Eine Kurzeinführung in das Computermathematiksystem

Maple

orientiert an der Vorlesung von Prof. Christoph Zenger
TU München
1. Semester, WS 1999 / 2000

von Michael Wack, Christoph Moder, Manuel Staebel

http://www.skriptweb.de

Inhaltsverzeichnis

Maple V Release 5.1

Ausdrücke

Ein Ausdruck ist ein ***Maple-Kommando*** (eine Formel). Er endet immer mit einem ***Strichpunkt*** (oder einem ***Doppelpunkt***), und wird nach der Eingabe ausgewertet:

```
x + y * 2;
```

ist ein Beispiel für einen Ausdruck.

Das Ergebnis wird ausgegeben, wenn es ein Semikolon ist; bei einem Doppelpunkt nicht.

Variablen

Mit „**:=**“ weist man einer Variablen einen Wert zu, wenn man nur den Variablennamen angibt, erhält man ihren Wert:

```
f := 3;   (f erhält den Wert 3)
f;        (liefert 3)
```

Substitution

Mit der Funktion **subs()** ersetzt man einen „Unter-Ausdruck“ in einem Ausdruck durch etwas anderes; eine Anwendung wäre z.B.:

```
f := x^2 + 3;
subs(x = 5, f);     (wertet die Funktion f an der Stelle 5 aus)
```

Um mehrere Variablen zu ersetzen, kann man **subs()** auch verschachtelt verwenden, oder auch mit mehreren Gleichungen:

```
subs( h=1./100., subs( R=6378., d));
subs( h=1./100., R=6378., d);          # sind beide gleich
```

Wenn man nicht wie hier einmal einer Variablen einen Wert zugewiesen werden soll, sondern die Variable den Wert dauerhaft haben soll, geht das wieder mit „**:=**“. Wenn der Wert wieder gelöscht werden soll, dann weist man der Variablen ihren Namen in Hochkommata zu.

```
f := x^2 + 3;
x := 5;
f;        (wertet ebenfalls die Funktion f an der Stelle 5 aus,
          aber x hat auch nachher den Wert 5)
(...)     (irgendwelche anderen Operationen mit x....)
x := 'x'; (so wird x wieder undefiniert gemacht)
```

Interessante Befehle:

Mit **restart** setzt man alle Einstellungen und Variablenzuweisungen zurück wie beim Programmstart.

```
restart;       (Dann hat Maple alles vergessen)
```

Mit **simplify()** kann man einen Ausdruck vereinfachen lassen (als zweites, drittes... Argument kann man angeben, nach welcher Operation vereinfacht werden soll, ansonsten geht das automatisch).

```
y := 4^(1/2) + 3;
simplify(y);        (liefert 5)
```

Mit **limit()** kann man den *Grenzwert* eines Ausdrucks ausrechnen:

```
y := x^2;
```

```
limit(y, x=infinity, left);  (der linke Grenzwert von y gegen
                              Unendlich ist Unendlich)
```

Mit **whattype()** kann man feststellen, ob das eingesetzte Ding eine Variable, eine Zahl... ist:

```
y := 7;
whattype(y);   (liefert integer zurück)
```

Mit **rhs()** und **lhs()** kann man die *rechte* bzw. *linke Seite* einer Gleichung, Ungleichung (nur „kleiner als“) oder eines Intervalls erhalten:

```
lhs( a+5 = b+4);    (liefert a+5)
rhs( a+5 < b+4 );   (liefert b+4)
rhs( a+5 > b+4 );   (liefert seltsamerweise a+5)
rhs( 2...5);        (liefert 5)
```

Mit **ithprime()** kann man sich die *i-te Primzahl* liefern lassen:

```
ithprime( 5 );      (liefert 11)
```

Mit **evalf()** wertet Maple den angegebenen Ausdruck nicht exakt, sondern als Gleitkommadarstellung aus (anstatt exakt), was viel schneller geht (wichtig bei komplizierten Ausdrücken); die Genauigkeit stellt man mit der Umgebungsvariablen **Digits** (Schreibweise!!!) ein:

```
a := 1/3; b := 1/7;
a + b;          (dauert lange, liefert 10/21)
evalf(a+b);     (geht schneller, liefert .4761904762)
Digits := 5;    (jetzt 5 Stellen Genauigkeit)
c := 3. ;       (der Punkt bedeutet: behandle die 3 als
        Gleitkommazahl statt Ganzzahl)
```

Mit **%** ist der zuletzt ausgewertete Ausdruck gemeint (mit **%%** der vorletzte, mit **%%%** der vorvorletzte):

```
a + 2*b;            (liefert 13/21)
evalf(%);           (liefert die entspr. Gleitkommazahl)
```

Mit **sum()** kann man Zahlen summieren, es entspricht der Sigma-Notation:

```
sum('a[k]*x^k','k'=0..2);     (summiert a₀*x⁰ bis  a₂*x²)
```

Mit **print()** kann man die Ausgabe erzwingen (z.B. wenn sie durch den Doppelpunkt unterdrückt wird).

