1 Lots of information about Plots & Animations

Instructors: Claus Fuhrer and Alexandros Sopasakis, LTH- Lund University

1.0.1 Simple 2D plots

We begin as usual by importing the necessary libraries

In [1]: from numpy import *
   from matplotlib.pyplot import *
   from pylab import *

plus the library for plotting in the notebook (instead of outside)

In [2]: %matplotlib inline

In [3]: x = linspace(0,pi,10)
y = x**2 # is our example function

In [8]: # in subplot provide:
   # rows, columns and plot number
   subplot(3,1,1)
   plot(x,y,'ro')
   subplot(3,1,2)
   plot(x,y,'g')
   subplot(3,1,3)
   plot(y,x,'bs')

Out[8]: [<matplotlib.lines.Line2D at 0x1a812fd6a58>]

To save the above figure use the command savefig('filename.png', dpi = 200)
Fonts and labeling  Global font change. Use the matplotlib.rcParams.update(**d)

In [7]: d = {'font.size':8, 'font.family':'sans'}
   matplotlib.rcParams.update(**d)

In [9]: plot(x,y,lw=4,ls='-.',marker='p',markersize=10,
   markerfacecolor="#15cc55")

Out[9]: [<matplotlib.lines.Line2D at 0x1a8130933c8>]

1.0.2  Advanced 2D plots

Subplots and other specific features within each such plot  We provide examples of scatter, step and bar type plots

In [14]: fig,ax = subplots(4,1,figsize=(10,8))
   n = array([0,1,2,3,4,5])

   ax[0].scatter(x,x+.25*np.random.randn(len(x)))
   ax[1].plot(x,y,'rp')
   ax[2].step(n,n**2,lw=2)
   ax[3].bar(n,n**2,width=.5, alpha = .5)

   # you can go back and add features to a given axis.
   ax[0].set_title("this is a scatter plot")

Out[14]: Text(0.5,1,'this is a scatter plot')
2 Plot within a plot

This is how to create a plot within a plot

In [18]: matplotlib.rcParams.update({'font.size':10})
   # adjusting again the size of the axis fonts

fig = figure()

   # for axes location: x start pt, y start pt, width, height
ax1 = fig.add_axes([.1,.1,.8,.8])    # first create the first plot
ax2 = fig.add_axes([.2,.5,.2,.3])    # now the axes for the other plot

ax1.plot(x,x**2+3,label='x^2+3')
ax2.plot(x,sin(x**2), color='r',label='sin(x^2)')

ax1.set_title('Plot within a plot')
ax1.legend(loc=4)
   # Info about legend location:
Range and scale of axis  Here we will produce 3 figures one next to the other.
Each figure will contain plots with different features:
multiple plots and axis limits, log scale, grid + latex labeling

In [16]: fig,ax=subplots(1,3,figsize=(12,4))
   # three plots at once
   ax[0].plot(x,x**2, x,x**3, x,x**4)
   ax[0].set_title("a plot")

   # manually adjust x and y axis limits
   ax[0].set_xlim([2,3.5])
   ax[0].set_ylim([0,60])

   ax[1].plot(x,sin(x**2))
   ax[1].set_xscale("log") # makes x axis to be log scale
   ax[1].set_title("log scale on x")
   ax[2].plot(x,x**3,linestyle='--')
   ax[2].set_xticks([1,2,3,4,5])
   ax[2].set_xticklabels([r'$\alpha$',r'$\beta$']) # latex symbols in axis
   ax[2].grid(True,color='b',alpha=.5,ls='dashed',lw=.5) # grid properties
Polar plots (use: polar = True)  For plotting in polar coordinates we simply need to provide the angle and the radius.

In the example below we plot the angle phi between 0 and 2*pi while the radius is changing linearly with respect to the angle as follows: phi(r) = r

In [19]: fig=figure()
    # again the axes loc: start x pt, start y pt,
    # then width and finally height
    # polar = true needed for polar plots
    ax = fig.add_axes([0.0,0.0,.6,.6],polar = True)
    phi = linspace(0,2*pi,100)
    r = phi
    ax.plot(phi,r,color='blue',lw=4)

Out[19]: [<matplotlib.lines.Line2D at 0x1a8151bfc18>]}
Histograms (use: hist(n))  Let us first create the data to be used in the histograms

In [22]: n = np.random.randn(100000)

