Sage 2: Analysis, Lineare Algebra

Symbolische Variablen und Funktionen

Variablen und symbolische Ausdrücke

In den bisherigen Beispielen in der Vorlesung waren Variablen jeweils durch den zugewiesenen Wert bestimmt. Aber in der Mathematik hat man sehr oft eigentlich nicht mit konkreten Werten, sondern mit symbolischen Variablen zu tun. Beispielsweise ist die Gleichung $(x+y)^2 = x^2 + 2 \cdot x \cdot y + y^2$ für jeden beliebigen Wert von $x, y \in \mathbb{R}$ wahr. Um in Sage mit Variablen zu arbeiten, die noch keinen Wert haben, muss man sie mit der var-Funktion als symbolische Variablen deklarieren:

```
var("x")
parent(x)
```

```
Symbolic Ring
```

Dann kann Sage beispielsweise Ausdrücke mit x ausmultiplizieren, oder allgemeine Gleichungen überprüfen:

```
2 * 7 * x * x * x^5
```

```
14*x^7
```

```
expand((x + 3)^5)
```

```
x^5 + 15*x^4 + 90*x^3 + 270*x^2 + 405*x + 243
```

Man kann auch den Wahrheitsgehalt von allgemeinen Rechenformeln bzw. Umformungen (die keine weiteren Hintergrundvoraussetzungen haben) überprüfen, z.B. binomische Formel:

```
var("x y")
bool((x+y)^2 == x^2+y^2+2*x*y)
```

```
True
```

Rechenregeln für komplexe Zahlen:

```
var("w z")
bool((x + y*I) * (w + z*I) == (w*x - z*y) + (y*w + x*z) * I)
```

```
True
```

```
var("phi")
bool(e^(I*phi) == cos(phi) + I*sin(phi))
```

```
True
```

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 arguments (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:

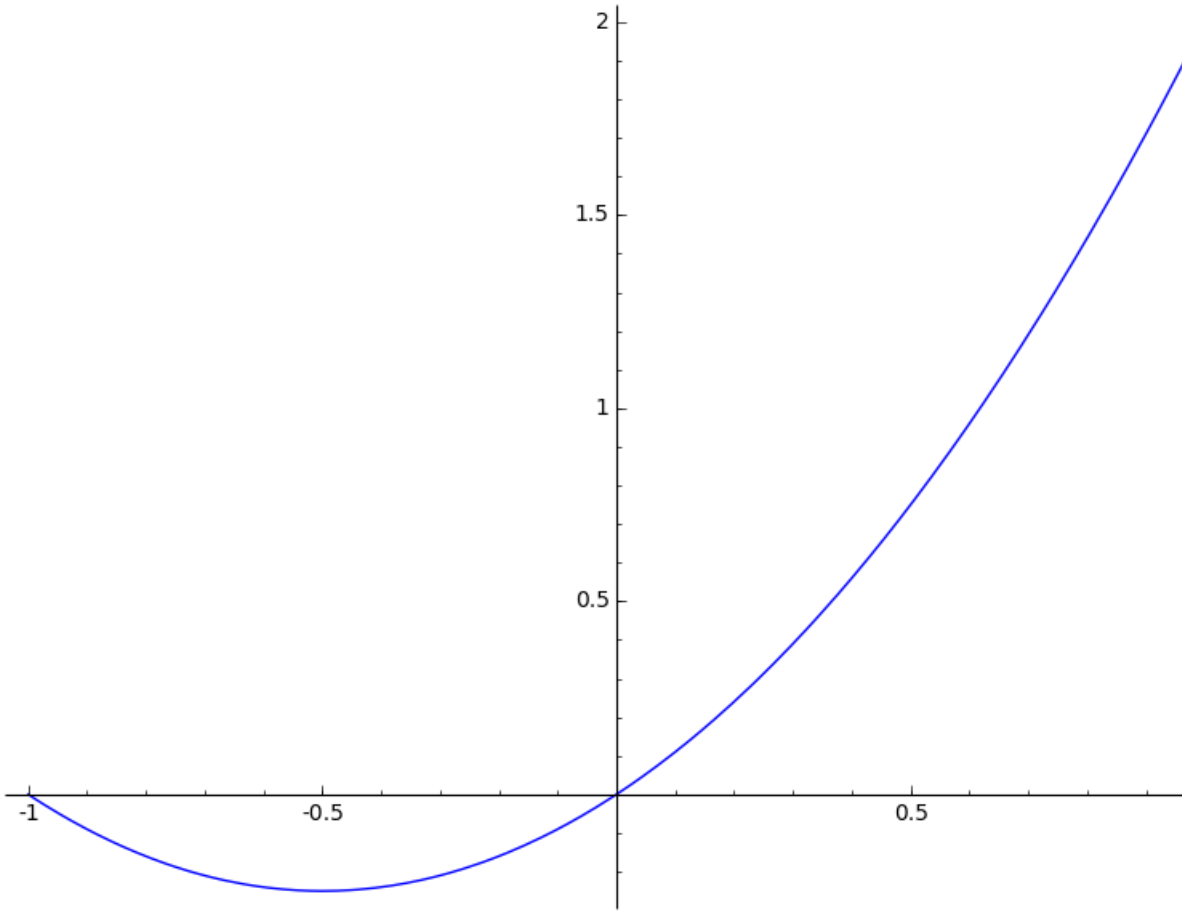
```
print integrate(f, x)
print integrate(f, x, 0, 4)
```

