Einführung in Sage

Dr.Jochen Schulz

10-10-11

Inhaltsverzeichnis

Kapitel 1

1.1 Einleitung

1.1.1 Was ist Sage?

Sage ist ein pythonbasiertes, objektorientiertes Open-source (GPL) Mathematik Software System, dass es seit dem 24. Februar 2005 gibt. In Sage findet man eine Alternative zu den 4 M's: Magma, Maple, Mathematica, Matlab.
Das (Haupt-)Interface ist der Browser, dennoch besitzt Sage Frontends für viele externe Software.
Zu den Stärken von Sage zählt sicherlich, dass es viele andere CAS und Libraries unter einer einheitlichen Oberfläche vereint (so z.B. Maxima, Pari, GAP, R, Magma, ...), sowie durch Python eine sehr mächtige Programmiersprache als Grundlage hat. Desweiteren ist der Source Code offen, es gibt ein umfangreiches Hilfesystem sowie viele freie Materialien im Internet.
Wie alles, so hat auch Sage trotz seiner vielen Stärken auch Schwächen, so ist der Befehlsumfang nicht so mächtig, wie der von Maple,Mthematica oder Matlab; außerdem fehlt eine standalone Entwicklungsumgebung. Als Alternative zum Webinterface findet sich Cantor.

Vorab:

Sage incl. online Testversion und gratis Download sowie einigen anderen Features findet sich unter http://www.sagemath.org/.
Hier ein paar grundlegende Tipps, Tricks und Dinge, die einfach beherzigt werden sollten:

- Mehrere Befehle in einer Zeile durch ; trennen.
- Bei Eingaben, die über mehrere Zeilen gehen, kann ein Zeilenumbruch durch <ENTER> erreicht werden.
- Das Auswerten eines Blocks erfolgt mit <SHIFT>+<ENTER>.
- Ein neues Eingabefeld erhält man durch klicken auf den blauen, horizontalen Balken
- _ refenziert die letzte Ausgabe.
- Löschen aller eigenen Variablen und Zurücksetzen auf den Anfangsstatus: reset()
- Das Feld aktivieren von \Typeset lässt alle Ausgaben von LaTeX rendern.

- Html- und/oder LaTeX-Dokumentation:<SHIFT>+<KLICK> auf den blauen Balken
- Autocompletion: mit der <TAB>-Taste erhält man alle möglichen Funktions- und/oder Variablen-Namen im gegebenen Kontext.
  Dies gilt insbesondere auch für Objektfunktionen (object.function())
- <command>?: gibt ausführliche Hilfe zu command an.
- help(<command>): öffnet ein Hilfefenster zu command.
- online Dokumentation:
  - Sage: http://www.sagemath.org/doc/index.html
  - Python: http://docs.python.org/

1.1.2 Überblick

Als Einstieg und kleine Motivation, wofür man Sage einsetzen kann, wollen wir erst einmal ein paar grundlegende Operationen vorstellen:

- Deklarieren von Variablen mit var(), z.B. var('a')
- Definieren von Variablen mit = , z.B. $a = 3$
- Definieren von Funktionen mit = , z.B. $f(x) = x^2 - 6 * x$
- Grenzwertbestimmung: f.limit(x=1, dir='$< plus|minus >$')
- Bilden von Ableitungen: f.differentiate(x)
- Lösen von Gleichungen: solve(f(x)==0, x)
- Berechnen numerischer Approximationen: float(f(sqrt(3)+ 4))
- Plotten einer Funktion: plot(sin,(0,4))
- symbolisch Integrieren: integrate(f,x)
- numerisch Integrieren: integrate(f,x,a,b)
- Faktorisieren: expand(f)
- Sortieren: f.collect(x)
- Partialbruchzerlegung: f.partial_fraction()
- vollständiges Vereinfachen: f.full_simplify()
- Vereinfachen mit radicals: f.radical_simplify()
- Matrix eingeben: matrix([[z1s1,z1s2],[z2s1,z2s2]])
- Vektor eingeben: vector([a,b,c])
- LGS lösen: A\b
- Matrixoperationen: $A + B, A - B, A * B$
- Matrix invertieren: A^{-1}; A.inverse()
- Substitutieren: f.subs(k=2)

Hiermit stehen uns nun einige Türen offen und wir wollen uns an ein paar Beispielen der Sagewelt nähern:

1. Kurvendiskussion

Wir wollen eine handelsübliche Kurvendiskussion führen. Dazu betrachten wir die durch die reelle Zahl a parametrisierte Funktionenschar:

$$f : x \quad \mapsto \quad \frac{2x^2 - 20x + 42}{x - 1} + a, \quad a \in \mathbb{R}$$

