2018-01-10_Sage3

Inhaltsverzeichnis

Rekursive Funktionen mit Gedächtnis

Wir betrachten die Folge der Fibonaccizahlen, die rekursiv definiert ist:

```
def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

```
fib(20)
```

```
10946
```

Bei dieser Definition werden für eine Auswertung unzählige Funktionsaufrufe getätigt, sodaß bald Schluß ist:

```
%time
  fib(34)
```

```
9227465
CPU time: 12.61 s,  Wall time: 13.29 s
```

Es ist daher wünschenswert, daß die Funktion sich "merkt", welche Werte bereits bekannt sind. Das läßt sich mit `cached_function` bewerkstelligen:

```
fib1 = cached_function(fib)
```

```
fib1(20)
```

```
10946
```

Allerdings löst das unser Problem nicht, weil `fib` ebenfalls `fib1` aufrufen müßte, um Zugriff auf das Gedächtnis zu bekommen. Die Lösung ist der "magische" sogenannte *Dekorator* `@cached_function`:

```
@cached_function
def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Er bewirkt, daß sofort eine Kopie von `fib` mit Gedächtnis angelegt wird, die den gleichen Namen hat.

Damit geht es jetzt sehr schnell.

```
%time
  fib(35)
```

```
14930352
```

```
CPU time: 0.00 s,  Wall time: 0.00 s
```

Der Wertespeicher kann direkt eingesehen werden

```
fib.get_cache()
```

```
__main__:2: DeprecationWarning: The .get_cache() method is
deprecated, use the .cache attribute instead.
See http://trac.sagemath.org/19694 for details.
```

```
{((12,), ()): 233, ((20,), ()): 10946, ((35,), ()): 14930352, ((17,), ()): 2584, ((5,), ()):
```

}

und gelöscht werden.

```
fib.clear_cache()
```

Allerdings muß man darauf achten, die maximale Rekursionstiefe nicht zu überschreiten:

```
fib(500)
```

```
WARNING: Output truncated!
full_output.txt
```

```
Traceback (click to the left of this block for traceback)
...
RuntimeError: maximum recursion depth exceeded

full_output.txt
```

Das kann vermieden werden, indem man in kleinen Schritten vorgeht:

```
%time
  fib(250)
```

```
127765235729247325860370338946550318986595564473522 49

CPU time: 0.00 s,  Wall time: 0.01 s
```

```
%time
  fib(500)
```

```
225591516161936330872512695036072072046011324913758190588638866418474627738686883405015987052796 9

CPU time: 0.01 s,  Wall time: 0.01 s
```

Manipulation von Listen

```
l = range(10)
l
```

$[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$

Teillisten extrahieren werden.

```
l[2:7]
```

$[2, 3, 4, 5, 6]$

```
l[2:10:2]
```

$[2, 4, 6, 8]$

```
l2 = range(20,30)
l2
```

$[20, 21, 22, 23, 24, 25, 26, 27, 28, 29]$

Verketten von Listen mit +

```
l + l2
```

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]

Eine Liste ist im Wesentlichen ein Zeiger auf Speicheradressen.

```
l1 = [1,2,3]
```

Bei Zuweisung an eine andere Variable wird nur die Adresse kopiert.

```
l2 = l1
l2
```

[1, 2, 3]

```
l1 == l2
```

True

Die Listen `l1` und `l2` sind *identisch* und zeigen auf die gleichen Speicheradressen. Wenn eine der Listen verändert wird, dann auch die andere.

```
l1.append(4)
l2
```

[1, 2, 3, 4]

Das kann zu Endlosschleifen führen, wenn man nicht aufpaßt.

```
for x in l1:
    l2.append(x)
```

```
^C
Traceback (click to the left of this block for traceback)
...
__SAGE__
```

Um alle Elemente einer Liste einzeln zu kopieren, muß explizit der Befehl `copy` verwendet werden.

```
l1 = [1,2,3]
l3 = copy(l1)
l3
```

[1, 2, 3]

```
l1.append(4)
l1
```

$[1, 2, 3, 4]$

```
l3
```

$[1, 2, 3]$

Allerdings wird dabei nur die erste Ebene kopiert.

```
l4 = [l1,l1]
l4
```

$[[1, 2, 3, 4], [1, 2, 3, 4]]$

```
l1.append(6)
l4
```

$[[1, 2, 3, 4, 6], [1, 2, 3, 4, 6]]$

```
l5 = copy(l4)
l5
```

$[[1, 2, 3, 4, 6], [1, 2, 3, 4, 6]]$

```
l1.append(7)
l5
```

$[[1, 2, 3, 4, 6, 7], [1, 2, 3, 4, 6, 7]]$

Dagegen werden beim Vergleich von Listen alle Elemente rekursiv verglichen.

```
l1
```

$[1, 2, 3, 4, 6, 7]$

```
l6 = copy(l1)
l6
```

$[1, 2, 3, 4, 6, 7]$

```
l1 == l6
```

True

Transformationen von Listen: map und filter

map(f, l) erzeugt aus der Liste `l` eine neue Liste, in der alle Elemente `x` durch `f(x)` ersetzt werden.

```
l = srange(1,15)
l
```

$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]$

```
map(factor, l)
```

$[1, 2, 3, 2^2, 5, 2 \cdot 3, 7, 2^3, 3^2, 2 \cdot 5, 11, 2^2 \cdot 3, 13, 2 \cdot 7]$

Die ursprüngliche Liste bleibt dabei unverändert.

```
l
```

$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]$

filter(f, l) behält nur diejenigen Elemente `x`, für die `f(x)` wahr ist.

```
filter(is_prime, l)
```

$[2, 3, 5, 7, 11, 13]$

Wieder bleibt die ursprüngliche Liste unverändert.

```
l
```

$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]$

Abfrage, ob ein Element in der Liste vorkommt.

```
4 in l
```

True

Zählen, wie oft ein Element in einer Liste vorkommt.

```
l.count(4)
```

1

```
l.append(4)
```

```
l
```

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 4]

```
l.count(4)
```

2

Um mit `map` arithmetische Funktionen anwenden zu können, müssen diese erst definiert werden.

```
def quadrat(x):
    return x^2
```

```
map(quadrat, l)
```

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 16]

Für solche Einmalfunktionen gibt es eine elegantere Methode.

Funktionen: Lambda

Sogenannte *anonyme Funktionen*, meistens nur einmalig gebrauchte Einzeiler, können mit dem lambda-Kalkül definiert werden. `return` kann hier entfallen.

```
f = lambda x: x^2
```

```
f(2)
```

4

```
type(f)
```

<type 'function'>

Das Resultat ist also eine ganz normale Funktion, und kann als solche auch mit mehreren Inputparametern deklariert werden:

```
add = lambda x,y: x+y
```

```
add(1,1)
```

2

Man kann eine neue Funktion bilden, in der eines der Argumente gebunden wird (Curry), z.B. eine

Funktion, die 42 addiert:

```
add42 = lambda y: add(y,42)
```

```
add42(1)
```

```
43
```

Man kann auch eine Funktion definieren, die eine Addierfunktion zurückgibt:

```
addy = lambda y: lambda x:add(x,y)
```

```
addy(42)
```

```
<function <lambda> at 0x17de73410>
```

```
addy(42)(1)
```

```
43
```

```
type(addy)
```

```
<type 'function'>
```

```
(lambda x,y:(x(y)))(lambda u:u^2,3)
```

```
9
```

Listenverarbeitung: reduce

Um eine Funktion rekursiv mit den Elementen einer Liste zu füttern, gibt es die rekursive Funktion **reduce**. Sie ist rekursiv definiert durch `reduce(f,[x0,x1,...]) = f(x0, reduce(f,[x1,x2,...]))`

```
l = [1,2,3,4]
```

Um z.B. die Elemente einer Liste aufzusummieren, kann man die Elemente rekursiv addieren,

```
reduce(+, l)
```

```
Traceback (click to the left of this block for traceback)
...
SyntaxError: invalid syntax
```

allerdings hat der Operator '+' den internen Namen `add`:

```
reduce(operator.add, l)
```

```
10
```

alternativ mit einer zweiargumentigen anonymen Funktion

```
reduce(lambda x,y: x+y, l)
```

```
10
```

```
reduce(operator.mul, l)
```

```
24
```

List Comprehension

Anstelle von `map` und `filter`, insbesondere wenn beide kombiniert werden, ist es oft eleganter und intuitiver, mit sogenannten *List Comprehensions* zu arbeiten. Anwendung einer Funktion analog zu `map`:

```
l2 = srange(1,13)
[is_prime(k) for k in l2]
```

```
[False, True, True, False, True, False, True, False, False, False, True, False]
```

```
[k^2  for k in l2]
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144]
```

ergibt das gleiche wie

```
map(lambda x:x^2, l2)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144]
```

Auswahl von Elementen

```
[k for k in l2 if is_prime(k)]
```

```
[2, 3, 5, 7, 11]
```

ergibt das gleiche wie

```
filter(is_prime, l2)
```

```
[2, 3, 5, 7, 11]
```

In Kombination: Quadrate der Primzahlen

```
[k^2 for k in l2 if is_prime(k)]
```

$[4, 9, 25, 49, 121]$

Als Beispiel erzeugen wir eine Liste aller rationalen Zahlen mit Nenner nicht größer als 5

```
rat5 = [[m/n for m in [1..n]] for n in [1..5]]
rat5
```

$\left[[1], \left[\frac{1}{2}, 1\right], \left[\frac{1}{3}, \frac{2}{3}, 1\right], \left[\frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\right], \left[\frac{1}{5}, \frac{2}{5}, \frac{3}{5}, \frac{4}{5}, 1\right]\right]$

```
rat5a = reduce(operator.add,rat5)
rat5a
```

$\left[1, \frac{1}{2}, 1, \frac{1}{3}, \frac{2}{3}, 1, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1, \frac{1}{5}, \frac{2}{5}, \frac{3}{5}, \frac{4}{5}, 1\right]$

Um doppelte Elemente hinauszuwerfen, kann man auf den Datentyp `set` zurückgreifen.

```
set(rat5a)
```

```
set([1, 1/2, 1/3, 2/3, 1/4, 3/4, 1/5, 2/5, 3/5, 4/5])
```

und in eine Liste zurückverwandeln.

```
ls = list(set(rat5a))
```

```
ls.sort()
ls
```

$\left[\frac{1}{5}, \frac{1}{4}, \frac{1}{3}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}, 1\right]$

Tupel

Tupel sind unveränderliche Listen.

```
t = 1,2,3
t
```

$(1, 2, 3)$

```
t.append(5)
```

```
Traceback (click to the left of this block for traceback)
...
AttributeError: 'tuple' object has no attribute 'append'
```

```
for i in t:
    print i
```

```
1
2
3
```

Praktische Anwendung: Austausch der Werte zweier Variablen ("*swap*"):

```
x = 1
y = 2
x,y
```

$(1, 2)$

```
x,y = y,x
```

```
x,y
```

$(2, 1)$

Tupel können dazu verwendet werden, mehrere Werte aus einer Funktion zurückzubekommen. Beispiel: Quotient und Rest einer ganzzahligen Division.

```
n = 17
m = 5
q,r = n.quo_rem(m)
```

```
q
```

3

```
r
```

2

Listenverarbeitung: zip

Wir wollen über zwei Listen im Reißverschlußsystem parallel iterieren, dazu müssen sie zunächst mit **zip** aneinandergeheftet werden:

```
l1 = ['a','b','c']
```

```
l2 = [1,2,3]
```

```
l12 = zip(l1,l2)
l12
```