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

Plots

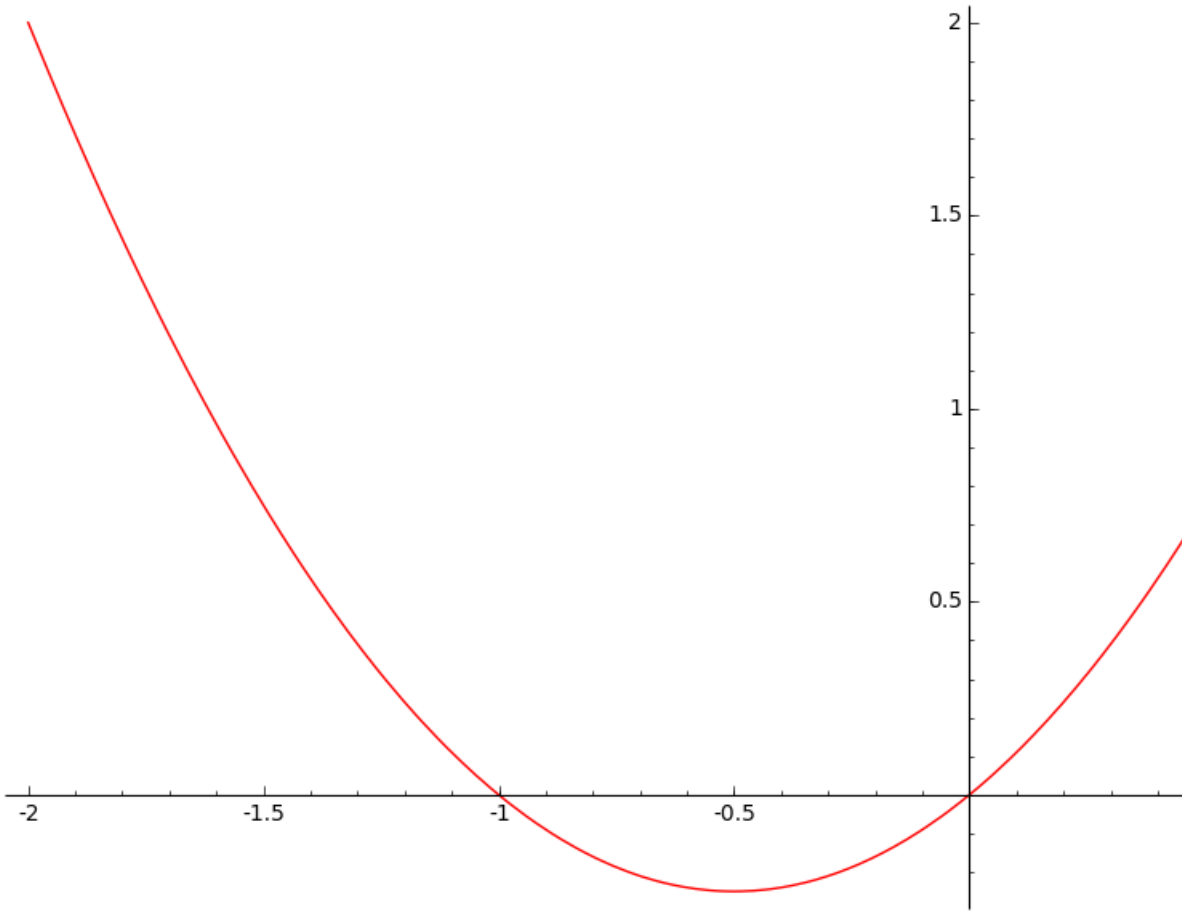
Funktionsgraph plotten:

```
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):

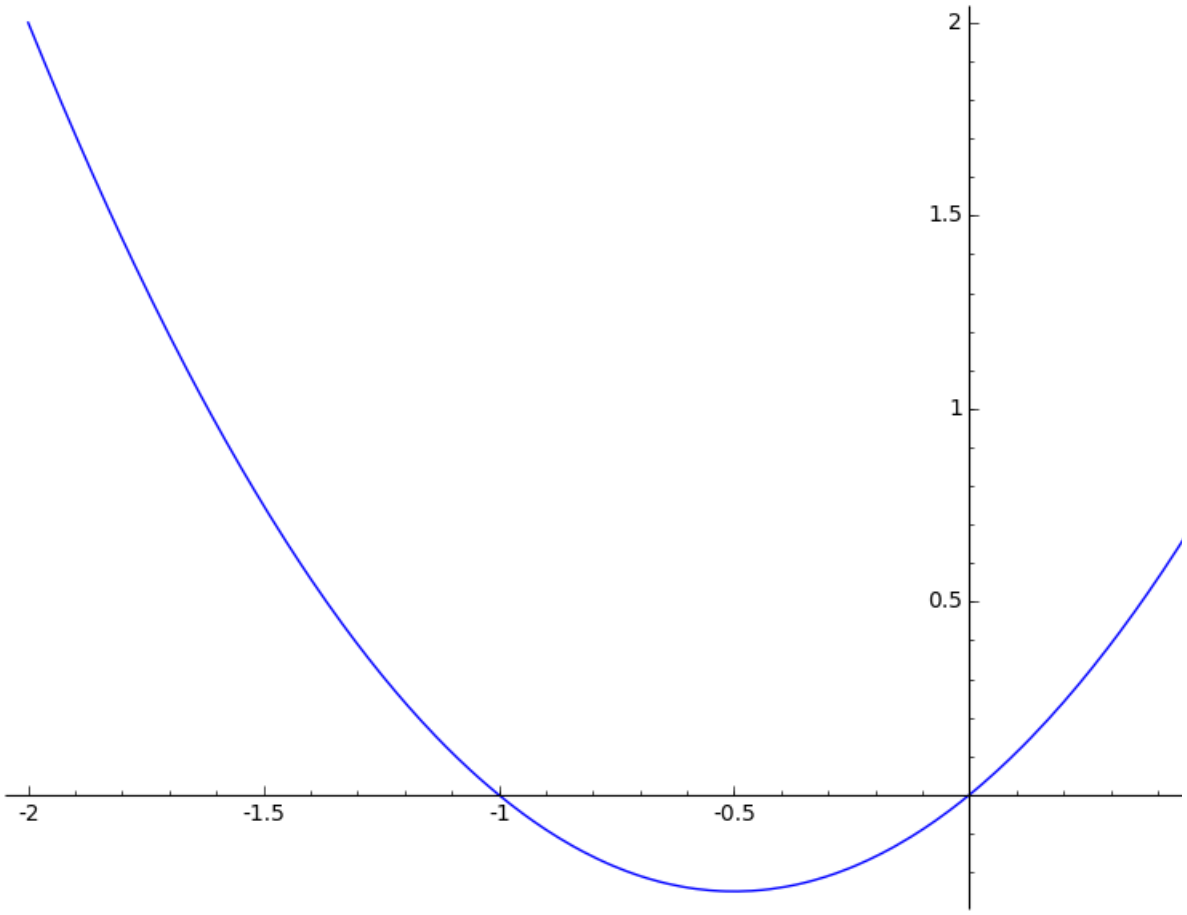
```
plot(f, (x, -2, 0.5), color="red")
