SymPy_SymbolischeMathematik

February 3, 2023

1 Symbolische Mathematik mit SymPy

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

SymPy ist unglaublich mächtig! Es ist sozusagen die kostenlose Version der kommerziellen Tool *Mathematica* oder *Maple*. Einige Funktionen heissen sogar exakt gleich (wie zum Beispiel *evalf*).

Wir wollen uns hier 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/). Wer ernsthaft mit *SymPy* arbeiten möchte, dem empfehle ich das Buch *Symbolic Computation with Python and SymPy* von Davide Sandona (ISBN: 979-8489815208).

1.1 Installation

Falls Anaconda verwendet wird, sollte *SymPy* bereits installiert sein. Ansonsten kann SymPy wie folgt installiert werden:

- Unter Anaconda: *conda install -c anaconda sympy*
- Mit PIP: pip install sympy

Was genau ist nun *SymPy*? 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*. Außerdem ist die 3, die von *SymPy* ausgegeben wird, anders formatiert. Der Grund hierfür liegt darin, dass *SymPy* LaTeX-Code generiert. Um welche Objekte handelt es sich hier? Geben wir mal den Typ des jeweiligen Objekts aus:

[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) Fließkomma-Ergebnis aus, sondern das symbolische Ergebnis $2\sqrt{2}$ in der LaTeX-Schreibweise. Die Anzahl der Nachkommastellen ist beim Datentyp *float* begrenzt (IEEE754). *SymPy* kann hier aber auch eine definierte Anzahl Nachkommastellen korrekt ausgeben (standardmäßig sind es 15 Stellen). Wir verwenden hierfür die Methode *evalf*, um 30 Stellen korrekt auszugeben. Mit 30 Stellen sind hier die Gesamtzahl der Stellen gemeint, also Vor- und Nachkommastellen.

[4]:
```
sympy.sqrt(8).evalf(30)
```

[4]: 2.82842712474619009760337744842

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]: 8.000000000000002

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(). Hier zum Beispiel $\sqrt{10}$.

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

[7]: 3.16227766016838

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

[8]:

3.16227766016838

1.2 Symbole (Variablen) und Funktionen definieren

Für die folgenden Beispiele importieren wir für die "schnelle" Berechnung ausnahmsweise alle *SymPy*-Funktionen mit einem *. Aber Vorsicht: Es werden ggf. andere Funktionen oder Klassen dadurch überschrieben!

[9]:
```
# Alles mit einem * importieren (eigentlich keine gute Idee!)
from sympy import *
```

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. Möchte man nur eine symbolische Variable erstellen, kann man auch den Konstruktor der Klasse *Symbol* verwenden.

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

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

[10]:
```
sympy.core.symbol.Symbol
```

1.2.1 Faktorisieren eines Polynoms mit *factor*

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

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

[11]:
$$3x^2 - 2x - 5$$

Können wir dieses Polynom durch Faktoren darstellen, um die Nullstellen zu finden?

[12]:
```
f.factor()
```

[12]:
$$(x + 1)(3x - 5)$$

Bei *SymPy* hat man häufig die Wahl, ob man eine Methode auf einem Objekt ausführt, oder direkt eine Funktion der Klasse aufruft. Um das Polynom zu faktorisieren kann man hier entweder die Methode *factor* auf dem Objekt oder die Funktion *factor* aus dem *SymPy*-Modul aufrufen:

[13]:
```
# Statt Methode: Funktionsaufruf mit Übergabe des Objekts als Argument
# statt f.factor()
factor(f)
```

[13]:
$$(x + 1)(3x - 5)$$

1.2.2 Gleichungen erstellen und lösen mit *Eq* und *solveset*

Die Nullstellen wären hier also -1 und $\frac{5}{3}$. Wir können die Nullstellen natürlich auch von SymPy berechnen lassen. Dazu erstellen wir ein Objekt der Klasse *Equation*. Der Konstruktor erwartet als erstes Argument die linke, als zweite Argument die rechte Seite der Gleichung.

[14]:
```
eq1 = Eq(3*x**2 - 2*x - 5, 0)
eq1
```

[14]:
$$3x^2 - 2x - 5 = 0$$

Mit *solveset* können wir die Gleichung lösen. Die Methode gibt ein Set, also eine Menge zurück, da es auch mehrere Lösungen geben kann.

