


# Handbook of Magma Functions

## Part IV: Matrices and Linear Algebra

Version 2.30-1

**Sydney**

November 10, 2026

## Editors

*John Cannon* *Wieb Bosma* *Claus Fieker* *Allan Steel*

## Handbook Contributors

*Eran Assaf* *Geoff Bailey* *Wieb Bosma* *Gavin Brown* *Nils Bruin* *John Cannon* *Jon Carlson* *Scott Contini* *Edgar Costa* *Bruce Cox* *Brendan Creutz* *Steve Donnelly* *Tim Dokchitser* *Willem de Graaf* *Andreas-Stephan Elsenhans* *Claus Fieker* *Damien Fisher* *Volker Gebhardt* *Sergei Haller* *Michael Harrison* *Florian Hess* *Derek Holt* *David Howden* *Al Kasprzyk* *Markus Kirschmer* *David Kohel* *Axel Kohnert* *Dimitri Leemans* *Paulette Lieby* *Graham Matthews* *Scott Murray* *Eamonn O'Brien* *Dan Roozemond* *Ben Smith* *Bernd Souvignier* *William Stein* *Allan Steel* *Damien Stehlé* *Nicole Sutherland* *Don Taylor* *Bill Unger* *Alexa van der Waall* *Paul van Wamelen* *Helena Verrill* *John Voight* *Mark Watkins* *Greg White*

## Production Editors

*Wieb Bosma* *Claus Fieker* *Allan Steel* *Nicole Sutherland*

## HTML Production

*Claus Fieker* *Allan Steel*

Copyright © 1994–2026 The University of Sydney. All rights reserved.

# Contents


**27 Matrices** **1**
- 27.1 Introduction . . . 1
- 27.2 Creation of Matrices . . . 1
- 27.3 Elementary Properties . . . 7
- 27.4 Accessing or Modifying Entries . . . 8
- 27.5 Building Block Matrices . . . 13
- 27.6 Changing Ring . . . 14
- 27.7 Elementary Arithmetic . . . 14
- 27.8 Nullspaces and Solutions of Systems . . . 15
- 27.9 Predicates . . . 17
- 27.10 Determinant and Other Properties . . . 18
- 27.11 Minimal and Characteristic Polynomials and Eigenvalues . . . 19
- 27.12 Canonical Forms . . . 20
- 27.13 Orders of Invertible Matrices . . . 25
- 27.14 Numerical Linear Algebra . . . 26
- 27.15 Miscellaneous Operations on Matrices . . . 32
- 27.16 Bibliography . . . 32

**28 Sparse Matrices** **34**
- 28.1 Introduction . . . 34
- 28.2 Creation of Sparse Matrices . . . 34
- 28.3 Accessing Sparse Matrices . . . 37
- 28.4 Accessing or Modifying Entries . . . 38
- 28.5 Building Block Matrices . . . 41
- 28.6 Conversion to and from Dense Matrices . . . 42
- 28.7 Changing Ring . . . 42
- 28.8 Predicates . . . 42
- 28.9 Elementary Arithmetic . . . 43
- 28.10 Multiplying Vectors or Matrices by Sparse Matrices . . . 43
- 28.11 Non-trivial Properties . . . 44
- 28.12 Determinant and Other Properties . . . 45
- 28.13 Minimal and Characteristic Polynomials and Eigenvalues . . . 45
- 28.14 Linear Systems (Structured Gaussian Elimination) . . . 46
- 28.15 Bibliography . . . 52

**29 Vector Spaces** **53**
- 29.1 Introduction . . . 53
- 29.2 Creation of Vector Spaces and Arithmetic with Vectors . . . 53
- 29.3 Subspaces, Quotient Spaces, and Homomorphisms . . . 59
- 29.4 Changing the Coefficient Field . . . 63
- 29.5 Basic Operations . . . 63
- 29.6 Reducing Vectors Relative to a Subspace . . . 65
- 29.7 Bases . . . 65
- 29.8 Operations with Linear Transformations . . . 67

**30 Polar Spaces** **69**
- 30.1 Introduction . . . 69

30.2 Reflexive Forms . . . 69
30.3 Inner Products . . . 70
30.4 Isotropic and Singular Vectors and Subspaces . . . 72
30.5 The Standard Forms . . . 75
30.6 Constructing Polar Spaces . . . 77
30.7 Isometries and Similarities . . . 82
30.8 Classical Groups . . . 86
30.9 Lie Algebras and Bilinear Forms . . . 87
30.10 Wall Forms . . . 89
30.11 Invariant Forms . . . 90
30.12 Polar Spaces More Generally . . . 94
30.13 Bibliography . . . 95

**References** **96**

**Index** **97**

# Chapter 27

# Matrices

## 27.1 Introduction

This chapter describes all the basic operations available for creating and working with matrices. Matrices arise in many different contexts and there are several types of matrix within Magma, but most of the operations listed here apply to all types of matrix.

The parent of any matrix will be one of several types of matrix-structure (module, matrix algebra, matrix group, etc.), and each of these matrix-structure types are described in other chapters, together with operations peculiar to their elements.

## 27.2 Creation of Matrices

This section describes the elementary constructs provided for creating a matrix or vector. For each of the following functions, the parent of the result will be as follows:

**(a)** If the result is a vector then its parent will be the appropriate $R$-space (of type `ModTupRng` or `ModTupFld`).

**(b)** If the result is a square matrix then its parent will be the appropriate matrix algebra (of type `AlgMatElt`).

**(c)** If the result is a non-square matrix then its parent will be the appropriate $R$-matrix space (of type `ModMatRng` or `ModMatFld`).

A matrix or a vector may also be created by coercing a sequence of ring elements into the appropriate parent matrix structure. There is also a virtual type `Mtrx` and all matrix types inherit from `Mtrx`. While writing package intrinsics, an argument should be declared to be of type `Mtrx` if it is a general matrix.

### 27.2.1 General Matrix Construction

`Matrix(R, m, n, Q): Rng, RngIntElt, RngIntElt, [ RngElt ] -> Mtrx`

`Matrix(R, m, n, Q): Rng, RngIntElt, RngIntElt, [ <RngIntElt, RngIntElt, RngElt> ] -> Mtrx`

`Matrix(R, m, n, Q): Rng, RngIntElt, RngIntElt, [ [ RngElt ] ] -> Mtrx`

Given a ring $R$, integers $m, n \geq 0$, and a sequence $Q$, return the $m \times n$ matrix over $R$ whose entries are those specified by $Q$, coerced into $R$. Either of $m$ and $n$ may be 0, in which case $Q$ must have length 0 (and may even be null), and the $m \times n$ zero matrix over $R$ is returned. There are several possibilities for $Q$:

**(a)** The sequence $Q$ may be a sequence of length $mn$ containing elements of a ring $S$, in which case the entries are given in row-major order. In this case, the function is equivalent to `MatrixRing(R, n)!Q` if $m = n$ and `RMatrixSpace(R, m, n)!Q` otherwise.

**(b)** The sequence $Q$ may be a sequence of tuples, each of the form `<i, j, x>`, where $1 \leq i \leq m$, $1 \leq j \leq n$, and $x \in S$ for some ring $S$. Such a tuple specifies that the $(i, j)$-th entry of the matrix is $x$. If an entry position is not given then its value is zero, while if an entry position is repeated then the last value overrides any previous value(s). This case is useful for creating sparse matrices.

**(c)** The sequence $Q$ may be a sequence of $m$ sequences, each of length $n$ and having entries in a ring $S$, in which case the rows of the matrix are specified by the inner sequences.

**(d)** The sequence $Q$ may be a sequence of $m$ vectors, each of length $n$ and having entries in a ring $S$, in which case the rows of the matrix are specified by the vectors.

**Example: Creating Matrices with Matrix** (ex-79e602)

This example demonstrates simple ways of creating matrices using the general `Matrix(R, m, n, Q)` function.

(a) Defining a $2 \times 2$ matrix over $\mathbb{Z}$:

```
> X := Matrix(IntegerRing(), 2, 2, [1,2, 3,4]);
> X;
[1 2]
[3 4]
> Parent(X);
Full Matrix Algebra of degree 2 over Integer Ring
```

(b) Defining a $2 \times 3$ matrix over $\mathbf{F}_{23}$:

```
> X := Matrix(GF(23), 2, 3, [1,-2,3, 4,100,-6]);
> X;
[ 1 21  3]
[ 4  8 17]
> Parent(X);
Full KMatrixSpace of 2 by 3 matrices over GF(23)
```

(c) Defining a sparse $5 \times 10$ matrix over $\mathbb{Q}$:

```
> X := Matrix(RationalField(), 5, 10, [<1,2,23>, <3,7,11>, <5,10,-1>]);
> X;
[ 0 23  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0 11  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0 -1]
> Parent(X);
Full KMatrixSpace of 5 by 10 matrices over Rational Field
```

(c) Defining a sparse $10 \times 10$ matrix over $\mathbf{F}_{101}$:

```
> X := Matrix(GF(101), 10, 10, [<2*i-1, 2*j-1, i*j>: i, j in [1..5]]);
> X;
[  1   0   2   0   3   0   4   0   5   0]
[  0   0   0   0   0   0   0   0   0   0]
[  2   0   4   0   6   0   8   0  10   0]
[  0   0   0   0   0   0   0   0   0   0]
[  3   0   6   0   9   0  12   0  15   0]
[  0   0   0   0   0   0   0   0   0   0]
[  4   0   8   0  12   0  16   0  20   0]
[  0   0   0   0   0   0   0   0   0   0]
[  5   0  10   0  15   0  20   0  25   0]
```

(continues on next page)

(continued from previous page)

```
[  0   0   0   0   0   0   0   0   0   0]
> Parent(X);
Full Matrix Algebra of degree 10 over GF(101)
```

### 27.2.2 Shortcuts

The following functions are "shortcut" versions of the previous general creation function, where some of the arguments are omitted since they can be inferred by Magma.

`Matrix(m, n, Q): RngIntElt, RngIntElt, [ RngElt ] -> Mtrx`

Given integers $m, n \geq 0$ and a sequence $Q$ of length $mn$ containing elements of a ring $R$, return the $m \times n$ matrix over $R$ whose entries are the entries of $Q$, in row-major order. Either of $m$ and $n$ may be 0, in which case $Q$ must have length 0 and some universe $R$. This function is equivalent to `MatrixRing(Universe(Q), n)!Q` if $m = n$ and `RMatrixSpace(Universe(Q), m, n)!Q` otherwise.