Mit **ERROR()** kann man eine Fehlermeldung ausgeben; aus Funktionen aufgerufen, springt das Programm dorthin zurück, von wo aus die Funktion aufgerufen wurde, und gibt den Funktionsnamen sowie die angegebenen Parameter aus:

```
ERROR('invalid x', x);   (liefert Error, (in f) invalid x, -3)
```

Ähnliches bewirkt die Konstante **FAIL**. Wenn sie zurückgegeben wird, heißt das, es ist ein Fehler aufgetreten. In *if*-Konstruktionen u.ä. wirkt sie wie *false*.

Mit **round()**, **ceil()**, **floor()**, **trunc()** und **frac()** werden Zahlen gerundet bzw. aufgeteilt: **round()** rundet die Zahl, **ceil()** rundet sie auf die nächste Ganzzahl auf, **floor()** rundet sie auf die nächste Ganzzahl ab, **trunc()** ist so ähnlich und liefert den Ganzzahlteil einer Zahl (entfernt alle Nachkommastellen), und **frac()** schmeißt den Ganzzahlteil weg und liefert die Nachkommastellen.

Einfache Funktionen:

Funktionen definiert man mit dem Operator ->.

```
(x -> x^2) (2);     (liefert 4)
```

```
f := x -> x^2;      (definiert die Funktion f)
f(2);               (wertet diese Funktion aus und liefert 4)
```

Das funktioniert auch mit mehreren Parametern:

```
f := (x,y) -> (x^2, y^3);
f(7,3);             (liefert 49,27)
```

Das funktionale „if“:

Die Funktion **if()** hat als ersten Parameter eine Bedingung, die entweder *true* oder *false* liefert; wenn das Ergebnis *true* ist, wird der zweite, ansonsten der dritte Parameter zurückgeliefert. Zu beachten: das Wort „if“ muss in Backticks geschrieben sein, weil es ein reserviertes Wort ist und wir es als Funktionsnamen verwenden!

```
gerade := n -> `if`(n mod 2 = 0, 'gerade', 'ungerade');
gerade(5);          (liefert ungerade)
```

Rekursion:

Mit Funktionen lassen sich auch ganz normal Rekursionen definieren.

Eine Rekursion bezeichnet man als **verschränkte Rekusion**, wenn sich mehrere Rekursionen gegenseitig aufrufen:

```
gerade := n -> `if`(n=1, false, ungerade(n-1));
ungerade := n -> `if`(n=1, true, gerade(n-1));
```

Hier ruft die Funktion *gerade* die Funktion *ungerade* auf und umgekehrt.

Umfangreichere Funktionen:

... passen nicht in eine Zeile, sondern hier geht man besser anders vor:

```
ggT := proc(a, b)
        option arrow, operator, trace;
        local x, y;           (lokale Variablen)
        global gx, gy;        (globale Variablen)

        `if`(b mod a = 0, a, ggT(b mod a, a))
end;

ggT(13*19, 13*17);       (liefert 13)
```

a und b sind also die Parameter der Funktion, die ggT heißt. Sie wird eingeschlossen durch **proc** und **end**. Man kann Optionen angeben, **trace** bewirkt z.B. dass man den Funktionsablauf debuggen kann.

Man kann den Variablentyp der Parameter festlegen, indem man ihn durch zwei Doppelpunkte angibt:

```
meine_funktion := proc (x::nonnegint)
```

Normalerweise wird die letzte in der Funktion ausgewertete Formel zurückgegeben, außer man gibt mittels **RETURN** ausdrücklich etwas anderes an.

Um einem Parameter einen Wert zurückzugeben, muss dieser Parameter in Hochkommata stehen, damit er nicht ausgewertet wird:

```
primsum := proc(n, m, liste)
 local i, s;  liste := [seq(ithprime(i), i=n..m)];
 s := 0;
```

```
 for i in op(liste) do s := s + i od;
end;  primsum(25, 30, 'ergebnis');  ergebnis;          (liefert
jetzt 630)
```

Einbinden von packages

Um Funktionen aus irgendwelchen Paketen zu benutzen, kann man folgendermaßen vorgehen:

```
with( linalg ):
det( A );           # berechnet die Determinante der Matrix A
```

Diesen Befehl führt man aus, bevor man das erste Mal einen Befehl aus dem entspr. Paket braucht (hier *linalg*); dahinter kommt sinnvollerweise ein Doppelpunkt statt eines Semikolons, weil sonst alle Funktionen des Pakets aufgelistet werden - eher nervig.

