

# SymPy\_SymbolischeMathematik

January 28, 2023

## 1 Symbolische Mathematik mit SymPy

Das Modul *sympy* ermöglicht es, symbolische Mathematik anzuwenden.

Falls SymPy noch nicht installiert wurde, kann man dies unter Anaconda in einer Anaconda-Konsole nachholen:

```
conda install -c anaconda sympy
```

Was bedeutet dies? Dazu zuerst eine Gegenüberstellung mit dem *math*-Modul, dem Standardmodul von Python für mathematische Operationen. Im folgenden Code importieren wir sowohl *sympy* als auch *math* und berechnen jeweils die Wurzel aus der Zahl 9. Da wir keine konkreten Funktionen aus den Modulen importieren, müssen wir den Modulnamen als Präfix davorschreiben, was uns hier auch bei der Unterscheidung der beiden Modulen hilft.

```
[1]: import sympy
import math

print(math.sqrt(9))
sympy.sqrt(9)
```

3.0

```
[1]: 3
```

Beide Ausgaben sehen ganz ähnlich aus: Es wird, wie erwartet, eine 3 ausgegeben. Allerdings sehen wir keine Kommazahl bei der Ausgabe von *sympy*. Um welche Objekte handelt es sich hier?

```
[2]: print(type(math.sqrt(9)))
print(type(sympy.sqrt(9)))
```

```
<class 'float'>
<class 'sympy.core.numbers.Integer'>
```

Die erste Zahl ist vom Typ *float*, einem Standard-Datentyp von Python. Die zweite Zahl ist allerdings ein Objekt einer Klasse aus der *sympy*-Bibliothek! Sehen wir uns ein zweites Beispiel an: Die Wurzel aus 8:

```
[3]: print(math.sqrt(8))
sympy.sqrt(8)
```

2.8284271247461903

[3]:  $2\sqrt{2}$

*sympy* gibt jetzt nicht das (ungenaue) Ergebnis aus, sondern das symbolische Ergebnis  $2\sqrt{2}$  in der L<sup>A</sup>T<sub>E</sub>X-Schreibweise. Die Anzahl der Nachkommastellen ist beim Datentyp *float* begrenzt (IEEE754). *sympy* kann hier aber auch eine beliebige Anzahl Nachkommastellen korrekt ausgeben. Wir verwenden hierfür die Methode *evalf*, um 30 Stellen korrekt auszugeben:

```
[4]: sympy.sqrt(8).evalf(40)
```

[4]: 2.828427124746190097603377448419396157139

Diese symbolische Berechnung führt auch dazu, dass wir keine Fehler mehr erhalten, die auf den begrenzten Möglichkeiten der Speicherung von Fließkommazahlen beruhen. Berechnen wir die Wurzel aus 8 und quadrieren dieses Ergebnis, sollte eigentlich wieder 8 herauskommen:

```
[5]: math.sqrt(8)**2
```

[5]: 8.0000000000000002

Tut es aber nicht! Berechnen wir das Ergebnis mit *sympy*, kommt exakt wieder die Zahl 8 heraus:

```
[6]: sympy.sqrt(8)**2
```

[6]: 8

Wollen wir das symbolische Ergebnis als Kommazahl ausgeben, so können wir wie schon gezeigt *evalf* verwenden, oder alternativ die Funktion *N()*:

```
[7]: result = sympy.sqrt(10)
result.evalf()
```

[7]: 3.16227766016838

```
[8]: sympy.N(result)
```

[8]: 3.16227766016838

## 1.1 Beispiele mit sympy

*sympy* ist unglaublich mächtig! Es ist sozusagen die kostenlose Version der kommerziellen Tool *Mathematica* oder *Maple*. Einige Funktionen heissen sogar exakt gleich (wie das schon gezeigte *evalf*).

Wir wollen uns hier noch einige Beispiele ansehen, um die Mächtigkeit von *sympy* zu demonstrieren. Wer tiefer einsteigen möchte, dem bieten sich natürlich viele Tutorials im Internet bzw. die Dokumentation auf der Webseite von *sympy* an (<https://www.sympy.org/>).

