Sage2

Inhaltsverzeichnis

Rechnen mit formalen Ausdrücken

Um mit formalen Ausdrücken zu rechnen, muß man entsprechende Variablennamen deklarieren; nur die Variable `x` ist von vornherein als solche deklariert.

```
x
```

```
x
```

```
parent(x)
```

```
Symbolic Ring
```

```
x^2+4
```

```
x^2 + 4
```

Formale Ausdrücke können natürlich auch Variablen zugeordnet werden.

```
a=x^3
a
```

```
x^3
```

Man mache sich den prinzipiellen Unterschied zwischen einer "Variable" im Sinn der Informatik (hier z.B. die Variable `a`) und der symbolischen "Variable" `x` klar: `a` bezeichnet einen Platz im Speicher des Computers, in dem ein gewisser Wert abgelegt ist, und dieser Wert hat einen Typ. Dagegen ist `x` hier eine Variable im Sinne der Analysis, d.h., ein Platzhalter, für den verschiedene Werte eingesetzt werden *können*.

Nicht die Variable a hat einen festen Typ, sondern ihr Wert; mit anderen Worten, die Typen sind *dynamisch*.

```
parent(a)
```

```
Symbolic Ring
```

```
a=1
a
```

```
1
```

```
parent(a)
```
```
Integer Ring
```

alle formalen Symbole außer x müssen als solche deklariert werden.

```
y
```
```
Traceback (click to the left of this block for traceback)
...
NameError: name 'y' is not defined
```
```
var('y')
y
```
```
y
```
```
parent(y)
```
```
Symbolic Ring
```

das kann auch gruppenweise passieren.

```
var('u,v')
```
```
(u, v)
```
```
parent(u)
```
```
Symbolic Ring
```

Formale Asdrücke werden vorerst nicht ausmultipliziert

```
f=(x+y)^2
f
```
```
(x + y)^2
```

das muß explizit angefordert werden.

```
g=expand(f)
g
```
```
x^2 + 2*x*y + y^2
```

alternativ kann die Methode `expand` direkt angefordert werden.

```
f.expand()
```
```
x^2 + 2*x*y + y^2
```

dabei wird der ursprüngliche Ausdruck nicht verändert

```
f
```
```
(x + y)^2
```

Das ist manchmal nützlich, um die Darstellung kompakt zu halten.

```
(1+x)^1000
```

```
(x + 1)^1000
```

Die inverse Operation zum ausmultiplizieren ist **factor**. Zum Vergleich: Wenn man **factor** auf eine ganze Zahl anwendet, wird trotz gleichen Namens der Funktion eine ganz andere Methode aufgerufen:

```
factor(g)
```

```
(x + y)^2
```

```
factor(442)
```

```
2 * 13 * 17
```

Mit formalen Ausdrücken können alle herkömmlichen symbolischen Operationen durchgeführt werden.

```
diff(sin(x),x)
```

```
cos(x)
```

```
integrate(sin(x),x)
```

```
-cos(x)
```

```
integrate(sin(x),x,0,pi)
```

```
2
```

Allerdings ist Vorsicht geboten: gewisse Fragen sind formal *nicht entscheidbar*, d.h., es gibt keinen Algorithmus, der alle Probleme lösen kann. Um trotzdem weiterrechnen zu können, werden dann sog.heuristische Methoden verwendet, wobei aber inkorrekte Ergebnisse nicht ausgeschlossen werden können.

Variablen können mit **reset** zurückgesetzt werden.

```
reset('y')
y
```

```
Traceback (click to the left of this block for traceback)
...
NameError: name 'y' is not defined
```

Mathematische Funktionen

Besonders wichtig sind symbolische Variablen für die Definition von mathematischen Funktionen. Mathematische Funktionen sind nicht zu verwechseln mit den normalen Prozedur-artigen "Funktionen" in C oder Python (mit **def**): Beispielsweise kann man mathematische Funktionen im Gegensatz zu C-Funktionen zueinander addieren, integrieren, ableiten, Grenzwerte berechnen, ...
Die Definition von Funktionen ist sehr intuitiv und folgt der üblichen mathematischen Notation dafür:

```
f(x) = x^2 + x
g(x) = x^3
```

```
print f
show(f)
print parent(f)
```

```
x |--> x^2 + x
```

$$x \mapsto x^2 + x$$

```
Callable function ring with argument x
```

Die Variablen einer Funktion müssen nicht unbedingt im Vorhinein mit `var("...")` deklariert werden, das passiert automatisch:

```
F(n) = n*(n-1)
show(F)
```

$$n \mapsto (n-1)n$$

Die gerade definierten Funktionen kann man nun addieren, hintereinanderausführen und so weiter:

```
show(f + g)
show(expand(g(f)))
```

$$x \mapsto x^3 + x^2 + x$$

$$x^6 + 3\,x^5 + 3\,x^4 + x^3$$

Auch das Ergebnis von Funktionen für bestimmte Eingaben lässt sich mit gewohnter Notation berechnen. (Vorsicht, das Ergebnis hat einen symbolischen Datentyp - Zahlen-spezifische Funktionen wie factor funktionieren erst wie gewohnt, wenn man das Ergebnis zum Zahlen-Datentyp zurückkonvertiert:)

```
print f(4)
print parent(f(4))
print factor(f(4))
print factor(Integer(f(4)))
```

```
20
Symbolic Ring
20
2^2 * 5
```

Differenzieren und Integrieren

Sage kann die gerade definierten Funktionen auch ableiten:

```
diff(f)
```

```
x |--> 2*x + 1
```

Bei Funktionen mit mehreren Variablen kann man entweder angeben, nach welcher Variable (partiell) abgeleitet werden soll:

```
h(x, y) = y^2 + sin(x)
diff(h, x)
```

```
(x, y) |--> cos(x)
```

Oder man lässt sich gleich den ganzen Vektor aller Ableitungen berechnen:

```
diff(h)
```

```
(x, y) |--> (cos(x), 2*y)
```