$[(a, 1), (b, 2), (c, 3)]$

Wir haben jetzt eine Liste von Paaren

```
type(l12[0])
```

```
<type 'tuple'>
```

über diese Liste kann jetzt mit einem Paar von Variablen iteriert werden:

```
for x,y in l12:
    print(x+str(y))
```

```
a1
b2
c3
```

Anwendungsbeispiel: wir bilden das Polynom summe ai x^i. Zunächst erzeugen wir die benötigten Variablen, das geht elegant mit einem Formatstring analog zu print. `%s` steht als Platzhalter für eine String.

```
print("x%s" % 'a')
```

```
xa
```

```
var("a%s" % 1)
```

a_1

```
aa = [var("a%s" % i) for i in range(10)]
aa
```

$[a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9]$

```
zip(aa, range(10))
```

$[(a_0, 0), (a_1, 1), (a_2, 2), (a_3, 3), (a_4, 4), (a_5, 5), (a_6, 6), (a_7, 7), (a_8, 8), (a_9, 9)]$

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

x

```
[a*x^b for a,b in zip(aa,range(10))]
```

$$[a_0, a_1 x, a_2 x^2, a_3 x^3, a_4 x^4, a_5 x^5, a_6 x^6, a_7 x^7, a_8 x^8, a_9 x^9]$$

```
reduce(operator.add, [a*x^b for a,b in zip(aa,range(10))])
```

$$a_9 x^9 + a_8 x^8 + a_7 x^7 + a_6 x^6 + a_5 x^5 + a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0$$

Der Datentyp Dictionary

Ein *Dictionary* ist eine Datenstruktur ähnlich einer Liste, die mit beliebigen Objekten anstelle der Zahlen 0...n indiziert werden kann. Die Eingabe erfolgt mit geschwungenen Klammern.

```
d = { 'a':1, 5:3, 'c':ZZ }
d
```

$$\{\mathtt{a} : 1, \mathtt{c} : \mathbf{Z}, 5 : 3\}$$

```
d['a']
```

$$1$$

Nicht zugewiesene Indices ergeben eine Fehlermeldung.

```
d['xy']
```

```
Traceback (click to the left of this block for traceback)
...
KeyError: 'xy'
```

können aber durch eine Zuweisung erzeugt

```
d['xy'] = 'z'
d['xy']
```

```
z
```

und danach beliebig verändert werden

```
d['xy'] = 42
d['xy']
```

$$42$$

Ein Dictionary besteht aus *Keys* und *Values*.

```
d.keys()
```

$[\mathtt{a}, \mathtt{c}, \mathtt{xy}, 5]$

```
d.values()
```

$[1, \mathbf{Z}, 42, 3]$

```
d.items()
```

$[(\mathtt{a}, 1), (\mathtt{c}, \mathbf{Z}), (\mathtt{xy}, 42), (5, 3)]$

Iteration über die Indices.

```
for x in d:
    print x
```

```
a
c
xy
5
```

```
for x in d:
    print d[x]
```

```
1
Integer Ring
42
3
```

Man kann sich die Lösungen von Gleichungen als Dictionary erzeugen lassen.

```
var('x,y')
solve(x^2-3 == 0, x, solution_dict=True)
```

$[\{x : -\sqrt{3}\}, \{x : \sqrt{3}\}]$

```
sol=solve([x+y == 1, x-3*y == 5], [x,y], solution_dict=True)
sol
```

$[\{y : -1, x : 2\}]$

```
sol[0][x]
```

2

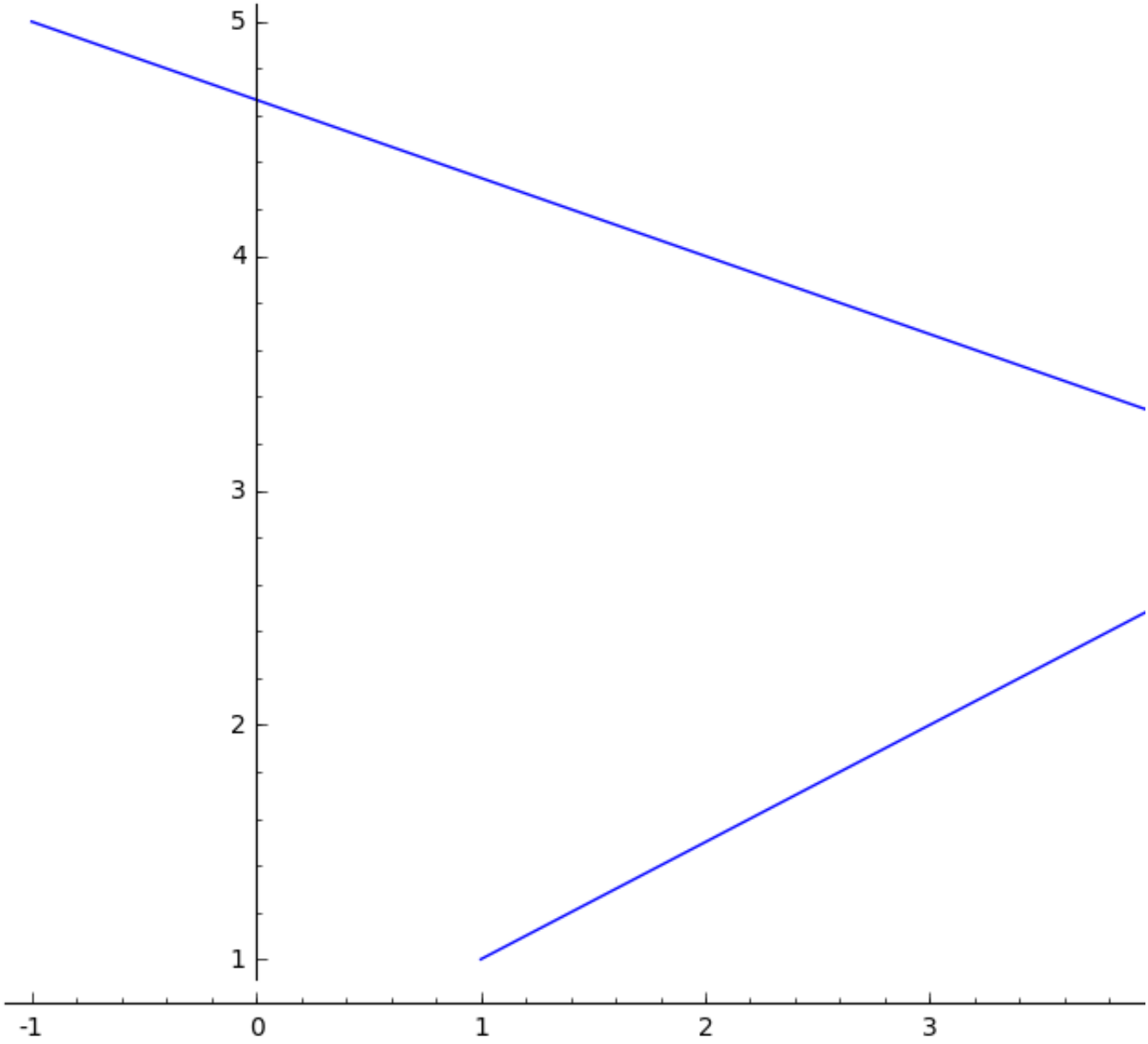
Ebene Geometrie

Wir zeichnen ein Dreieck mit seinen Schwerlinien.

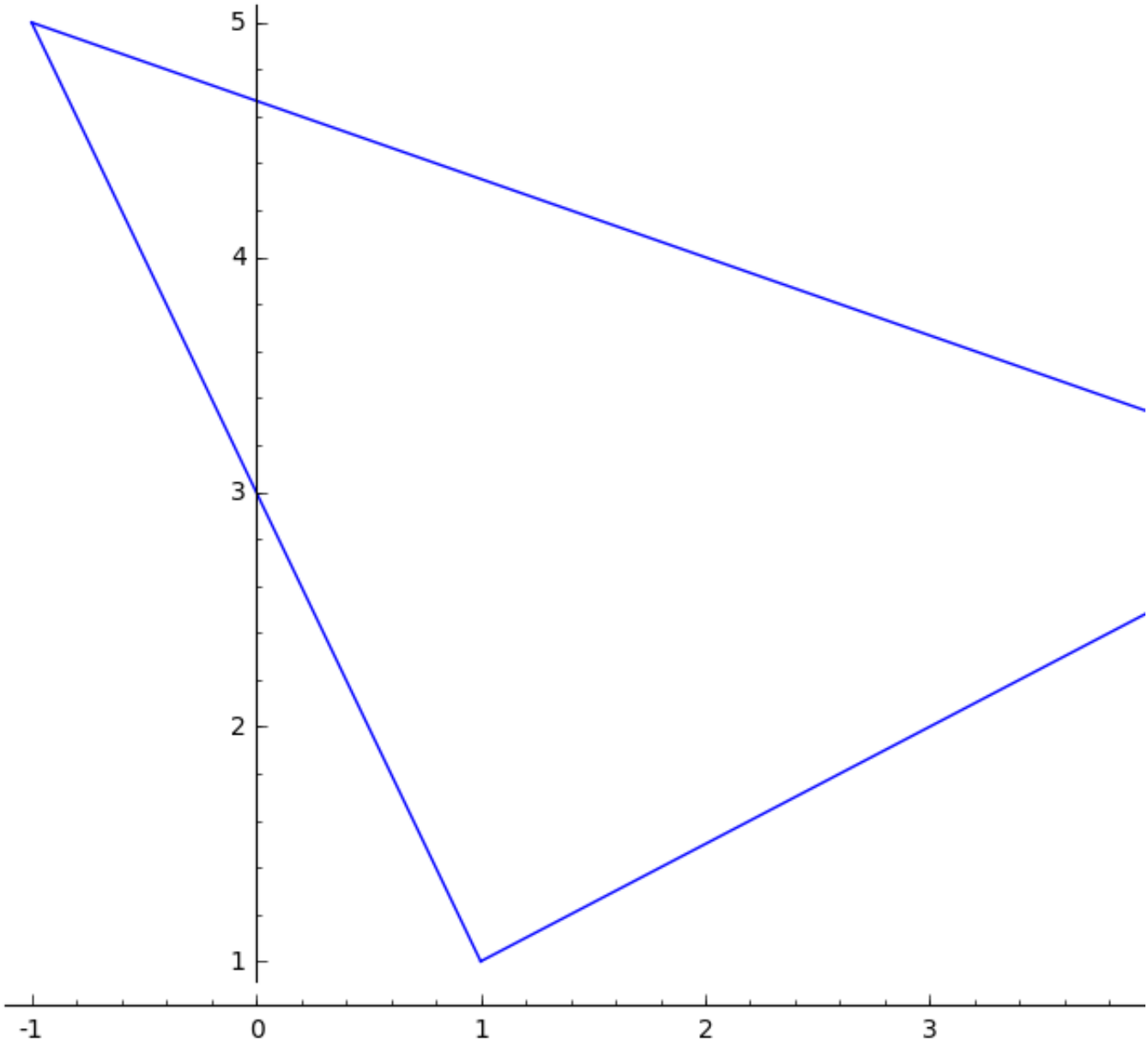
```
A=[1,1]
B=[5,3]
C=[-1,5]
```

Die Befehle **line**, **point**, **circle** etc. erzeugen die entsprechenden graphischen Objekte und man kann das Bild der Reihe nach aufbauen. Koordinaten können als Listen, Tupel oder Vektoren übergeben werden.

```
g = line([A,B,C])
g
```




```
g = g+line([A,C])
g.show()
```



Die Schwerlinien verbinden die Mittelpunkte der Seiten mit den gegenüberliegenden Scheiteln.