`Matrix(m, n, Q): RngIntElt, RngIntElt, [ [ RngElt ] ] -> Mtrx`

Given integers $m$ and $n$, and a sequence $Q$ consisting of $m$ sequences, each of length $n$ and having entries in a ring $R$, return the $m \times n$ matrix over $R$ whose rows are given by the inner sequences of $Q$.

`Matrix(Q): [ Mtrx ] -> Mtrx`

Given a sequence $Q$ of $m$ vectors, each of length $n$ over a ring $R$, return the $m \times n$ matrix over $R$ whose rows are the entries of $Q$.

`Matrix(R, n, Q): Rng, RngIntElt, [ RngElt ] -> Mtrx`

Given a ring $R$, an integer $n \geq 0$, and a sequence $Q$ of length $l$ containing elements of a ring $S$, such that $n$ divides $l$, return the $(l/n) \times n$ matrix over $R$ whose entries are the entries of $Q$, coerced into $R$, in row-major order. The argument $n$ may be 0, in which case $Q$ must have length 0 (and may even be null), in which case the $0 \times 0$ matrix over $R$ is returned. This function is equivalent to `MatrixRing(R, n)!Q` if $l = n^2$ and `RMatrixSpace(R, #Q div n, n)!Q` otherwise.

`Matrix(n, Q): RngIntElt, [ RngElt ] -> Mtrx`

Given an integer $n \geq 0$ and a sequence $Q$ of length $l$ containing elements of a ring $R$, such that $n$ divides $l$, return the $(l/n) \times n$ matrix over $R$ whose entries are the entries of $Q$, in row-major order. The argument $n$ may be 0, in which case $Q$ must have length 0 and some universe $R$, in which case the $0 \times 0$ matrix over $R$ is returned. This function is equivalent to `MatrixRing(Universe(Q), n)!Q` if $l = n^2$ and `RMatrixSpace(Universe(Q), #Q div n, n)!Q` otherwise.

`Matrix(Q): [ [ RngElt ] ] -> Mtrx`

Given a sequence $Q$ consisting of $m$ sequences, each of length $n$ and having entries in a ring $R$, return the $m \times n$ matrix over $R$ whose rows are given by the inner sequences of $Q$.

`Matrix(R, Q): Rng, [ [ RngElt ] ] -> Mtrx`

Given a sequence $Q$ consisting of $m$ sequences, each of length $n$ and having entries in a ring $S$, return the $m \times n$ matrix over $R$ whose rows are given by the inner sequences of $Q$, with the entries coerced into $R$.

**Example: Matrix Construction Shortcuts** (ex-cb789b)

The first matrix in the previous example may be created thus:

```
> X := Matrix(2, [1,2, 3,4]);
> X;
[1 2]
[3 4]
> X := Matrix([[1,2], [3,4]]);
> X;
[1 2]
[3 4]
```

The second matrix in the previous example may be created thus:

```
> X := Matrix(GF(23), 3, [1,-2,3, 4,100,-6]);
> X;
[ 1 21  3]
[ 4  8 17]
> Parent(X);
Full KMatrixSpace of 2 by 3 matrices over GF(23)
> X := Matrix(GF(23), [[1,-2,3], [4,100,-6]]);
> X;
[ 1 21  3]
[ 4  8 17]
> X := Matrix([[GF(23)|1,-2,3], [4,100,-6]]);
> X;
[ 1 21  3]
[ 4  8 17]
```

### 27.2.3 Construction of Structured Matrices

**ZeroMatrix(R, m, n): Rng, RngIntElt, RngIntElt -> Mtrx**

Given a ring $R$ and integers $m, n \geq 0$, return the $m \times n$ zero matrix over $R$.

**IdentityMatrix(R, n): Rng, RngIntElt -> Mtrx**

Given a ring $R$ and integer $n \geq 0$, return the $n \times n$ identity matrix over $R$.

**ScalarMatrix(n, s): RngIntElt, RngElt -> Mtrx**

Given an integer $n \geq 0$ and an element $s$ of a ring $R$, return the $n \times n$ scalar matrix over $R$ which has $s$ on the diagonal and zeros elsewhere. The argument $n$ may be 0, in which case the $0 \times 0$ matrix over $R$ is returned. This function is equivalent to `MatrixRing(Parent(s), n)!s`.

**ScalarMatrix(R, n, s): Rng, RngIntElt, RngElt -> Mtrx**

Given a ring $R$, an integer $n \geq 0$, and an element $s$ of a ring $S$, return the $n \times n$ scalar matrix over $R$ which has $s$, coerced into $R$, on the diagonal and zeros elsewhere. The argument $n$ may be 0, in which case the $0 \times 0$ matrix over $R$ is returned. This function is equivalent to `MatrixRing(R, n)!s`.

**DiagonalMatrix(R, n, Q): Rng, RngIntElt, [ RngElt ] -> Mtrx**

Given a ring $R$, an integer $n \geq 0$, and a sequence $Q$ of $n$ ring elements, return the $n \times n$ diagonal matrix over $R$ whose diagonal entries correspond to the entries of $Q$, coerced into $R$.

**DiagonalMatrix(R, Q): Rng, [ RngElt ] -> Mtrx**

Given a ring $R$ and a sequence $Q$ of $n$ ring elements, return the $n \times n$ diagonal matrix over $R$ whose diagonal entries correspond to the entries of $Q$, coerced into $R$.

**DiagonalMatrix(Q): [ RngElt ] -> Mtrx**

Given a sequence $Q$ of $n$ elements from a ring $R$, return the $n \times n$ diagonal matrix over $R$ whose diagonal entries correspond to the entries of $Q$.

**Matrix(A): Mtrx -> Mtrx**

Given a matrix $A$ of any type, return the same matrix but having as parent the appropriate matrix algebra if $A$ is square, or the appropriate $R$-matrix space otherwise. This is useful, for example, if it is desired to convert a matrix group element or a square $R$-matrix space element to be an element of a general matrix algebra.

**LowerTriangularMatrix(Q): [ RngElt ] -> Mtrx**

Given a sequence $Q$ of length $l$ containing elements of a ring $R$, such that $l = \binom{n+1}{2} = n(n+1)/2$ for some integer $n$ (so $l$ is a triangular number), return the $n \times n$ lower-triangular matrix $F$ over $R$ such that the entries of $Q$ describe the lower triangular part of $F$, in row major order.

`LowerTriangularMatrix(R, Q): Rng, [ RngElt ] -> Mtrx`

Given a ring $R$ and a sequence $Q$ of length $l$ containing elements of a ring $S$, such that $l = \binom{n+1}{2} = n(n+1)/2$ for some integer $n$ (so $l$ is a triangular number), return the $n \times n$ lower-triangular matrix $F$ over $R$ such that the entries of $Q$, coerced into $R$, describe the lower triangular part of $F$, in row major order.

`UpperTriangularMatrix(Q): [ RngElt ] -> Mtrx`

Given a sequence $Q$ of length $l$ containing elements of a ring $R$, such that $l = \binom{n+1}{2} = n(n+1)/2$ for some integer $n$ (so $l$ is a triangular number), return the $n \times n$ upper-triangular matrix $F$ over $R$ such that the entries of $Q$ describe the upper triangular part of $F$, in *row major* order.

`UpperTriangularMatrix(R, Q): Rng, [ RngElt ] -> Mtrx`

Given a ring $R$ and a sequence $Q$ of length $l$ containing elements of a ring $S$, such that $l = \binom{n+1}{2} = n(n+1)/2$ for some integer $n$ (so $l$ is a triangular number), return the $n \times n$ upper-triangular matrix $F$ over $R$ such that the entries of $Q$, coerced into $R$, describe the upper triangular part of $F$, in *row major* order.

`SymmetricMatrix(Q): [ RngElt ] -> Mtrx`

Given a sequence $Q$ of length $l$ containing elements of a ring $R$, such that $l = \binom{n+1}{2} = n(n+1)/2$ for some integer $n$ (so $l$ is a triangular number), return the $n \times n$ symmetric matrix $F$ over $R$ such that the entries of $Q$ describe the lower triangular part of $F$, in row major order. This function allows the creation of symmetric matrices without the need to specify the redundant upper triangular part.

`SymmetricMatrix(R, Q): Rng, [ RngElt ] -> Mtrx`

Given a ring $R$ and a sequence $Q$ of length $l$ containing elements of a ring $S$, such that $l = \binom{n+1}{2} = n(n+1)/2$ for some integer $n$ (so $l$ is a triangular number), return the $n \times n$ symmetric matrix $F$ over $R$ such that the entries of $Q$, coerced into $R$, describe the lower triangular part of $F$, in row major order. This function allows the creation of symmetric matrices without the need to specify the redundant upper triangular part.

`AntisymmetricMatrix(Q): [ RngElt ] -> Mtrx`

Given a sequence $Q$ of length $l$ containing elements of a ring $R$, such that $l = \binom{n}{2} = n(n-1)/2$ for some integer $n$ (so $l$ is a triangular number), return the $n \times n$ antisymmetric matrix $F$ over $R$ such that the entries of $Q$ describe the proper lower triangular part of $F$, in row major order. The diagonal of $F$ is zero and the proper upper triangular part of $F$ is the negation of the proper lower triangular part of $F$.

`AntisymmetricMatrix(R, Q): Rng, [ RngElt ] -> Mtrx`

Given a ring $R$ and a sequence $Q$ of length $l$ containing elements of a ring $S$, such that $l = \binom{n}{2} = n(n-1)/2$ for some integer $n$ (so $l$ is a triangular number), return the $n \times n$ antisymmetric matrix $F$ over $R$ such that the entries of $Q$, coerced into $R$, describe the proper lower triangular part of $F$, in row major order.

`PermutationMatrix(R, Q): Rng, [ RngIntElt ] -> Mtrx`

Given a ring $R$ and a sequence $Q$ of length $n$, such that $Q$ is a permutation of $[1, 2, \ldots, n]$, return the $n$ by $n$ permutation matrix over $R$ corresponding $Q$.

`PermutationMatrix(R, x): Rng, GrpPermElt -> Mtrx`

Given a ring $R$ and a permutation $x$ of degree $n$, return the $n$ by $n$ permutation matrix over $R$ corresponding $x$.

**Example: Scalar and Structured Matrices** `(ex-c66a52)`

This example demonstrates ways of creating special matrices.

