# Computer Algebra  using  Maple
# Part IV: [Numerical] Linear Algebra

**Winfried Auzinger, Kevin Sturm (SS 2019)**

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

[*&x*, *Add*, *Adjoint*, *BackwardSubstitute*, *BandMatrix*, *Basis*, *BezoutMatrix*, *BidiagonalForm*, *BilinearForm*, *CARE*, *CharacteristicMatrix*, *CharacteristicPolynomial*, *Column*, *ColumnDimension*, *ColumnOperation*, *ColumnSpace*, *CompanionMatrix*, *CompressedSparseForm*, *ConditionNumber*, *ConstantMatrix*, *ConstantVector*, *Copy*, *CreatePermutation*, *CrossProduct*, *DARE*, *DeleteColumn*, *DeleteRow*, *Determinant*, *Diagonal*, *DiagonalMatrix*, *Dimension*, *Dimensions*, *DotProduct*, *EigenConditionNumbers*, *Eigenvalues*, *Eigenvectors*, *Equal*, *ForwardSubstitute*, *FrobeniusForm*, *FromCompressedSparseForm*, *FromSplitForm*, *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*, *ProjectionMatrix*, *QRDecomposition*, *RandomMatrix*, *RandomVector*, *Rank*, *RationalCanonicalForm*, *ReducedRowEchelonForm*, *Row*, *RowDimension*, *RowOperation*, *RowSpace*, *ScalarMatrix*, *ScalarMultiply*, *ScalarVector*, *SchurForm*, *SingularValues*, *SmithForm*, *SplitForm*, *StronglyConnectedBlocks*, *SubMatrix*, *SubVector*, *SumBasis*, *SylvesterMatrix*, *SylvesterSolve*, *ToeplitzMatrix*, *Trace*, *Transpose*, *TridiagonalForm*, *UnitVector*, *VandermondeMatrix*, *VectorAdd*, *VectorAngle*, *VectorMatrixMultiply*, *VectorNorm*, *VectorScalarMultiply*, *ZeroMatrix*, *ZeroVector*, *Zip*]

# 1 Vectors and Matrices

Long and short forms:

```
> Vector([a,b,c]), <a,b,c>;        # column vector
```

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

```
> v:=<a,b,c>: v[2]; # vector element
```

$$b$$

**WARNING:** You can also use round brackets, v(2), but this has a different meaning when the value of the index is not correct. Not recommended for basic usage.

```
> Vector[row]([1,2,3]), <1|2|3>; # row vector
```

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

```
> M:=Matrix([[a,b,c],
             [d,e,f]]);
```

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

```
> <<a|b|c>,<d|e|f>>; # specification row-wise
```

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

```
> <<a,d>|<b,e>|<c,f>>; # specification column-wise
```

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

```
> Row(M,2), M[2,..];
  Column(M,3), M[..,3];
```

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

$$\begin{bmatrix} c \\ f \end{bmatrix}, \begin{bmatrix} c \\ f \end{bmatrix}$$

```
> A[2,3]; # matrix element
```

$$A_{2,\,3}$$

Or specification via '**generating function**' (i,j)->f(i,j) defining the entries:

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

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

Special case: constant numerical value

```
> Matrix(3,3,7);
```

$$\begin{bmatrix} 7 & 7 & 7 \\ 7 & 7 & 7 \\ 7 & 7 & 7 \end{bmatrix}$$

Playing LEGO (block form):

```
> x:=<1,2,3>; Matrix([x,x]);
```

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

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

Accessing submatrices via **vector index** notation (read or write):

```
> A[1..2,2..3]:=ZeroMatrix(2): A;
```

$$\begin{bmatrix} 2 & 0 & 0 & 5 \\ 3 & 0 & 0 & 6 \end{bmatrix}$$

NOTE:
**A row of a matrix is a row vector.**
**A column of a matrix is a column vector.**

```
> whattype(A[1,..]), whattype(A[..,1]);
```

$$Vector_{row}, Vector_{column}$$

```
> A[1,..]:=ZeroVector[row](4): A;
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 6 \end{bmatrix}$$

```
> A[..,4]:=ZeroVector(2): A;
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 \end{bmatrix}$$

NOTE: If a vector or matrix is 'filled' with values by some algorithm (e.g., a loop), you must first **initialize** the object (static memory allocation)

Example:

```
> n:=5:
  v:=Vector(n): # allocate memory, initialize to zero
  for i from 1 to n do
    v[i] := something
  end do:
```

For **special internal formats** like symmetric, banded, sparse, etc., see:

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

# 2 Package LinearAlgebra: Basic Operations

**Dimension** of Vectors and Matrices:

```
> x:=Vector([alpha,beta,gamma]);
  Dimension(x);
```

$$x := \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix}$$

$$3$$

```
> 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);
```

$$2, 3$$

```
> Dimension(A);
```

$$2, 3$$

Elementary operations:

```
> A,Rank(A); # generic rank!
```

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

```
> Transpose(A); # transposition
```

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

In LinearAlgebra, many functions have long names.