Unbestimmtes ($\int f\,\mathrm{d}x$) und bestimmtes Integral ($\int_0^4 f\,\mathrm{d}x$) in Bezug auf die Integrationsvariable x:

```
print integrate(f, x)
print integrate(f, x, 0, 4)
```

```
x |--> 1/3*x^3 + 1/2*x^2
88/3
```

Integrale und Ableitungen können auch weitere "unbeteiligte" symbolische Variablen (Konstanten) enthalten (nebenbei: Integrations- und Ableitungs-Funktion hören auf mehrere ähnliche Namen, und auch die Integrationsgrenzen können in verschiedener Notation angegeben werden):

```
var("a")
integral(a*exp(-x), (x, 1, oo))
```

```
a*e^(-1)
```

Oft hängt die Integrierbarkeit eines Ausdrucks aber von zusätzlichen Informationen zu diesen Konstanten ab, etwa ob die Konstante negativ/null/positiv ist. Solche Zusatzannahmen (assumptions) müssen dann vor dem Integral deklariert werden:

```
integral(exp(-a*x), (x, 1, oo))
```

```
Traceback (click to the left of this block for traceback)
...
Is a positive, negative or zero?
```

```
assume(a>0)
integral(exp(-a*x), (x, 1, oo))
```

```
e^(-a)/a
```

Bei Ausdrücken, deren symbolische Integration scheitert, lässt sich eventuell dennoch eine numerische Approximation des Ergebnisses durchführen:

```
h=integral(tan(x)/x, (x, 1, pi/3))
h.show()
```

$$\int_{1}^{\frac{1}{3}\,\pi} \frac{\tan(x)}{x}\,dx$$

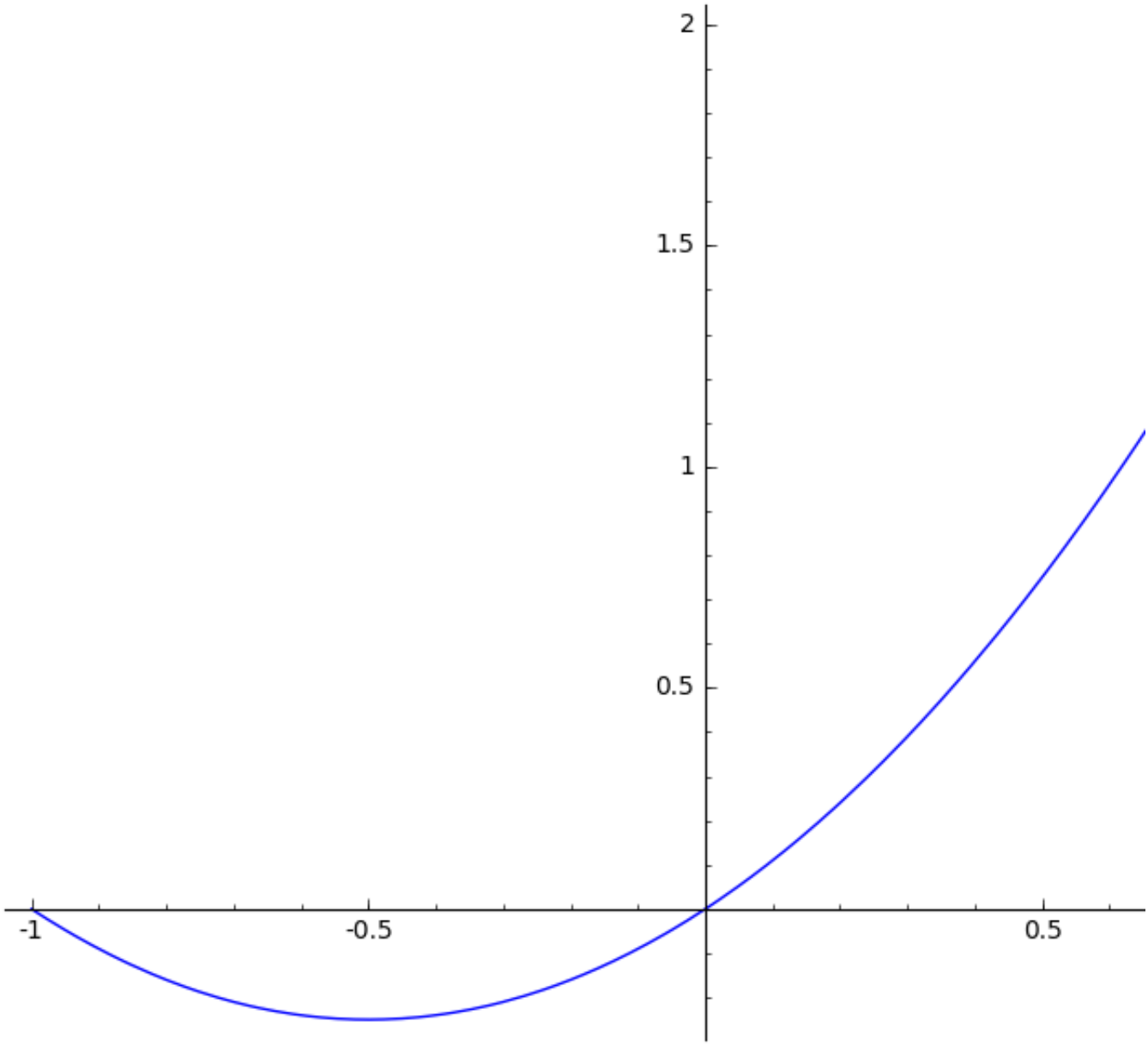
```
h.n()
```

```
0.07571599101702896
```

