Skript CP-Vorkurs Teil 2

March 27, 2021

Santiago Rodriguez, 14.3.2021

1 Pakete/Module

Python ist eine sehr flexible Programmiersprache, die sich schnell und einfach durch Pakete -das sind kollektionen von Modulen, die wiederum Programme mit nützlichen Funktionen enthalten- erweitern lässt, um dessen Kapazitäten auszuweiten. Hierfür müssen sie zuerst das Paket in ihren aktuellen Environment installiert haben; ansonstens müssen sie es zuvor über ihren Package Manager installieren. Anaconda, die in diesen Tutorials empfohlene Distribution, sowie der Jupyter-Hub auf den Servern des Physik Instituts enthalten schon viele wichtige, wissenschaftliche Pakete und einen Package Manager mit Zugang zu vielen mehr.

Sie können Module/Pakete in Python ber das "import" Kommando einbinden. Die wichtigsten Pakete, mit denen wir uns in diesen Tutorium befassen werden sind

1. Numpy - Modul zur Einführung algebraischer Strukturen und Operationen wie Matrizen und Determinanten

2. SciPy - Modul mit Funktionen/Libraries zu numerischen Verfahren und Algorithmen wie Integration oder LGS-Löser

3. Matplotlib - Modul zur Darstellung von 2D, 3D sowie interaktiven/animierten Grafiken

[1]:
```
#Pakete importieren mit einem beliebigen Namen

import numpy as np      #numpy Funktionen können somit unter den Kommando np.␣
 ↪aufgerufen werden

import matplotlib.pyplot as plt #sie können auch nur bestimmte Teile des Pakets␣
 ↪importieren mit .

import scipy as sp

from scipy.optimize import curve_fit #sowie auch nur explizite Funktionen mit␣
 ↪from
```

2 Numpy

Numpy ist ein Paket, der mit dem Dateityp ndarray matrizenähnliche Strukturen zur Verfügung stellt und somit in Python algebraische Rechenoperationen wie Matrizenmultiplikationen, Determinanten u.s.w. hinzufügt.

2.1 Arrays

ndarrays funktionieren grundsätzlich wie Listen, mit denen man wie Matrizen rechnen kann. Weiterhin können sie darauf auch praktische Funktionen und Kommandos anwenden, um sie zu modifizieren, sortieren, abkürzen u.s.w.

[16]:
```
#numpy arrays können über die Funktion array() erstellt werden

A = np.array([[1,2,3],[4,5,6],[7,8,9]])           #sie können numpy arrays mit␣
 ↪dem Befehl np.array erstellen, indem sie eine gewöhnliche Liste innerhalb der␣
 ↪Funktion aufstellen.
                                                  #formel müssen ndarrays alle␣
 ↪Einträge enthalten, die ihre Dimension impliziert

b = np.array([8,5,7])                             #für einen Vektor im R^3

Liste_NP = [[1,2,5],[8,5,6],[5,9,0]]              #alternativ können sie die␣
 ↪Liste im vornherein erstellen und in die Funktion als Variable eingeben

np.array(Liste_NP)                                #sie können fast alles in␣
 ↪ndarrays reinpacken
```

[16]:
```
array([[1, 2, 5],
       [8, 5, 6],
       [5, 9, 0]])
```

[19]:
```
#allgemein werden numpy arrays pro Zeile geschrieben. Sie schreiben also zuerst␣
 ↪für die n-te Zeile ihre jeweiligen m Spalteneinträge. Sie können den Dateityp␣
 ↪auch explizit angeben über die dtype Option.

np.array([["a11","a12","a13"],["a21","a22","a23"],["a31","a32","a33"]] ,␣
 ↪dtype=str)  #numpy arrays akzeptieren auch dtypes wie str.
```

[19]:
```
array([['a11', 'a12', 'a13'],
       ['a21', 'a22', 'a23'],
       ['a31', 'a32', 'a33']], dtype='<U3')
```

[24]:
```
#man muss den dtype aber vorsichtig wählen, da bestimmte dtypes mit einigen␣
 ↪Rechenoperationen nicht kompatibel sind
a = np.ones(200, dtype = np.bool)
for i in range(len(a)):
    a[i] = i;
```

```
a.sum() #beim dtype bool liefert np.sum einen kleineren Wert zurück, als zu␣
 ↪erwarten wäre
```

[24]:
```
dtype('float64')
```

[4]:
```
#arrays können gelesen werden über das gleiche Verfahren wie Listen
A[0]            #die nullte Komponente eines Arrays ist die 1 Zeile
```

