einfuehrung_full

February 22, 2024

1 Python Grundlagen

Wir verwenden meist die Bibliothek **numpy** (numerical python) und eine Bibliothek, z.B. **matplotlib.pyplot** für plots. numpy ist an MATLAB angelehnt. Das Folgende bezieht sich hauptsächlich auf diese zwei Bibliotheken

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

1.0.1 Vektoren und Vektoroperationen

Vektoren können explizit durch Aufzählung der Elemente definiert werden

[3]:
```
x = np.array([1,3,6,np.pi,np.exp(1)])
print(x)
```

```
[1.         3.         6.         3.14159265 2.71828183]
```

Einen Vektor von äquidistanten Werten erhält man z.B. mit **np.arange** oder **np.linspace**. Achtung: bei **arange** gehört die obere Grenze **stop** nicht mehr dazu, beim **linspace** schon. Vorsicht vor Rundungsfehlern…

[11]:
```
x = np.arange(start = 0, stop = 10, step = 1)
print(x)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

[10]:
```
x = np.linspace(start = 0, stop = 10, num = 13)
print(x)
```

```
[ 0.          0.83333333  1.66666667  2.5         3.33333333  4.16666667
  5.          5.83333333  6.66666667  7.5         8.33333333  9.16666667
 10.        ]
```

Die allermeisten Rechenoperationen lassen sich auf Vektoren anwenden und wirken dann auf jedes Element einzeln. Solche "Vektoroperationen" sind kürzer und besser lesbar als Schleifen (und auch schneller)

[13]:
```
x = np.arange(10)
print("x = ", x)
print("x^2 = ", x**2) # Schreibweise a^b -> a**b
```

```
print("x + 1 =", x + 1)
```

```
x =  [0 1 2 3 4 5 6 7 8 9]
x^2 =  [ 0  1  4  9 16 25 36 49 64 81]
x + 1 = [ 1  2  3  4  5  6  7  8  9 10]
```

1.0.2 Skalarprodukt und Matrixprodukte

Der Operator für Matrizenprodukte ist @ (anstelle von *)

[32]:
```
x = np.array([1,2,3])
y = np.array([-2,3,6])
print(x @ y)
print(x * y)
print(np.sum(x*y))
```

```
22
[-2  6 18]
22
```

Schleifen gibt es natürlich auch. Die Syntax ist "*for x in v*", wobei v eine Menge ist, über die iteriert werden kann, also z.B. ein Vektor

[15]:
```
for i in np.arange(10):
    print(i)
```

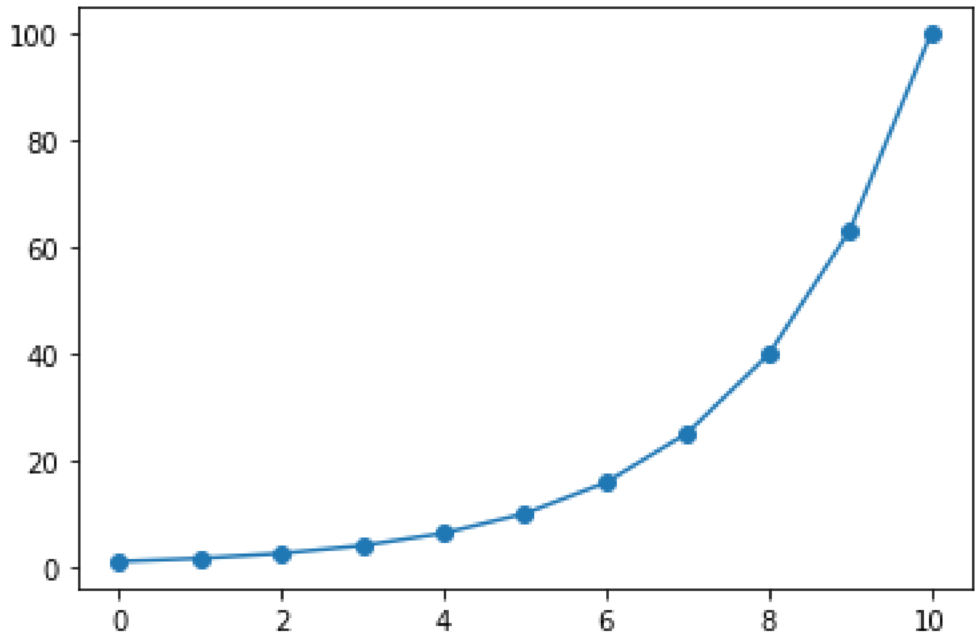
```
0
1
2
3
4
5
6
7
8
9
```

Häufig ist auch *logspace* praktisch: *np.logspace(a, b, n) = 10**np.linspace(a, b, n)*

[23]:
```
x = np.logspace(0, 2, 11)
print(x)
plt.plot(x, 'o-')
```

