► ComputerMathematik - Einführung in Maple / Teil IV

Winfried Auzinger (SS 2011)

Teil IV stellt eines der wichtigsten Packages vor, nämlich

LinearAlgebra für [Numerische] Lineare Algebra,

plus Beispiele und ein paar Ergänzungen (Numerik, Grafik).

► 1. Vektoren und Matrizen

► 2. Das Paket LinearAlgebra

► 3. Beispiele zur Programmierung

► 4. Spezielle Datenstrukturen

► Anhang: Differentialgleichungen; packages numapprox und plots

ComputerMathematik - Einführung in Maple / Teil IV

Winfried Auzinger (SS 2011)

Teil IV stellt eines der wichtigsten Packages vor, nämlich

LinearAlgebra für [Numerische] Lineare Algebra,

plus Beispiele und ein paar Ergänzungen (Numerik, Grafik).

1. Vektoren und Matrizen

Für den Bereich der Linearen Algebra verwendet Maple zwei Varianten
des Array-Typs (intern basierend auf **rtables** = 'rectangular tables'): **Vector** und **Matrix** .

ACHTUNG... **keine** 'last-name-evaluation' (wie z.B. bei tables) - siehe Beispiel:

```
> restart;
```

```
> ? rtable
```

```
> t1:=table([[1,b]]); t2:=t1;
  t2=eval(t2);
```

$$t1 := table([1 = [1, b]])$$

$$t2 := t1$$

$$t1 = table([1 = [1, b]])$$

```
> rt1:=rtable([[a,b],[1]]); rt2:=rt1;
  rt2=eval(rt2);
```

$$rt1 := \begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix}$$

$$rt2 := \begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix}$$

-- **rtables** sind die 'Basis-Datenstruktur' für Arrays, Vektoren und Matrizen,etc.
(optional mit effizienter interner Speicherorganisation für spezielle Fälle wie
symmetrisch, Bandamatrix, datendünne Matrix, etc.). In der Praxis verwendet man
meist die speziellen rtable-basierten Typen **Array**,
für Lineare Algebra insbesondere **Vector** und **Matrix** (Indizierung beginnt fix mit 1).

(Anmerkung: Listen und tables sind für algebraische Berechnungen im allgemeinen nicht gut
geeignet.)

```
> x:=Vector([a,b,c]); # Vector()-Konstruktor erwartet Liste
```


$$x := \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

```
> whattype(x), type(x,rtable);
```

$$Vector_{column}, true$$

-- Es werden also (ähnlich wie in Matlab) **Zeilen- und Spaltenvektoren**
unterschieden, gemäß dem üblichen Matrix-Vektor-Kalkül.
Default = Spaltenvektor (column)

```
> Vector[column]([a,b,c]), Vector[row]([1,2,3]);
```

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix}, \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

-- Alternative Syntax:

```
> <1,2,3>; #   , steht für 'neue Zeile'
```

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

```
> <alpha|beta|delta>; #   |  steht für 'neue Spalte'
```

$$\begin{bmatrix} \alpha & \beta & \delta \end{bmatrix}$$

-- **Matrix-Typ** mit analoger Syntax (Spezifikation mittels listlist):

```
> A:=Matrix([[a,b,c],
             [d,e,f]]); # listlist wird zeilenweise interpretiert
```

$$A := \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$$

-- Alternative Syntax:

```
> <<a|b|c>,<d|e|f>>;   # zeilenweise
```

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$$

```
> <<a,d>|<b,e>|<c,f>>; # spaltenweise
```

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$$

-- Vektorelement, Teilvektor:

```
> x, x[2], x[1..2];
```

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix}, b, \begin{bmatrix} a \\ b \end{bmatrix}$$

-- Matrixelement, Teilmatrix:

```
> A, A[2,3], A[1..2,2..3];
```

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}, f, \begin{bmatrix} b & c \\ e & f \end{bmatrix}$$

-- Zeile, Spalte einer Matrix:

```
> LinearAlgebra[Row](A,1), LinearAlgebra[Column](A,2);
```

$$\begin{bmatrix} a & b & c \end{bmatrix}, \begin{bmatrix} b \\ e \end{bmatrix}$$

-- Diverse Operationen und Transformationen aus der [Numerischen] Linearen Algebra bedinden sich im package **LinearAlgebra,** siehe Abschnitt 2.

Es gibt auch **linalg** (andere Datenstrukturen, weniger mächtig, veraltet).

-- 'Leere' Objekte erzeugen (Initialisierung mit Nullelementen)

```
> Vector(3), Matrix(1..3,2);
```

$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

-- Angabe einer `**generating function**' (i,j)->f(i,j), die die Einträge definiert:

```
> f:=(i,j)->i+j; B:=Matrix(2,4,f);
```

$$f := (i, j) \rightarrow i + j$$

$$B := \begin{bmatrix} 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \end{bmatrix}$$

-- Auch 'Lego spielen' kann man:

```
> x; B:=Matrix([x,x]);
```

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

$$B := \begin{bmatrix} a & a \\ b & b \\ c & c \end{bmatrix}$$

... + weitere syntaktische Varianten...

-- **Elementare Arithmetik:** (**Punkt .** für MV bzw. MM-Multiplikation, nicht * !):

```
> B, x, B.x[1..2];
```

$$\begin{bmatrix} a & a \\ b & b \\ c & c \end{bmatrix}, \begin{bmatrix} a \\ b \\ c \end{bmatrix}, \begin{bmatrix} a^2 + a\,b \\ a\,b + b^2 \\ c\,a + c\,b \end{bmatrix}$$

```
> B+2*B, B[1..2,1..2].B[1..2,1..2];
```

$$\begin{bmatrix} 3\,a & 3\,a \\ 3\,b & 3\,b \\ 3\,c & 3\,c \end{bmatrix}, \begin{bmatrix} a^2 + a\,b & a^2 + a\,b \\ a\,b + b^2 & a\,b + b^2 \end{bmatrix}$$

-- Zu beachten:

Gemäß dem zugrundeliegenden, sehr flexiblen Datentyp **rtable** können Objekten vom Typ **Vector** und **Matrix** verschiedene Eigenschaften zugewiesen werden (z.B. Symmetrie, `sparsity'), die auch auf die Art der Speicherung einen Einfluss haben (später). Siehe z.B.

```
> ? shape
> ? storage
```

Für nicht zu große Objekte braucht man sich darum jedoch im Allgemeinen nicht weiter zu kümmern.