Es geht aber auch mit **Paketname[Funkionsname](Parameter)**:

```
linalg[det](A);     # berechnet die Determinante der Matrix A
```

Und Subpackages gibt es auch, z.B. im Paket **stats** (nur als Beispiel; *random* ist das Subpackage):

```
stats[random, normald](20);
```

Sequenzen:

Mehrere Objekte hintereinander, durch Komma getrennt, bilden eine ***Sequenz***.

```
muh, maeh, sin(muh)^(1/2), muh=maeh;
```

Man kann Sequenzen bequem mit der **seq()**-Funktion bilden; auf ihre Elemente kann man mit einem Index, den man in eckigen Klammern angibt und der ab 1 zählt, zugreifen:

```
sequenz := seq(x^2, x=1..4);       (liefert 1,4,9,16)
sequenz[4];                        (liefert 16)
sequenz[2..4];                     (liefert 4,9,16)
sequenz2 := seq( seq(j, j=0..i), i=0..4);
sequenz3 := seq( x^3, x = sequenz );   (hier keine
              Zählvariable, sondern eine Sequenz)
```

Man kann die Funktion **seq()** also auch verschachteln, und erhält aber wieder eine einzige Sequenz.

Listen:

Listen (bzw. Vektoren) sind geordnete Sequenzen, sie werden in ***eckigen Klammern*** dargestellt.

```
liste := [sequenz];
liste;              (liefert [1,4,9,16])
```

Man kann sie genauso definieren wie Sequenzen einfach durch Angabe ihrer Elemente.

```
liste := [1,2,3,7,15,4];
```

Man kann genauso auf die Elemente zugreifen, sie aber auch verändern:

```
liste[4];           (liefert 7)
liste[4] := 4;
```

Mit **zip** kann man z.B. zwei Listen einfach addieren:

```
zip( (x,y) -> x+y, [1,2,3], [4,5,6] );      (ergibt [5,7,9])
```

Mengen:

Sind wie Listen, aber ungeordnet, und sie enthalten keine doppelten Elemente. Sie werden in ***geschweiften Klammern*** geschrieben.

```
menge := {1,2,4,5,6};
menge intersect {2,3,4};      (liefert {2,4})
```

Anwendungen von Sequenzen, Listen und Mengen:

Sie definieren z.B. eine Punktemenge, die es zu zeichnen gibt:

```
pts := [seq([i, sin(Pi*i/10)], i=0..10)];
with plots: pointplot(pts);
```

Die Funktion **member()** ermittelt, ob der Ausdruck im ersten Parameter in der Liste oder Menge, die im zweiten Parameter angegeben ist, vorkommt, und liefert *true* oder *false* zurück. Falls als drittes Argument eine Variable gegeben ist, wird in diese die Position des gefundenen Elements in der Liste oder Menge zurückgegeben:

```
member(w, [x, y, w, u], 'k');      (liefert true, und k = 3)
```

Zwei Kreise zeichnen, ihre Schnittpunkte ermitteln:

Kreise werden durch die Kreisgleichung definiert:

```
k1 := x^2 + y^2 = 1;                    (Radius = 1)
k2 := (x-1/2)^2 + (y-1/3)^2 =1/4;       (Radius = ½)
```

Zeichnen kann man sie mit **implicitplot()**:

```
with(plots):
implicitplot({k1,k2}, x=-2..2, y=-2..2, scaling=CONSTRAINED);
```

Gezeichnet wird also eine ***Menge von Funktionen***, deshalb in geschweiften Klammern. Die *scaling* -Angabe bewirkt, dass die Skalierung so ist, dass die Kreise auch rund aussehen (nicht elliptisch).

Mit **solve()** löst man beide Gleichungen nach x und y auf:

```
lsg := solve({k1,k2},{x,y});
```

Das Ergebnis enthält **RootOf()**, was bedeutet, das es nicht eine Wurzel ist, sondern dass es mehrere Wurzeln gibt, hier nämlich zwei (positiv und negativ). Um alle Ergebnisse zu erhalten, verwendet man **allvalues()**, das aus RootOf alle Lösungen herauszieht:

```
alle_lsg := allvalues(lsg);
```

Lösen eines Gleichungssystems:

Angenommen, man hat eine Gleichung mit mehreren Unbekannten, und entsprechende Werte, um daraus ein lösbares Gleichungssystem zu machen:

```
gleichung := a*x^2 + b*x*y + c*y^2 + d*x +e*y = f;
X := [1, -1, -3, 4, 2];  Y := [1, 2, -1, 5, -3];
```

Das ist übrigens die Gleichung für Kegelschnitte. 5 Lösungen sind bekannt (jeweils x und y).

