

```
1  '''
2  Created on 29 Oct 2014
3
4  @author: bob
5  '''
6  import multiprocessing
7  import scipy
8  from workers.worker2 import Worker2 as worker2
9  from workers.workerSimple import workerSimple as workerSimple
10 from workers.worker_nls import WorkerNLS
11 import matplotlib.pyplot as plot
12
13
14 if __name__ == '__main__':
15     manager = multiprocessing.Manager()
16     return_dict = manager.dict()
17     jobs = []
18     nb_ofLoopsPerProces = 10
19     nb_Proces = 8
20     Ksqr = [1]
21     sigma = scipy.linspace(0.1, 500, nb_ofLoopsPerProces*nb_Proces).tolist()
22     g = [-1]
23     n = 10
24     y0=[0.,1.];
25     t0=0
26     tend=1
27     h=0.01
28     for i in range(nb_Proces):
29         name = 'Worker %s'%(i+1)
30         filename = 'File%s.txt'%(i+1)
31         eigenSys = workerSimple(Ksqr,sigma[nb_ofLoopsPerProces*i:nb_ofLoopsPerProces*(i+1)])
32         p = multiprocessing.Process(target=eigenSys.task,args=(i+1,return_dict,filename))
33         jobs.append(p)
34         p.start()
35     i = 1;
36     for p in jobs:
37         p.join()
38         print 'Job %s finished, from %s'% (i,len(jobs))
39         i+=1
40     result = scipy.zeros((nb_ofLoopsPerProces*nb_Proces,n+3))
41     for i in range(nb_Proces):
42         result[i*nb_ofLoopsPerProces:(i+1)*nb_ofLoopsPerProces,:] = return_dict['%s'%filename]
43     fig = plot.figure()
44     fig.suptitle('Dispersion Relation', fontsize=18)
45     ax = fig.add_subplot(111)
46     fig.subplots_adjust(top=0.85)
47     ax.set_xlabel('Sigma', fontsize=16)
48     ax.set_ylabel('Omega sqr', fontsize=16)
```

```
49     ax.tick_params(axis='both', which='major', labelsize=14)
50     for i in range(n):
51         ax.plot(result[:,1],result[:,3+i])
52     plot.savefig('../..plot/dispersionsigma.eps')
53
54     plot.show()
55
56
57
58
59
60
```

```
1  '''
2  Created on 14 Oct 2014
3  @author: bob
4  '''
5  class Function(object):
6      '''
7      A class to represent function objects, these functions
8      must be able to be evaluated and to be derived at some point.
9      '''
10     def __init__(self, delta=0.001):
11         '''
12         Constructor
13         '''
14         self._DELTA = delta
15         pass
16     def evaluate(self, x):
17         '''
18         A method to evaluate the function
19         '''
20         raise NotImplementedError
21     def derivative(self, x):
22         '''
23         A method to calculate the derivative of the function
24         '''
25         return (self.evaluate(x+self._DELTA/2) - self.evaluate(x-self._DELTA/2))/self._DELTA
26 class P(Function):
27     def __init__(self, Ksqr, sigma, g, wsqr):
28         Function.__init__(self)
29         self.Ksqr = Ksqr
30         self.sigma = sigma
31         self.g = g
32         self.wsqr = wsqr
33     def evaluate(self, x):
34         return self.wsqr*(rho0(self.Ksqr, self.sigma, self.g, self.wsqr).evaluate(x))
35
36 class Q(Function):
37     def __init__(self, Ksqr, sigma, g, wsqr):
38         Function.__init__(self)
39         self.Ksqr = Ksqr
40         self.sigma = sigma
41         self.g = g
42         self.wsqr = wsqr
43     def evaluate(self, x):
44         return self.Ksqr*((rho0(self.Ksqr, self.sigma, self.g, self.wsqr).evaluate(x)) -
45                             (rho0(self.Ksqr, self.sigma, self.g, self.wsqr).evaluate(x-self._DELTA/2)))
46
47 class rho0(Function):
48     def __init__(self, Ksqr, sigma, g, wsqr):
```

```
49         Function.__init__(self)
50         self.Ksqr = Ksqr
51         self.sigma = sigma
52         self.g = g
53         self.wsqr = wsqr
54     def evaluate(self, x):
55         return (1+self.sigma*x)
```

```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Wed Oct  8 17:17:48 2014
4
5 @author: bob
6 """
7
8 class ODESystem(object):
9
10     def __init__(self):
11         """
12         Constructor for ODE system. Input can take the form that the
13         subclasses desire. This abstract class is meant to create a common
14         interface that can be used by Integrator objects.
