Thematische Anleitungen

Release 6.10

The Sage Group

21.12.2015

Die folgenden Dokumente sind thematische Anleitungen, welche sich mit speziellen Einsatzgebieten von Sage befassen.

KAPITEL 1

Sage als Taschenrechner im Gymnasium

1.1 Einleitung

1.1.1 An wen richtet sich diese Anleitung?

Diese Anleitung richtet sich an Lehrpersonen, welche an Gymnasien im deutschsprachigen Raum unterrichten. Das Ziel ist, eine Anleitung zur Verfügung zu stellen, welche erklärt, wie das Computer-Algebra-System Sage anstelle der an Gymnasien häufig eingesetzten CAS Taschenrechner verwendet werden kann.

Als grober Leitfaden dienen Lehrpläne von Schweizer Gymnasien. Diese unterscheiden sich nicht stark von Lehrplänen in anderen deutschsprachigen Ländern. Daher wird diese Anleitung für alle deutschsprachigen Mathematiklehrpersonen nützliche Informationen enthalten.

Die bestehende englische Dokumentation von Sage ist sehr ausführlich und auf deutsch existiert ein gutes Tutorial für den Einstieg in Sage [1]. Es hat sich aber als schwer herausgestellt, die für den gymnasialen Unterricht relevanten Funktionen zu finden, da oft Fälle beschrieben werden, wie sie typischerweise in Hochschulmathematik anzutreffen sind.

Daher soll diese Anleitung als eigenständige Dokumentation der Funktionen von Sage aus Sicht einer Gymnasiallehrperson dienen. Dieses Dokument soll die Benutzung von Sage für Gymnasiallehrpersonen vereinfachen, ohne Vorkenntnisse von Sage oder der Programmiersprache Python vorauszusetzen.

1.1.2 Was ist Sage?

Sage ist ein quelloffenes Computer-Algebra-System, welches weltweit von hunderten Studenten und Forschern an verschiedenen Universitäten entwickelt wird, aber auch Privatpersonen und Schüler haben Beiträge geleistet. Es wurde und wird noch immer in vielen mathematischen Arbeiten benutzt und im Zuge dessen auch immer weiterentwickelt. Daher ist es sehr mächtig und löst komplexe Probleme in vielen, sehr spezialisierten Teilgebieten der Mathematik.

Trotzdem ist Sage auch als Computer-Algebra-System für an Gymnasien unterrichtete Mathematik gut geeignet. Es bietet einen grossen Funktionsumfang und kann kostenlos an Schulen eingesetzt werden und so teure Taschenrechner und Software ersetzen.

Die Tatsache, dass Sage quelloffen und kostenlos verfügbar ist, macht es speziell geeignet für Schulen. Das System kann von Schülern und Schulen problemlos auf allen Rechnern installiert werden, ohne dass man sich um Lizenzkosten und die Anzahl verfügbarer Lizenzen kümmern muss. Es kann auch verändert und angepasst werden und jede Person kann helfen, es weiter zu entwickeln.

Sage ist zum grossen Teil in der Programmiersprache Python geschrieben und kann auch in Python programmiert werden. Diese Sprache wird von Schulen weltweit in Einführungskursen ins Programmieren benutzt. Diese Tatsache

[1] http://www.sagemath.org/de/html/tutorial/

macht Sage umso geeigneter, um in Schulen - auch im Zusammenhang mit einer Einführung ins Programmieren - eingesetzt zu werden.

1.1.3 Wo bekomme ich Sage?

Die Onlineversion dieser Anleitung bietet eine erste Möglichkeit, die Beispiele direkt in Sage auszuprobieren und abzuändern, ohne selbst Sage installiert zu haben.

Sage kann aber auch lokal installiert oder online ausprobiert werden. Auf der Sage-Webseite findet man weitere Informationen dazu:

http://sagemath.org/

1.1.4 Hilfe abfragen

Bevor wir beginnen die für den gymnasialen (Mathematik-) Unterricht relevanten Funktionen von Sage ausführlich zu betrachten, soll hier erklärt werden, wie man für eine bestimmte Funktion die Dokumentation direkt in Sage abrufen kann. Dies geschieht durch einfaches Anhängen eines Fragezeichens (?) an den Befehl, über den man mehr wissen möchte.

Wenn wir zum Beispiel wissen möchten, wie die Funktion `limit` eingesetzt wird, geben wir folgendes ein:

```
sage: limit?                                                    # Not tested
```

Sage gibt uns anstelle einer Evaluation der Funktion die komplette Hilfe für die entsprechende Funktion zurück. Falls wir noch tiefer ins Innere von Sage schauen möchten und sogar den Sourcecode der Funktion anschauen möchten, hängen wir ein zweites Fragezeichen an:

```
sage: limit??                                                   # Not tested
```

gibt uns den Python-Code der Funktion `limit` zurück. Dies ist natürlich nur möglich, weil Sage ein quelloffenes System ist.

1.2 Allgemeine Funktionen

1.2.1 Grundoperationen

In Sage werden die Grundoperationen wie vielerorts üblich in der Infix-Notation ausgedrückt. Dies bedeutet, dass das Operationszeichen wie gewohnt zwischen den Operanden steht. Es stehen die folgenden Grundoperationen zur Verfügung:

Operation	Bedeutung
+	Addition
−	Subtraktion / negatives Vorzeichen
*	Multiplikation
/	Division (auch für Bruchterme)
^	Potenz

Es gelten die üblichen Regeln zur Operatorassoziativität: Höchste Priorität haben Potenzen, anschliessend werden Punktoperationen (\ und *) vor Strichoperationen (+ und −) ausgeführt:

```
sage: 12 - 2*3^2
-6
sage: 12 - (2*(3^2))
-6
sage: 3^2 -1
8
```

Unerwarteterweise führt Sage mehrere Potenzen hintereinander nicht von links nach rechts aus:

```
sage: 3^3^3
7625597484987
sage: 3^(3^3)
7625597484987
```

Wie üblich können Klammern `()` verwendet werden, um die Gruppierung der Operationen zu verändern:

```
sage: -3^2
-9
sage: (-3)^2
9
sage: (12 - 2)*3^(3-1)
90
sage: (3^3)^3
19683
```

Weiter gibt es auch die Möglichkeit, eine ganzzahlige Division auszuführen:

