

Vorlesung 11: Analytisches Rechnen mit SymPy

Till Bargheer, Hendrik Weimer

Institut für Theoretische Physik, Leibniz Universität Hannover

Programmieren für Physikerinnen und Physiker, 13.01.2020

1. Einführung
2. Erste Schritte in Python
3. Den Ablauf eines Programms steuern
4. Zusatzfunktionen durch Python-Module
5. Wichtige Datentypen
6. Daten eingeben und ausgeben
7. Datenvisualisierung mit Matplotlib
8. Datenvisualisierung mit Matplotlib Teil II
9. SciPy: Module für die Wissenschaft
10. Optimierung mit SciPy
11. **Analytisches Rechnen mit SymPy**
12. Objektorientiertes Programmieren
13. Bonusthema

- ▶ Numerische Lösungen sind nicht exakt
- ▶ Numerische Berechnungen können sehr lange dauern
- ▶ Analytische Lösungen sind exakt und meist schnell, lassen sich aber nur in seltenen Fällen finden
- ▶ Guter Mittelweg: Analytische Vereinfachungen bis es nicht mehr weiter geht, dann numerisch Lösen

SymPy: Modul für analytisches Rechnen

- ▶ Website: <https://www.sympy.org/>
- ▶ Funktionsumfang vergleichbar mit Computer-Algebra-Systemen wie Mathematica
- ▶ Ableitungen, Integrale, Fourier-Transformationen, Gleichungen lösen, Kombinatorik, Lineare Algebra, Geometrie etc.
- ▶ Kennt viele Funktionen: `factorial`, `binomial`, `exp`, `log`, `sin`, `sinh`, `gamma`, `hyper`, `DiracDelta`, `Bessel`, ...
- ▶ Vollständig in Python integriert, mit NumPy/SciPy/Matplotlib kombinierbar

Symbolische Variablen definieren

- ▶ In Python müssen alle Variablen zuerst definiert werden, dies gilt auch für symbolische Variablen für analytische Berechnungen
- ▶ Sympy stellt hierfür die Funktion `symbols` zur Verfügung
- ▶ Mehrere symbolische Variablen durch Komma/Leerzeichen getrennt

```
>>> from sympy import *
>>> x
Traceback (most recent call last):
...
NameError: name 'x' is not defined
>>> x = symbols('x')
>>> x
x
>>> type(x)
<class 'sympy.core.symbol.Symbol'>
>>> x + 1
x + 1
```

Einfache Operationen

```
>>> a, b = sympy.symbols('a b')
>>> a+b
a + b
>>> sympy.atan(1)
pi/4
>>> sympy.sqrt(8)
2*sqrt(2)
>>> expr = x * (x + 1)
>>> expand(expr)
x**2 + x
```

- ▶ Besser lesbare Ausgabe im Python-Terminal:
`sympy.init_printing()`
- ▶ Im Folgenden nehmen wir an, dass die benötigten SymPy-Funktionen in den Hauptnamensraum importiert wurden

```
>>> from sympy import *
>>> init_printing()
```

```
>>> x = Symbol('x')
>>> y = x + 1
>>> x = 2
>>> y
x + 1
```

- ▶ Problem: $x = 2$ ändert die Python-Variable x , hat aber keinen Einfluss auf andere SymPy-Ausdrücke, die x enthalten!
- ▶ Lösung: subs-Funktion (Substitution)

```
>>> y.subs(x, 2)
3
>>> y
x + 1
```

- Ausdrücke an einem numerischen Punkt auswerten
- Unterausdrücke durch weitere Ausdrücke ersetzen

Vereinfachungen

- ▶ Ausdrücke vereinfachen: `simplify`

```
>>> simplify(sin(x)**2 + cos(x)**2)
1
```

- ▶ Auf Gleichheit prüfen: Ist `simplify(a-b)` gleich Null?

```
>>> a = (x + 1)**2
>>> b = x**2 + 2*x + 1
>>> simplify(a - b)
0
```

- ▶ Achtung: Python macht aus rationalen Ausdrücken floats!

```
>>> simplify(x**(1/2)-sqrt(x))
-sqrt(x) + x**0.5
```

Ausweg: `Rational(1/2)` benutzen. Oder: Einen der beiden Integer mit der `S`-Funktion zu einem SymPy-Objekt machen

```
>>> simplify(x**(S(1)/2)-sqrt(x))
0
```


Weitere Vereinfachungsfunktionen

- ▶

```
>>> expand((x + 1)**2)
x**2 + 2*x + 1
>>> factor(x**3 - x**2 + x - 1)
(x - 1)*(x**2 + 1)
```
- ▶ Ausdrücke nach Potenzen sortieren:

```
>>> collect(x*y + x - 3 + 2*x**2 - z*x**2 + x**3, x)
x**3 + x**2*(-z + 2) + x*(y + 1) - 3
```
- ▶ Kürzen, Partialbruchzerlegung:

```
>>> cancel((x**2 + 2*x + 1)/(x**2 + x))
(x + 1)/x
>>> apart(2*x/(x**2 - 1))
1/(x + 1) + 1/(x - 1)
```
- ▶ Trigonometrische Funktionen:

```
>>> trigsimp(sin(x)*tan(x)/sec(x))
sin(x)**2
>>> expand_trig(sin(x + y))
sin(x)*cos(y) + sin(y)*cos(x)
```

```
>>> diff(sin(x), x)
cos(x)
```