Wie bekommen wir nun diese Aufgabe mittels Sage gelöst? Zuerst sollten wir diese Funktionenschar wohl Sage "erklären", um weiter mit ihr arbeiten zu können. Dies geschiet, indem wir zuerst die Variable a und anschließend die Funktion selbst deklarieren:

```
var('a')
f(x) = (2*x^2-20*x +42)/(x-1)+a;f
```

Bemerkung: Die Multiplikationszeichen dürfen NICHT weggelassen werden.
Wir erhalten als Antwort

```
x |--> a + 2*(x^2 - 10*x + 21)/(x - 1)
```

Bemerkung: Es findet keine Ausgabe statt, wenn wir in unserer Eingabe ;f weglassen.
Wir möchten nun herausfinden, ob unsere Funktion Polstellen besitzt; dazu benutzen wir den limit Befehl:

```
f.limit(x=1, dir='minus')
```

```
x |--> -Infinity
```

```
f.limit(x=1, dir='plus')
```

```
x |--> +Infinity
```

Wir sehen nun, dass unsere Funktion f an der Stelle x=1 einen Pol hat. Wir verfahren weiter mit der Suche nach Nullstellen und wenden dafür den solve Befehl an:

```
solve(f==0,x)
```

```
[x == -1/4*a - 1/4*sqrt(a^2 - 32*a + 64) + 5, x == -1/4*a + 1/4*
   sqrt(a^2 - 32*a + 64) + 5]
```

Wir haben nun 2 Nullstellen gefunden und stellen uns sogleich die Frage nach der Ableitunge von f, die wir über den differentiate Befehl erhalten:

```
f.differentiate(x)
```

```
x |--> 4*(x - 5)/(x - 1) - 2*(x^2 - 10*x + 21)/(x - 1)^2
```

Da unsere Funktion offenbar eine Ableitung besitzt, interessiert uns, ob und, wenn ja, wo f Extrema besitzt sowie die Antwort auf die Frage, ob es sich um Minima oder MAxima handelt:

```
maxi = solve(f.differentiate(x)==0,x); maxi
```

```
[x == -2*sqrt(3) + 1, x == 2*sqrt(3) + 1]
```

```
float( ((f.diff(x)).diff(x))(maxi[0].rhs()) )
```

```
-1.1547005383792501
```

```
float( ((f.diff(x)).diff(x))(maxi[1].rhs()) )
```

```
1.1547005383792515
```

Wir erkennen die Existenz zweier Extremstellen, von denen eines ein Maximum, das andere ein Minimum ist.
Bemerkung: Als "Speicherplatz" der Nullstellen von f' besitzt maxi zwei Werte- diese werden mit maxi[0].rhs() bzw maxi[1].rhs() angesprochen.
Ohne den float() Befehl sähe die Ausgabe im ersten Fall so aus:

```
-1/18*((2*sqrt(3) - 1)^2 + 20*sqrt(3) + 11)*sqrt(3) + 2/3*sqrt(3) +
   8/3
```

und wäre somit unbrauchbar, um direkt ein Maximum oder Minimum erkennen zu können.
Zuletzt betrachten wir noch das Verhalten am Rande:

```
f.limit(x=oo); f.limit(x=-oo)
```

```
x |--> +Infinity
x |--> -Infinity
```

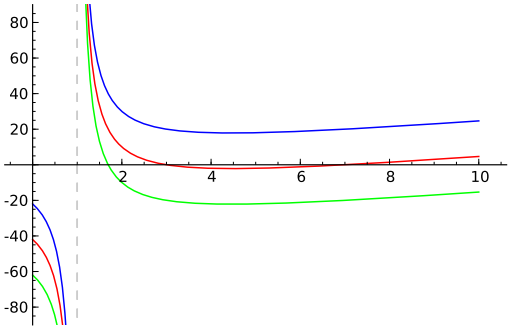
Bemerkung: Wir erkennen, in Sage stellt sich ∞ dar via oo, $-\infty$ entspricht offenbar -oo.
Viel interessanter jedoch ist die Tatsache, dass Sage mit jenen "Werten" direkt arbeiten zu können scheint.
Zuletzt wollen wir uns mit der grafischen Darstellung einiger Repräsentanten von f auseinandersetzen. Hierzu wollen wir zuerst 3 jener Repräsentanten deklarieren und diese anschließend mit dem plot Befehl visualisieren:

```
f0 = f(x, a=0)
f1 = f(x, a=-20)
f2 = f(x, a=20);f0,f1,f2
```

```
(2*(x^2 - 10*x + 21)/(x - 1), 2*(x^2 - 10*x + 21)/(x - 1) - 20, 2*(
   x^2 - 10*x + 21)/(x - 1) + 20)
```

```
p = plot(f0,detect_poles='show',xmin=0, xmax=10,color='red')
p += plot(f1,detect_poles='show',xmin=0, xmax=10,color='green')
p += plot(f2,detect_poles='show',xmin=0, xmax=10,color='blue'); p.
   show(ymin=-80, ymax=80)
```