Funktionsgraphen

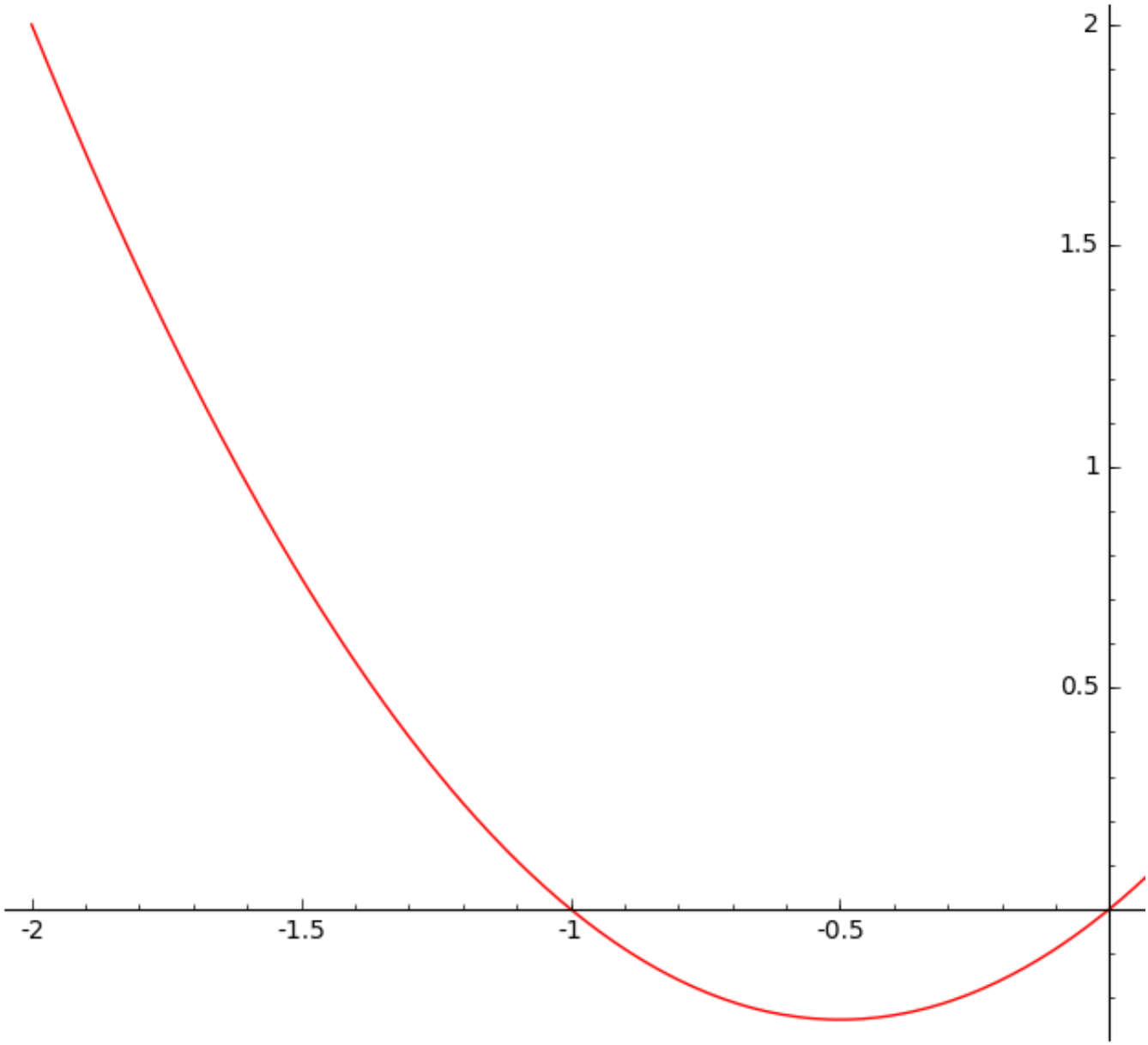
Funktionsgraph plotten:

```
f(x) = x^2 + x
plot(f)
```



Standardmäßig wird für x-Werte im Intervall $[-1, 1]$ geplottet; normalerweise möchte man den dargestellten Bereich selbst spezifizieren und vielleicht auch Ausgabe-Details wie Farben usw. angeben (die vielen weiteren Optionen kann man via `plot?` nachschlagen):

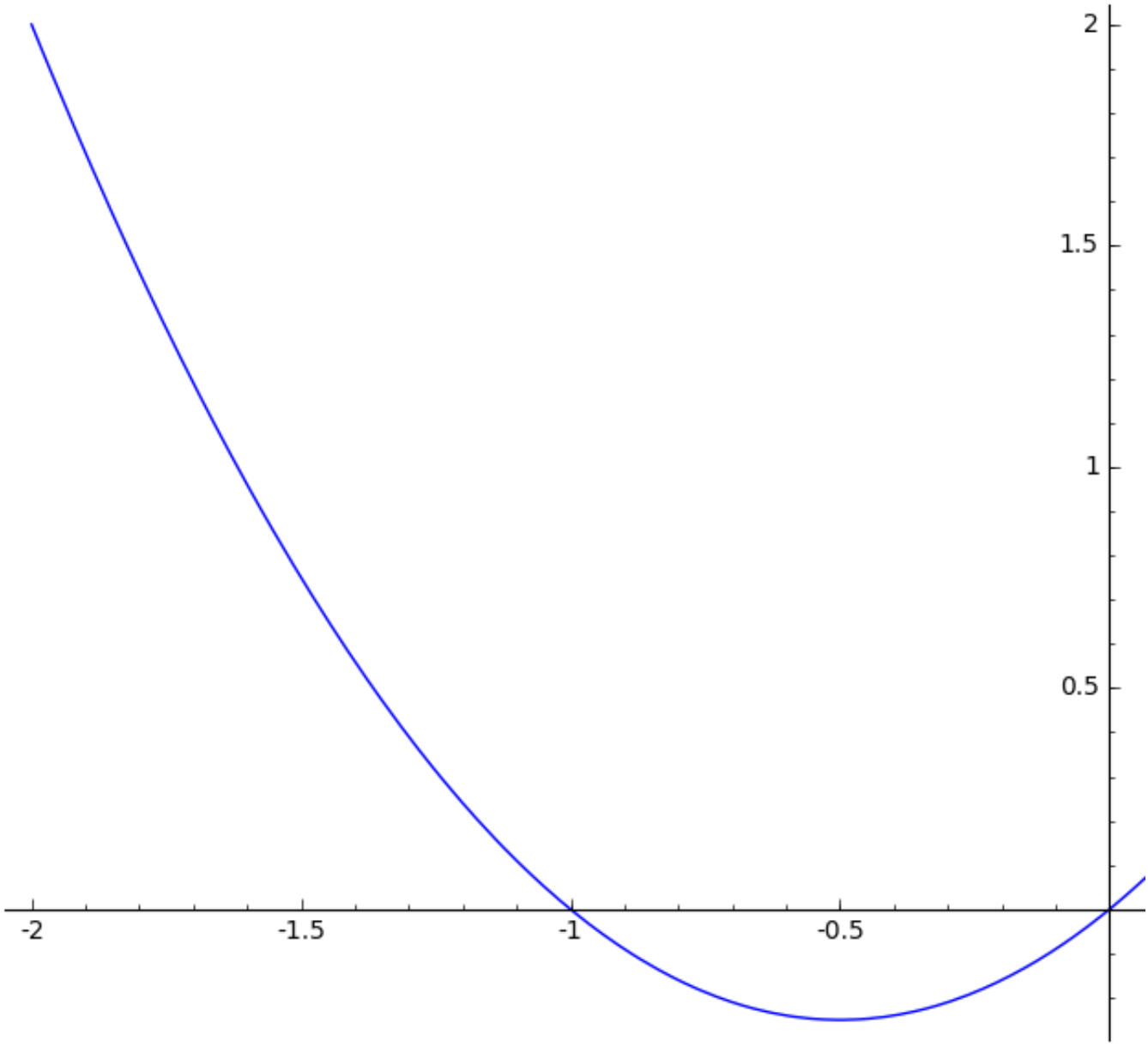
```
plot(f, (x, -2, 0.5), color="red")
```



