# Erste Hinweise zum Einstieg in das Computeralgebra System Maxima

Jochen Ziegenbalg  Email: ziegenbalg.edu@gmail.com

## 1 Vorbemerkungen

Das Computeralgebra System Maxima ist ein facettenreiches, leistungsstarkes Software System mit folgenden Eigenschaften:

- Es ist ein voll entwickeltes *Programmiersystem* mit allem, was dazu gehört: funktionales Programmieren (einschließlich Rekursion), klassische Kontrollstrukturen mit den dazu gehörenden Datenstrukturen (Felder, bzw. engl. arrays), Modularität und last not least Listenverarbeitung (im Sinne der Programmiersprache Lisp).
- Die *Arithmetik* von Maxima ist im Gegensatz zu derjenigen von praktisch allen anderen Software Systemen (außer, natürlich, Computeralgebra Systemen) bestrebt, stets *numerisch korrekte Ergebnisse* zu liefern − näheres dazu in Abschnitt 4.
- Maxima verfügt über eine Fülle von Operationen zur *Symbolverarbeitung* mit den folgenden Anwendungsmöglichkeiten: Umformung (insbesondere Vereinfachung) algebraischer Terme, Lösen von Gleichungen mit Wurzelausdrücken (Radikalen), Grenzwertberechnungen, formales Differenzieren, Integrieren, Lösen von Differential- und Differenzengleichungen u.v.m.
- Das System bietet seit einigen Jahren mit dem (neueren) wxMaxima-Interface trotz einiger Eigenheiten eine recht gute *Benutzerschnittstelle*.

  Das sogenannte „*frontend*“, also das Editier- und Interaktions-System zum Betreiben von Maxima, ist insgesamt etwas gewöhnungsbedürftig, aber das gilt schließlich für jedes etwas komplexere System, in das man sich eingewöhnen muss – später kommt einem dann vieles ganz natürlich vor. Das Maxima-frontend ist vielleicht nicht ganz so komfortabel wie das anderer (insbesondere kommerziell vertriebener) Computeralgebra Systeme, aber man kann sehr gut damit klar kommen.
- Maxima bietet eine gute *Graphik*-Unterstützung − einschließlich einer Schnittstelle zu dem Platform-übergreifenden Graphik-System gnuplot. Gnuplot ist ein skript- bzw. kommandozeilengesteuertes Computerprogramm zur grafischen Darstellung von Messdaten und mathematischen Funktionen.

  Zitat aus *Wikipedia*: Trotz seines Namens steht Gnuplot nicht in Verbindung mit dem GNU-Projekt und verwendet auch nicht die GNU General Public License. Ursprünglich sollte das Programm *Newplot* heißen. Da unter diesem Namen bereits eine Software existierte, benannten die Autoren ihr Projekt kurzerhand in *Gnuplot* um, was sich im amerikanischen Englisch vom ursprünglichen Namen phonetisch nicht unterscheidet;

  siehe z.B.: http://de.wikipedia.org/wiki/Gnuplot oder http://www.gnuplot.info .
- Das eingebaute *Hilfe-System* von Maxima ist im Prinzip sehr umfangreich. Allerdings würden ihm mehr Beispiele für Maxima-Einsteiger gut tun. Und die Suchfunktion des Hilfe-Systems ist durchaus verbesserungswürdig. In diesem Text sind deshalb besonders auch die Dinge dargestellt, nach denen man erfahrungsgemäss sehr lange im

Hilfe-System von Maxima suchen muss − allem voran die Sache mit dem set_display(ascii) - Befehl; siehe Abschnitt 3.

- Und schließlich: Maxima ist das „dienstälteste“ Computeralgebra System. Es geht auf das am MIT entwickelte System *Macsyma* zurück, also auf das erste Computeralgebra überhaupt. Macsyma / Maxima wird schon über einen Zeitraum von mehreren Jahrzehnten entwickelt, gepflegt und gewartet; es ist mit Sicherheit keine Software-Eintagsfliege.

  *Zitat* (Quelle: http://maxima.sourceforge.net/): "Maxima is a descendant of Macsyma, the legendary computer algebra system developed in the late 1960s at the Massachusetts Institute of Technology. It is the only system based on that effort still publicly available and with an active user community, thanks to its open source nature. Macsyma was revolutionary in its day, and many later systems, such as Maple and Mathematica, were inspired by it."

- Seit einigen Jahren findet die Weiterentwicklung von Maxima im Rahmen der „Open Source“ Welt statt. Das System ist somit kostenlos beziehbar, downloads sind z.B. über das Open Source Portal von Sourceforge (http://maxima.sourceforge.net/ ) möglich, siehe auch Anschnitt 2.

  In einem sehr lebendigen Benutzerforum, werden regelmäßig design- und implementierungsspezifische Dinge in Bezug auf Maxima diskutiert − und Fragen der unterschiedlichsten Art beantwortet.

  Ein Blick in die mailing list http://maxima.sourceforge.net/maximalist.html zeigt, wie hochgradig komplex die implementierungsspezifischen Probleme bei Computeralgebra Systemen sind.

  Schließlich sei noch auf das Maxima wiki (einschliesslich FAQ’s) unter https://sourceforge.net/p/maxima/wiki/FAQ/ hingewiesen, das viele besonders für Einsteiger nützliche Informationen enthält.

Der Einstieg in hinreichend komplexe Software-Systeme ist für den Anfänger oftmals schwierig bis frustrierend. Hochgradig komplex ist Maxima zweifellos. Aber es ist auch ein ausgezeichnetes, leistungsstarkes Computeralgebra System, das dem Nutzer, der die ersten Anfangsschwierigkeiten überwunden hat, viel bietet. Diesen Einstieg in Maxima zu erleichtern, ist eines der vorrangigen Ziele dieses Textes.

Dieses Skriptum richtet sich an den gelegentlichen, anwendungsorientierten Nutzer, der Maxima z.B. für ein kleines zahlentheoretisches Problem oder eine kleine mathematische Modellierung nutzen will, der sich aber, aus was für Gründen auch immer, nicht allzu tief in systemspezifische Aspekte hineinziehen lassen kann oder will. Deshalb stehen Kriterien wie „kognitive Effizienz“ (vgl. AHG, Abschnitt 5.2) und „kognitive Stabilität“ im Vordergrund; gelegentlich (aber relativ selten) sogar zu Lasten von Laufzeiteffizienz oder anderen systemspezifischen Kriterien. Einen Schwerpunkt dieses Skriptums bilden die Programmierbeispiele in den Abschnitten 8 und 9. Ich bin ein großer Freund des Lernens anhand gut ausgewählter, typischer Beispiele. Und bei diesen Beispielen habe ich bewusst nicht immer die laufzeiteffizienteste oder die aus einer „inneren“ Sicht von Maxima heraus gesehen eleganteste Version gewählt.

Die folgenden Ausführungen basieren auf meinem mehr oder weniger systematischen Experimentieren mit dem Maxima System. Was ich beschreibe, hat bei mir geklappt; und zwar unter der Windows- und meist auch unter der Ubuntu-Linux-Implementierung von Maxima.

Ich sehe mich als interessierten Nutzer von Maxima, aber nicht als Experten. Deshalb bin mir nicht sicher, ob manches nicht auch anders oder gar besser geht. Den Herren Johann Weilharter, Volker van Nek (besonders zum Thema „Zahlen, Zahldarstellungen und Zahlenformate“), Wilhelm Haager (besonders zum Thema „Graphiken in Maxima“) und Gunter Königsmann („Ubuntu-Linux-Implementierung“) bin ich für wertvolle Hinweise und Kommentare dankbar − verbleibende Unzulänglichkeiten gehen allein auf das Konto des Autors. Auch für weitere Hinweise bin ich stets dankbar.

# 2 Download und Installation

Download und Installation verlaufen (unter Windows) ziemlich natürlich („straightforward“). Für den Download verwende ich in der Regel das Portal von Sourceforge unter der Adresse: http://maxima.sourceforge.net .

Die Installation resultiert in dem ausführbaren Programm **wxMaxima**, in dem das Kern-System von Maxima mit einem „frontend“ verbunden ist, welches das übliche Arbeiten mit „Fenstern“ möglich macht.

Die meisten der Beispiele in diesem Manuskript sind in Maxima 5.28 (wxMaxima 16.04) geschrieben. Da die Beispiele sehr elementar sind, sollten sie weitestgehend aufwärtskompatibel sein. In neueren Versionen kann das eine oder andere (systemspezifische) Detail anders ausfallen. Es lohnt sich, immer mal wieder nach updates zu schauen.

Maxima läuft inzwischen auf praktisch allen gängigen Betriebssystemen; z.B. auch auf Ubuntu-Linux. Kürzlich hat Yasuaki Honda eine Version für Android erstellt, wobei die Ausgabe mit Hilfe der MathJax Software in einem sehr schönen Formelsatz-Format erfolgt.

# 3 Start, erster interaktiver Betrieb und einige Besonderheiten

Der Start des Systems und der erste interaktive Betrieb (unter wxMaxima) verlaufen insgesamt problemlos. Mit dem Start öffnet sich ein zellen-basiertes Arbeitsblatt (worksheet, notebook). Die Ausführung von Befehlen erfolgt über die *Tastenkombination* [Shift-Return].

*Beispiele*:

*Eingabe*: `2*3 [Shift-Return]`
*Ausgabe*: `6`

*Eingabe*: `sin(1.2) [Shift-Return]`
*Ausgabe*: `0.963558185417193`

*Besonderheit* 1: Eine kleine Zusatzbemerkung (siehe Abschnitt 11.1): Die Zifferndarstellung großer natürlicher Zahlen erfolgt standardmäßig (per „default“) in einer abgekürzten Form.

*Beispiel*:

*Eingabe*: `100! [Shift-Return]`
*Ausgabe*: `933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000`

Die vollständige (nicht abgekürzte) Ausgabe erhält man nach Eingabe des Kommandos `set_display(ascii)`:

*Eingabe*: `set_display(ascii); [Shift-Return]`
*Eingabe*: `100! [Shift-Return]`
*Ausgabe*: `933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000`

*Besonderheit* 2: Zur Verwendung des Gleichheitszeichens – mehr dazu in (11.2).

*Eingabe*: `2+3 = 5  [Shift-Return]`
*Ausgabe*: `5=5`
*Eingabe*: `is(2+3 = 5)  [Shift-Return]`
*Ausgabe*: `true`

*Eingabe-Terme* werden (im interaktiven Modus) grundsätzlich mit einem Strichpunkt (Semikolon) abgeschlossen. Das letzte Semikolon in einer Eingabezeile setzt das System ggf. selbständig. Wenn die Anzeige des Outputs unterdrückt werden soll, kann dies durch ein abschließendes Dollarzeichen ($, statt des Strichpunktes) geschehen.

*Beispiele*:

Korrekte Eingabe:

```
2+3; 2*3;  [Shift-Return]   oder
2+3; 2*3   [Shift-Return]
```

Ausgabe (Display) jeweils:

```
5
6
```

Inkorrekte Eingabe:

```
2+3  2*3;   [Shift-Return]
```

Ausgabe (Display):

```
incorrect syntax …
```

Unterdrückung der Darstellung des Ausgabewerts;

*Eingabe*: `f : 20! $  [Shift-Return]`

Der Wert von `20!` wird an die Variable `f` gebunden, aber es erfolgt keine Darstellung des Werts am Ausgabegerät (Bildschirm).

