Link do strony:

<https://cac.cornell.edu/myers/teaching/ComputationalMethods/ComputerExercises/Pendulum/Pendulum.html>

# <nbformat>2</nbformat>

# <codecell>

import numpy

import scipy, scipy.integrate, scipy.optimize, scipy.interpolate

import pylab

# <codecell>

def EulerIntegratePendulum(InitialTheta=1,InitialOmega=0,InitialTime=0,FinalTime=16\*numpy.pi,dt=0.1):

""" EulerIntegratePendulum uses the Euler algorithm to integrate the differntial

equation for a pendulum

InitialTheta is the initial angle

InitialOmega is the initial angular velocity

InitialTime is the starting time

FinalTime is the ending time

dt is the timestep

This routine returns a tuple of the form (times, angles, angular frequency)

where times is a list of all of the times from InitialTime to FinalTime

in steps of dt

and angles is a list of the angle at each of those times """

# Start by making lists Theta, Omega, and Times which each contain

# the initial angle,angular frequency, and time as their only element

pass

# Make a loop which increments a time variable from InitialTime

# to FinalTime in steps of dt

# In each step of the loop append the new time to the Times list

# Generate and append the new Theta and Omega

pass

# Finally give a return statement which returns the desired tuple

pass

# <codecell>

def PendulumTimeUnit(mass,Length,g=(9.8,"m/s^2")):

""" Give this function the mass of the pendulum bob, the length of the pendulum,

and the acceleration due to gravity, and it will return what the unit of time is.

The syntax will be that all quantities should be given as tuples. The first element

of the tuple will be the numeric value of that constant, while the second will be a string

that represents the units"""

pass

# <codecell>

def PendulumEnergy(Theta,Omega):

""" Gives the energy of a pendulum of length 1 with g=1 for a given theta and omega"""

pass

# <codecell>

def SymplecticIntegratePendulum(InitialTheta=1,InitialOmega=0,InitialTime=0,FinalTime=16\*numpy.pi,dt=0.1):

""" SymplectIntegratePendulum uses the Symplectic Euler algorithm to integrate the differntial

equation for a pendulum

InitialTheta is the initial angle

InitialOmega is the initial angular velocity

InitialTime is the starting time

FinalTime is the ending time

dt is the timestep

This routine returns a tuple of the form (times, angles, angular frequency)

where times is a list of all of the times from InitialTime to FinalTime

in steps of dt

and angles is a list of the angle at each of those times """

# Start by making lists Theta, Omega, and Times which each contain

# the initial angle,angular frequency, and time as their only element

pass

# Make a loop which increments a time variable from InitialTime

# to FinalTime in steps of dt

# In each step of the loop append the new time to the Times list

# Generate and append the new Theta and Omega

pass

# Finally give a return statement which returns the desired tuple

pass

# <codecell>

fsolve=scipy.optimize.fsolve

# <codecell>

def findsqrt(num,estimate=1) :

""" Finds the square root of num, using estimate as the initial guess. This

uses the technique we suggest using for BackwardIntegratePendulum. The function

sqrtfun is defined next

"""

return fsolve(sqrtfun,estimate,(num))

# <codecell>

def sqrtfun(ans,anssquared) :

""" This function is equal to ans^2-anssquared:

its important property is that it vanishes when ans is the

square root of anssquared"""

return ans\*\*2-anssquared

# <codecell>

def findsqrt2(num,estimate=1):

""" Here is an alternate sqrt function -- this time using

a technique where you define a function inside your function"""

def mysqrtfun(ans) :

return ans\*\*2-num

return fsolve(mysqrtfun,estimate)

# <codecell>

def findsqrt3(num,estimate=1):

""" Here is the same algorithm as fidsqrt2, but using "lambda" functions"""

return fsolve(lambda x:x\*\*2-num,estimate)

# <codecell>

def findsqrt4(num,estimate=1):

""" This one uses a constructor function """

return fsolve(makesqrtfun(num),estimate)

# <codecell>

def makesqrtfun(num):

""" This is a constructor function which returns a function

of one variable which vanishes when its argument is equal

to the square root of num.