You can also define shortcuts:

```
> alias(H=HermitianTranspose): H(A);
```

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

Matrix-Vector and (non-commutative) Matrix-Matrix multiplication is specified as follows:

```
> MatrixVectorMultiply(A,x), A.x;
```

$$\begin{bmatrix} \alpha + 2\beta + 3\gamma \\ a\beta + b\gamma + \alpha \end{bmatrix}, \begin{bmatrix} \alpha + 2\beta + 3\gamma \\ a\beta + b\gamma + \alpha \end{bmatrix}$$

```
> MatrixMatrixMultiply(A,H(A)), A.H(A);
```

$$\begin{bmatrix} 14 & 1 + 2\bar{a} + 3\bar{b} \\ 1 + 2a + 3b & 1 + a\bar{a} + b\bar{b} \end{bmatrix}, \begin{bmatrix} 14 & 1 + 2\bar{a} + 3\bar{b} \\ 1 + 2a + 3b & 1 + a\bar{a} + b\bar{b} \end{bmatrix}$$

You can perform **symbolic** / **numeric** / **mixed** calculations.

```
> A:=Matrix(3,3,(i,j)->a[i]*b[j]); # a rank-1-Matrix
```

$$A := \begin{bmatrix} a_1 b_1 & a_1 b_2 & a_1 b_3 \\ a_2 b_1 & a_2 b_2 & a_2 b_3 \\ a_3 b_1 & a_3 b_2 & a_3 b_3 \end{bmatrix}$$

```
> Determinant(A);
```

$$0$$

```
> MatrixInverse(A), A^(-1);
Error, (in MatrixInverse) singular matrix
```

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

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

```
> A^(-1); # generically regular (for b<>2*a)
```

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

```
> A := evalf(RandomMatrix(3,3));
```

$$A := \begin{bmatrix} 27. & 99. & 92. \\ 8. & 29. & -31. \\ 69. & 44. & 67. \end{bmatrix}$$

```
> A^(-1);
```

$$\begin{bmatrix} -0.0101056092701470 & 0.00789930449450563 & 0.0175312610773612 \\ 0.00817432863551356 & 0.0138703841781667 & -0.00480681082006087 \\ 0.00503905342802313 & -0.0172440136411974 & 0.0000275024141007932 \end{bmatrix}$$

For row or colums vectors, the dot operator evaluates the inner product:

```
> x:=Vector[row](4,symbol='xi');
  y:=Vector[row](4,symbol='eta');
```

$$x := \begin{bmatrix} \xi_1 & \xi_2 & \xi_3 & \xi_4 \end{bmatrix}$$

$$y := \begin{bmatrix} \eta_1 & \eta_2 & \eta_3 & \eta_4 \end{bmatrix}$$

```
> x.y;
```

$$\overline{\eta_1}\,\xi_1 + \overline{\eta_2}\,\xi_2 + \overline{\eta_3}\,\xi_3 + \overline{\eta_4}\,\xi_4$$

This is equivalent to:

```
> DotProduct(x,y);
```

$$\overline{\eta_1}\,\xi_1 + \overline{\eta_2}\,\xi_2 + \overline{\eta_3}\,\xi_3 + \overline{\eta_4}\,\xi_4$$

If you are assuming real data, avoid conjugation:

```
> DotProduct(x,y,conjugate=false);
```

$$\xi_1\,\eta_1 + \xi_2\,\eta_2 + \xi_3\,\eta_3 + \xi_4\,\eta_4$$

Numerical data:

```
> x,y:=Vector([1+I,2+I,3+I]),Vector([1,2,3]);
```

$$x, y := \begin{bmatrix} 1+\mathrm{I} \\ 2+\mathrm{I} \\ 3+\mathrm{I} \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

```
> x.y;
```

$$14 - 6\,\mathrm{I}$$

```
> x,y:=RandomVector(5),RandomVector(5):
> x.y;
```

$$11752$$

Euclidian norm:

```
> sqrt(x.x), Norm(x,2);
```

$$2\sqrt{2926},\, 2\sqrt{2926}$$

Testing vectors or matrices for equality: use **Equal:**

```
> x,y,Equal(x,y);
```

$$\begin{bmatrix} -72 \\ -2 \\ -32 \\ -74 \\ -4 \end{bmatrix}, \begin{bmatrix} -77 \\ 57 \\ 27 \\ -93 \\ -76 \end{bmatrix}, \mathit{false}$$

