW
    previousW = newW
    counter = 0
    while(abs(end_point)>0.001):
        counter = counter +1
        nb_of_zero_new , index_last_min_new , end_point_new , length_Set_n
        print counter, newW , nb_of_zero_new , index_last_min_new , end_po
        if (abs(end_point_new)<0.001):
            break
```

```
        if (nb_of_zero_new>=nb_of_zero):
            # Vergroot w
            previousW = newW
            newW = newW*(1+((length_Set-index_last_min_new)+0.0)/index_las
        if (nb_of_zero_new < nb_of_zero):
            # Nu weten we dat het vorige punt wel nog achter het nulpunt l
            # gemmidelde tussen het slechte punt en het vorige goede punt.
            newW = ((newW+previousW)+0.0)/2
    print newW
    plot_ode(Ksqr=1, sigma=1, g=1, wsqr=newW)

def task():
    eigenSys = workerSimple(Ksqr,sigma,g,y0,n,t0,tend,h)
    eigenSys.task()
    pass

if __name__ == '__main__':
    Ksqr = [1]
    sigma = [1]
    g = [1]
    n = 10
    y0 = 0
    t0 = 1
    tend = 1
    h = 0.1
    Ksqr=[1];sigma=[1];g=[1];y0=[0.,1.];n=10;t0=0;tend=1;h=0.01
    #nbOfZerosnumber_of_zeros()
    #find_eigen_mode()
    #plot_ode(Ksqr=1, sigma=1, g=1, wsqr=wsqrnum)
    task()
```