*Eingabe*: `f  [Shift-Return]`
*Ausgabe*: `2432902008176640000`

Der Wert der Variablen `f` wird am Bildschirm dargestellt.

**Zur Rolle von Komma und Semikolon**

*Komma*:

- zur Abgrenzung von Parametern in Funktionen (auch in selbst geschriebenen)
- zur Abgrenzung von Anweisungen einer Anweisungsfolge in Programmen

*Semikolon*:

- zur Abgrenzung von Anweisungen einer Anweisungsfolge im interaktiven Eingabemodus (aber nicht in selbst geschriebenen Programmen)

*Bemerkung*: Warum Komma und Semikolon in der soeben beschriebenen Weise unterschiedlich gehandhabt werden, habe ich noch nicht entdecken können. Möglicherweise hängt das damit zusammen, dass bei rein funktionaler Programmierung (im Sinne von „pure Lisp“) sowieso keine Anweisungsfolgen auftreten (sollten).

Ich favorisiere einen Arbeitsstil, bei dem im interaktiven Betrieb in der Regel jeweils nur eine Anweisung bzw. ein Funktionsaufruf abgeschickt wird (hinter dem das Semikolon sowieso automatisch vom System gesetzt wird). Somit ist mir nur höchst selten eine Situation begegnet, wo ich ein Semikolon notwendigerweise hätte explizit setzen müssen. (Zugegebenermaßen mache ich jedoch, auch im interaktiven Modus, aus einer Anweisungsfolge von mehr als drei Anweisungen meist ein kleines Programm.)

**Klammern existieren in den folgenden Varianten:**

- *runde Klammern* (round brackets, parentheses):
  - zur Festlegung der Auswertungs-Priorität von arithmetischen und algebraischen Ausdrücken aller Art
  - zur Gruppierung bzw. Abgrenzung von Parametern (auch bei selbst geschriebenen Funktionen und Prozeduren),
  - zur Gruppierung von Anweisungen in Programmtexten,
- *eckige Klammern* [square brackets]: für Listen und zur Abgrenzung von Indizes für Tabellen, Felder oder Listen
- *Kommentar-Klammern*: In Maxima-Programmen werden „Klammern" der Art `/*` und `*/` (wie in einigen anderen Programmiersprachen auch) zur Kennzeichnung von Kommentaren, also als nicht auszuführende Bestandteile von Programmtexten gewertet:
  `/* Dies ist ein Kommentar. */`
  Kommentare können sich auch über mehrere Zeilen erstrecken.
- *geschweifte Klammern* {curly brackets, braces}: werden für Mengen verwendet.
- *spitze Klammern* <angle brackets>: Diese Zeichen scheinen nicht wirklich als Klammern sondern als das kleiner- bzw. größer-Zeichen benutzt zu werden – je nach Bedarf auch in der Form <= (kleiner oder gleich) bzw. >= (größer oder gleich). .

Runde ( ), eckige [ ] und geschweifte { } Klammern werden in manchen Implementierungen vom Maxima-Editier-System immer gleich paarweise gesetzt. Das reduziert die Fehleranfälligkeit gewaltig. Falls das paarweise Setzen von Klammern nicht automatisch geschieht, kann es zur Reduzierung von Fehlern ratsam sein, Klammern auch manuell immer gleich paarweise zu setzen.

*Bemerkung*: Wie viele Software-Systeme, so ist auch wxMaxima in einem gewissen Umfang an die Benutzerwünsche anpassbar („customizable"). So lässt sich z.B. die Tastenkombination [Shift-Return] auch auf [Return] umstellen.

**Im folgenden wird auf die Darstellung von [Shift-Return] zum Abschluss der jeweiligen Eingabe verzichtet.**

# 4 Bemerkungen zur Ermittlung und Darstellung der Ergebnisse in Computeralgebra Systemen

In Computeralgebra Systemen wird (ähnlich wie in Lisp-Systemen, auf denen sie oft basieren) grundsätzlich zwischen der Ermittlung von Ergebnissen und ihrer Darstellung unterschieden. In Maxima können Funktionswerte (als Listen) beliebig komplex strukturiert sein. Liegt ein Ergebnis als (u.U. strukturierte) Liste vor, so hat man die Freiheit, dieses Ergebnis in einem weiteren Verarbeitungsschritt z.B. in der Tabellenform oder als Graphik darzustellen (ggf. auch mit Animation) – oder es z.B. in Musik umzusetzen.

Wie bei Computeralgebra Systemen üblich, ist auch Maxima durchweg bemüht, die Ergebnisse in größtmöglicher Korrektheit und Präzision zu liefern. Dies hat einige Konsequenzen, an die sich der Einsteiger oft erst gewöhnen muss. Dazu gehört, dass Rechnungen soweit wie möglich rational bzw. symbolisch durchgeführt werden. Das Rechnen mit Gleitkommazahlen wird dagegen vermieden, soweit es geht, denn die Verwendung von Gleitkommazahlen führt in der Regel zu Rundungsfehlern, also zu bestenfalls näherungsweise korrekten Ergebnissen (zu den Fehlern in der Arithmetik herkömmlicher Software vgl. z.B. [AHG: Kapitel 6]).

## 4.1 Zahlen, Zahlenformate, Numerik und deren Darstellung

Siehe hierzu auch: http://maxima.sourceforge.net/docs/manual/de/maxima_5.html

### 4.1.1 Die Zahlenformate von Maxima

Maxima kennt die Zahlenformate Ganzzahl (integer), Bruch (ratnum), Geitkommazahl (floatnum) und große Gleitkommazahl (bigfloat). Für diese numerischen Datentypen stellt Maxima auch arithmetische Basis-Operationen, insbesondere die Grundrechenarten, zur Verfügung. Dabei „funktionieren" die Arithmetiken für Brüche, Gleitkommazahlen und große Gleitkommazahlen unterschiedlich – und dies führt in der Regel *auch für gleiche Terme* zu unterschiedlichen Ergebnissen, die bestenfalls näherungsweise gleich sind (man vergleiche hierzu insbesondere Abschnitt 8).

Man bedenke auch, dass sich das, was am Bildschirm zu sehen ist, durchaus von dem unterscheiden kann, was als zugehöriger Wert im Speicher steht. So haben Gleitkommazahlen z.B. intern eine Binärdarstellung. Wenn sie ausgedruckt werden, werden sie vorher „im Vorübergehen" (on the fly) in das Dezimalzahlformat überführt. Wichtig für das Weiterrechnen oder für Vergleiche u.s.w. ist jedoch nicht die dezimale Darstellung am Bildschirm sondern die internen binäre Speicherinhalte.

Hierzu ein Beispiel:

*Eingabe*: `is(2/10 + 2/10 + 2/10 = 6/10);`
Ausgabe: `true`

*Eingabe*: `is(0.2+0.2+0.2 = 0.6);`
Ausgabe: `false` (denn die Gleitkommazahlen sind nur näherungsweise gleich)

Besonders kritisch ist der Datentyp Gleitkommazahl (floatnum). Obwohl Maxima mit Gleitkommazahlen grundsätzlich in doppelter Genauigkeit rechnet, kann es bei diesem Zahlentyp zu praktisch nicht kontrollierbaren Fehlern in beliebiger Größenordnung kommen. Man vergleiche dazu auch die kleine Fallstudie in Abschnitt 4.1.2.

Die Datentypen ergeben sich in Maxima nicht aufgrund von formalen Deklarierungen sondern durch die Art der Eingabe und durch den Gebrauch. Durch Abfragefunktionen, die man oft daran erkennt, dass ihnen der Buchstabe p (für property) angehängt ist, lässt sich der jeweilige Datentyp im Zweifelsfall genau ermitteln. Dies sind: numberp, integerp, ratnump, floatnump, bigfloatp (und weitere, wie z.B. evenp, oddp, primep …). Bei Bedarf kann sich der Benutzer auch eigene „Erkennungs"-Funktionen schreiben.

*Beispiele*:
*Eingabe*: `numberp(8)`
*Ausgabe*: `true`

In der folgenden Tabelle ist ein kleiner Überblick gegeben. Dabei sei

*Eingabe*: `a : float(sqrt(5))`
*Ausgabe*: `2.23606797749979`
*Eingabe*: `b : bfloat(sqrt(5))`
*Ausgabe*: `2.23606797749979b0`
*Eingabe*: `fpprec : 50`
*Ausgabe*: `50`
*Eingabe*: `c : bfloat(sqrt(5));`
*Ausgabe*: `2.2360679774997896964091736687312762354406183596115b0`

| | “sechs“ | 6 | 6.0 | 3 / 4 | 1.75 | sqrt(5) | a | b | c |
|---|---|---|---|---|---|---|---|---|---|
| numberp | false | true | true | true | true | false | true | true | true |
| integerp | false | true | false | false | false | false | false | false | false |
| ratnump | false | true | false | true | false | false | false | false | false |
| floatnump | false | false | true | false | true | false | true | false | false |
| bfloatp | false | false | false | false | false | false | false | true | true |

Maxima kennt keinen eigenen Typ für komplexe Zahlen. Komplexe Zahlen werden von Maxima intern als die Addition von Realteil und dem mit der Imaginären Einheit `%i` multiplizierten Imaginärteil dargestellt.

*Eingabe*: `%i^2`
*Ausgabe*: `-1`

### 4.1.2 Die Problematik der Gleitkommazahlen

Verlässlich korrekte Ergebnisse sind nur mit den Datentypen integer und rational zu erzielen. Die Genauigkeit von Berechnungen, mit `bigfloat`-Zahlen ist über die Variable `fpprec` zu steuern.

Man könnte meinen, dass Gleitkommazahlen wegen der Existenz von großen Gleitkommazahlen überflüssig seien. Es gilt jedoch: Die Existenzberechtigung von Gleitkommazahlen basiert auf der größeren Laufzeit-Effizienz beim Rechnen mit diesem Zahlenformat. Für viele laufzeitintensive Berechnungen (wie z.B. im Ingenieurwesen) reicht die Genauigkeit von Gleitkomma-Ergebnissen in der Regel aus. Das Problem ist jedoch, dass man sich bei Rechnungen mit Gleitkommazahlen des Ergebnisses nie wirklich sicher sein kann. Derartige Ergebnisse sind somit kritisch zu bewerten. Man sollte wissen, worauf man sich dabei einlässt.

Man vergleiche dazu die folgende kleine **Fallstudie**.

*Eingabe*:
```
xi  : 665857;
xf  : 665857.0;
xbf : bfloat(665857.0);
```
*Ausgabe*:
```
665857
665857.0
6.65857b5
```
*Eingabe*:
```
yi : 470832;
yf : 470832.0;
ybf : bfloat(470832.0);
```
*Ausgabe*:
```
470832
470832.0
4.70832b5
```
Die Funktion `f(x, y)` sei wie folgt definiert:
```
f(x, y) :=  x*x*x*x - 4*y*y*y*y - 4*y*y;
```

Die Auswertung liefert nun die folgenden Werte:

```
f(xi, yi)   = 1                      (dies ist das korrekte Ergebnis)
f(xf, yf)   =  1.1885568 * 10^7      !!!
f(xbf, ybf)  =  1.0  b0
```

*Also*: Vorsicht beim Rechnen mit Gleitkommazahlen! Das Tückische daran ist, dass man nie weiß, ob man dem Ergebnis trauen kann oder nicht. In kritischen Fällen (s.o.) leuchtet keine rote Lampe zur Warnung auf.