# 3 Some useful general functions from LinearAlgebra

For **Vector**:

```
> n:=3:
> x,y:=Vector(3,symbol='xi'),Vector(3,symbol='eta');
```

$$x, y := \begin{bmatrix} \xi_1 \\ \xi_2 \\ \xi_3 \end{bmatrix}, \begin{bmatrix} \eta_1 \\ \eta_2 \\ \eta_3 \end{bmatrix}$$

```
> Transpose(x); # convert to row vector
```

$$\begin{bmatrix} \xi_1 & \xi_2 & \xi_3 \end{bmatrix}$$

```
> Transpose(%);
```

$$\begin{bmatrix} \xi_1 \\ \xi_2 \\ \xi_3 \end{bmatrix}$$

```
> x.Transpose(y); # this is the same as:
```

$$\begin{bmatrix} \xi_1 \eta_1 & \xi_1 \eta_2 & \xi_1 \eta_3 \\ \xi_2 \eta_1 & \xi_2 \eta_2 & \xi_2 \eta_3 \\ \xi_3 \eta_1 & \xi_3 \eta_2 & \xi_3 \eta_3 \end{bmatrix}$$

```
> OuterProductMatrix(x,y);
```

$$\begin{bmatrix} \xi_1 \eta_1 & \xi_1 \eta_2 & \xi_1 \eta_3 \\ \xi_2 \eta_1 & \xi_2 \eta_2 & \xi_2 \eta_3 \\ \xi_3 \eta_1 & \xi_3 \eta_2 & \xi_3 \eta_3 \end{bmatrix}$$

```
> DotProduct(x,y); # inner product
```

$$\overline{\xi_1}\, \eta_1 + \overline{\xi_2}\, \eta_2 + \overline{\xi_3}\, \eta_3$$

```
> CrossProduct(x,y); # cross product of 3-dimensional vectors
```

$$\begin{bmatrix} -\xi_3 \eta_2 + \xi_2 \eta_3 \\ \xi_3 \eta_1 - \xi_1 \eta_3 \\ -\xi_2 \eta_1 + \xi_1 \eta_2 \end{bmatrix}$$

Special Vectors:

```
> ZeroVector(n);
```

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

```
> seq(UnitVector(i,n),i=1..n);
```

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

```
> ConstantVector(c,n);
```

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

For **Matrix**:

```
> n:=3:
> A:=Matrix(n,n,symbol='a');
```

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

```
> Diagonal(A);
```

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

```
> Determinant(A);
```

$$a_{1,1} a_{2,2} a_{3,3} - a_{1,1} a_{2,3} a_{3,2} - a_{1,2} a_{2,1} a_{3,3} + a_{1,2} a_{2,3} a_{3,1} + a_{1,3} a_{2,1} a_{3,2} - a_{1,3} a_{2,2} a_{3,1}$$

```
> CharacteristicPolynomial(A,lambda);
```

$$\lambda^3 - \left(a_{3,3} + a_{2,2} + a_{1,1}\right) \lambda^2 - \left(-a_{2,2} a_{1,1} - a_{3,3} a_{1,1} + a_{2,1} a_{1,2} + a_{3,1} a_{1,3} - a_{3,3} a_{2,2}\right.$$

$$+ a_{3,2}\, a_{2,3}) \lambda - a_{1,1}\, a_{2,2}\, a_{3,3} + a_{1,1}\, a_{2,3}\, a_{3,2} + a_{1,2}\, a_{2,1}\, a_{3,3} - a_{1,2}\, a_{2,3}\, a_{3,1} - a_{1,3}\, a_{2,1}\, a_{3,2} + a_{1,3}\, a_{2,2}\, a_{3,1}$$

Special Matrices:

```
> ZeroMatrix(n), IdentityMatrix(n);
```

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

```
> ConstantMatrix(c,n);
```

$$\begin{bmatrix} c & c & c \\ c & c & c \\ c & c & c \end{bmatrix}$$

```
> V:=VandermondeMatrix(x);
```

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

The names of most functions in **LinearAlgebra** are self-explanatory.

If you know what the companion matrix of a polynomial is, use **CompanionMatrix**:

```
> p:=1+2*t+3*t^2+t^3;
```

$$p := t^3 + 3\,t^2 + 2\,t + 1$$

```
> C:=CompanionMatrix(p,t);
```

$$C := \begin{bmatrix} 0 & 0 & -1 \\ 1 & 0 & -2 \\ 0 & 1 & -3 \end{bmatrix}$$

```
> CharacteristicPolynomial(C,t); # this just the given polynomial
  p
```

$$t^3 + 3\,t^2 + 2\,t + 1$$

Solution of a **linear system of equations:**

```
> b:=Vector([alpha,beta,gamma]);
```