```



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:

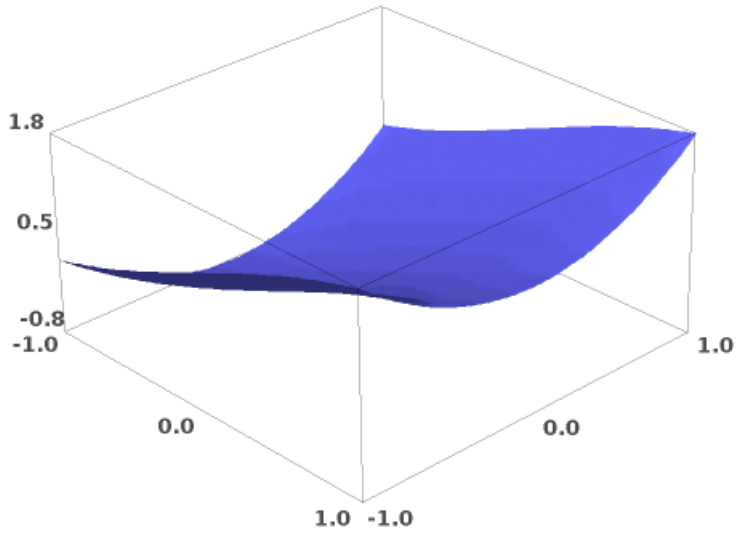
```
p_ableitung = plot(diff(f), (x, -2, 0.5), color = "blue")
show(p + p_ableitung)
```

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))
```