Das Ergebnis eines Plots ist ein eigener Grafik-Datentyp, den man beispielsweise auch abspeichern kann:

```
p = plot(f, (x, -2, 0.5))
print parent(p)
show(p)
p.save("funktionsplot.pdf")
```

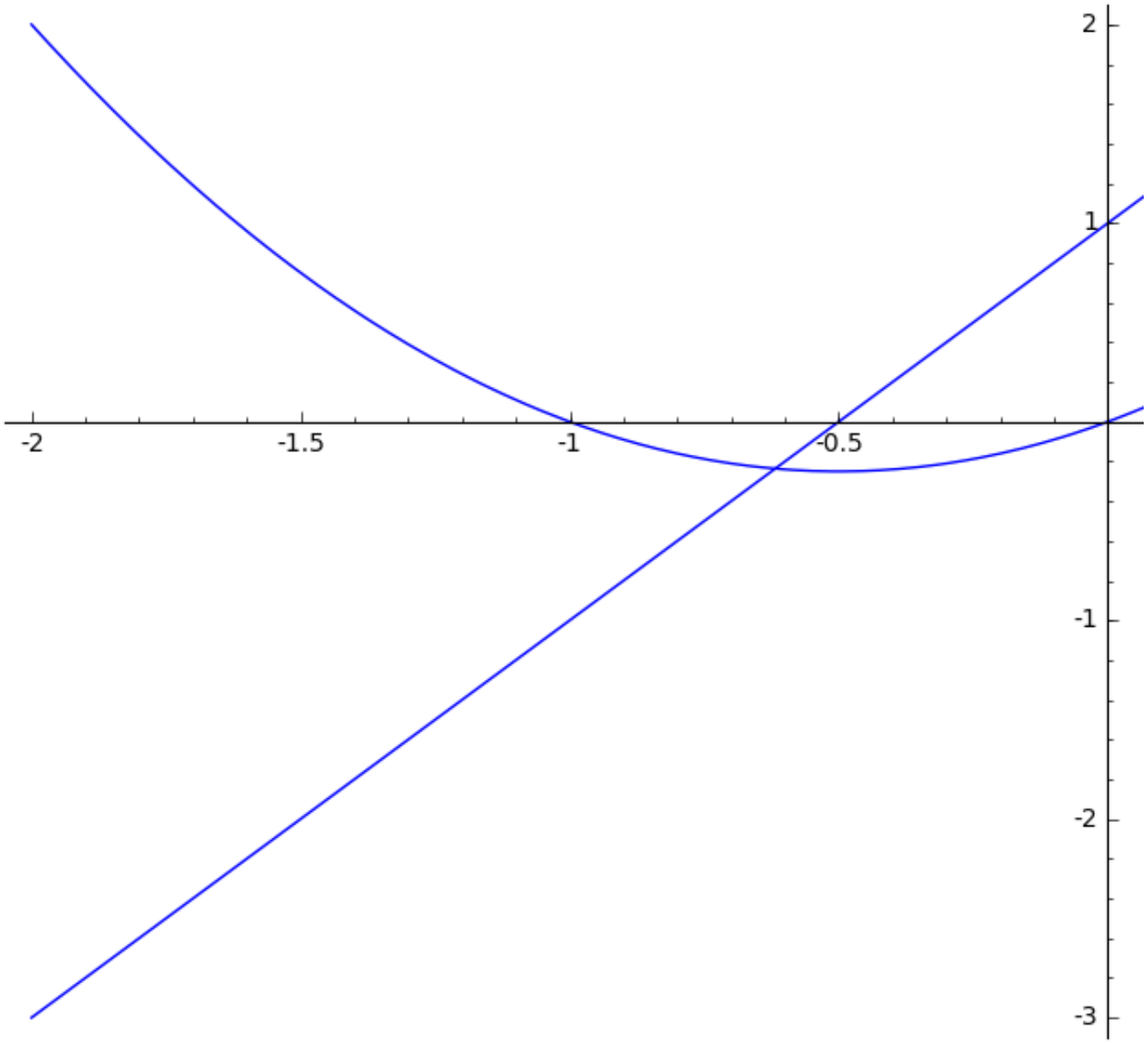
```
<class 'sage.plot.graphics.Graphics'>
```