Operation	Bedeutung
`//`	Ganzzahliger Anteil einer Division
`%`	Divisionsrest

So ist zum Beispiel $\frac{17}{3} = 5 * 3 + 2$:

```
sage: 17//3
5
sage: 17%3
2
```

Bei negativen Zahlen verhält sich die Division nicht wunschgemäss, falls man damit den ganzzahligen Anteil und den Rest bestimmen möchte:

```
sage: -7//2
-4
sage: -7%2
1
```

1.2.2 Wurzeln

Sage kennt nur für die Quadratwurzel eine spezielle Funktion, nämlich `sqrt()`. Alle anderen Wurzeln müssen als Potenz mit rationalem Exponenten geschrieben werden. Quadratwurzeln werden wie folgt berechnet:

```
sage: sqrt(9)
3
sage: sqrt(2)
sqrt(2)
```

Wurzeln werden so weit wie möglich vereinfacht, aber nicht numerisch evaluiert, solange dies nicht explizit gefordert wird (mehr dazu weiter unten):

```
sage: sqrt(18)
3*sqrt(2)
```

Natürlich kennt Sage auch komplexe Zahlen und evaluiert die Wurzel aus -1 als eine imaginäre Einheit:

```
sage: sqrt(-1)
I
```

Für die n-te Wurzel gibt es in Sage keine spezielle Funktion, da wir die n-te Wuzel mit

$$\sqrt[n]{a} = a^{\frac{1}{n}}$$

ausdrücken können. So berechnen wir zum Beispiel $\sqrt[3]{64}$ mit:

```
sage: 64^(1/3)
4
```

1.2.3 Numerische Evaluation

Als Computer-Algebra-System rechnet Sage grundsätzlich immer symbolisch. Dies bedeutet, dass es zwar Terme vereinfacht, aber immer ein exaktes Ergebnis angibt. Hierfür gibt es zwei Ausnahmen. Zum einen können wir eine Kommazahl in der Rechnung benutzen. Dann wird der Teil, welcher die Kommazahl enthält, numerisch evaluiert. Zum anderen können wir eine numerische Evaluation explizit fordern.

Das folgende Beispiel zeigt, dass das Benutzen einer Kommazahl automatisch zu numerischem Rechnen führt. Dies selbst, wenn die Kommazahl eigentlich eine ganze Zahl repräsentiert:

```
sage: sqrt(2.0)
1.41421356237310
```

Dies funktioniert aber nur, wenn der Term selbst keine weitere Funktion oder symbolische Konstante enthält. Falls dies der Fall ist, wird nur ein Teil des Terms numerisch ausgewertet:

```
sage: sqrt(pi/4.0)
0.500000000000000*sqrt(pi)
sage: sqrt(2)/1.41
0.709219858156028*sqrt(2)
```

Falls wir einen gesamten Term numerisch evaluieren möchten, benutzen wir die Funktion `n()`. Wird sie ohne Argument aufgerufen, so führt sie die Evaluation mit einer vordefinierten Genauigkeit durch:

```
sage: sqrt(2).n()
1.41421356237310
```

Falls wir mehr oder weniger Stellen benötigen, können wir die Option `digits` benutzen. Sie gibt aber nicht die Anzahl Nachkommastellen sondern die Anzahl Stellen insgesamt an:

```
sage: sqrt(2).n(digits=50)
1.4142135623730950488016887242096980785696718753769
sage: sqrt(10025).n(digits=5)
100.12
```

1.2.4 Konstanten

Sage kennt unter anderem die folgenden Konstanten:

`pi`	π	Kreiszahl
`e`	e	Eulersche Zahl
`I`	i	Imaginäre Einheit

Diese können auch numerisch ausgewertet werden:

```
sage: pi.n()
3.14159265358979
sage: e.n()
2.71828182845905
```

1.2.5 Variablen

In Sage können sowohl Werte als auch Terme in einer Variable gespeichert werden. Als Variablenname ist alles zulässig, was auch in Python zulässig ist. Dies heisst insbesondere:

- Variablennamen müssen mit einem Buchstaben oder einem Unterstrich (_) beginnen.
- Variablennamen dürfen nur Buchstaben, Zahlen und Unterstriche enthalten.
- Variablennamen unterscheiden Gross- und Kleinschreibung.
- Es dürfen keine in Python reservierten Worte benutzt werden. [2]

Als Zuordnungsoperator wird das Zeichen = benutzt. Wir können so zum Beispiel das Ergebnis einer Rechnung als Variable speichern:

```
sage: ergebnis = sin(pi/4)*2
```

Im folgenden werden wir auch lernen, wie man Terme speichern kann. In einem späteren Abschnitt ist auch beschrieben, wie Funktionen gespeichert werden.

Wie jedes Computer-Algebra-System kann Sage symbolisch rechnen. Es muss aber jeweils festgelegt werden, welche Buchstaben oder Zeichenketten als Parameter benutzt werden sollen.

Dies geschieht mit dem Befehl `var`. Ihm wird als Argument eine Zeichenkette übergeben, welche die durch Kommas getrennten Variablennamen enthält. (Zeichenketten in Sage respektive Python werden mit `'` oder `"` gekennzeichnet.)

Das folgende Beispiel definiert x, y und z als Variablen:

```
sage: var('x,y,z')
(x, y, z)
```

1.3 Arithmetik und Algebra

1.3.1 Terme

Sage kann mit symbolischen Ausdrücken rechnen. Ein Ausdruck wird als symbolischer Ausdruck betrachtet, sobald eine Variable darin vorkommt. Wir benutzen in den folgenden Beispielen die Funktion `type` um herauszufinden, von welchem Typ ein Ausdruck ist. Ein einfacher Term mit einer Variable ist ein symbolischer Ausdruck:

[2] http://docs.python.org/2/reference/lexical_analysis.html#keywords

```
sage: var('x')
x
sage: type(x*2)
<type 'sage.symbolic.expression.Expression'>
```

Auch eine Funktion (siehe unten) ist ein symbolischer Ausdruck:

```
sage: f(x) = x + 1
sage: type(f)
<type 'sage.symbolic.expression.Expression'>
```