```
[  1.           1.58489319   2.51188643   3.98107171   6.30957344
  10.          15.84893192  25.11886432  39.81071706  63.09573445
 100.        ]
```

[23]:
```
[<matplotlib.lines.Line2D at 0x1344567a040>]
```

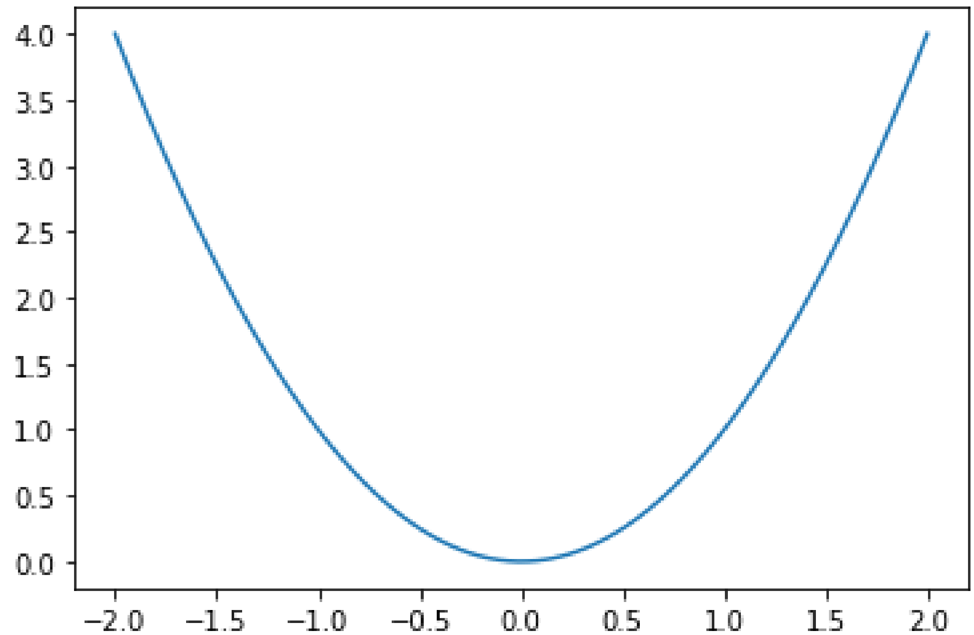


1.0.3 Plots

plots von Funktionswerten y = f(x) können nach dem Schema **plt.plot(x, f(x))** erstellt werden, wobei die x-Achse als Vektor zuvor definiert wird.

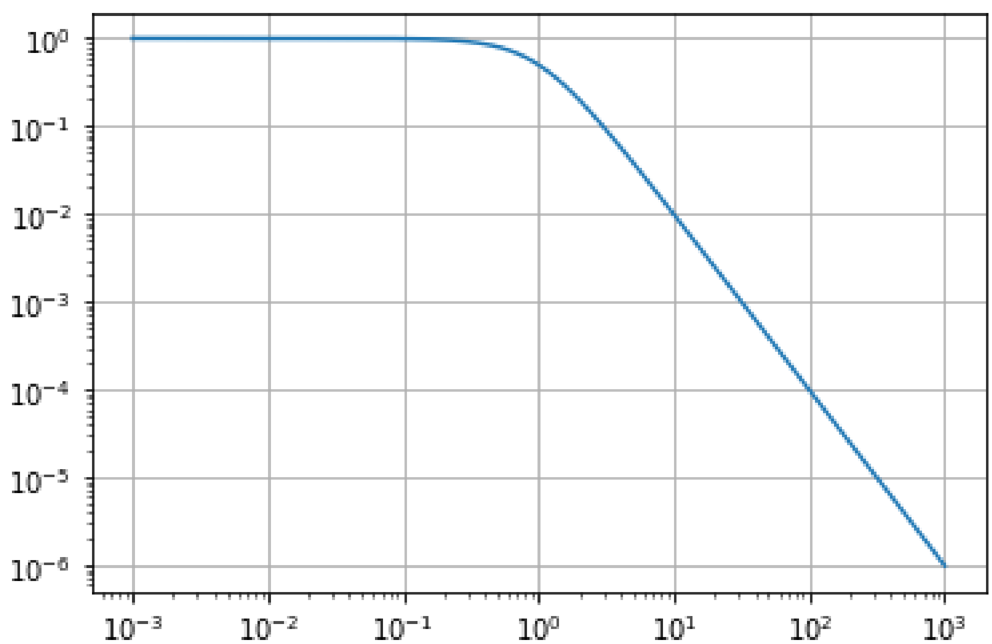
[50]:
```
x = np.linspace(-2, 2, 500)
plt.plot(x, x**2)
```

[50]:
```
[<matplotlib.lines.Line2D at 0x13447764ac0>]
```



Es gibt natürlich auch logarithmische und halblogarithmische Plots

[53]:
```
w = np.logspace(-3, 3, 500)
plt.loglog(w, 1/(1+w**2))
plt.grid()
```



1.1 Funktionen definieren

Eigene Funktionen können mit dem Schlüsselwort **def** erklärt werden:

[25]:
```
def f(x, y):
    return np.sqrt(x**2 + y**2)