Die Punktedarstellung als Listen erlaubt leider nicht, daß man die Punkte vektorartig miteinander verknüpfen kann:

```
(A+B)/2
```

```
Traceback (click to the left of this block for traceback)
...
TypeError: unsupported operand parent(s) for /: '<type
'list'>' and 'Integer Ring'
```

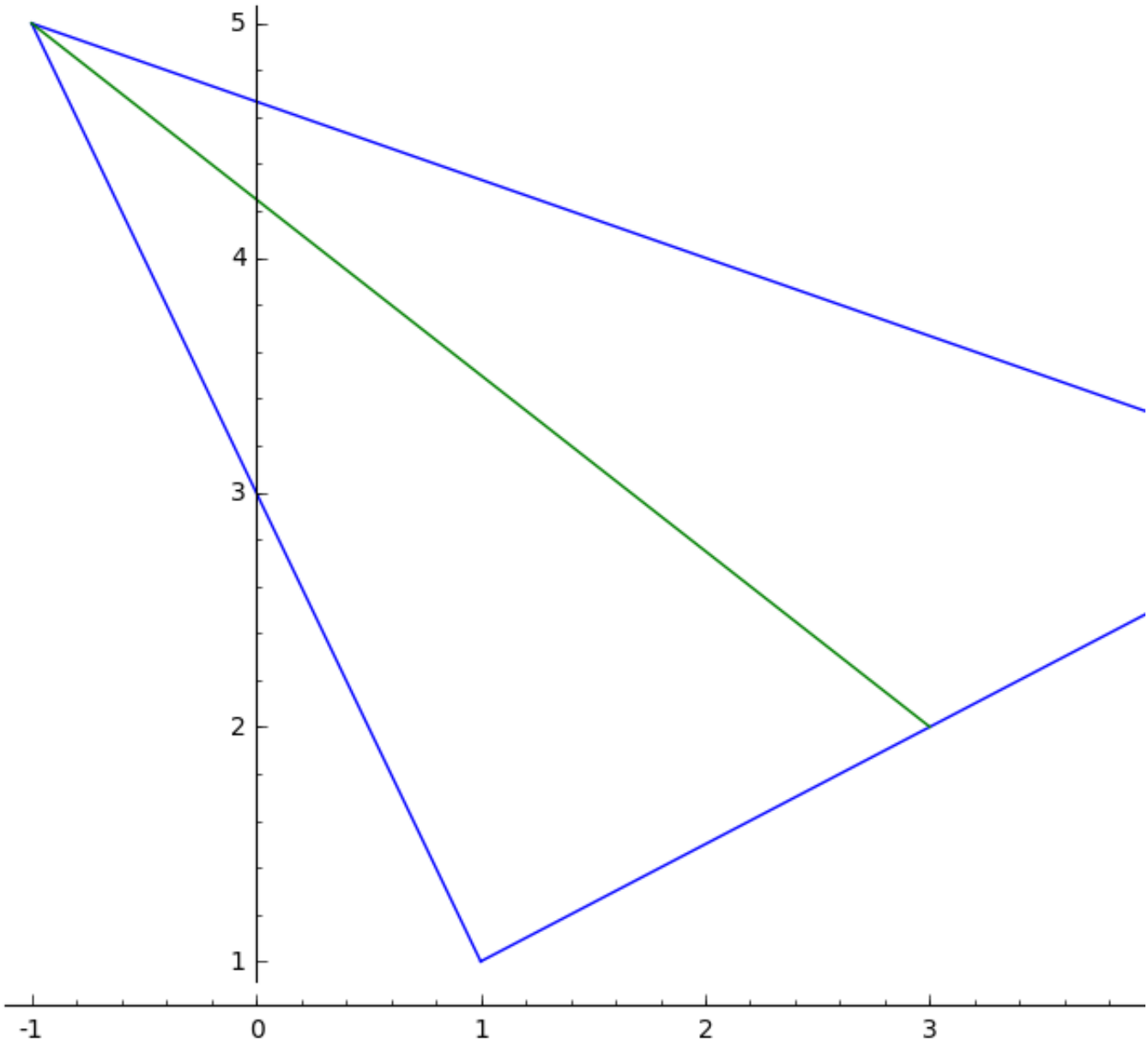
Zunächst aber müssen die Punkte in Vektoren umgewandelt werden.

```
A=vector(A)
B=vector(B)
C=vector(C)
```

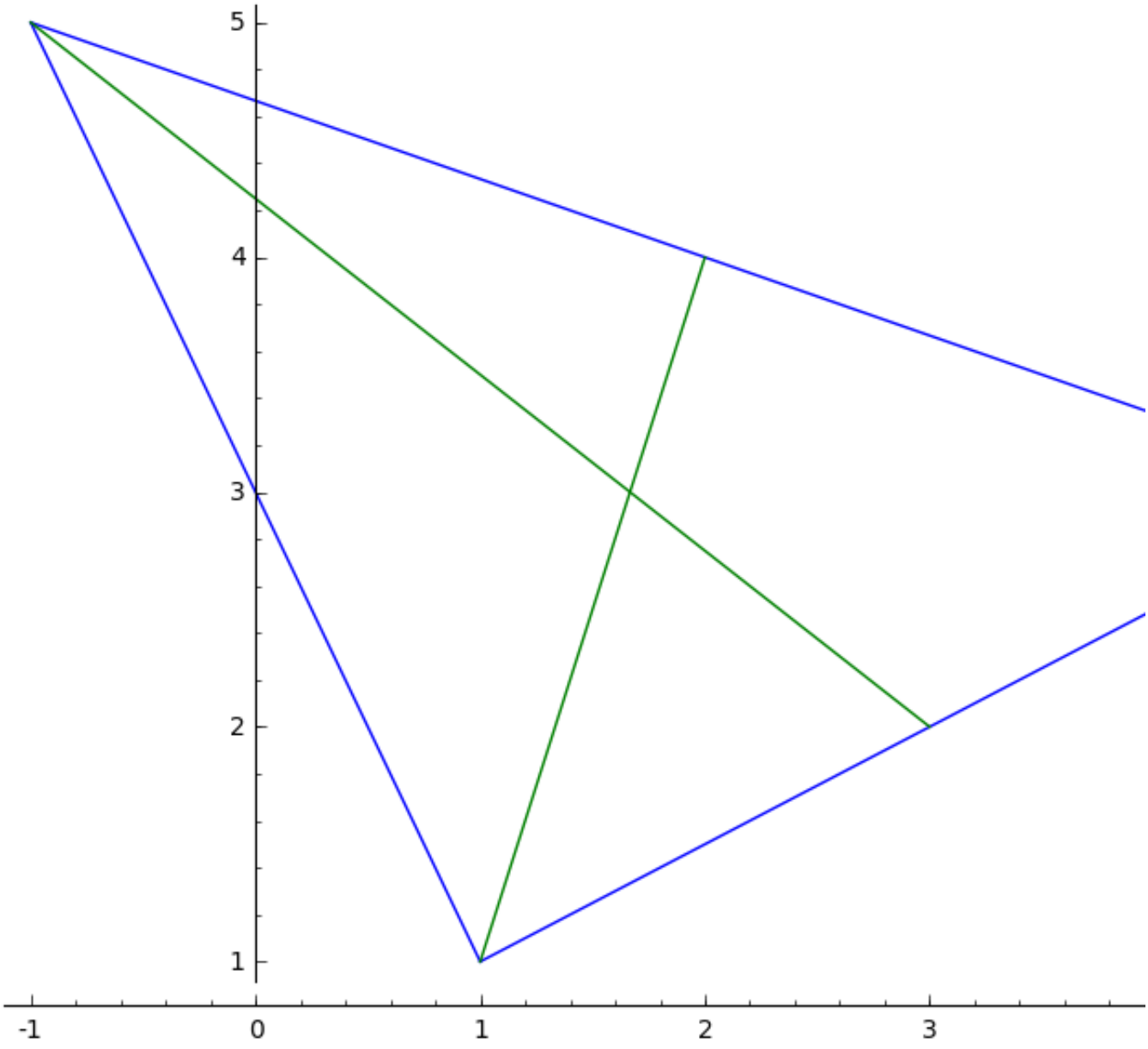
```
(A+B)/2
```

$(3, 2)$

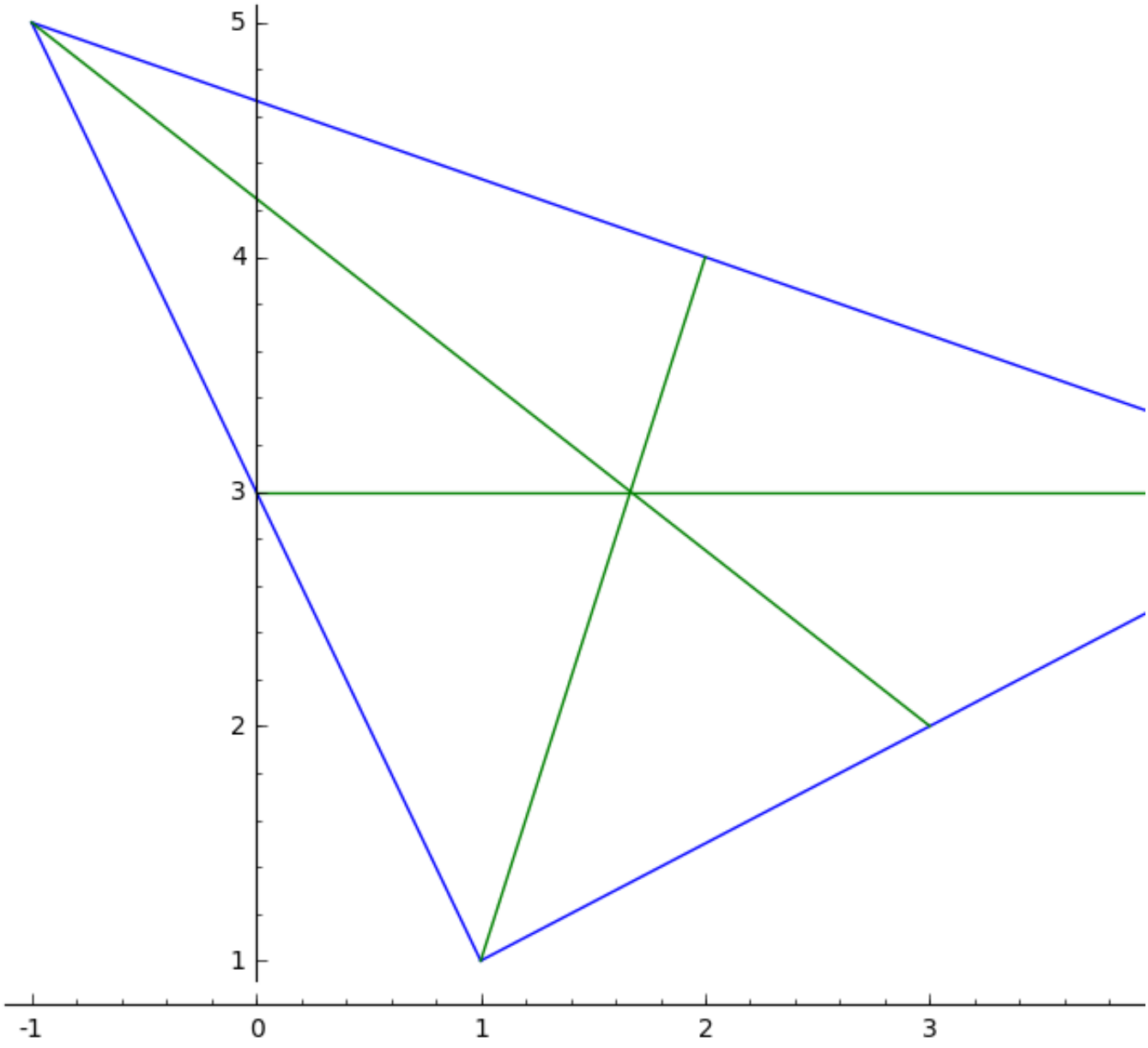
```
g = g + line([(A+B)/2,C],color='green')
g
```




```
g = g+line([(B+C)/2,A],color='green')
g
```




```
g = g+line([(C+A)/2,B],color='green')
g
```



