Matplotlib

Allgemeines

Matplotlib braucht NumPy

2 Schichten für GUI:

- Renderer: erledigt das Zeichnen
  Standard: AGG lib = Anti-Grain Geometry
- Canvas: hierauf wird gezeichnet

bei Matplotlib gibt es ein einfaches (prozedurales) und ein komplizierteres objektorientiertes Interface. Die Beispiele im Internet und in Büchern benutzen entweder die eine oder die andere Methode, so dass man für ein bestimmtes Ziel eine verwirrende Vielzahl von Lösungen finden kann.
Das einfache Interface pyplot erlaubt sehr einfache Programme.

Genauer:

- pyplot = prozedurales Interface
- pylab = Modul das als Ersatz für Matlab dienen soll (enthält matplotlib.pyplot + NumPy).
- matplotlib: die ganze Bibliothek, erlaubt prozedurales und objektorientiertes Arbeiten.

Äquivalente Imports:

```
import matplotlib.pyplot as plt
from numpy import sin, exp, linspace, arange, pi
```

macht das Gleiche wie

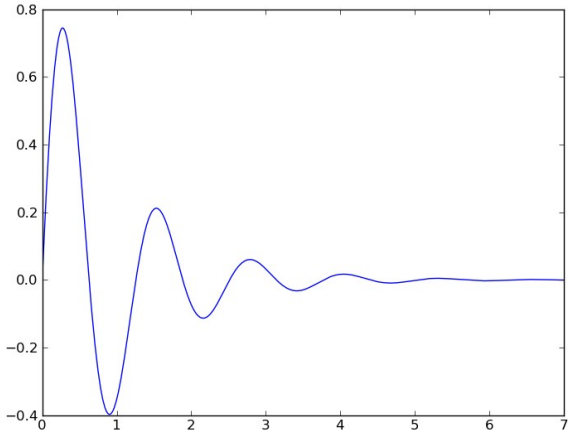
```
import pylab as plt
from pylab import sin, exp, linspace, pi
```

Die Autoren von Matplotlib empfehlen den Gebrauch von pylab nicht. Es ist besser zusehen welche Funktionen von Numpy und welche von pyplot stammen.

Eine Kurve mit Pyplot

Dieses Beispiel zeigt eine gedämpfte Sinusschwingung:

```
import matplotlib.pyplot as plt
from numpy import sin, exp, linspace

x=linspace(0.0, 7.0, 1000)
y= sin(5*x)*exp(-x)

plt.plot(x, y)
plt.show()
```



Die erste Zeile importiert das pyplot – Interface als Objekt plt.
Für mathematische Aufgaben ist das Modul numpy gedacht. In der zweiten Zeile werden davon 3 Funktionen importiert.

Mit linspace kann ein Intervall (hier von 0 bis 7) in beliebig viele (hier 1000) Teile zerlegt und als Vektor (Array) dargestellt werden. Dies erlaubt eine schnelle und einfache Berechnung der Funktionswerte. Die Sinus- und Exponentzialfunktionen von numpy können bei Bedarf Vektoren verarbeiten, hier werden also in der Zeile

```
y= sin(5*x)*exp(-x)
```

die 1000 Werte des y-Vektors berechnet.

Mit dem Befehl plt.plot(x,y) werden diese dann als Kurve dargestellt.

Damit das Diagramm sichtbar wird, muss der Befehl plt.show() ausgeführt werden.

Mehrere Kurven

Will man **mehrere Kurven** in einem Diagramm darstellen, braucht man die plot-Funktion nur mehrmals aufzurufen:

```
import matplotlib.pyplot as plt
from numpy import sin, exp, linspace

x=linspace(0.0, 7.0, 1000)
y1 = sin(5*x)*exp(-x)
y2 = y1* 0.5

plt.plot(x, y1)
plt.plot (x, y2)
plt.show()
```

Mehrere Kurven mit unterschiedlichen Amplituden:

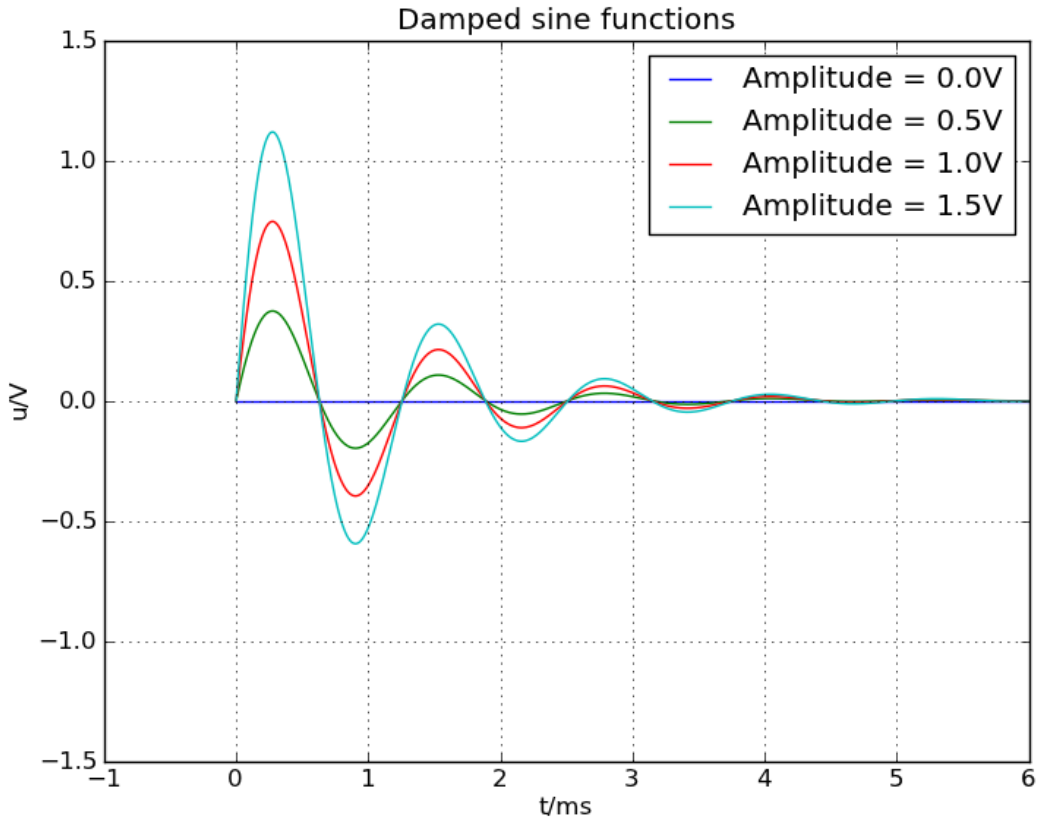
```
import matplotlib.pyplot as plt
from numpy import sin, exp, linspace, arange, pi

x=linspace(0.0, 2*pi, 1000)

for a in arange(0,2, 0.5):
    y = a * sin(5*x)*exp(-x)
    plt.plot(x, y)

plt.show()
```

Die Kurven werden alle gezeichnet da die Hold-Eigenschaft defaultmäßig auf True gesetzt ist. Mit plt.hold(False) wird nur eine (die letzte) Kurve gezeichnet.

Kosmetik

plt.grid(True)	**Gitter** einschalten
plt.xlim(-1.0, 6.0) plt.ylim(-1.5, 1.5)	Grenzen **xmin, xmax** setzen Grenzen ymin, ymax setzen
plt.xlabel("t/ms") plt.ylabel("u/V")	**Label für x- und y-Achse**
plt.title (" ..... ")	Diagramm-Titel
plt.plot(x, y, label = ".....") ..... plt.legend()	Label für die **Legende** für jede Kurve einzeln festlegen Legende einfügen

```
import matplotlib.pyplot as plt
from numpy import sin, exp, linspace, arange, pi

plt.hold(True)    # is anyway the default

x=linspace(0.0, 2*pi, 1000)

for a in arange(0,2, 0.5):
    y = a * sin(5*x)*exp(-x)
    plt.plot(x, y, label = "Amplitude = " + str(a)+ "V" )


plt.grid(True)
plt.xlim(-1.0, 6.0)
plt.ylim(-1.5, 1.5)
plt.xlabel("t/ms")
plt.ylabel("u/V")
plt.title ("Damped sine functions")
plt.legend()        # plots legend with labels defined in plot command

plt.show()
```