Benutzen wir jedoch eine Variable, um etwas zu speichern und nicht als symbolischen Parameter, so ist ein Ausdruck, welcher sie enthält, nicht mehr ein symbolischer Ausdruck:

```
sage: x = 3
sage: type(x*2)
<type 'sage.rings.integer.Integer'>
```

Terme können auch in Variablen gespeichert werden. Dazu benutzen wir wie bei Zahlenwerten den Operator = für die Zuordnung:

```
sage: var('x')
x
sage: term = 3*(x + 1)^2
sage: term
3*(x + 1)^2
```

Diese Terme können wir an einer bestimmten Stelle auswerten, oder, anders gesagt, für eine Variable einen Wert einsetzen. Dies geschieht wie folgt:

```
sage: var('x, a')
(x, a)
sage: term = 3*(x + 1)^a
sage: term
3*(x + 1)^a
sage: term(a = 2)
3*(x + 1)^2
sage: term(x = 2, a = 2)
27
```

Variablen können nicht nur durch Zahlen sondern auch durch symbolische Ausdrücke substituiert werden. Wenn wir im obigen Beispiel x mit a^2 substituieren möchten, geht dies wie erwartet:

```
sage: var('x, a')
(x, a)
sage: term = 3*(x + 1)^a
sage: term(x = a^2)
3*(a^2 + 1)^a
```

Vereinfachen

Beim symbolischen Rechnen mit Termen vereinfacht Sage diese nach dem Eingeben. So wird zum Beispiel der Term $x \cdot x + x^2$ vereinfacht:

```
sage: x*x + x^2
2*x^2
```

Komplexere Vereinfachungen werden nicht automatisch ausgeführt. So weiss Sage zwar, dass $\sin^2(x) + \cos^2(x) = 1$. Wenn wir den Term aber so eingeben, wird er nicht vereinfacht:

```
sage: sin(x)^2 + cos(x)^2
cos(x)^2 + sin(x)^2
```

Falls wir alle Terme so weit wie möglich vereinfachen möchten, erreichen wir dies mit der `simplify_full()` Funktion:

```
sage: (sin(x)^2 + cos(x)^2).simplify_full()
1
```

Dabei werden auch Additionstheoreme für trigonometrische Funktionen eingesetzt:

```
sage: var('x, y, z')
(x, y, z)
sage: sin(x + y).simplify_full()
cos(y)*sin(x) + cos(x)*sin(y)
sage: (sin(x)^2 + cos(x)^2).simplify_full()
1
```

Mit der verwandten Funktion `simplify_real()` werden auch Additionstheoreme bei Logarithmen angewandt, die nur mit reellen Werten erlaubt sind:

```
sage: x, y = var('x, y')
sage: (log(x) + log(y)).simplify_real()
log(x*y)
```

Faktorisieren und ausmultiplizieren

Mit dem Befehl `expand()` kann ein Term, der aus mehreren polynomialen Faktoren besteht, ausmultipliziert werden. Umgekehrt wird `factor()` verwendet, um einen Term so weit als möglich zu faktorisieren:

```
sage: ((x + 1)*(2*x - 4)).expand()
2*x^2 - 2*x - 4

sage: (2*x^2 - 2*x - 4).factor()
2*(x + 1)*(x - 2)
```

1.3.2 Gleichungen

In Sage können sowohl Gleichungen als auch Ungleichungen formuliert und ausgewertet werden. Dazu verwendet man dieselben boolschen Operatoren wie in Python:

Symbol	Bedeutung
==	Gleichheit
>=	Grösser oder gleich
<=	Kleiner oder gleich
>	Grösser als
<	Kleiner als
!=	Unleichheit

Symbolisch Lösen

Falls möglich löst Sage Gleichungen symbolisch. Falls dies nicht gelingen sollte, bleibt einem noch das numerische Lösen von Gleichungen, welches weiter unten beschrieben wird.

Für symbolisches Lösen von Gleichungen steht der Befehl `solve()` zur Verfügung. Dieser benötigt zwei Argumente: Zum einen die Gleichung, welche gelöst werden soll und zum anderen die Variable, nach welcher die Gleichung aufgelöst werden soll:

```
sage: solve(x^2 == 3, x)
[x == -sqrt(3), x == sqrt(3)]

sage: solve(x^3 - x^2 + x == 1, x)
[x == -I, x == I, x == 1]
```

Oft wird uns Sage auch komplexe Lösungen angeben. Um dies zu verhindern, können wir Sage mit dem Befehl `assume()` angeben, dass eine Variable nur mit einer reellen Zahl besetzt werden kann:

```
sage: var('x')
x
sage: assume(x, 'real')
sage: solve(x^3 - x^2 + x == 1, x)
[x == 1]
```

Achtung, solche Annahmen sollten anschliessend mit `forget()` wieder rückgängig gemacht werden. Ansonsten gelten sie für folgende Rechnungen noch immer.

Lösungen weiterverwenden

Es stellt sich das Problem, dass wir nun zwei Gleichungen als Ergebnis erhalten und die Lösung nicht direkt verwenden können, um weiterzurechnen.

Wir erhalten von `solve()` immer eine Liste von Lösungen. Diese sind intern ab 0 nummeriert. Die erste Lösung ist also die Lösung 0, die zweite die Lösung 1 usw. Wir können nun einzelne Lösungen aus der Liste mit ihrer Nummer herausholen:

```
sage: solve(x^2 == 3, x)
[x == -sqrt(3), x == sqrt(3)]
sage: solve(x^2 == 3, x)[0]
x == -sqrt(3)
sage: solve(x^2 == 3, x)[1]
x == sqrt(3)
```

Die Lösungen werden uns aber wiederum als Gleichung angegeben und nicht als numerischen Wert. Daher können wir das so erhaltene Ergebnis nicht in einer Variable abspeichern und weiterverwenden. Um wirklich die Lösung für x aus dem Term zu extrahieren, können wir die Funktion `rhs()` benutzen, was für "Right Hand Side" steht und genau das tut, was der Name sagt: Es gibt uns die rechte Seite der Gleichung zurück. Dies benutzen wir wie folgt, um den Wert für x aus dem Ergebnis zu extrahieren:

```
sage: (x == -sqrt(3)).rhs()
-sqrt(3)

sage: solve(x^2 == 3, x)[0].rhs()
-sqrt(3)
```