Um den Schwerpunkt zu bestimmen, müssen die Schwerlinien geschnitten werden.

```
var('s,t')
```

(s, t)

```
sA = A + s * ((B+C)/2-A)
sA
```

$(s + 1, 3\,s + 1)$

```
sB = B + t * ((A+C)/2-B)
sB
```

$(-5\,t + 5, 3)$

Gleichungen mit Vektoren kann man nicht direkt lösen:

```
solve(sA-sB,[s,t])
```

```
Traceback (click to the left of this block for traceback)
...
TypeError: The first argument must be a symbolic expression or a
list of symbolic expressions.
```

wir müssen sie zuerst in Listen zurückverwandeln.

```
list(sA-sB)
```

$$[s + 5\,t - 4, 3\,s - 2]$$

```
st = solve(list(sA-sB),[s,t])
st
```

$$\left[\left[s = \left(\frac{2}{3}\right), t = \left(\frac{2}{3}\right)\right]\right]$$

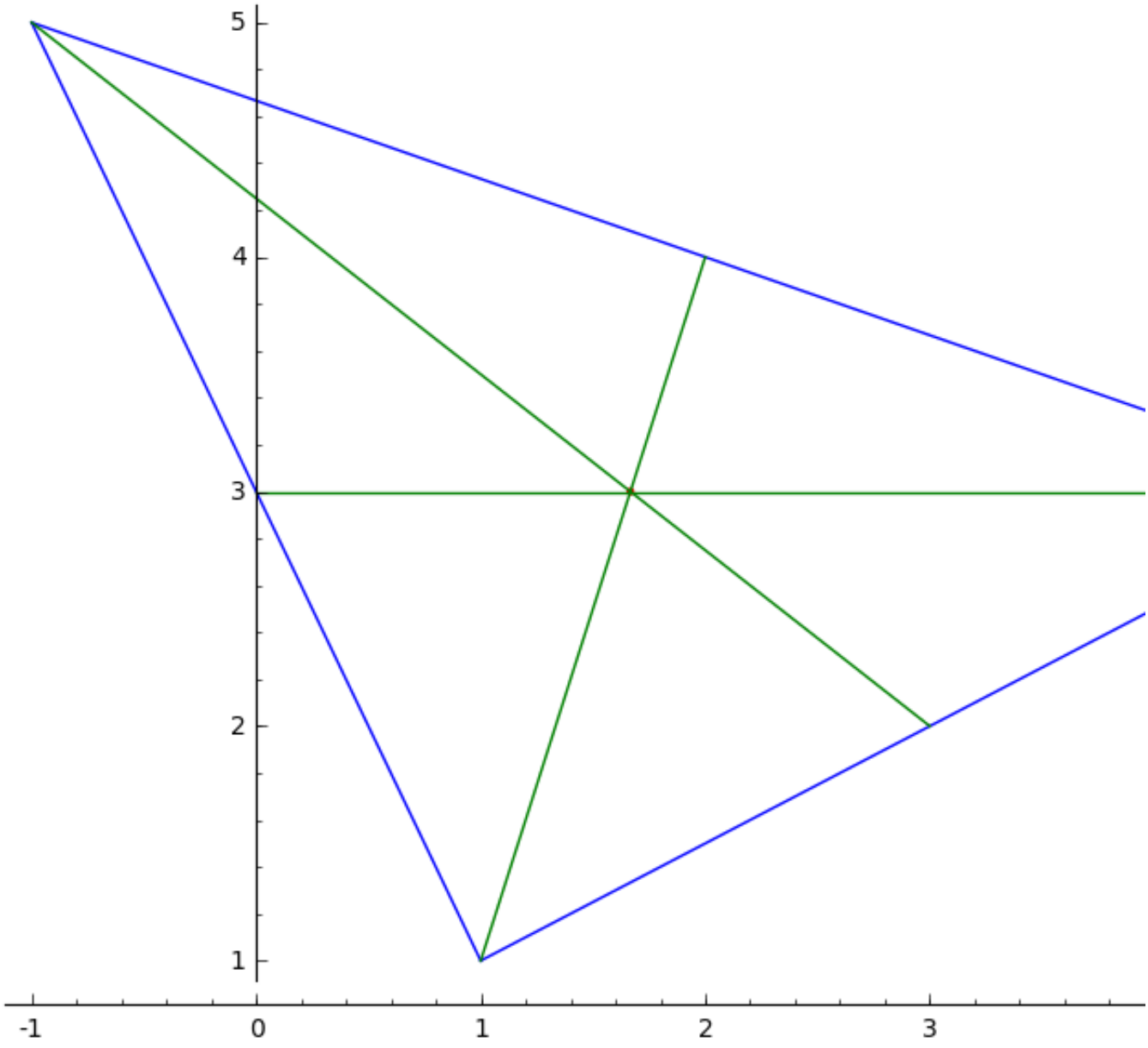
Wie immer wird eine Liste von Lösungen zurückgegeben, auch wenn die Lösung eindeutig ist.

```
S = sA.subs(st[0])
S
```

$$\left(\frac{5}{3}, 3\right)$$

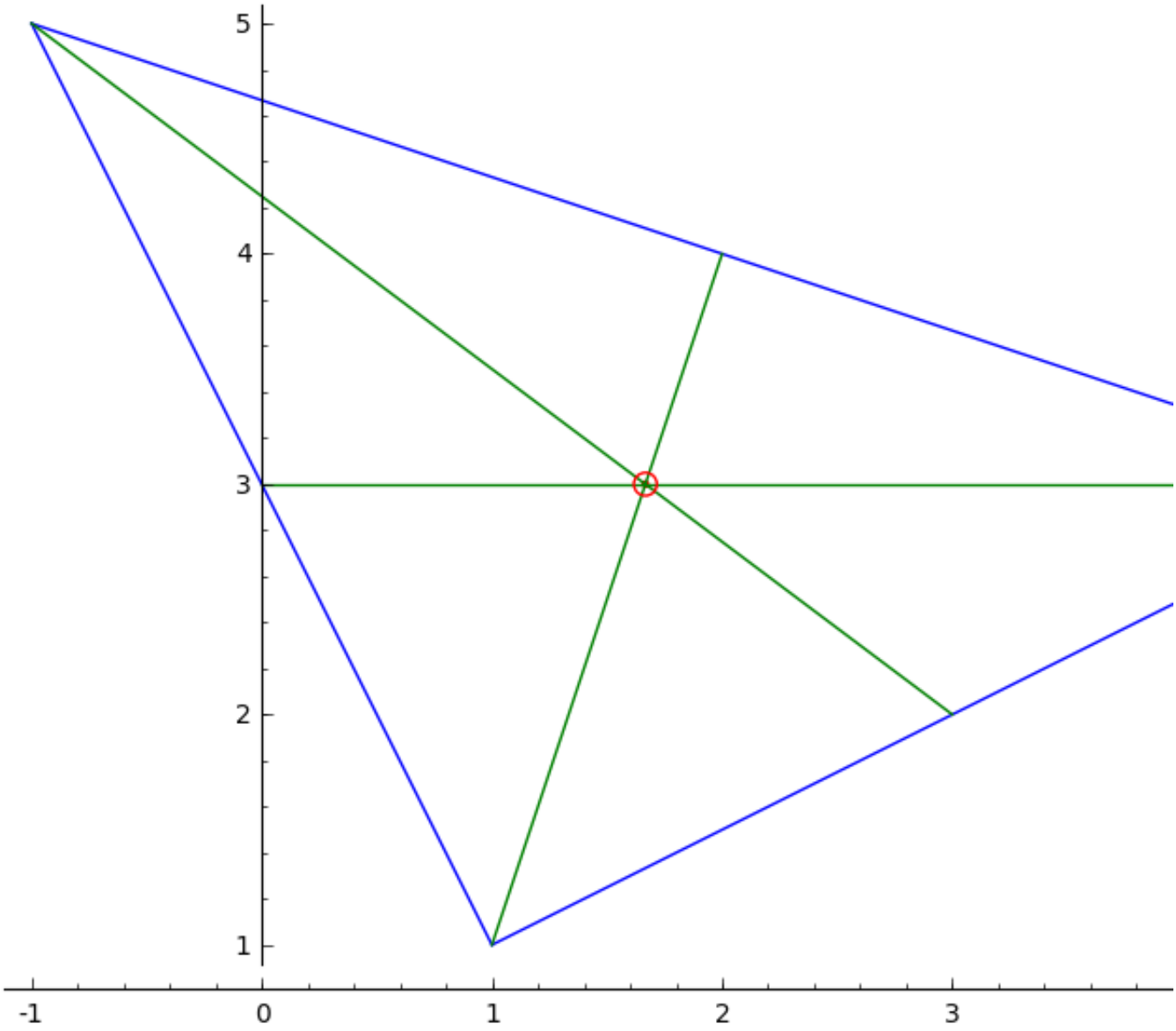
Damit können wir den Schwerpunkt einzeichnen.

```
g += point(S, color = 'red')
g
```