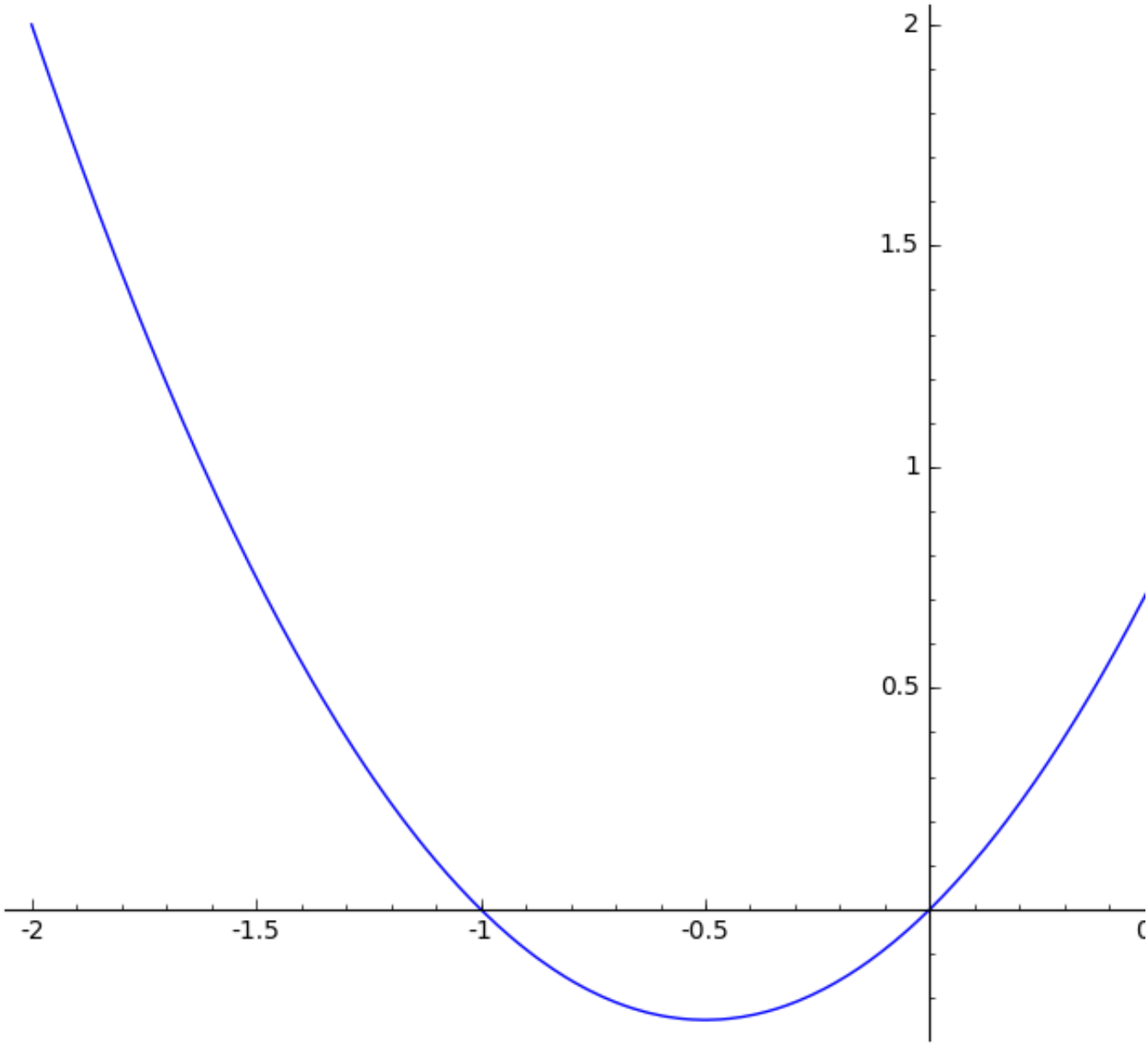
funktionsplot.pdf

Mehrere Plots in einem Bild zusammenkombinieren:

```
p_ableitung = plot(diff(f), (x, -2, 0.5), color = "blue")
show(p + p_ableitung)
```

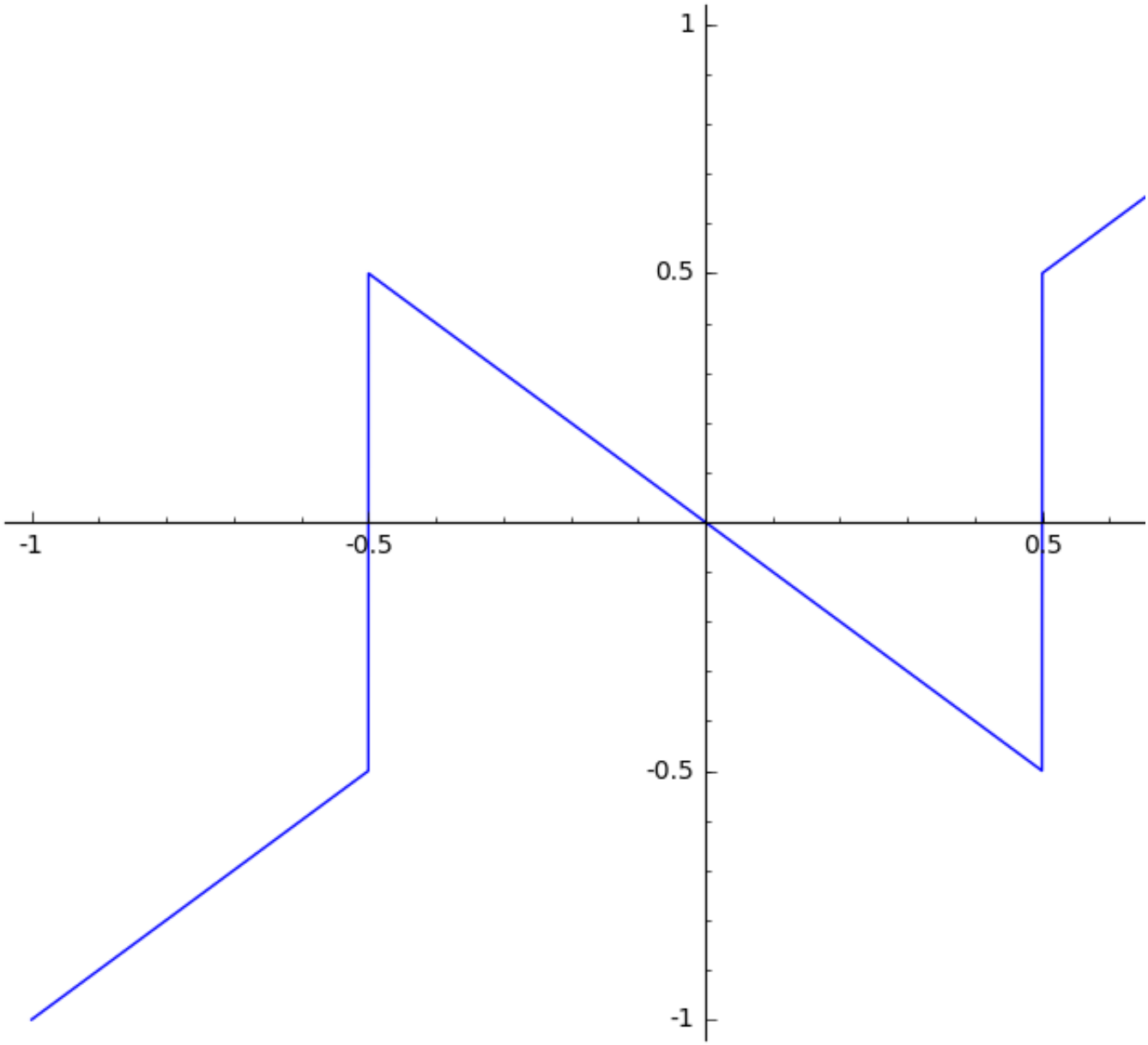



```
show(p+point((1,1),color="green",size=50))
```