2. Das Paket LinearAlgebra

Bemerkung: Es gibt auch ein älteres Paket **linalg** (weniger umfangreich).

```
> restart;
> with(LinearAlgebra);
```

[*&x, Add, Adjoint, BackwardSubstitute, BandMatrix, Basis, BezoutMatrix, BidiagonalForm, BilinearForm, CARE, CharacteristicMatrix, CharacteristicPolynomial, Column, ColumnDimension, ColumnOperation, ColumnSpace, CompanionMatrix, ConditionNumber, ConstantMatrix, ConstantVector, Copy, CreatePermutation, CrossProduct, DARE, DeleteColumn, DeleteRow, Determinant, Diagonal, DiagonalMatrix, Dimension, Dimensions, DotProduct, EigenConditionNumbers, Eigenvalues, Eigenvectors, Equal, ForwardSubstitute, FrobeniusForm, GaussianElimination, GenerateEquations, GenerateMatrix, Generic, GetResultDataType, GetResultShape, GivensRotationMatrix, GramSchmidt, HankelMatrix, HermiteForm, HermitianTranspose, HessenbergForm, HilbertMatrix, HouseholderMatrix, IdentityMatrix, IntersectionBasis, IsDefinite, IsOrthogonal, IsSimilar, IsUnitary, JordanBlockMatrix, JordanForm, KroneckerProduct, LA_Main, LUDecomposition,*

LeastSquares, LinearSolve, LyapunovSolve, Map, Map2, MatrixAdd, MatrixExponential, MatrixFunction, MatrixInverse, MatrixMatrixMultiply, MatrixNorm, MatrixPower, MatrixScalarMultiply, MatrixVectorMultiply, MinimalPolynomial, Minor, Modular, Multiply, NoUserValue, Norm, Normalize, NullSpace, OuterProductMatrix, Permanent, Pivot, PopovForm, QRDecomposition, RandomMatrix, RandomVector, Rank, RationalCanonicalForm, ReducedRowEchelonForm, Row, RowDimension, RowOperation, RowSpace, ScalarMatrix, ScalarMultiply, ScalarVector, SchurForm, SingularValues, SmithForm, StronglyConnectedBlocks, SubMatrix, SubVector, SumBasis, SylvesterMatrix, SylvesterSolve, ToeplitzMatrix, Trace, Transpose, TridiagonalForm, UnitVector, VandermondeMatrix, VectorAdd, VectorAngle, VectorMatrixMultiply, VectorNorm, VectorScalarMultiply, ZeroMatrix, ZeroVector, Zip]

2.1 Grundsätzliches zu Verwendung von LinearAlgebra

```
> A:=Matrix([[1,2,3],[1,a,b]]);
  Dimension(A);
```

$$A := \begin{bmatrix} 1 & 2 & 3 \\ 1 & a & b \end{bmatrix}$$

$$2, 3$$

```
> RowDimension(A), ColumnDimension(A);
```

$$2, 3$$

```
> Rank(A); # 'generischer' Rang!
```

$$2$$

-- Viele Befehle haben lange Namen, daher sind hier Abkürzungen ganz praktisch (**alias**):

```
> alias(T=Transpose): T(A);
```

$$\begin{bmatrix} 1 & 1 \\ 2 & a \\ 3 & b \end{bmatrix}$$

-- **LinearAlgebra** eignet sich sowohl für **symbolische** als auch für **numerische** bzw. auch **gemischte** Berechnungen.

```
> B := A.T(A);
```

$$B := \begin{bmatrix} 14 & 1+2a+3b \\ 1+2a+3b & 1+a^2+b^2 \end{bmatrix}$$

```
> Determinant(B); B^(-1); # generisch!
```

$$13+10a^2+5b^2-4a-6b-12ab$$

$$\begin{bmatrix} \frac{1+a^2+b^2}{13+10a^2+5b^2-4a-6b-12ab} & -\frac{1+2a+3b}{13+10a^2+5b^2-4a-6b-12ab} \\ -\frac{1+2a+3b}{13+10a^2+5b^2-4a-6b-12ab} & \frac{14}{13+10a^2+5b^2-4a-6b-12ab} \end{bmatrix}$$

```
> C := T(A).A;
```

$$C := \begin{bmatrix} 2 & 2+a & 3+b \\ 2+a & 4+a^2 & 6+ab \\ 3+b & 6+ab & 9+b^2 \end{bmatrix}$$

```
> Determinant(C);  # diese Matrix ist 'generisch singulär'
                   # (unabh. von a,b)
  C^(-1);
```

$$0$$

Error, (in rtable/Power) singular matrix

-- Für **numerische Berechnungen** gibt es zwei Varianten.

```
> UseHardwareFloats;
```

deduced

Der Wert der Umgebungsvariablen **UseHardwareFloats** steuert gemeinsam mit der eingestellten Rechengenauigkeit **Digits** das Verhalten:

```
> UseHardwareFloats := false;
  Digits := 20;
```

UseHardwareFloats := *false*

Digits := 20

```
> x:=Vector([1/3.0,1/2.0]);
```

$$x := \begin{bmatrix} 0.33333333333333333333 \\ 0.50000000000000000000 \end{bmatrix}$$

```
> x.x; # inneres Produkt auf 20 Stellen genau berechnet
```

$$0.36111111111111111111$$

(Aber Achtung: # der angezeigten Stellen (hier: 10) einzustellen über **Tools/Options**

-- Oder:

```
> UseHardwareFloats := true;
```

UseHardwareFloats := *true*

Dies bedeutet, dass die maschineninterne **doppelt genaue IEEE-Arithmetik** wird (binär, ca. 16 Dezimalstellen, wie in Matlab!)

```
> x:=Vector([1/3.0,1/2.0]);
```

$$x := \begin{bmatrix} 0.33333333333333333333 \\ 0.50000000000000000000 \end{bmatrix}$$

Oder konsequenter auch mit 'IEEE-Daten':
(**evalhf** steht für 'evaluate hardware float' (= IEEE double),
nicht zu verwechseln mit evalf)

(Beachte den Unterschied zu vorher - weniger als die 20 angezeigten Stellen sind signifikant:)

```
> x := map(evalhf,Vector([1/3,sqrt(2)]));
```

$$x := \begin{bmatrix} 0.333333333333333315 \\ 1.41421356237309515 \end{bmatrix}$$