print(f(3,4))
```

```
5.0
```

Die Funktion f könnte natürlich mehr Code enthalten. Funktionen wie das obige Beispiel, die nur aus *return irgendwas* bestehen, können auch als sogenannte **lambdas** erklärt werden.

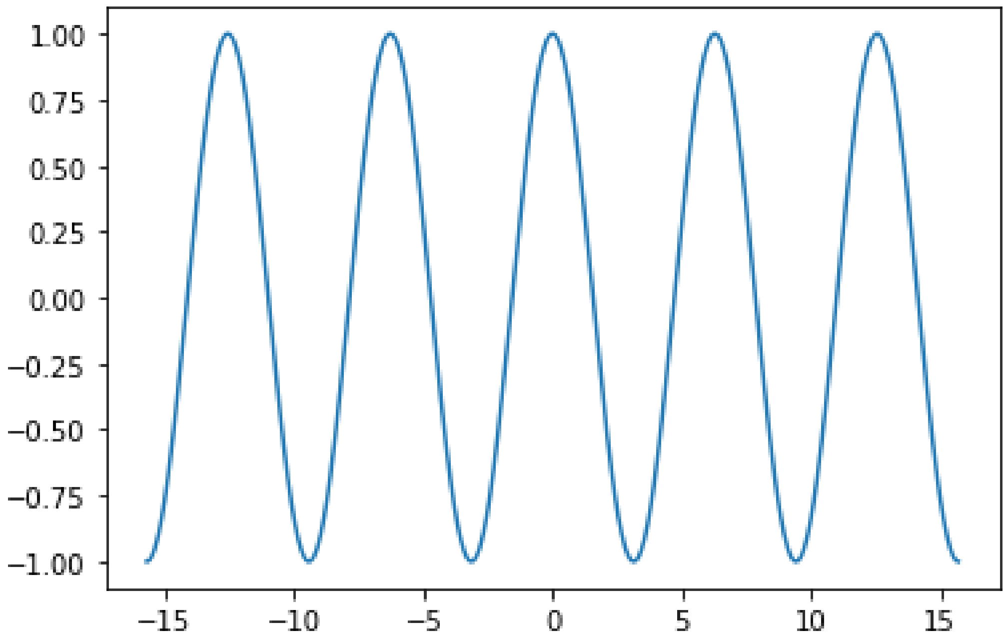
[34]:
```
f = lambda x, y: np.sqrt(x**2 + y**2)