Aber Vorsicht: bei Sprungstellen werden die Funktionen nicht immer richtig gezeichnet:

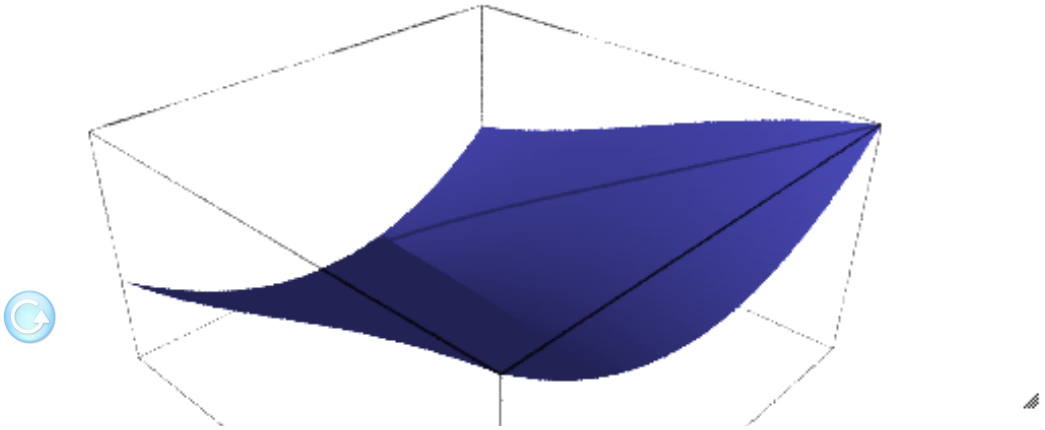
```
f1(x)=x*sign(x^2-1/4)
plot(f1,x,-1,1)
```



3D-Plots

3D-Plots für Funktionen mit mehreren Variablen sind interaktiv:

```
var("x y")
h(x, y) = y^2 + sin(x)
plot3d(h, (x, -1, 1), (y, -1, 1))
```



Grenzwerte

Grenzwerte von Folgen und Funktionen berechnen, hier beispielsweise $\lim_{x\to\infty} \frac{1}{x} = 0$ und $\lim_{x\to 0^-} \frac{1}{x} = \infty$:

```
f(x) = 1/x
limit(f, x = +oo)   # Grenzwert Richtung +unendlich
```

```
    x |--> 0
```