$$b := \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix}$$

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

$$\begin{bmatrix} \dfrac{\alpha\,\xi_2^2\,\xi_3 - \alpha\,\xi_2\,\xi_3^2 - \beta\,\xi_1^2\,\xi_3 + \beta\,\xi_1\,\xi_3^2 + \gamma\,\xi_1^2\,\xi_2 - \gamma\,\xi_1\,\xi_2^2}{\left(\xi_2\,\xi_1 - \xi_3\,\xi_1 - \xi_3\,\xi_2 + \xi_3^2\right)\,\left(\xi_1 - \xi_2\right)} \\ -\dfrac{\alpha\,\xi_2^2 - \alpha\,\xi_3^2 - \beta\,\xi_1^2 + \beta\,\xi_3^2 + \gamma\,\xi_1^2 - \gamma\,\xi_2^2}{\left(\xi_2\,\xi_1 - \xi_3\,\xi_1 - \xi_3\,\xi_2 + \xi_3^2\right)\,\left(\xi_1 - \xi_2\right)} \\ \dfrac{\xi_2\,\alpha - \xi_3\,\alpha - \beta\,\xi_1 + \xi_3\,\beta + \gamma\,\xi_1 - \gamma\,\xi_2}{\left(\xi_2\,\xi_1 - \xi_3\,\xi_1 - \xi_3\,\xi_2 + \xi_3^2\right)\,\left(\xi_1 - \xi_2\right)} \end{bmatrix}$$

The right-hand side can also be a matrix;
this is equivalent to solving several systems with
the columns of the righ-hand side matrix.

```
> B:=Matrix([b,UnitVector(1,3)]);
```

$$B := \begin{bmatrix} \alpha & 1 \\ \beta & 0 \\ \gamma & 0 \end{bmatrix}$$

```
> LinearSolve(C,B);
```

$$\begin{bmatrix} \beta - 2\,\alpha & -2 \\ \gamma - 3\,\alpha & -3 \\ -\alpha & -1 \end{bmatrix}$$

# 4 Finding a rule by experiment

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

Given: a special **bidiagonal Matrix**

We want to understand how the powers of this matrix look like.

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

$$n := 4$$

```
> B:=DiagonalMatrix([seq(lambda[i],i=1..n)]):
  for i from 1 to n-1 do
     B[i,i+1]:=1
  end do:
> B,B^2,map(expand,B^3); # 'map' is explained in Part V
```

$$\begin{bmatrix} \lambda_1 & 1 & 0 & 0 \\ 0 & \lambda_2 & 1 & 0 \\ 0 & 0 & \lambda_3 & 1 \\ 0 & 0 & 0 & \lambda_4 \end{bmatrix}, \begin{bmatrix} \lambda_1^2 & \lambda_1 + \lambda_2 & 1 & 0 \\ 0 & \lambda_2^2 & \lambda_2 + \lambda_3 & 1 \\ 0 & 0 & \lambda_3^2 & \lambda_3 + \lambda_4 \\ 0 & 0 & 0 & \lambda_4^2 \end{bmatrix},$$

$$\begin{bmatrix} \lambda_1^3 & \lambda_1^2 + \lambda_2 \lambda_1 + \lambda_2^2 & \lambda_1 + \lambda_2 + \lambda_3 & 1 \\ 0 & \lambda_2^3 & \lambda_2^2 + \lambda_3 \lambda_2 + \lambda_3^2 & \lambda_2 + \lambda_3 + \lambda_4 \\ 0 & 0 & \lambda_3^3 & \lambda_3^2 + \lambda_4 \lambda_3 + \lambda_4^2 \\ 0 & 0 & 0 & \lambda_4^3 \end{bmatrix}$$

```
> map(expand,B^4);
```

$$\Big[\big[\lambda_1^4, \lambda_1^3 + \lambda_1^2 \lambda_2 + \lambda_2^2 \lambda_1 + \lambda_2^3, \lambda_1^2 + \lambda_2 \lambda_1 + \lambda_1 \lambda_3 + \lambda_2^2 + \lambda_3 \lambda_2 + \lambda_3^2, \lambda_1 + \lambda_2 + \lambda_3 + \lambda_4\big],$$
$$\big[0, \lambda_2^4, \lambda_2^3 + \lambda_2^2 \lambda_3 + \lambda_3^2 \lambda_2 + \lambda_3^3, \lambda_2^2 + \lambda_3 \lambda_2 + \lambda_2 \lambda_4 + \lambda_3^2 + \lambda_4 \lambda_3 + \lambda_4^2\big],$$
$$\big[0, 0, \lambda_3^4, \lambda_3^3 + \lambda_3^2 \lambda_4 + \lambda_4^2 \lambda_3 + \lambda_4^3\big],$$
$$\big[0, 0, 0, \lambda_4^4\big] \qquad , \qquad \Big]$$

From this observation, you may guess the general form of B^p.
Not very difficult, but also not completely obvious.