Nun können wir den Wert in einer Variablen speichern oder numerisch auswerten. (Oder beides, wie uns das folgende Beispiel vorführt):

```
sage: erg = solve(x^2 == 3, x)[0].rhs()
sage: erg
-sqrt(3)
sage: erg.n()
-1.73205080756888
```

Numerisch Lösen

Sage kann gewisse Gleichungen nicht symbolisch lösen und gibt uns daher eine leere Lösungsliste zurück, auch wenn die Gleichung eigentlich eine Lösung besitzt. So findet Sage zum Beispiel für die Gleichung

$$\frac{2}{\sqrt{x}} - \frac{1}{x^2} = 0$$

keine sinnvolle, explizite Lösung:

```
sage: solve(2/x^(1/2) - 1/(x^2) == 0, x)
[x == -sqrt(1/2)*x^(1/4), x == sqrt(1/2)*x^(1/4)]
```

Mit der Option `explicit_solutions` können wir Sage zwingen, nur explizite Lösungen anzugeben. Im oben aufgezeigten Fall erhalten wir dann eine leere Liste von Lösungen:

```
sage: solve(2/x^(1/2) - 1/(x^2) == 0, x, explicit_solutions=True)
[]
```

Mit Hilfe von `find_root()` können wir Nullstellen numerisch berechnen. Dafür müssen wir schon eine Ahnung haben, in welchem Bereich wir nach der Nullstelle suchen. Wissen wir zum Beispiel, dass eine Funktion $f(x)$ eine Nullstelle auf dem Intervall $[a, b]$ hat, finden wir eine numerische Approximation dieser Nullstelle mit `find_root(f == 0, a, b)`. Nun wollen wir also die Lösung der obigen Gleichung finden:

```
sage: f(x) = 2/x^(1/2) - 1/(x^2)
sage: find_root(f, 0.5, 5)
0.6299605249475858
```

1.3.3 Funktionen

Wie oben gesehen, sind Funktionen in Sage auch symbolische Ausdrücke. Sie unterscheiden sich daher auch von Funktionen in Python oder einer anderen Programmiersprache. Es wird die Schreibweise

$$f(x) = x^3 + x$$

für Funktionen benutzt. Dies ist ein Speichervorgang, welcher in Sage keinen Rückgabewert hat:

```
sage: f(x) = x^3 + x
```

Nun sind sowohl f als auch $f(x)$ symbolische Ausdrücke mit zwei verschiedenen Bedeutungen, wie das folgende Beispiel deutlich macht:

```
sage: f(x) = x^3 + x
sage: f
x |--> x^3 + x
sage: f(x)
x^3 + x
```

Stückweise definierte Funktionen

Mit der Funktion `Piecewise()` können wir in Sage auch mit stückweise definierten Funktionen arbeiten. Die Syntax für den Befehl ist jedoch etwas umständlich. Wollen wir zum Beispiel die Funktion

$$f(x) = \begin{cases} -x^2 & \text{für } x \leq 0 \\ x^2 & \text{für } x > 0 \end{cases}$$

definieren, so müssen wir der Funktion eine Liste von Listen übergeben, wo jede einzelne Liste aus einem Tupel besteht, welches das Interval des Definitionsbereichs angibt, also z.B. `(-oo, 0)` für das Interval $[-\infty, 0]$ und einer für das Interval geltende Funktionsgleichung. Als letztes Argument muss angegeben werden, welche Variable durch die Funktion gebunden werden soll:

```
sage: f = Piecewise([[(-oo,0), -x^2],[(0,oo), x^2]], x)
sage: f(3)
9
sage: f(-3)
-9
```

Die Zeichen `-oo` und `oo` werden in Sage für $-\infty$, respektive ∞ benutzt. Mehr dazu wird im Abschnitt über das Berechnen von Grenzwerten erklärt.

1.3.4 Partialbruchzerlegung

Vor allem in der Analysis wird im Gymnasium teilweise die Partialbruchzerlegung benutzt, um die Stammfunktion einer rationalen Funktion zu finden. Diese Zerlegung kann auch mit Sage gemacht werden. Wenn wir die Funktion

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

betrachten, kann diese als Summe von zwei Brüchen geschrieben werden:

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

Diese Zerlegung findet `partial_fraction()` in Sage für uns:

```
sage: f(x) = 1/(x^2 -1)
sage: f.partial_fraction()
x |--> -1/2/(x + 1) + 1/2/(x - 1)
```

1.3.5 Funktions-Graphen darstellen

Im Folgenden werden die Grundlagen für das Darstellen von Funktionsgraphen aufgezeigt. Der `plot()` Befehl kann auch für das Darstellen von Vektoren oder Daten aus dem Fachbereich Stochastik benutzt werden. Dies wird aber im entsprechenden Abschnitt beschrieben.

Wir können den Graphen der Funktion $f(x) = x^2$ mit dem folgenden Befehl darstellen:

```
sage: f(x) = x^2
sage: plot(f)
Graphics object consisting of 1 graphics primitive
```

Sage versucht einen vernünftigen Bereich von x-Werten zu finden, um den Funktionsgraphen darzustellen. Falls dies nicht dem gewünschten Bereich entspricht, können wir diesen mit den Optionen `xmin` und `xmax` für die x-Achse, respektive `ymin` und `ymax` für die y-Achse den zu darstellenden Bereich festlegen:

```
sage: f(x) = x^2
sage: plot(f, xmin=-12, xmax=12, ymin=-10, ymax=150)
Graphics object consisting of 1 graphics primitive
```

Wollen wir mehrere Funktionsgraphen im selben Koordinatensystem darstellen, können wir die beiden Plots einzeln erstellen und in Variabeln abspeichern. Dies verhindert, dass sie einzeln angezeigt werden. Anschliessend verwenden wir den Plot-Befehl erneut um beide zusammen anzuzeigen. Die Plots werden mit einem +-Zeichen zusammengefügt. Mit der Option `color` können wir die Farbe der einzelnen Graphen festlegen:

```
sage: graph1 = plot(x^2 + 1, color="green", xmin = 0, xmax = 3)
sage: graph2 = plot(e^x, color="red", xmin = 0, xmax = 3)
sage: plot(graph1 + graph2, )
Graphics object consisting of 2 graphics primitives
```

Optionen, welche für beide Plots gültig sind (z.B. `xmin` oder `xmax`) müssen auch bei beiden Plots angegeben werden, da sonst Sage sonst beim Graph, wo es nicht angegeben wird wie üblich versucht, vernünftige Standartwerte auszuwählen.