### 4.1.3 Die Unterscheidung von Gleitkommazahlen (floatnums) und großen Gleitkommazahlen (bigfloats) - besonders bei der Ausgabe

Hierzu in folgenden einige typische Beispiele:

Darstellung von Gleitkommazahlen (floatnums):

*Eingabe*: `2345678.9`
*Ausgabe*: `2345678.9`
*Eingabe*: `23456789.1`
*Ausgabe*: `234567891 10^7`

Darstellung von großen Gleitkommazahlen (bigfloats):

*Eingabe*: `bfloat(2345678.9)`
*Ausgabe*: `2.3456789b6`
*Eingabe*: `bfloat(23456789.1)`
*Ausgabe*: `2.34567891b7`

(Einfache) Gleitkommazahlen sind in Maxima intern als Binärzahlen nach dem Prinzip Mantisse / Exponent realisiert, und zwar grundsätzlich in der „double precision“ Variante. Alle diese Binärzahlen beanspruchen denselben Speicherplatz (64 bit). Diese Form der Darstellung wirkt sich vor allem positiv auf die Laufzeiteffizienz aus, aber negativ auf die Korrektheit und Verlässlichkeit der Ergebnisse.

Die großen Gleitkommazahlen (bigfloats) und ihre Arithmetik in Maxima sind so implementiert sind, dass auf ihnen basierende Rechenergebnisse im Bereich der Grundrechenarten mit einer durch die Variable `fpprec` vorgegebenen Genauigkeit korrekt sind. Dies geht allerdings nur auf Kosten eines deutlich höheren „Verwaltungsaufwands“ für das Rechnen mit den bigfloats.

### 4.1.4 Die Konversion von Gleitkommazahlen

Kommazahlen (wie in den angelsächsischen Ländern üblich, werden „Kommazahlen“ mit einem Dezimalpunkt an Stelle eines Dezimalkommas geschrieben), die wie im folgenden direkt eingegeben werden, werden in Gleitkommazahlen (floatnums) konvertiert:

*Eingabe*: `3.14`
*Ausgabe*: `3.14`

Man beachte:

*Eingabe*: `is(float(3.14) = 3.14)` bzw. `floatnump(3.14)`
*Ausgabe*: `true`
*Eingabe*: `is(bfloat(3.14) = 3.14)` bzw. `bfloatp(3.14)`
*Ausgabe*: `false`
*Eingabe*: `is(314/100 = 3.14)`
*Ausgabe*: `false`

Manchmal hat man es mit Termen zu tun, die gemischt aus Gleitkommazahlen (floatnums) und großen Gleitkommazahlen (bigfloats) bestehen.

Maxima versucht, den bigfloat-Status des Ergebnisses zu retten und konvertiert die floatnums (automatisch) in bigfloats. Dies geschieht bei manueller Eingabe der Kommazahlen durch Auffüllen des Nachkommateils mit Nullen, und zwar mit so vielen, wie durch fpprec erforderlich ist.

Dies ist eine besonders kritische Operation, da das Ergebnis eine Präzision signalisiert, die durch die Qualität der Eingabedaten u.U. nicht gestützt wird. Wenn z.B. der Wert 2.5 durch eine Messung zustande gekommen ist, dann liegt der wahre Wert irgendwo zwischen 2.45 und 2.55. Und ein Ergebnis, das in Abhängigkeit von `fpprec` auf 100 oder 1000 Stellen angezeigt wird, würde unter diesen Umständen eine völlig überzogene Genauigkeit suggerieren.

Dennoch kann es ja sein, dass der Wert 2.5, wie auch immer er zustande gekommen ist, der exakte Wert ist. In dem Fall würde nichts dagegen sprechen, ihn wie eine bigfloat-Zahl zu behandeln. Und dies macht Maxima durch das Auffüllen der Nachkommastellen mit Nullen.

Aber das ist ein kritischer Moment. Deshalb stellt Maxima die Variable `float2bf` bereit, mit der sich die Ausgabe von Warnungen kontrollieren lässt. Der Standardwert (default value) vom `float2bf` ist `true`. Bei diesem Wert erfolgt keine Warnung bei der Konversion von floatnums in bigfloats. (D.h.: Diese Konversionen werden als „unkritisch“ gewertet.) Ist dagegen der Wert von `float2bf false`, so wird bei diesen Konversionen eine Warnung ausgegeben.

Hierzu einige Beispiele:

*Eingabe*:
```
float2bf : true
2.5 * bfloat(3.75)
```
*Ausgabe*: `9.375b0`

*Eingabe*:
```
float2bf : false
2.5 * bfloat(3.75)
```
Warnungen:
bfloat: converting float 3.75 to bigfloat.
bfloat: converting float 2.5 to bigfloat.

*Ausgabe*: `9.375b0`

Bei algorithmisch mit beliebiger Genauigkeit berechenbaren Zahlen (wie z.B. bei der Quadratwurzel von 5 oder der Kreiszahl Pi) entfällt die Warnung. Die entsprechende Zahl wird einfach bis zu der (durch den Wert von `fpprec`) erforderlichen Genauigkeit berechnet.

*Beispiele*:

*Eingabe*: `1.75 * bfloat(sqrt(5))`
bfloat: converting float 1.75 to bigfloat.
*Ausgabe*: `3.913118960624632b0`

*Eingabe*: `2.4 * bfloat(%pi)`
bfloat: converting float 2.4 to bigfloat.
*Ausgabe*: `7.539822368615504b0`

Da „Warnung“ besser ist als „keine Warnung“ wird übrigens von einigen Maxima-Experten die Auffassung vertreten, dass der Standardwert der Variablen `float2bf` auf `false` gesetzt werden sollte.

### 4.1.5 Weitere Aspekte des Bemühens von Maxima um maximale Korrektheit

Da die Gleitkommarechnung Ergebnisse von minderer Qualität produziert, wird sie in Computeralgebra Systemen so weit wie möglich zugunsten der Bruchdarstellung mit Zähler und Nenner vermieden.

*Beispiel*:
*Eingabe*: `1234 / 12`
*Ausgabe*:

```
617
---
 6
```

Maximale numerische Präzision ist nur bei der Verwendung von Ganzen Zahlen und Brüchen aus ganzen Zahlen gewährleistet. Bei der Verwendung von Gleitkommazahlen erhält man grundsätzlich nur Näherungswerte – man vergleiche dazu auch die Auswertungsbeispiele zum Programm `heron` weiter unten.

Enthält ein algebraischer Ausdruck irgendwann einmal eine Gleitkommazahl, so ist er „korrumpiert“ und alle weiteren Rechnungen, die auf diesem Ausdruck basieren, werden als (numerisch möglicherweise nicht völlig korrekte) Gleitkommaoperationen durchgeführt.

*Beispiel*: Ausgabe als Bruch ist möglich – und wird deshalb auch so realisiert:
*Eingabe*: `2 * 3 * 4 / 15 ;`
*Ausgabe*:

```
8
-
5
```

*Beispiel*: Ausgabe als Bruch ist (wegen der Gleitkommazahl 4.0) nicht möglich; deshalb wird das Ergebnis „nur“ als Gleitkommazahl ausgegeben:
*Eingabe*: `2 * 3 * 4.0 / 15;`
*Ausgabe*: `1.6` (Anzeige auf dem Computer-Bildschirm)

Der am Bildschirm angezeigte Wert `1.6` ist nur scheinbar korrekt; im Speicher des Computers steht im Falle von Gleitkommazahlen sehr oft ein Wert, der sich geringfügig von dem unterscheidet, was am Bildschirm zu sehen ist. Denn Gleitkommazahlen werden in der Regel binär gespeichert. So hat z.B. der Bruch `8/5` die nicht-abbrechende periodische Binärdarstellung `1.10011001100110011001100110011…` und die muss für die Speicherung im Computer irgendwo „gekappt“ werden. Bei der Rückwandlung in die Dezimaldarstellung wird diese Kappung cachiert, so dass der vermeintlich korrekte Wert 1.6 angezeigt wird. Im Speicher befindet sich aber der (inkorrekte) gekappte binäre Wert, der dann für etwaige weitere Berechnungen verwendet wird. Bei numerisch kritischen Rechnungen kann dies zu sich lawinenartig vergrößernden Fehlern führen („Schmetterlingseffekt“).

Wenn man also z.B. einmal manuell den Wert 3.24 eingeben möchte und wenn man sich sicher ist, dass dies der korrekte Wert ist und wenn mit maximaler Korrektheit weitergerechnet werden soll, dann sollte man statt 3.24 den Term 324 / 100 eingeben.

Mehr zum Thema „Korrektheit von Computerergebnissen z.B. in AHG, Kapitel 6.

## 4.2 Symbolisches Rechnen

Die für Computeralgebra Systeme typische Fähigkeit, symbolische Rechnungen durchzuführen ermöglicht Termumformungen, wie sie z.B. in der Algebra üblich sind.
Ein Beispiel:

*Eingabe*: `expand((a+b)^2);`
*Ausgabe*: `a^2 + 3 a b^2  + 3 a^2 b + b^2`

Wenn möglich (auch dies ist eine Konsequenz des Bestrebens nach größtmöglicher Genauigkeit der Ergebnisse), wird in Computeralgebra Systemen mit symbolischen Werten gerechnet.

Ein *Beispiel*: Eine der berühmtesten Formeln der Mathematik, welche Zahl Eins (1), die Basis des natürlichen Wachstums (%e), die Kreiszahl (%pi), die komplexe Einheit (%i) und die Zahl Null (0) miteinander verknüpft, ist: $e^{\pi \cdot i} + 1 = 0$. In Maxima:

*Eingabe*: `%e^(%pi*%i)+1;`
*Ausgabe*: `0`

Das symbolische Rechnen ist ein eigenständiges Thema, das eine eigenständige Behandlung (an anderer Stelle) verdient.

## 4.3 Tippfehler, Fehlermeldungen und die Verwendung undefinierter Funktionen

Wie bei jeder Computeranwendung, vertippt man sich auch manchmal in Maxima. Nehmen wir an, wir hätten eine Funktion `hugo(x)` definiert, wollen `hugo(5)` aufrufen, vertippen uns aber und schreiben `higo(5)` - wobei eine Funktion `higo(x)` nicht definiert sei.

Dann sieht der Maxima-Dialog folgendermaßen aus:

*Eingabe*: `higo(5);`
*Ausgabe*: `higo(5)`

Dies ist (wenn auch vielleicht zunächst nicht besonders hilfreich, so zumindest doch) auf jeden Fall korrekt, denn natürlich ist stets `higo(5) = higo(5)`. Der Ausdruck `higo(5)` ist für Maxima einfach ein neues Symbol, das auf sich selbst verweist, d.h., dessen Wert das Symbol selbst ist. (Ähnliches würde z.B. bei der Eingabe des undefinierten Symbols `abrakadabra` passieren.)