15         """
16         raise NotImplementedError
17
18     def f(self,t,y):
19         """
20         Return the righthand side of the ODE
21         """
22         raise NotImplementedError
23
24     def y_exact(self,t,y0):
25         """
26         Returns the exact solution for the ODE at times t,
27         starting from y0 at time t=0
28         """
29         raise NotImplementedError
30
31
```

```
1  '''
2  Created on 29 Oct 2014
3
4  @author: bob
5  '''
6  import multiprocessing
7  import scipy
8  from workers.worker2 import Worker2 as worker2
9  from workers.workerSimple import workerSimple as workerSimple
10 from workers.worker_nls import WorkerNLS
11 import matplotlib.pyplot as plot
12 from test import plottest
13
14
15 if __name__ == '__main__':
16     manager = multiprocessing.Manager()
17     return_dict = manager.dict()
18     jobs = []
19     nb_ofLoopsPerProces = 5
20     nb_Proces = 8
21     Ksqr = scipy.linspace(1, 500, nb_ofLoopsPerProces*nb_Proces).tolist()
22     sigma = scipy.linspace(1, 1.5, 1).tolist()
23     g = scipy.linspace(-1, -1.5, 1).tolist()
24     n = 10
25     y0=[0.,1.];
26     t0=0
27     tend=1
28     h=0.01
29     for i in range(nb_Proces):
30         name = 'Worker %s'%(i+1)
31         filename = 'File%s.txt'%(i+1)
32         eigenSys = workerSimple(Ksqr[nb_ofLoopsPerProces*i:nb_ofLoopsPerProces*(i+1)], sigma, g, n, y0, t0, tend, h, filename)
33         p = multiprocessing.Process(target=eigenSys.task, args=(i+1, return_dict, filename))
34         jobs.append(p)
35         p.start()
36     i = 1;
37     for p in jobs:
38         p.join()
39         print 'Job %s finished, from %s'% (i, len(jobs))
40         i+=1
41     result = scipy.zeros((nb_ofLoopsPerProces*nb_Proces,n+3))
42     print return_dict
43     for i in range(nb_Proces):
44         result[i*nb_ofLoopsPerProces:(i+1)*nb_ofLoopsPerProces,:] = return_dict[i*nb_ofLoopsPerProces:(i+1)*nb_ofLoopsPerProces,:]
45     ax1 = plot.subplot2grid((1,2), (0,0), colspan=1)
46     ax7 = plot.subplot2grid((1,2), (0, 1), rowspan=1,colspan=1)
47
48     ax1.plot(result[:,0],result[:,3])
```

```
49     ax1.plot(result[:,0],result[:,4])
50     ax1.plot(result[:,0],result[:,5])
51     ax1.plot(result[:,0],result[:,6])
52     ax1.plot(result[:,0],result[:,7])
53     ax1.plot(result[:,0],result[:,8])
54     ax1.plot(result[:,0],result[:,9])
55     ax1.plot(result[:,0],result[:,10])
56     ax1.plot(result[:,0],result[:,11])
57     ax1.plot(result[:,0],result[:,12])
58     plottest.PlotWave(data=result,fig = ax7)
59     plot.savefig(' ../.. /plot/mainResult.eps')
60     plot.show()
61
62
63
64
65
66
```

```
1  '''
2  Created on 29 Oct 2014
3
4  @author: bob
5  '''
6
7  import scipy
8  from scipy.integrate import odeint
9  import function.function as func
10 import system.waveSystem as wave
11
12
13
14 class PlotWave(object):
15     '''
16     A class to plot the curves for the given parameters.
17     '''
18     def __init__(self, data,fig=None):
19         self.fig = fig
20         for i in range(len(data[:,0])):
21             self.plotRow(data[i,:])
22     def f(self,y,t):
23         return self.vgl.f(t, y)
24     def plotRow(self,row):
25         y0 = [0.,1.]