Mit **seq()** kann man alle Punkte in die Gleichung einsetzen:

```
glsystem := {seq(subs({x=X[k], y=Y[k]}, gleichung), k=1..5)};
```

glsystem ist jetzt also die Menge der Gleichungen, mit jeweils einem anderen Punkt eingesetzt.

Mit **solve()** kann man diese Menge von Gleichungen nach allen Koeffizienten auflösen (einer bleibt natürlich übrig - bei 6 Koeffizienten und 5 Lösungen):

```
koeffizienten := solve(glsys, {a, b, c, d, e, f});
```

Jetzt setzt man die Koeffizienten in die urspr. Gleichung ein:

```
neue_gleichung := subs(koeffizienten, gleichung);
```

Voilà. Nur noch ein Koeffizient übrig.

Schleifen:

Die Zählschleife:

```
for i from 4 by 2 to 20
do
        i;
od;
```

Diese Schleife zählt in Zweierschritten von 4 bis 20. Lässt man ***from*** oder ***by*** weg, wird dafür 1 angenommen.

```
for i from 1 while( sin(evalf(i)) > 0 )
do
        i;
od;
```

Zählschleifen können also auch mit einer Bedingung enden. Hier ist **evalf()** nötig, weil die Sinus-Funktion mit dem Integer der Zählschleife nichts anfangen kann.

Oft kann man Zählschleifen kurz durch **seq()** ersetzen:

```
seq( ithprime(i), i=100..110 );
```

Bedingungsschleifen:

```
i := 1;
while( sin(evalf(i)) > 0 )
do
        i;
        i := i + 1;
od;
```

... ist dieselbe Schleife wie die zweite for-Schleife.

Bemerkungen zu Schleifen:

Um die Schleife vorzeitig zu verlassen oder einen Durchlauf zu überspringen gibt es **break** und **next**.

Wenn man hinter **od** einen Doppelpunkt setzt, so beeinflusst es die Ausgabe der gesamten Schleife. Siehe auch **printlevel**.

Ein besonderer Fall: Wenn man eine Funktion mehrmals ausführen will, gibt es hierfür eine ***Kurzschreibweise***:

```
plot( (leite_ab@@4)(x^5), x=-10..10 );
```

Hier wird die Funktion *leite_ab* viermal aufgerufen, zuerst mit dem angegebenen Parameter (*x^5*), dann jeweils mit dem Rückgabewert des vorigen Durchlaufs, und der letzte Rückgabewert wird schließlich gezeichnet.

Verzweigungen - die „if“-Anweisung

Statt der schon erwähnten kurzen **if**-Funktion gibt es auch eine ausführliche:

```
if a = b
then
        "gleich"
elif a = c
        "wenigstens a und c gleich"
else
        "ungleich"
fi;
```

Graphische Ausgabe

Funktionen lassen sich ganz einfach mit den **Plot**-Funktionen zeichnen:

```
plot( x^2, x=-2..2 );          (zeichnet eine Parabel)
plot3d( {x^2, 10*sin(y)}, x=-10..10, y=-10..10); (drehbar 3D)
```

Außerdem bietet das **plots**-Paket eine ganze Menge von Routinen für jeden Einsatzbereich, z.B.:

```
with(plots):
implicitplot({x^2 + y^2 = 1,y = exp(x)},x=-Pi..Pi,y=-Pi..Pi);
        (zeichnet einen Kreis und die Exponentialfunktion)
pointplot({[0,1],[1,-1],[3,0],[4,-3]});
        (zeichnet eine Liste oder Menge von Punkten)
```

Um mehrere Funktionen in eine Grafik zeichnen zu können, speichert man die *Pointlist*, die alle Plot-Funktionen zurückliefern, in Variablen ab und füttert diese Menge von Variablen dann an **display**:

```
with(plots):
p1 := implicitplot( x^2 + y^2 = 1, x=-Pi..Pi, y=-Pi..Pi ):
p2 := plot( sin(x), x=-10..10, color=BLUE ):
display( {p1, p2} );
```

Leider kann man dabei nicht 2D- und 3D-Grafiken mischen. Es empfiehlt sich, die Zeichenfunktionen mit einem Doppelpunkt abzuschließen, weil sonst die riesigen Plot-Listen angezeigt werden.