"""

def sfun(x):

return x\*\*2-num

return sfun

# <codecell>

def findnthroot(num,n,estimate=1) :

""" Finds (num)^(1/n) using the same algorithm as findsqrt

this demonstrates how to pass multiple arguments in fsolve"""

return fsolve(nthrootfun,estimate,(num,n))

# <codecell>

def nthrootfun(ans,num,n):

return ans\*\*n-num

# <codecell>

def BackwardIntegratePendulum(InitialTheta=1,InitialOmega=0,InitialTime=0,FinalTime=16\*numpy.pi,dt=0.1):

""" BackwardIntegratePendulum uses the Backward Euler algorithm to integrate the differntial

equation for a pendulum

InitialTheta is the initial angle

InitialOmega is the initial angular velocity

InitialTime is the starting time

FinalTime is the ending time

dt is the timestep

This routine returns a tuple of the form (times, angles, angular frequency)

where times is a list of all of the times from InitialTime to FinalTime

in steps of dt

and angles is a list of the angle at each of those times """

# Start by making lists Theta, Omega, and Times which each contain

# the initial angle,angular frequency, and time as their only element

pass

# Make a loop which increments a time variable from InitialTime

# to FinalTime in steps of dt

# In each step of the loop append the new time to the Times list

# Generate and append the new Theta and Omega

pass

# Finally give a return statement which returns the desired tuple

pass

# <codecell>

def BackwardPendulumFunction(NewThetaOmegaList,OldThetaOmegaList,dt):

""" This function returns a length 2 tuple. Both elements of the tuple are

zero if the theta and omega in the NewThetaOmegaList are the correct

theta and omega for the next step of the Backward Euler algorithm with

timestep dt.

"""

# For clarity, first unpack the lists which this function is called with

# This probably costs some overhead, but it makes reading the program so

# much easier that it is certainly worth it

pass

# <codecell>

def EulerIntegrate(DerivativeArrayFunc,InitialValues,times):

""" Uses the Forward Euler Algorithm to integrate forward in time the differential equation

dy/dt=DerivativeArrayFunc(y,t) with initial condition y=InitialValues.

The array of times in the argument gives the times at which things are evaluated.

This syntax is supposed to be indicative of the syntax used by odeint.

You may find it convenient to use numpy arrays instead of lists for the

various lists that you pass around.

"""

pass

# <codecell>

def PendulumDerivArray(vars,t):

""" variables are (theta, omega)

returns array([omega,-sin(theta)])

"""

pass

# <codecell>

def PendulumPeriod(s):

""" Gives the period of a pendulum with amplitude s.

"""

# First define a function integrand which is the function to be integrated

pass

# then call scipy.integrate.quad with that function

pass

# <codecell>

def PlotPeriod():

""" Plots the period of a pendulum as a function of amplitude"""

# First make a list of amplitudes

pass

# Then use a list comprehension to make a list of the periods

pass

# Then call pylab.plot and pylab.show

pass

# <codecell>

def PendulumDOmegaTDTheta(vars,theta):

"""

Define pendulum evolution law dvars/dtheta = dvarsdt / dthetadt

The input vars = [omega,t]

Return dvarsdt / omega

"""

pass

# <codecell>

def ODEPendulumPeriod(theta0):

"""

Find first quarter of period by starting at rest with angle theta0,

running forward until you pass zero (using PendulumDerivArray), then back up until

you hit zero (using PendulumDOmegaTDTheta).

(1) Pack initial conditions into vector y = [theta0, thetaDot0]

Velocity thetaDot0 starts at zero

(2) While theta > 0, integrate forward in small steps dt = 0.1

using scipy.integrate.odeint.

(a) odeint takes an array of time points to output, starting with

the original time. We just want time dt, so the array is

times=scipy.array([t,t+dt]). (odeint does not check to convert

lists into arrays on input, so we need to do it explicitly.)