[15]:
```
solveset(eq1)
```

[15]:
$$\left\{-1, \frac{5}{3}\right\}$$

1.2.3 Vereinfachen von Funktionen: mit *expand* (Ausmultiplizieren)

Mit der Funktion *expand* können wir auch ausklammern, z.B. $f(x) = (x-3)(x+5)(x+\pi)$

[16]:
```
f2 = (x-3)*(x+5)*(x+pi)
f2
```

[16]:
$$(x - 3)\,(x + 5)\,(x + \pi)$$

[17]:
```
f2.expand()
```

[17]:
$$x^3 + 2x^2 + \pi x^2 - 15x + 2\pi x - 15\pi$$

1.2.4 Vereinfachen von Funktionen: mit *simplify*

simplify vereinfacht einen Ausdruck. Allerdings liefert es nicht immer einen Ausdruck, den man wirklich als *Vereinfachung* interpretieren würde. Als erstes Beispiel betrachten wir die Funktion $f(x) = sin^2(x) + cos^2(x)$, was bekanntlich 1 ergibt.

[18]:
```
expr1 = sin(x)**2 + cos(x)**2
expr1.simplify()
```

[18]:
$$1$$

Als zweites Beispiel betrachten wir die Funktion

$f(x) = \frac{x^3+x^2-x-1}{x^2+2x+1}$

Diese Funktion kann man per Hand zum Beispiel mit einer Polynomdivision vereinfachen. Einfacher geht es mit *simplify*:

[19]:
```
expr2 = (x**3 + x**2 - x - 1)/(x**2 + 2*x + 1)
expr2.simplify()
```

[19]:
$x - 1$

1.2.5 Lösen von Linearen Gleichungssystemen

Wir können auch ein Gleichungssystem lösen! Hierzu verwenden wir die Methode *linsolve*. Zum Beispiel:

$$-3x - 2y + 2z = 1 \quad (1)$$
$$-2x + 5y - 6z = 4 \quad (2)$$
$$4x + 3y - 2z = 2 \quad (3)$$

[20]:
```
x,y,z = symbols("x y z")
f1 = Eq(-3*x-2*y+2*z,1)
f2 = Eq(-2*x+5*y-6*z,4)
f3 = Eq(4*x+3*y-2*z,2)

linsolve([f1,f2,f3], x,y,z)
```

[20]:
$\{(-1,\ 4,\ 3)\}$

1.2.6 Differenzieren mit *diff*

Wie lautet die erste Ableitung der Funktion $f(x) = 3x^2 - 2x - 5$? Ermitteln wir diese mit der Funktion *diff*. Gibt es nur eine Variable, muss man diese nicht angeben. Ansonsten gibt man als Argument die Variable an, nach der differenziert werden soll.

[21]:
```
f = 3*x**2 - 2*x - 5
diff(f)
```

[21]:
$6x - 2$

Oder:

[22]:
```
diff(f,x)
```

[22]:
$6x - 2$

Wir definieren eine Funktion mit zwei Variablen. Nun kann man partiell differentieren:

$f_2(x) = 7x^3 + 8x - 2y^2$

[23]:
```
f2 = 7*x**3 - 2*y**2 + 8 *x
f2
```

[23]:
$7x^3 + 8x - 2y^2$

[24]:
```
# Partiell ableiten nach x
diff(f2,x)
```

[24]:
$21x^2 + 8$

[25]:
```
# Partiell ableiten nach y
diff(f2,y)
```

[25]:
$-4y$

Wie lautet die zweite Ableitung von $f(x) = 3x^2 - 2x - 5$?

