ComputerMathematik - Einführung in Maple / Teil IV

Winfried Auzinger (SS 2010)

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:=t1;
```

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

$$rt2 := t1$$

rtables sind die 'Basis-Datenstruktur' für Arrays, Vektoren und Matrizen,etc. (mit variabler interner Speicherorganisation,
z.B. für symmetrische, datendünne, ... etc. Fälle). In der Praxis verwendet man **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);
```

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

-- '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: (**.** 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 i.Allg. 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!
```

$$2$$

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

$$T$$

-- **LinearAlgebra** eignet sich sowohl für symbolische als auch für numerische bzw. 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!
```

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

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

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

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

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

$$0$$

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]);
```

$$x := \begin{bmatrix} 0.3333333333 \\ 0.5000000000 \end{bmatrix}$$

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

$$0.3611111111$$

(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]);
```

$$x := \begin{bmatrix} 0.3333333333 \\ 0.5000000000 \end{bmatrix}$$

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)]));
```

$$x := \begin{bmatrix} 0.3333333333 \\ 1.4142135624 \end{bmatrix}$$

-- 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
```

$$2.1111111111$$

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

$$UseHardwareFloats := false$$

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

$$y := \begin{bmatrix} 1.0000000000 \\ 2.0000000000 \end{bmatrix}$$

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

$$5.0000000000$$

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
```

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

```
> 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:-LA_Main:-LinearSolve) inconsistent system
> n:=20; A:=Matrix(n,n); A[1,1];
```

$$n := 20$$

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

$$0$$

-- 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);
```

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

```
> Eigenvectors(A): %[1];
```

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

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

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

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

$$X := \left[\left[-\frac{c^2-d^2}{-2c^2+d^2+da}, \frac{1}{2}\,\frac{da\sqrt{c^2+4da+4d^2}-c^2\sqrt{c^2+4da+4d^2}+cda+c^3-2d^2c}{\sqrt{c^2+4da+4d^2}\,(-2c^2+d^2+da)}, \right.\right.$$

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

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

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

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

Anderes Beispiel:

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

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

```
> JordanForm(A); # i.Allg. 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);
```

$$\varepsilon := 0$$

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

```
> 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]]);
```

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

```
> HessenbergForm(V); # das geht nicht symbolisch,
                     # der Algorithmus benötigt
                     # explizite numerische Vergleiche!
```

Error, (in LinearAlgebra:-LA_Main:-HessenbergForm) Matrix does not evaluate to floating point

Aber:

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

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

```
> HessenbergForm(V); # rechnet automatisch in floating point
                     # rechts oben steht eigentlich 0
```

$$\begin{bmatrix} 1.0000000000 & -1.4142135624 & 2.0000000000\,10^{-30} \\ -1.4142135624 & 9.0000000000 & -4.0000000000 \\ 0.0000000000 & -3.0000000000 & 2.0000000000 \end{bmatrix}$$

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

$$UseHardwareFloats := true$$

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

$$\begin{bmatrix} 1.0000000000 & -1.4142135624 & 2.0274580625\,10^{-17} \\ -1.4142135624 & 9.0000000000 & -4.0000000000 \\ 0.0000000000 & -3.0000000000 & 2.0000000000 \end{bmatrix}$$

```
> 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 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];
```

$$H := \begin{bmatrix} \textit{20 x 20 Matrix} \\ \textit{Data Type: anything} \\ \textit{Storage: rectangular} \\ \textit{Order: Fortran\_order} \end{bmatrix}$$

$$\begin{bmatrix} 1.0000000000 & 0.5000000000 & 0.3333333333 \\ 0.5000000000 & 0.3333333333 & 0.2500000000 \\ 0.3333333333 & 0.2500000000 & 0.2000000000 \end{bmatrix}$$

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

$$\begin{bmatrix} \textit{1 .. 20 Vector}_{column} \\ \textit{Data Type: complex}_{8} \\ \textit{Storage: rectangular} \\ \textit{Order: Fortran\_order} \end{bmatrix}$$

$$1.9071347204 + 0.0000000000\ I$$

```
> JordanForm(H); # numerisch nicht sinnvoll definiert!
Error, (in LinearAlgebra:-LA_Main:-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)]; x:=map(evalhf,[seq(i^3,i=1..n)]);
```

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

$$x := [1.0000000000, 8.0000000000, 27.0000000000, 64.0000000000, 125.0000000000, 216.0000000000, 343.0000000000]$$