Sleeping... Make Interactive



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])
```

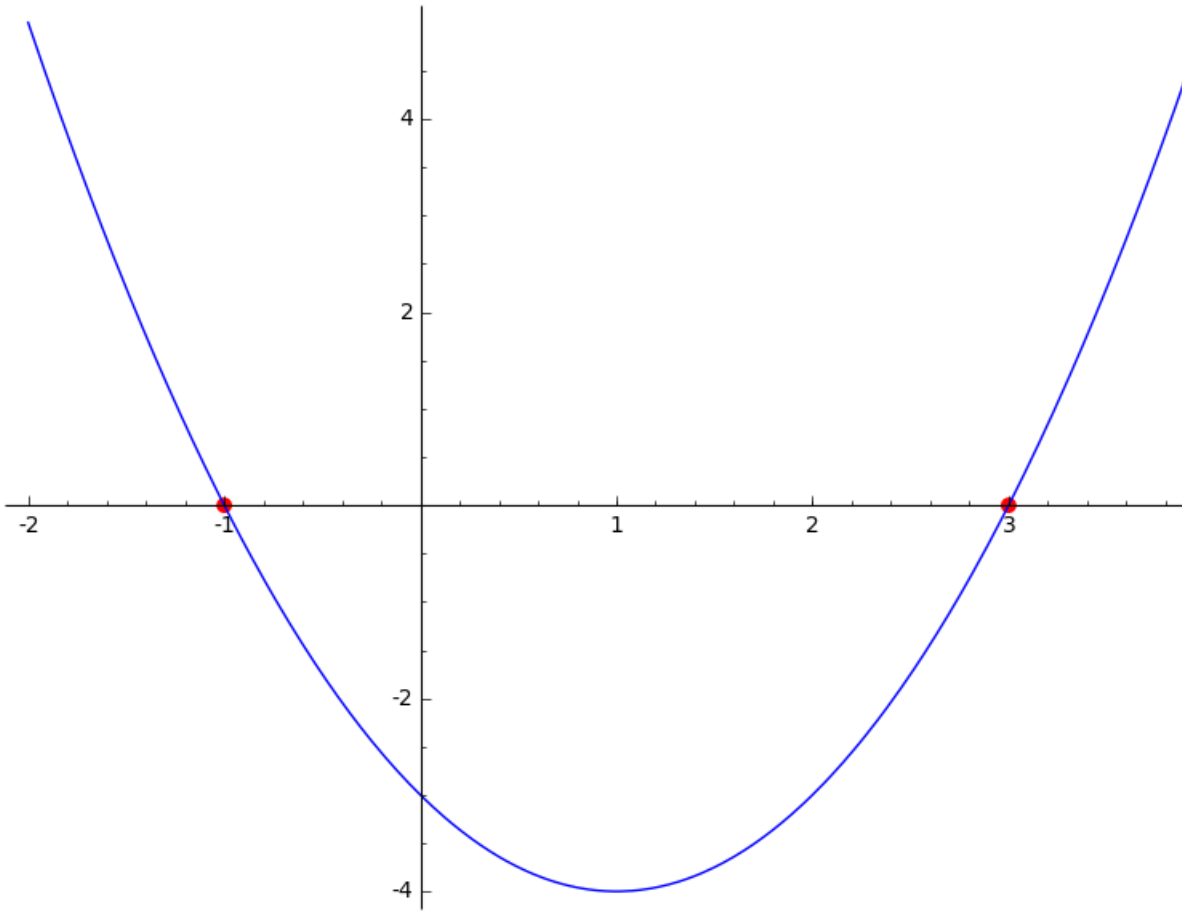
```
x |--> 0
```

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. Eine alternative ist das numerische Suchen 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
```