Linienstil

Angabe mit Kürzeln:

plt.plot (x,y,"m") plt.plot (x,y,"#0000FF")	**Farbe** „m“ = Magenta Farbcodes: b c g m r y, k = black, w = white Farbcode RGB (wie bei HTML)
plt.plot (x,y,"--")	**Linienstil** - solid -- dashed -. dash-dot : dotted
plt.plot (x,y,"o")	**Punktstil** . Point marker , pixel o circle s square * star **+ cross**
plt.plot (x,y,"r-+")	Red, solid line, cross markers

Angabe mit keyword – Argumenten:

```
plt.plot (x,y, color = "red", linestyle="solid", linewidth=4, marker="o",
   markerfacecolor="blue", markeredgecolor = "black", markersize=10 )
```

Diagramm speichern
Speichern mit den Default-Werten:

```
plt.savefig ("test.png")
```

Mit Angabe eines dpi-Wertes:

```
plt.savefig ("test.png", dpi = 200)
```

Polar-Diagramme

```
import matplotlib.pyplot as plt
from numpy import sin, exp, linspace, pi

alpha=linspace(0.0,20*pi, 1000)
r = 3 * exp(-alpha/10)
plt.polar(alpha, r)
plt.show()
```

Achtung: der Winkel wird im Diagramm „mathematisch“ interpretiert, also in Radianten.
Die Beschriftung des Diagramms ist aber in Grad.

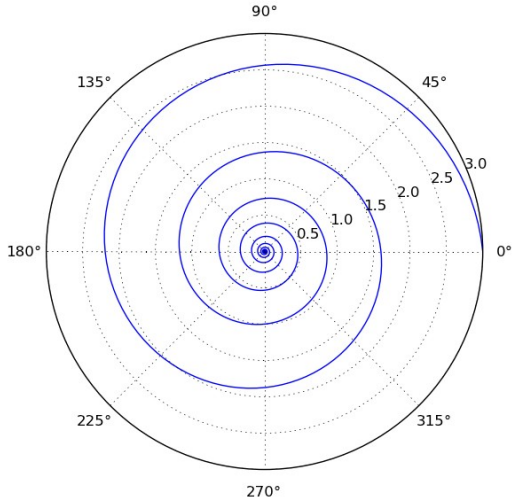


Diagramme auf die „Pythonische“ Art (objektorientiert)

Objekthierarchie:

- FigureCanvas: Container für eine Figure Instanz
- Figure: Container für eine oder mehrere Axes Instanzen
- Axes: die Diagrammfäche

Das erste Beispiel objektorientiert:

Statt

```
plt.plot(x, y)
```

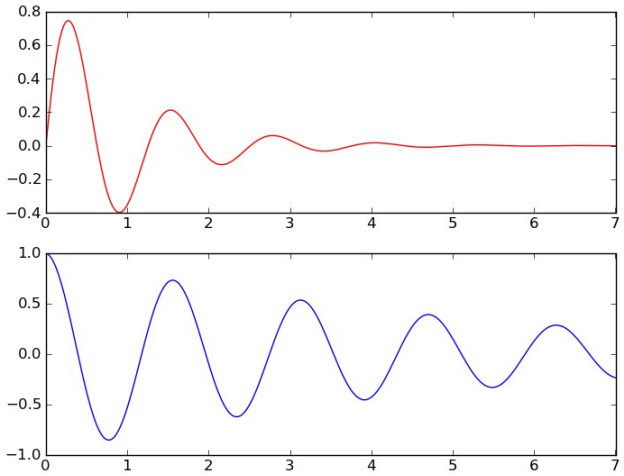
kommt nun:

```
fig = plt.figure()
ax = fig.add_subplot(111)
l, = plt.plot(x, y)
```

- fig = plt.figure() erzeugt die Zeichenfläche für einen oder mehrere Subplots
- ax = fig.add_subplot(111) erzeugt darin eine leere Diagrammfläche mit Achsen + Skala (mit Defaultwerten)
- l, = plt.plot(x, y) zeichnet die xy-Kurve als Line2D-Objekt und gibt eine Referenz l darauf zurück. Man kann alle für ein Line2D-Objekt erlaubten Operationen darauf anwenden und so das Aussehen der Kurve ändern,
  z.B. l.set_color("red"), dies geht auch nach dem plot(x,y).