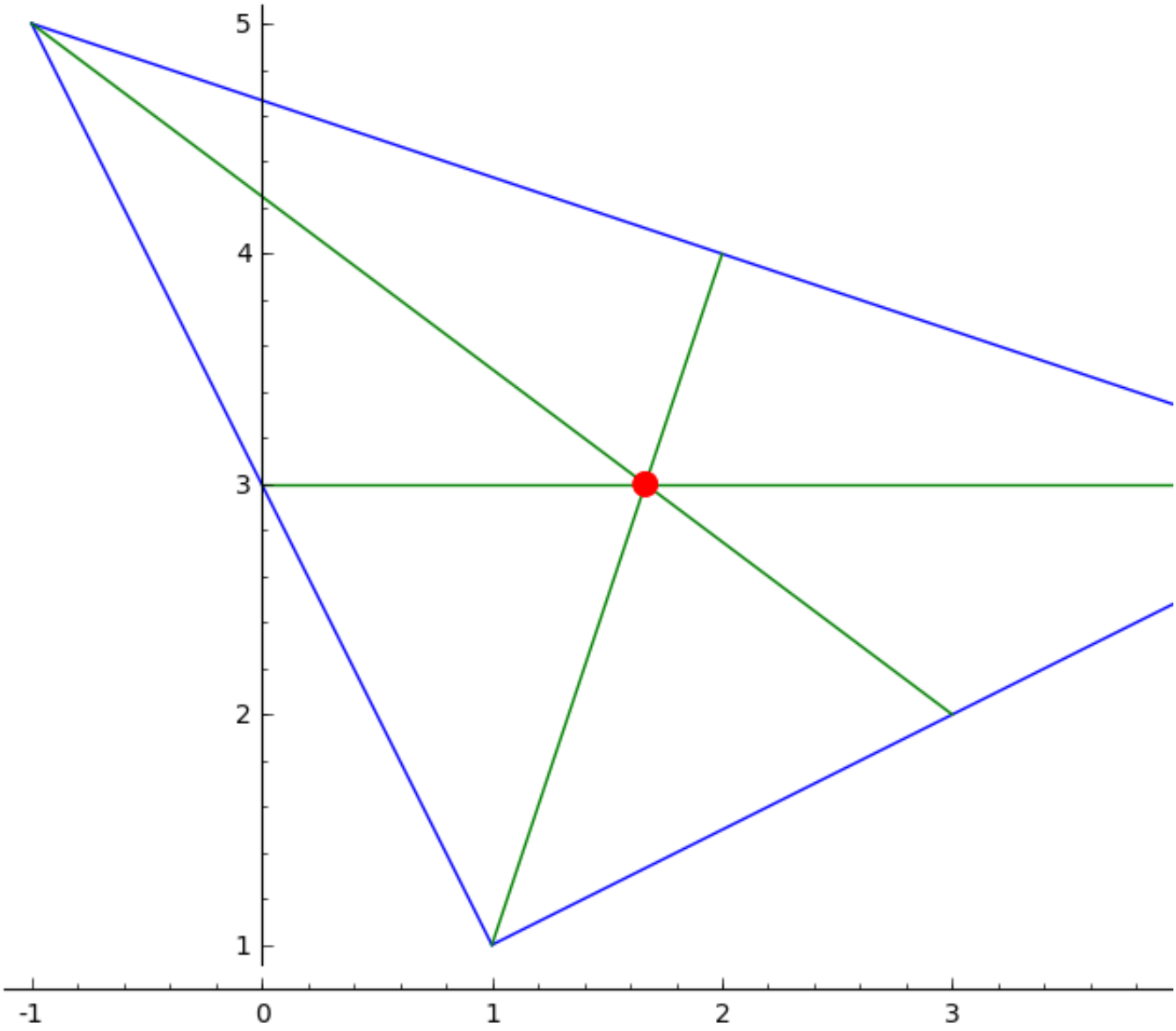
Um ihn besser sichtbar zu machen, ein Kreis.

```
g=g+ circle(S, 0.05,color='red')
g
```



Um den Kreis zu füllen:

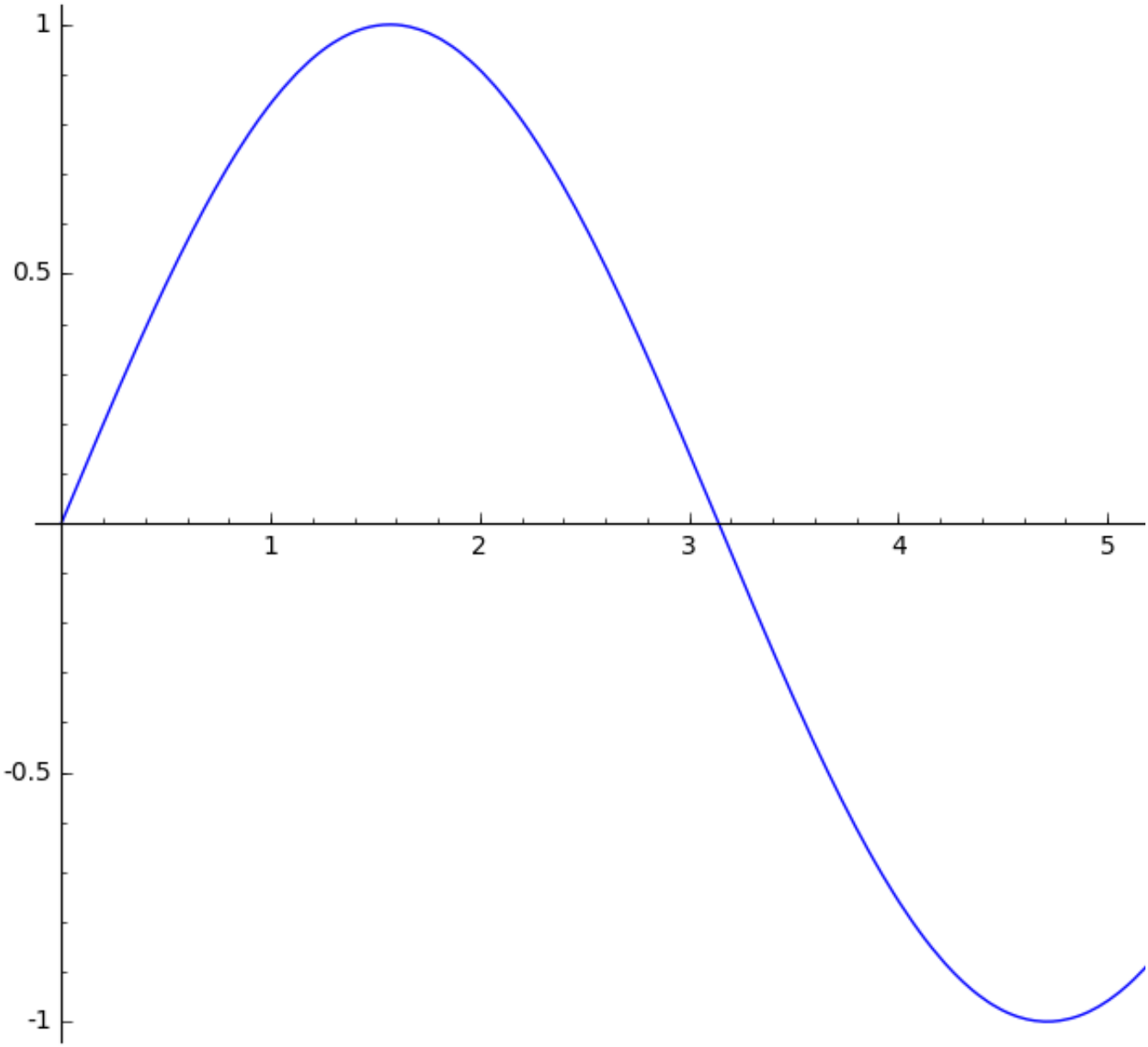
```
g=g+ circle(S, 0.05,color='red',fill='red')
g
```



Einfache Grafik

Wir haben bereits Funktionsgraphen gesehen.

```
g1 = plot(sin(x), x, 0, 2*pi)
g1.show()
```

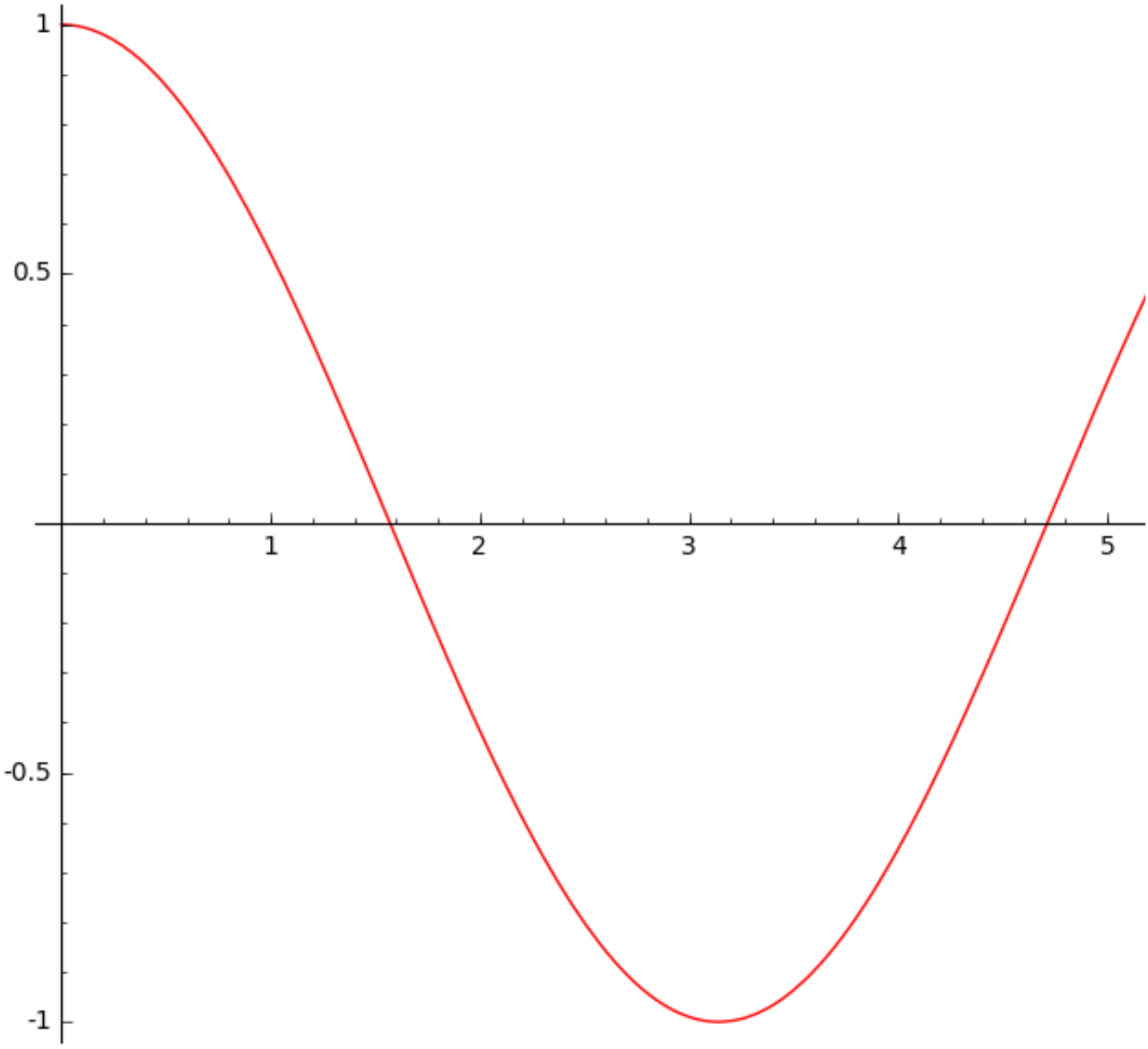


Abspeichern des Bildes (das Format wird anhand der Endung erkannt):

```
g1.save('plot.pdf')
```