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

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

```
> X:=convert(x,Vector);
```

$$X := \begin{bmatrix} 1.0000000000 \\ 8.0000000000 \\ 27.0000000000 \\ 64.0000000000 \\ 125.0000000000 \\ 216.0000000000 \\ 343.0000000000 \end{bmatrix}$$

```
> LinearSolve(V,X);
```

Error, (in LinearAlgebra:-LA_Main:-LinearSolve) inconsistent system

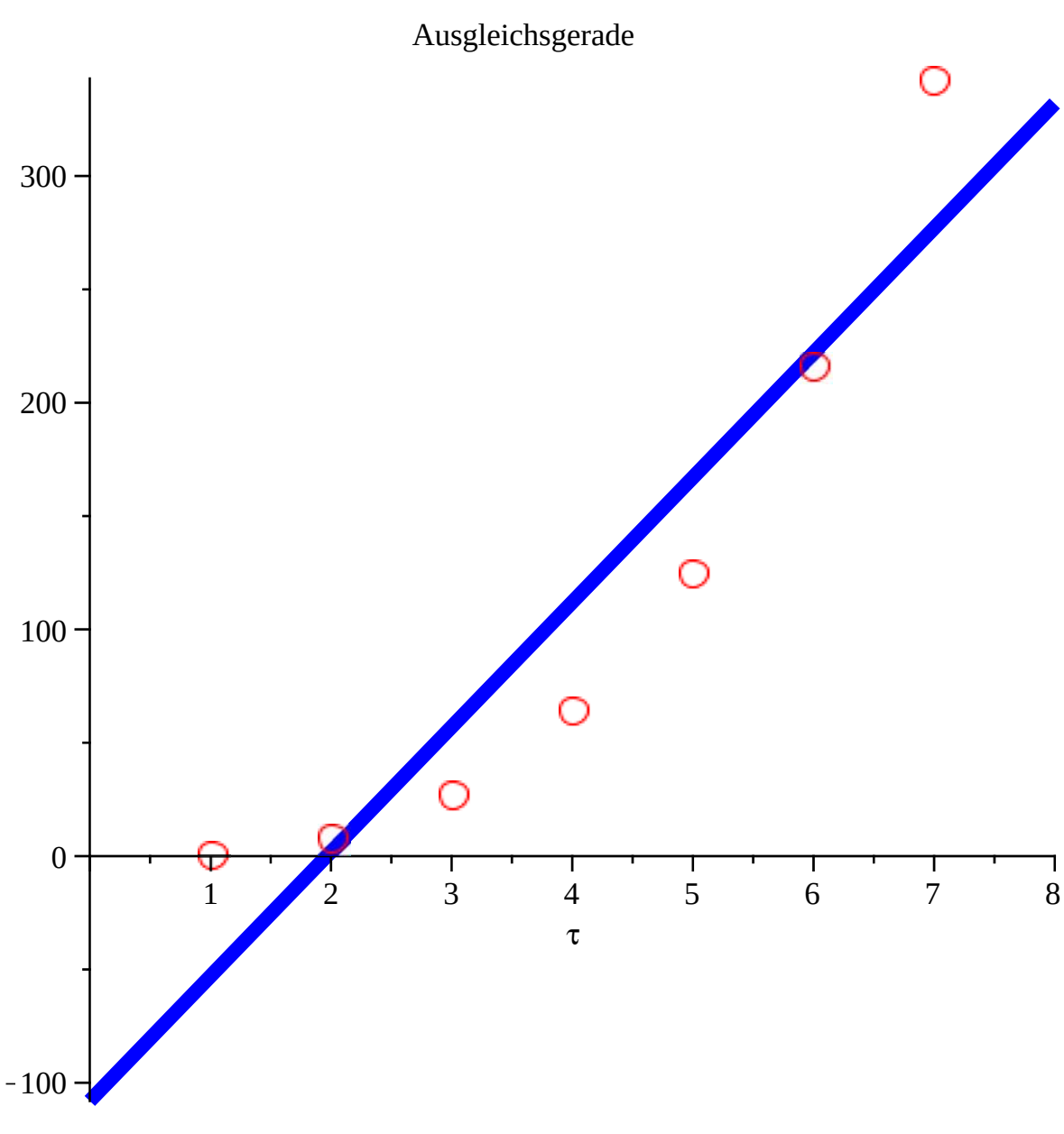
```
> y:=LeastSquares(V,X); # löst Ausgleichsproblem
```