Genauso kann man eine Zeichnung „nach und nach“ erstellen, z.B. in einer Rekursion, wenn in jedem Durchlauf etwas zur Zeichnung dazukommt, und nicht bekannt ist, was noch dazukommt in tieferen Rekursionsebenen. Dann zeichnet man jedesmal in die Punktliste, und erst nach der Rekursion lässt man das Ergebnis mit **display()** raus:

```
neuer_Teil := plot( irgendwas, irgendwie );
Grafik := Grafik union neuer_Teil;
```

Alle **plot**-Funktionen akzeptieren die selben Optionen (siehe **?plot[options]**), die als letzte Parameter angegeben werden, z.B.:

```
view=[xmin..xmax, ymin..ymax] (zeichnet nur diesen Bereich)
scaling=CONSTRAINED           (dann wird nicht verzerrt)
```

Mit **plotsetup** kann man sogar das Ausgabeformat einstellen, z.B. als *Gif*-, *JPEG*-, oder *PostScript*-Datei.

Wenn man eine dreidimensionale Funktion hat, und möchte die Werte nicht als Punkte im Raum, sondern die Funktion als Fläche darstellen, erzeugt man zuerst die Eckpunkte der Einzelflächen (vier Punkte ergeben ein Quadrat), und lässt das Ergebnis dann mit **polygonplot3d()** zeichnen:

```
flaeche := (i,j) -> [ punkt(i,j), punkt(i+1,j),  punkt(i+1,j+1),
punkt(i,j+1), punkt(i,j) ];

polygonplot3d( [ seq( seq( flaeche(i,j), i=0..n), j=0..n ) ],
        axes=BOXED );
```

(Die Funktion *punkt()* würde hier also den Wert des Punktes an den angegebenen Koordinaten liefern, die Werte könnten auch aus einem zweidimensionalen Array stammen.)

Tabellen und Felder

Tabellen (Hash-Tabellen bzw. assoziative Arrays)

Eine Tabelle wird mit dem Befehl **table()** eingerichtet, dabei können auch gleich Werte eingetragen werden:

```
tabelle1 := table();          (erzeugt eine leere Tabelle)
tabelle2 := table( ["2 DM", "3 DM", "1 DM"] );
           (erzeugt eine Tabelle mit 3 Einträgen, die über den
           Index 1, 2 und 3 erreichbar sind)
tabelle3 := table( [("Apfel") = "2 DM", ("Birne") = "3 DM"] );
          (erzeugt eine Tabelle mit zwei Einträgen, die über
          die Indices „Apfel“ und „Birne“ erreichbar sind)
```

Die Elemente der Tabelle werden folgendermaßen erreicht:

```
tabelle2[2];              (liefert "3 DM")
tabelle3["Apfel"];        (liefert "2 DM")
```

Genauso funktioniert die Zuweisung:

```
tabelle3["Birne"] := "heute kostenlos";
```

Um eine Tabelle anzuzeigen, reicht es nicht, wie bei anderen Variablen, einfach den Variablennamen mit einem Strichpunkt hinzuschreiben; man muss mit **op()** arbeiten:

```
op( tabelle3 );           (liefert die gesamte Tabelle 3)
```

Um alle Indices zu erhalten (also die „linken Seiten“ der Tabelle), benutzt man die Funktion **indices()**:

```
indices( tabelle2 );      (liefert [1],[2],[3] )
indices( tabelle3 );      (liefert ["Apfel"],["Birne"] )
```

Damit die Indices nicht als Listen erscheinen (in eckigen Klammern), sondern als Sequenz, entpackt man sie mit **op()** und listet sie der Reihe nach mit **seq()** auf:

```
seq(op(i), i=indices( tabelle3 ) );
        (liefert "Apfel","Birne" )
```

Arrays (Felder)

Arrays sind vergleichbar mit Tabellen, aber (1) restriktiver - sie haben eine feste Dimension - und (2) effizienter bei großen Datenmengen.

```
feld1 := array( 2..5 );       (erzeugt eindimensionales Array)
feld2 := array( 2..5, 1..3 ); (zweidimensionales Array)
```

Um auf ein Element eines zweidimensionalen Arrays zuzugreifen, verwendet man folgende Syntax:

```
feld2[ 3, 2 ] := 7;
```

Sowohl für Tabellen als auch für Felder gilt:

1. Das Schlüsselwort **symmetric** sorgt dafür, dass bei mehrdimensionalen Array oder Tabelle die Indices vertauscht werden können:

```
A := array( symmetric, 1..5, 1..5 );
(dann ist A[2, 3] gleichbedeutend mit A[3, 2])
```

2. Um Operationen auf alle Elemente des Arrays oder der Tabelle anzuwenden, verwendet man am besten die Anweisung **map**:

```
map( diff, A, x );
(differenziert alle Elemente des Arrays A nach x)
```

Matrizen und Vektoren

Eindimensionale Arrays, die mit dem Index 1 beginnen, werden in Maple auch als ***Vektoren*** (Typ **vector**), zweidimensionale Arrays, die mit dem Index 1 beginnen, als ***Matrizen*** (Typ **matrix**) bezeichnet.
Mit Matrizen kann man die zahlreichen Funktionen aus dem **linalg**-Paket benutzen, wie z.B. zur Matrixmultiplikation.