-- Zur Kontrolle, was intern passiert:

```
> infolevel[LinearAlgebra] := 1;
```

$$infolevel_{LinearAlgebra} := 1$$

```
> x.x; #  hier werden 'Hardware-Routinen' aktiviert
DotProduct: calling external function
DotProduct: NAG hw_f06eaf
```

$$2.11111111111111160$$

```
> UseHardwareFloats := false;
```

$$UseHardwareFloats := false$$

```
> y:=Vector([1.0,2.0]); y.y;
              #  hier werden intern 'Software-Routinen'
  aktiviert
```

$$y := \begin{bmatrix} 1.0 \\ 2.0 \end{bmatrix}$$

```
DotProduct: calling external function
DotProduct: NAG sw_f06eaf
```

$$5.00$$

-- ANMERKUNG: In Prozeduren kann man generell mittels **option hfloat** die Verwendung der Hardware-Arithmetik **erzwingen**.

```
> restart: with(LinearAlgebra): UseHardwareFloats;
                                   # default deduced='automatisch'
```

$$deduced$$

2.2 Symbolisches und exaktes Lösen von Problemen

(anhand von Beispielen)

-- Ein lineares Gleichungssystem:

```
> n := 4;
```



```
A := Matrix(n,n,(i,j)->alpha*i-j);
```

$$n := 4$$

$$A := \begin{bmatrix} \alpha-1 & \alpha-2 & \alpha-3 & \alpha-4 \\ 2\alpha-1 & 2\alpha-2 & 2\alpha-3 & 2\alpha-4 \\ 3\alpha-1 & 3\alpha-2 & 3\alpha-3 & 3\alpha-4 \\ 4\alpha-1 & 4\alpha-2 & 4\alpha-3 & 4\alpha-4 \end{bmatrix}$$

```
> b := Vector([seq(i,i=1..n)]);
```

$$b := \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

```
> LinearSolve(A,b); # i.Allg. 2-dimensionale Lösungsschar
                    # mit freien Parametern, die automatisch
                    # durch Variablen benannt werden
```

$$\begin{bmatrix} \frac{_t_3\,\alpha+2\,_t_4\,\alpha+2}{\alpha} \\ -\frac{2\,_t_3\,\alpha+3\,_t_4\,\alpha+1}{\alpha} \\ _t_3 \\ _t_4 \end{bmatrix}$$

```
> simplify(subs([_t[3]=1,_t[4]=2],%));
```

$$\begin{bmatrix} \frac{5\,\alpha+2}{\alpha} \\ -\frac{8\,\alpha+1}{\alpha} \\ 1 \\ 2 \end{bmatrix}$$

```
> alpha:=0; A;
```

$$\alpha := 0$$

$$\begin{bmatrix} -1 & -2 & -3 & -4 \\ -1 & -2 & -3 & -4 \\ -1 & -2 & -3 & -4 \\ -1 & -2 & -3 & -4 \end{bmatrix}$$

```
> LinearSolve(A,b); # jetzt ist die Matrix singulär!
Error, (in LinearAlgebra:-LinearSolve) inconsistent system
> n:=20; A:=Matrix(n,n); # größere Objekte werden nicht voll
  angezeigt
```

$$n := 20$$

$$A := \begin{bmatrix} \textit{20 x 20 Matrix} \\ \textit{Data Type: anything} \\ \textit{Storage: rectangular} \\ \textit{Order: Fortran_order} \end{bmatrix}$$

-- **Eigenproblem und Jordan'sche Normalform:**

```
> A := Matrix([[c,a,c],[d,c,d],[c,d,c]]);
```

$$A := \begin{bmatrix} c & a & c \\ d & c & d \\ c & d & c \end{bmatrix}$$

```
> Eigenvalues(A); # Ausgabe als Spaltenvektor
```

$$\begin{bmatrix} 0 \\ \frac{3}{2}c + \frac{1}{2}\sqrt{c^2+4da+4d^2} \\ \frac{3}{2}c - \frac{1}{2}\sqrt{c^2+4da+4d^2} \end{bmatrix}$$

```
> Eigenvectors(A); # Ausgabe als Spaltenvektor, und
  Eigenvektoren als Matrix (spaltenweise)
```

$$\begin{bmatrix} \frac{3}{2}c + \frac{1}{2}\sqrt{c^2+4da+4d^2} \\ \frac{3}{2}c - \frac{1}{2}\sqrt{c^2+4da+4d^2} \\ 0 \end{bmatrix}, \Bigg[\Bigg[\Big(da^2 + d^2a + ca\Big(\frac{3}{2}c + \frac{1}{2}\sqrt{c^2+4da+4d^2}\Big)$$

$$-c^2a + c\Big(\frac{3}{2}c + \frac{1}{2}\sqrt{c^2+4da+4d^2}\Big)d - dc^2\Big) \Big/ \Big(\Big(\Big(\frac{3}{2}c$$

$$+ \frac{1}{2}\sqrt{c^2+4da+4d^2}\Big)d + ac - dc\Big)\Big(\frac{1}{2}c + \frac{1}{2}\sqrt{c^2+4da+4d^2}\Big)\Big), \Big(da^2$$

$$+d^2a + ca\Big(\frac{3}{2}c - \frac{1}{2}\sqrt{c^2+4da+4d^2}\Big) - c^2a + c\Big(\frac{3}{2}c$$

$$-\frac{1}{2}\sqrt{c^2+4da+4d^2}\Big)d - dc^2\Big) \Big/ \Big(\Big(\Big(\frac{3}{2}c - \frac{1}{2}\sqrt{c^2+4da+4d^2}\Big)d + ac$$

$$-dc\Big)\Big(\frac{1}{2}c - \frac{1}{2}\sqrt{c^2+4da+4d^2}\Big)\Big), -1\Bigg],$$

$$\Bigg[\frac{da + d^2 + c\Big(\frac{3}{2}c + \frac{1}{2}\sqrt{c^2+4da+4d^2}\Big) - 2c^2}{\Big(\frac{3}{2}c + \frac{1}{2}\sqrt{c^2+4da+4d^2}\Big)d + ac - dc},$$

$$\frac{da + d^2 + c\Big(\frac{3}{2}c - \frac{1}{2}\sqrt{c^2+4da+4d^2}\Big) - 2c^2}{\Big(\frac{3}{2}c - \frac{1}{2}\sqrt{c^2+4da+4d^2}\Big)d + ac - dc}, 0\Bigg],$$

$$\Big[1, 1, 1\Big]\Bigg]\Bigg]$$

```
> J := JordanForm(A); # A ist immer diagonalisierbar,
                      # ggfs. mit konj.komplexen Eigenwerten
```

$$J := \begin{bmatrix} 0 & 0 & 0 \\ 0 & \frac{3}{2}c - \frac{1}{2}\sqrt{c^2+4da+4d^2} & 0 \\ 0 & 0 & \frac{3}{2}c + \frac{1}{2}\sqrt{c^2+4da+4d^2} \end{bmatrix}$$