[4]:
```
[1, 2, 3]
```

[5]:
```
A[0][0]        #und die nullte Komponente davon ist der Eintrag der 1 Spalte in␣
 ↪der 1 Zeile
```

[5]:
```
1
```

[6]:
```
np.sort(b)     #arrays können der größe Nach von links -> rechts sowie oben ->␣
 ↪unten sortiert werden
```

[6]:
```
array([5, 7, 8])
```

[8]:
```
#weiterhin können sie besondere Arrays mit den unteren Funktionen erstellen

np.arange(0,10, 0.1)   #erstellt einen array von 0 bis n-i in Schritten der␣
 ↪Größe i. Nützlich, um Funktionen auszuwerten

np.linspace(0,10,100)  #ähnlich zu arange, teilt aber den Intervall von 0 bis n␣
 ↪in genau i äquidistante Punkte

np.zeros(10)           #erstellt einen leeren array aus nullen. Man kann auch den␣
 ↪dateityp über die dtype option spezifizieren

np.zeros((3, 3))       #kann auch für leere nxn arrays mithilfe eines tupels␣
 ↪angewendet werden

np.diag((5,1,2,3,4,5)) #erstellt eine quadratische nxn Matrix mit den␣
 ↪angegebenen Diagonaleinträgen

np.identity(3)        #erstellt eine quadratische nxn Einnheitsmatrix

np.eye(5, k=2)     #erstellt eine quadratische nxn Matrix mit den angegebenen␣
 ↪Diagonaleinträgen ab der k-ten Spalte
```

[8]:
```
array([[0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

2.2 Array Operationen

Man kann auf ndarrays einen Großteil der selben Matrizenoperationen anwenden, die sie ebenfalls aus der Algebra kennen. Weiterhin können sie aber auch etwas informellere Operationen durchführen, die dennoch ziemlich praktisch sind

[67]:
```
#Wir definieren uns zuerst ein paar 3x3 Matrizen
A = np.array([[1,3,4],[5,6,8],[9,2,7]])
B = np.array([[1,3,4],[7,8,9],[2,5,6]])
#sowie ein paar R^3 Vektoren
x = np.array([1, 2, 3])
y = np.array([4, 5, 6])
```

[68]:
```
A+B         #bei Addition wird wie gewöhnlich elementenweise addiert, so wie bei␣
 ↪Matrizen

x+y

A+2         #addition mit arrays funktioniert auch wenn nicht formal mathematisch␣
 ↪definiert mit Ints/Skalaren, der Skalar wird dann an alle Elemente im Array␣
 ↪addiert

x+2

A*B         #Bei Multiplikation wird aber anders als bei Matrizen auch␣
 ↪elementenweise Multipliziert

x*y

np.sum(x)  #Hiermit kann man alle Elemente in einem array miteinander␣
 ↪aufsummieren

np.prod(x) #sowie das Produkt aller Elemente

np.dot(A, B) #Für gewöhnliche Matrizenmultiplikation benutzen sie die Funktion␣
 ↪np.dot

np.dot(B,A)  #so wie sie aus der Algebra bereits kennen, ist diese Operation i.A.
 ↪ nicht kommutativ

np.dot(x,y)  #die np.dot Funktion liefert aber bei Vektoren das Skalarprodukt␣
 ↪und ist schon kommutativ

np.cross(x,y) #für den Kreuzprodukt zweier Vektoren gibt es zusätzlich noch die␣
 ↪np.cross Funktion
```

```
#bei allen Operationen müssen beide arrays die gleichen np.shape() haben, bzw.␣
 ↪bei Matrizen Multiplikation die Spalten des ersten mit dem Zeilen des zweiten␣
 ↪Arrays übereinstimmen
print(np.shape(A))
print(np.shape(B))
```