(a) Defining a $3 \times 3$ scalar matrix over $\mathbb{Z}$:

```
> S := ScalarMatrix(3, -4);
> S;
[-4  0  0]
[ 0 -4  0]
[ 0  0 -4]
> Parent(S);
Full Matrix Algebra of degree 3 over Integer Ring
```

(b) Defining a $3 \times 3$ diagonal matrix over $\mathbf{F}_{23}$:

```
> D := DiagonalMatrix(GF(23), [1, 2, -3]);
> D;
[ 1  0  0]
[ 0  2  0]
[ 0  0 20]
> Parent(D);
Full Matrix Algebra of degree 3 over GF(23)
```

(c) Defining a $3 \times 3$ symmetric matrix over $\mathbb{Q}$:

```
> S := SymmetricMatrix([1, 1/2,3, 1,3,4]);
> S;
[  1 1/2   1]
[1/2   3   3]
[  1   3   4]
> Parent(S);
Full Matrix Algebra of degree 3 over Rational Field
```

(d) Defining $n \times n$ lower- and upper-triangular matrices for various $n$:

```
> low := func<n | LowerTriangularMatrix([i: i in [1 .. Binomial(n + 1, 2)]])>;
> up := func<n | UpperTriangularMatrix([i: i in [1 .. Binomial(n + 1, 2)]])>;
> sym := func<n | SymmetricMatrix([i: i in [1 .. Binomial(n + 1, 2)]])>;
> low(3);
[1 0 0]
[2 3 0]
[4 5 6]
> up(3);
[1 2 3]
[0 4 5]
[0 0 6]
> sym(3);
[1 2 4]
[2 3 5]
[4 5 6]

> up(6);
[ 1  2  3  4  5  6]
[ 0  7  8  9 10 11]
[ 0  0 12 13 14 15]
[ 0  0  0 16 17 18]
[ 0  0  0  0 19 20]
[ 0  0  0  0  0 21]
```

### 27.2.4 Construction of Random Matrices

`RandomMatrix(R, m, n): Rng, RngIntElt, RngIntElt -> Mtrx`

Given a *finite* ring $R$ and positive integers $m$ and $n$, construct a random $m \times n$ matrix over $R$.

`RandomUnimodularMatrix(n, M): RngIntElt, RngIntElt -> Mtrx`

Given positive integers $M$ and $n$, construct a random integral $n \times n$ matrix having determinant 1 or $-1$. Most entries will lie in the range $[-M, M]$.

`RandomSLnZ(n, k, l): RngIntElt, RngIntElt, RngIntElt -> AlgMatElt`

A random element of $SL_n(\mathbb{Z})$, obtained by multiplying $l$ random matrices of the form $I + E$, where $E$ has exactly one nonzero entry, which is off the diagonal and has absolute value at most $k$.

`RandomGLnZ(n, k, l): RngIntElt, RngIntElt, RngIntElt -> AlgMatElt`

A random element of $GL_n(\mathbb{Z})$, obtained in a similar way to `RandomSLnZ`.

`RandomSymplecticMatrix(g, m): RngIntElt, RngIntElt -> Mtrx`

Given positive integers $n$ and $m$, construct a (somewhat) random $2n \times 2n$ symplectic matrix over the integers. The entries will have the same order of magnitude as $m$.

`RandomSymmetricMatrix(R, n): Rng, RngIntElt -> AlgMatElt`

`RandomSymmetricMatrix(R, n, M): RngOrd, RngIntElt, RngIntElt -> AlgMatElt`

Generate a random symmetric matrix of dimension $n$ over the ring $R$, where each coordinate is in $[-M, M]$.

`RandomPositiveDefiniteSymmetricMatrix(n, M): RngIntElt, RngIntElt -> AlgMatElt`

Generates a random positive definite symmetric matrix over the integers of dimension $n$, with entries in $[-M, M]$.

### 27.2.5 Creating Vectors

`Vector(n, Q): RngIntElt, [ RngElt ] -> ModTupRngElt`

Given an integer $n$ and a sequence $Q$ of length $n$ containing elements of a ring $R$, return the vector of length $n$ whose entries are the entries of $Q$. The integer $n$ may be 0, in which case $Q$ must have length 0 and some universe $R$. This function is equivalent to `RSpace(Universe(Q), n)!Q`.

`Vector(Q): [ RngElt ] -> ModTupRngElt`

Given a sequence $Q$ of length $l$ containing elements of a ring $R$, return the vector of length $l$ whose entries are the entries of $Q$. The argument $Q$ may have length 0 if it has a universe $R$ (i.e., it may not be null). This function is equivalent to `RSpace(Universe(Q), #Q)!Q`.

`Vector(R, n, Q): Rng, RngIntElt, [ RngElt ] -> ModTupRngElt`

Given a ring $R$, an integer $n$, and a sequence $Q$ of length $n$ containing elements of a ring $S$, return the vector of length $n$ whose entries are the entries of $Q$, coerced into $R$. The integer $n$ may be 0, in which case $Q$ must have length 0 (and may even be null). This function is equivalent to `RSpace(R, n)!Q`.

`Vector(R, Q): Rng, [ RngElt ] -> ModTupRngElt`

Given a ring $R$ and a sequence $Q$ of length $l$ containing elements of a ring $S$, return the vector of length $l$ whose entries are the entries of $Q$, coerced into $R$. The argument $Q$ may have length 0 and may be null. This function is equivalent to `RSpace(R, #Q)!Q`.

## 27.3 Elementary Properties

The following functions yield elementary properties of matrices and may be applied to matrices of any type, including vectors.

`NumberOfRows(A): Mtrx -> RngIntElt`

`Nrows(A): Mtrx -> RngIntElt`

Given an $m \times n$ matrix $A$, return $m$, the number of rows of $A$.

`NumberOfColumns(A): Mtrx -> RngIntElt`

`Ncols(A): Mtrx -> RngIntElt`

Given an $m \times n$ matrix $A$, return $n$, the number of columns of $A$.

`NumberOfNonZeroEntries(A): Mtrx -> RngIntElt`

`NNZEntries(A): Mtrx -> RngIntElt`

Given a matrix $A$, return the number of non-zero entries in $A$.

`Density(A): Mtrx -> FldRe`

Given a matrix $A$, return the density of $A$ as a real number, which is the number of non-zero entries in $A$ divided by the product of the number of rows of $A$ and the number of columns of $A$ (or zero if $A$ has zero rows or columns).

`BaseRing(A): Mtrx -> Rng`

`CoefficientRing(A): Mtrx -> Rng`

Given a matrix $A$ with entries lying in a ring $R$, return $R$.

`ElementToSequence(A): Mtrx -> [ RngElt ]`

`Eltseq(A): Mtrx -> [ RngElt ]`

Given a matrix $A$ over the ring $R$ having $m$ rows and $n$ columns, return the entries of $A$, in row-major order, as a sequence of $mn$ elements of $R$.

`RowSequence(A): Mtrx -> [ [RngElt] ]`

Returns the entries of $A$ as a sequence of rows where a row is represented as a sequence of entries of $A$.

## 27.4 Accessing or Modifying Entries

### 27.4.1 Indexing

The following functions and operators enable one to access individual entries or rows of matrices or vectors.

`A[i]: Mtrx, RngIntElt -> ModTupRngElt`

Given a matrix $A$ over the ring $R$ having $m$ rows and $n$ columns, and an integer $i$ such that $1 \leq i \leq m$, return the $i$-th row of $A$, as a vector of length $n$.

`A[i, j]: Mtrx, RngIntElt, RngIntElt -> RngElt`

Given a matrix $A$ over the ring $R$ having $m$ rows and $n$ columns, integers $i$ and $j$ such that $1 \leq i \leq m$ and $1 \leq j \leq n$, return the $(i, j)$-th entry of $A$, as an element of the ring $R$.

`A[Q]: Mtrx, [ RngIntElt ] -> RngElt`

`A[i .. j]: Mtrx, RngIntElt, RngIntElt -> RngElt`

Given a matrix $A$ over the ring $R$ having $m$ rows and $n$ columns, and a sequence $Q$ of integers in the range $[1..m]$, return the sequence consisting of the rows of $A$ specified by $Q$. This is equivalent to `[A[i]: i in Q]`. If $Q$ is a range, then the second form `A[i .. j]` may be used to specify the range directly.

`A[i] := v: Mtrx, RngIntElt, Mtrx`

Given a matrix $A$ over the ring $R$ having $m$ rows and $n$ columns, an integer $i$ such that $1 \leq i \leq m$, and a vector $v$ over $R$ of length $n$, modify the $i$-th row of $A$ to be $v$. The integer 0 may also be given for $v$, indicating the zero vector.

`A[i, j] := x: Mtrx, RngIntElt, RngIntElt, RngElt`

Given a matrix $A$ over the ring $R$ having $m$ rows and $n$ columns, integers $i$ and $j$ such that $1 \leq i \leq m$ and $1 \leq j \leq n$, and a ring element $x$ coercible into $R$, modify the $(i, j)$-th entry of $A$ to be $x$.

**Example: Accessing Matrix Entries** (ex-e9c35f)

This example demonstrates simple ways of accessing the entries of matrices.

```
> X := Matrix(4, [1,2,3,4, 5,4,3,2, 1,2,3,4]);
> X;
[1 2 3 4]
[5 4 3 2]
[1 2 3 4]
> X[1];
(1 2 3 4)
> X[1, 2];
2
> X[1, 2] := 23;
> X;
[ 1 23  3  4]
[ 5  4  3  2]
[ 1  2  3  4]
> X[3] := Vector([9,8,7,6]);
> X[2] := 0;
> X;
[ 1 23  3  4]
[ 0  0  0  0]
[ 9  8  7  6]
```

## 27.4.2 Extracting and Inserting Blocks

The following functions enable the extraction of certain rows, columns, or general submatrices, or the replacement of a block by another matrix.

`Submatrix(A, i, j, p, q): Mtrx, RngIntElt, RngIntElt, RngIntElt, RngIntElt -> Mtrx`

`ExtractBlock(A, i, j, p, q): Mtrx, RngIntElt, RngIntElt, RngIntElt, RngIntElt -> Mtrx`

Given an $m \times n$ matrix $A$ and integers $i$, $j$, $p$ and $q$ such that $1 \leq i \leq i+p \leq m+1$ and $1 \leq j \leq j+q \leq n+1$, return the $p \times q$ submatrix of $A$ rooted at $(i, j)$. Either or both of $p$ and $q$ may be zero, while $i$ may be $m+1$ if $p$ is zero and $j$ may be $n+1$ if $q$ is zero.