Polstellen darstellen

Wollen wir eine Funktion mit Polstellen darstellen, zum Beispiel die Funktion

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

wird Sage uns keinen vernünfigen y-Wertebereich auswählen. Da die Funktionswerte an der Polstelle gegen Unendlich streben, nimmt Sage an, dass wir an sehr grossen oder sehr kleinen y-Werten interessiert sind und wählt die Auflösung der y-Achse entsprechend. Falls es sich um eine ungerade Polstelle handelt, bei der die y-Werte vom Negativen ins Positive wechseln oder umgekehrt, verbindet Sage den positiven und den negativen Teil des Graphen mit einer unerwünschten, senkrechten Linie an der Polstelle.

Wie wir oben gelernt haben, können wir den Wertebereich einfach einschränken:

```
sage: f(x)=(x^2 +1)/(x^2-1)
sage: plot(f, xmin=-2, xmax=2, ymin=-10, ymax = 10)
Graphics object consisting of 1 graphics primitive
```

Nun haben wir nur noch das Problem, dass der Graph zwei unerwünschte senkrechte Linien an den Polstellen hat. Dies kann mit der Option `detect_poles` verhindert werden. Falls wir die Option auf `True` stellen, werden die Linien nicht mehr dargestellt:

```
sage: f(x)=(x^2 +1)/(x^2-1)
sage: plot(f, xmin=-2, xmax=2, ymin=-10, ymax = 10, detect_poles=True)
Graphics object consisting of 4 graphics primitives
```

Möchten wir hingegen die vertikalen Asymptoten trotzdem darstellen, aber nicht in derselben Farbe wie den Funktionsgraphen, können wir die Option `detect_poles` auf `"show"` stellen:

```
sage: f(x)=(x^2 +1)/(x^2-1)
sage: plot(f, xmin=-2, xmax=2, ymin=-10, ymax = 10, detect_poles="show")
Graphics object consisting of 6 graphics primitives
```

1.4 Logarithmen

Im Folgenden wird beschrieben, wie in Sage Logarithmen benutzt werden können. Grundsätzlich ist es so, dass in Sage der 10er Logarithmus keine spezielle Bedeutung hat. Fall bei einer Logarithmusfunktion die Basis nicht angegeben wird, geht Sage immer vom natürlichen Logarithmus aus.

Wenn wir also zum Beispiel $\log(10)$ berechnen, wird Sage nichts tun, da sich dieser Term nicht auf eine offensichtliche Art vereinfachen lässt. Berechnen wir jedoch $\log(e^5)$, wird uns Sage die für den natürlichen Logarithmus zu erwartende Antwort geben, nämlich 5:

```
sage: log(10)
log(10)

sage: log(e^5)
5
```

Möchten wir die Logarithmusfunktion zu einer anderen Basis berechnen, müssen wir der `log()` Funktion als zweites Argument die Basis übergeben. Wollen wir zum Beispiel $\log_{10}(10^5)$ berechnen, geben wir dies wie folgt ein:

```
sage: log(10^5, 10)
5

sage: log(8,2)
3
```

Man kann auch die Logarithmengesetze benutzen, um Terme zu zerlegen. So können wir zum Beispiel Sage die Zerlegung

$$\log(10^5) = 5\log(2) + 5\log(5)$$

machen lassen. In diesem Fall benutzen wir nicht `simplify_full()`, sondern die ähnliche Funktion `canonicalize_radical`:

```
sage: log(10^5).canonicalize_radical()
5*log(5) + 5*log(2)
```

Diese Gesetze können auch umgekehrt verwendet werden, wie in diesem Beispiel:

```
sage: (5*log(2) + 5*log(5)).simplify_log()
log(100000)
```

Es geben weitere mögliche Vereinfachungen, die wir hier nicht weiter erwähnen.

1.5 Trigonometrie

1.5.1 Winkelmass

Im folgenden Abschnitt möchten wir uns einen Überblick über die Trigonometrischen Funktionen in Sage verschaffen. Es macht Sinn, zuerst zu betrachten, wie Sage mit Winkelmassen umgeht.

Grundsätzlich rechnet Sage immer im Bogenmass. Falls man ein Ergebnis im Gradmass erhalten oder ein Argument im Gradmass in eine Funktion eingeben möchte, muss man dies immer von Hand konvertieren. Es gibt im Moment keine Möglichkeit, Sage so einzustellen, dass es allgemein im Gradmass rechnet.

Im Moment bietet Sage hierzu nur das sehr rudimentäre `units` Paket, mit welchem man Werte mit einer Einheit versehen und anschliessend auch umrechnen kann. Dies ist sehr umständlich und vermutlich ist es einfacher, die

Umrechnung von Hand vorzunehmen, in dem man mit $\frac{\pi}{180^\circ}$ respektive $\frac{180^\circ}{\pi}$ multipliziert um vom Gradmass ins Bogenmass respektive vom Bogenmass ins Gradmass zu konvertieren:

```
sage: 45*(pi/180)
1/4*pi

sage: pi/3*(180/pi)
60
```

Falls man dies viel benutzt, lohnt es sich natürlich, diese beiden Konvertierungen als Funktion abzuspeichern. Dies kann wie erwartet mit den in früheren Kapiteln beschriebenen Funktionen geschehen:

```
sage: deg2rad(x) = x*(pi/180)
sage: rad2deg(x) = x*(180/pi)
sage: deg2rad(45)
1/4*pi
sage: rad2deg(pi/3)
60
```

In Zukunft wird diese Funktion vermutlich vom `units` Paket übernommen. Im Moment wird hier auf eine Dokumentation dieses Pakets verzichtet, da es sich sehr schlecht mit den unten beschriebenen trigonometrischen Funktionen verträgt. Weitere Informationen finden sich in der Sage Dokumentation. [3]