Wir erhalten als Grafikausgabe:



2. Symbolisches Rechnen

Wir wollen uns nun ein wenig mit symbolischem Rechnen vertraut machen; dazu zählt das numerische Lösen von Integralen, sicherlich aber auch die Vereinfachung oder die Faktorisierung eines Terms. Wir betrachten einige Beispiele:
Wir möchten gerne das Integral $\int_0^\infty x^4 e^{-x} dx$ numerisch berechnen, dazu benötigen wir lediglich

```
integrate(x^4*exp(-x),x,0,oo)
```

und erhalten als Ausgabe

```
24
```

. Wollen wir eine Stammfunktion von $\frac{1+\sin(x)}{1+\cos(x)}$ berechnen, so genügt uns die Deklaration zweier Funktion f und g wie folgt:

```
f(x) = (1+sin(x))/(1+cos(x))
g = f.integrate(x);g
```

```
x |--> sin(x)/(cos(x) + 1) - log(cos(x) + 1)
```

Die Funktion g sieht eher unschön aus, wir möchten sie daher vereinfachen und benutzen dafür

```
g.full_simplify()
```

```
x |--> -((cos(x) + 1)*log(cos(x) + 1) - sin(x))/(cos(x) + 1)
```

Die Verschönerung hat hier nur bedingt geklappt, aber der Versuch war es wert und oft erhält man wirklich ein schöneres Ergebnis als zuvor. Ein wenig effektiver funktioniert es mit $(\frac{e^x-1}{e^{(1/2)x}+1})$

```
g = (exp(x)-1)/(exp(x/2)+1)
g.simplify_radical()
```

```
e^(1/2*x) - 1
```

Es gibt noch andere Möglichkeiten, Termen ein schöneres Aussehen zu geben. Betrachten wir den Term

```
x^4 - 10*x^3 + 35*x^2 - 50*x + 24
```

und fragen uns: Können wir ihn Faktorisieren? Eine Antwort darauf liefert

```
factor(x^4 - 10*x^3 + 35*x^2 - 50*x + 24)
```

```
(x - 4)*(x - 3)*(x - 2)*(x - 1)
```

Als kleine Probe können wir die Faktorisierung rückgängig machen:

```
expand(_)
```

```
x^4 - 10*x^3 + 35*x^2 - 50*x + 24
```

Bemerkung: Wir erinnern uns, dass _ eine Referenz auf die letzte Ausgabe ist.
Wir können Terme Terme auch bzgl. einer Variablen sortieren lassen:

```
var('b,a')
g = x^2+2*x+b*x^2+sin(x)+a*x
g.collect(x)
```

```
(b + 1)*x^2 + (a + 2)*x + sin(x)
```

Zuletzt steht uns hier noch die Partialbruchzerlegung zur Verfügung:

```
g = x^ 2/( x^ 2- 1)
g.partial_fraction()
```

```
1/2/(x - 1) - 1/2/(x + 1) + 1
```

3. Analytische Geometrie und Lineare Algebra

Wir wollen uns ein wenig mit analytischer Geometrie und linearer Algebra beschäftigen und zu Anfang mal den Schnittpunkt einer Ebene mit einer Geraden berechnen. Dazu sei die Ebene E gegeben durch

$$E : \vec{x} = \begin{pmatrix} 2 \\ 1 \\ -1 \end{pmatrix} + l \begin{pmatrix} 1 \\ -1 \\ -1 \end{pmatrix} + m \begin{pmatrix} -3 \\ 1 \\ 4 \end{pmatrix}, \quad l, m \in \mathbb{R}$$

und die Gerade g

$$g : \vec{x} = \begin{pmatrix} 3 \\ 0 \\ 1 \end{pmatrix} + k \begin{pmatrix} 4 \\ -1 \\ 2 \end{pmatrix}, \quad k \in \mathbb{R}$$