Special case: Jordan block

```
> B:=DiagonalMatrix([seq(lambda,i=1..n)]):
  for i from 1 to n-1 do
     B[i,i+1]:=1
  end do:
> B,B^5;
```

$$\begin{bmatrix} \lambda & 1 & 0 & 0 \\ 0 & \lambda & 1 & 0 \\ 0 & 0 & \lambda & 1 \\ 0 & 0 & 0 & \lambda \end{bmatrix}, \begin{bmatrix} \lambda^5 & 5\lambda^4 & 10\lambda^3 & 10\lambda^2 \\ 0 & \lambda^5 & 5\lambda^4 & 10\lambda^3 \\ 0 & 0 & \lambda^5 & 5\lambda^4 \\ 0 & 0 & 0 & \lambda^5 \end{bmatrix}$$

# 5 Numerical Linear Algebra I: The basics

Many algorithms from linear algebra make only sense in general for **numerical data.**

We use 20 digits working precision and display 10 digits.

```
> Digits:=20;
```

$$Digits := 20$$

```
> n:=3;
```

$$n := 3$$

```
> A:=evalf(RandomMatrix(n,n));
```

$$A := \begin{bmatrix} 27. & 99. & 92. \\ 8. & 29. & -31. \\ 69. & 44. & 67. \end{bmatrix}$$

```
> b:=evalf(RandomVector(n));
```

$$b := \begin{bmatrix} -32. \\ -74. \\ -4. \end{bmatrix}$$

Solve **linear system of equations**:

Algorithm: Stabilized Gaussian elimination based on PLU-decomposition of A

```
> LinearSolve(A,b);
```

$$\begin{bmatrix} -0.33129408025815599371 \\ -1.2687597022405300021 \\ 1.1146972900954639352 \end{bmatrix}$$

Solve **eigenproblem** for A:

Algorithm: QR iteration

```
> Eigenvalues(A); # Vector of eigenvalues
```

$$\begin{bmatrix} -47.58707854005080573 + 0.\,I \\ 105.24966957637869910 + 0.\,I \\ 65.337408963672106623 + 0.\,I \end{bmatrix}$$

```
> evalues,evectors:=Eigenvectors(A); # Vector of eigenvalues,
                                     # Matrix of eigenvectors
  (columns)
```

$$evalues, evectors := \begin{bmatrix} -47.58707854005080573 + 0.\,I \\ 105.24966957637869910 + 0.\,I \\ 65.33740896367210 6623 + 0.\,I \end{bmatrix}, [[-0.86794440159591751915 + 0.\,I,$$

$0.58691806695558544700 + 0.\,I,\ 0.34225315163027785406 + 0.\,I],$
$[0.26155854653069927359 + 0.\,I,\ -0.25132955699930870848 + 0.\,I,$
$-0.56508112930395915496 + 0.\,I],$
$[0.42220805590970510659 + 0.\,I,\ 0.76964968424579383647 + 0.\,I,$
$0.75069707439399564335 + 0.\,I]]$

Evidently, the eigensystem is real. 0. I is an imaginary rounding noise.

Convert to real:

```
> evalues:=Re(evalues);
  evectors:=Re(evectors);
```

$$evalues := \begin{bmatrix} -47.58707854005080573 \\ 105.24966957637869910 \\ 65.33740896367210 6623 \end{bmatrix}$$

$$evectors := \begin{bmatrix} -0.86794440159591751915 & 0.58691806695558544700 & 0.34225315163027785406 \\ 0.26155854653069927359 & -0.25132955699930870848 & -0.56508112930395915496 \\ 0.42220805590970510659 & 0.76964968424579383647 & 0.75069707439399564335 \end{bmatrix}$$

Check:

```
> evalf(A.evectors-evectors.DiagonalMatrix(evalues));
```

$$\begin{bmatrix} -2.\,10^{-18} & -2.4\,10^{-17} & -2.7\,10^{-17} \\ 3.\,10^{-18} & 4.\,10^{-18} & 6.\,10^{-18} \\ 3.\,10^{-18} & 1.4\,10^{-17} & -1.\,10^{-18} \end{bmatrix}$$

Extract eigenvectors from matrix:

```
> for i from 1 to n do
    ev[i] := Column(evectors,i)
  end do;
```

$$ev_1 := \begin{bmatrix} -0.86794440159591751915 \\ 0.26155854653069927359 \\ 0.42220805590970510659 \end{bmatrix}$$

$$ev_2 := \begin{bmatrix} 0.58691806695558544700 \\ -0.25132955699930870848 \\ 0.76964968424579383647 \end{bmatrix}$$

$$ev_3 := \begin{bmatrix} 0.34225315163027785406 \\ -0.56508112930395915496 \\ 0.75069707439399564335 \end{bmatrix}$$

# 6 Numerical Linear Algebra II: Hardware floats

In Maple, you can also perform computation in hardware floats (like double in C or Matlab). However, the usage is less convenient as in Matlab.

General recommendation:

- For efficient problem solving in (hardware-) double precision arithmetic, Matlab is to be preferred.

- If you need higher precision, use Maple with normal [s]floats and an appropriate value for Digits. Such a computation is much slower but may be very useful in certain cases.

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

For efficient solution of larger problems, resort to hardware float operations (64 bit IEEE double precision). The data type is **hfloat** = **float[8]**

**Use HFloat** or **evalhf** instead of evalf to generate a hfloat object.

```
> half:=HFloat(0.5);
  whattype(half),
  type(half,hfloat),
  type(half,float[8]);
```

$$half := 0.500000000000000$$

$$float, true, true$$

WARNING: **evalhf** evaluates an expression in hardware floating point arithmetic, but the result is given back as a normal float (**sfloat**).

```
> pi2:=evalhf(Pi^2);
```

$$\pi 2 := 9.86960440108935799$$

```
> type(pi2,hfloat), type(pi2,sfloat);
```

$$false, true$$

Use **convert(...,hfloat)** to convert an object to a hfloat:

```
> convert(pi2,hfloat);
```

$$9.86960440108936$$

```
> type(%,hfloat);
```

$$true$$

This looks a little bit complicated.