(b) odeint returns a 'trajectory', a vector y(t) for each time

in the array.

y\_traj = scipy.integrate.odeint(PendulumDerivArray, y, times)

(c) The new y is thus y\_traj[1] (or, when using intermediate

time points, it is, y\_traj[-1], the last element in the trajectory

array)

(3) When theta first < 0,

(a) Pack current values into z0 = [thetaDot, t]

(b) Run odeint on PendulumDOmegaTDTheta with current z0 initial condition, for

an array thetas = scipy.array([theta, 0.]), store as z\_traj

(c) Pull out final t from z\_traj[-1]

(d) Return 4.0 times t for zero crossing

"""

# Pack evolving variables into vector of initial conditions

# Start at rest. Run to where theta crosses zero: 1/4 of a period

pass

# Run backwards to zero crossing

pass

# Period is four times time to first zero crossing

pass

# <markdowncell>

# Samples of alternative interface to odeint

# <codecell>

def smartodeint(func,y0,t,\*pargs,\*\*kargs):

"""This is a wrapper around odeint which returns an interpolating function instead

of a list of numbers. func is a function of (y,t) which returns a vector (or list or tuple)

of the rhs of the equation dy/dt=func(y,t). y0 is the initial conditions for y. t is a vector

which lists the times which are passed to odeint. odeint essentially uses only the first and

last element -- it makes its addaptive grid so that interpolation to these points works well.

I am using the same naming convention as odeint so that you can just run smartodeint with the

same code that you would normally call odeing. You can pass all of the arguments that odeint

accepts -- they just get forwarded directly to the integrator.

See the odeint documentation for more info.

For now it uses scipy's UnivariateSpline objects. These are very poorly documented.

In the future one might want to introduce a mechanism for passing arguments to the

UnivariateSpline generator, and a mechanism to get all of the info back from the ode integrator.

Sample Usage:

Autonomous ODE:

w1=smartodeint(lambda x,t:-x,[10],[0,10])

Coupled Autonomous ODE's:

w2=smartodeint(lambda x,t:(x[1],-x[0]),[10,0],[0,10])

Non-autonomous ODE's:

w3=smartodeint(lambda x,t:(x[1],-x[0]-0.05\*x[1]-0.1\*sin(0.9\*t)),[0,0],[0,500],mxstep=10000)

(the mxstep needs to be set, as it needs to subdivide a lot. An alternative notation is)

w3=smartodeint(lambda x,t:(x[1],-x[0]-0.05\*x[1]-0.1\*sin(0.9\*t)),[0,0],arange(501))

"""

values=[]

#print(len(t))

def wrappedfunc(y,t) :

yw=y.copy()

values.append((t,yw))

return func(y,t)

result=scipy.integrate.odeint(wrappedfunc,y0,t,\*pargs,\*\*kargs)

#print(len(values))

values.reverse() # reverse the list so that later (more accurate) points come first

valdict=dict(values) # convert to a dictionary to remove duplicate time values

vallist=list(valdict.items()) # convert to a list so we can sort it

vallist.sort() # sort the list

#print(vallist)

#valarray=scipy.array(vallist) # convert to an array so we can slice it

xvals=[x for x,y in vallist] # slice out the x-values

yvals=scipy.array([y for x,y in vallist]) # slice out the y-values

#xvals=vallist[:][0]

#yvals=vallist[:][1]

#print(xvals)

#print(yvals)

interpfun=[scipy.interpolate.UnivariateSpline(xvals,yv,s=0) for yv in scipy.transpose(yvals)]

return interpfun

# <codecell>

norangeerror="Need range for plot"

# <codecell>

def plotfunction(func,xvals=None,gridpoints=10,maxcurv=0.01,returnlist=False,show=True,maxrecur=10,nospline=False):

"""Plots the function of one variable func on the range specified by xval, using a

recursive algorithm with at least gridpoint points. maxcurv is a parameter which

specifies when a segment is subdivided.