```
(3, 3)
(3, 3)
```

2.3 Erweiterte Funktionen

numpy enthält auch weitere nützliche Funktionen außerhalb der Matrizen-ähnlichen Rechnung, so wie

[11]:
```
np.sqrt(4) #quadratische Wurzel

np.log(2.71) #natürlicher Logarithmus

np.log10(10) #dekadischer Logarithmus

def log_b(x, b):                        #Logarithmus zu einer beliebigen Basis b
    output = np.log10(x)/np.log10(b)
    return output

log_b(8, 2)

np.exp(1)    #Exponentialfunktion

np.pi        #Pi als float
```

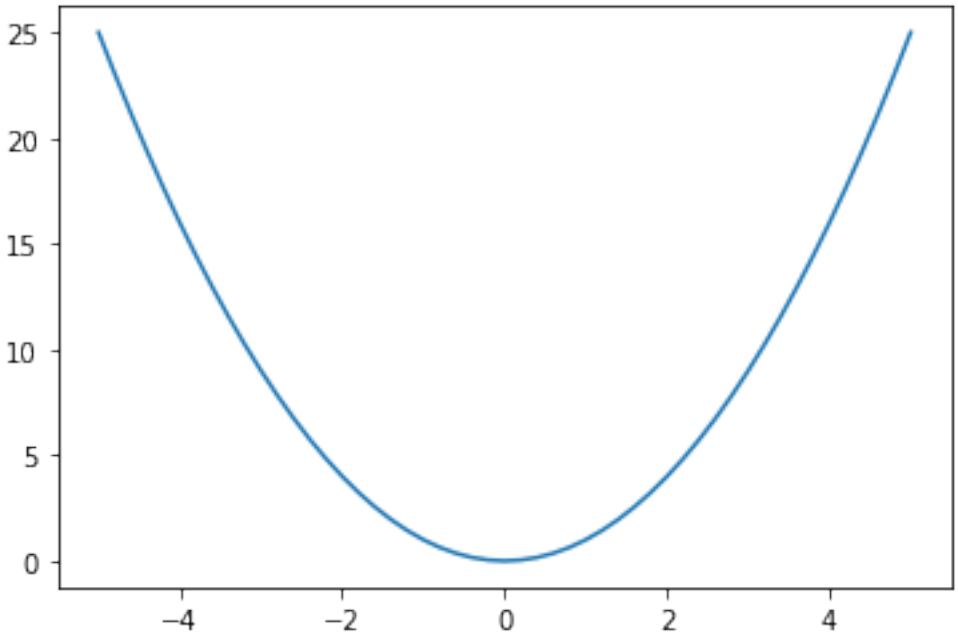
[11]:
```
3.141592653589793
```

3 Matplotlib

Mit matplotlib können sie 2D und 3D Grafiken aus Daten oder Funktionen erzeugen, die auch Animationen für zeitliche Veränderungen sowie Interaktive Sliders eingebaut haben können.

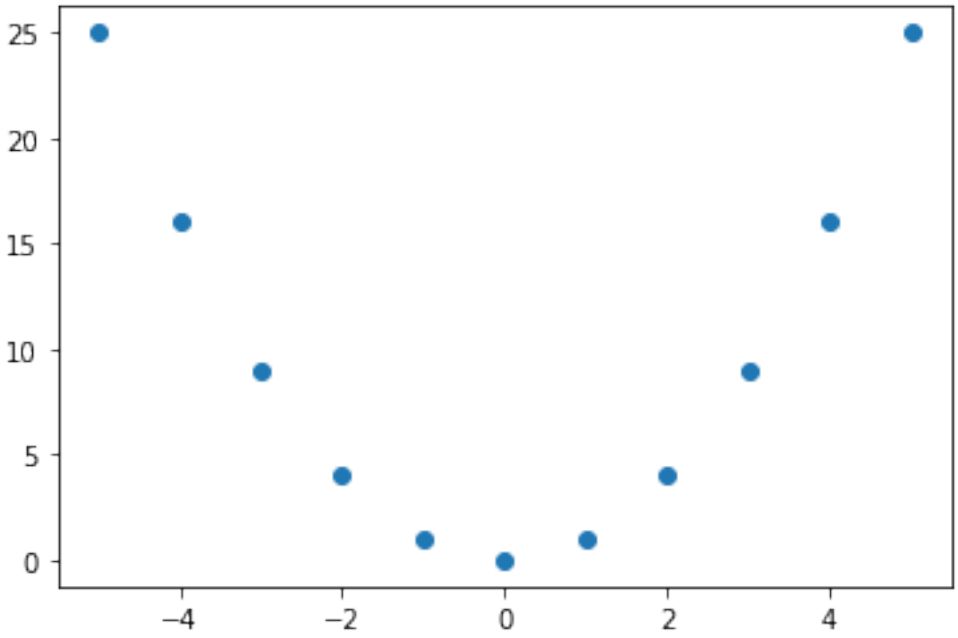
[12]:
```
#Einfache 2D Plots können über die .plot Funktion erstellt werden, mit dem␣
 ↪Format plot(x,y)
x = np.arange(-5,5.2,0.2)
f = lambda x: x**2          #mit lambda x: können sie kurze Funktionen auch␣
 ↪deklarieren
plt.plot(x, f(x))
```

[12]:
```
[<matplotlib.lines.Line2D at 0x7f1b373cc8d0>]
```



[13]:
```
#Sie können auch Datenpunkte plotten mit der .scatter Funktion
x = np.arange(-5,6,1)
y = f(x)
plt.scatter(x,y)
```

[13]: <matplotlib.collections.PathCollection at 0x7f1b2f31fd10>

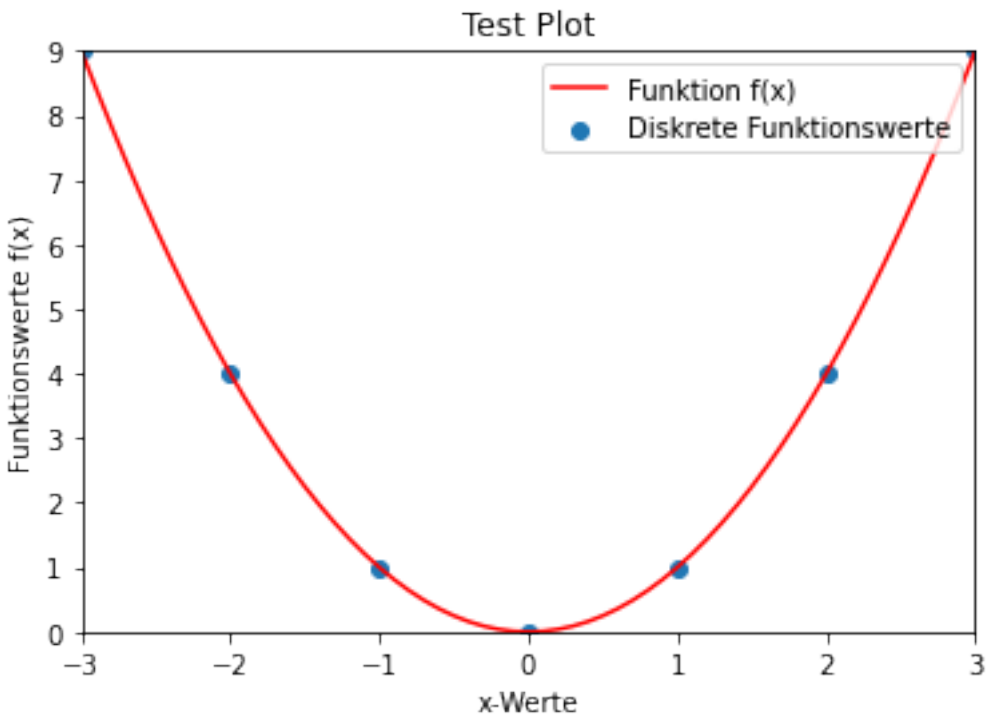


[60]:
```
#Weiterhin können sie ihren Plot sowie dessen Achsen beschriften, eine Legende␣
 ↪einbauen, die Skala definieren, mehrere Plots zur gleichen Zeit erstellen u.s.
 ↪w.
x = np.arange(-5,6,1)
y = f(x)
plt.scatter(x,y, label="Diskrete Funktionswerte")    #der Label gibt Matplotlib␣
 ↪den Namen des Plots an