Um eine einfache Testmatrix zu erzeugen, kann man z.B. einen Befehl in dieser Art verwenden:

```
TestM := n -> matrix(n,n, (i,j)->1/(i+j));
```

Oder man bemüht den Befehl **randmatrix()**, der als Parameter die Anzahl der Zeilen und Spalten hat. Optional kann man noch Parameter angeben, wie **entries=f** (die Funktion f gibt an, wie die Werte erzeugt werden, standardmäßig mit **rand(-99..99)**), **unimodular** (Matrix in Zeilen-Stufenform), **symmetric** (an der Diagonalen [1,1]-[n,n] sind alle Werte gespiegelt) und **antisymmetric** (wie symmetric, aber Diagonale ist 0, auf beiden Seiten gegensätzliche Vorzeichen). Das Vektor-Pendant zu **randmatrix()** ist **randvector()**.

Um die Dimension einer Matrix herauszufinden, eignen sich die Funktionen **rowdim()** und **coldim()**. Als Parameter wird jeweils die Matrix erwartet, deren Größe bestimmt werden soll.

Mit **addcol()** und **addrow()** berechnet man die Linearkombination aus zwei Matrixspalten bzw. -zeilen:

```
addrow(A,1,2,10);   (addiert das 10-fache der 1. Zeile zur 2.)
```

Um ein Element einer Matrix zu eliminieren, kann man die Linearkombination auch verwenden:

```
addrow(A, Spalte, Zeile, -A[Zeile,Spalte]/A[Spalte,Spalte]);
```

Dabei wird also das Element an [Spalte,Spalte] (Wert y), das in derselben Spalte wie das zu entfernende Element steht, mit einer Zahl a = seinem negativen Kehrwert und dem zu entfernenden Element multipliziert; mit a werden alle Elemente in dieser Zeile multipliziert und jeweils zu den Elementen der Zeile, in der das zu entfernende Element steht, addiert. Für das zu entfernende Element (Wert x) heißt das: $x + a \cdot y = x + -\frac{x}{y} \cdot y = x + (-x) = 0$.

Um die Matrix auf Zeilen-Stufenform zu bringen, führt man diesen Schritt mehrfach aus: in jeder Spalte ab der <Spaltennummer + 1>.ten Zeile werden die Elemente auf 0 gebracht.

Mit **transpose()** wird jedes Element der Matrix an der Diagonalen (1,1)(n,n) gepiegelt, es wandert von (Zeile,Spalte) nach (Spalte,Zeile): aus einer Matrix mit x Spalten und y Zeilen wird eine mit y Spalten und x Zeilen, aus den Zeilen werden Spalten und umgekehrt.
inverse() erzeugt die Inverse einer Matrix, die, wenn man sie mit der ursprünglichen Matrix multipliziert,

die Einheitsmatrix ergibt.

Eine Einheitsmatrix mit 10 Zeilen und 15 Spalten erzeugt man folgendermaßen:

```
array(identity, 1..10, 1..15 );
```

rank() liefert den Rang einer Matrix, d.h. die Anzahl der Zeilen, die sie nach einer Gauß-Elimination hat.

Um zwei oder mehrere Matrizen nebeneinander zu kombinieren, gibt es **augment()** bzw. **concat()** (sind identische Funktionen); um sie vertikal übereinander zu stapeln verwendet man **stackmatrix()**:

```
concat( A, B );
stackmatrix( A, B );
```

Das ist z.B. praktisch, wenn man eine Matrix und einen Vektor hat und das sich ergebende Gleichungssystem mittels Gauß-Elimination (Funktion **gausselim()**) lösen möchte:

```
M := gausselim( concat( A, b ) );
```

Danach hat M Zeilenstufenform, alles links unterhalb der Diagonale ist 0.