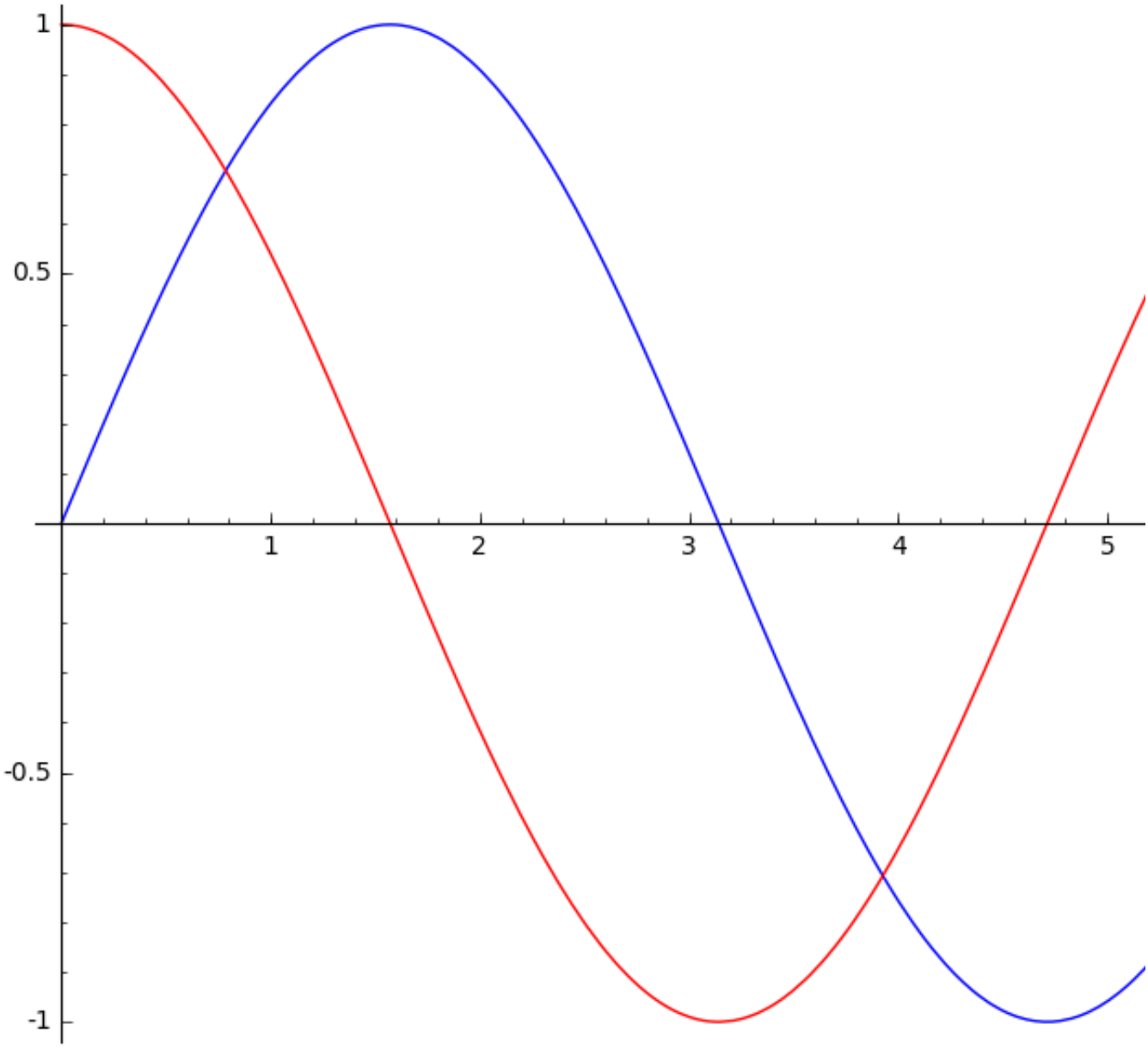
plot.pdf

```
g2 = plot(cos(x), x, 0, 2*pi, color='red')
g2.show()
```



Grafiken können beliebig kombiniert werden.

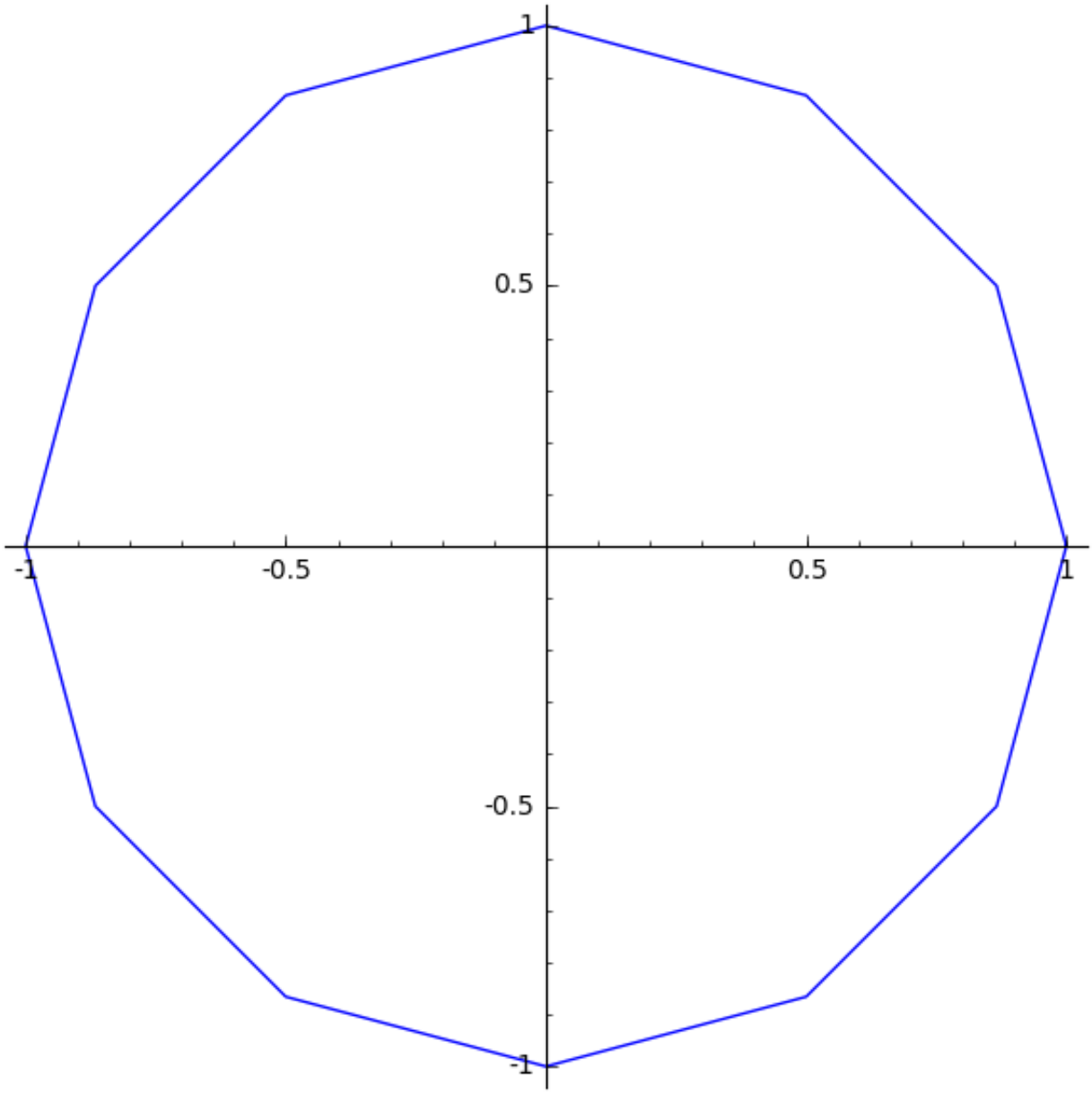
```
h = g1+g2
h.show()
```



Analog kann man mit `list_plot` auch Punktmengen und geometrische Objekte zeichnen lassen. Um ein regelmäßiges *n*-Eck zu Zeichnen, braucht man die Liste der Eckpunkte.

```
def ngon(n):
    return [[cos(2*pi*k/n), sin(2*pi*k/n)] for k in range(0,n+1)]
```

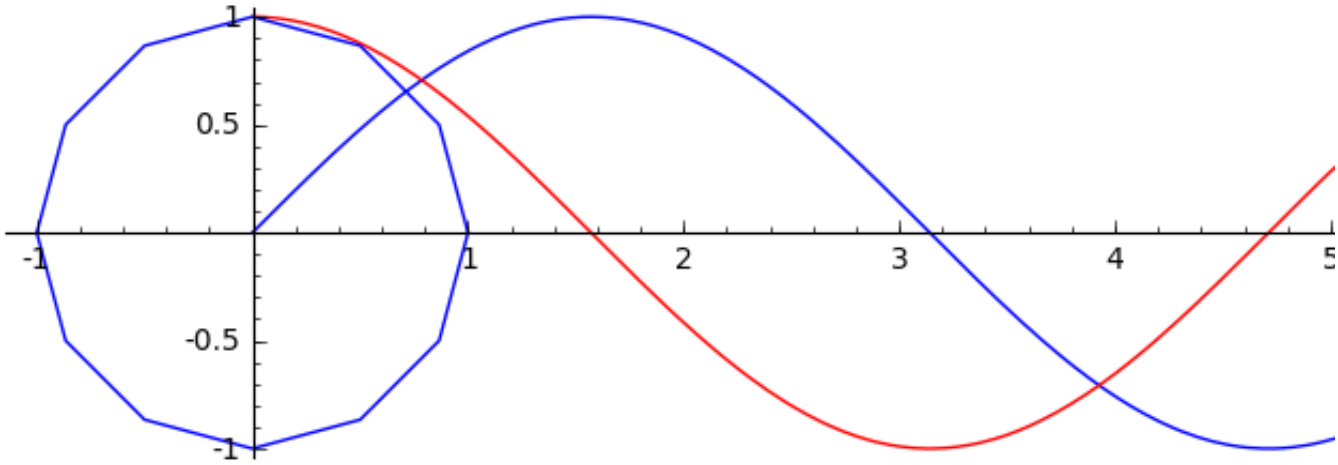
```
g12 = list_plot(ngon(12), aspect_ratio=1, plotjoined=True)
g12.show()
```



Ohne die Option `aspect_ratio` würde das Bild etwas verzerrt werden(flach gedrückt), ohne die Option `plotjoined` würden die Linien nicht gezeichnet werden, sondern nur die Eckpunkte.

Das Bild kann zu den anderen hinzugefügt werden.