-- Ein typischer Aufruf, bei dem man über **ein Schlüsselwort zusätzliche Information** anfordert, in diesem Fall die Transformations (Eigenvektor-) Matrix:

```
> X := JordanForm(A,output='Q');
```

$$X := \Bigg[\Bigg[-\frac{c^2-d^2}{-2c^2+d^2+da},$$

$$\frac{1}{2}\,\frac{da\sqrt{c^2+4da+4d^2} - c^2\sqrt{c^2+4da+4d^2} + cda + c^3 - 2d^2c}{\sqrt{c^2+4da+4d^2}\,(-2c^2+d^2+da)},$$

$$\frac{1}{2}\,\frac{-c^3 - c^2\sqrt{c^2+4da+4d^2} - cda + da\sqrt{c^2+4da+4d^2} + 2d^2c}{\sqrt{c^2+4da+4d^2}\,(-2c^2+d^2+da)}\Bigg],$$

$$\left[0, -\frac{d}{\sqrt{c^2+4\,d\,a+4\,d^2}}, \frac{d}{\sqrt{c^2+4\,d\,a+4\,d^2}}\right],$$

$$\left[\frac{c^2-d^2}{-2\,c^2+d^2+d\,a},\right.$$

$$-\frac{1}{2}\,\frac{c^2\sqrt{c^2+4\,d\,a+4\,d^2}-d^2\sqrt{c^2+4\,d\,a+4\,d^2}-c^3+2\,c\,d\,a-d^2\,c}{\sqrt{c^2+4\,d\,a+4\,d^2}\,\left(-2\,c^2+d^2+d\,a\right)},$$

$$\left.-\frac{1}{2}\,\frac{c^3+c^2\sqrt{c^2+4\,d\,a+4\,d^2}+d^2\,c-d^2\sqrt{c^2+4\,d\,a+4\,d^2}-2\,c\,d\,a}{\sqrt{c^2+4\,d\,a+4\,d^2}\,\left(-2\,c^2+d^2+d\,a\right)}\right]\Bigg]$$

```
> ? JordanForm
```

-- Anderes Beispiel:

```
> A:=Matrix([[1,epsilon],[0,1]]);
```

$$A := \begin{bmatrix} 1 & \varepsilon \\ 0 & 1 \end{bmatrix}$$

```
> JordanForm(A); # im allgemeinen ('generisch') nicht
  diagonalisierbar
                # (offenbar für epsilon<>0)
```

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

-- ABER: Für epsilon=0 ist A diagonalisierbar!

```
> epsilon:=0; JordanForm(A);
```

$$\varepsilon := 0$$

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

-- Singulärwerte (bzw. volle Singuläwertzerlegung):

```
> SingularValues(HilbertMatrix(2));
```

$$\begin{bmatrix} \frac{2}{3}+\frac{1}{6}\sqrt{13} \\ \frac{2}{3}-\frac{1}{6}\sqrt{13} \end{bmatrix}$$

-- Einige weitere Transformationen der Linearen Algebra:

```
> V := VandermondeMatrix([xi[1],xi[2],xi[3]]);
```


$$V := \begin{bmatrix} 1 & \xi_1 & \xi_1^2 \\ 1 & \xi_2 & \xi_2^2 \\ 1 & \xi_3 & \xi_3^2 \end{bmatrix}$$

-- Eine spezielle Ähnlichkeitstransformation, die in numerischen Algorithmen verwendet wird:

```
> HessenbergForm(V); # das geht nicht symbolisch,
                     # dieser Algorithmus benötigt
                     # explizite numerische Vergleiche!
Error, (in LinearAlgebra:-HessenbergForm) Matrix does not evaluate to floating point
```

-- Aber:

```
> V:=VandermondeMatrix([1/2,2/3,1]);
```

$$V := \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{4} \\ 1 & \frac{2}{3} & \frac{4}{9} \\ 1 & 1 & 1 \end{bmatrix}$$

```
> UseHardwareFloats := false;
```

$$UseHardwareFloats := false$$

```
> HessenbergForm(V); # rechnet automatisch in floating point
                     # obwohl Matrixelemente rational
```

$$\begin{bmatrix} 1. & -0.5303300860 & -0.1767766953 \\ -1.414213562 & 1.555555554 & 0.1111111109 \\ 0. & -0.4444444438 & 0.1111111112 \end{bmatrix}$$

```
> UseHardwareFloats := true;
```

$$UseHardwareFloats := true$$

```
> HessenbergForm(V); # double-Version
```

$$\begin{bmatrix} 1. & -0.530330085889910 & -0.176776695296637 \\ -1.41421356237310 & 1.55555555555555 & 0.111111111111111 \\ 0. & -0.444444444444445 & 0.111111111111111 \end{bmatrix}$$

```
> H := HilbertMatrix(3);
```

$$H := \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{bmatrix}$$

-- **Orthogonalisierung** der Spalten (geht auch exakt):

```
> GramSchmidt([seq(Column(H,i),i=1..RowDimension(H))]);
```

$$\left[\begin{bmatrix} 1 \\ \frac{1}{2} \\ \frac{1}{3} \end{bmatrix}, \begin{bmatrix} -\frac{5}{98} \\ \frac{17}{294} \\ \frac{13}{196} \end{bmatrix}, \begin{bmatrix} \frac{1}{2190} \\ -\frac{1}{365} \\ \frac{1}{365} \end{bmatrix} \right]$$

-- **Matrix-Exponentialfunktion** (basiert auf Jordan-Zerlegung):

```
> C:=CompanionMatrix('x^2+rho*x','x');
```

$$C := \begin{bmatrix} 0 & 0 \\ 1 & -\rho \end{bmatrix}$$

```
> MatrixExponential(C); # offenbar ist rho=0 Sonderfall!
```

$$\begin{bmatrix} 1 & 0 \\ -\frac{e^{-\rho}-1}{\rho} & e^{-\rho} \end{bmatrix}$$

-- **QR-Zerlegung** (verwandt mit Gram-Schmidt):

```
> QRDecomposition(C);
```

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 & -\rho \end{bmatrix}$$

... und ... und ...

Grundprinzip:

-- Symbolisch exakte Lösungen werden versucht, wenn der zugrunde liegende Algorithmus dies grundsätzlich zulässt.

-- Numerische Ergebnisse in exakter Arithmetik möglich, aber manche Algorithmen, die nur in der Numerischen linearen Algebra verwendet werden, sind nur als Gleitpunkt-Versionen implementiert (z.B. HessenbergForm).

Kompetente Verwendung erfordert teilweise Kenntnisse über [Numerische] Lineare Algebra.
UE: Ggf. werden Hinweise zu den Beispielen gegeben.

```
> restart: with(LinearAlgebra): UseHardwareFloats:=true:
```

2.3 Numerische Lineare Algebra, mit Beispielen

Mit der Einstellung **UseHardwareFloats:=true** bietet Maple/LinearAlgebra einen Funktionsumfang wie Matlab, teilweise auch darüber hinaus; Implementierung teilweise weniger effizient (?) - für 'nicht zu große' Probleme egal.