[26]:
```
diff(f,x,2)
```

[26]:
6

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

[27]:
```
diff(f).evalf(subs={x:3})
```

[27]:
16.0

1.2.7 Integrieren mit *integrate*

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 $x^3 - x^2 + 5x\,(+c)$

[28]:
```
integrate(f)
```

[28]:
$x^3 - x^2 - 5x$

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:

[29]:
```
integrate(f, (x,0,5))
```

[29]:
75

1.2.8 Polynom bestimmen mit *interpolate*

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:

[30]:
```
interpolate((1,4,9),x)
```

[30]:
x^2

[31]:
```
interpolate((6,13,26),x)
```

[31]: $3x^2 - 2x + 5$

1.2.9 Gleichungen lösen mit *solve*

Wir wollen eine Gleichung lösen: Berechne die Nullstellen der Funktion

$f(x) = x^2 - 2x - 8 = 0$

[32]:
```
# Beispiel 1
solve(x**2-2*x-8)
```

[32]:
```
[-2, 4]
```

[33]:
```
solve(5*sin(x))
```

[33]:
```
[0, pi]
```

1.2.10 Vergleich von Ausdrücken

Mit etwas mathmatischem Schulwissen, Stichwort Binomische Formeln, weiß man, dass folgende zwei Gleichungen mathematisch gleich sind:

$f_1(x) = (2x + 3)^2$ und $f_1(x) = 4x^2 + 12x + 9$

Vergleichen wir dies mit dem == Operator von Python ist das Ergebnis allerdings *False*. SymPy-Objekte bieten jedoch die Methode *equals* an (Java-Programmierer kennen das!). Diese liefert *True*:

[34]:
```
from sympy import *
f1 = (2*x+3)**2
f2 = 4*x**2 +12*x + 9
x = Symbol("x")
print(f1==f2)
print(f1.equals(f2))
```

```
False
True
```

1.3 Beispiel: Berechnen der Länge einer Kurve

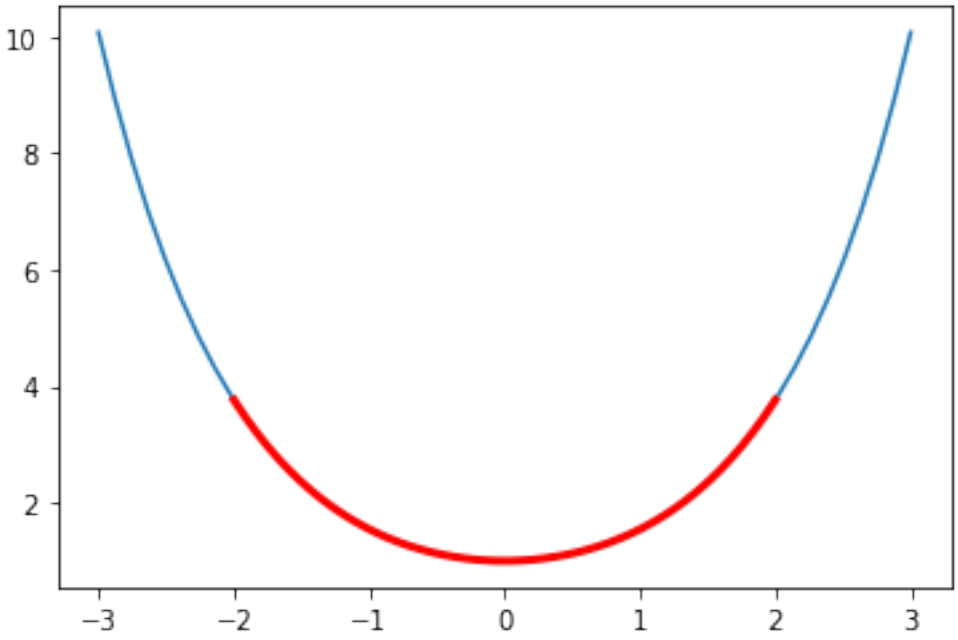
Wir wollen die Länge der Kurve für die Funktion $f(x) = \frac{1}{2}(e^x + e^{-x})$ in den Grenzen von $[-2; 2]$ berechnen (in der Grafik den rot markierten Bereich der Kurve).

[35]:
```
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:

[36]:
```
from sympy import *

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

# Die Funktion für die Berechnung 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
```

[36]:
$$\int_{-2}^{2} \sqrt{\left(\frac{e^{x}}{2} - \frac{e^{-x}}{2}\right)^{2} + 1}\, dx$$

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

[37]:
```
N(length)
```

[37]:
7.25372081569404