$$y := \begin{bmatrix} -108.0000000000 \\ 55.0000000000 \end{bmatrix}$$

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

$$gerade := \tau \rightarrow y_1 + \tau\, y_2$$

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



3. Programmierbeispiele

3.1 Ein allgemeineres lineares Ausgleichsproblem

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);
```

$$\int_a^b \left(c_0 + c_1\,\xi + c_2\,\xi^2 + c_3\,\xi^3 + c_4\,\xi^4 + c_5\,\xi^5 + c_6\,\xi^6 + c_7\,\xi^7 - f(\xi)\right)^2 d\xi$$

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

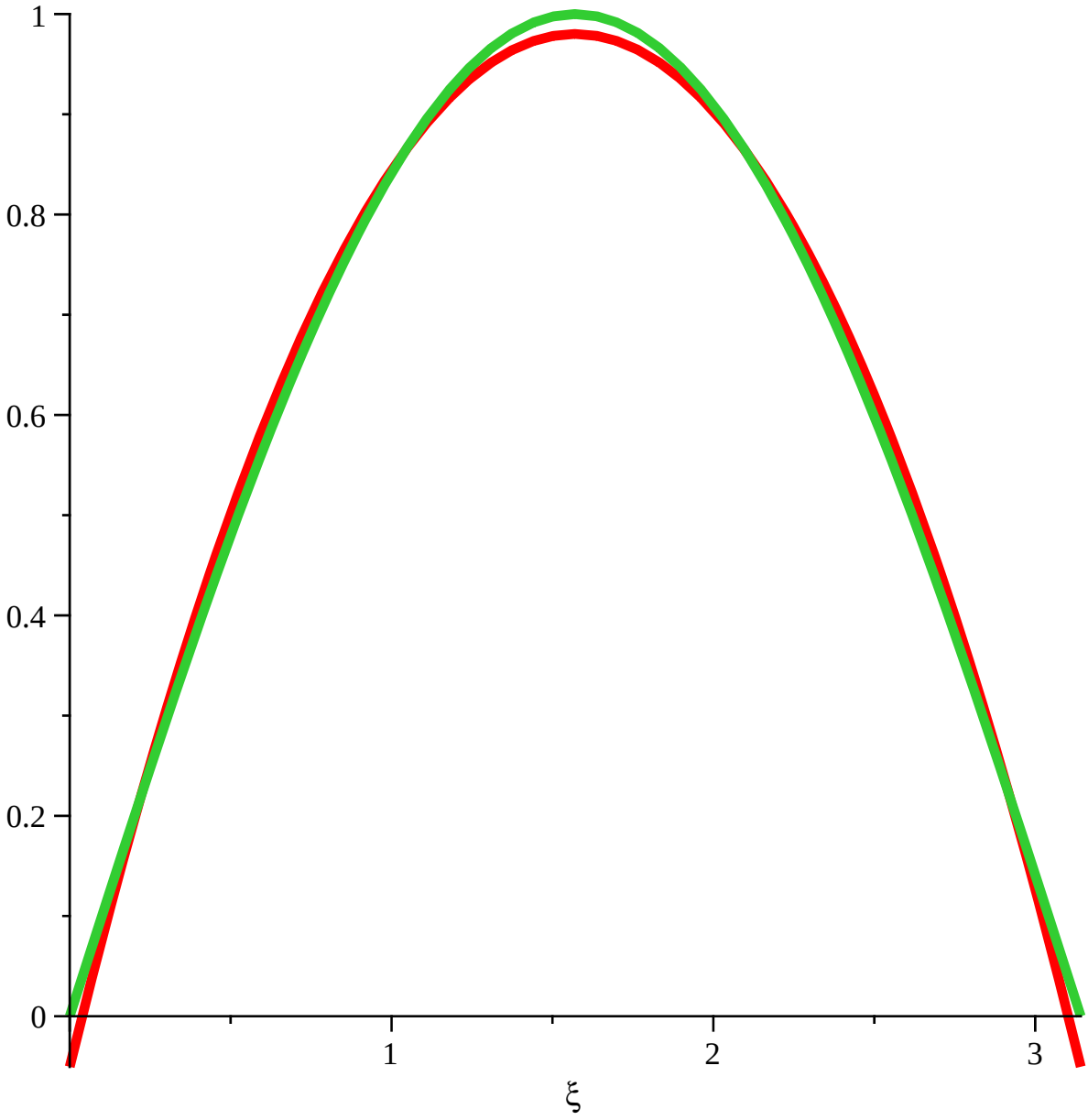
$$-\frac{1}{6}\,u_2+u_0+\left(u_2+u_1\right)\,\xi,\ \frac{1}{180}\,u_2^2$$

Eine numerische Rechnung: Approximation für sin

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

$$p,\ minimum := -0.0504654978+1.3122362048\,\xi-0.4176977570\,\xi^2,\ 0.0009362943$$