For practice, change the value of environment variable **UseHardwareFloats**:

```
> UseHardwareFloats;
```

$$deduced$$

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

$$UseHardwareFloats := true$$

Now, all operations involving exclusively hfloat data are automatically performed in machine arithmetic,
and the result is given back as a hfloat object.

```
> A:=HilbertMatrix(4,4,datatype=hfloat); # generate matrix with hfloat entries
```

$$A := \begin{bmatrix} 1. & 0.500000000000000 & 0.333333333333333 & 0.250000000000000 \\ 0.500000000000000 & 0.333333333333333 & 0.250000000000000 & 0.200000000000000 \\ 0.333333333333333 & 0.250000000000000 & 0.200000000000000 & 0.166666666666667 \\ 0.250000000000000 & 0.200000000000000 & 0.166666666666667 & 0.142857142857143 \end{bmatrix}$$

```
> A^(-1);
```

$$\begin{bmatrix} 15.9999999999995 & -119.999999999994 & 239.999999999985 & -139.999999999990 \\ -119.999999999994 & 1199.99999999992 & -2699.99999999981 & 1679.99999999988 \\ 239.999999999984 & -2699.99999999981 & 6479.99999999952 & -4199.99999999969 \\ -139.999999999989 & 1679.99999999987 & -4199.99999999968 & 2799.99999999979 \end{bmatrix}$$

```
> type(%[1,1],hfloat);
```

$$true$$

```
> eig:=Eigenvalues(A);
```

$$eig := \begin{bmatrix} 1.50021428005924 + 0.\,I \\ 0.169141220221450 + 0.\,I \\ 0.00673827360576074 + 0.\,I \\ 0.0000967023040226002 + 0.\,I \end{bmatrix}$$

```
> type(eig[1],hfloat);
```

$$false$$

Hilbert matrix is symmetric, with real eigenvalues. The numerical result contains imaginary rounding noise. Therefore the data type is **complex[8]** instead of float[8]:

```
> type(eig[1],complex[8]);
```

$$true$$

```
> eig:=Re(eig);
```

$$eig := \begin{bmatrix} 1.50021428005924 \\ 0.169141220221450 \\ 0.00673827360576074 \\ 0.0000967023040226002 \end{bmatrix}$$

```
> type(eig[1],hfloat);
```

*true*

Performance comparison: We invert the 120x120 Hilbert matrix

- exactly,
- in software floating point arithemtic,
- in hardware floating point arithmetic.

```
> n:=120;
```

$n := 120$

```
> H := HilbertMatrix(n,n):
> start:=time():
  H^(-1):
  time()-start;
```

3.203

```
> Hs := evalf(H):
> start:=time():
  Hs^(-1):
  time()-start;
```

0.093

```
> Hh:=evalhf(H): # this works: result matrix stored as hfloat
  object
> type(Hh[1,1],hfloat);
```

*true*

```
> start:=time():
  Hf^(-1):
  time()-start;
```

0.

# 7 Numerical Linear Algebra III: Examples

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

$$UseHardwareFloats := true$$

For hfloat data, the operations in LinearAlgebra automatically resort to an optimized built-in numerical linear algebra library.

## 7.1 Solution of a least squares problem

Function **LeastSquares** solves a least squares problem **||Ax-b||_2 --> min!**

We use it to compute a regression line for given data.

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

$$n := 7$$

```
> t := [seq(HFloat(i),i=1..n)];
  y := [seq(HFloat(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,datatype=hfloat);
```

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

```
> Y := convert(y,Vector): type(Y[1],hfloat);
```

$$true$$

```
> LinearSolve(V,Y); # too many equations, no solution
Error, (in LinearAlgebra:-LinearSolve) inconsistent system
```

```
> a := LeastSquares(V,Y);
```