26         t = scipy.linspace(0, 1., 1000)
27         Solution = row[3:]
28         Ksqr = row[0]
29         sigma = row[1]
30         g = row[2]
31         for i in Solution:
32             funcP = func.P(Ksqr,sigma,g,i)
33             funcQ = func.Q(Ksqr,sigma,g,i)
34             # create the ODE
35             self.vgl = wave.WaveSystem(funcP,funcQ)
36             # solve the DEs
37             soln = odeint(self.f, y0, t)
38             S =soln[:, 0]
39             # Normalising the solution
40             MAX = max(S[:])
41             S = scipy.multiply(1/MAX,S)
42             self.fig.plot(t,S)
43
44
45
46
47
```



```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Wed Oct  8 17:17:48 2014
4
5 @author: bob
6 """
7
8 import scipy
9
10 class RungeKutta(object):
11
12     def __init__(self,ode,A=None,b=None,c=None):
13         """
14         Initializes a runge-kutta time integration object for the system of
15         ODEs specified by the object ode
16         Input:
17             ode -- an object of the class ODESystem that contain the ODE to
18                   integrated
19         Output:
20             an object of the class RungeKutta that can integrate
21             the ODE specified in the object ode
22
23         #If one of the parameters is left None all the parameters A,b,c will
24         #set to the default runge-kutta 4th order scheme (RG4)
25
26             0
27             1/2    1/2
28             1/2    0    1/2
29             1      0    0    1
30             1/6    1/3    1/3    1/6
31
32         """
33         if (A==None) or (b==None) or (c==None) :
34             A = [[0 * j * i for i in range(4)] for j in range(4)]
35             A[1][0] = 1.0/2
36             A[2][1] = 1.0/2
37             A[3][2] = 1.0
38             c = [0 * i for i in range(4)]
39             c[1] = 1.0/2
40             c[2] = 1.0/2
41             c[3] = 1.0
42             b = [0 * i for i in range(4)]
43             b[0] = 1.0/6
44             b[1] = 1.0/3
45             b[2] = 1.0/3
46             b[3] = 1.0/6
47         self.A = A
48         self.b = b
49         self.c = c
```

```
49         self.ode = ode
50
51     def step(self,tn,yn,h):
52         """
53         takes a single time step using the runge-kutta method
54         y_(n+1) = y_n + sum(b_i*k_i)
55         Input:
56             tn -- current time
57             yn -- state at time tn
58             h -- size of time step
59         Output:
60             y -- state at time t0+h
61         """
62         k = self.kValues(tn,yn,h)
63         lincombinatie = scipy.zeros(len(yn))
64         for i in range(len(self.b)):
65             lincombinatie = scipy.add(scipy.multiply(k[i],self.b[i]*h), lincombinatie)
66         return yn + lincombinatie
67     def kValues(self,tn,yn,h):
68         #Initialise an empty vector k of the same length as b and init tnew
69         A = self.A
70         b = self.b
71         c = self.c
72         k = [ [0. * i *j for j in range(len(yn))] for i in range(len(b))]
73         tnew = 0
74         ynew = scipy.zeros(len(yn))
75         lincombinatie = ynew
76         for i in range(len(b)):
77             tnew = tn + c[i]*h
78             ynew = scipy.zeros(len(yn))
79             lincombinatie = scipy.zeros(len(yn))
80             for j in range(i):
81                 prod = scipy.multiply(A[i][j]*h,k[j])
82                 lincombinatie = scipy.add(lincombinatie,prod)
83             ynew = scipy.add(yn,lincombinatie)
84             k[i] = scipy.multiply(1,self.ode.f(tnew,ynew))
85             #k[i] = scipy.multiply(h,self.ode.f(tnew,ynew))
86         return k
87     def scalarProductArray(self,sc,ar):
88         return [x*sc for x in ar]
89     def sumOfArray(self,ar1,ar2):
90         if len(ar1) != len(ar2):
91             raise AttributeError
92         return [ar1[i]+ar2[i] for i in range(len(ar1))]
93
94     def integrate(self,y0,t0,tend,h):
95         """
96         Integrates using forward Euler time steps
```

```
97         Input:
98             t0 -- initial time
99             y0 -- initial condition at time t0
100            tend -- time horizon of time integration
101            Dt -- size of time step
102        """
103        # obtain the number of time steps
104        N = int(scipy.ceil(tend/h))
105        # create a vector of time instances
106        t = scipy.arange(t0,N*h+h/2.,h)
107        # obtain the number of equations
108        D = scipy.size(y0)
109        # create the matrix that will contain the solutions
110        y = [[0*i*j for i in range(D)] for j in range(N+1)]
111        # set the initial condition
112        y[0]=y0
113        # perform N time steps
114        for n in range(N):
115            y[n+1]=self.step(t[n],y[n],h)
116        return t,y
117
118
```

```
1  '''
2  Created on 14 Oct 2014
3
4  @author: bob
5  '''
6  import numpy as np
7  import matplotlib.pyplot as plt
8  from scipy.integrate import odeint
9  import function.function as func
10 import system.waveSystem as wave
11 import integrators.rungeKutta as rK
12 from math import ceil
13 from scipy.io.matlab.mio5_utils import scipy
14
15 def f(y,t):
16     return vgl.f(t, y)
17
18 wsqr = 0.1
19 sigma = 1.