`SubmatrixRange(A, i, j, r, s): Mtrx, RngIntElt, RngIntElt, RngIntElt, RngIntElt -> Mtrx`

`ExtractBlockRange(A, i, j, r, s): Mtrx, RngIntElt, RngIntElt, RngIntElt, RngIntElt -> Mtrx`

Given an $m \times n$ matrix $A$ and integers $i$, $j$, $r$, and $s$ such that $1 \leq i$, $i-1 \leq r \leq m$, $1 \leq j$, and $j-1 \leq s \leq n$, return the $r-i+1 \times s-j+1$ submatrix of $A$ rooted at the $(i, j)$-th entry and extending to the $(r, s)$-th entry, inclusive. $r$ may equal $i-1$ or $s$ may equal $j-1$, in which case a matrix with zero rows or zero columns, respectively, will be returned.

`Submatrix(A, I, J): Mtrx, [RngIntElt], [RngIntElt] -> Mtrx`

Given an $m \times n$ matrix $A$ and integer sequences $I$ and $J$, return the submatrix of $A$ given by the row indices in $I$ and the column indices in $J$.

`InsertBlock(A, B, i, j): Mtrx, Mtrx, RngIntElt, RngIntElt -> Mtrx`

`InsertBlock(~A, B, i, j): Mtrx, Mtrx, RngIntElt, RngIntElt -> Mtrx`

Given an $m \times n$ matrix $A$ over a ring $R$, a $p \times q$ matrix $B$ over $R$, and integers $i$ and $j$ such that $1 \leq i \leq i+p \leq m+1$ and $1 \leq j \leq j+q \leq n+1$, insert $B$ at position $(i, j)$ in $A$. In the functional version ($A$ is a value argument), this function returns the new matrix and leaves $A$ untouched, while in the procedural version ($\sim A$ is a reference argument), $A$ is modified in place so that the $p \times q$ submatrix of $A$ rooted at $(i, j)$ is now equal to $B$.

`RowSubmatrix(A, i, k): Mtrx, RngIntElt, RngIntElt -> Mtrx`

Given an $m \times n$ matrix $A$ and integers $i$ and $k$ such that $1 \leq i \leq i+k \leq m+1$, return the $k \times n$

submatrix of $X$ consisting of rows $[i \dots i+k-1]$ inclusive. The integer $k$ may be zero and $i$ may also be $m+1$ if $k$ is zero, but the result will always have $n$ columns.

**RowSubmatrix(A, i): Mtrx, RngIntElt -> Mtrx**

Given an $m \times n$ matrix $A$ and an integer $i$ such that $0 \leq i \leq m$, return the $i \times n$ submatrix of $X$ consisting of the first $i$ rows. The integer $i$ may be 0, but the result will always have $n$ columns.

**RowSubmatrixRange(A, i, j): Mtrx, RngIntElt, RngIntElt -> Mtrx**

Given an $m \times n$ matrix $A$ and integers $i$ and $j$ such that $1 \leq i$ and $i-1 \leq j \leq m$, return the $j-i+1 \times n$ submatrix of $X$ consisting of rows $[i \dots j]$ inclusive. The integer $j$ may equal $i-1$, in which case a matrix with zero rows and $n$ columns will be returned.

**ColumnSubmatrix(A, i, k): Mtrx, RngIntElt, RngIntElt -> Mtrx**

Given an $m \times n$ matrix $A$ and integers $i$ and $k$ such that $1 \leq i \leq i+k \leq n+1$, return the $m \times k$ submatrix of $X$ consisting of columns $[i \dots i+k-1]$ inclusive. The integer $k$ may be zero and $i$ may also be $n+1$ if $k$ is zero, but the result will always have $m$ rows.

**ColumnSubmatrix(A, i): Mtrx, RngIntElt -> Mtrx**

Given an $m \times n$ matrix $A$ and an integer $i$ such that $0 \leq i \leq n$, return the $m \times i$ submatrix of $X$ consisting of the first $i$ columns. The integer $i$ may be 0, but the result will always have $m$ rows.

**ColumnSubmatrixRange(A, i, j): Mtrx, RngIntElt, RngIntElt -> Mtrx**

Given an $m \times n$ matrix $A$ and integers $i$ and $j$ such that $1 \leq i$ and $i-1 \leq j \leq n$, return the $m \times j-i+1$ submatrix of $X$ consisting of columns $[i \dots j]$ inclusive. The integer $j$ may equal $i-1$, in which case a matrix with zero columns and $n$ rows will be returned.

**Example: Extracting Submatrices** (ex-0ed08c)

The use of the submatrix operations is illustrated by applying them to a $6 \times 6$ matrix over the ring of integers $\mathbb{Z}$.

```
> A := Matrix(6,
>     [ 9, 1, 7, -3, 2, -1,
>       3, -4, -5, 9, 2, 7,
>       7, 1, 0, 1, 8, 22,
>       -3, 3, 3, 8, 8, 37,
>       -9, 0, 7, -1, 2, 3,
>       7, 2, -2, 4, 3, 47 ]);
> A;
[ 9  1  7 -3  2 -1]
[ 3 -4 -5  9  2  7]
[ 7  1  0  1  8 22]
[-3  3  3  8  8 37]
[-9  0  7 -1  2  3]
[ 7  2 -2  4  3 47]
> Submatrix(A, 2,2, 3,3);
[-4 -5  9]
[ 1  0  1]
[ 3  3  8]
> SubmatrixRange(A, 2,2, 3,3);
[-4 -5]
[ 1  0]
> S := $1;
> InsertBlock(~A, S, 5,5);
> A;
[ 9  1  7 -3  2 -1]
[ 3 -4 -5  9  2  7]
[ 7  1  0  1  8 22]
[-3  3  3  8  8 37]
[-9  0  7 -1 -4 -5]
```

(continues on next page)

(continued from previous page)

```
[ 7  2 -2  4  1  0]
> RowSubmatrix(A, 5, 2);
[-9  0  7 -1 -4 -5]
[ 7  2 -2  4  1  0]
> RowSubmatrixRange(A, 2, 3);
[ 3 -4 -5  9  2  7]
[ 7  1  0  1  8 22]
> RowSubmatrix(A, 2, 0);
Matrix with 0 rows and 6 columns
```

### 27.4.3 Row and Column Operations

The following functions and procedures provide elementary row or column operations on matrices. For each operation, there is a corresponding function which creates a new matrix for the result (leaving the input matrix unchanged), and a corresponding procedure which modifies the input matrix in place.

**SwapRows(A, i, j): Mtrx, RngIntElt, RngIntElt -> Mtrx**

**SwapRows(~A, i, j): Mtrx, RngIntElt, RngIntElt**

Given an $m \times n$ matrix $A$ and integers $i$ and $j$ such that $1 \leq i \leq m$ and $1 \leq j \leq m$, swap the $i$-th and $j$-th rows of $A$.

**SwapColumns(A, i, j): Mtrx, RngIntElt, RngIntElt -> Mtrx**

**SwapColumns(~A, i, j): Mtrx, RngIntElt, RngIntElt**

Given an $m \times n$ matrix $A$ and integers $i$ and $j$ such that $1 \leq i \leq n$ and $1 \leq j \leq n$, swap the $i$-th and $j$-th columns of $A$.

**ReverseRows(A): Mtrx -> Mtrx**

**ReverseRows(~A): Mtrx**

Given a matrix $A$, reverse all the rows of $A$.

**ReverseColumns(A): Mtrx -> Mtrx**

**ReverseColumns(~A): Mtrx**

Given a matrix $A$, reverse all the columns of $A$.

**AddRow(A, c, i, j): Mtrx, RngElt, RngIntElt, RngIntElt -> Mtrx**

**AddRow(~A, c, i, j): Mtrx, RngElt, RngIntElt, RngIntElt**

Given an $m \times n$ matrix $A$ over a ring $R$, a ring element $c$ coercible into $R$, and integers $i$ and $j$ such that $1 \leq i \leq m$ and $1 \leq j \leq m$, add $c$ times row $i$ of $A$ to row $j$ of $A$.

**AddColumn(A, c, i, j): Mtrx, RngElt, RngIntElt, RngIntElt -> Mtrx**

**AddColumn(~A, c, i, j): Mtrx, RngElt, RngIntElt, RngIntElt**

Given an $m \times n$ matrix $A$ over a ring $R$, a ring element $c$ coercible into $R$, and integers $i$ and $j$ such that $1 \leq i \leq n$ and $1 \leq j \leq n$, add $c$ times column $i$ of $A$ to column $j$.

**MultiplyRow(A, c, i): Mtrx, RngElt, RngIntElt -> Mtrx**

**MultiplyRow(~A, c, i): Mtrx, RngElt, RngIntElt**

Given an $m \times n$ matrix $A$ over a ring $R$, a ring element $c$ coercible into $R$, and an integer $i$ such that $1 \leq i \leq m$, multiply row $i$ of $A$ by $c$ (on the left).

**MultiplyColumn(A, c, i): Mtrx, RngElt, RngIntElt -> Mtrx**

**MultiplyColumn(~A, c, i): Mtrx, RngElt, RngIntElt**

Given an $m \times n$ matrix $A$ over a ring $R$, a ring element $c$ coercible into $R$, and an integer $i$ such that $1 \leq i \leq n$, multiply column $i$ of $A$ by $c$ (on the left).

**RemoveRow(A, i): Mtrx, RngIntElt -> Mtrx**

`RemoveRow(~A, i): Mtrx, RngIntElt`

Given an $m \times n$ matrix $A$ and an integer $i$ such that $1 \leq i \leq m$, remove row $i$ from $A$ (leaving an $(m-1) \times n$ matrix).

`RemoveColumn(A, j): Mtrx, RngIntElt -> Mtrx`

`RemoveColumn(~A, j): Mtrx, RngIntElt`

Given an $m \times n$ matrix $A$ and an integer $j$ such that $1 \leq j \leq n$, remove column $j$ from $A$ (leaving an $m \times (n-1)$ matrix).

`RemoveRowColumn(A, i, j): Mtrx, RngIntElt -> Mtrx`

`RemoveRowColumn(~A, i, j): Mtrx, RngIntElt`

Given an $m \times n$ matrix $A$ and integers $i$ and $j$ such that $1 \leq i \leq m$ and $1 \leq j \leq n$, remove row $i$ and column $j$ from $A$ (leaving an $(m-1) \times (n-1)$ matrix).

`RemoveZeroRows(A): Mtrx -> Mtrx`

`RemoveZeroRows(~A): Mtrx`

Given a matrix $A$, remove all the zero rows of $A$.