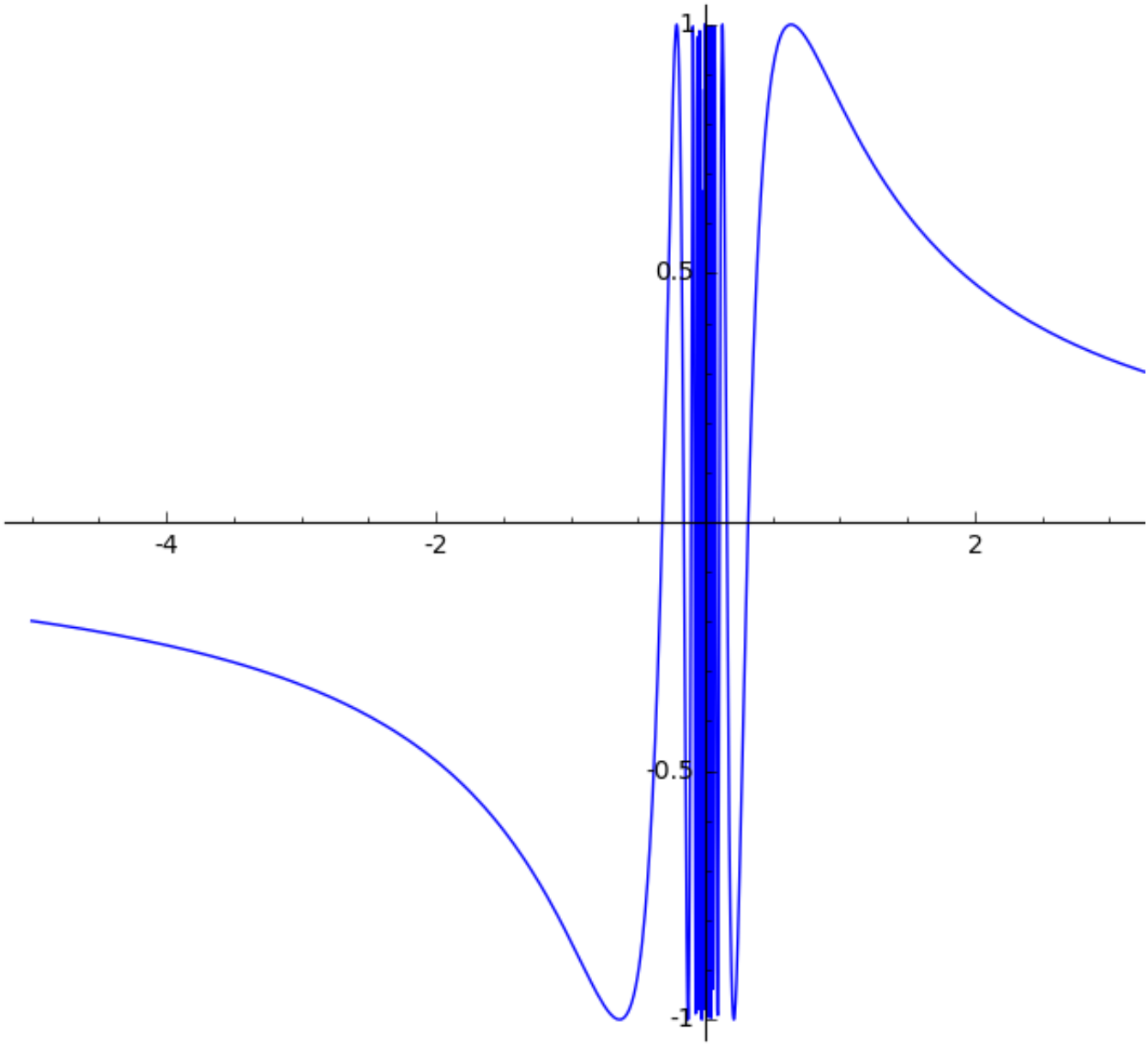
```
limit(f, x = 0, dir = "minus")   # Grenzwert Richtung 0 (von links)
```

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

Bei manchen Funktionen ist der Grenzwert undefiniert:

```
f(x) = sin(1/x)
print limit(f(x), x = 0)
plot(f, (x, -5,5))
```

```
    ind
```



Konvergenz einer Reihe, beispielsweise $\sum_{n=1}^{\infty} \frac{1}{n^2}$:

```
a(n) = 1/n^2
sum(a(n), n, 1, +oo)
```

```
1/6*pi^2
```

Gleichungen, Nullstellen

Nullstellen der Funktion $f(x) = x^2 - 2x - 3$ finden:

```
f(x) = x^2 - 2*x - 3
solutions = solve(f(x) == 0, x)
print solutions
```

```
[
x == 3,
x == -1
]
```

Um die Lösungen einer Gleichung im Code weiterzuverwenden, gibt es mehrere Möglichkeiten. Wenn man (eine der) Lösungen einfach in eine Funktion oder andere Gleichung einsetzen möchte:

```
f.subs(solutions[0])
```