EinigeBeispiele:

-- **Eigenproblem** für eine symmetrische 20x20 - Matrix:

```
> H := map(evalhf,HilbertMatrix(20)); H[1..3,1..3];
```

$$H := \begin{bmatrix} \text{20 x 20 Matrix} \\ \text{Data Type: anything} \\ \text{Storage: rectangular} \\ \text{Order: Fortran_order} \end{bmatrix}$$

$$\begin{bmatrix} 1. & 0.500000000000000000 & 0.333333333333333315 \\ 0.500000000000000000 & 0.333333333333333315 & 0.250000000000000000 \\ 0.333333333333333315 & 0.250000000000000000 & 0.200000000000000010 \end{bmatrix}$$

```
> Eigenvalues(H);%[1];
```

$$\begin{bmatrix} \text{1 .. 20 } \mathit{Vector}_{column} \\ \text{Data Type: } \mathit{complex}_8 \\ \text{Storage: rectangular} \\ \text{Order: Fortran_order} \end{bmatrix}$$

$$1.90713472040725 + 0.\,I$$

```
> JordanForm(H); # dies ist numerisch nicht sinnvoll definiert!
Error, (in LinearAlgebra:-JordanForm) Jordan form not defined for a Matrix with floating-point values in its entries
```

-- Berechnung einer **Ausgleichsgeraden**, mit Grafik:

```
> n:=7;
```

$$n := 7$$

```
> t := [seq(i,i=1..n)];
  y := map(evalhf,[seq(i^3,i=1..n)]);
```

$$t := [1, 2, 3, 4, 5, 6, 7]$$

$$y := [1., 8., 27., 64., 125., 216., 343.]$$

```
> V := VandermondeMatrix(t,n,2);
```

$$V := \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \\ 1 & 5 \\ 1 & 6 \\ 1 & 7 \end{bmatrix}$$

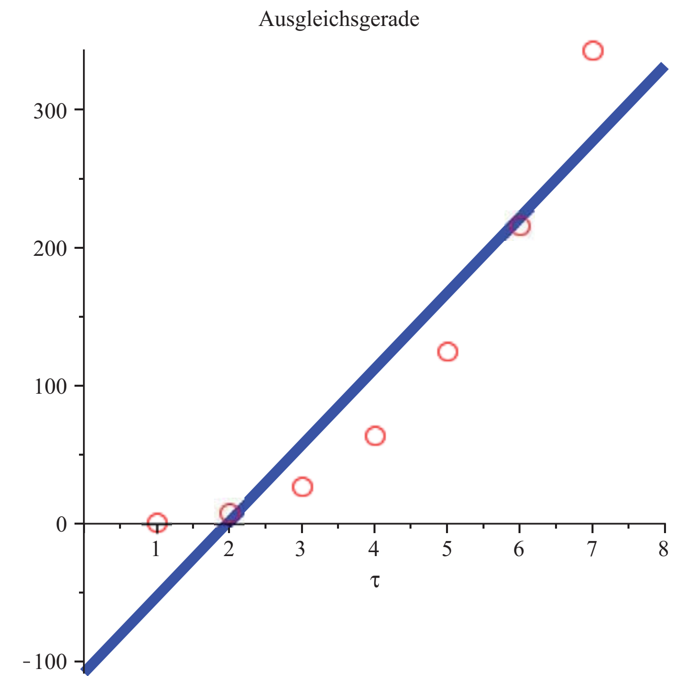
```
> Y := convert(y,Vector):
> LinearSolve(V,Y); # überbestimmt
Error, (in LinearAlgebra:-LinearSolve) inconsistent system
> a := LeastSquares(V,Y); # löst Ausgleichsproblem
```

$$a := \begin{bmatrix} -108. \\ 55. \end{bmatrix}$$

```
> gerade := tau->a[1]+tau*a[2];
```

$$gerade := \tau \rightarrow a_1 + \tau\, a_2$$

```
> p1 := plot(gerade(tau),tau=t[1]-1..t[n]+1,thickness=6,color=
  blue):
> p2 := plots[pointplot]([seq([t[i],y[i]],i=1..n)],
        symbolsize=20,symbol=circle,color=red):
> plots[display](p1,p2,title="Ausgleichsgerade");
```

3. Beispiele zur Programmierung

3.1 Ein allgemeineres lineares Ausgleichsproblem

```
> restart: with(LinearAlgebra):
```

Gegeben sei eine beliebige Funktion

```
> x:='x': f:=x->'f'(x); type(f,procedure), f(x);
```

$$f := x \rightarrow 'f'(x)$$

$$true, f(x)$$

und ein approximierender Ansatz für f als Polynom vom Grad n:

```
> p:=(x,n)->add(c[i]*x^i,i=0..n);
```

$$p := (x, n) \rightarrow add\left(c_i\, x^i, i = 0..n\right)$$

Die Parameter c[i] sind so zu bestimmen, dass folgendes Integral minimal wird:

```
> 'Int((p(xi,n)-f(xi))^2,xi=a..b)';
```

$$\int_a^b (p(\xi, n) - f(\xi))^2 \, d\xi$$

Wir schreiben dafür eine Prozedur, die die Minimalbedingung

'partielle Ableitungen nach den c[i] müssen 0 sein'

mittels LinearAlgebra implementiert:

```
> minimiere:=proc(f::procedure,a,b,n)
  # hier fehlen eigentlich noch die description
  # und ein Kommentar zur Bedeutung der lokalen Variablen...
  local A,ableitung,c,i,k,p,r,sol,zielfunktional;
  # Ansatz für das Polynom:
  p:=xi->add(c[i]*xi^i,i=0..n);
  # Zielfunktional:
  zielfunktional:=int((p(xi)-f(xi))^2,xi=a..b);
  # partiell nach den c[i] differenzieren:
  for i from 0 to n do
     ableitung[i]:=simplify(diff(zielfunktional,c[i]));
    # printf("\n ableitung[%d] = %a",i,ableitung[i]);
  end do;
  # Koeffizientenvergleich in allen Ableitungen
  # nach allen c[k], in Matrix A speichern:
  # Achtung: Indizierung muss in A mit 1 beginnen
  A:=Matrix(n+1,n+1);
  for i from 0 to n do
     for k from 0 to n do
       A[i+1,k+1]:=coeff(ableitung[i],c[k]);
     end do;
  end do;
  # In den Ableitungen sind auch Terme enthalten,
  # in denen die c[k] nicht vorkommen.
  # Diese kommen auf die rechte Seite des Gleichungssystems.
  r:=Vector(n+1);
  for i from 0 to n do
    r[i+1]:=subs(seq(c[k]=0,k=0..n),ableitung[i]);
  end do;
  # Lösen des Gleichungssystems A.c+r=0,
  # zurückspeichern in die c[i] (Indizierung mit 0 beginnend)
  sol:=LinearSolve(A,-r);
  for i from 0 to n do
    c[i]:=sol[i+1];
```



```
  end do;
  # Zurückgegeben wird das bestapproximierende
  # Polynom als Ausdruck in der Variablen xi
  # und der Wert des Zielfunktionals (nochmals ausgewertet)
  zielfunktional:=int((p(xi)-f(xi))^2,xi=a..b);
  return collect(expand(p(xi)),xi),zielfunktional;
  end proc:
```