**Example: Row and Column Operations** (ex-80d37c)

The use of row and column operations is illustrated by applying them to a $5 \times 6$ matrix over the ring of integers $\mathbb{Z}$.

```
> A := Matrix(5, 6,
>     [ 3, 1, 0, -4, 2, -12,
>       2, -4, -5, 5, 23, 6,
>       8, 0, 0, 1, 5, 12,
>       -2, -6, 3, 8, 9, 17,
>       11, 12, -6, 4, 2, 27 ]);
> A;
[  3   1   0  -4   2 -12]
[  2  -4  -5   5  23   6]
[  8   0   0   1   5  12]
[ -2  -6   3   8   9  17]
[ 11  12  -6   4   2  27]
> SwapColumns(~A, 1, 2);
> A;
[  1   3   0  -4   2 -12]
[ -4   2  -5   5  23   6]
[  0   8   0   1   5  12]
[ -6  -2   3   8   9  17]
[ 12  11  -6   4   2  27]
> AddRow(~A, 4, 1, 2);
> AddRow(~A, 6, 1, 4);
> AddRow(~A, -12, 1, 5);
> A;
[  1   3   0  -4   2 -12]
[  0  14  -5 -11  31 -42]
[  0   8   0   1   5  12]
[  0  16   3 -16  21 -55]
[  0 -25  -6  52 -22 171]
> RemoveRow(~A, 1);
> A;
[ 2 -4 -5  5 23  6]
[ 8  0  0  1  5 12]
[-2 -6  3  8  9 17]
[11 12 -6  4  2 27]
> RemoveRowColumn(~A, 4, 6);
```

(continues on next page)

(continued from previous page)

```
> A;
[ 2 -4 -5  5 23]
[ 8  0  0  1  5]
[-2 -6  3  8  9]
```

## 27.5 Building Block Matrices

Block matrices can be constructed either by listing the blocks, or by joining together smaller matrices horizontally, vertically, or diagonally.

**`BlockMatrix(m, n, blocks): RngIntElt, RngIntElt, [ Mtrx ] -> Mtrx`**

The matrix constructed from the given block matrices, which should all have the same dimensions, and should be given as a sequence of $m \cdot n$ block matrices (given in row major order, in other words listed across rows).

**`BlockMatrix(m, n, rows): RngIntElt, RngIntElt, [ [ Mtrx ] ] -> Mtrx`**

**`BlockMatrix(rows): [ [ Mtrx ] ] -> Mtrx`**

The matrix constructed from the given block matrices, which should all have the same dimensions, and should be given as a sequence of $m$ rows, each containing $n$ block matrices.

**`HorizontalJoin(X, Y): Mtrx, Mtrx -> Mtrx`**

Given a matrix $X$ with $r$ rows and $c$ columns, and a matrix $Y$ with $r$ rows and $d$ columns, both over the same coefficient ring $R$, return the matrix over $R$ with $r$ rows and $(c+d)$ columns obtained by joining $X$ and $Y$ horizontally (placing $Y$ to the right of $X$).

**`HorizontalJoin(Q): [ Mtrx ] -> Mtrx`**

**`HorizontalJoin(T): < Mtrx > -> Mtrx`**

Given a sequence $Q$ or tuple $T$ of matrices, each having the same number of rows and being over the same coefficient ring $R$, return the matrix over $R$ obtained by joining the elements of $Q$ or $T$ horizontally in order.

**`VerticalJoin(X, Y): Mtrx, Mtrx -> Mtrx`**

Given a matrix $X$ with $r$ rows and $c$ columns, and a matrix $Y$ with $s$ rows and $c$ columns, both over the same coefficient ring $R$, return the matrix with $(r+s)$ rows and $c$ columns over $R$ obtained by joining $X$ and $Y$ vertically (placing $Y$ underneath $X$).

**`VerticalJoin(Q): [ Mtrx ] -> Mtrx`**

**`VerticalJoin(T): < Mtrx > -> Mtrx`**

Given a sequence $Q$ or tuple $T$ of matrices, each having the same number of columns and being over the same coefficient ring $R$, return the matrix over $R$ obtained by joining the elements of $Q$ or $T$ vertically in order.

**`DiagonalJoin(X, Y): Mtrx, Mtrx -> Mtrx`**

Given matrices $X$ with $a$ rows and $b$ columns and $Y$ with $c$ rows and $d$ columns, both over the same coefficient ring $R$, return the matrix with $(a+c)$ rows and $(b+d)$ columns over $R$ obtained by joining $X$ and $Y$ diagonally (placing $Y$ diagonally to the right of and underneath $X$, with zero blocks above and below the diagonal).

**`DiagonalJoin(Q): [ Mtrx ] -> Mtrx`**

**`DiagonalJoin(T): < Mtrx > -> Mtrx`**

Given a sequence $Q$ or tuple $T$ of matrices, each being over the same coefficient ring $R$, return the matrix over $R$ obtained by joining the elements of $Q$ or $T$ diagonally in order.

**`KroneckerProduct(A, B): Mtrx, Mtrx -> Mtrx`**

Given an $m \times n$ matrix $A$ and a $p \times q$ matrix $B$, both over a ring $R$, return the Kronecker product of $A$ and $B$, which is the $mp \times nq$ matrix $C$ over $R$ such that the $((i-1)p+r, (j-1)q+s)$-th entry of $C$ is the $(i,j)$-th entry of $A$ times the $(r,s)$-th entry of $B$, for $1 \leq i \leq m$, $1 \leq j \leq n$, $1 \leq r \leq p$, and $1 \leq s \leq q$.

## 27.6 Changing Ring

`ChangeRing(A, R): Mtrx, Rng -> Mtrx`

`Matrix(R, A): Rng, Mtrx -> Mtrx`

Given a matrix $A$ over a ring $S$ having $m$ rows and $n$ columns, and another ring $R$, return the $m \times n$ matrix over $R$ obtained by coercing the entries of $A$ from $S$ into $R$. The argument order to `ChangeRing(A, R)` here is consistent with other forms of `ChangeRing`, while the `Matrix(R, A)` form of this function is provided to be consistent with the matrix creation functions above, for which the destination ring is the first argument, if supplied.

`ChangeRing(A, R, f): Mtrx, Rng, Map -> Mtrx`

`ChangeRing(A, f): Mtrx, Map -> Mtrx`

Given a matrix $A$ over a ring $S$ having $m$ rows and $n$ columns, another ring $R$, and a map $f : S \to R$, return the $m \times n$ matrix over $R$ obtained by applying $f$ to each of the entries of $A$. The $R$ may be omitted, in which case it is taken to be the codomain of $f$.

`CanChangeRing(A, R): Mtrx, Rng -> BoolElt, Mtrx`

Return whether the entries in the matrix $A$ can be coerced into $R$ and the matrix resulting from `ChangeRing(A, R)` if so.

## 27.7 Elementary Arithmetic

`A + B: Mtrx, Mtrx -> Mtrx`

Given $m \times n$ matrices $A$ and $B$ over a ring $R$, return $A + B$.

`A - B: Mtrx, Mtrx -> Mtrx`

Given $m \times n$ matrices $A$ and $B$ over a ring $R$, return $A - B$.

`A * B: Mtrx, Mtrx -> Mtrx`

Given an $m \times n$ matrix $A$ over a ring $R$ and an $n \times p$ matrix $B$ over $R$, return the $m \times p$ matrix $A \cdot B$ over $R$. This function attempts to preserve the maximal amount of information in the choice of parent for the product. For example, if $A$ and $B$ are both square and have the same matrix algebra $M$ as parent, then the product will also have $M$ as parent. Similarly, if the parents of $A$ and $B$ are $R$-matrix spaces such that the codomain of $B$ equals the domain $A$, then the product will have domain equal to that of $A$ and codomain equal to that of $B$.

`x * A: RngElt, Mtrx -> Mtrx`

`A * x: Mtrx, RngElt -> Mtrx`

Given an $m \times n$ matrix $A$ over a ring $R$ and a ring element $x$ coercible into $R$, return the scalar product $x \cdot A$.

Note that if $x$ is not coercible into the ring $R$ then $A$ may be automatically coerced into the ring of $x$. On the other hand, one particular common case is that multiplying an integral matrix by the real number 1.0 will **not** coerce the matrix into the reals, as the scalar is coercible into the ring $R$ (the integers) in that case.

`- A: Mtrx -> Mtrx`

Given a matrix $A$, return $-A$.

`A ^ -1: Mtrx, RngIntElt -> Mtrx`

Given a invertible square matrix $A$ over a ring $R$, return the inverse $B$ of $A$ so that $A \cdot B = B \cdot A = 1$. The coefficient ring $R$ must be either a field, a Euclidean domain, or a ring with an exact division algorithm and having characteristic equal to zero or greater than $m$ (this includes most commutative rings).

`A ^ n: Mtrx, RngIntElt -> Mtrx`

Given a square matrix $A$ over a ring $R$ and an integer $n$, return the matrix power $A^n$. $A^0$ is defined to be the identity matrix for any square matrix $A$ (even if $A$ is zero). If $n$ is negative, $A$ must be invertible (see the previous function), and the result is $(A^{-1})^{-n}$.

`Transpose(A): Mtrx -> Mtrx`

Given an $m \times n$ matrix $A$ over a ring $R$, return the transpose of $A$, which is simply the $n \times m$ matrix over $R$ whose $(i, j)$-th entry is the $(j, i)$-th entry of $A$.

`AddScaledMatrix(A, s, B): Mtrx, RngElt, Mtrx -> Mtrx`

Given a matrix $A$ over a ring $R$, a scalar $s$ coercible into $R$, and a matrix $B$ over $R$ with the same shape as $A$, return $A + s \cdot B$. This is generally quicker than the call `A + s*B`.

`AddScaledMatrix(~A, s, B): Mtrx, RngElt, Mtrx`

Given a matrix $A$ over a ring $R$, a scalar $s$ coercible into $R$, and a matrix $B$ over $R$ with the same shape as $A$, set $A$ to $A + s \cdot B$. This is generally quicker than the statement `A := A + s*B;`.

## 27.8 Nullspaces and Solutions of Systems

The following functions compute nullspaces of matrices (solving equations of the form $V \cdot A = 0$), or solve systems of the form $V \cdot A = W$, for given $A$ and $W$.

Magma possesses a very rich suite of internal algorithms for computing nullspaces of matrices efficiently, including a fast $p$-adic algorithm for matrices over $\mathbb{Z}$ and $\mathbb{Q}$, and also algorithms which take advantage of sparsity if it is present.

`Nullspace(A): Mtrx -> ModTupRng`

`Kernel(A): Mtrx -> ModTupRng, Map`

Given an $m \times n$ matrix $A$ over a ring $R$, return the nullspace of $A$ (or the kernel of $A$, considered as a linear transformation or map), which is the $R$-space consisting of all vectors $v$ of length $m$ such that $v \cdot A = 0$. If the parent of $A$ is an $R$-matrix space, then the result will be the appropriate submodule of the domain of $A$. The function `Kernel(A)` also returns the inclusion map from the kernel into the domain of $A$, to be consistent with other forms of the `Kernel` function.