20 Ksqr = 1.
21 g = 1.
22
23
24 # create the ODE
25 vgl = wave.WaveSystem(func.P(Ksqr,sigma,g,wsqr),func.Q(Ksqr,sigma,g,wsqr))
26
27 # initial condition
28 y0 = [0.,1.]
29 t0 = 0
30 tend = 1.
31 h = 0.01
32 NbSteps = ceil((tend-t0)/h)
33 t_scipy = np.linspace(t0, tend, NbSteps+1)
34
35 # solve the ODE using the integrated solver
36 soln_scipy = odeint(f, y0, t_scipy)
37 solution_scipy = soln_scipy[:, 0]
38
39
40 # solve the ODE using the self written runge kutta integrator
41 fe = rK.RungeKutta(vgl)
42 t_runge,soln_runge = fe.integrate(y0,t0,tend,h)
43 solution_runge = [soln_runge[i][0] for i in range(len(soln_runge))]
44
45 # plot results
46 plt.subplot(211)
47 plt.plot(t_runge,solution_runge)
48 plt.plot(t_scipy,solution_scipy)
```

```
49
50 # plot the error of the scipy method and the self implemented method
51 error = [scipy.absolute(solution_runge[i]-solution_scipy[i]) for i in range(
52 plt.subplot(212)
53 plt.plot(t_runge,error)
54
55
56 plt.show()
```

```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Wed Oct  8 17:17:48 2014
4
5 @author: bob
6
7 A class to visually check the solutions of the differential equations.
8
9 """
10 import numpy as np
11 import matplotlib.pyplot as plt
12 from scipy.integrate import odeint
13 import system.waveSystem as wave
14 import function.function as func
15
16 # A dummy function for the runge kutta solver.
17 def f(y,t):
18     return vgl.f(t, y)
19 # Values for wsqr
20 Solution = [ 0.1 ]
21 # Parameters of the differential equations
22 sigma = 1
23 Ksqr = 1
24 g = -1
25 # initial condition
26 y0 = [0.,1.]
27 t = np.linspace(0, 1., 1000)
28 # Start the calculation of the ode for the differert values of wsqr
29 for i in Solution:
30     funcP = func.P(Ksqr,sigma,g,i)
31     funcQ = func.Q(Ksqr,sigma,g,i)
32     # create the ODE
33     vgl = wave.WaveSystem(funcP,funcQ)
34     # solve the DEs
35     soln = odeint(f, y0, t)
36     S=soln[:, 0]
37     plt.plot(t,S)
38 # plot results
39 plt.plot(t,S)
40 plt.show()
41
42
43
44
45
46
```

```
1  '''
2  Created on 13 Oct 2014
3
4  @author: bob
5  '''
6  import ode_system
7  class WaveSystem(ode_system.ODESystem):
8
9      def __init__(self,P=None,Q=None):
10         """
11         Creates an object to represent a differential equation of the form
12             
$$d/dx[p(x,w) d/dx e(x)] - q(x,w) e(x) = 0$$

13         This equation is internally converted to a system of first order ODEs
14             
$$x'_1 = x_2$$

15             
$$x'_2 = (q(x,w) x_1 - p(x,w) x_2)/(p(x,w)')$$

16
17         Input:
18             p -- an object of type function to represent p(x,w) in the equation
19             q -- an object of type function to represent q(x,w) in the equation
20         Output:
21             an object of the class WaveSystem
22
23         """
24         self.P = P
25         self.Q = Q
26
27     def f(self,x,y):
28         """
29         Return the righthand side of the ODE
30         """
31         P = self.P
32         Q = self.Q
33         dy_1 = y[1]/P.evaluate(x)
34         dy_2 = Q.evaluate(x)*y[0]
35         return [dy_1, dy_2]
36
37
38
39
40
41
```

```
1  '''
2  Created on 08 Nov 2014
3
4  @author: bob
5  '''
6  from workers.worker import Worker as worker
7  import scipy
8  from scipy.optimize import anderson
9  #from scipy.optimize import newton_krylov
10
11  GUESS_W = 1
12
13  class WorkerNLS(worker):
14      '''
15      A class to represent a worker to find the eigenmodes as
16      a function of K^2, sigma and g
17      '''
18      def __init__(self, Ksqr=[1],sigma=[1],g=[1],y0=[0.,1.],n=3,t0=0,tend=1):
19          '''
20          The constructor to set up the right parameters and to create
21          the ode's
22          '''
23          super(WorkerNLS, self).__init__(Ksqr, sigma, g, y0, n, t0, tend, h)
24          self.f = open(filename, 'w')
25          self.filename = filename
26          self.name = name
27      def search(self,Ksqrnum,sigmanum,gnum,n):
28          '''