Ein erster Test: Beliebiges Polynom vom Grad n reproduziert sich selbst:

```
> n:=4;
  minimiere(x->sum('u[i]*x^i','i'=0..n),a,b,n);
```

$$n := 4$$

$$u_0 + u_1 \xi + u_2 \xi^2 + u_3 \xi^3 + u_4 \xi^4, 0$$

Approximation eines Polynoms vom Grad 2 auf [0,1] durch eine Gerade:

```
> minimiere(x->u[0]+u[1]*x+u[2]*x^2,0,1,1);
```

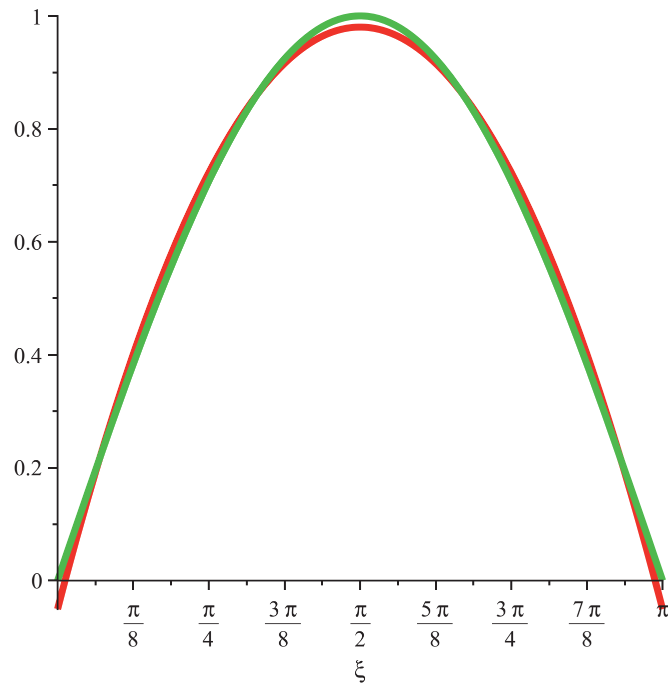
$$-\frac{1}{6} u_2 + u_0 + (u_2 + u_1) \xi, \frac{1}{180} u_2^2$$

Eine numerische Rechnung: Approximation für Sinusfunktion

```
> p,minimum := minimiere(sin,0.0,Pi,2);
```

$$p, minimum := -0.0504655059121495 + 1.31223621992019 \xi - 0.417697761734345 \xi^2,$$
$$0.0009362943259$$

```
> plot([p,sin(xi)],xi=0..Pi,thickness=4);
```



3.2 Eine einfache Matrix-Rekursion:
Berechnung von A^(2^n) = A^(2^(n-1)).A^(2^(n-1)) = ...

```
> restart: with(LinearAlgebra):
> Apower:=proc(A::Matrix,n::nonnegint)
> description "Rekursive Berechnung von A^(2^n)";
> local B;
> if (RowDimension(A)<>ColumnDimension(A)) then
>   error "A muss quadratische Matrix sein"
> end if;
> if n=0 then
     return A
  else
     B:=Apower(A,n-1);
     return B.B
  end if;
```



```
> end proc:
> M := Matrix([[1,2],[3,4]]);
```

$$M := \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

```
> Apower(M,0);
```

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

```
> Apower(M,1), M^2;
```

$$\begin{bmatrix} 7 & 10 \\ 15 & 22 \end{bmatrix}, \begin{bmatrix} 7 & 10 \\ 15 & 22 \end{bmatrix}$$

```
> Apower(M,5), M^(2^5);
```

$$\begin{bmatrix} 55368923250321279115351 & 80696169672298405404130 \\ 121044254508447608106195 & 176413177758768887221546 \end{bmatrix},$$

$$\begin{bmatrix} 55368923250321279115351 & 80696169672298405404130 \\ 121044254508447608106195 & 176413177758768887221546 \end{bmatrix}$$

```
> Apower(Matrix([[a,0,0],[a,a,0],[a,a,a]]),3);
```

$$\begin{bmatrix} a^8 & 0 & 0 \\ 8\,a^8 & a^8 & 0 \\ 36\,a^8 & 8\,a^8 & a^8 \end{bmatrix}$$

3.3 Arnoldi - Iteration

```
> restart: with(LinearAlgebra):
  Digits := 10;
```

$$Digits := 10$$

Die sogenannte Arnoldi-Iteration ist ein algorithmisches Werkzeug,
das bei der iterativen lösung sehr großer, speziell strukturierter Gleichungssysteme
eingesetzt wird.

Man geht aus von einer n*n-Matrix **A** und einem n-Vektor **b**,
und betrachtet den Unterraum **U** des **R^n**, der von den Vektoren

b, A.b, A^2.b, ..., A^m.b

aufgespannt wird. Gesucht ist eine Orthonormalbasis (ONB) in **U**.

Vorgangsweise analog wie bei **Gram-Schmidt:**

-- Normiere **b** --> **v_1**
-- Berechne **A.v_1**
-- Bilde Linearkombination **w_2** von **v_1** und **A.v_1** so
dass **w_2** orthogonal auf **v_1**
-- Normiere **w_2** --> **v_2**
-- Dann ist **{v_1,v_2}** ONB der linearen Hülle von **{b,A.b}**.
-- usw.