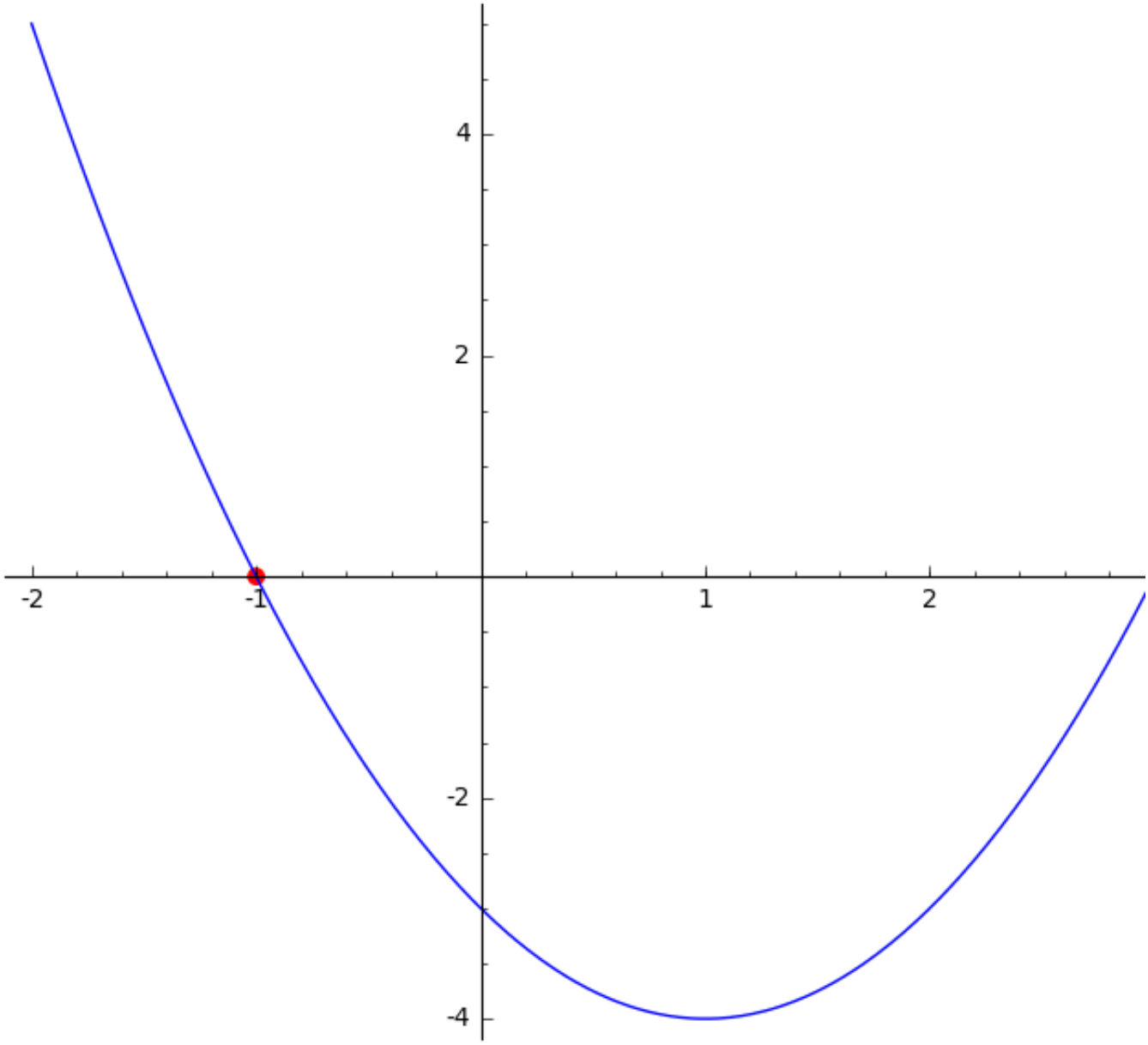
```
x |--> 0
```

Alternative kann man die Lösungen als "Liste von Dictionaries" ausgeben, die man normal verwenden kann (erster Index = Lösungsnummer, zweiter Index = Variablenname):

```
solutionlist = solve(f(x) == 0, x, solution_dict=True)
print solutionlist
solutionlist[0][x]
```

```
[{x: 3}, {x: -1}]
3
```

```
N1 = point((solutionlist[0][x],0),  color="red", size=50)
N2 = point((solutionlist[1][x],0), color="red", size=50)
plot(f, (x, -2, 4)) + N1 + N2
```



`solve` versucht die Gleichung symbolisch zu lösen, was bei komplizierteren Funktionen oft nicht möglich

ist. Für Polynomgleichungen vom Grad höchstens 4 gibt es Formeln (G. Cardano (1501-1576) zugeschrieben), und man kann beweisen, dass es für Polynome höheren Grades keine Lösungsformel gibt, ganz zu schweigen von Gleichungen mit transzendenten Funktionen (*sin*, *exp*, usw), die man nur in Ausnahmefällen explizit lösen kann. Eine Alternative ist die numerische Suche nach (approximierten) Nullstellen mit `find_root`, wobei man zusätzlich ein Suchintervall angeben muss und nur eine (nur approximierte) Lösung bekommt:

```
find_root(f, -2, 2)
```

```
-0.9999999999999241
```

Polynome

Wenn man von vornherein weiß, daß man es nur mit Polynomen zu tun hat, ist es effizienter, in einem Polynomring zu arbeiten.

Hier der Ring der Polynome in der einer Variable über dem Körper der rationalen Zahlen

```
QQ
```

```
Rational Field
```

```
Px = QQ[x]
Px
```

```
Univariate Polynomial Ring in x over Rational Field
```

Zu beachten ist, dass die verwendete Variable zuvor deklariert werden muss.

```
QQ[z]
```

```
Traceback (click to the left of this block for traceback)
...
NameError: name 'z' is not defined
```

```
var("z")
QQ[z]
```

```
Univariate Polynomial Ring in z over Rational Field
```

Zurück zur Variable x. Das Symbol x weiß nichts von diesem Ring

```
parent(x)
```

```
Symbolic Ring
```

Dazu muß es explizit umgewandelt werden. Wir behalten eine Kopie.

```
x1 = x
x = Px(x)
```

```
parent(x)
```

```
Univariate Polynomial Ring in x over Rational Field
```

Achtung, die Variable x (im Sinn der Informatik) und das Symbol x sind nicht das gleiche. Im vorherigen Ausdruck ist `x` nach wie vor Element des `Symbolic Ring`

```
x1
```

```
x
```

```
parent(x1)
```

```
Symbolic Ring
```

Man kann die obigen Operationen (Erzeugung des Polynomrings und Zuweisung der Variablen) auch in einem Schritt durchführen

```
P1.<y> = QQ[]
P1
```

```
Univariate Polynomial Ring in y over Rational Field
```

```
parent(y)
```

```
Univariate Polynomial Ring in y over Rational Field
```

Das gleiche in mehreren Variablen

```
P2.<u,v> = QQ[]
P2
```

```
Multivariate Polynomial Ring in u, v over Rational Field
```

```
parent(u)
```

```
Multivariate Polynomial Ring in u, v over Rational Field
```

```
parent(v)
```

```
Multivariate Polynomial Ring in u, v over Rational Field
```

Dabei sind die Ringe strikt getrennt.

```
u+x
```

```
Traceback (click to the left of this block for traceback)
...
TypeError: unsupported operand parent(s) for +: 'Multivariate
Polynomial Ring in u, v over Rational Field' and 'Univariate
Polynomial Ring in x over Rational Field'
```

Im Ring der Polynome sind immer alle Nullstellen berechenbar.

```
p= x^2-3
p
```

```
x^2 - 3
```

```
parent(p)
```

```
Univariate Polynomial Ring in x over Rational Field
```

Allerdings nur im zugrundeliegenden Ring.

```
p.roots()
```

```
[]
```

Wenn in anderen Ringen gesucht werden soll, muß dies explizit angegeben werden.

```
p.roots(ring=RR)
```

```
[(-1.73205080756888, 1), (1.73205080756888, 1)]
```

```
p.roots(ring=QQbar)
```

```
[(-1.732050807568878?, 1), (1.732050807568878?, 1)]
```

Das Polynom kann auch in den `Symbolic Ring` eingebettet werden.

```
pe = SR(p)
pe
```

```
x^2 - 3
```

```
parent(pe)
```

```
Symbolic Ring
```

jetzt können die dortigen Methoden angewendet werden.

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

```
Traceback (click to the left of this block for traceback)
...
TypeError: x is not a valid variable.
```

```
parent(x)
```

```
Univariate Polynomial Ring in x over Rational Field
```

```
var('x')
```

```
x
```

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

```
[x == -sqrt(3), x == sqrt(3)]
```