`NullspaceMatrix(A): Mtrx -> ModTupRng`

`KernelMatrix(A): Mtrx -> ModTupRng`

Given an $m \times n$ matrix $A$ over a ring $R$, return a basis matrix of the nullspace of $A$. This is a matrix $N$ having $m$ columns and the maximal number of independent rows subject to the condition that $N \cdot A = 0$. This function has the advantage that the nullspace is not returned as a $R$-space, so echelonization of the resulting nullspace may be avoided.

`NullspaceOfTranspose(A): Mtrx -> ModTupRng`

This function is equivalent to `Nullspace(Transpose(A))`, but may be more efficient in space for large matrices, since the transpose may not have to be explicitly constructed to compute the nullspace.

`IsConsistent(A, W): Mtrx, Mtrx -> BoolElt, Mtrx, ModTupRng`

Given an $m \times n$ matrix $A$ over a ring $R$, and a vector $W$ of length $n$ over $R$ or a $r \times n$ matrix $W$ over $R$, return `true` iff and only if the system of linear equations $V \cdot A = W$ is consistent. If the system is consistent, then the function will also return:

**(a)** A particular solution $V$ so that $V \cdot A = W$;

**(b)** The nullspace $N$ of $A$ so that adding any elements of $N$ to any rows of $V$ will yield other solutions to the system.

`IsConsistent(A, Q): Mtrx, [ ModTupRng ] -> BoolElt, [ ModTupRngElt ], ModTupRng`

Given an $m \times n$ matrix $A$ over a ring $R$, and a sequence $Q$ of vectors of length $n$ over $R$, return `true` if and only if the system of linear equations $V[i] * A = Q[i]$ for all $i$ is consistent. If the system is consistent, then the function will also return:

**(a)** A particular solution sequence $V$;

**(b)** The nullspace $N$ of $A$ so that $(V[i] + u) * A = Q[i]$ for $u \in N$ for all $i$.

`Solution(A, W): ModMatRngElt, ModTupRng -> ModTupRngElt, ModTupRng`

Given an $m \times n$ matrix $A$ over a ring $R$, and a vector $W$ of length $n$ over $R$ or a $r \times n$ matrix $W$ over $R$, solve the system of linear equations $V \cdot A = W$ and return:

**(a)** A particular solution $V$ so that $V \cdot A = W$;

**(b)** The nullspace $N$ of $A$ so that adding any elements of $N$ to any rows of $V$ will yield other solutions to the system. If there is no solution, an error results.

`Solution(A, Q): ModMatRngElt, [ ModTupRng ] -> [ ModTupRngElt ], ModTupRng`

Given an $m \times n$ matrix $A$ over a ring $R$, and a sequence $Q$ of vectors of length $n$ over $R$, solve the system of linear equations $V[i] * A = Q[i]$ for each $i$ and return:

**(a)** A particular solution sequence $V$;

**(b)** The nullspace $N$ of $A$ so that $(V[i] + u) * A = Q[i]$ for $u \in N$ for all $i$. If there is no solution, an error results.

**Example: Nullspace of a Random Integer Matrix** (ex-a44c79)

We compute the nullspace of a $301 \times 300$ matrix over $\mathbb{Z}$ with random entries in the range [0..10]. The nullity is 1 and the entries of the non-zero null vector are integers, each having about 455 decimal digits.

```
> m := 301; n := 300;
> X := Matrix(n, [Random(0, 10): i in [1 .. m*n]]);
> time N := NullspaceMatrix(X);
Time: 9.519
> Nrows(N), Ncols(N);
1 301
> time IsZero(N*X);
true
Time: 0.429
> {#Sprint(N[1,i]): i in [1..301]};
{ 452, 455, 456, 457, 458 }
```

**Example: Enumerating All Solutions of a System** (ex-83882a)

We show how one can enumerate all solutions to the system $V \cdot X = W$ for a given matrix $X$ and vector $W$ over a finite field. The `Solution` function gives a particular solution for $V$, and then adding this to every element in the nullspace $N$ of $X$, we obtain all solutions.

```
> K := GF(3);
> X := Matrix(K, 4, 3, [1,2,1, 2,2,2, 1,1,1, 1,0,1]);
> X;
[1 2 1]
[2 2 2]
[1 1 1]
[1 0 1]
> W := Vector(K, [0,1,0]);
> V, N := Solution(X, W);
> V;
(1 1 0 0)
> N;
Vector space of degree 4, dimension 2 over GF(3)
Echelonized basis:
(1 0 1 1)
(0 1 1 0)
> [V + U: U in N];
```

(continues on next page)

(continued from previous page)

```
[
    (1 1 0 0),
    (2 1 1 1),
    (0 1 2 2),
    (0 2 0 2),
    (1 2 1 0),
    (2 2 2 1),
    (2 0 0 1),
    (0 0 1 2),
    (1 0 2 0)
]
> [(V + U)*X eq W: U in N];
[ true, true, true, true, true, true, true, true, true ]
```

## 27.9 Predicates

The functions in this section test various properties of matrices. See also the Lattices chapter for a description of the function `IsPositiveDefinite(F): Mtrx → BoolElt` and related functions.

**`IsZero(A): Mtrx -> BoolElt`**

Given an $m \times n$ matrix $A$ over the ring $R$, return `true` iff $A$ is the $m \times n$ zero matrix.

**`IsOne(A): Mtrx -> BoolElt`**

Given a square $m \times m$ matrix $A$ over the ring $R$, return `true` iff $A$ is the $m \times m$ identity matrix.

**`IsMinusOne(A): Mtrx -> BoolElt`**

Given a square $m \times m$ matrix $A$ over the ring $R$, return `true` iff $A$ is the negation of the $m \times m$ identity matrix.

**`IsScalar(A): Mtrx -> BoolElt`**

Given a square $m \times m$ matrix $A$ over the ring $R$, return `true` iff $A$ is scalar, i.e., iff $A$ is the product of some element of $R$ and the $m \times m$ identity matrix.

**`IsDiagonal(A): Mtrx -> BoolElt`**

Given a square matrix $A$ over the ring $R$, return `true` iff $A$ is diagonal, i.e., iff the only non-zero entries of $A$ are on the diagonal.

**`IsSymmetric(A): Mtrx -> BoolElt`**

Given a square matrix $A$ over the ring $R$, return `true` iff $A$ is symmetric, i.e., iff $A$ equals its transpose.

**`IsHermitian(A, f): Mtrx, Map -> BoolElt`**

Given a square matrix $A$ over the ring $R$, and an involution $f : R \to R$, return `true` iff $A$ is Hermitian with respect to $f$, i.e., iff $A$ equals applying $f$ to its transpose.

**`IsUpperTriangular(A): Mtrx -> BoolElt`**

Given a matrix $A$ over the ring $R$, return `true` iff $A$ is upper triangular, i.e., iff the only non-zero entries of $A$ are on or above the diagonal.

**`IsLowerTriangular(A): Mtrx -> BoolElt`**

Given a matrix $A$ over the ring $R$, return `true` iff $A$ is lower triangular, i.e., iff the only non-zero entries of $A$ are on or below the diagonal.

**`IsUnit(A): Mtrx -> BoolElt`**

Given a square matrix $A$ over the ring $R$, return `true` iff $A$ is a unit, i.e., iff $A$ has an inverse. The coefficient ring $R$ may be any commutative ring (since the computation depends on testing if the determinant is a unit – a calculation which is supported in all commutative rings).

`IsSingular(A): Mtrx -> BoolElt`

Given a square $m \times m$ matrix $A$ over the ring $R$, return `true` iff $A$ is singular, i.e., iff the determinant of $A$ is zero (or, equivalently, iff the rank of $A$ is less than $m$). Note that `(not IsSingular(A))` is *not* equivalent to `IsUnit(A)` whenever $R$ is not a field: if the determinant of $A$ is non-zero but not a unit, then $A$ is non-singular but not invertible. The coefficient ring $R$ may be any commutative ring (since the computation involves only computing the determinant and testing whether it is zero).

`IsSymplecticMatrix(A): Mtrx -> BoolElt`

Given an $m \times m$ matrix $A$ over the integers, return `true` if and only if $A$ is an integer symplectic matrix, that is, $AJ^tA = J$, where $J = \begin{pmatrix} 0 & \mathbf{1}_g \\ -\mathbf{1}_g & 0 \end{pmatrix}$.

## 27.10 Determinant and Other Properties

`Determinant(A: parameters): Mtrx -> RngElt`

| | | |
|---|---|---|
| MonteCarloLevel | : RngIntElt | Default: 0 |
| Proof | : BoolElt | Default: true |
| pAdic | : BoolElt | Default: true |
| Divisor | : RngIntElt | Default: 0 |

Given a square matrix $A$ over the ring $R$, return the determinant of $A$ as an element of $R$. $R$ may be any commutative ring. The determinant of the $0 \times 0$ matrix over $R$ is defined to be `R!1`. If the coefficient ring is the integer ring $\mathbb{Z}$ or the rational field $\mathbb{Q}$ then a modular algorithm based on that of Abbott et al. [ABM99] is used, which first computes a divisor $d$ of the determinant $D$ using a fast $p$-adic nullspace computation, and then computes the quotient $D/d$ by computing the determinant $D$ modulo enough small primes to cover the Hadamard bound divided by $d$. This always yields a correct answer. If the parameter `MonteCarloLevel` is set to a small positive integer $s$, then a probabilistic Monte-Carlo modular technique is used. Rather than using sufficient primes to cover the Hadamard bound divided by the divisor $d$, this version of the algorithm terminates when the constructed residue remains constant for $s$ steps. The probability of this being wrong is non-zero but extremely small, even if $s$ is only 1 or 2. If the level is set to 0, then the normal deterministic algorithm is used. Setting the parameter `Proof` to `false` is equivalent to setting `MonteCarloLevel` to 2. If the coefficient ring is $\mathbb{Z}$ and the parameter `Divisor` is set to an integer $d$, then $d$ must be a known exact divisor of the determinant (the sign does not matter), and the algorithm may be sped up because of this knowledge.

`Trace(A): Mtrx -> RngElt`

Given a square matrix $A$ over the ring $R$, return the trace of $A$ as an element of $R$, which is simply the sum of the diagonal elements of $A$.