Diagramm mit 2 Subplots

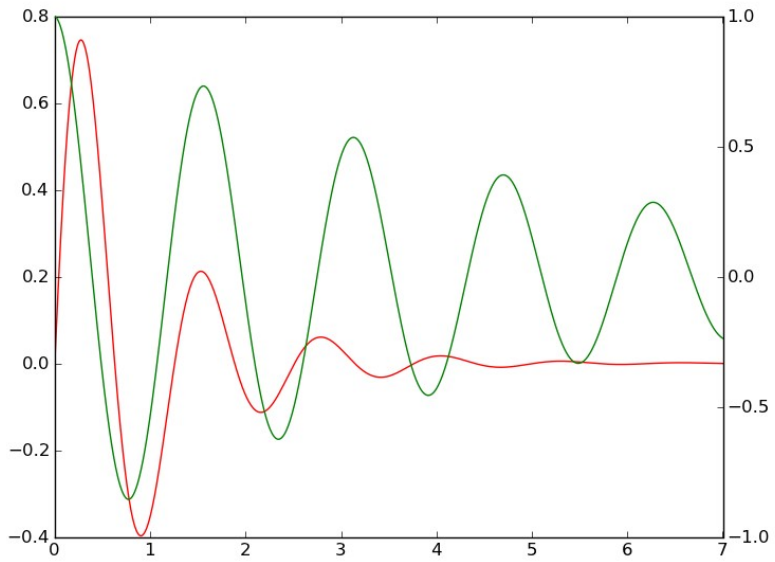
```
import matplotlib.pyplot as plt
from numpy import sin, cos, exp, linspace

# produce data
x=linspace(0.0, 7.0, 1000)
y1 = sin(5*x)*exp(-x)
y2 = cos(4*x)*exp(-x/5)

# figure = drawing area
fig = plt.figure()

# 2 subplots, one for each curve
ax1 = fig.add_subplot(211)  # 2 rows, 1 col, fig1
l1, = ax1.plot(x, y1)
l1.set_color("red")

ax2 = fig.add_subplot(212)  # 2 rows, 1 col, fig2
l1, = ax2.plot(x, y2)
l1.set_color("blue")

plt.show()
```



Mehrere unabhängige Diagramme erzeugen

```
import matplotlib.pyplot as plt
from numpy import sin, cos, exp, linspace

# produce data
x=linspace(0.0, 7.0, 1000)
y1 = sin(5*x)*exp(-x)
y2 = cos(4*x)*exp(-x/5)

# Create 2 figures each containig 1 plot

fig1 = plt.figure()
ax1 = fig1.add_subplot(111)  # 2 rows, 1 col, fig1
l1, = ax1.plot(x, y1)
l1.set_color("red")

fig2 = plt.figure()
ax2 = fig2.add_subplot(111)  # 2 rows, 1 col, fig2
l1, = ax2.plot(x, y2)
l1.set_color("blue")

plt.show()
```

Diagramm mit zusätzlicher y-Achse



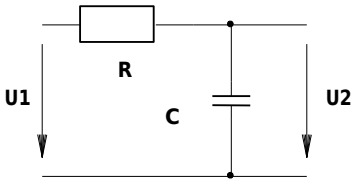

```
import matplotlib.pyplot as plt
from numpy import sin, cos, exp, linspace

# produce data
x=linspace(0.0, 7.0, 1000)
y1 = sin(5*x)*exp(-x)
y2 = cos(4*x)*exp(-x/5)

# figure = drawing area
fig = plt.figure()

# first plot
ax1 = fig.add_subplot(111)
l1, = ax1.plot(x, y1)
l1.set_color("red")
```

```
# second plot
ax2 = ax1.twinx()          # same x-scale as other plot
l2, = ax2.plot(x, y2)
l2.set_color("green")

plt.show()
```

Eigentlich werden hier zwei Diagramme mit gemeinsamer x-Achse übereinander gezeichnet.