x = np.arange(-5,6,0.1)
plt.plot(x, f(x), label="Funktion f(x)", color='red') #mit color können sie␣
 ↪ihren Plot eine beliebige Farbe zuweisen

plt.legend(loc="upper right")                        #ruft die Legende auf an␣
 ↪der angegebenen loc
plt.title("Test Plot")                               #gibt der Figur einen Titel
plt.xlabel("x-Werte")                                #setzt die Beschriftung der␣
 ↪x-Achse fest
plt.ylabel("Funktionswerte f(x)")                    #setzt die Beschriftung der␣
 ↪y-Achse fest
plt.xlim([-3,3])                                     #setzt die Grenzen der␣
 ↪x-Achse fest
plt.ylim([0,9])                                      #setzt die Grenzen der␣
 ↪y-Achse fest
```

[60]:
```
(0.0, 9.0)
```



[15]:
```
#Weiterhin können sie interaktive Elemente über die widget extension in␣
 ↪Zusammenspiel mit dem ipywidgets Paket einbauen,so bspw. für Funktionsscharen␣
 ↪ein Slider für den Parameter a
import ipywidgets as widgets
#%matplotlib widget
#Figur definieren
x = np.arange(-5,5,0.2)
fig, ax = plt.subplots(figsize=(6, 4))
ax.set_xlim([-5,5])
ax.set_ylim([0,20])
ax.grid(True)
plt.title("Funktionsschar f_a(x) = ax^2")
#Funktion und Updater
f = lambda x,a: a*x**2
@widgets.interact(a=(0, 5, 0.1))
def update(a = 1.0):
    """Remove old lines from plot and plot new one"""
    [l.remove() for l in ax.lines]
    ax.plot(x, f(x, a), color='C0')
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
```