`TraceOfProduct(A, B): Mtrx, Mtrx -> RngElt`

Given square matrices $A$ and $B$ over the ring $R$, with the same size, return the trace of $A \cdot B$ as an element of $R$. This is in general much faster than the call `Trace(A*B)`.

`Rank(A): Mtrx -> RngIntElt`

Given an $m \times n$ matrix $A$ over a ring $R$, return the rank of $A$. This is defined to be the largest $r$ such that there exists a non-zero $r \times r$ subdeterminant of $A$, so $r \leq m$ and $r \leq n$. The rank may have to be obtained by computing the Smith form or echelon form of $A$, and this computation may be quite expensive over some rings.

`Minor(M, i, j): Mtrx, RngIntElt, RngIntElt -> RngElt`

The determinant of the submatrix of $M$ (which must be square) formed by removing the $i$-th row and $j$-th column.

`Minor(M, I, J): Mtrx, [RngIntElt], [RngIntElt] -> RngElt`

The determinant of the submatrix of $M$ given by the row indices in $I$ and the column indices in $J$.

`Minors(M, r): Mtrx, RngIntElt -> SeqEnum`

Returns a sequence of all the $r$ by $r$ minors of the matrix $M$.

`Cofactor(M, i, j): Mtrx, RngIntElt, RngIntElt -> RngElt`

The appropriate cofactor of $M$, equal to $(-1)^{i+j}$ times the corresponding minor.

`Cofactors(M): Mtrx -> SeqEnum`

Returns a sequence of all the cofactors of the matrix $M$.

`Cofactors(M, r): Mtrx, RngIntElt -> SeqEnum`

Returns a sequence of all the $r$ by $r$ cofactors of the matrix $M$.

`Pfaffian(M): Mtrx -> RngElt`

`Pfaffian(M, I, J): Mtrx, [RngIntElt], [RngIntElt] -> RngElt`

`Pfaffians(M, r): Mtrx, RngIntElt -> SeqEnum`

Let $M$ be an anti-symmetric square matrix. Then its determinant is always a square and a particular square-root of this, which can be described by a universal polynomial in its entries, is called the Pfaffian of $M$. The first function returns this. The second function returns the Pfaffian of the submatrix of $M$ described by the indices in $I$ and $J$. The third function returns the sequence of Pfaffians of the $\binom{n}{r}$ principal $r$ by $r$ submatrices of $M$ ($n$ = the number of rows of $M$).

These are primarily convenience functions and are computed naively by Pfaffian row-expansion.

## 27.11 Minimal and Characteristic Polynomials and Eigenvalues

The functions in this section deal with minimal and characteristic polynomials.

`MinimalPolynomial(A: parameters): Mtrx -> RngUPolElt`

```
Proof: BoolElt                    Default: true
```

Given a square matrix $A$ over a ring $R$, return the minimal polynomial of $A$. This is defined to be the unique monic univariate polynomial $f(x)$ of minimal degree such that $f(A) = 0$, and $f(x)$ always divides the characteristic polynomial of $A$. The coefficient ring $R$ is currently restricted to being a field or the integer ring $\mathbb{Z}$. Setting the parameter `Proof` to false suppresses proof of correctness.

`CharacteristicPolynomial(A: parameters): Mtrx -> RngUPolElt`

```
Al   : MonStg                     Default: "Modular"
Proof: BoolElt                    Default: true
```

Given a square matrix $A$ over a ring $R$, return the characteristic polynomial of $A$. This is defined to be the monic univariate polynomial $\mathrm{Det}(x - A) \in R[x]$ where $R[x]$ is the univariate polynomial ring over $R$. $R$ may be any commutative ring. The parameter `Al` allows the user to specify which algorithm that is to be employed. The algorithm `"Modular"` (the default) may be used for matrices over $\mathbb{Z}$ and $\mathbb{Q}$—in such a case the parameter `Proof` can also be used to suppress proof of correctness. The algorithm `"Hessenberg"`, available for matrices over fields, works by first reducing the matrix to Hessenberg form. The algorithm `"Interpolation"`, available for matrices over $Z$ and $Q$, works by evaluating the characteristic matrix of $a$ at various points and then interpolating. The algorithm `"Trace"`, available for matrices over fields, works by calculating the traces of powers of $a$. Since V2.8, none of these algorithms are now recommended for matrices over $\mathbb{Z}$ or $\mathbb{Q}$, as the new $p$-adic modular algorithm over the integers is extremely fast.

`MinimalAndCharacteristicPolynomials(A: parameters): Mtrx -> RngUPolElt, RngUPolElt`

`MCPolynomials(A): Mtrx -> RngUPolElt, RngUPolElt`

```
Proof: BoolElt                    Default: true
```

Given a square matrix $A$ over a ring $R$, return the minimal and characteristic polynomials of $A$. For some rings, both polynomials can be computed at the same time, so in such cases it will be more efficient to use this function than to call `MinimalPolynomial` and `CharacteristicPolynomials` separately. Setting the parameter `Proof` to false suppresses proof of correctness.

`FactoredMinimalPolynomial(A: parameters): Mtrx -> [ <RngUPolElt, RngIntElt>]`

| | | | |
|---|---|---|---|
| `Proof:` | `BoolElt` | `Default:` | `true` |

Given a square matrix $A$ over a ring $R$, return the factorization of the minimal polynomial of $A$. This is equivalent to `Factorization(MinimalPolynomial(A))`, but may be faster than that for some coefficient rings (in particular, $\mathbb{Z}$ and $\mathbb{Q}$). Setting the parameter `Proof` to false suppresses proof of correctness.

`FactoredCharacteristicPolynomial(A: parameters): Mtrx -> [ <RngUPolElt, RngIntElt>]`

| | | | |
|---|---|---|---|
| `Al` | `MonStg` | `Default:` | `"Modular"` |
| `Proof:` | `BoolElt` | `Default:` | `true` |

Given a square matrix $A$ over a ring $R$, return the factorisation of the characteristic polynomial of $A$. This function returns the same result as the command `Factorisation(CharacteristicPolynomial(A))`, but may be faster than that for some coefficient rings (in particular, $\mathbb{Z}$ and $\mathbb{Q}$). The parameters are as for the function `CharacteristicPolynomial` above (setting the parameter `Proof` to false suppresses proof of correctness).

`FactoredMinimalAndCharacteristicPolynomials(A: parameters): Mtrx -> [<RngUPolElt, RngIntElt>], [<RngUPolElt, RngIntElt>]`

`FactoredMCPolynomials(A: parameters): Mtrx -> [<RngUPolElt, RngIntElt>], [<RngUPolElt, RngIntElt>]`

| | | | |
|---|---|---|---|
| `Al` | `MonStg` | `Default:` | `"Modular"` |
| `Proof:` | `BoolElt` | `Default:` | `true` |

Given a square matrix $A$ over a ring $R$, return the factorizations of the minimal and characteristic polynomials of $A$, respectively. For some rings, both polynomials can be computed and factored at the same time, so in such cases it will be more efficient to use this function than to call the above functions separately. Setting the parameter `Proof` to false suppresses proof of correctness.

`Eigenvalues(A): Mtrx -> { <FldElt, RngIntElt> }`

Given a square matrix $A$ over a ring $R$, return the eigenvalues of $A$ as a set of pairs, each of which gives the value of a distinct eigenvalue and its multiplicity. Factorization of polynomials over the base ring $R$ must be possible.

`Eigenspace(A, e): AlgMatElt, FldElt -> ModTup`

Given a square matrix $A$ over a ring $R$, and an element $e$ of $R$, return the eigenspace of $A$ corresponding to $e$, which is `Nullspace(A - e)`. If the ring element $e$ is not a eigenvalue for the matrix $A$ then the trivial nullspace is returned.

## 27.12 Canonical Forms

### 27.12.1 Canonical Forms over General Rings

The functions defined here apply to matrices defined over fields or Euclidean domains. See also the section on Reduction in the Lattices chapter for a description of the function `LLL(X): ModMatRngElt → ModMatRngElt, AlgMatElt, RngIntElt` and related basis-reduction functions for matrices.

`EchelonForm(A): Mtrx -> Mtrx, AlgMatElt`

Given an $m \times n$ matrix $A$ over the ring $R$, return the (reduced) row echelon form $E$ of $A$, and also an invertible $m \times m$ transformation matrix $T$ over $R$ such that $T \cdot A = E$. Recall that $T$ is a product

of elementary matrices that transforms $A$ into the echelon form $E$. If $R$ is a Euclidean domain, the function *HermiteForm* (described below) is invoked. Note however, that the user cannot set the parameters for *HermiteForm* when invoking it via `EchelonForm`.

`Adjoint(A): Mtrx -> AlgMatElt`

Given a square $m \times m$ matrix $A$ over the ring $R$, return the adjoint of $A$ as an $m \times m$ matrix. The base ring $R$ must be a ring with exact division whose characteristic is zero or greater than $m$ (this includes most commutative rings).

### 27.12.2 Canonical Forms over Fields

The functions described in this section apply to square matrices defined over fields which support factorization of univariate polynomials. See [Ste97] for a description of the single algorithm which is the basis of most of these functions.

`PrimaryRationalForm(A): Mtrx -> AlgMatElt, AlgMatElt, [ <RngUPolElt, RngIntElt ]`

Given a square matrix $A$ over a field $K$ such that factorization of polynomials is possible over $K$, return the primary rational form of $A$. Each block in the form is the companion matrix of a power of an irreducible polynomial. This function returns three values:

**(a)** The primary rational canonical form $F$ of $A$;

**(b)** An invertible matrix $T$ such that $T \cdot A \cdot T^{-1} = F$;

**(c)** A sequence of pairs corresponding to the blocks of $F$ where each pair consists of the irreducible polynomial and the multiplicity making up the block. This is the value returned by `PrimaryInvariantFactors(A)`.

`JordanForm(A): Mtrx -> Mtrx, AlgMatElt, [ <RngUPolElt, RngIntElt> ]`

Given a square matrix $A$ over a field $K$ such that factorization of polynomials is possible over $K$, return the generalized Jordan form of $A$. Each block in the form is a Jordan block (which itself is derived from a power of an irreducible polynomial), and the generalized Jordan form corresponds to the usual Jordan form if the minimal polynomial splits over $K$. This function returns three values:

**(a)** The Jordan canonical form $F$ of $A$;

**(b)** An invertible matrix $T$ such that $T \cdot A \cdot T^{-1} = F$;

**(c)** A sequence of pairs corresponding to the blocks of $F$ where each pair consists of the irreducible polynomial and the multiplicity making up the block. This is the value returned by `PrimaryInvariantFactors(A)`.