Bode-Diagramme

Es soll die Durchlasskurve eines RC-Tiefpasses dargestellt werden.



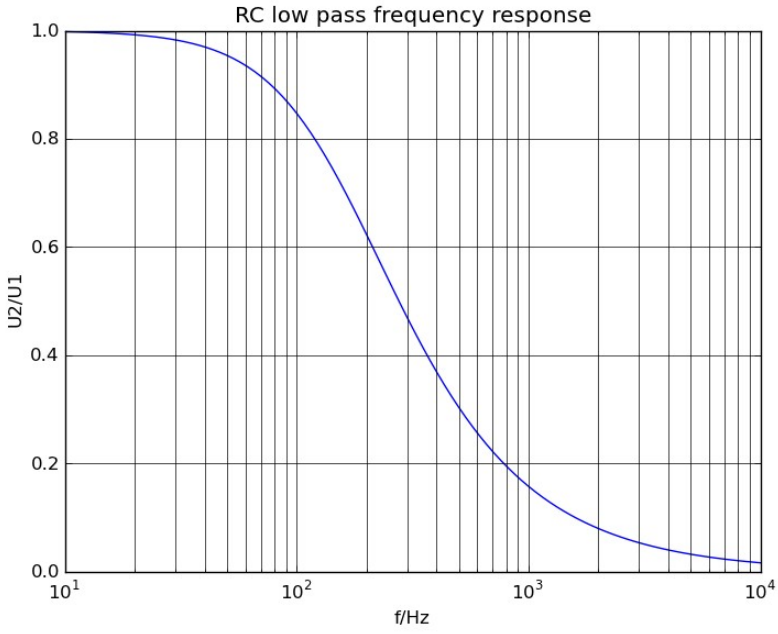
$$\bar{F} = \frac{\bar{Z}_C}{\bar{Z}_R + \bar{Z}_C} \quad \text{mit} \quad \bar{Z}_R = R \quad \bar{Z}_C = \frac{1}{j\,\omega\,C}$$

$$\bar{F} = \frac{1}{1 + j\,\omega\,RC}$$

Der komplexe Frequenzgang errechnet sich nach der Spannungsteilerformel mit den komplexen Impedanzen.

Als Funktion der Frequenz geschrieben: $\bar{F} = \frac{1}{1 + j\,2\pi\,f\,RC}$

Die Durchlasskurve ist der Betrag von $\bar{F}$ als Funktion der Frequenz.




```
import matplotlib.pyplot as plt
from numpy import pi, linspace, log10

#  EDIT
```

```
R = 10.0E3
C = 100.0E-9

# END EDIT


RC = R*C

# create equally spaced log(f) values array f
flog = linspace(1.0, 4.0, 100)
f = 10.0 ** flog

# calculate F (complex) and absolute value Fabs
F = 1 / (1 + 1j*  2 * pi * f * RC)
Fabs = abs(F)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(f, Fabs)
ax.grid(True, which = "both", linestyle = "-")
ax.set_xscale ("log")
ax.set_xlabel("f/Hz")
ax.set_ylabel("U2/U1")
ax.set_title("RC low pass frequency response")
plt.show()
```

In diesem Beispiel kommt die Fähigkeit Pythons, mit komplexen Zahlen rechnen zu können, zur Geltung.

Im Programm wird zunächst ein Array f mit Frequenzwerten berechnet. Damit diese in logarithmischer Darstellung gleichverteilt sind, wird zunächst ein Array flog mit linear verteilten Werten berechnet und daraus dann die Frequenzwerte als $f = 10^{flog}$.