Wäre z.B. stattdessen (wie in vielen anderen Programmiersystemen) bei der Eingabe `higo(5)` eine Fehlermeldung der Art „unbekannte Funktion“ ausgedruckt worden, so wäre möglicherweise eine offene (und später noch abzuschließende) Auswertungskette unterbrochen worden. Dies wäre im Hinblick auf die Belange der Zielsetzungen von Computeralgebra Systemen (Symbolverarbeitung) meist unangenehmer als das (vermeintliche) Nicht-Aufdecken eines Tippfehlers.

# 5 Die Dokumentation und das Hilfe-System

Die Dokumentation und das Hilfe-System (man vergleiche dazu das entsprechende drop-down-Menue) scheint mehr auf den harten Kern des Maxima-Systems und nicht so sehr auf die Benutzeroberfläche ausgerichtet zu sein. Man vergleiche dazu die Bemerkungen zum Befehl `set_display(ascii)` oder zur Suche gewisser Kontrollstrukturen – wie z.B. „while“ (vgl. Bemerkungen in Punkt 8).

Nach einer Weile findet man meist, was man sucht; man muss aber hartnäckig „dran“ bleiben. Jedoch: Alles, was ich im Hinblick auf die Beschreibung des `printf`-Befehls zum formatierten Ausdrucken gefunden habe, ist außerordentlich lückenhaft. (Einige rudimentäre Beispiele sind in 9.3 und 9.4 gegeben.) Der `printf`-Befehl scheint zu einer relativ großen Software-Gruppe (Fortran, COBOL, C, Lisp, PHP, Python, Java, ...) zu gehören und dort beschrieben zu sein. Bei „externen“ Beschreibungen, wie z.B. der unter der Adresse

http://en.wikipedia.org/wiki/Printf oder http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html stellt sich allerdings immer die Frage, ob und in wie weit sie jeweils auch auf Maxima zutreffen.

# 6 Speichern und Exportieren

Auswahl im drop-down-Menue unter „Datei“. Das Standard-Format für wxMaxima Arbeitsblätter hat die Dateikennung wxm (Datei ohne output-Zellen) oder wxmx (Datei mit output-Zellen).

Alle Speicherungs- bzw. Export-Formate:

## 6.1 Speichern / Speichern als:

- wxMaxima Dokument: Dateikennung: wxm
- wxMaxima XML Dokument: Dateikennung: wxmx
  Das sind komprimierte XML-Dokumente. Ich habe mir angewöhnt, Maxima Dateien in diesem Format abzuspeichern. Es eignet sich auch vergleichsweise gut für die Präsentation von Maxima-Dateien im Internet
- Maxima Batch-Datei: Dateikennung: mac

## 6.2 Exportieren / Drucken

- HTML-Datei: Dateikennung: html
- pdfLaTeX Datei (vgl. Datei-Menue): Dateikennung: tex
  (Die interaktive Konvertierung von Ausdrücken im frontend ist möglich mit dem Befehl tex (vgl. Hilfe-System)).

Natürlich kann man Maxima-Programme (i.w. als ASCII-Text) z.B. für Präsentationszwecke stets auch per "cut-and-paste" in geeignete andere Dateien kopieren.

Mehr zum Thema „Speichern und Exportieren“: siehe z.B. auch das Tutorial:
http://www.austromath.at/daten/maxima/index.htm, Kapitel "Grundlagen / Speichern"

Und schliesslich: wxMaxima Dateien lassen sich natürlich auch ausdrucken; auch im pdf-Format.

# 7 Öffnen und Einlesen von Dateien

Dazu sind im "Datei"-Menue einige Möglichkeiten (selbsterklärend) vorgesehen.

*Bemerkung*: Notfalls kann man Programme auch mit Hilfe von "cut-and-paste" aus Text-Dateien heraus in den Maxima-Editor kopieren.

# 8 Programmieren

Funktionales Programmieren, Rekursion und Listenverarbeitung werden voll unterstützt (wie soll es bei einem Lisp-Abkömmling auch anders sein).

Das Thema *Kontrollstrukturen* (und insbesondere Kontrollstrukturen für die Iteration) ist im Hilfe-System etwas verborgen: Man findet es unter dem Stichwort "Program Flow". Ausführungen zu den Kontrollstrukturen „`while`“ und „`for`“ findet man recht gut versteckt unter den Ausführungen zu der sehr variantenreichen Kontrollstruktur „`do`“.

Einige einfache, aber typische Verwendungsformen der Kontrollstrukturen von Maxima (man vergleiche dazu auch die unten gegebenen Beispiele):

```
while  (Bedingung)  do  (Anweisung)
for  i : 1  thru  10  do  (Anweisung)
```

Nicht zu vergessen: Rekursive Aufrufe von Funktionen (oder Prozeduren) stellen auch eine Form der Wiederholung dar.

Ein wichtiges Hilfsmittel zur Modularisierung ist die *„Block"-Strukturierung* (man vergleiche dazu die Beispiele weiter unten). Der erste Parameter eines Blocks ist (optional) eine Liste mit lokalen Variablen (wie alle Listen in eckigen Klammern geschrieben). Dabei kann ggf. auch gleich die Zuweisung von Anfangswerten erfolgen. Blöcke werden bei der Abarbeitung in der Regel verlassen, wenn der letzte Befehl eines Blockes abgearbeitet ist. Dabei ist ggf. vorliegenden Iterationsstrukturen (einschließlich rekursiver Aufrufe) Rechnung zu tragen. Der Funktionswert eines Blocks ist in der Regel gleich dem Funktionswert des letzten Funktionsaufrufs innerhalb des Blocks.

Mit Hilfe des Befehls `return(XYZ)` wird der gerade bearbeitete Block (u.U. vorzeitig) verlassen und zwar mit dem Wert der Variablen `XYZ` als Funktionswert. An Stelle der Variablen `XYZ` kann ggf. auch ein ad-hoc Term wie z.B. `(2+3)*(4+5)` zurückgegeben werden.

Die folgenden Programme sind (sparsam) mit Kommentaren versehen. In Maxima-Programmen werden „Klammern" der Art /* und */ (wie in einigen anderen Programmiersprachen auch) als *Kommentare*, also als nicht auszuführende Bestandteile von Programmtexten gewertet: `/*  Dies ist ein Kommentar.  */`

Noch ein Hinweis: Bei den Programmen kam es mir nicht in erster Linie auf Laufzeiteffizienz oder Knappheit der Formulierung sondern auf Transparenz und auf die Natürlichkeit der Darstellung (auch unter dem Aspekt der Portabilität zu anderen Betriebssystemen, Plattformen, Computeralgebra Systemen oder Programmiersprachen). Deshalb habe ich gewisse Redundanzen absichtlich in Kauf genommen.

## 8.1 Ein erstes Beispiel zum Programmieren: Das Heron-Verfahren

Der Programmtext des extrem einfachen Programms:

```
heron(a) :=
 block([x],    /* Definition der lokalen Variablen  x  */
  x : a,       /* Wertzuweisung */
  while abs(a - x*x) > 0.000001 do x : (x+a/x)/2,
  x )          /* Funktionswert-Rueckgabe */ ;
```

Der letzte „Funktionsaufruf" innerhalb des Blocks ist in diesem Fall der Trivialaufruf `x` und der Wert der Variablen `x` wird dementsprechend als Funktionswert des Blocks `block()` und damit auch als Funktionswert des Aufrufs `heron(a)` zurückgegeben. Ohne den Trivialaufruf `x` wäre `do x : (x+a/x)/2` der letzte Aufruf im Block gewesen, und der hätte den Wert *done* (done ist ein Symbol, das auf sich selbst verweist). An Stelle des letzten Aufrufs `x` hätte man auch `return(x)` verwenden können, aber eine der Hauptfunktionen von `return()`, nämlich das vorzeitige Verlassen von Blöcken, wäre dabei überhaupt nicht zum Tragen gekommen.

## 8.2 Die Vorgehensweise im Detail

- wxMaxima starten.
- Das obige Programm manuell eingeben oder mit Hilfe von cut-and-paste in eine Zelle einfügen.
- Das Programm (mit [Shift-Return]) übersetzen.
- Das Programm (am besten in einer neuen Zelle) ausführen; ein Beispiel:
  *Eingabe*: `heron(5);`
  *Ausgabe* (als Bruch, wegen maximaler Genauigkeit):
  ```
  4870847
  -------
  2178309
  ```
  Zum Erzwingen der Ausgabe als Dezimalzahl:
  *Eingabe*: `heron(5.0);`
  (Die Eingabe `5.0` wird als Gleitkommazahl erkannt, und damit werden alle weiteren Rechnungen als Gleitkommaoperationen ausgeführt.)
  *Ausgabe*: `2.236067977499978`
  Auch mit dem Operator `float` kann man die Ausgabe als Dezimalzahl auch explizit erzwingen:

*Eingabe*: `float(heron(5));`
*Ausgabe*: `2.236067977499978`

## 8.3 Einige Fallstudien zum Aufruf von heron

Mit den großen Gleitkommazahlen (bigfloats) lässt sich die Genauigkeit (im Vergleich zu den gewöhnlichen floatnum-Gleitkommazahlen) erheblich steigern. Die Genauigkeit der Ergebnisse wird durch die Variable `fpprec` gesteuert:

*Eingabe*:
```
fpprec : 100 $
set_display(ascii) $
```
*Eingabe*:
```
bfloat(heron(5))
```
*Ausgabe*:
```
2.23606797749997819409459355858145010648167913734920068732213841103351 2\
6926436974735907531943356061976b0
```
*Eingabe*:
```
heron(bfloat(5.0))
```
*Ausgabe*:
```
2.23606797749997819409459355858145010648167913734920068732213841103351 2\
6926436974735907531943356061976b0
```
*Eingabe*:
```
bfloat(4870847/2178309)
```
*Ausgabe*:
```
2.23606797749997819409459355858145010648167913734920068732213841103351 2\
6926436974735907531943356061976b0
```
Man beachte aber:

*Eingabe*: `bfloat(heron(5.0))` (Ein möglicher, aber nicht sehr sinnvoller Aufruf, siehe nachfolgende Erläuterung.)

*Ausgabe*: `2.23606797749997809887645416893064975738525390625b0`

Der Wert `heron(5)` wurde mit der Arithmetik für Brüche (ratnums), der Wert `heron(bfloat(5.0))` wurde mit der Arithmetik für große Gleitkommazahlen (bigfloats), und der Wert `heron(5.0)` wurde mit der Arithmetik für Gleitkommazahlen (floatnums) be-

rechnet. Dies sind unterschiedliche arithmetische Prozesse und es ist nicht verwunderlich, dass sich deren Ergebnisse unterscheiden. Nachdem im letzten Beispiel `heron(5.0)` auf der Basis der gewöhnlichen Gleitkomma-Arithmetik berechnet wurde, kann die Genauigkeit durch eine simple Konversion von floatnum in bigfloat auch nicht mehr gesteigert werden.

Man vergleiche dazu auch den Aufruf:

*Eingabe*: `is(heron(bfloat(5.0)) = bfloat(heron(5.0)))`

*Ausgabe*: `false`

Auch hier wurden unterschiedliche Arithmetiken (die für bigfloats und die für floatnums) verwendet und es ist nur natürlich, dass sich diese Werte unterscheiden (wenn auch nur geringfügig).