1.5.2 Trigonometrische Funktionen

Wie oben beschrieben rechnen die trigonometrischen Funktionen in Sage nur im Bogenmass. Die Namen der in der Schule gebräuchlichen trigonometrischen Funktionen sind wie erwartet `sin()`, `cos()`, `tan()` und `cot()`. Diese können direkt benutzt werden, falls wir im Bogenmass rechnen möchten. Ansonsten müssen wir wie oben beschrieben vom Gradmass in Bogenmass konvertieren:

```
sage: cos(pi/6)
1/2*sqrt(3)

sage: deg2rad(x)= x*(pi/180)
sage: sin(deg2rad(45))
1/2*sqrt(2)
sage: tan(deg2rad(-60))
-sqrt(3)
```

Ihre Umkehrfunktionen sind auch mit den nicht sehr überraschenden Namen `asin()`, `acos()`, `atan()` und `acot()` versehen. Sie geben uns aber wie oben erklärt nur Winkel im Bogenmass zurück. Möchten wir im Gradmass rechnen, müssen wir wieder konvertieren. Die exakte Berechnung der Werte funktioniert in die Gegenrichtung nur, falls im ursprünglichen Wert keine Wurzeln vorkommen:

```
sage: atan(1)
1/4*pi
sage: acos(1/2)
1/3*pi
sage: rad2deg(x) = x*(180/pi)
sage: rad2deg(acos(-1/2))
120
```

Falls wir Wurzelterme verwenden, müssen wir mit der Funktion `simplify_full()` vereinfachen:

[3] http://www.sagemath.org/doc/reference/calculus/sage/symbolic/units.html

```
sage: acos(sqrt(3)/2)
arccos(1/2*sqrt(3))
sage: (acos(sqrt(3)/2)).simplify_full()
1/6*pi
```

Sage kann auch weitere Regeln für trigonometrische Funktionen anwenden, um Terme zu vereinfachen. Es kennt zum Beispiel auch die Additionstheoreme:

```
sage: var('x, y')
(x, y)
sage: (sin(x+y)).simplify_full()
cos(y)*sin(x) + cos(x)*sin(y)
sage: (sin(x)^2 + cos(x)^2).simplify_full()
1
```

1.6 Vektorgeometrie

Ein Vektor in Sage kann mit dem Befehl `vector()` erstellt werden. Ihm wird eine Liste der Komponenten als Argument übergeben:

```
sage: vector([3,2,-1])
(3, 2, -1)
```

1.6.1 Grundoperationen

Sage kennt die üblichen Vektoroperationen. Es werden die üblichen Zeichen + für die Addition sowie $\star$ für die Multiplikation mit einem Skalar verwendet:

```
sage: a = vector([3,2,-1])
sage: b = vector([1,-2,2])
sage: a + b
(4, 0, 1)
sage: 4 * b
(4, -8, 8)
sage: -b + 2*(a + b)
(7, 2, 0)
```

Innerhalb eines Vektors können auch Parameter verwendet werden. Natürlich müssen auch diese zuerst mit `var()` als solche deklariert werden:

```
sage: var('vx, vy, vz')
(vx, vy, vz)
sage: v = vector([vx, vy, vz])
sage: 2*v
(2*vx, 2*vy, 2*vz)
```

1.6.2 Länge eines Vektors

Die (euklidische) Länge eines Vektors wird mit dem Befehl `norm()` berechnet:

```
sage: norm(vector([9,6,2]))
11
```

Falls die Länge nicht ganzzahlig ist, wird natürlich auch ein vereinfachter Term und kein numerischer Wert zurückgegeben:

```
sage: norm(vector([2,6,2]))
2*sqrt(11)
```

1.6.3 Skalar- und Vektorprodukt

Das *Skalarprodukt* wird mit der Funktion `dot_product()` berechnet. Diese ist eine Methode der Vektor-Klasse. Sie wird also als Methode eines Vektors ausgeführt, welcher der zweite Vektor übergeben wird:

```
sage: v = vector([2,3,3])
sage: w = vector([2,2,1])
sage: v.dot_product(w)
13
```

Da das Skalarprodukt kommutativ ist, spielt es natürlich keine Rolle, von welchem Vektor aus wir die Methode aufrufen:

```
sage: v = vector([2,3,3])
sage: w = vector([2,2,1])
sage: w.dot_product(v)
13
```

Das Skalarprodukt kann auch als Multiplikation zweier Vektoren mit $\star$ geschrieben werden:

```
sage: v = vector([2,3,3])
sage: w = vector([2,2,1])
sage: v*w
13
```

Es besteht hier aber Verwechslungsgefahr mit der Skalarmultiplikation. Es ist Vorsicht geboten.

Das *Vektorprodukt* kann auf die gleiche Weise mit `cross_product()` berechnet werden. Hier ist es natürlich relevant, von welchem Vektor aus die Methode ausgeführt wird, da das Vektorprodukt antikommutativ ist.

Wollen wir für zwei Vektoren $\vec{v}$ und $\vec{w}$ das Produkt $\vec{v} \times \vec{w}$ berechnen, wird die Methode von $\vec{v}$ aus ausgeführt und vice versa:

```
sage: v = vector([2,3,3])
sage: w = vector([2,2,1])
sage: v.cross_product(w)
(-3, 4, -2)
sage: w.cross_product(v)
(3, -4, 2)
```

1.6.4 Vektoren grafisch darstellen

Sage kann Vektoren nicht direkt graphisch darstellen. Das 2D Graphikmodul [4] kann jedoch Pfeile darstellen, welche sehr gut geeignet sind, um zweidimensionale Vektorrechnungen zu veranschaulichen.

Die Funktion `arrow()` erstellt einen Pfeil. Da wir keine Vektoren sondern Pfeile darstellen, müssen wir immer einen Anfangspunkt und einen Endpunkt angeben. Mit der Option `color` können wir die Farbe des Pfeils festlegen. Die erstellten Pfeile können anschliessend mit dem uns schon bekannten Befehl `plot()` dargestellt werden.