Anschließend wird das Array der komplexen $\overline{F}$-Werte, und das Array des Betrags Fabs berechnet. Sehr angenehm ist hierbei wieder, dass das Modul Numpy Vektorfunktionen unterstützt. Man braucht keine FOR-NEXT-Schleife zur Berechnung der Werte und der Code wird klar und übersichtlich.

Zum Schluss wird das Diagramm gezeichnet, wobei die Frequenzachse logarithmisch skaliert wird. Für eine schöne Darstellung ist es wichtig, das Gitter richtig zu setzen:

```
ax.grid(True, which = "both", linestyle = "-")
```

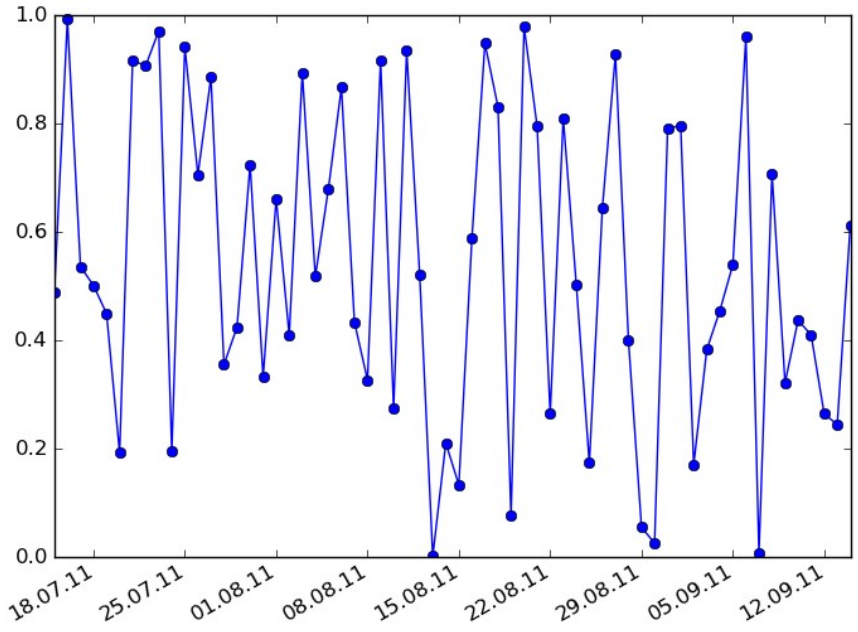
Hier wird das Gitter eingeschaltet, mit „both“ werden beide Gitter (Haupt- und Nebengitter für die Unterteilung) eingeschaltet, und der Linienstyl wird auf durchgezogene Linien festgelegt.

Ohne Probleme kann man noch den Phasengang hinzufügen, wenn man folgende Zeilen vor plt.show() einfügt:

```
# plot phi = f(f)
phi = angle(F)*180.0/pi
ax2 = fig.add_subplot(212)
ax2.plot(f, phi)
ax2.grid(True, which = "both", linestyle = "-")
ax2.set_xscale ("log")
ax2.set_xlabel("f/Hz")
ax2.set_ylabel("phi/degrees")
```

Plot_date

```
import matplotlib.pyplot as plt
import numpy as np
import datetime as dt
import matplotlib as mpl

# dates range for plot   15.7.2011 to  15.9.2011
date2=dt.datetime(2011,9,15)
date1 = dt.datetime(2011, 7, 15)
deltat=dt.timedelta(days=1)               # interval 1 day
dates=mpl.dates.drange(date1, date2, deltat)
print dates

# random y values
y = np.random.rand(len(dates))

# plot using plot_date
fig1 = plt.figure()
ax1 = fig1.add_subplot(111)
ax1.plot_date(dates , y, linestyle = "-")

# format nicely
dateFmt= mpl.dates.DateFormatter("%d.%m.%y")
ax1.xaxis.set_major_formatter(dateFmt)
fig1.autofmt_xdate()

plt.show()
```



Markierungsstriche jeweils am Montag

Eine Markierung für jeden Tag (minor_locator) kann mit

```
daysLoc = mpl.dates.DayLocator()
ax1.xaxis.set_minor_locator(daysLoc)
```

eingefügt werden.