29          Search for the first n roots.
30          '''
31          guess = GUESS_W
32          self.tempKsqrnum = Ksqrnum
33          self.tempsigmanum = sigmanum
34          self.tempgnum = gnum
35          if n ==1:
36              orde = 1
37              while(orde!=0):
38                  oneOnGuess = 1./guess
39                  print 'Root guess: %s'%(1./oneOnGuess)
40                  root = (1./scipy.absolute(anderson(self.aid_f, oneOnGuess,
41                  info = self.zero_point_info(Ksqrnum,sigmanum,gnum,root)
42                  orde = info[0]
43                  print 'Root and order: %s , %s'%(root,orde)
44                  guess = root*(orde + 1)
45          else:
46              pass
47
48
```



```
49     return [root]
50 def aid_f(self,oneonguess):
51     return self.endPoint(scipy.absolute(1/oneonguess))
52
53 def _foundAll(self,Nroots):
54     for root in Nroots:
55         if (root==0):
56             return False
57     return True
58 def _nextGuess(self,Nroots,guess,previous=None):
59     '''
60     A method to do an educated guess for the next omega value
61     '''
62     # Find index of lowest 0
63     index = 0
64     for root in Nroots:
65         if (root==0):
66             break
67     index = index +1
68     print '-'*40
69     print previous
70     print index
71     print '-'*40
72     if (previous!=0 and previous==index):
73         return (guess/2,index)
74     newguess = 0
75     count = 1
76     nonzero = 0
77     for root in Nroots:
78         newguess = newguess + root/count
79         count = 1 + count
80         if (root!=0):
81             nonzero = nonzero +1
82     if nonzero!=0:
83         newguess = (newguess/nonzero)/(index+1)
84         return (newguess,index)
85     else:
86         return (guess/2,index)
```

97
98
99
100
101

```
1  '''
2  Created on 14 Oct 2014
3
4  @author: bob
5  '''
6
7  import  scipy
8  import multiprocessing
9  import function.function as func
10 import system.waveSystem as wave
11 import integrators.rungeKutta as rK
12 from scipy.signal import argrelextrema
13
14 UPPERZERO = 0.05
15
16 class Worker(object):
17     '''
18     A class to represent a worker
19     '''
20
21
22     def __init__(self, Ksqr=[1],sigma=[1],g=[1],y0=[0.,1.],n=10,t0=0,tend=
23         '''
24         Constructor
25         '''
26         self.Ksqr = Ksqr
27         self.sigma = sigma
28         self.g = g
29         self.n = n
30         self.y0 = y0
31         self.t0 = t0
32         self.tend = tend
33         self.h = h
34         self.spectrum = scipy.zeros((len(self.Ksqr)*len(self.sigma)*len(se
35     def task(self,procnum, return_dict):
36         '''
37         Defines the task that has to be preformed for a parallel worker.
38         '''
39         print '%s : started the task'%(self.name)
40         teller = 0
41         for i in range(len(self.Ksqr)):
42             for j in range(len(self.sigma)):
43                 for k in range(len(self.g)):
44                     Ksqr = self.Ksqr[i]
45                     sigma = self.sigma[j]
46                     g = self.g[k]
47                     eigen_nodes = scipy.zeros(self.n)
48                     eigen_nodes = self.search(Ksqr,sigma,g,self.n)
```

```
49         a = scipy.append([Ksqr ,sigma ,g] , eigen_nodes,1)
50         self.spectrum[teller,:] = a
51         teller +=1
52         print ('%s : Eigen Nodes for (%i,%i,%i) = %s'%(self.n
53     return_dict[procnum] = self.spectrum
54 def taskNonParallel(self):
55     '''
56     Creates a task with default proces number.
57     '''
58     manager = multiprocessing.Manager()
59     return_dict = manager.dict()
60     self.task(0, return_dict)
61     print return_dict
62     return return_dict[0]
63
64 def search(self,Ksqrnum,sigmanum,gnum,n):
65     '''