```
        <ipython-input-15-6472ab8afdc1> in <module>
          1 #Weiterhin können sie interaktive Elemente über die widget extension␣
       ↪in Zusammenspiel mit dem ipywidgets Paket einbauen,so bspw. für␣
       ↪Funktionsscharen ein Slider für den Parameter a
          ----> 2 import ipywidgets as widgets
                3 #%matplotlib widget
                4 #Figur definieren
                5 x = np.arange(-5,5,0.2)


        ModuleNotFoundError: No module named 'ipywidgets'
```

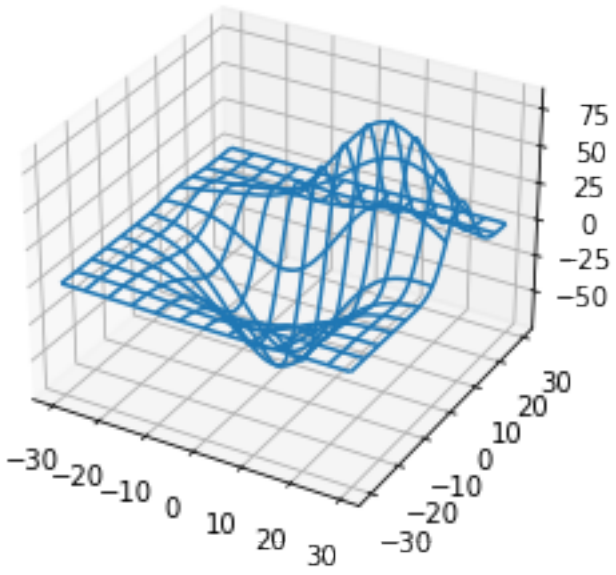
[ ]:
```
#für 3D Plots benutzen sie das axes3D Modul aus den erweiterten mpl_toolkits␣
 ↪Paket
from mpl_toolkits.mplot3d import axes3d

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Grab some test data.
X, Y, Z = axes3d.get_test_data(0.05)

# Plot a basic wireframe.
ax.plot_wireframe(X, Y, Z, rstride=10, cstride=10)
```

[ ]:
```
<mpl_toolkits.mplot3d.art3d.Line3DCollection at 0x7f9459739c70>
```



[ ]:
```
#um wieder auf normale Plots zu wechseln benutzen sie
%matplotlib inline
```

4 Scipy

Scipy enthält viele Funktionen und Libraries zur numerischen Auswertung von LGS, DGL, Fourier Transformationen, numerische Integration, Fits u.s.w.

4.1 LGS

Ein LGS der Form $A\vec{x} = \vec{b}$ kann bspw. definiert und gelöst werden mit

[61]:
```
A = np.array([[2,4,0],[8,8,4],[2,0,4]])      #Definieren der Matrix A als (3,3)␣
↪Array
b = np.array([1,6,8])                        #Definieren des Vektors b als (3,)␣
↪Array
```

[62]:
```
#die Funktion sp.linalg.solve enthaltet den LGS Löser und akzeptiert einen Input␣
↪(A,b)
x_vec = sp.linalg.solve(A,b)
print(x_vec)                    #als Output liefert es dann den Lösungsvektor x
```