In [25]: fig,ax = subplots(1,2,figsize=(12,4))
ax[0].hist(n)
# use below for tighter limits on axes
# ax[0].set_xlim(min(n),max(n)) # or even (-4, 4)
# to suppress output put a ; at the end below
ax[1].hist(n,cumulative=True,bins=50)

Out[25]: (array([3.0000e+00, 3.0000e+00, 3.0000e+00, 3.0000e+00, 1.3000e+01,
  1.8000e+01, 3.6000e+01, 6.4000e+01, 1.2400e+02, 2.0300e+02,
  3.3800e+02, 5.8100e+02, 9.8900e+02, 1.5940e+03, 2.4720e+03,
  3.6770e+03, 5.3470e+03, 7.5220e+03, 1.0450e+04, 1.4039e+04,
  1.8423e+04, 2.3631e+04, 2.9337e+04, 3.5876e+04, 4.2893e+04,
  4.9873e+04, 5.6970e+04, 6.3721e+04, 7.0383e+04, 7.6191e+04,
  8.1355e+04, 8.5840e+04, 8.9469e+04, 9.2341e+04, 9.4612e+04,
  9.6277e+04, 9.7541e+04, 9.8406e+04, 9.8982e+04, 9.9408e+04,
  9.9656e+04, 9.9790e+04, 9.9876e+04, 9.9939e+04, 9.9966e+04,
  9.9982e+04, 9.9991e+04, 9.9996e+04, 9.9998e+04, 1.0000e+05]),
array([-4.66639100e+00, -4.48708612e+00, -4.30778125e+00, -4.12847638e+00,
  -3.94917150e+00, -3.76986663e+00, -3.59056176e+00, -3.41125688e+00,
  -3.23195201e+00, -3.05264713e+00, -2.87334226e+00, -2.69403739e+00,
  -2.51473251e+00, -2.33542764e+00, -2.15612277e+00, -1.97681789e+00,
  -1.79751302e+00, -1.61820815e+00, -1.43890327e+00, -1.25959840e+00,
  -1.08029353e+00, -9.00988653e-01, -7.21683780e-01, -5.42378907e-01,
  -3.63074033e-01, -1.83769160e-01, -4.64286166e-03, 1.74840587e-01,
  3.54145646e-01, 5.33450334e-01, 7.12755208e-01, 8.92060081e-01,
  1.07136495e+00, 1.25066983e+00, 1.42997470e+00, 1.60927957e+00,
  1.78858445e+00, 1.96788932e+00, 2.14719420e+00, 2.32649907e+00,
  2.50580394e+00, 2.68510882e+00, 2.86441369e+00, 3.04371856e+00,
  3.22302344e+00, 3.40328381e+00, 3.58163318e+00, 3.76093806e+00,
  3.9402493e+00, 4.11954780e+00, 4.29885268e+00]),
<a list of 50 Patch objects>)
3 3D plots and animations

In [4]: from scipy import *
from numpy import *
from matplotlib.pyplot import *
%matplotlib inline
from mpl_toolkits.mplot3d.axes3d import Axes3D

3.0.1 Wireframes, Surfaces and colorbars in 3D

We first define an appropriate function in 3D

In [5]: def f(x,y):
    return 2.7 -2*cos(y)*cos(x)-.7*cos(2*pi*.5-2*y)

Now create the X, Y and Z points for this plot using the meshgrid command (same as in Matlab if you are familiar)

In [6]: p_m = linspace(0,2*pi,100)
    p_p = linspace(0,2*pi,100)
    X,Y = meshgrid(p_p,p_m)
    Z = f(X,Y)

In the example below we create both a surface as well as a wireframe for the same figure

In [9]: fig = figure(figsize=(12,6))

    # note the need for projection='3d' below!
    ax1 = fig.add_subplot(1,2,1,projection='3d')
    p = ax1.plot_wireframe(X,Y,Z,rstride=2,cstride=17,lw=1)

    ax2 = fig.add_subplot(1,2,2,projection='3d')
    q = ax2.plot_surface(X,Y,Z,cmap=matplotlib.cm.coolwarm)
    # if desired you can even add a colorbar
    #cb = fig.colorbar(q, shrink=0.5)
3.0.2 Contour plots and angle of viewing in 3D

In [10]: fig = figure(figsize=(14,6))

ax = fig.add_subplot(1,2,1, projection='3d')
# rstride, cstride are used to step in arrays
# alpha for transparency
ax.plot_surface(X, Y, Z, rstride=4, cstride=4, alpha=0.1)