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)
plot(f, (x, -5,5))
limit(f(x), x = 0)
```

```
ind
```

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

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

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

Lineare Algebra

Heute nur ein Kurzeinblick in die Definition von Vektoren und Matrizen und grundsätzlichste Rechenoperationen; mehr Details werden wir uns nächste Stunde ansehen.

Basics: Vektoren und Matrizen

Zur Deklaration eines Vektors erstellt man eine Liste der Koordinateneinträge und baut daraus eine Variable vom Typ `vector`:

```
v = vector([1,2,3])
w = vector([0,0,4])
print v
show(v)
parent(v)
```

```
(1, 2, 3)
```

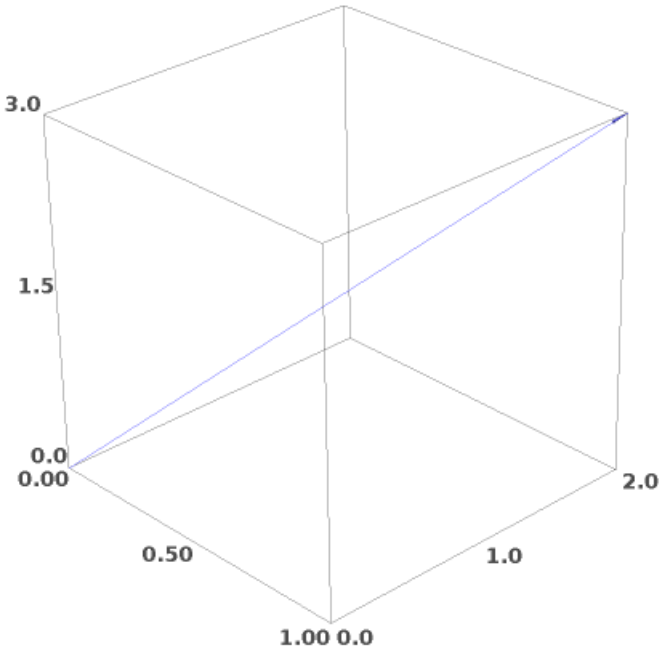
$$(1, 2, 3)$$

```
Ambient free module of rank 3 over the principal ideal domain
Integer Ring
```

Vektoren können auch geplottet werden (2D oder 3D, als Pfeil dargestellt):