```
[-2.    1.25  3.  ]
```

[63]:
```
np.dot(A, x_vec)   #überprüfen mit Matrizenmultiplikation dass A*x = b gilt
```

[63]:
```
array([1., 6., 8.])
```

[64]:
```
np.dot(A, x_vec) == b #kann auch über einen True/False Statement Vergleich␣
↪bestätigt werden
```

[64]:
```
array([ True,  True,  True])
```

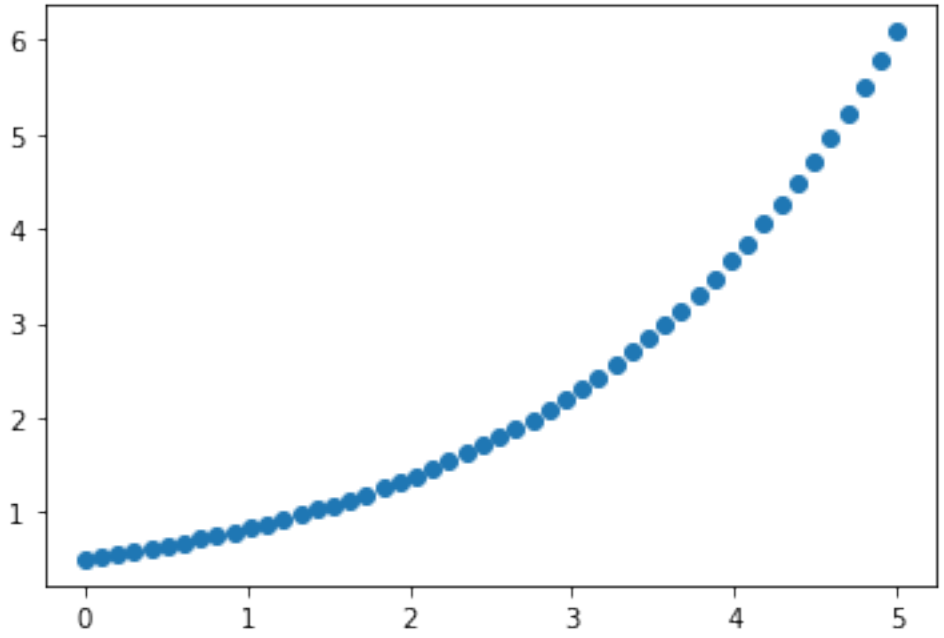
4.2 Fits

Eine beliebige Modelfunktion zu einem Datenset kann definiert und nach unterschiedlichen Verfahren (abhängig von der verwendeten scipy Funktion) gefittet werden, hier bspw. nach der Method der kleinsten Quadrate $min(\chi^2)$

[ ]:
```
def exp(x, a, b):              #definieren der Modelfunktion mit Fit-Parametern a␣
↪und b
    y = a*np.exp(b*x)          #hierbei handelt es sich um einen exponentiellen␣
↪Fit mit np.exp()
    return y
```

[ ]:
```
x = np.linspace(0,5,50)       #definieren eines Test-Datensets
y = exp(x, 0.5, 0.5)          #wir wollen, dass diese Daten Punkte einen␣
 ↪exponentiellen Anstieg bilden
plt.scatter(x,y)
```

[ ]:
```
<matplotlib.collections.PathCollection at 0x7f945aaf1670>
```

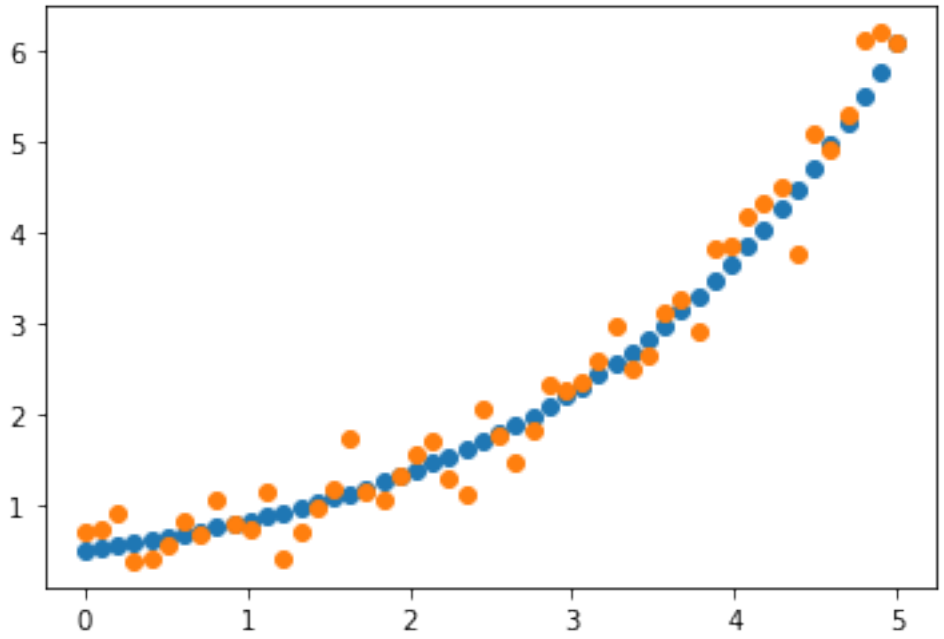


[ ]:
```
rausch_signal = 0.3*np.random.normal(size=y.size)  #numpy enthaltet einen␣
 ↪random-number Generator über den sich ein Rausch-Signal sehr gut einbauen␣
 ↪lässt
y_data = y + rausch_signal       #dazu wird dieses lediglich auf die␣
 ↪idealisierten Daten hinzugefügt
print("Der ursprüngliche erste Wert des Datenset ", y[0])
print("ist somit nun Rauschbedingt", y_data[0])
```