66     A method to search for the first n roots given the specified value
67     K sigma and g.
68     '''
69     raise NotImplementedError
70
71 def endPoint(self,wguess):
72     #Create the ode
73     funcP = func.P(self.tempKsqrnum,self.tempsigmanum,self.tempgnum,wg
74     funcQ = func.Q(self.tempKsqrnum,self.tempsigmanum,self.tempgnum,wg
75     vgl = wave.WaveSystem(funcP,funcQ)
76     fe = rK.RungeKutta(vgl)
77     t_runge,soln_runge = fe.integrate(self.y0, self.t0, self.tend, sel
78     # Now we are going to calculate the local minima of the absolute v
79     solution_runge = [soln_runge[i][0] for i in range(len(soln_runge))
80     return solution_runge[len(solution_runge)-1]
81
82 def zero_point_info(self,Ksqrnum,sigmanum,gnum,wsqrnum):
83     '''
84     This method will give some information about the zero points of th
85     function.
86     Output:
87         - nb_of_zero: Holds the number of 0 points
88         - index_last_min: Holds the index of the last 0 point
89         - value_end_point: Holds the value of the endpoint of the equa
90     '''
91     #Create the ode
92     funcP = func.P(Ksqrnum,sigmanum,gnum,wsqrnum)
93     funcQ = func.Q(Ksqrnum,sigmanum,gnum,wsqrnum)
94     vgl = wave.WaveSystem(funcP,funcQ)
95     fe = rK.RungeKutta(vgl)
96     t_runge,soln_runge = fe.integrate(self.y0, self.t0, self.tend, sel
```

```
97     # Now we are going to calculate the local minima of the absolute v
98     solution_runge = [soln_runge[i][0] for i in range(len(soln_runge))
99     index_local = argrextrema(scipy.absolute(solution_runge), scipy.
100     # This will in theory give all the points where the data is zero,
101     # plus the starting point and possible also the end point.
102     # But due to the possible un smoothness of the data some points co
103     # appear several times.
104     # Seen that we are looking for the first n values of we can safely
105     # assume that two 0 points should lie at a minimum distance of say
106     # ceil(tend/h) + 1)/20 = N/20
107     presision = scipy.ceil(self.tend/self.h)/20
108     nb_of_zero = 0
109     # Count the number real of local 0 points.
110     end_point = solution_runge[len(solution_runge)-1]
111     if (len(index_local) == 0):
112         return 0,0,end_point,len(solution_runge)
113     if index_local[0]>presision:
114         nb_of_zero = 1
115     for i in range(1,len(index_local)):
116         if ((index_local[i]-index_local[i-1])<presision):
117             # if this is the case the two points are to close to
118             # each other and are the same minima.
119             pass
120         else:
121             # In this case we check the number of actual 0
122             if(solution_runge[index_local[i]]< UPPERZERO):
123                 nb_of_zero = nb_of_zero + 1
124     # nb_of_zero should now be the number of times the function was 0
125     index_last_min = index_local[len(index_local)-1]
126     return nb_of_zero , index_last_min , end_point , len(solution_rung
127
```

```
1  '''
2  Created on 27 Oct 2014
3
4  @author: bob
5  '''
6
7  from workers.worker import Worker as worker
8  import scipy
9  from scipy.signal import argrelextrema
10
11
12
13 # Global parameters to tweak the algorithm
14 DIST = 1
15 GUESS_W = 10
16 UPPERZERO = 0.05
17 PRECISION = 400
18
19
20
21 class Worker2(worker):
22     '''
23     A class to represent a worker to find the eigenmodes as
24     a function of  $K^2$ , sigma and g
25     '''
26     def __init__(self, Ksqr=[1],sigma=[1],g=[1],y0=[0.,1.],n=3,t0=0,tend=1000,h=0.01):
27         '''
28         The constructor to set up the right parameters and to create
29         the ode's
30         '''
31         super(Worker2, self).__init__(Ksqr, sigma, g, y0, n, t0, tend, h)
32         self.f = open(filename, 'w')
33         self.filename = filename
34         self.name = name
35
36     def search(self,Ksqrnum,sigmanum,gnum,n):
37         '''
38         A method to search for the first n eigen modes, the hard way.
39         This method will first guess the value of w. And based on the number of
40         modes it will find the value for tune these guess.
41         If it finds a good value for w it will search for the neighbour values
42         for w who are eigenmodes.