If func returns a tuple (or list), it will plot multiple curves. [For example, if func returns

two elements, it will plot two curves.]

func can also be a list (or tuple) of functions -- in which case it will plot multiple curves

-- ie same functionality as if you had one function returning a tuple.

If this is called with a UnivariateSpline object, it just plots it using the knots.

set nospline=True if you want to force it to use the subdivision algorithm on a spline object.

Set returnlist=True if you want the list of plotted points.

Future: Add some sort of handling for singularities.

Sample usage:

plotfunction(sin,[0,20])

plotfunction(lambda x:sin(x\*\*2),[0,18])

plotfunction(lambda x:sin(x),cos(x),[0,18])

plotfunction((lambda x:x,lambda x:-x),[-2,2])

w3=smartodeint(lambda x,t:(x[1],-x[0]-0.05\*x[1]-0.1\*sin(0.9\*t)),[0,0],[0,500],mxstep=10000)

plotfunction(w3)

"""

if not(nospline) and hasattr(func,"get\_knots") : # we have a spline object -- just evaluate at knots

knots=func.get\_knots()

vals=func(knots)

pylab.plot(knots,vals)

pntsarray=(knots,vals)

elif hasattr(func,"\_\_iter\_\_"): # we have a list of functions -- map plotfunction over them

pntsarray=[plotfunction(f,xvals,gridpoints,maxcurv,returnlist,False,maxrecur,nospline) for f in func]

else : # we have a boring old function -- run subdivision algorithm

if xvals == None: # need xvals to plot the function

# print("Need range for plot")

raise norangeerror

# We throw an exception so the user knows that they need to include a range

# return

maxx=max(xvals)

minx=min(xvals)

biggeststep=(1.\*(maxx-minx))/gridpoints

currentx=minx

currenty=scipy.array(func(currentx))

pnts=[(currentx,currenty)]

#print(xvals)

xlist=scipy.array(xvals)

xlist.sort()

#print(xlist)

for x in xlist[1:] :

y=scipy.array(func(x))

addpoint(func,pnts,(currentx,currenty),(x,y),biggeststep,maxcurv,maxrecur,0)

currentx,currenty=x,y

xvals=[x for x,y in pnts]

yvals=[y for x,y in pnts]

pylab.plot(xvals,yvals)

pntsarray=(xvals,scipy.array(yvals))

if show : pylab.show()

if returnlist :

return pntsarray # this should be formatted so you can plot it with plot(\*pntsarray)

# <codecell>

def addpoint(func,pnts,old,new,biggeststep,maxcurve,maxrecur,recur):

""" Private function used by plotfunction

used to recursively subdivide region """

x=new[0]

y=new[1]

oldx=old[0]

oldy=old[1]

tryx=(x+oldx)/2.

tryy=scipy.array(func(tryx))

curv=scipy.array([abs(y+oldy-2.\*tryy)/(x-oldx)])

#print(curv)

#print(("x=",x,"tryx=",tryx,"oldx=",oldx,"curv=",curv,"tryx-x",tryx-x,"biggeststep",biggeststep))

if ((curv.max()<maxcurve) and (abs(tryx-x)<biggeststep)) or recur>maxrecur :

pnts.append((tryx,tryy))

pnts.append((x,y))

return

else :

#print((curv,maxcurve))

addpoint(func,pnts,old,(tryx,tryy),biggeststep,maxcurve,maxrecur,recur+1)

addpoint(func,pnts,(tryx,tryy),new,biggeststep,maxcurve,maxrecur,recur+1)

return

# <markdowncell>

# If you want to play with the object oriented integrator: ode

# <codecell>

def revPendulumDerivArray(t,vars):

# variables are (theta, omega)

# returns array([-omega,sin(theta)])

pass

# <markdowncell>

# Copyright (C) Cornell University

# All rights reserved.

# Apache License, Version 2.0