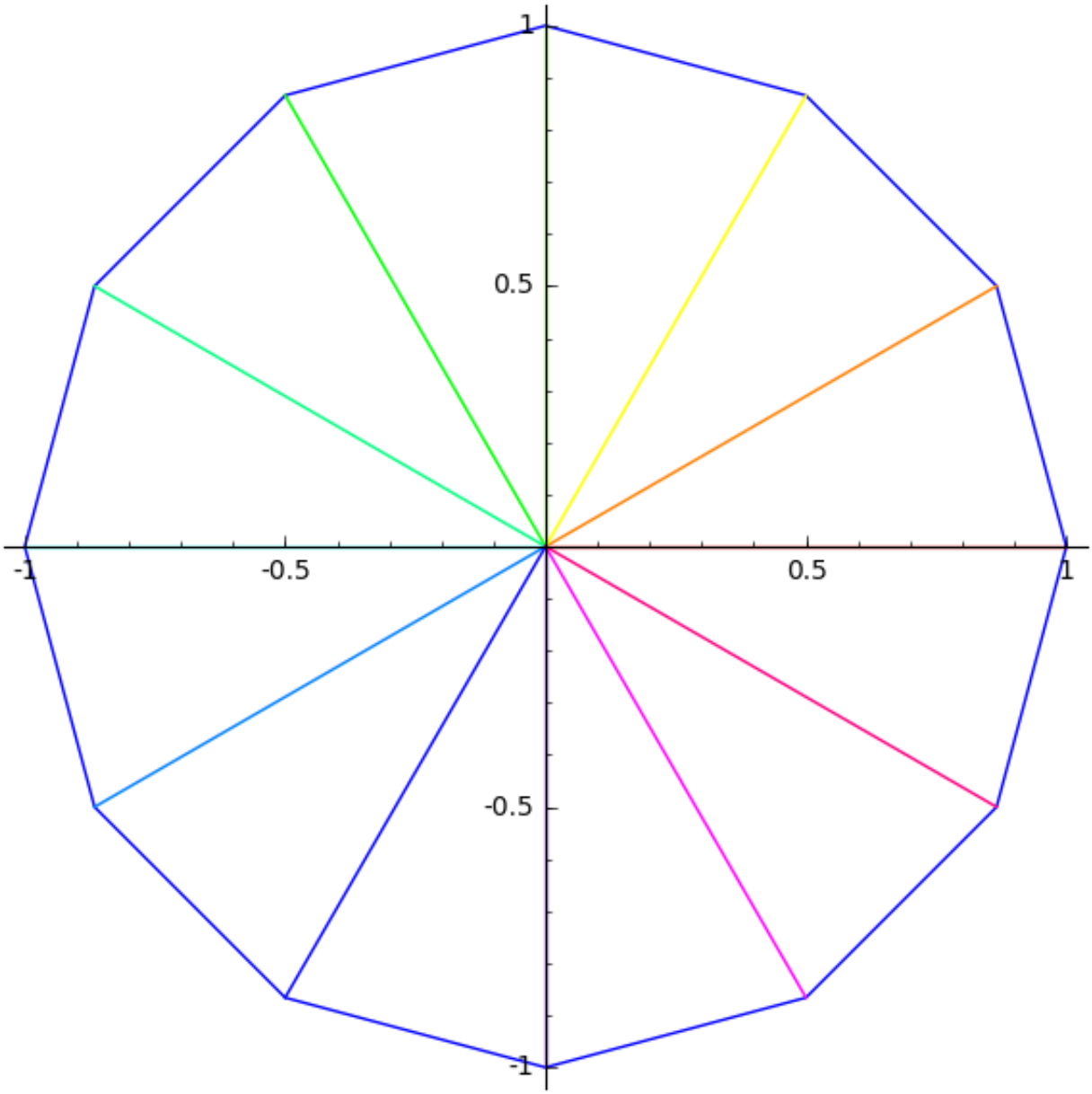
```
h += g12
h.show()
```



Um verschiedene Farben automatisch erzeugen zu lassen, empfiehlt es sich, im sogenannten HSV-Raum (hue-saturation-value) zu arbeiten, siehe unten.

```
for k in range(12):
    g12 += line([[0,0], [cos(2*pi*k/12),sin(2*pi*k/12)]], hue=k/12)
```

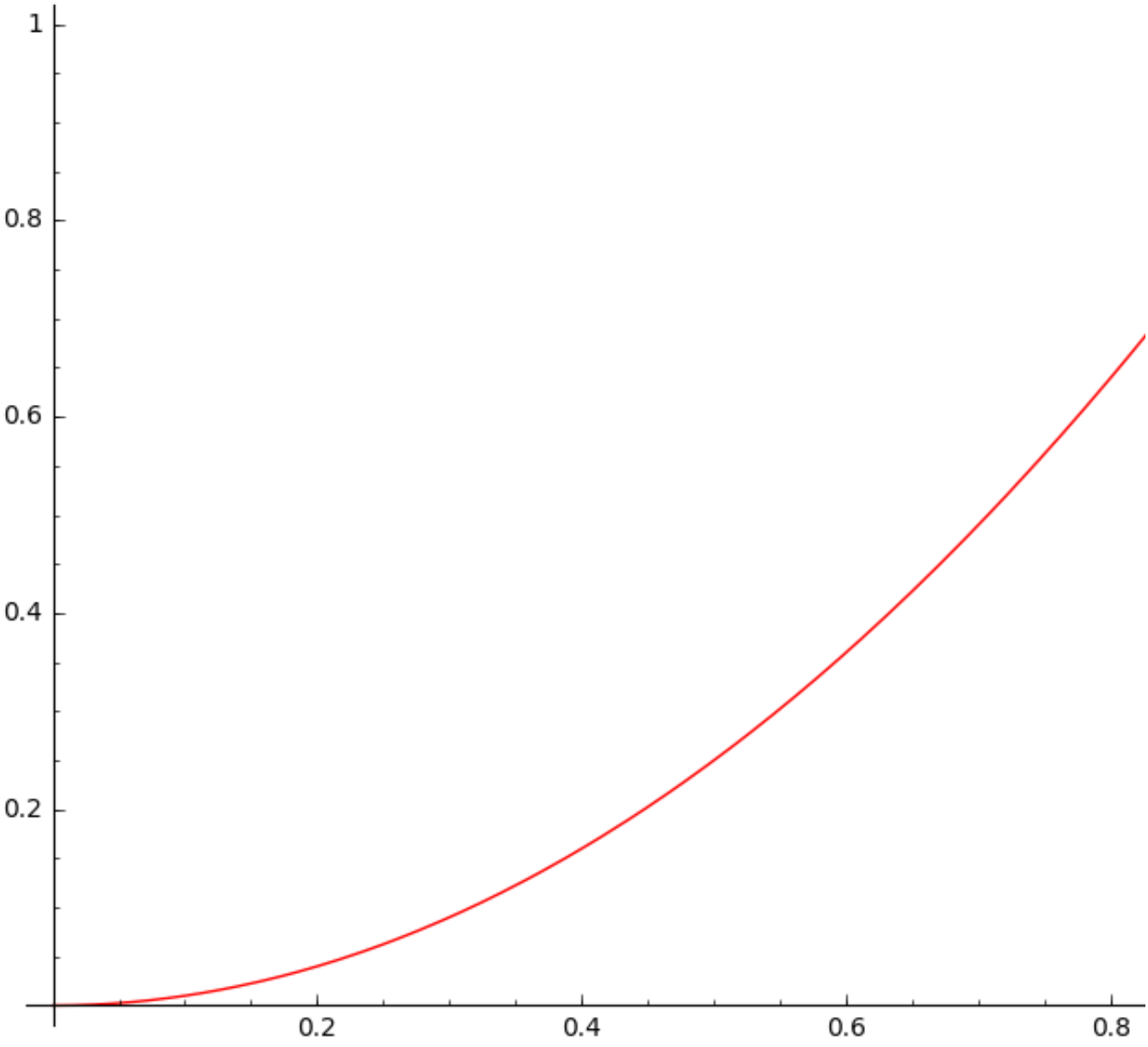
```
g12.show()
```



Grafik: Farben und HSV-Koordinaten

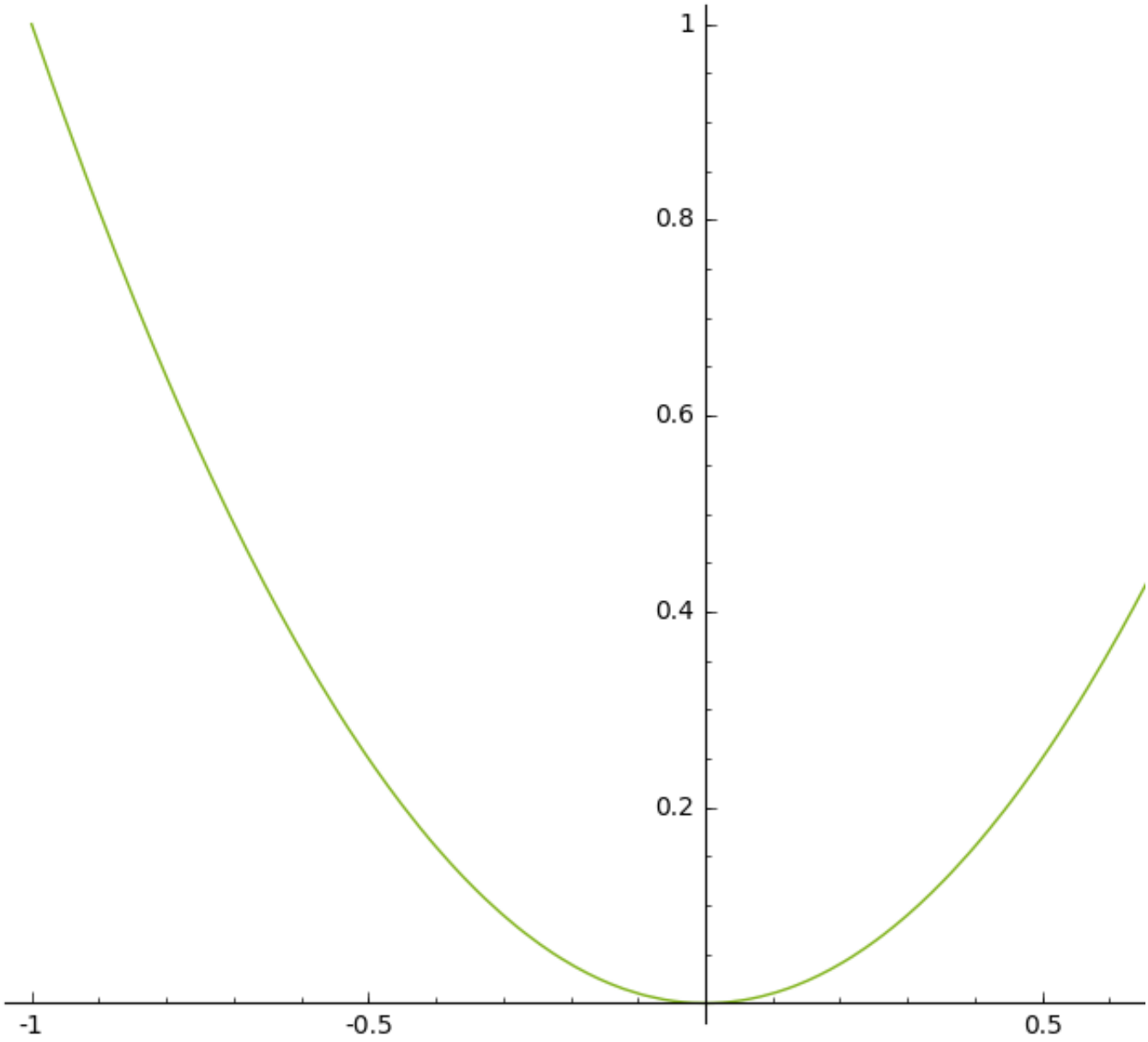
Die Farbe von Graphen kann mit der Option **color** festgelegt werden.

```
p = plot(x^2,0,1, color='red')
p
```