```
Der ursprüngliche erste Wert des Datenset  0.5
ist somit nun Rauschbedingt 0.6993827709231326
```

[ ]:
```
plt.scatter(x,y) #ursprüngliche Datenpunkte
plt.scatter(x,y_data) #Datenpunkte mit Rausch
```

[ ]:
```
<matplotlib.collections.PathCollection at 0x7f945aaca100>
```

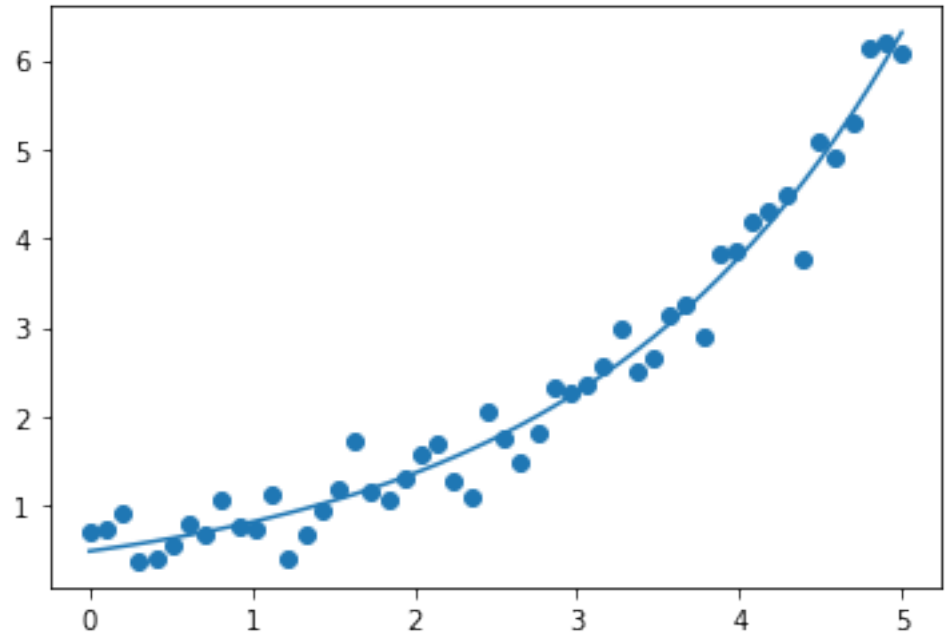


[ ]:
```
#um die nun rauschbedingten Daten an die Modelfunktion exp(x,a,b) wieder zu␣
 ↪fitten ruft man mit curvefit die vorhin importierte Funktion scipys
fit_par, fit_cov = curve_fit(exp, x, y_data) #Fits der Datenpunkte nach Methode␣
 ↪der kleinsten Quadrate X^2
a_fit, b_fit = fit_par #gefittete Parameter extrahieren
print(a_fit)
print(b_fit)
```