Gleichsetzen ergibt:

$$\begin{pmatrix} 2 \\ 1 \\ -1 \end{pmatrix} + l \begin{pmatrix} 1 \\ -1 \\ -1 \end{pmatrix} + m \begin{pmatrix} -3 \\ 1 \\ 4 \end{pmatrix} = \begin{pmatrix} 3 \\ 0 \\ 1 \end{pmatrix} + k \begin{pmatrix} 4 \\ -1 \\ 2 \end{pmatrix}$$

oder

$$\underbrace{\begin{pmatrix} 1 & -3 & -4 \\ -1 & 1 & 1 \\ -1 & 4 & -2 \end{pmatrix}}_{=: A} \underbrace{\begin{pmatrix} l \\ m \\ k \end{pmatrix}}_{=: L} = \underbrace{\begin{pmatrix} 1 \\ -1 \\ 2 \end{pmatrix}}_{=: b}$$

oder $AL = b$. Wir wollen nun lineare Gleichungssysteme (wie dieses) wie gewöhnlich durch MAtrizen beschreiben und via Sage lösen. Dazu benötigen wir ein paar Grundlagen:

- Definieren der Matrix A

```
A = matrix([[1,-3,-4],[-1,1,1],[-1,4,-2]]); A
```

```
[ 1 -3 -4]
[-1  1  1]
[-1  4 -2]
```

- Definieren des Vektors b

```
b = vector([1,-1,2])
```

- Lösen von $A\ L = b$

```
A.solve_right(b)
```

oder

```
A\b
```

ergibt

```
(6/5, 3/5, -2/5)
```

- Einsetzen in die Geradengleichung

```
x_s = matrix([g1,g2,g3]).subs(k=L[2]); x_s
```

```
[7/5 2/5 1/5]
```

So haben wir nun recht aufwandsarm den Schnittpunkt der beiden Objekte E und g erhalten. Wir geben noch einen kurzen Ausblick auf weitere Matrizenoperationen:

```
B = matrix([[1,0,0],[0,1,1],[1,1,1]])
A*B; A-B; A+B
```

```
[-3 -7 -7]  [ 0 -3 -4] [ 2 -3 -4]
[ 0  2  2]  [-1  0  0] [-1  2  2]
[-3  2  2]  [-2  3 -3] [ 0  5 -1]
```

Berechnen der Inversen (mit Probe)

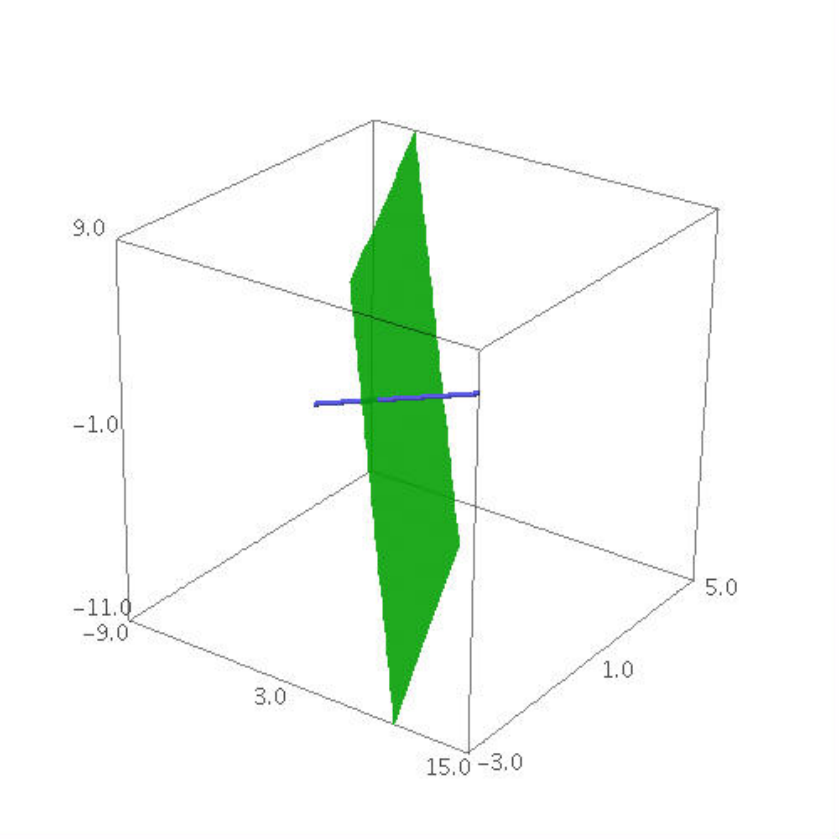
```
A^(-1), A*A^(-1)
```

```
[  -2/5 -22/15   1/15]
[  -1/5   -2/5    1/5]
[  -1/5  -1/15  -2/15]

[1 0 0]
[0 1 0]
[0 0 1]
```