```
> 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);
```

$$\begin{bmatrix} 55368923250321279115351 & 8069616967229840540413 0 \\ 12104425450844760810619 5 & 17641317775876888722154 6 \end{bmatrix},$$

$$\begin{bmatrix} 55368923250321279115351 & 8069616967229840540413 0 \\ 12104425450844760810619 5 & 17641317775876888722154 6 \end{bmatrix}$$

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

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

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). Einige Beispiele:

```
> ? shape
```

Meist wird der 'storage mode' aus dem 'shape' 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[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: packages numapprox und plots

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:

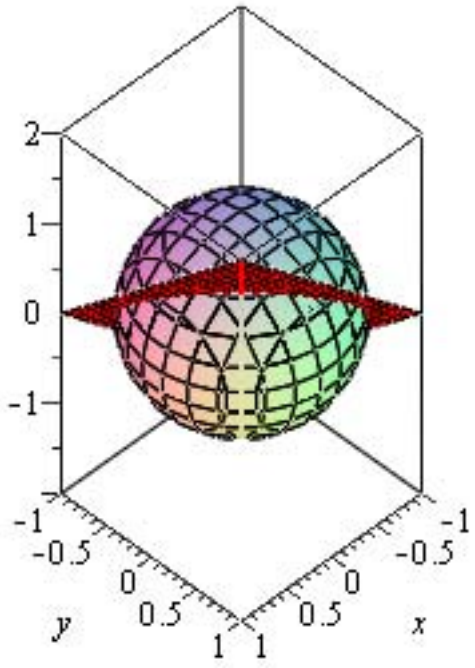
```
> with(plots);
```
[*animate, animate3d, animatecurve, arrow, changecoords, complexplot, complexplot3d, conformal, conformal3d, contourplot, contourplot3d, coordplot, coordplot3d, densityplot, display, dualaxisplot, fieldplot, fieldplot3d, gradplot, gradplot3d, graphplot3d, implicitplot, implicitplot3d, 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); # Funktion, deren
  Ausführung den Plot bewirkt
                                                  # (interne Datenstruktur
  für Grafikdaten)
```
plot1 := *PLOT3D*(...)