- ▶ Höhere Ableitungen:

```
>>> diff(sin(x), x, 2)
-sin(x)
```

- ▶ Verschiedene Ableitungen nacheinander:

```
>>> diff(exp(2*x)*y**2, x, y)
4*y*exp(2*x)
```

- ▶ Ausdruck für Ableitung ohne Ausführung:

```
>>> Derivative(sin(x), x)
Derivative(sin(x), x)
>>> Derivative(sin(x), x).doit()
cos(x)
```

- Taylor-Entwicklung

```
>>> cos(x).series()
1 - x**2/2 + x**4/24 + O(x**6)
```

- Entwicklungspunkt und Ordnung kontrollieren:

```
>>> log(x).series(x, 1, 3)
-1 - (x - 1)**2/2 + x + O((x - 1)**3, (x, 1))
>>> log(x).series(x, 1, 3).removeO()
x - (x - 1)**2/2 - 1
```

- Entwicklung um unendlich (Notation: oo):

```
>>> expr = sqrt(1+x**2)
>>> expr.series(x, oo, 2)
1/(2*x) + x + O(x**(-2), (x, oo))
```

- Laurent-Entwicklung:

```
>>> expr = gamma(x - 3)
>>> expr.series(x, 0, 1)
-1/(6*x) + EulerGamma/6 - 11/36 + O(x)
```

- Auswertung an einem Punkt mit singulären Unterausdrücken:

```
>>> (sin(x)/x).subs(x, 0)
nan
>>> limit(sin(x)/x, x, 0)
1
```

- Richtung vorgeben:

```
>>> limit(1/x, x, 0)
oo
>>> limit(1/x, x, 0, '-')
-oo
```

- SymPy kann mit unendlich richtig umgehen:

```
>>> oo + 1
oo
>>> 1000 < oo
True
>>> 1/oo
0
```

Folgen und Reihen

Folgen definieren:

```
>>> a_n = 1/n  
>>> b_n = 1/factorial(n)
```

Werte eine Folge aus (list comprehension):

```
>>> [b_n.subs(n, i) for i in range(8)]  
[1, 1, 1/2, 1/6, 1/24, 1/120, 1/720, 1/5040]
```

Unendliche Reihe: Summation einer Folge:

```
>>> summation(a_n, [n, 1, oo])  
oo  
>>> summation(b_n, [n, 0, oo])  
E  
>>> summation(b_n, [n, 0, oo]).n(50)  
2.7182818284590452353602874713526624977572470937000
```

- ▶ Unbestimmtes Integral:

```
>>> integrate(cos(x))  
sin(x)
```

- ▶ Bestimmtes Integral:

```
>>> integrate(exp(-x**2), (x, -oo, oo))  
sqrt(pi)  
>>> integrate(1/log(t), (t, 0, x))  
li(x)
```

- ▶ Wenn SymPy auch nicht weiter weiß:

```
>>> integrate(x**x)  
Integral(x**x, x)
```

- Definition:

$$F(k) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x k} dx.$$

```
>>> ft = fourier_transform(exp(-x**2+x), x, k)
>>> ft
sqrt(pi)*exp((2*I*pi*k - 1)**2/4)
```

- Inverse Fouriertransformation:

$$f(k) = \int_{-\infty}^{\infty} F(x) e^{2\pi i x k} dx.$$

```
>>> inverse_fourier_transform(ft, k, x)
exp(-x**2 + x)
```

Matrizen

Definiere Matrix mit symbolischen Einträgen:

```
>>> u = symbols('u', cls = Function)
>>> mat = Matrix([[u(i, j) for j in range(2)]
                  for i in range(2)])

>>> mat
Matrix([
[u(0, 0), u(0, 1)],
[u(1, 0), u(1, 1)]])
```

Determinante, charakteristisches Polynom:

```
>>> mat.det()
u(0, 0)*u(1, 1) - u(0, 1)*u(1, 0)
>>> factor(mat.charpoly(x))
x**2 - x*u(0, 0) - x*u(1, 1)
      + u(0, 0)*u(1, 1) - u(0, 1)*u(1, 0)
```

Weitere Methoden: `eigenvals`, `eigenvects`, `nullspace`, ...