```
plot(v)
```

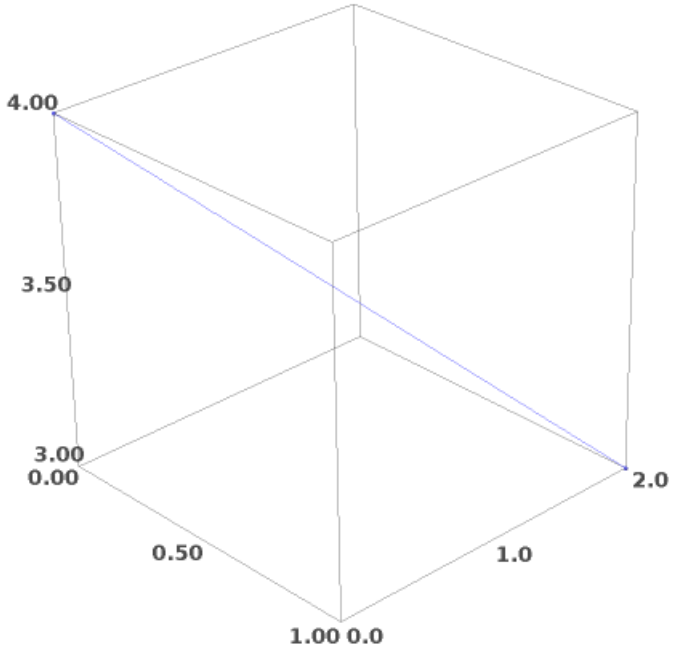
Sleeping... Make Interactive



Alternativ Darstellung als Punkt, oder Verbindungslinien zwischen einzelnen Punkten:

```
point(v) + point(w) + line([v, w])
```

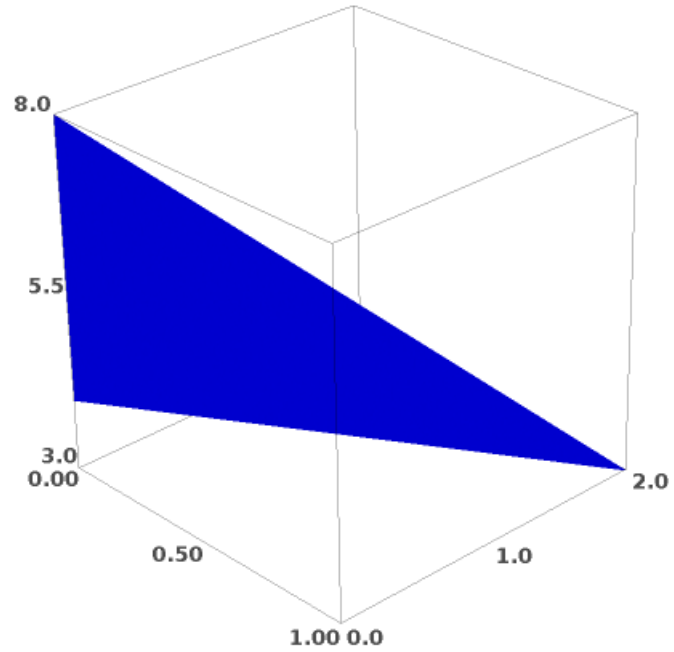
Sleeping... Make Interactive



Punkte als Eckpunkte eines Polygons verwenden:

```
polygon([v, w, 2*w])
```

Sleeping... Make Interactive



Einfache Rechenoperationen: Addition und Subtraktion; Skalarmultiplikation (aka innere Multiplikation, mit `*` oder `dot_product`); Kreuzprodukt (aka äußeres Produkt, mit `cross_product`); Länge des Vektors (Norm):

```
print v + w
```

```
print v - w
print v * w
print v.dot_product(w)
print v.cross_product(w)
print v.norm()
```

```
(1, 2, 7)
(1, 2, -1)
12
12
(8, -4, 0)
sqrt(14)
```

Matrizen werden ähnlich definiert, mit einer Liste der einzelnen Zeilenvektoren der Matrix:

```
A = matrix([[1,2,3], [4,5,6]])
print A
show(A)
```

```
[1 2 3]
[4 5 6]
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

Einzeilige Matrizen werden im Gegensatz zu Vektoren mit eckigen statt runden Klammern sowie ohne trennende Kommas dargestellt; Vektoren können flexibel sowohl als Zeilen-, als auch als Spalten-Vektor interpretiert werden (wobei Sage im Zweifel zu "Zeilenvektor" tendiert), während die Dimensionen bei einer (einzeiligen/einspaltigen) Matrix festgelegt sind:

```
print vector([1,2,3])
print matrix([[1,2,3]])
print matrix([[1],[2],[3]])
```

```
(1, 2, 3)
[1 2 3]
[1]
[2]
[3]
```

Multiplikation von Matrix mit anderer Matrix (oder Vektor), sofern die Dimensionen zusammenpassen:

```
A * v
```

```
(14, 32)
```