[4] http://www.sagemath.org/doc/reference/plotting/index.html

Die Addition von Vektoren könnte also zum Beispiel wie folgt veranschaulicht werden:

```
sage: v1 = arrow((0,0), (3,4))
sage: v2 = arrow((3,4), (6,1))
sage: sum_v1_v2 = arrow((0,0), (6,1), color='red')
sage: v1 + v2 + sum_v1_v2
Graphics object consisting of 3 graphics primitives
```

Falls die Vektorpfeile zu dick oder zu dünn sind, kann mit der `width` Option die Strichbreite angepasst werden. Der Plot-Befehl besitzt eine `gridlines` option, welche wir auf `true` setzen können, falls Gitternetzlinien in der Grafik erwünscht sind:

```
sage: v1 = arrow((0,0), (3,4), width=5)
sage: v2 = arrow((3,4), (6,1), width=5)
sage: sum_v1_v2 = arrow((0,0), (6,1), color='red', width=6)
sage: G = v1 + v2 + sum_v1_v2
sage: G.show(gridlines=true)
```

1.7 Analysis

1.7.1 Folgen und Reihen

In der gymnasialen Mathematik werden oft Folgen betrachtet. Diese sind grundsätzlich Funktionen von der Menge der natürlichen auf die reellen Zahlen. Sage hat einige Funktionen, um mit Zahlenmengen zu arbeiten. Diese sind aber für unsere Zwecke ungeeignet.

Es ist einfacher, Folgen als Funktion zu definieren und um zu verdeutlichen, dass wir diese als Folge benutzen möchten, die Variable mit n zu benennen. So speichern wir die Folge

$$a_n = \frac{1}{n^2}$$

in Sage als:

```
sage: a(n) = 1/n^2
```

Um nun Elemente der dazugehörigen Reihe (d.h. die Folge der Teilsummen) zu berechnen, können wir den Befehl `sum()` benutzen. Die Summe

$$s_6 = \sum_{k=1}^{6} a_k = \sum_{k=1}^{6} \frac{1}{k^2}$$

wird in Sage mit:

```
sage: var('k')
k
sage: a(n) = 1/n^2
sage: sum(a(k), k, 1, 6)
5369/3600
```

berechnet. Allgemein wird die Summe

$$\sum_{k=a}^{b} \text{Ausdruck}$$

mit `sum(Ausdruck, k, a, b)` berechnet. Wir können also die Reihe als Folge von Teilsummen in Sage wieder als Funktion speichern:

```
sage: var('k')
k
sage: a(n) = n^2
sage: s(n) = sum(a(k), k, 1, n)
sage: s
n |--> 1/3*n^3 + 1/2*n^2 + 1/6*n
sage: s(6)
91
```

Folgen und Reihen graphisch darstellen

Sage kennt die Funktion `scatter_plot()` welcher eine Liste von Tupeln der Form `(x, y)` übergeben wird. [5] Diese werden anschliessen als Punkte dargestellt. Leider können wir dem Befehl aber keine Funktion der Form $a_n = \frac{1}{n^2}$ übergeben.

Wir müssen also eine Liste von Punkten generieren, welche wir gerne darstellen möchten. Dazu können wir sogenannte Python List Comprehensions [6] benutzen, um die Liste zu generieren:

```
sage: a(n) = 1/n^2
sage: punkte = [(n, a(n)) for n in range(1,10)]
sage: scatter_plot(punkte)
Graphics object consisting of 1 graphics primitive
```

Mit den Funktion `range()` geben wir an, welchen Bereich wir gerne darstellen möchten. Dabei wird immer die letzte Zahl ausgeschlossen. Im obigen Beispiel werden also Punkte für $n \in \{1, \ldots, 9\}$ dargestellt.

Wie im Abschnitt über das Darstellen von Funktionsgraphen können wir auch diese Plots kombinieren und so zum Beispiel auch den Plot der Funktion `a(n)` hinter die Punkte legen, um die Tendenz der Folge darzustellen:

```
sage: a(n) = 1/n^2
sage: points = [(n, a(n)) for n in range(1,6)]
sage: plot1 = scatter_plot(points)
sage: plot2 = plot(a(x), xmin=1, xmax=5.4)
sage: plot(plot1 + plot2)
Graphics object consisting of 2 graphics primitives
```

1.7.2 Grenzwerte

Sage kann sowohl Grenzwerte berechnen, bei welchen eine Variable gegen ∞ oder $-\infty$ strebt, sowie Grenzwerte an einer bestimmten Stelle.

So können wir zum Beispiel den Grenzwert

$$\lim_{x \to \infty} \frac{1}{x}$$

wie folgt berechnen:

```
sage: var('x')
x
sage: ((2*x+1)/x).limit(x=oo)
```

[5] http://www.sagemath.org/doc/reference/plotting/sage/plot/scatter_plot.html
[6] http://docs.python.org/2/tutorial/datastructures.html#list-comprehensions

```
2
sage: (1/x).limit(x=-oo)
0
```

Wie schon bei Stückweise definierten Funktion erklärt wird für ∞ zweimal der Kleinbuchstaben "o" also `oo` und für $-\infty$ entsprechend `-oo` benutzt.

Grenzwerte an einer bestimmten Stelle werden genauso berechnet. Der Grenzwert

$$\lim_{x \to -2} \frac{(x+2)(x-3)}{(x+2)}$$

kann wie folgt berechnet werden:

```
sage: var('x')
x
sage: term = (x + 2)*(x-3)/(x + 2)
sage: term.limit(x=-2)
-5
```

In beiden Beispielen darf nicht vergessen werden, die Variable zuerst als solche zu definieren. Falls wir den links- und den rechtsseitigen Grenzwert einzeln berechnen wollen, können wir die Option `dir` verwenden:

```
sage: var('x')
x
sage: term = (x^2 + 1)/(x-1)
sage: term.limit(x=1)
Infinity
sage: term.limit(x=1, dir='+')
+Infinity
sage: term.limit(x=1, dir='-')
-Infinity
```

Wir sehen, dass das Ergebnis `Infinity` unbestimmt ist. Es ist nicht klar, ob $+\infty$ oder $-\infty$ gemeint ist. Erst wenn wir die links- und rechtsseitigen Grenzwerte analysieren wird klar, in welchem Fall der Wert gegen $+\infty$ respektive $-\infty$ strebt.