Wir wollen uns das Ganze mal visualisieren, hierzu bietet uns Sage den 3D plot:

```
var('l,m'); E1 = 2+l-3*m; E2 = 1-l+m; E3 =-1-l+4*m
p = parametric_plot3d([E1,E2,E3],(l,-2,2),(m,-2,2), color='green',
   opacity=0.8)
var('k'); g1 = 3+4*k; g2 = -k; g3 = 1+2*k
p += parametric_plot3d( (g1,g2,g3), (k, -3, 3),thickness='3' )
p.show()
```



Bemerkung: In Sage lässt sich der 3D plot drehen.

4. Programmieren

- Wir möchten eine einzeilige Funktion definieren. Dies geschieht via

```
def <name>(<Argumente>) : return <Rueckgabe>
```

Beispiel:

```
def fd(ex) : return diff(ex)
fd(x^2)
```

```
2*x
```

- Wir können Objekte in Listen und Tupeln zusammenfassen:
Eine *Liste* ist in Sage (und Python) mit `[..,..]` gekennzeichnet

```
liste = [21,22,24,23]
liste.sort(); liste
```

```
[21, 22, 23, 24]
```

Ein *Tuple* ist in Sage (und Python) mit `(..,..)` gekennzeichnet

```
tuple = (liste[0], liste[2])
tuple, tuple[0]
```

```
((21, 24), 21)
```

Liste von ganzen Zahlen von `a` bis `b`

```
[a..b] ; range(a,b+1)
```

- Es gibt einzeilige (bedingte) Schleifen:

```
[<expr(var)> for <var> in <range|liste>]
[<expr(var)> for <var> in <range|liste> if <expr>]
```

Beispiel:

```
[m^2 for m in [1..5] ]
```

```
[1, 4, 9, 16, 25]
```

Beispiel mit Abfrage:

```
[m^2 for m in [1..5] if m%2==0]
```

```
[4, 16]
```

6. Zahlentheorie

Wir sind nun in der Lage, etwas kompliziertere Aufgabenstellungen zu bearbeiten:
1.Die Fermatschen Primzahlen sind gegeben durch $F_n = 2^{2^n} + 1$. Wir wollen die kleinste dieser Zahlen finden, die keine Primzahl ist. Dazu schreiben wir uns ein kleines Programm:

```
def F(n): return 2^(2^n)+1
[[F(m),is_prime(F(m))] for m in range(1,6)]
```

```
[[5, True], [17, True], [257, True], [65537, True], [4294967297,
   False]]
```

Uns interssieren nun noch die Teiler von F_5

```
divisors(int(F(5)))
```

```
[1, 641, 6700417, 4294967297]
```

2. Wir möchten eine Liste der Primzahlen bis 100:

```
menge = range(1,101)
[m for m in menge if is_prime(m)]
oder filter(is_prime,menge)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,
   61, 67, 71, 73, 79, 83, 89, 97]
```

3.Die Mersenne Primzahlen sind gegeben durch $2^p - 1$, dabei p Primzahl. Wir wollen die Mersenne Primzahlen im Bereich ≤ 200 bestimmen:

```
menge = range(1,201)
primes = [m for m in menge if is_prime(m)]
[2^m-1 for m in primes if is_prime(2^m-1)]
```

```
[3, 7, 31, 127, 8191, 131071, 524287, 2147483647,
   2305843009213693951,
 618970019642690137449562111, 162259276829213363391578010288127,
 170141183460469231731687303715884105727]
```

4.Wir wollen für die Zahlen ≤ 1000 bestimmen, wieviele Zahlen 1,2,3,... Teiler haben:

```
menge = range(1,1001)
liste = [number_of_divisors(int(m)) for m in menge]
[(i,len([m for m in liste if m==i]))for i in range(1,51)]
```

```
[(1, 1), (2, 168), (3, 11), (4, 292), (5, 3), (6, 110), (7, 2), (8,
180), (9, 8), (10, 22), (11, 0), (12, 97), (13, 0), (14, 5), (15,
   4),
...
```

Bis hierher haben wir einen kleinen Rundumblick genommen, einige sehr elementare Grundlagen kennengelernt und sind schon recht vertraut mit Sage. Vieles von dem, was wir bisher gesehen haben, wird uns wieder begegnen, wenn wir uns weiter mit Sage befassen.