Jetzt greift man zur Rückwärtssubstitution, die durch **backsub()** ausgeführt wird:

```
x := backsub( M );
```

Lösen eines linearen Gleichungssystems

Zuerst wird das Gleichungssystem aufgestellt: in diesem Beispiel (eindimensionale Wärmeleitung von Blatt 11) hängt der Wert einer Variablen u[i] von zwei anderen Variablen ab, u[i-1] und u[i+1], die erste ist 0, die letzte 1.

```
gls:=[u[0]=0, seq(u[i]=(u[i-1]+u[i+1])/2, i=1..n), u[n+1]=1];
```

Die Unbekannten sind u[0] bis u[n+1]:

```
unbek := [ seq( u[i], i=0..n+1)];
```

Aus dem Gleichungssystem und den Variablen wird eine erweiterte Koeffizientenmatrix erzeugt:

```
Ab := genmatrix( gls, unbek, flag );
```

flag bedeutet: es wird eine erweiterte Koeffizientenmatrix erzeugt. Man kann aber auch stattdessen eine Variable angeben, in die der Vektor gespeichert wird, die Matrix hat dann 'normales' Format.

```
A := genmatrix( gls, unbek, 'b' );
```

Jetzt wird das Gaußsche Eliminationsverfahren angewandt, wir erhalten die Matrix in Zeilen-Stufen-Form:

```
Ab_elim := gausselim( Ab );
```

Und mit der Rücksubstitution erhält man die Lösung, einen Vektor mit den Variablen u[0] bis u[n+1]:

```
lsg := backsub( Ab_elim );
```

Bäume

Ein Baum besteht aus ***Knoten***, die Daten enthalten, und von denen jeder ein oder mehrere Sohnknoten hat; an jedem Sohnknoten können weitere Knoten hängen, also hängt an jedem Knoten ein ***Teilbaum*** des Baumes. Hat ein Knoten keinen Sohnknoten, bezeichnet man ihn nicht als Knoten, sondern als ***Blatt***.

Wenn alle Knoten eines Baums nur höchstens zwei Sohnknoten besitzen, heißt er ***binärer Baum***.

Die ***Tiefe*** eines Baumes ist die Zahl der Knoten auf dem Weg von der Wurzel bis zum tiefsten Blatt. Besteht der Baum nur aus dem Wurzelknoten, so hat der die Tiefe 1 (die Wurzel ist also dann ein Blatt);

ist der Baum ganz leer, enthält also die Wurzel auch keine Daten, so ist die Tiefe des Baumes 0. Man kann definieren, dass z.B. ein (Teil-)Baum leer ist, wenn er den Wert *LEER* hat (statt einer Liste aus Knoten und Teilbaum).

Wenn ein binärer Baum Knoteninhalte hat, die man bezüglich ihrer Größe vergleichen kann, und der erste (linke) Teilbaum eines Knotens Inhalte hat, die kleiner als der Knoten sind, und der zweite, rechte Teilbaum Inhalte hat, die alle größer als der Knoteninhalt sind, nennt man den Baum einen ***Suchbaum***, weil er die Suche nach einem Wert natürlich stark vereinfacht: kennt man den Inhalt eines Knotens, und weiß, ob der gesuchte Wert größer oder kleiner sein muss, kann man bei der Suche einen kompletten Unterbaum ignorieren. Ein Suchbaum mit 2^{t-1} Knoten hat höchstens die Tiefe 2^{t-1} (nämlich wenn jeder Knoten nur einen Sohnknoten hat, also alle Knoten untereinander stehen und pro Knoten eine Tiefenebene hinzukommt) und mindestens die Tiefe t (wenn jeder Knoten zwei Sohnknoten hat). Einen Suchbaum mit möglichst geringer Tiefe nennt man ***gut balanciert***, weil dann der Suchaufwand klein ist, man muss wenige Rekursionsebenen durchlaufen.

In Maple kann man Bäume z.B. als verschachtelte Listen darstellen: als erstes Listenelement steht der Inhalt des Knotens, als zweites eine Liste, die die Unterbäume enthält: jeder Zweig ist wieder eine Liste, mit dem Knoteninhalt als erstem und der Unterbaum-Liste als zweitem Argument. Weil die Verschachtelungstiefe und die Anzahl der Unterbäume nicht bekannt ist, durchläuft man einen Baum am besten durch Rekursion.
Ein Baum sieht z.B. so aus:

```
baum1 := [K01, [[K11, []], [K12, [[K221, []], [K222, []]]],
        [K13, [[K23, []]]]]];
```

Die Wurzel, der Knoten *K01*, hat 3 Zweige (*K11*, *K12*, *K13*), von denen *K11* ein Blatt ist, denn es hat keine eigenen Zweige. Von *K12* gehen zwei Zweige, von *K13* geht ein Zweig aus.

```
baum2 := [K0, []];
```

Dieser Baum besteht nur aus dem Wurzelknoten und hat also die Tiefe 1.

```
baum3 := LEER;
baum4 := [K0, [LEER, [K12, []] ] ];
```

Der Baum3 ist komplett leer, hat also die Tiefe 0; Baum4 hat die Tiefe 2.