Sobald die while-Schleife im Programm heron auch nur einmal durchlaufen wird, kann `heron` nur näherungsweise korrekte Ergebnisse liefern.

## 8.4 Anonyme Funktionen

In funktionalen Programmiersprachen heißt „ein Programm zu schreiben“ meist „eine Funktion zu programmieren“ (zum Begriff der Funktion in Programmiersprachen siehe z.B. AHG Abschnitt 8.6). Auch das obige Programm `heron(x)` ist eine Funktion, denn es liefert einen Funktionswert, der beliebig weiterverarbeitet werden kann. Manchmal benötigt man derartige Funktionen nur kurz „im Vorübergehen“ (on the fly) und möchte darauf verzichten, einen eigenen Funktionsnamen für die Funktion zu vergeben. Dazu dient in den Lisp-artigen Programmiersprachen (und somit auch in Maxima) das lambda-Konstrukt.

Ein *Beispiel*: Die anonyme Funktion `lambda([x], 2*x)` verdoppelt alle ihre Eingabewerte. Der Aufruf dieser Funktion an der Stelle 3, `lambda([x], 2*x)(3)`, hat den Funktionswert 6. Dies funktioniert auch mit symbolischen statt numerischen Eingabewerten:

Eingabe: `lambda([x], 2*x)(s)`
Ausgabe: `2 s`

Ein lambda-Aufruf mit zwei Parametern:

*Eingabe*: `lambda([x,y], x^2+y^2) (3, 4);`
*Ausgabe*: `25`

Im Falle des heron-Programms sähe die anonyme Version, ausgewertet an der Stelle `5.0` folgendermaßen aus:

```
lambda([a],
  block([x],
   x : a,
   while abs(a - x*x) > 0.000001 do x : (x+a/x)/2,
   x ) )  (5.0)  ;
```

Funktionen, die über lambda-Ausdrücke definiert sind, können, wie anderen Funktionen auch, miteinander verknüpft werden; ein Beispiel:

```
lambda([x],x^2)(lambda([y], 5*y)(a));
25 a^2
lambda([x],7*x)(lambda([y], y^2)(b));
7 b^2
lambda([x],11*x)(lambda([y], y^2)(c^3));
11 c^6
```

Auch im Graphik-Bereich werden lambda-Ausdrücke gern verwendet, um Funktionsgraphen zu erzeugen (vgl. Abschnitt 10).

Das Arbeiten mit durch lambda-Ausdrücke definierte Funktionen ist fundamental für die Lisp-artigen Programmiersprachen und gibt Anlass zu erheblichen Vertiefungen. Man ver-

gleiche das Hilfe-System von Maxima (Stichwort: lambda) bzw. die weiterführende Literatur.

# 9 Weitere Beispiele

Die Vorgehensweise beim Editieren und Ausführen der Programme entspricht der im Abschnitt 8.2.

## 9.1 Einige Versionen des Euklidischen Algorithmus

Für weitere (insbesondere inhaltliche) Erläuterungen zu den folgenden Programmen wird z.B. verwiesen auf [Ziegenbalg 2010].

```
Euklid_Subtraktionsform(a, b) :=
 block([x : a, y : b],
  while x*y # 0 do    /* solange x und y von Null verschieden sind */
   (if x>y then x : x-y else y : y-x),
  return(x+y) ) ;
```

An Stelle des letzten Aufrufs `return(x+y)` könnte man hier auch einfach `x+y` schreiben. In beiden Fällen wird der Wert von `x+y` als Funktionswert zurückgegeben. (Durch die explizite Bemühung des return-Befehls soll nur noch mal die Funktionswertrückgabe verdeutlicht werden.)

```
euclid_verbose(a, b) :=
 block([x : a, y : b],
  while x*y # 0 do
   (if x>y then x : x-y else y : y-x,
    print(x, " ", y) ),
  x+y ) ;

euclid_recursive(a, b) :=
 if  a*b = 0  then a+b
     elseif a>b then euclid_recursive(a-b, b)
     else euclid_recursive(a, b-a) ;

euclid_rec_mod(a, b) :=
 if  a*b = 0 then a+b
     elseif a>b then euclid_rec_mod(mod(a, b), b)
     else euclid_rec_mod(a, mod(b, a)) ;
```

## 9.2 Das Sieb des Eratosthenes

Das Sieb des Eratosthenes ist einer der bedeutendsten Algorithmen der griechischen Antike. Seine Realisierung als Maxima-Programm wurde hier aufgenommen, weil anhand dieses Beispiels Themen wie Felder (arrays), Listen und ihre Verarbeitung recht gut dargestellt werden können.

In der folgenden Version wird mit dem Datentyp „Feld“ (array) gearbeitet. Zu Ausgabezwecken wird das Feld zum Schluss in eine Liste konvertiert.

```
Eratosthenes(UpperLimit) :=
  block([A, i, k],
    A : make_array(fixnum, UpperLimit+1),
    for i : 0 thru UpperLimit do (A[i] : i),
    A[1] : 0,
    i : 2,
    while i*i <= UpperLimit do
     (k : i+i,
      while k <= UpperLimit do
```

```
          (A[k] : 0,
           k : k+i),
        i : i+1),
      delete(0, listarray(A)) ) ;
```

Ein Aufrufbeispiel:

```
Eratosthenes(100);
[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]
```

In der folgenden Version wird nur mit Listen gearbeitet. Dadurch kann der mit Nebenwirkungen verbundene (und bei manchen Verwendungen deshalb kritische) Befehl „delete“ vermieden werden.

```
Eratosthenes_List(UpperLimit) :=
   /* for avoiding the destructive delete-command */
 block([E, i, k],
   E : makelist(j, j, 1, UpperLimit),
   E[1] : 0,
   i : 2,
   while i*i <= UpperLimit do
    (k : i+i,
     while k <= UpperLimit do
       (E[k] : 0,
        k : k+i),
     i : i+1),
   sublist(E, lambda([x], not(is(x=0)))) ) ;
```

Ein Aufrufbeispiel:

```
Eratosthenes_List(150);
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,
 97,101,103,107,109,113,127,131,137,139,149]
```

In der Regel sind Programme, die auf der Felverarbeitung basieren, laufzeiteffizienter als Programme, welche die Listenverarbeitung bemühen.

Der Aufruf `Eratosthenes_List(50000)` benötigt für seine Abarbeitung etwa 80 Sekunden, während `Eratosthenes(50000)` auf demselben Computersystem etwa eine Sekunde benötigt.

## 9.3 Die Kreiszahl Pi nach Archimedes
### (mit „großen“ Gleitkommazahlen und tabellarischem Ausdruck)

Zur Erzeugung übersichtlicher, tabellarischer Ausdrucke eignet sich in Maxima der `printf`-Befehl, von dem im folgenden Gebrauch gemacht wird (`printf` steht dabei für `print formatted`). Leider habe ich noch keine gute Dokumentation dieses Befehls gefunden. Deshalb kann ich hier nur auf die folgenden sehr einfachen Beispiele verweisen. Der volle `printf`-Befehl scheint aber wesentlich leistungsfähiger zu sein als das, was in den Beispielen zu sehen ist.

```
Pi_Archimedes_Wolff_computation_as_bigfloat(s) :=
  /* in this version all computation is done in bigfloat mode */
  block([r:1, se, su, ue, uu, i, n:3],
    se : bfloat(sqrt(3)),  /* initial values             */
    ue : bfloat(3 * se),   /* for the "triangle"-polygon */
    su : bfloat(2 * sqrt(3)),
    uu : bfloat(3 * su),
    printf(true, "~2d ~10d ~13,10h ~13,10h ~43,40h ~%",
                     0, n, ue/2, uu/2, se*se),
```

```
for i : 1 step 1 thru s do
  (n  : bfloat(n * 2),
   se : bfloat(r*sqrt(2-2*sqrt(1-(se/(2*r))*(se/(2*r))))),
   ue : bfloat(n * se),
   su : bfloat(se / sqrt(1 - (se/(2*r)) * (se/(2*r)))),
   uu : bfloat(n * su),
   printf(true, "~2d ~10d ~13,10h ~13,10h ~43,40h ~%",
                  i, n, ue/2, uu/2, se*se) ),
   /* in printf:  ~2d :  drucke Ganzzahl, 2 Stellen, rechtsbündig
         ~13,10h :  drucke Kommazahl (bigfloat number) rechtsbuendig,
                    13 Stellen insgesamt, davon 10 Nachkommastellen */
bfloat(0.5*(0.5*ue+0.5*uu)) ) ;
```

Ein Aufrufbeispiel:

```
Pi_Archimedes_Wolff_computation_as_bigfloat(30);
```

```
 0          3  2.5980762114  5.1961524227  3.000000000000000000000000000000000000000
 1          6  3.0000000000  3.4641016151  1.000000000000000000000000000000000000000
 2         12  3.1058285412  3.2153903092  0.267949192431122700000000000000000000000
 3         24  3.1326286133  3.1596599421  0.068148347421863430000000000000000000000
 4         48  3.1393502030  3.1460862151  0.017110277252379180000000000000000000000
 5         96  3.1410319509  3.1427145996  0.004282153522792986000000000000000000000
 6        192  3.1414524723  3.1418730500  0.001070825047268714000000000000000000000
 7        384  3.1415576079  3.1416627471  0.000267724180876449700000000000000000000
 8        768  3.1415838921  3.1416101766  0.000066932165197819420000000000000000000
 9       1536  3.1415904632  3.1415970343  0.000016733111298711250000000000000000000
10       3072  3.1415921060  3.1415937488  0.000004183282199643479000000000000000000
11       6144  3.1415925167  3.1415929274  0.000001045820823331045000000000000000000
12      12288  3.1415926193  3.1415927220  0.000000261455222916318100000000000000000
13      24576  3.1415926453  3.1415926710  0.000000065363806811546960000000000000000
14      49152  3.1415926507  3.1415926571  0.000000016340951758397890000000000000000
15      98304  3.1415926453  3.1415926469  0.000000004085237925721685000000000000000
16     196608  3.1415926453  3.1415926457  0.000000001021309481430421000000000000000
17     393216  3.1415926453  3.1415926454  0.000000000255327370357605300000000000000
18     786432  3.1415923038  3.1415923038  0.000000000063831828711613530000000000000
19    1572864  3.1415923038  3.1415923038  0.000000000015957957177903380000000000000
20    3145728  3.1415868397  3.1415868397  0.000000000003989475416688038000000000000
21    6291456  3.1415868397  3.1415868397  0.000000000000997368854172009400000000000
22   12582912  3.1416742650  3.1416742650  0.000000000000249356091330810200000000000
23   25165824  3.1416742650  3.1416742650  0.000000000000062339022832702540000000000
24   50331648  3.1430727402  3.1430727402  0.000000000000015598633495983450000000000
25  100663296  3.1374750995  3.1374750995  0.000000000000003885780586188048000000000
26  201326592  3.1374750995  3.1374750995  0.000000000000000971445146547012000000000
27  402653184  3.1819805153  3.1819805153  0.000000000000000249800180540660200000000
28  805306368  3.0000000000  3.0000000000  0.000000000000000055511151231257830000000
29 1610612736  0.0000000000  0.0000000000  0.000000000000000000000000000000000000000
30 3221225472  0.0000000000  0.0000000000  0.000000000000000000000000000000000000000
```

```
Ergebnis:  0.0b0
```