Möchte man den Grenzwert einer Summe berechnen, kann man dies direkt mit der Summenfunktion berechnen. Möchten wir zum Beispiel den Grenzwert

$$\sum_{k=1}^{\infty} \frac{1}{k^2}$$

berechnen, tun wir dies in Sage wie folgt:

```
sage: var('k')
k
sage: sum(1/k^2, k, 1, oo)
1/6*pi^2
```

1.7.3 Differenzial- und Integralrechnung

Die Ableitung und das unbestimmte Integral (d.h. die Stammfunktion) einer Funktion $f(x)$ wird in Sage mit den Funktionen `diff()` und `integral()` berechnet. Wir können sowohl Funktionen sowie symbolische Ausdrücke integrieren und differenzieren. Funktionen können ohne Angabe einer Variable differenziert werden, es wird nach der durch die Funktionsdefinition gebundenen Variable differenziert. Da Sage Funktionen in mehreren Unbekannten zulässt, müssen wir bei der Integration die Integrationsvariable immer angeben.

Bei einem symbolischen Ausdruck muss die Variable, nach welcher differenziert oder integriert wird, auf jeden Fall angegeben werden, da mehrere freie Variablen vorkommen können und keine davon durch die Funktionsschreibweise gebunden ist.

Ableiten und integrieren einer Funktion in Sage:

```
sage: f(x) = x^2
sage: f.diff()
x |--> 2*x
sage: f.integral(x)
x |--> 1/3*x^3
```

Wie wir sehen, erhalten wir beim Ableiten und Integrieren einer Funktion die Ableitungs- bzw. eine Stammfunktion wiederum als Sage-Funktionen zurück. Wie in Computer-Algebra-Systemen üblich wird die Stammfunktion ohne eine addierte Konstante zurückgegeben.

Im Folgenden leiten wir keine Funktionen, sondern symbolische Ausdrücke ab. Das heisst, dass wir den Funktionen `integrate()` und `diff()` als erstes Argument die Variable übergeben, nach der wir integrieren respektive ableiten wollen:

```
sage: var('x, t')
(x, t)
sage: sin(x).diff(x)
cos(x)
sage: sin(x).diff(t)
0
sage: sin(x).integral(x)
-cos(x)
sage: (e^(2*t^2+1)).diff(t)
4*t*e^(2*t^2 + 1)
```

Bestimmtes Integral

Die Funktion `integral()` berechnet in Sage auch bestimmte Integrale. Dazu werden die Integrationsgrenzen als weitere Argumente übergeben. Ein Integral der Form:

$$\int_a^b f(x)\mathrm{d}x$$

wird in Sage mit `f.integral(x, a, b)` berechnet. Auch hier gilt die Regel, dass die Integrationsvariable nur angegeben werden muss, falls es sich bei `f` um einen symbolischen Ausdruck handelt:

```
sage: var('t')
t
sage: f(x) = sqrt(4 - x^2)
sage: f.integral(x, -2, 2)
2*pi
sage: sin(t).integral(t, 0, pi)
2
```

1.8 Stochastik

Neben den Grundoperationen ist die in der Stochastik am häufigsten benutzte Funktion die Fakultät. Für eine Zahl $n \in \mathbb{N}$ ist dies definiert als

$$n! = n(n-1)(n-2)\cdots 2 \cdot 1$$

Sage berechnet dies mit der Funktion `factorial()`. Im Folgenden wird die Fakultät von 5 berechnet:

```
sage: factorial(5)
120
```

Sage kann mit Fakultäten auch symbolisch rechnen. Die Terme müssen aber mit der `simplify_full()` Funktion vereinfacht werden:

```
sage: var('n')
n
sage: (factorial(n)/factorial(n-2)).simplify_full()
n^2 - n
```

1.8.1 Binomialkoeffizienten

Um die Anzahl Möglichkeiten zu berechnen, mit denen wir eine Teilmenge mit r Elementen aus einer n elementigen Menge wählen können, benötigen wir den Binomialkoeffizienten. Dieser ist definiert als

$$\binom{n}{r} = \frac{n!}{(n-r)!n!}$$

Diese Berechnung wird mit der Funktion `binomial()` ausgewertet, wobei diese die beiden Argumente n und r übernimmt:

```
sage: binomial(10, 7)
120
```

Auch Terme, welche die `binomial()` Funktion benutzen können mit `simplify_full()` vereinfacht werden:

```
sage: var('n, r')
(n, r)
sage: (binomial(n,r)*factorial(r)).simplify_full()
factorial(n)/factorial(n - r)
```

1.9 Verschiedene nützliche Funktionen

Im folgenden Abschnitt sind einige nützliche Funktionen aufgeführt, welche bisher in keinem Kapitel dieser Anleitung sinnvoll untergebracht werden konnten.

Die Funktionen `gcd()` und `lcm()` können benutzt werden, um den **grössten gemeinsamen Teiler** respektive das **kleinste gemeinsame Vielfache** zweier Zahlen zu finden:

```
sage: gcd(124, 56)
4
sage: lcm(21, 15)
105
```

Für das Aufstellen von Aufgaben ist es oft nützlich, **Primzahlen** zu finden. Wir können uns mit dem Befehl `prime_range()` einen ganzen Bereich von Primzahlen zwischen zwei Grenzen zurückgeben lassen:

```
sage: prime_range(1050, 1100)
[1051, 1061, 1063, 1069, 1087, 1091, 1093, 1097]
```

Wissen wir genau, oberhalb oder unterhalb welcher Zahl wir eine Primzahl benötigen, können wir mit `next_prime()` die nächst grössere Primzahl sowie mit `previous_prime()` die nächst kleinere Primzahl bestimmen:

```
sage: next_prime(1050)
1051
sage: previous_prime(1100)
1097
```

Mit `factor()` können wir eine Zahl in ihre **Primfaktoren** zerlegen. Sind wir an allen Teilern einer Zahl interessiert, benutzen wir `divisors()`:

```
sage: factor(156)
2^2 * 3 * 13
sage: divisors(156)
[1, 2, 3, 4, 6, 12, 13, 26, 39, 52, 78, 156]
```

1.10 Weiterführende Links und Literatur

Das folgende Tutorial erklärt (auf englisch) wie Sage als einfacher Rechner benutzt werden kann. Hier finden sich auch viele Funktionen und Beispiele, welche für unsere Zwecke interessant sind.

- http://www-rohan.sdsu.edu/~mosulliv/sagetutorial/sagecalc.html

Die offizielle deutsche Dokumentation von Sage ist noch im Aufbau und weit entfernt von einer vollständigen Dokumentation. Das Einführungstutorial ist jedoch auch auf deutsch verfügbar. Die offizielle Seite der deutschen Version von Sage findet sich hier:

- http://www.sagemath.org/de/