print(f(3,4))
```

```
5.0
```

Schliesslich lassen sich Funktionen wie gewöhnliche Variablen behandeln, als Funktionsargumente übergeben und zuweisen. Alle drei Funktionen f, g, h im folgenden Beispiel tun exakt dasselbe

[43]:
```
def f(x):
    return np.cos(x)

g = lambda x: np.cos(x)

h = np.cos

x = np.linspace(-np.pi, np.pi, 500) * 5
plt.plot(x, h(x))
```

[43]:
```
[<matplotlib.lines.Line2D at 0x1344726f190>]
```



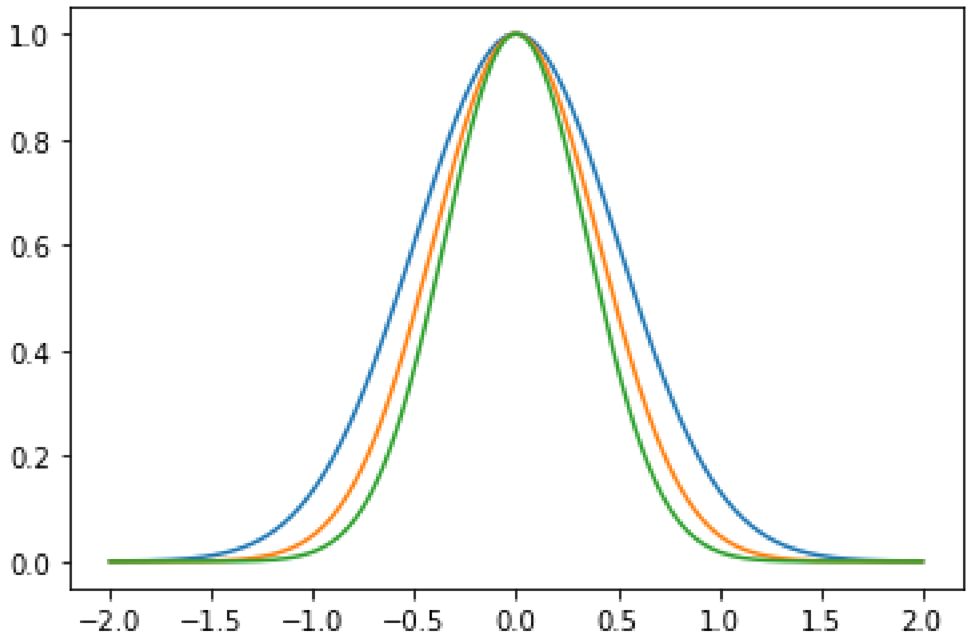
1.1.1 Erster Teil von Praktikum 1

Aufgabe 1 Erstellen Sie einen plot von

$$e^{-x^2/\sigma}$$

für $x \in [-2, 2]$ und $\sigma \in \{1/2, 1/3, 1/4\}$

[21]:
```
x = np.linspace(-2, 2, 500)
for sigma in [1/2, 1/3, 1/4]:
    plt.plot(x, np.exp(-x**2 / sigma))
```



Aufgabe 2 Programmieren Sie eine *effiziente* Funktion, welche die geometrische Folge

$$\{q^k\}_{k=0}^n$$

berechnet

[27]:
```
def geo(q, n):
    return q**np.arange(n+1)

print(geo(0.5, 3))
```

```
[1.    0.5   0.25  0.125]
```

[33]:
```
print (geo(2, 10) @ geo(0.5, 10))
```