Für die folgenden Beispiele importieren wir zuerst die hierfür benötigten Module. Für die “schnelle” Berechnung zwischendurch kann man ausnahmsweise auch einen `*` - Import verwenden (aber Vorsicht: Es werden ggf. andere Funktionen oder Klassen dadurch überschrieben!).

```
[9]: ## Beispiele für Symbolische Mathematik mit SymPy
# from sympy import *
from sympy import symbols
from sympy.core.function import diff
from sympy.polys.polyfuncs import interpolate
from sympy.integrals.integrals import integrate
```

Zuerst definieren wir symbolische Variablen. Das geht mit der Methode *symbols*, der wir Komma- oder Leerzeichengetrennt gleich mehrere Variablennamen übergeben können. Die Methode gibt ein Tupel zurück.

```
[10]: # Definiere symbolische Variablen
x = symbols("x")

## auch möglich
x, y = symbols("x y")
print(type(x))
```

```
<class 'sympy.core.symbol.Symbol'>
```

Wir definieren eine Funktion  $f$  mit  $f(x) = 3x^2 - 2x + 5$

```
[11]: f = 3*x**2 - 2*x + 5
f
```

```
[11]: 3x2 - 2x + 5
```

Wie lautet die erste Ableitung dieser Funktion? Ermitteln wir diese mit der Funktion *diff*:

```
[12]: diff(f)
```

```
[12]: 6x - 2
```

Welchen Wert hat die 1. Ableitung an der Stelle  $x = 3$ ?

```
[13]: diff(f).evalf(subs={x:3})
```

```
[13]: 16.0
```

Mit der Funktion *integrate* können wir das unbestimmte Integral unserer Funktion  $f(x)$  ermitteln, also  $\int x^3 - x^2 + 5x dx$ . Das Ergebnis ist  $x^3 - x^2 + 5x (+c)$

```
[14]: integrate(f)
```

```
[14]: x3 - x2 + 5x
```

*integrate* kann als weiteres Argument ein Tupel entgegennehmen, das als erstes die Variable, nach der integriert werden soll, sowie die untere und obere Grenze des bestimmten Integrals enthält. Wir wollen beispielsweise das Integral von  $f(x)$  für den Bereich 0 bis 5 ausrechnen, also

$$\int_0^5 3x^2 - 2x + 5 dx$$

so liefert uns *integrate* das Ergebnis 125:

```
[15]: integrate(f, (x,0,5))
```

```
[15]: 125
```

Nehmen wir an, wir kennen unser Polynom  $f(x)$  gar nicht, sondern nur empirisch ermittelte Messwerte und wollen ein Polynom erstellen, das sich möglichst exakt durch die gemessenen Punkte bewegt. Hier hilft die Funktion *interpolate*. Als erstes Beispiel geben wir für  $y$  die Werte 1, 4 und 9 an. Beginnen die  $x$ -Werte bei 1 und werden jeweils inkrementiert, braucht man diese nicht extra anzugeben:

```
[16]: interpolate((1,4,9),x)
```

```
[16]: x2
```

```
[17]: interpolate((6,13,26),x)
```

```
[17]: 3x2 - 2x + 5
```

Zuletzt wollen wir noch eine Gleichung lösen: Berechne die Nullstellen der Funktion