# the offset says where to display the contour
cset = ax.contour(X,Y,Z, zdir='z', offset=0)
cset = ax.contour(X,Y,Z, zdir='x',offset = -pi)

# changing the default angle of viewing the plot
ax.view_init(70,75)

ax1 = fig.add_subplot(1,2,2, projection='3d')
ax1.plot_wireframe(X, Y, Z, rstride=2, cstride=15)
cset = ax1.contour(X, Y, Z, zdir='z', offset=0) # the offset says where to display the
cset = ax1.contour(X,Y,Z, zdir='x',offset = -pi)
ax.view_init(10,75) # angle of viewing the plot from
4 Animations

Matplotlib uses a special library for generating animations for sequences of figures.

from matplotlib import animation

Furthermore you can use the FuncAnimation function to create a movie file from sequences of figures. This function uses inputs such as: fig, a figure canvas, func, a user defined function to update the figure, init_func, a user defined function to setup the figure, frame, the number of frames to generate, and blit, which updates parts of the frame changed. Below are some examples of those functions:

def init(): # setup figure
def update(frame_counter): # update figure for new frame

animvideo = animation.FuncAnimation(fig, update, init_func=init, frames=200, blit=True)

animvideo.save('videofile.mp4', fps=20) # fps = frames per second

In [1]: #
# RESTART THE NOTEBOOK: the matplotlib backend can only be selected before pylab is imported
# (e.g. Kernel > Restart)
#
import matplotlib
matplotlib.use('Qt4Agg') # or for example MacOSX
import matplotlib.pyplot as plt
import numpy as np

In [2]: """
============
The double pendulum problem
============

This animation illustrates the double pendulum problem.
"""
from numpy import sin, cos
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate as integrate
import matplotlib.animation as animation

G = 9.8  # acceleration due to gravity, in m/s^2
L1 = 1.0  # length of pendulum 1 in m
L2 = 1.0  # length of pendulum 2 in m
M1 = 1.0  # mass of pendulum 1 in kg
M2 = 1.0  # mass of pendulum 2 in kg

def derivs(state, t):
    dydx = np.zeros_like(state)
    dydx[0] = state[1]

    del_ = state[2] - state[0]
    den1 = (M1 + M2)*L1 - M2*L1*cos(del_)*cos(del_)
    dydx[1] = (M2*L1*state[1]*state[1]*sin(del_)*cos(del_) +
               M2*G*sin(state[2])*cos(del_) +
               M2*L2*state[3]*state[3]*sin(del_) -
               (M1 + M2)*G*sin(state[0]))/den1


    den2 = (L2/L1)*den1
               (M1 + M2)*G*sin(state[2])*cos(del_) -
               (M1 + M2)*L1*state[1]*state[1]*sin(del_) -
               (M1 + M2)*G*sin(state[0]))/den2

    return dydx

# create a time array from 0..100 sampled at 0.05 second steps
dt = 0.05
T = np.arange(0.0, 20, dt)

# th1 and th2 are the initial angles (degrees)
# w10 and w20 are the initial angular velocities (degrees per second)

th1 = 120.0
w1 = 0.0
th2 = -10.0
\[ w_2 = 0.0 \]

# initial state
state = np.radians([th1, w1, th2, w2])

# integrate your ODE using scipy.integrate.
y = integrate.odeint(derivs, state, t)

x1 = L1*sin(y[:, 0])
y1 = -L1*cos(y[:, 0])

x2 = L2*sin(y[:, 2]) + x1
y2 = -L2*cos(y[:, 2]) + y1

fig = plt.figure()
ax = fig.add_subplot(111, autoscale_on=False, xlim=(-2, 2), ylim=(-2, 2))
ax.grid()

line, = ax.plot([], [], 'o-', lw=2)
time_template = 'time = %.1fs'
time_text = ax.text(0.05, 0.9, '', transform=ax.transAxes)

def init():
    line.set_data([], [])
time_text.set_text('')
return line, time_text

def animate(i):
    thisx = [0, x1[i], x2[i]]
    thisy = [0, y1[i], y2[i]]

    line.set_data(thisx, thisy)
time_text.set_text(time_template % (i*dt))
return line, time_text

ani = animation.FuncAnimation(fig, animate, np.arange(1, len(y)),
                               interval=25, blit=True, init_func=init)