Einen Baum durchläuft man rekursiv z.B. so:

```
finde := proc(baum, ding)
        local teilbaum;

        if baum<>LEER then

          if baum[1]=ding then
               print(baum)
          fi;

          for tb in baum[2] do
               finde(teilbaum, ding);
          od;

        fi;
end:
```

Algorithmen

Bresenham-Algorithmus

Der Bresenham-Algorithmus eignet sich zum schnellen Zeichnen von Linien auf einem gerasterten Ausgabegerät wie z.B. dem Bildschirm, weil er mit Integer-Arithmetik auskommt.

Angenommen, eine Linie soll zwischen [*x1*, *y1*] und [*x2*, *y2*] gezeichnet werden:

```
dx := x1-x0; dy := y1-y0;
d := (x,y) -> 2*(dy*x - dx*y + dx*b);   (Entscheidungsfunktion)
```

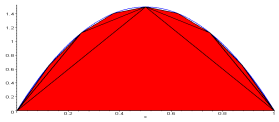
Jetzt iteriert man mit *xp* von *x1* bis *x2*:

```
if( d(xp,yp) < 0 )
then
        xp := xp + 1;
else
        xp := xp + 1;  yp := yp + 1;
fi;
```

Man wandert also nur dann einen Punkt nach oben, wenn die Entscheidungsfunktion einen Wert größer 0 liefert, was bedeutet, dass der letzte Punkt unter der Ideallinie war.

Archimedischer Algorithmus

Um die Fläche unter einer Kurve anzunähern, kann man den Archimedischen Algorithmus verwenden. Dabei wird im gesuchten Intervall ein Dreieck konstruiert, mit den beiden Intervallgrenzen als Grundlinie und dem Funktionswert in der Mitte ($x = (x1 + x2) / 2$, $y = f(x)$) als Spitze. Die Fläche dieses Dreiecks lässt sich einfach ausrechnen.
Um die Fläche genauer zu ermitteln, nimmt man jeweils die Strecken zwischen der Dreiecksspitze und den beiden anderen Punkten als neue Grundlinien und konstruiert darauf ebenfalls Dreiecke mit Spitze in der x-Mitte und dem dazugehörigen Funktionswert (d.h. die Spitze liegt wieder auf der Kurve) und addiert ihre Flächen zur vorherigen Fläche.



Berechnen von Zeilensummen einer Matrix

Entweder macht man das mit einer Schleife, oder - noch eleganter - als Sequenz:

```
Summe := vector([seq(add(A[i,j], j=1..m), i=1..n)]);
```

Hier ist A die Matrix, der Befehl **add()** addiert die Elemente in Zeile i (von Spalte 1 bis m), und **seq()** lässt i von der 1. bis zur n-ten Zeile laufen.

Oder man multipliziert die Matrix mit einem Vektor, der nur aus Einsen besteht, und soviele Elemente hat wie die Matrix Spalten (nachrechnen!):

```
evalm(A &* vector([1$coldim(A)]) );
```

Testen, ob eine Matrix ein magisches Quadrat ist (wie auf Blatt 8)

Bei einem magischen Quadrat ist die Spalten- und die Zeilensumme überall gleich.

1. Man ermittelt die Spalten- oder Zeilensumme der ersten Spalte bzw. Zeile (ist egal), und speichert sie in einer Variablen.

2. In einer Schleife testet man, ob die Zeilensummen ab der 2. Zeile gleich diesem Wert sind, und auch die Spaltensummen ab der 2. Spalte (die erste Spaltensumme haben wir ja schon berechnet, und die erste Zeilensumme muss richtig sein, weil alle Elemente in dieser Zeile bereits in anderen Summen drin sind, die auch alle stimmen müssen - bzw. Spaltensumme, wenn man zuerst die erste Zeilensumme berechnet hat usw.). Sobald eine Summe nicht stimmt, ist es kein magisches Quadrat mehr!

Rekursive Umrechnung in ein anderes Zahlensystem

```
if n=0 then []
else
        [ op( binaer( trunc( n/basis ) ) ), n mod basis ]
fi
```

Hier wird also eine Liste erzeugt, als zweiten Eintrag die umzurechnende Zahl modulo die Zahlenbasis (= der Rest der Division), und als ersten Eintrag die Sequenz, die mit **op()** aus der Liste gemacht wird, die der rekursive Aufruf der Funktion (hier lautet ihr Name *binaer*) (als Parameter der Ganzzahlteil der durch die Basis geteilten Zahl) zurückliefert.