43         '''
44         self.tempKsqrnum = Ksqrnum
45         self.tempsigmanum = sigmanum
46         self.tempgnum = gnum
47         N = 100
48         #print w
```

```
49         fx = scipy.zeros(N)
50         nb_of_eigen = 0
51         count = 0
52         solutionW = []
53         while nb_of_eigen < n:
54             count = count + 1
55             w = scipy.logspace(0.1*count,10*count,N,base=0.5)
56             count = count+1
57             for x in xrange(N):
58                 fx[x] = self.endPoint(w[x])
59             index_local = argrelextrema(scipy.absolute(fx), scipy.less)[0]
60             solutW = scipy.zeros(len(index_local))
61             #matplotlib.pyplot.show(block=False)
62             for i in xrange(len(index_local)):
63                 solutW[i] = w[index_local[i]]
64             nb_of_eigen = len(index_local) + nb_of_eigen
65             solutionW = scipy.append(solutionW, solutW, 1)
66             #matplotlib.pyplot.draw()
67             #matplotlib.pyplot.show(block=False)
68             return solutW[0:n]
69
70     def getAnswer(self):
71         if (self.spectrum==None):
72             raise RuntimeError('No spectrum calculated')
73         return self.spectrum
74
75
76
77
78
```

```
1  '''
2  Created on 08 Nov 2014
3
4  @author: bob
5  '''
6
7
8  '''
9  Created on 08 Nov 2014
10
11 @author: bob
12 '''
13 from workers.worker import Worker as worker
14 import scipy
15
16 GUESS_W = 10
17 MAX_TOL = 0.000001
18
19 class workerSimple(worker):
20     '''
21     A class to represent a worker to find the eigenmodes as
22     a function of  $K^2$ , sigma and g
23     '''
24     def __init__(self, Ksqr=[1],sigma=[1],g=[1],y0=[0.,1.],n=3,t0=0,tend=1):
25         '''
26         The constructor to set up the right parameters and to create
27         the ode's
28         '''
29         super(workerSimple, self).__init__(Ksqr, sigma, g, y0, n, t0, tend)
30         self.f = open(filename, 'w')
31         self.filename = filename
32         self.name = name
33     def search(self,Ksqrnum,sigmanum,gnum,n):
34         guess = GUESS_W/0.9
35         self.tempKsqrnum = Ksqrnum
36         self.tempsigmanum = sigmanum
37         self.tempgnum = gnum
38         endP = self.endPoint(guess)
39         while endP<0:
40             guess = guess*10
41             endP = self.endPoint(guess)
42         guesses = scipy.zeros(n)
43         positiveGuess = guess
44         for i in range(n):
45             taken = (-1)**i
46             endP = self.endPoint(guess)
47             shrinksize = 0.9
48             overCount = 1
```



```
49         while scipy.absolute(endP)>MAX_TOL:
50             previous = guess
51             guess = previous * shrinksize
52             endP = self.endPoint(guess)
53
54             if (scipy.absolute(endP)<MAX_TOL):
55                 break
56             if (teken*endP<0):
57                 overCount = overCount +1
58                 shrinksize = shrinksize + 9./(10**overCount)
59                 guess = positiveGuess
60                 guess = guess/shrinksize
61             elif (teken*endP>0):
62                 positiveGuess = guess
63         guesses[i] = guess
64         while (scipy.absolute(endP)<MAX_TOL ):
65             guess = positiveGuess
66             guess = guess/shrinksize
67             endP = self.endPoint(guess)
68             pass
69     return guesses
70
71
72
73
74
75
76
77
78
```

```
1  '''
2  Created on 15 Oct 2014
3
4  @author: bob
5
6  This task tests the worker class in more detail.
7  '''
8
9  from workers.worker2 import Worker2 as worker2
10 import scipy
11 from matplotlib import pyplot as plot
12
13 if __name__ == '__main__':
14     Ksqr = scipy.linspace(0.5, 1.5, 1).tolist()
15     sigma = scipy.linspace(0.5, 1.5, 10).tolist()
16     g = scipy.linspace(0.5, 1.5, 1).tolist()
17     n = 3
18     y0=[0.,1.];
19     t0=0
20     tend=1
21     h=0.01
22     eigenSys = worker2(Ksqr,sigma,g,y0,n,t0,tend,h)
23     data = eigenSys.taskNonParallel()
24     print data
25     plot.plot(data[:,1],data[:,3])
26     plot.show()
27
28
29
30
```

```
1  '''