```
0.49262692327336377
0.5100640231376883
```

[ ]:
```
plt.plot(x, exp(x, a_fit, b_fit)) #Plot des Fits
plt.scatter(x,y_data) # Plot der Datenpunkte des Fits
```

[ ]:
```
<matplotlib.collections.PathCollection at 0x7f945abb1d30>
```



4.3 Integration

Scipy enthaltet weiterhin einen numerischen Integrator Quad sowie DGL Löser odeint mit dem sie Funktionen und DGL numerisch Integrieren können. Für die unterschiedlichen Routinen wird hier auf die Dokumentation von scipy https://docs.scipy.org/doc/scipy/reference/integrate.html verwiesen

Sei nun bspw. die Funktion $f(x) = 4x + 2$ numerisch im Intervall $[0, 1]$ zu integrieren oder das unbestimmte Integral grafisch darzustellen

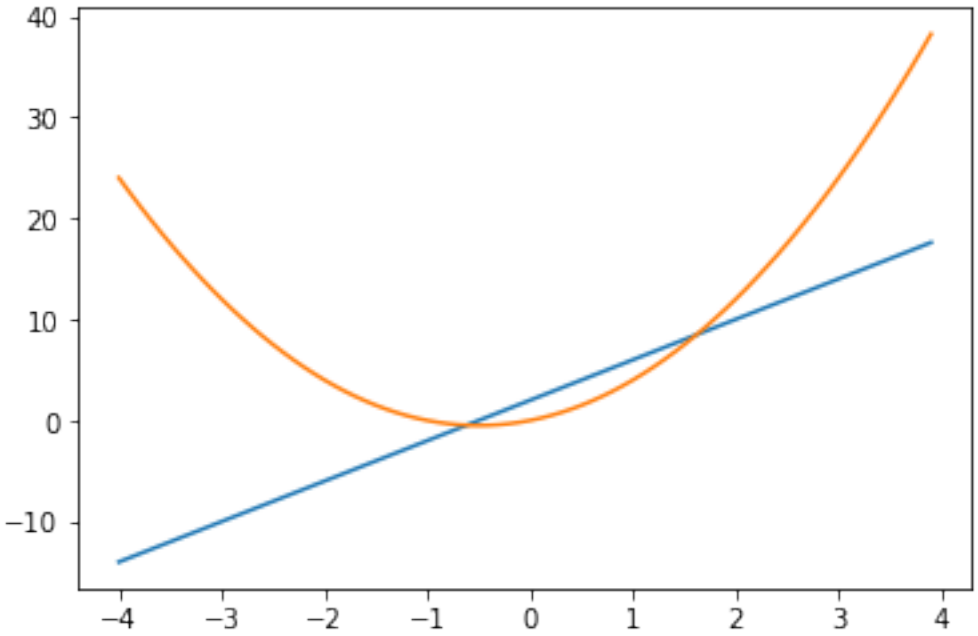
[ ]:
```
#um eine Funktion numerisch zu integrieren definieren sie diese zuerst so wie␣
 ↪beim Fitten und importieren die scipy Funktion
import scipy.integrate as integrate
f = lambda x: 4*x+2
integrate.quad(f,0,1)   #bestimmtes Integral der Funktion von 0 bis 1, der␣
 ↪Output ist ein Tupel (Ergebnis,abs. Fehler der Integration)
```

[ ]:
```
(4.0, 4.440892098500626e-14)
```

[ ]:
```
#eine Stammfunktion, bzw. unbestimmtes Integral kann numerisch durch␣
 ↪Implementierung eines Loops geplottet werden
x = np.arange(-4,4,0.1)
y_int = [] #initialisieren einer Liste und Variablen
int_val = 0
for i in range(len(x)):
```

```
    int_val = integrate.quad(f,0,x[i])[0]     #bestimmen des Integrals von 0 bis␣
 ↪x für jeden gezeichneten x-Wert
    y_int.append(int_val)                     #als Liste speichern
y_int_array = np.array(y_int)                 #und in ndarray konvertieren
plt.plot(x,f(x))         #Plot der integrierten Funktion
plt.plot(x,y_int_array) #Plot des unbestimmten Integrals
```

[ ]:
```
[<matplotlib.lines.Line2D at 0x7f945a9b5220>]
```



Sei nun weiterhin eine Differentialgleichung gegeben der Form $\frac{y'(t)}{y(t)} = t^2 e^{-t}$ und numerisch auf alle möglichen Werte aus einem gegebenen Intervall $ t ∈ ℝ$nachy(t)$ zu integrieren, so müssen sie diese wie gewohnt auf die Form $y'(t) = t^2 e^{-t} y(t)$ bringen. Dann wird sie zusammen mit einer Anfangsbedingung y_0 in den numerischen Integrator eingegeben.

[73]:
```
from scipy.integrate import odeint
# Funktion die y'(t)=dy/dt zurückliefert
def dgl(y,t):
    dydt = t**2*np.exp(-t)*y
    return dydt
#Anfangsbedingung
y0 = 1
#Definieren des Integrationsintervalls als array
t = np.linspace(0,10,1000)
#Integrieren des DGL Systems auf den gegebenen Intervall samt Bedingungen
y  = odeint(dgl, y0, t)
#Plot der Ergebnisse
```

```
plt.plot(t,y)
plt.xlabel('x-Werte')
plt.ylabel('f(x)')
plt.show()
```