Ein weiteres Aufrufbeispiel mit erhöhter Präzision:

```
fpprec : 100;
set_display(ascii);
Pi_Archimedes_Wolff_computation_as_bigfloat(30);
```

```
 0          3  2.5980762114  5.1961524227  3.000000000000000000000000000000000000000
 1          6  3.0000000000  3.4641016151  1.000000000000000000000000000000000000000
 2         12  3.1058285412  3.2153903092  0.267949192431122706472555365849412763305 72
 3         24  3.1326286133  3.1596599421  0.068148347421863426500513600542205264732 2
 4         48  3.1393502030  3.1460862151  0.017110277252379177710884946142874257444 5
 5         96  3.1410319509  3.1427145996  0.004282153522792986523860417454444790936 3
 6        192  3.1414524723  3.1418730500  0.001070825047268711140327107514280108232 7
 7        384  3.1415576079  3.1416627471  0.000267724180876434274505701879799956093 1
 8        768  3.1415838921  3.1416101766  0.000066932165197793084792378841721775259 9
 9       1536  3.1415904632  3.1415970343  0.000016733111298701704882758862330583587 4
10       3072  3.1415921060  3.1415937488  0.000004183282199637916672551346299059760 3
11       6144  3.1415925167  3.1415929274  0.000001045820823344777803525071971844408 9
12      12288  3.1415926194  3.1415927220  0.000000261455222925902849689656765306044 3
13      24576  3.1415926450  3.1415926707  0.000000065363806799582522255701784015750 5
14      49152  3.1415926515  3.1415926579  0.000000016340951766652306723939732049312 3
15      98304  3.1415926531  3.1415926547  0.000000004085237945835368949508229234055 4
16     196608  3.1415926535  3.1415926539  0.000000001021309486719610504292925874773 1
17     393216  3.1415926536  3.1415926537  0.000000000255327371696200642757553918966 4
18     786432  3.1415926536  3.1415926536  0.000000000063831842925068786732191143272 4
19    1572864  3.1415926536  3.1415926536  0.000000000015957960731330860810723460263 6
20    3145728  3.1415926536  3.1415926536  0.000000000003989490182836694210660602655 9
21    6291456  3.1415926536  3.1415926536  0.000000000000997372545709422240663884387 4
22   12582912  3.1415926536  3.1415926536  0.000000000000249343136427371103165891956 5
23   25165824  3.1415926536  3.1415926536  0.000000000000062335784106843747228968042 9
24   50331648  3.1415926536  3.1415926536  0.000000000000015583946026710997522085451 6
25  100663296  3.1415926536  3.1415926536  0.000000000000003895986506677753175199077 9
26  201326592  3.1415926536  3.1415926536  0.000000000000000973996626669438530967126 7
27  402653184  3.1415926536  3.1415926536  0.000000000000000243499156667359647564741 5
28  805306368  3.1415926536  3.1415926536  0.000000000000000060874789166839912817620 4
29 1610612736  3.1415926536  3.1415926536  0.000000000000000015218697291709978262307 3
30 3221225472  3.1415926536  3.1415926536  0.000000000000000003804674322927494569195 7
```

*Ergebnis*:
```
3.14159265358979323871165873540889325073781729847488891360714657974006257920
022763542873596519258917 87b0
```
Auch dieses Ergebnis stimmt natürlich nicht mit dem Wert von π überein, denn es wurde ja „nur“ der Umfang von Polygonen ermittelt.

## 9.4 Freies Wachstum (mit tabellarischem Ausdruck)

```
Freies_Wachstum_mit_Tabellen_Ausdruck(B0, gr, sr, n) :=
 block([B:B0, G:0, S:0, L:[[0,0,0,B0]] ],
  print( " i        G            S            B"),
  for i:1 thru n do
   (G : gr*B,    S : sr*B,    B : B+G-S,
    printf(true, "~2d ~13,4f ~13,4f ~13,4f ~%",
                  i, G, S, B)
    /* in printf:  ~2d :  drucke Ganzzahl, 2 Stellen, rechtsbündig
                ~13,4f :  drucke Kommazahl (floating point number)
                          rechtsbuendig,
                 13 Stellen insgesamt, davon 4 Nachkommastellen */  ) ) ;
```

Ein Aufruf-Beispiel:

```
Freies_Wachstum_mit_Tabellen_Ausdruck(1,2,1,15);
i        G            S            B
1        2.0000       1.0000       2.0000
2        4.0000       2.0000       4.0000
3        8.0000       4.0000       8.0000
4       16.0000       8.0000      16.0000
5       32.0000      16.0000      32.0000
6       64.0000      32.0000      64.0000
7      128.0000      64.0000     128.0000
8      256.0000     128.0000     256.0000
9      512.0000     256.0000     512.0000
10    1024.0000     512.0000    1024.0000
11    2048.0000    1024.0000    2048.0000
12    4096.0000    2048.0000    4096.0000
13    8192.0000    4096.0000    8192.0000
14   16384.0000    8192.0000   16384.0000
15   32768.0000   16384.0000   32768.0000
```

# 10 Maxima und Graphik

„Maxima und Graphik“ ist eigentlich ein eigenständiges Thema. Im folgenden sind nur einige rudimentäre Varianten zur Erstellung von einfachen Graphiken aufgezeigt. Für eine grundlegende Darstellung des Themas „Graphiken in Maxima“ sei auf das online manual
http://maxima.sourceforge.net/docs/manual/de/maxima_12.html#SEC81
sowie auf das folgende ausführliche Manuskript von Wilhelm Haager verwiesen:
http://www.austromath.at/daten/maxima/zusatz/Grafiken_mit_Maxima.pdf

Zunächst einmal ist zu unterscheiden zwischen den „plot“-Befehlen plot2d, plot3d und den „draw“-Befehle draw2d und draw3d. Die plot-Befehle dienen der (relativ) einfachen Erzeugung einiger standardisierter Schaubild-Typen. Mit den draw-Befehle lassen sich komplexere Graphiken erstellen; sie sind in der Regel flexibler handhabbar als die plot-Befehle. Die draw-Befehle ermöglichen insbesondere die Generierung und Gruppierung von stark unterschiedlichen Graphik-Typen und -Objekten und den flexiblen Umgang mit Graphik-Optionen.

Leider scheint es keinen fließenden Übergang zwischen der Welt der plot-Befehle und der Welt der draw-Befehle zu geben; die Syntax und auch die Optionen sind in der Regel nicht kompatibel.

Wird eine Graphik mit einem von den genannten vier Befehlen erzeugt, so geht ein neues Graphik-Fenster auf, in dem die Graphik dargestellt wird. Will man in Maxima weiterarbeiten, so muss jeweils immer zuerst dieses Graphik-Fenster geschlossen werden.

Jeden der oben genannten vier Befehle gibt es auch in der Variante mit der „Vorsilbe“ wx: Bei Verwendung der „wx“-Befehle wxplot2d, wxplot3d, wxdraw2d und wxdraw3d wird kein neues Fenster geöffnet, sondern die Graphik wird direkt im Maxima-Arbeitsblatt erzeugt. Dies erleichtert den interaktiven Betrieb, aber die Graphiken besitzen nicht dieselbe Darstellungsqualität und nicht die Manipulationsmöglichkeiten wie die ohne die wx-Vorsilbe erzeugten Graphiken.

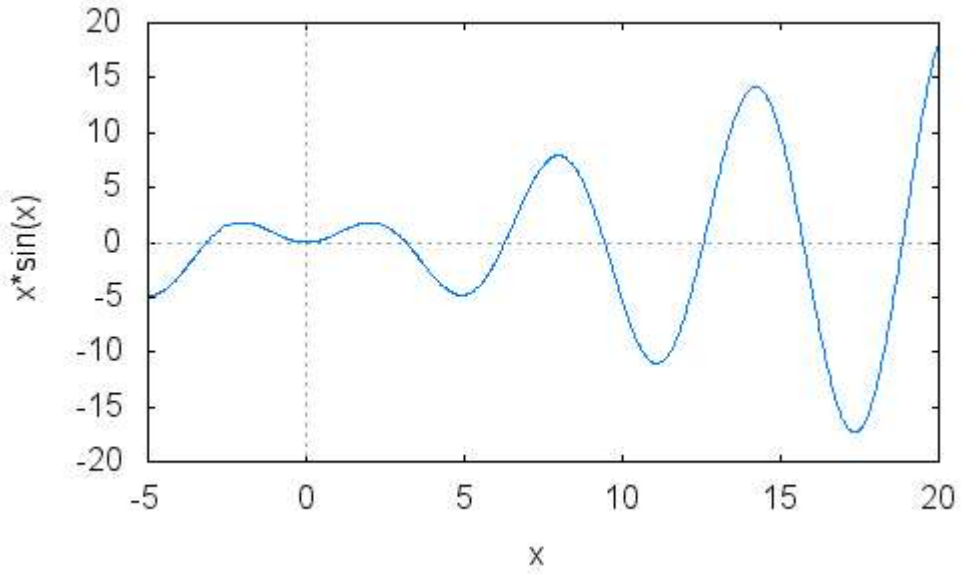
Im folgenden sind exemplarisch jeweils nur einige wenige Beispiele aufgezeigt. Die Graphik-Befehle sind in der Regel in der „wx“-Version formuliert, sie können aber auch in der „wx“-freien Version verwendet werden.

Für eine weitergehende Beschäftigung mit dem Thema „Graphiken in Maxima“ wird auf die genannte Literatur verwiesen.