$$f(x) = x^2 - 2x - 8$$

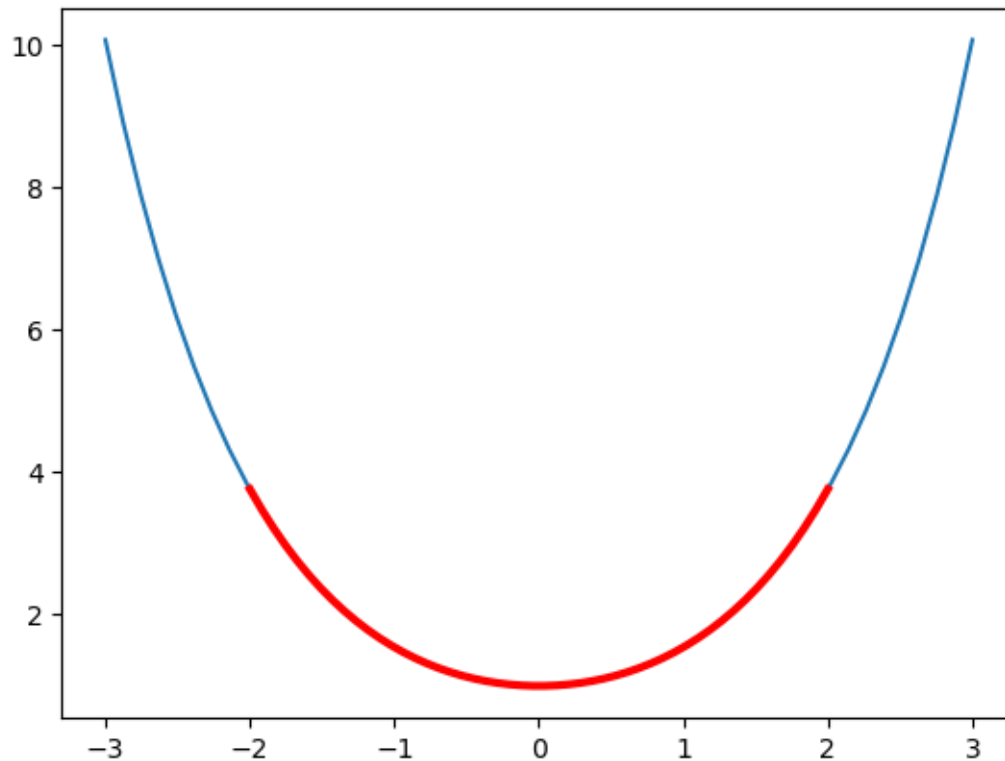
```
[18]: from sympy import *  
solve(x**2-2*x-8)
```

```
[18]: [-2, 4]
```

## 1.2 Beispiel: Berechnen der Länge einer Kurve

Wir wollen die Länge der Kurve für die Funktion  $f(x) = \frac{1}{2}(e^x + e^{-x})$  im Bereich von  $[-2; 2]$  berechnen.

```
[19]: import matplotlib.pyplot as plt  
import numpy as np  
  
x = np.linspace(-3,3,50)  
xb = np.linspace(-2,2,50)  
  
y = [1/2*(np.exp(x)+np.exp(-x)) for x in x]  
yb = [1/2*(np.exp(x)+np.exp(-x)) for x in xb]  
  
plt.plot(x,y)  
plt.plot(xb,yb, c="r", linewidth=3)  
plt.show()
```



Die Formel für die Berechnung der Länge eines Kurvenstücks lautet:

$$L = \int_a^b \sqrt{1 + (f'(x))^2}$$

Mit *sympy* lässt sich die Länge einfach berechnen:

```
[20]: from sympy import *

# Wir beschränken die symbolische Variable x auf reelle Zahlen (also keine
# ↪komplexen), dies beschleunigt die Berechnung.
x = symbols("x", real=True)

# Die Funktion für die Berchnung einer Kurve. Wir verwenden sympy.Rational, um
# ↪den Bruch 1/2 nicht als Fließkommazahl, sondern als sympy-Objekt zu
# ↪erstellen.
f = Rational(1,2) * (exp(x)+exp(-x))

# 1. Ableitung bestimmen. Die Methode doit() wertet den Ausdruck direkt aus.
deriv = diff(f,x).doit()

# Bestimme das Integral in den Grenzen von -2 bis 2
length = Integral(sqrt(1+deriv**2), (x,-2,2))
length
```

[20] : 
$$\int_{-2}^2 \sqrt{\left(\frac{e^x}{2} - \frac{e^{-x}}{2}\right)^2 + 1} dx$$

Schließlich wandeln wir dieses Ergebnis noch in einen numerischen Wert um:

[21] : `N(length)`

[21] : 7.25372081569404