Gleichungen lösen

Gleichungen mit Eq aufstellen: == testet auf exakte strukturelle Identität, Eq bleibt im Zweifel unevaluiert:

```
>>> (x == y, Eq(x, y), Eq(x, y).subs(x, y))  
(False, Eq(x, y), True)
```

```
>>> solveset(Eq(x**2, 1))  
{-1, 1}
```

```
>>> solveset(x**2-1)  
{-1, 1}
```

```
>>> solveset(sin(x) - 1, x)  
ImageSet(Lambda(_n, 2*_n*pi + pi/2), Integers)
```

Wenn exakte Lösung nicht möglich ist: nsolve löst numerisch.

```
>>> solveset(Eq(1-x, tan(x)))  
ConditionSet(x, Eq(-x - tan(x) + 1, 0),  
              Complexes(Reals x Reals, False))  
>>> nsolve(Eq(1-x, tan(x)), x, 1)  
0.479731007280410
```

Differentialgleichungen

Deklariere y als Funktion:

```
>>> y = symbols('y', cls = Function)
>>> y = Function('y') # Alternative
```

Benutze dsolve: $y'(t) + y(t) = 0$

```
>>> dsolve(Derivative(y(t), t) + y(t))
Eq(y(t), C1*exp(-t))
```

Beispiel: Gedämpfter getriebener harmonischer Oszillator.

Lineare Dämpfung: γ . Eigenkreisfrequenz: ω_0 . Treibende Kraft: $f \sin(t)$.

Bewegungsgleichung: $\ddot{x}(t) + 2\gamma \dot{x}(t) + \omega_0^2 x(t) = f \sin(t)$

```
>>> eom = Eq(y(t).diff(t, t) + 2*g*y(t).diff(t)
          + w0*y(t), f*sin(t))
>>> dsolve(eom.subs({w0: 1, f: 1, g: 1}), y(t))
Eq(y(t), (C1 + C2*t)*exp(-t) - cos(t)/2)
```

Ausdrücke manipulieren

Schon gesehen: `expand`, `factor`, `collect`, `simplify`

Weitere Funktionen: `trigsimp`, `expand_trig`

Substitutionen: `expr.subs(x, y)` ersetzt alle `x` in `expr` durch `y`

Weitere Substitutionsfunktionen: `rewrite`, `replace`

- ▶ Funktionale Identitäten:

```
>>> sin(x).rewrite(exp)
-I*(exp(I*x) - exp(-I*x))/2
>>> (sin(x) + cos(x)).rewrite(cos, exp)
exp(I*x)/2 + sin(x) + exp(-I*x)/2
```

- ▶ Ersetzungen:

```
>>> f = log(sin(x)) + tan(sin(x**2))
>>> f.replace(sin, cos)
log(cos(x)) + tan(cos(x**2))
>>> f.replace(sin, lambda arg: sin(2*arg))
log(sin(2*x)) + tan(sin(2*x**2))
```

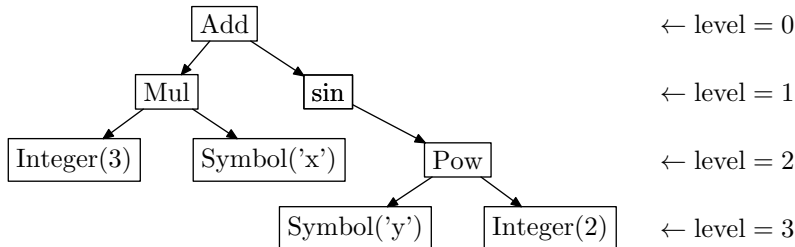
Expression Tree

Sympy-Ausdrücke haben eine Baumstruktur.

Die interne Struktur kann mit `srepr` sichtbar gemacht werden:

```
>>> sprepr(3*x + sin(y**2))
```

```
Add(Mul(Integer(3), Symbol('x')), sin(Pow(Symbol('y'), Integer(2))))
```



Der Baum ist in level unterteilt.

Das level eines Knotens ist seine Entfernung von der Wurzel.

Expression Tree: head und arguments

Jeder Sympy-Ausdruck `expr` lässt sich zerlegen in einen head (Wurzelknoten, zugänglich mit `expr.func`) und Argumente (Knoten auf Level 1, zugänglich mit `expr.args`).

Für jeden Ausdruck `expr` gilt: `expr == expr.func(*expr.args)`

```
>>> expr = 3*x + sin(y**2)
>>> expr.func
<class 'sympy.core.add.Add'>
>>> expr.args
(3*x, sin(y**2))
>>> expr.func(*expr.args)
3*x + sin(y**2)
>>> expr.args[1]
sin(y**2)
```

Arbeiten mit dem Expression Tree

use: Wende eine Funktion nur auf ein bestimmtes Level an

```
>>> expr = x * (x + y)**2 + 1
>>> use(expr, expand, level=2)
x*(x**2 + 2*x*y + y**2) + 1
```

EPath: Spezifiziert Unterausdrücke. Syntax (ähnlich wie Dateisystem):

- ▶ `"/"`: Trennt verschiedene Level
- ▶ `"*"`: Selektiert alle Ausdrücke eines Levels
- ▶ `"[n]"`: Selektiert Knoten n des Levels

EPath.select: Extrahiere Unterausdrücke

EPath.apply: Wende Funktionen auf Unterausdrücke an

```
>>> expr = 2*cos(x) + 3*sin(x + y)*cos(x + y)
>>> EPath("*/cos").select(expr)
[cos(x + y), cos(x)]
>>> EPath("*/cos").apply(expr, expand_trig)
3*(-sin(x)*sin(y) + cos(x)*cos(y))*sin(x + y) + 2*cos(x)
```