# ani.save('double_pendulum.mp4', fps=15)
plt.show()
import matplotlib
matplotlib.use('Qt4Agg') # or for example MacOSX
from __future__ import division
from scipy import *
from matplotlib.pyplot import *
import numpy
from numpy import linalg as LA
from pylab import *
from matplotlib.widgets import Slider, Button, RadioButtons

#%matplotlib inline

eps = .0000000000000000000001

def Bspline(t,v,dt):
    #Evaluate the piecewise cubic B-spline curve at intervals of dt
    #knots t(i) and de-boor points v(i)
    # IMPORTANT:
    #(1) there should be 4 more knots t(i) than points v(i).
    #(2) This function evaluates the B-spline curve over [t_4, t_{m-3}],
    # where m is the number of knots. (Note the spline is not
    # well-defined outside this interval.)
    m = len(t) #number of knots
    i = 4       #index of first knot
    ptNum = 0   #counter for points generated
    q=[]
    for u in arange(t[3],t[m-3]+dt,dt):
        # check if u value has moved to the next knot interval
        # include small tolerance on knots to avoid round-off error in comparisons.
        while (u>(t[i]+eps)):
            i+=1
            # Now evaluate the spline at u using the deBoor algorithm.
            # Start with the relevant control points.
            # w used here to simplify indices.
            w = i-4
            #print("w={}".format(w))
            qq = zeros(len(v))
            for j in arange(1,5,1):
                qq[j-1]=v[w+j-1]
                #print("u={}, qq[{}]={}".format(u, j-1,v[w+j-1]))
            for j in arange(1,4,1):
                #print(j)
                for k in arange(1,4-j+1,1):
                    #print("k={}".format(k))
                    qq[k-1] = ((t[w + k + 4-1] - u)/(t[w + k + 4-1] - t[w + k + j-1])) * qq[k]
                    #print("qq[{}]={}", t41={}, t2={}, u={}".format(k-1,qq[k-1],t[w + k + 4-1],t[w + k + j-1],u))
            #Create vector of points on the B-spline curve.
q.append(qq[0])
# print("u={}, q[{}]={}".format(u,0,qq[0]))
ptNum +=1
return q

# B-spline example
# Evaluate and plot a cubic B-spline curve;
# Needs B-spline.m function (included below)

#fig1=figure()
subplots_adjust(left=0.10, bottom=0.25)
grid()
# Evenly spaced knots, with 4 more knots than y-values
x = [0, 0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 8, 8, 8]  # nodes
db = [2, 1, 3, 8, 7, 1, 3, 2, 3, 4, 5]  # deBoor points
y0 = db[1]
y1 = db[2]
y2 = db[3]
y3 = db[4]
y4 = db[5]
y5 = db[6]
y6 = db[7]
y7 = db[8]
y8 = db[9]
y9 = db[10]
dx = .1  # how fine to plot resulting curve
n = len(x)

# evaluate at a number of points for plotting
y = Bspline(x, db, dx)

# Plot of B-spline has t in [t_4, t_{n-3}]
u = arange(x[3],x[-3]+dx,dx)
t, = plot(x[2:-2], db,'k--',label='Control Pts')
s, = plot(u, y, 'r-',label='B-spline')
legend(loc='upper right');
w, = plot([0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 8], db,'bo',markersize=12)
xlabel('nodes')
ylabel('Spline y')
eps=.2
axis([x[3]-eps, x[n-2]+eps, min(db)-eps, max(db)+eps])
axcolor = 'lightgoldenrodyellow'
loc = .20
sl=[]
for i in range(0,5):#len(x)):
    ax = axes([.10, loc, 0.30, 0.02], axisbg=axcolor)
loc -= .04
text = "x[{}].format(i+1)
sl.append(Slider(ax, text, 0.0, 9.0, valinit=db[i+1]))  # node t[3]

loc = .20
for i in range(5, 10):
    ax = axes([.60, loc, 0.30, 0.02], axisbg=axcolor)
    loc -= .04
    text = "x[{}].format(i+1)
    sl.append(Slider(ax, text, 0.0, 9.0, valinit=db[i+1]))

    def update(val):
        for i in range(0, 10):
            db[i+1] = sl[i].val
            ll = Bspline(x, db, dx)[x[0]]
            s.set_ydata(ll)
            t.set_ydata(db)
            w.set_ydata(db)
            grid()
            draw()

for i in range(0, 10):
    sl[i].on_changed(update)

show()