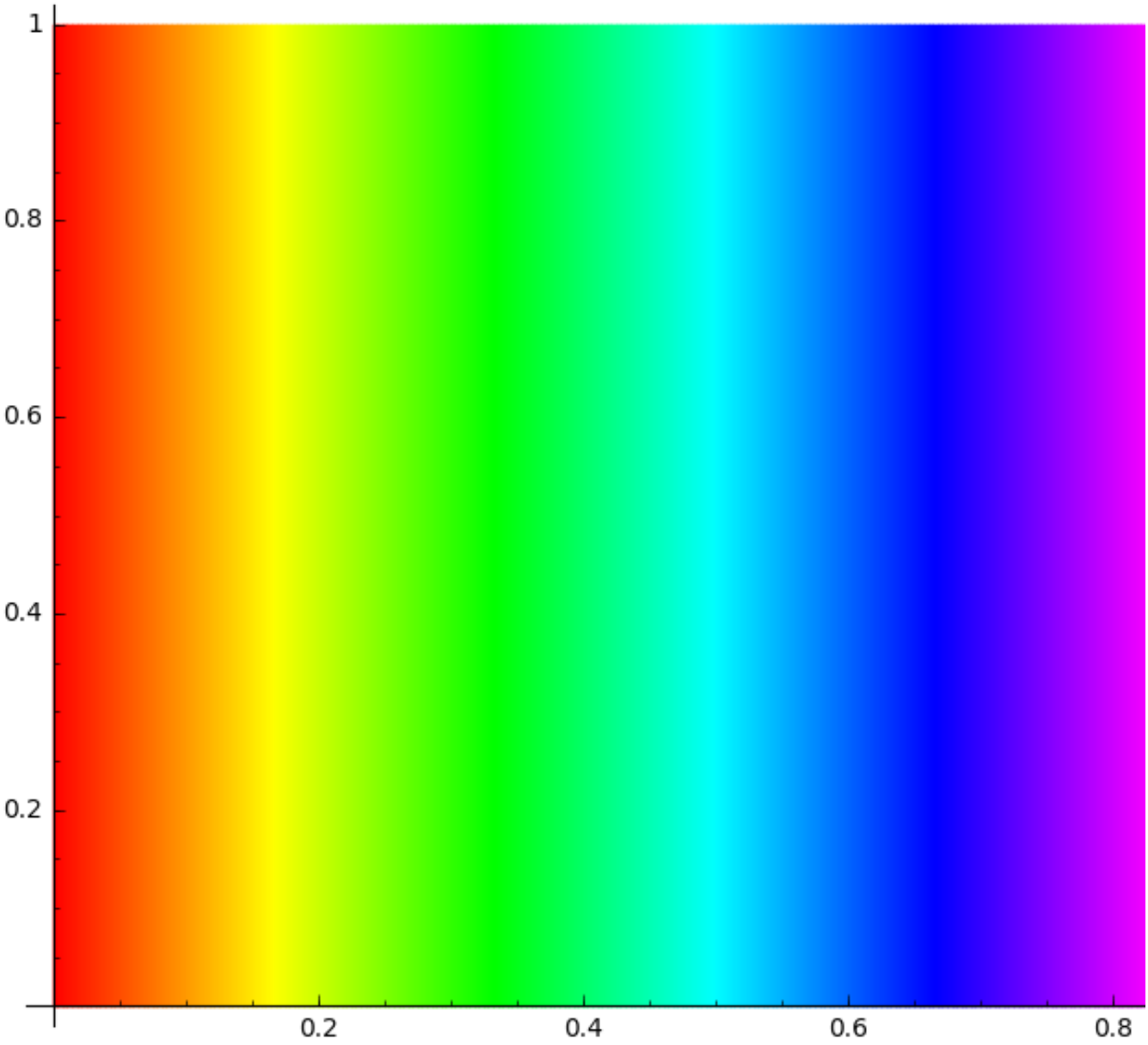
Wenn man viele verschiedene Farben benötigt, ist es besser, eine Parametrisierung des Farbenraums zu verwenden. Entweder, indem die drei Farbkanäle direkt angegeben werden,

```
plot(x^2, x, -1, 1, rgbcolor = [0.5,0.7,0.1])
```



oder intuitiver anhand anhand des Farbtons in HSV-Koordinaten (engl. *hue-saturation-value*, hue=Farbwert). Alle drei Koordinaten verlaufen in sage im Intervall [0..1]. Reine Grundfarben erhält man mit s=v=1:

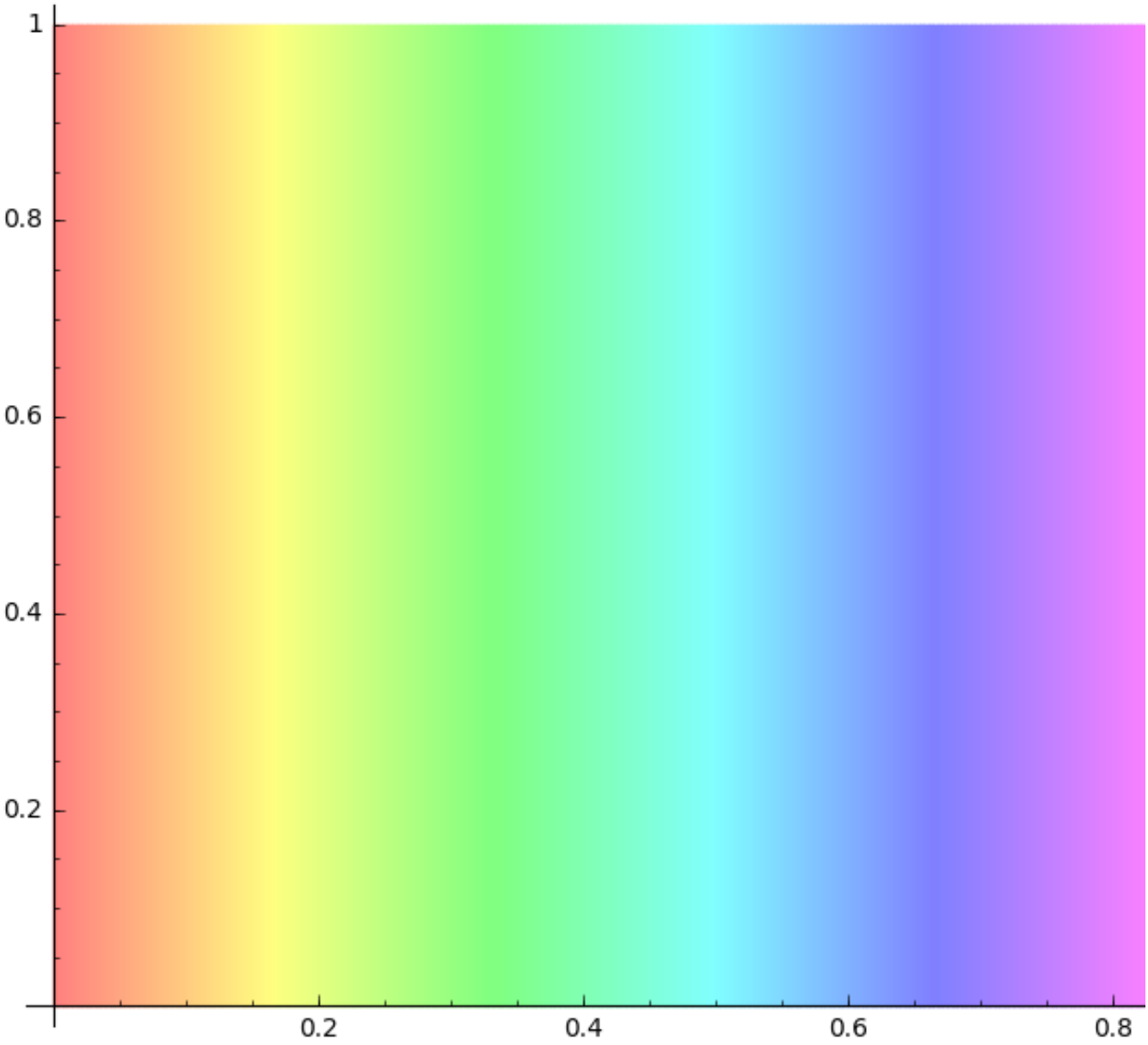
```
p = Graphics()
for x in srange(0,1,0.001):
    p = p+line([(x,0),(x,1)], hue = x)
p
```



Den Befehl `p = Graphics()` benötigt man, um die Grafik zu initialisieren für den ersten Aufruf in der For-Schleife. Hier verwendet man den Befehl `srange`, da wir zusätzlich die Schreitweiten als drittes Argument angeben.

Verringerung der Sättigung entspricht dem Zumischen von weißer Farbe:

```
p = Graphics()
for x in srange(0,1,0.001):
    p = p+line([(x,0),(x,1)], rgbcolor = hue(x,s=0.5))
p
```



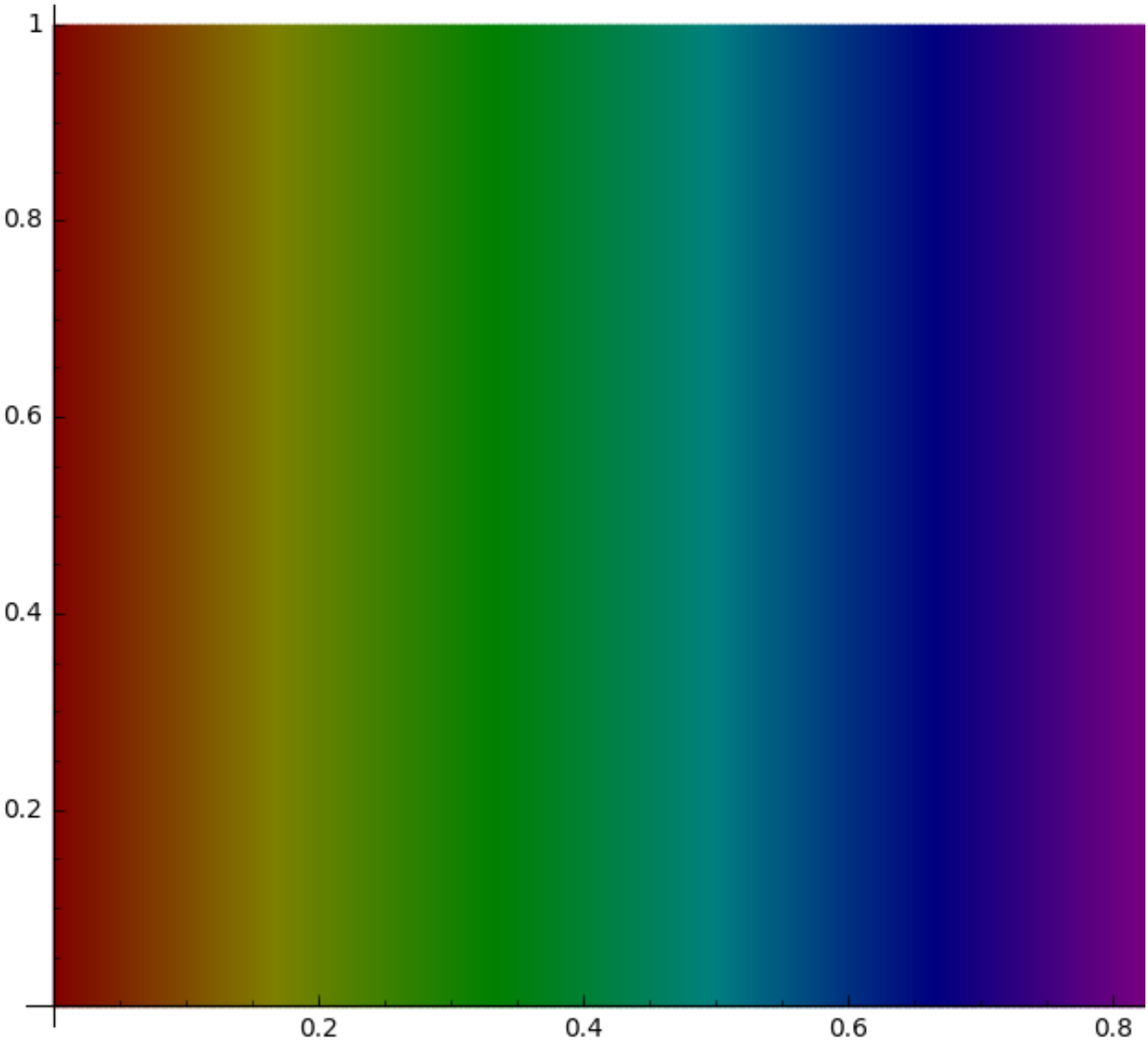
In dieser Variante wird der hue-Wert direkt in RGB-Koordinaten umgewandelt.

```
hue(0.6)
```

(0.0, 0.4, 1.0)

Der Parameter *value* variiert die Helligkeit.

```
p = Graphics()
for x in srange(0,1,0.001):
    p = p+line([(x,0),(x,1)], rgbcolor = hue(x,v=0.5))
p
```


1
0.8
0.6
0.4
0.2
0.2
0.4
0.6
0.8