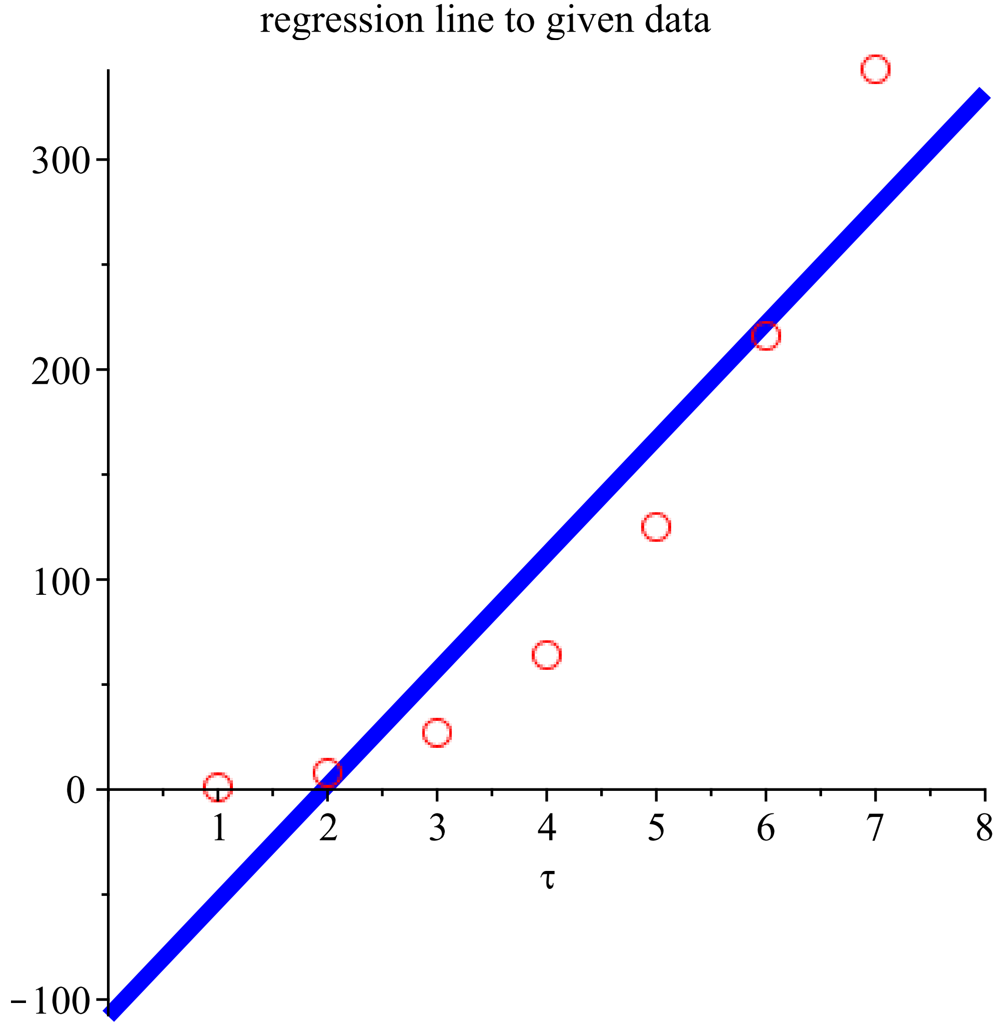
$$a := \begin{bmatrix} -108.000000000000 \\ 55.0000000000000 \end{bmatrix}$$

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

$$line := \tau \mapsto a_1 + \tau\, a_2$$

```
> p1 := plot(line(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="regression line to given data");
```




## 7.2 Best approximation and orthogonal projection

Let U be a subspace of R^n. For given x in R^n, find u in U such that
|| **u-x** ||_**2** becomes minimal (best approximation problem).

The solution is given by the <u>orthogonal projection</u> of x onto u.

Algorithm:

- Choose a basis (b1,...,b_m) of U
- Convert it to an orthonormal basis (q_1,...,q_m) using the Gram-Schmidt algorithm
- Compute u = sum_(i=1)^m (x,q_i) q_i

Example (n=4, m=3):

```
> B:=RandomMatrix(4,3,datatype=float); # columns of B define
  basis of 3-dimensional subspace U
```

$$B := \begin{bmatrix} -70. & -7. & -25. \\ 13. & 12. & 40. \\ -58. & -53. & 97. \\ -94. & 21. & 43. \end{bmatrix}$$

```
> Q:=GramSchmidt([seq(B(..,i),i=1..3)],normalized=true); #
  construct orthonormal basis of U
```

$$Q := \left[ \begin{bmatrix} -0.532677613510000 \\ 0.0989258425090000 \\ -0.441361451194000 \\ -0.715309938142000 \end{bmatrix}, \begin{bmatrix} 0.00135464330718282 \\ 0.186965093099809 \\ -0.824730554203763 \\ 0.533724397049074 \end{bmatrix}, \begin{bmatrix} -0.619745551074306 \\ 0.618873703754637 \\ 0.352558159942963 \\ 0.329565674971488 \end{bmatrix} \right]$$

```
> x:=RandomVector(4,datatype=float)
```

$$x := \begin{bmatrix} 89. \\ -55. \\ -67. \\ 77. \end{bmatrix}$$

```
> u:=add((x.Q[i])*Q[i],i=1..3)
```

$$u := \begin{bmatrix} 96.0464169287540 \\ -45.7512425083110 \\ -67.3285912741561 \\ 73.2344995848048 \end{bmatrix}$$

## 7.3 Polynomial differentiation weights

Assume that you want to compute the derivatives of (many) polynomials
of degree n at a given, fixed evaluation point tau.
We assume that the polynomials are specified by their values at n+1 nodes.
This can be expressed by algebraic operations.