Dies führt auf folgende iterative Prozedur:

```
> arnoldi := proc(A::Matrix,b::Vector,m::posint)
  uses LinearAlgebra:
  description "Arnoldi-Iteration":
  local Av,i,ip,j,v,w: # lokale Funktion für inneres Produkt
  ip := (u,v) -> evalf(Transpose(u).v):
  v[1] := evalf(b/Norm(b,2)):
  for i from 2 to m do
   Av := evalf(A.v[i-1]):
   w := evalf(Av - add(ip(Av,v[j])*v[j],j=1..i-1)):
   v[i] := evalf(w/Norm(w,2))
  end do;
  return Matrix([seq(v[j],j=1..m)]):
  end proc:
```

```
> A,b := HilbertMatrix(5),Vector([1,1,1,1,1]);
```

$$A, b := \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

```
> V := arnoldi(A,b,3);
```

$$V := \begin{bmatrix} 0.4472135954 & 0.806819005400000 & 0.376069389300000 \\ 0.4472135954 & 0.129091041800000 & -0.678667834100000 \\ 0.4472135954 & -0.161363800100000 & -0.335050205500000 \\ 0.4472135954 & -0.330795791100000 & 0.115815373300000 \\ 0.4472135954 & -0.443750452000000 & 0.521833253400000 \end{bmatrix}$$

Wir überprüfen die Orhonormalität der **v[i]** :


```
> printf("%+10.9f",Transpose(V).V);
+1.000000000 +0.000000002 -0.000000011
+0.000000002 +1.000000000 -0.000000013
-0.000000011 -0.000000013 +1.000000000
```

... numerisch OK (gerechnet wurde mit 10 Dezimalstellen).

4. Spezielle Datenstrukturen

Der **Matrix** - Typ ist von rtables abgeleitet und unterstützt verschiedenste spezielle Matrixstrukturen und damit zusammenhängend spezielle Speicherungsmuster über sogenannte 'indexing functions' - relevant für große, speziell strukturierte Matrizen (ähnlich für Vekoren).

Algorithmischer umgang analog wie bei 'vollen', normal gespeicherten Matrizen, um die speicher- und rechentechnischen interna kümmert sich Maple.

Einige Beispiele:

```
> ? shape
```

Meist wird der 'storage mode' aus dem 'shape mode abgeleitet.

```
> A:=Matrix(3,3,symbol=a,shape=symmetric);
```

$$A := \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{1,2} & a_{2,2} & a_{2,3} \\ a_{1,3} & a_{2,3} & a_{3,3} \end{bmatrix}$$

```
> op(A);
```

$$3, 3, \{(1,1) = a_{1,1}, (1,2) = a_{1,2}, (1,3) = a_{1,3}, (2,2) = a_{2,2}, (2,3) = a_{2,3}, (3,3) = a_{3,3}\}, datatype = anything, storage = triangular_{upper}, order = Fortran_order, shape = [symmetric]$$

```
> A[2,1]:=2; A;
```

$$A_{2,1} := 2$$

$$\begin{bmatrix} a_{1,1} & 2 & a_{1,3} \\ 2 & a_{2,2} & a_{2,3} \\ a_{1,3} & a_{2,3} & a_{3,3} \end{bmatrix}$$

```
> A:=Matrix(4,4,symbol=a,shape=band[1]); # tridiagonal
```

$$A := \begin{bmatrix} a_{1,1} & a_{1,2} & 0 & 0 \\ a_{2,1} & a_{2,2} & a_{2,3} & 0 \\ 0 & a_{3,2} & a_{3,3} & a_{3,4} \\ 0 & 0 & a_{4,3} & a_{4,4} \end{bmatrix}$$

```
> A.A;
```

$$[[a_{1,1}^2 + a_{1,2} a_{2,1}, a_{1,1} a_{1,2} + a_{1,2} a_{2,2}, a_{1,2} a_{2,3}, 0], [a_{2,1} a_{1,1} + a_{2,2} a_{2,1}, a_{1,2} a_{2,1} + a_{2,2}^2 + a_{2,3} a_{3,2}, a_{2,2} a_{2,3} + a_{2,3} a_{3,3}, a_{2,3} a_{3,4}], [a_{3,2} a_{2,1}, a_{3,2} a_{2,2} + a_{3,3} a_{3,2}, a_{2,3} a_{3,2} + a_{3,3}^2 + a_{3,4} a_{4,3}, a_{3,3} a_{3,4} + a_{3,4} a_{4,4}], [0, a_{4,3} a_{3,2}, a_{4,3} a_{3,3} + a_{4,4} a_{4,3}, a_{3,4} a_{4,3} + a_{4,4}^2]]$$

```
> A[1,4]:=1;
Error, attempt to assign a value outside Matrix bands
> op(A);
```

$4, 4, \{(1,1) = a_{1,1}, (1,2) = a_{1,2}, (2,1) = a_{2,1}, (2,2) = a_{2,2}, (2,3) = a_{2,3}, (3,2) = a_{3,2}, (3,3) = a_{3,3}, (3,4) = a_{3,4}, (4,3) = a_{4,3}, (4,4) = a_{4,4}\}, datatype = anything, storage = band_{1,1}, order = Fortran_order, shape = [band_{1,1}]$

```
> A:=Matrix(5,5,storage=sparse);
```

$$A := \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

```
> A[1,1]:=1; op(A);
```

$$A_{1,1} := 1$$

$5, 5, \{(1,1) = 1\}, datatype = anything, storage = sparse, order = Fortran_order, shape = [\,]$

Anhang: Differentialgleichungen; packages numapprox und plots

Für die Lösung gewöhnlicher und partieller Differentialgleichungen gibt es

-- dsolve

Beispiel:


```
> diffeq := D(u)(t)=u(t)^2;
```

$$diffeq := D(u)(t) = u(t)^2$$

```
> initial_value := u(0)=1;
```

$$initial_value := u(0) = 1$$

```
> dsolve({diffeq,initial_value},u(t));
```

$$u(t) = -\frac{1}{t-1}$$

Neben der Numerischen Linearen Algebra (**LinearAlgebra**) enthält das

-- **numapprox** - package

einige für die Numerische Mathematik relevante Tools.

```
> restart: with(numapprox);
```

[chebdeg, chebmult, chebpade, chebsort, chebyshev, confracform, hermite_pade, hornerform, infnorm, laurent, minimax, pade, remez]

```
> # z.B.: Maximum-Norm einer Funktion:
> infnorm(sin,0..1);
```

$$0.8414709848$$

Das package

-- **plots**

enthält diverse Funktionen für grafische Darstellung.