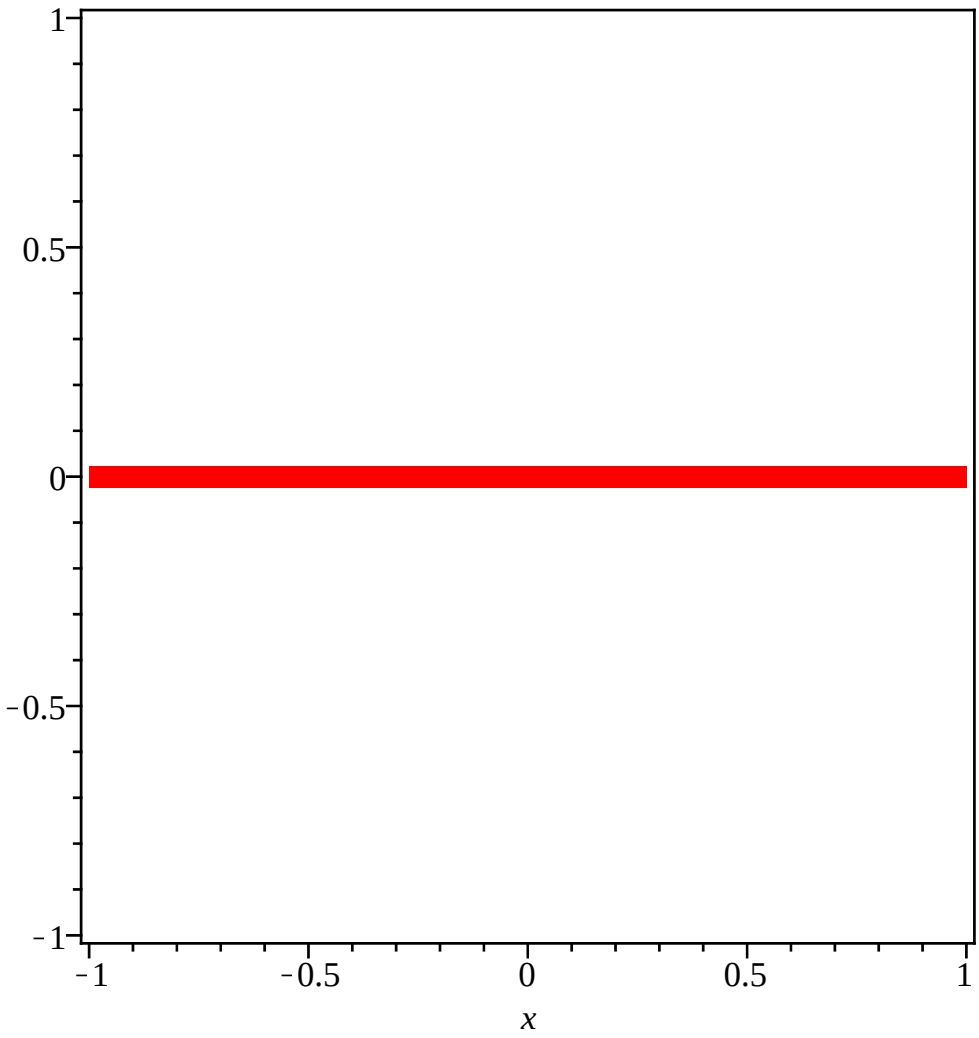
```
> whattype(plot1);
```
function

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

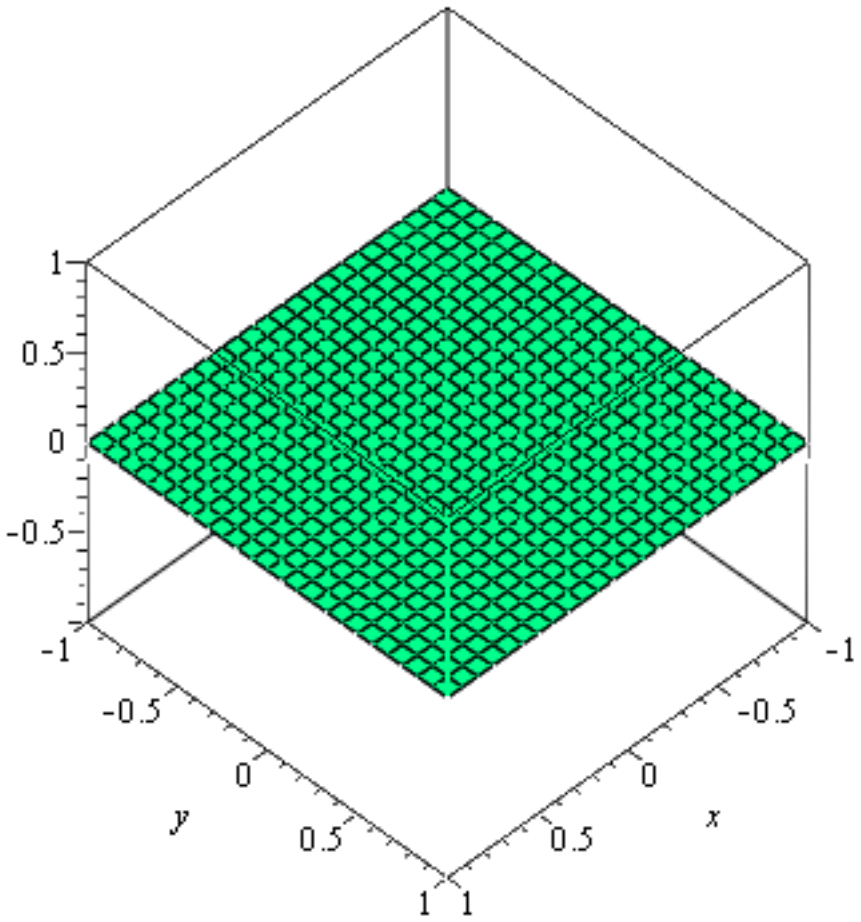
```
> display(plot1,plot2,axes=boxed,scaling=constrained);
```




```
> animate((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.
plots: Beachte auch Kontextmenü und Export-Varianten.

=== Ende Maple Teil IV ===