- For nodes t[1],...,t[n+1] :
- Given y[1]:=p(t[1]), ..., y[n+1] := p(t[n+1])
- For an evaluation point tau: Find formula

p'(tau) = sum_(i=1)^(n+1) alpha[i](tau)*p(t[i])

- You can find it by hand or delegate this job to Maple.

We consider a particular case, namely with integer nodes, where we an solve this in exact rational arithmetic.

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

$$n := 4$$

```
> t:=[0,1,2,3,4];
```

$$t := [0, 1, 2, 3, 4]$$

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

$$c_4 \tau^4 + c_3 \tau^3 + c_2 \tau^2 + c_1 \tau + c_0$$

```
> desired_identity:=D(p)(tau)-add(alpha[i]*p(t[i]),i=1..n+1);
  # this should be 0
```

$$\begin{aligned} desired\_identity &:= 4 \tau^3 c_4 + 3 \tau^2 c_3 + 2 \tau c_2 + c_1 - \alpha_1 c_0 - \alpha_2 (c_4 + c_3 + c_2 + c_1 + c_0) \\ &- \alpha_3 (16 c_4 + 8 c_3 + 4 c_2 + 2 c_1 + c_0) - \alpha_4 (81 c_4 + 27 c_3 + 9 c_2 + 3 c_1 + c_0) \\ &- \alpha_5 (256 c_4 + 64 c_3 + 16 c_2 + 4 c_1 + c_0) \end{aligned}$$

Now we compare coefficients of the c[i] using **coeff**:

```
> for i from 0 to n do
     eq[i]:=coeff(desired_identity,c[i])
  end do;
```

$$eq_0 := -\alpha_1 - \alpha_2 - \alpha_3 - \alpha_4 - \alpha_5$$

$$eq_1 := 1 - \alpha_2 - 2 \alpha_3 - 3 \alpha_4 - 4 \alpha_5$$

$$eq_2 := 2 \tau - \alpha_2 - 4 \alpha_3 - 9 \alpha_4 - 16 \alpha_5$$

$$eq_3 := 3 \tau^2 - \alpha_2 - 8 \alpha_3 - 27 \alpha_4 - 64 \alpha_5$$

$$eq_4 := 4 \tau^3 - \alpha_2 - 16 \alpha_3 - 81 \alpha_4 - 256 \alpha_5$$

These are 5 equations in 5 unknowns, depending on tau.
We can solve them directly using **solve,** but for practice we convert it into a linear system:

```
> A:=Matrix([seq([seq(coeff(eq[i],alpha[j]),j=1..n+1)],i=0..n)
  ]);
```

$$A := \begin{bmatrix} -1 & -1 & -1 & -1 & -1 \\ 0 & -1 & -2 & -3 & -4 \\ 0 & -1 & -4 & -9 & -16 \\ 0 & -1 & -8 & -27 & -64 \\ 0 & -1 & -16 & -81 & -256 \end{bmatrix}$$

```
> b := -Vector([seq(subs(seq(alpha[j]=0,j=1..n+1),eq[i]),i=0..
  n)]);
```

$$b := \begin{bmatrix} 0 \\ -1 \\ -2\,\tau \\ -3\,\tau^2 \\ -4\,\tau^3 \end{bmatrix}$$

```
> Alpha := LinearSolve(A,b);
```

$$\mathrm{A} := \begin{bmatrix} -\frac{25}{12} + \frac{1}{6}\,\tau^3 - \frac{5}{4}\,\tau^2 + \frac{35}{12}\,\tau \\ 4 - \frac{2}{3}\,\tau^3 + \frac{9}{2}\,\tau^2 - \frac{26}{3}\,\tau \\ \tau^3 - 6\,\tau^2 + \frac{19}{2}\,\tau - 3 \\ -\frac{2}{3}\,\tau^3 + \frac{7}{2}\,\tau^2 - \frac{14}{3}\,\tau + \frac{4}{3} \\ \frac{1}{6}\,\tau^3 - \frac{3}{4}\,\tau^2 + \frac{11}{12}\,\tau - \frac{1}{4} \end{bmatrix}$$

These are the weights we have been looking for. We check it for tau=10:

```
> for i from 1 to n+1 do alpha[i]:=subs(tau=10,Alpha[i]) end
  do;
```

$$\alpha_1 := \frac{275}{4}$$

$$\alpha_2 := -\frac{898}{3}$$

$$\alpha_3 := 492$$

$$\alpha_4 := -362$$

$$\alpha_5 := \frac{1207}{12}$$

Now we compare:

```
> add(alpha[i]*p(t[i]),i=1..n+1); # am inner product
```

$$4000\,c_4 + 300\,c_3 + 20\,c_2 + c_1$$

```
> D(p)(10);
```

$$4000\,c_4 + 300\,c_3 + 20\,c_2 + c_1$$

```
> # OK
```

======================================================= end of Part IV ==