## 10.1 Arbeiten mit den „plot“-Befehlen

### 10.1.1 Ein ganz einfaches Funktionenschaubild:

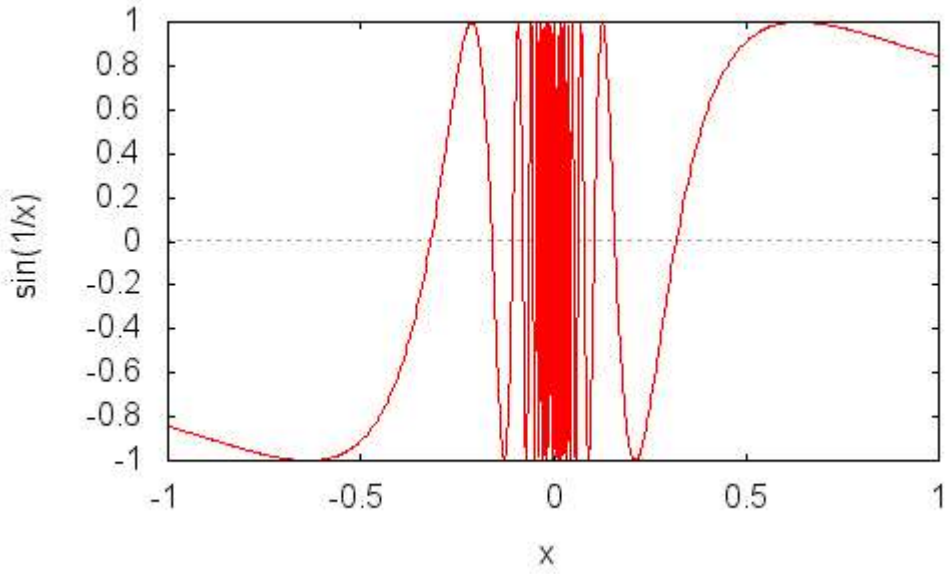
```
wxplot2d([x*sin(x)], [x,-5,20]) $
```




**10.1.2 Nutzung der Farb-Option** für das Schaubild (und Fehlermeldung wegen Division durch Null):

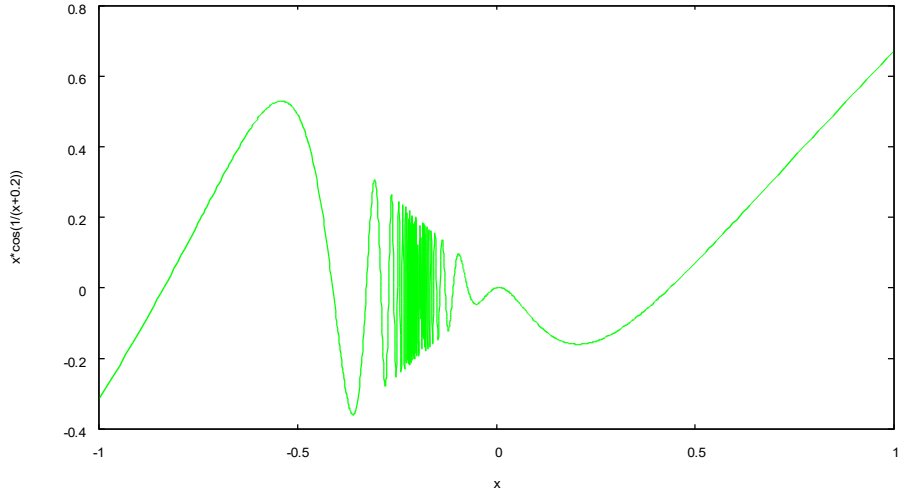
```
wxplot2d( [sin(1/x)], [x,-1,1 ], [color, red]) $
```

*System-Kommentar*: `plot2d: expression evaluates to non-numeric value somewhere in plotting range.`




### 10.1.3 Schaubild ohne Achsen:

```
plot2d([x*cos(1/(x+0.2))], [x,-1,1], [color, green], [axes, false] ) $
```



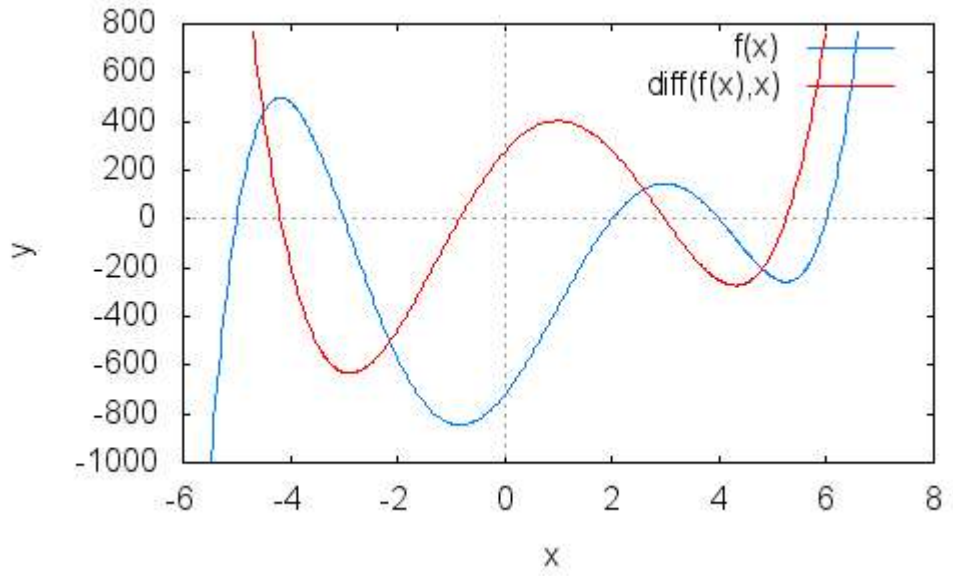

Die Standardwert (default value) für die Achsen-Option ist „true“. Das Setzen der Option auf „false“ scheint nur im modus ohne „wx“ zu funktionieren. D.h. im wx-Modus werden die Achsen immer gezeichnet.

### 10.1.4 Verwendung „anonymer“ Funktionen:

```
f : lambda([x], (x-2)*(x+3)*(x-4)*(x+5)*(x-6)) $
wxplot2d([f(x), diff(f(x),x)], [x, -6, 8], [y, -1000, 800],[legend, "f(x)",
"diff(f(x),x)"] ) $
```

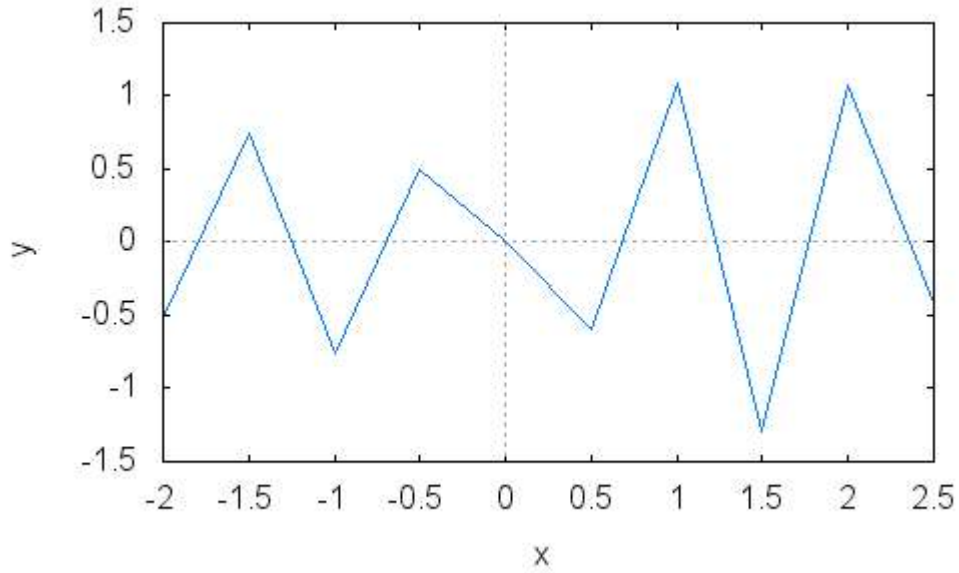
*System-Kommentar*:

```
plot2d: some values were clipped.
plot2d: some values were clipped.
```




### 10.1.5 Ein Schaubild auf der Basis diskreter Werte in einer Liste:

```
L0 : makelist(-2.0+0.5*i, i, 0, 9) $
L : makelist([x, (1.2^x)*sin(20*x)], x, L0) $
wxplot2d([discrete, L]) $
```




### 10.1.6 Parameterdarstellung:

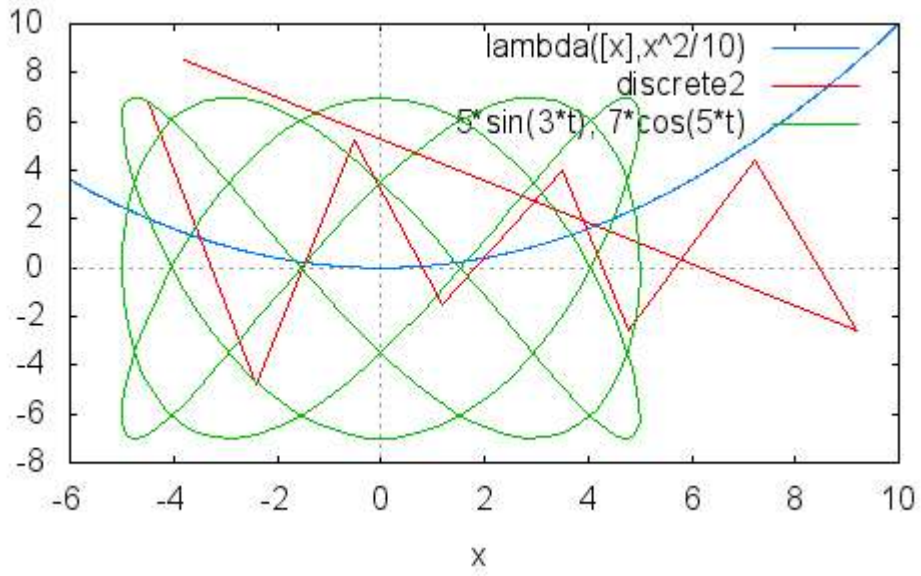
```
wxplot2d([parametric, sin(3*t), cos(5*t), [t, 0, 2*%pi], [nticks, 200]]) $
```

Probieren Sie es aus!

### 10.1.7 Mehrere Graphiken in einem Schaubild

Im folgenden Beispiel (das übrigens zeigt, dass man die Listen mit den x-Werten und den y-Werten auch getrennt eingeben kann) sind verschiedene Graphik-Typen in einem Schaubild vereint.

```
Lx : [ -4.5, -2.4, -0.5,  1.2, 3.5, 4.8,  7.2, 9.2, -3.8 ] $
Ly : [  6.8,  -4.8,  5.2, -1.5, 4, -2.6,  4.4, -2.6, 8.5 ] $
wxplot2d(
  [ lambda([x],x^2/10), [discrete,Lx,Ly],
    [parametric,5*sin(3*t),7*cos(5*t), [t,0,2*%pi] ] ],
      [x,-6,10], [nticks,200]) $
```



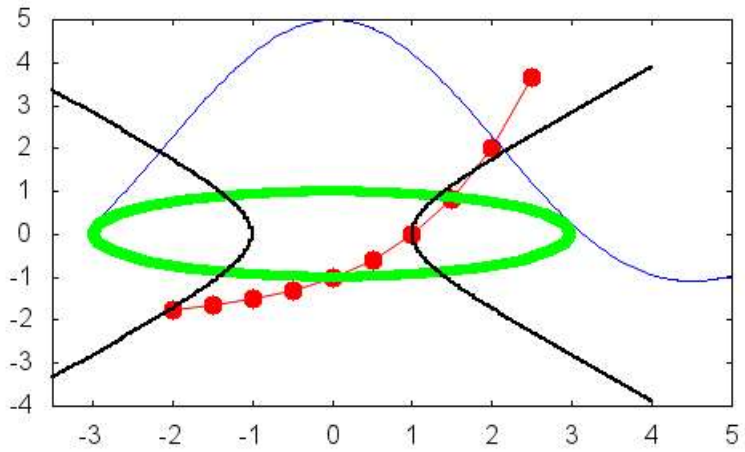

## 10.2 Arbeiten mit den „draw“-Befehlen

Zunächst ist erst einmal das draw-Paket zu laden: `load(draw);`

```
L0 : makelist(-2.0+0.5*i, i, 0, 9)  $
L : makelist([x, 2^x-2], x, L0) ;
Ausgabe:  [[-2.0,-1.75],[-1.5,-1.646446609406726],[-1.0,-1.5],[-0.5,-
1.292893218813453],[0.0,-1.0],[0.5,-0.585786437626905],[1.0,0.0],[1.5,
0.828427124746191],[2.0,2.0],[2.5,3.656854249492383]]
```