Einige Beispiele:

```
> with(plots);
```

[animate, animate3d, animatecurve, arrow, changecoords, complexplot, complexplot3d, conformal, conformal3d, contourplot, contourplot3d, coordplot, coordplot3d, densityplot, display, dualaxisplot, fieldplot, fieldplot3d, gradplot, gradplot3d, implicitplot, implicitplot3d, inequal, interactive, interactiveparams, intersectplot, listcontplot, listcontplot3d, listdensityplot, listplot, listplot3d, loglogplot, logplot, matrixplot, multiple, odeplot, pareto, plotcompare, pointplot, pointplot3d, polarplot, polygonplot, polygonplot3d, polyhedra_supported, polyhedraplot, rootlocus, semilogplot, setcolors, setoptions, setoptions3d, spacecurve, sparsematrixplot, surfdata, textplot, textplot3d, tubeplot]

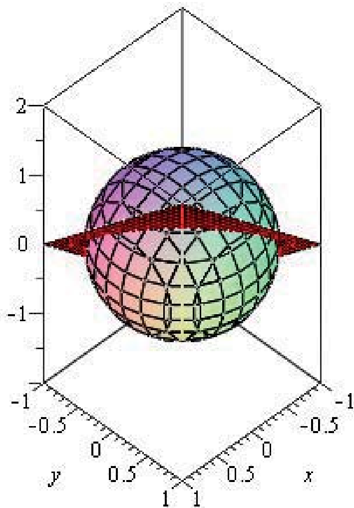
```
> plot1 := plot3d(x+y,x=-1..1,y=-1..1,color=red); # plot-Befehl
  erzeugt Datenstruktur,
                                                  # (eigentlich:
```

```
eine Funktion), die die plot-Daten enthält;
                                                      # wird
zwischengespeichert und Variable zugewiesen
```
$plot1 := PLOT3D(...)$

```
> whattype(plot1);
```
function

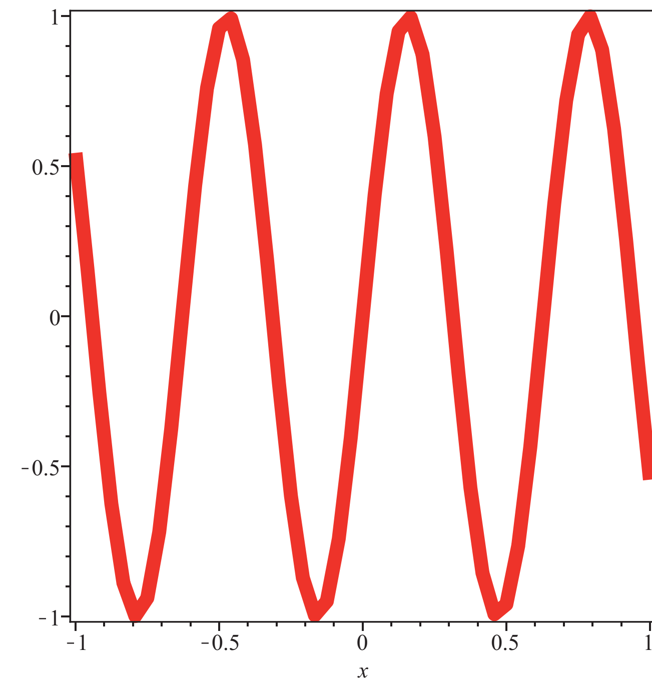
```
> plot2 := implicitplot3d(x^2+y^2+z^2=1,x=-1..1,y=-1..1,z=-1..1);
```
$plot2 := PLOT3D(...)$

```
> display(plot1,plot2,axes=boxed,scaling=constrained);
```

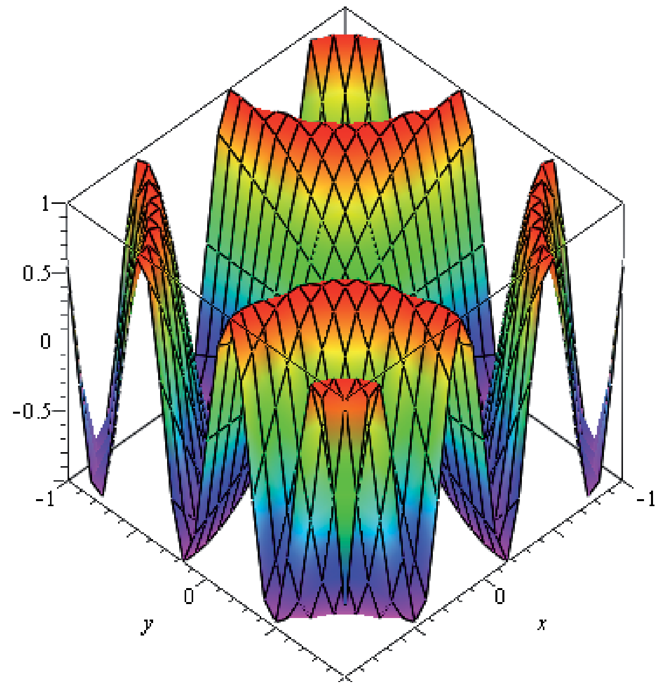


-- **animate** und **animate 3d** dienen zum erstellen kleiner Filme (parameterabhängiger Plot). Abspielen mittels Maussteuerung.

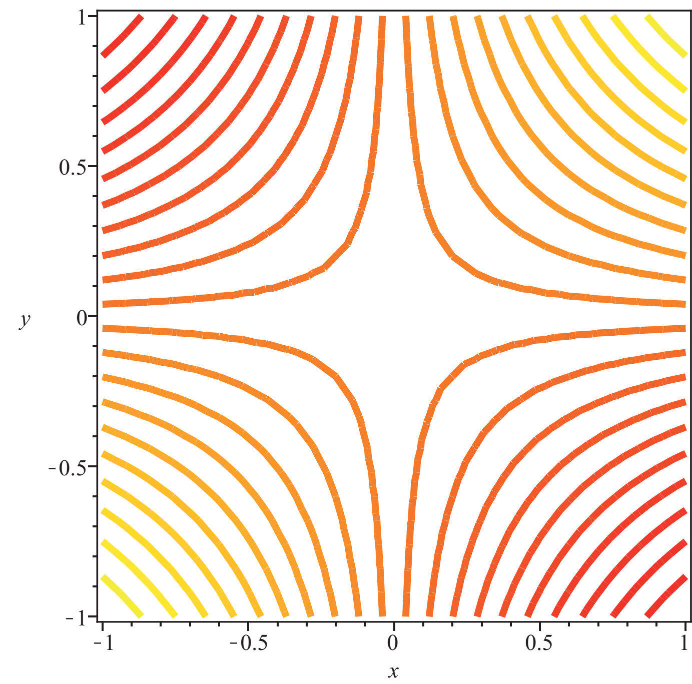
```
> animate((sin(t*x),x=-1..1),t=0..10,thickness=8,axes=boxed);
```


```
> animate3d((sin(t*x*y),x=-1..1,y=-1..1),t=0..10,shading=zhue,
  axes=boxed);
```




```
> contourplot(sin(x*y),x=-1..1,y=-1..1,thickness=4,axes=boxed,
  contours=20);
```

.. und vieles Weitere.

Beachte bei den plots auch auch Kontextmenü und Export-Varianten.

=== Ende Maple Teil IV ===