```
11.0
```

Aufgabe 3 Bestimmen Sie eine Näherung an die Eulersche Zahl e mit der Formel

$$e = \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n$$

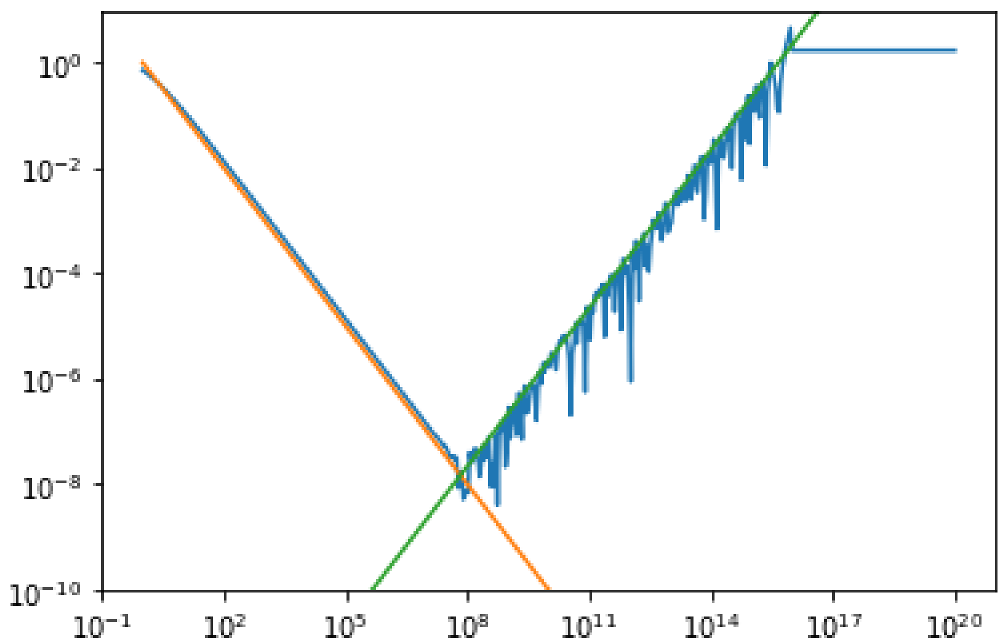
Wie genau wird die Näherung im besten Fall, und was ist der optimale Wert für n?

[39]:
```
e = np.exp(1)

n = 10
en = (1+1/n)**n
print("e  = ", e, "\nen = ",en, "\nFehler:",e-en)
```

```
e  =  2.718281828459045
en =  2.5937424601000023
Fehler: 0.12453936835904278
```

[59]:
```python
n = np.logspace(0,20,500)
en = (1+1/n)**n
plt.loglog(n, np.abs(e - en), '-')

eps = np.finfo(float).eps
plt.loglog(n, 1/n)
plt.loglog(n, eps*n)
plt.ylim([1e-10,1e1])
```

[59]:
```
(1e-10, 10.0)
```



Die zweite Praktikumsaufgabe kann sehr ähnlich gelöst werden.

[64]:
```python
x0 = 1 # Stelle, wo die Ableitung berechnet werden soll
f = np.cos # Funktion
df = lambda x: -np.sin(x) # Ableitung exakt
h = 1e-5 # Schrittweite für Differenzenquotienten

Df = (f(x0 + h) - f(x0) ) / h # Vorwärtsdifferenzenquotient

print("df =", df(x0), "\nDf = ", Df, "\nFehler: ", df(x0) - Df)
```

```
df = -0.8414709848078965
```

```
Df =  -0.8414736863193716
Fehler:  2.7015114750783553e-06
```

Jetzt nochmal dasselbe wie oben für exp: verwenden Sie für h einen geeigneten logspace und erstellen Sie einen Plot des Fehlers in Abhängigkeit von h

[ ]:

```
```