Im folgenden Beispiel ist einiges in einem kleinen Programm zusammenpepackt. Es demonstriert die Technik, zunächst die Graphik-Objekte (hier G1, … , G4) zu beschreiben und erst zu Schluss in den draw-Befehl zu packen. Wie das Beispiel (anhand der color-Option) zeigt, können Optionen zu unterschiedlichen Zeiten formuliert werden.

```
Beispiel() :=
(G1 : [color=red, point_type=filled_circle, point_size=2,
       points_joined=true, points(L)] ,
 G2 : explicit(5*sin(x)/x, x, -3, 5) ,
 G3 : [color=green, line_width=6, parametric(3*cos(t), sin(t),
       t, 0,  2*%pi)] ,
 G4 : [color=black, line_width=2, implicit(x^2-y^2=1,x,-3.5,4,y,-4,4)],
 wxdraw2d( G1, color=blue, G2, G3, G4) )
```



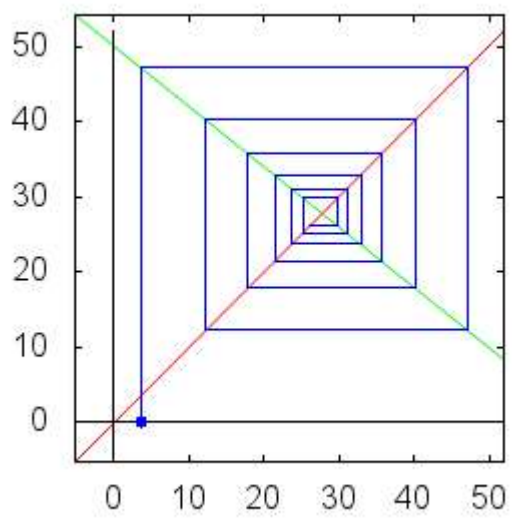
Abschließend soll als Anwendung für das Arbeiten mit „draw“ ein kleines Szenario für „cobweb-Diagramme“ zusammengestellt werden. Wie die Beispiele zeigen, gibt es keinen eigenständigen „line“-Befehl, mit dem man Strecken zeichnen könnte. Stattdessen hat man mit dem Befehl „points“ zwei Punkte zu zeichnen und die option „points-joined“ auf „true“ zu setzen.

```
load(draw);
set_draw_defaults(
 user_preamble="set size ratio -1",   /* keine Verzerrung in y-Richtung  */

 dimensions=[500,500] ) $
noframe : [axis_bottom=false, axis_left=false, axis_right=false,
axis_top=false, xaxis=true,yaxis=true,xtics_axis=true,ytics_axis=true] $
noframe_option : [] $        /* Rahmen wird gezeichnet */
noframe_option : noframe $   /* Rahmen wird nicht gezeichnet */
cobweb(a, b, y0, n) :=
 block([xmin, xmax, ymin, ymax, y:y0, ynew, L :[[y0,0]] ],
  for k : 1 thru n do
   (ynew : y*a+b, L : append(L, [[y, ynew], [ynew, ynew] ] ), y : ynew),
  xmin : lmin(map(first, L)),
  xmax : lmax(map(second, L)) ,
  if xmin >= 0 then xmin : -0.1*abs(xmax),
  if xmax <= 0 then xmax :  0.1*abs(xmin),
  xmin : xmin * 1.1, xmax : xmax *1.1 ,
  ymin : xmin, ymax : xmax,
  G1 : [color=green, point_size=0, points_joined=true,
        points([[xmin, xmin*a+b],[xmax, xmax*a+b]])],
  G2 : [color=red,   point_size=0, points_joined=true,
        points([[xmin, xmin],[xmax, xmax]])],
  G3 : [color=black, point_size=0, points_joined=true,
        points([[xmin, 0],[xmax, 0]])],
  G4 : [color=black, point_size=0, points_joined=true,
        points([[0, ymin],[0, ymax]])],
  G5 : [color=blue, point_type=filled_circle,
        point_size=1, points([[y0,0]])],
  G6 : [color=blue, point_size=0, points_joined=true, points(L)],
  wxdraw2d(noframe_option, G1, G2, G3, G4, G5, G6) );
```

Ein Aufrufbeispiel:

```
noframe_option : [] $
cobweb(-0.8, 50, 3.5, 12) $
```

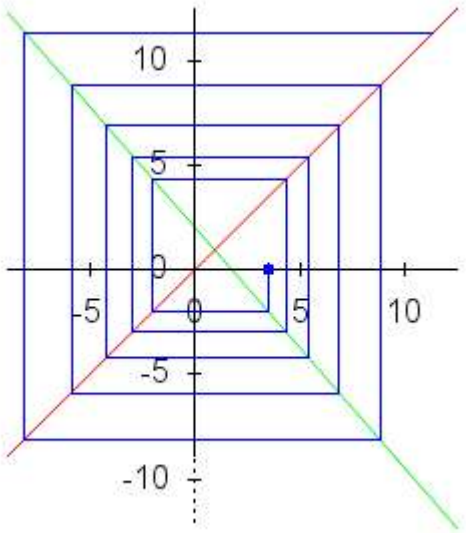



Einige weitere typische Fälle (Demonstration von Konvergenz / Divergenz und Monotonie / Nicht-Monotonie) - probieren Sie es aus!:

```
cobweb(0.75, 20, 3.5, 10) $
cobweb(1.15, 20, 23, 10) $
cobweb(-1.15, 2, 3.5, 10) $
```

Im folgenden Beispiel wird der äußere Rahmen durch ein klassisches Koordinatensystem ersetzt.

```
noframe_option : noframe $
cobweb(-1.15, 2, 3.5, 10);
```



# 11 Ausgewählte Hinweise auf einige Besonderheiten, bei denen man sich besonders am Anfang schwer tun kann

## 11.1 Zur Zifferndarstellung großer Zahlen

Die Zifferndarstellung großer natürlicher Zahlen erfolgt standardmäßig (per „default“) in einer abgekürzten Form.
*Beispiel*:

*Eingabe*: `100!`

*Ausgabe*: `933262154439441526816[98 digits]916864000000000000000`

Die vollständige (nicht abgekürzte) Ausgabe erhält man nach Eingabe des Kommandos `set_display(ascii)`:

*Eingabe*: `set_display(ascii); [Shift-Return]`

*Eingabe*: `100! [Shift-Return]`

*Ausgabe*:
```
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000
```

Die Beschreibung des Befehls `set_display(ascii)` ist im Hilfe-System leider ziemlich gut versteckt.

Wenn man die Variante „`set_display(ascii)`“ als Standardeinstellung („default“-Einstellung) haben möchte, wird man vermutlich irgendeine Initialisierungs- oder setup-Datei definieren müssen. Man kann den Befehl (etwas behelfsmäßig und u.U. auch noch mit anderen global gültigen Kommandos) als erste Zelle in neu zu schreibende Arbeitsblätter aufnehmen und am Anfang des Arbeitens mit dem jeweiligen notebook ausführen.

In neueren Versionen von wxMaxima (getestet ab Version 16.04.02) kann man die Länge der Ausgabe auch über das Menue (Bearbeiten – Einstellungen – Arbeitsblatt – Maximale angezeigte Zahl an Stellen) kontrollieren. Die Option „Auch sehr lange Ausdrücke anzeigen“ hat bei mir für diesen Zweck nicht funktioniert.

## 11.2 Zur Verwendung des Gleichheitszeichens

Die Verwendung des Gleichheitszeichens ist sehr flexibel handhabbar, aber etwas gewöhnungsbedürftig. Im folgenden ist direkt aus dem Hilfe-System von Maxima zitiert.

Operator: **=**

The equation operator.

An expression `a = b`, by itself, represents an unevaluated equation, which might or might not hold. Unevaluated equations may appear as arguments to `solve` and `algsys` or some other functions.

The function `is` evaluates `=` to a Boolean value. `is(a = b)` evaluates `a = b` to `true` when *a* and *b* are identical. That is, *a* and *b* are atoms which are identical, or they are not atoms and their operators are identical and their arguments are identical. Otherwise, `is(a = b)` evaluates to `false`; it never evaluates to `unknown`. When `is(a = b)` is `true`, *a* and *b* are said to be syntactically equal, in contrast to equivalent expressions, for which `is(equal(a, b))` is `true`. Expressions can be equivalent and not syntactically equal.

The negation of = is represented by #. As with =, an expression *a # b*, by itself, is not evaluated. is(*a # b*) evaluates *a # b* to true or false.

In addition to is, some other operators evaluate = and # to true or false, namely if, and, or, and not.

Note that because of the rules for evaluation of predicate expressions (in particular because not *expr* causes evaluation of *expr*), not *a = b* is equivalent to is(*a # b*), instead of *a # b*.

rhs and lhs return the right-hand and left-hand sides, respectively, of an equation or inequation.

See also equal and notequal.

# 12 Einige Literaturhinweise und einschlägige Internet-Adressen

Zu **Maxima** allgemein:

http://maxima.sourceforge.net
http://maxima.sourceforge.net/documentation.html
(viele Graphik-Beispiele in: „Maxima and the Calculus“ von Leon Q. Brin)
http://maxima.sourceforge.net/docs/tutorial/en/minimal-maxima.pdf
http://crategus.users.sourceforge.net/Manual/maxima.html
http://de.wikipedia.org/wiki/Maxima_(Computeralgebrasystem)
http://en.wikipedia.org/wiki/Maxima_CAS
http://www.computermathematik.info
http://www.neng.usu.edu/cee/faculty/gurro/Maxima.html
The Maxima Book. Paulo Ney de Souza. Richard J. Fateman. Joel Moses. Cliff Yapp. 19th September 2004 (ist auf den SourceForge Seiten zu finden)
http://www.austromath.at/daten/maxima/
http://www.neng.usu.edu/cee/faculty/gurro/Maxima.html (Gilberto E. Urroz. My Maxima Page)

Zum (Schwerpunkt-) Thema **Graphik in Maxima**:
http://www.austromath.at/daten/maxima/zusatz/Grafiken_mit_Maxima.pdf
(eine ausführliche Diskussion des Themas „Graphiken“ in Maxima von Wilhelm Haager)

Zum printf-Befehl für **formatiertes Ausdrucken**:
http://en.wikipedia.org/wiki/Printf
http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html

Zum Thema **xwMaxima**:
http://andrejv.github.com/wxmaxima/help.html
Dort findet man viele weitere Hinweise auf themenspezifische Einführungen, Tutorials (auch per Video) und dergleichen.

Literaturhinweise zum Themenbereich **„Algorithmen / Informatik / Programmiersprachen / Lisp“** siehe: https://jochen-ziegenbalg.github.io/materialien/

**Ergänzende Literatur** (vor allem im Hinblick auf die behandelten Programme):

**AHG**: Ziegenbalg, „Algorithmen – von Hammurapi bis Gödel“, Springer Spektrum 2016
https://www.springer.com/de/book/9783658123628