2  Created on 15 Oct 2014
3
4  @author: bob
5
6
7  A test file for the worker class.
8
9  '''
10
11 from workers.workerSimple import workerSimple as workerSimple
12 import system.waveSystem as wave
13 import function.function as func
14 import integrators.rungeKutta as rK
15 import matplotlib.pyplot as plt
16
17 Ksqr = [1]
18 sigma = [1]
19 g = [1]
20 n = 10
21 y0 = 0
22 t0 = 1
23 tend = 1
24 h = 0.01
25
26 #wsqrnum = 0.3-0.285
27 wsqrnum = 0.015*(1+23./50)
28 wsqrnum = wsqrnum*(1+7./50)
29 wsqrnum = 0.001
30
31 class rho0(func.Function):
32     def __init__(self,Ksqr,sigma,g,wsqr):
33         func.Function.__init__(self)
34         self.Ksqr = Ksqr
35         self.sigma = sigma
36         self.g = g
37         self.wsqr = wsqr
38     def evaluate(self, x):
39         return (1+self.sigma*x)
40 # Create the two objects to represent the functions P and Q
41 class P(func.Function):
42     def __init__(self,Ksqr,sigma,g,wsqr):
43         func.Function.__init__(self)
44         self.Ksqr = Ksqr
45         self.sigma = sigma
46         self.g = g
47         self.wsqr = wsqr
48     def evaluate(self, x):
```

```
49     return self.wsqr*rho0(self.Ksqr,self.sigma,self.g,self.wsqr).eval
50 class Q(func.Function):
51     def __init__(self,Ksqr,sigma,g,wsqr):
52         func.Function.__init__(self)
53         self.Ksqr = Ksqr
54         self.sigma = sigma
55         self.g = g
56         self.wsqr = wsqr
57     def evaluate(self, x):
58         return -self.Ksqr*(rho0(self.Ksqr,self.sigma,self.g,self.wsqr).eva
59                             rho0(self.Ksqr,self.sigma,self.g,self.wsqr).der
60
61
62 def plot_ode(Ksqr, sigma, g, wsqr):
63     funcP = P(Ksqr,sigma,g,wsqr)
64     funcQ = Q(Ksqr,sigma,g,wsqr)
65     vgl = wave.WaveSystem(funcP,funcQ)
66     fe = rK.RungeKutta(vgl)
67     t_runge,soln_runge = fe.integrate(y0,t0,tend,h)
68     solution_runge = [soln_runge[i][0] for i in range(len(soln_runge))]
69     plt.plot(t_runge,solution_runge)
70     plt.show()
71     pass
72
73
74 def nbOfZerosnumber_of_zeros():
75     eigenSys = workerSimple(Ksqr,sigma,g,y0,n,t0,tend,h)
76     print eigenSys.zero_point_info(Ksqrnum=1, sigmanum=1, gnum=1, wsqrnum=
77
78 def find_eigen_mode():
79     w = wsqrnum
80     eigenSys = workerSimple(Ksqr,sigma,g,y0,n,t0,tend,h)
81     nb_of_zero , index_last_min , end_point , length_Set = eigenSys.zero_p
82     newW = w
83     while (nb_of_zero==0):
84         newW = (newW+0.0)/2
85         nb_of_zero , index_last_min , end_point , length_Set = eigenSys.ze
86     print nb_of_zero , index_last_min , end_point , length_Set
87
88     print newW
89     previousW = newW
90     counter = 0
91     while(abs(end_point)>0.001):
92         counter = counter +1
93         nb_of_zero_new , index_last_min_new , end_point_new , length_Set_r
94         print counter, newW , nb_of_zero_new , index_last_min_new , end_po
95         if (abs(end_point_new)<0.001):
96             break
```

```

97         if (nb_of_zero_new>=nb_of_zero):
98             # Vergroot w
99             previousW = newW
100             newW = newW*(1+((length_Set-index_last_min_new)+0.0)/index_last_min_new)
101         if (nb_of_zero_new < nb_of_zero):
102             # Nu weten we dat het vorige punt wel nog achter het nulpunt is
103             # gemiddelde tussen het slechte punt en het vorige goede punt.
104             newW = ((newW+previousW)+0.0)/2
105     print newW
106     plot_ode(Ksqr=1, sigma=1, g=1, wsqr=newW)
107
108 def task():
109     eigenSys = workerSimple(Ksqr,sigma,g,y0,n,t0,tend,h)
110     eigenSys.task()
111     pass
112
113 if __name__ == '__main__':
114     Ksqr = [1]
115     sigma = [1]
116     g = [1]
117     n = 10
118     y0 = 0
119     t0 = 1
120     tend = 1
121     h = 0.1
122     Ksqr=[1];sigma=[1];g=[1];y0=[0.,1.];n=10;t0=0;tend=1;h=0.01
123     #nbOfZerosnumber_of_zeros()
124     #find_eigen_mode()
125     #plot_ode(Ksqr=1, sigma=1, g=1, wsqr=wsqrnum)
126     task()
127
128
129
130
```