`RationalForm(A): Mtrx -> Mtrx, AlgMatElt, [ RngUPolElt ]`

Given a square matrix $A$ over a field $K$ such that factorization of polynomials is possible over $K$, return the rational form of $A$. For each block other than the final block, the polynomial corresponding to that block divides the polynomial corresponding to the next block. This function returns three values:

**(a)** The rational form $F$ of $A$;

**(b)** An invertible matrix $T$ such that $T \cdot A \cdot T^{-1} = F$;

**(c)** A sequence containing the polynomials corresponding to the successive blocks (where each polynomial, other than the last, divides the next polynomial). This is the value returned by `InvariantFactors(A)`.

`PrimaryInvariantFactors(A): Mtrx -> [ <RngUPolElt, RngIntElt> ]`

Given a square matrix $A$ over a field $K$ such that factorization of polynomials is possible over $K$, return the primary invariant factors of $A$. This is the same as the third return value of `PrimaryRationalForm(A)` or `JordanForm(A)`.

`InvariantFactors(A): Mtrx -> [ RngUPolElt ]`

Given a square matrix $A$ over a field $K$ such that factorization of polynomials is possible over $K$, return the invariant factors of $A$. This is the same as the third return value of `RationalForm(A)`.

`IsSimilar(A, B): AlgMatElt, AlgMatElt -> BoolElt, AlgMatElt`

Given square $m \times m$ matrices $A$ and $B$, both over a field $K$ such that factorization of polynomials is possible over $K$, return `true` iff and only if $A$ is similar to $B$, and if so, return also an invertible $m \times m$ transformation matrix $T$ such that $T \cdot A \cdot T^{-1} = B$.

`HessenbergForm(A): Mtrx -> AlgMatElt`

Given a square $m \times m$ matrix $A$ over the ring $R$, return the Hessenberg form of $A$ as an $m \times m$ matrix. The form has zero entries above the super-diagonal. (This form is used in one of the characteristic polynomial algorithms.) The base ring $R$ must be a field.

`FrobeniusFormAlternating(A): AlgMatElt -> SeqEnum`

Given a non-singular $2n \times 2n$ alternating matrix $A$ over the integers, this function returns the (alternating) Frobenius form $F$ of $A$. That is, a block matrix $F = \begin{pmatrix} 0 & D \\ -D & 0 \end{pmatrix}$, where $D$ is a diagonal matrix with positive diagonal entries, $d_i$, satisfying $d_1 | d_2 | \cdots | d_n$. The second return value is the change of basis matrix $B$, such that $BA^tB = F$.

**Example: Canonical Forms over a Finite Field** `(ex-1f87a1)`

We construct a $5 \times 5$ matrix over the finite field with 5 elements and then calculate various canonical forms. We verify the correctness of the polynomial invariant factors corresponding to the rational form by calculating the Smith form of the characteristic matrix of the original matrix (see below).

```
> K := GF(5);
> A := Matrix(K, 5,
>     [ 0, 2, 4, 2, 0,
>       2, 2, 2, 3, 3,
>       3, 4, 4, 1, 3,
>       0, 0, 0, 0, 1,
>       0, 0, 0, 1, 0 ]);
> A;
[0 2 4 2 0]
[2 2 2 3 3]
[3 4 4 1 3]
[0 0 0 0 1]
[0 0 0 1 0]
> PrimaryInvariantFactors(A);
[
    <x + 1, 1>,
    <x + 1, 1>,
    <x + 4, 1>,
    <x + 4, 1>,
    <x + 4, 1>
]
> JordanForm(A);
[4 0 0 0 0]
[0 4 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
> R, T, F := RationalForm(A);
> R;
[1 0 0 0 0]
[0 0 1 0 0]
[0 1 0 0 0]
```

(continues on next page)

(continued from previous page)

```
[0 0 0 0 1]
[0 0 0 1 0]
> T;
[1 3 0 2 1]
[2 1 2 2 0]
[3 4 3 4 1]
[1 0 0 0 0]
[0 2 4 2 0]
> T*A*T^-1 eq R;
true;
> F;
[
    x + 4,
    x^2 + 4,
    x^2 + 4
]
> P<x> := PolynomialRing(K);
> PM := MatrixAlgebra(P, 5);
> Ax := PM ! x - PM ! A;
> Ax;
[    x     3     1     3     0]
[    3 x + 3     3     2     2]
[    2     1 x + 1     4     2]
[    0     0     0     x     4]
[    0     0     0     4     x]
> SmithForm(Ax);
[      1       0       0       0       0]
[      0       1       0       0       0]
[      0       0   x + 4       0       0]
[      0       0       0 x^2 + 4       0]
[      0       0       0       0 x^2 + 4]
> ElementaryDivisors(Ax);
[
    1,
    1,
    x + 4,
    x^2 + 4,
    x^2 + 4
]
```

### 27.12.3 Canonical Forms over Euclidean Domains

The functions defined here apply to matrices defined over Euclidean domains. See also the section on Reduction in the Lattices chapter for a description of the function `LLL(X): ModMatRngElt → ModMatRngElt, AlgMatElt, RngIntElt` and related functions, which are very useful for integer matrices.

**`HermiteForm(A): Mtrx -> Mtrx, ModMatRngElt`**

```
Al       : MonStg                    Default: "Default"
Optimize: BoolElt                    Default: true
Integral: BoolElt                    Default: true
```

Given an $m \times n$ matrix $A$ over the Euclidean ring $R$, return the Hermite form $H$ of $A$, and also an invertible $m \times m$ transformation matrix $T$ over $R$ such that $T \cdot A = H$. The basic algorithm used is the classical Kannan-Bachem algorithm [CC82, KB79], which has classical complexity (but does not suffer from bad coefficient growth). Since V2.13, for matrices over the integers there is also a

fast modular algorithm by Allan Steel. By default, Magma chooses between these two algorithms, usually favouring the new modular algorithm. But one may set the parameter `Al` to `"Modular"` to force the modular algorithm to be used, and to `"Classical"` to force the classical algorithm to be used. If $R$ is the ring of integers $\mathbb{Z}$ and the matrix $T$ is requested (i.e., if an assignment statement is used with two variables on the left side), then the LLL algorithm will also be used by default to improve $T$ (using the kernel of $A$) so that the size of its entries are very small. If the parameter `Optimize` is set to `false`, then this will not happen (which will be faster but the entries of $T$ will not be as small). If the parameter `Integral` is set to `true`, then the integral (de Weger) LLL method will be used in the LLL step, instead of the default floating point method.

`SmithForm(A): ModMatRngElt -> ModMatRngElt, ModMatRngElt, ModMatRngElt`

Given an $m \times n$ matrix $A$ over the Euclidean ring $R$, return the Smith normal form of $A$. This function returns three values:

**(a)** The Smith normal form $S$ of $A$; and

**(b)** Unimodular matrices $P$ and $Q$ such that $P \cdot A \cdot Q = S$, i.e., $P$ and $Q$ are matrices which transform $A$ into Smith normal form. The algorithm implemented first uses the sparse techniques described in [HHR93] to reduce the matrix to a dense submatrix, then, if this is non-trivial, it either repeatedly calls the Hermite normal form algorithm (see above) and transposes until a diagonal form is obtained, or uses the modular algorithm of F. Lübeck [Lub02]. Unless one wishes one or both of the transformation matrices, it is preferable to use the following function `ElementaryDivisors` since it gives the same information, but saves memory since the matrix $S$ does not need to be constructed.

`ElementaryDivisors(A): Mtrx -> [RngElt]`

Given an $m \times n$ matrix $A$ over the Euclidean ring or field $R$, return the elementary divisors of $A$. These are simply the non-zero diagonal entries of the Smith form of $A$, in order. The divisors are returned as a sequence $[e_1, \ldots, e_r]$ of $r$ elements of $R$ (which may include ones), where $r$ is the rank of $A$ and $e_i | e_{i+1}$ for $i = 1, \ldots, r-1$. The divisors are normalized, so the result is unique. If $R$ is a field, the result is always the sequence of $r$ ones, where $r$ is the rank of $A$. Note that if $m = n = r$, then the determinant of $A$ is the product of the $e_i$ and if $R$ is also a domain, then $e_r$ is the lowest common denominator of the inverse of $A$ over the field of fractions of $R$.

`Saturation(A): Mtrx -> Mtrx`

Given an $m \times n$ matrix $A$ over the integer ring $\mathbb{Z}$, having rank $r$, return an $m \times n$ matrix $S$ over $\mathbb{Z}$ whose first $r$ rows form a basis of the saturation w.r.t. $\mathbb{Q}$ of the $\mathbb{Q}$-vector space spanned by the rows of $A$. The rows of $S$ thus span the same space over $\mathbb{Q}$ as those of $A$, while the $\mathbb{Z}$-module spanned by the rows of $S$ is the set of all $v$ such that for some non-zero scalar $s$, $s \cdot v$ is in the $\mathbb{Z}$-module spanned by the rows of $A$.

**Example: Echelon Forms over a Field and the Integers** (ex-bd35e5)

We illustrate some of these operations for a $4 \times 3$ matrix over $\mathbf{F}_8$.

```
> K<w> := GF(8);
> A := Matrix(K, 4, 3, [1,w,w^5, 0,w^3,w^4, w,1,w^6, w^3,1,w^4]);
> A;
[  1   w w^5]
[  0 w^3 w^4]
[  w   1 w^6]
[w^3   1 w^4]
> EchelonForm(A);
[  1   0   0]
[  0   1   0]
[  0   0   1]
[  0   0   0]
```

We now illustrate some of these operations for a $4 \times 5$ matrix over $\mathbb{Z}$.

```
> A := Matrix(4, 5,
>     [ 2,-4,12,7,0,
>       3,-3,5,-1,4,
>       2,-1,-4,-5,-12,
>       0,3,6,-2,0]);
> A;
[  2  -4  12   7   0]
[  3  -3   5  -1   4]
[  2  -1  -4  -5 -12]
[  0   3   6  -2   0]
> Rank(A);
4
> HermiteForm(A);
[   1    1    1    6 -164]
[   0    3    0   16 -348]
[   0    0    2   13 -200]
[   0    0    0   19 -316]
> SmithForm(A);
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 2 0]
> ElementaryDivisors(A);
[ 1, 1, 1, 2 ]
```

## 27.13 Orders of Invertible Matrices

The functions defined here apply to invertible square matrices. Magma can efficiently compute the order of an invertible matrix over a finite field, using the Cunningham database to factorize the numbers of the form $p^n - 1$ which arise. The algorithm employed is that described in [CLG97b].

Magma also contains efficient algorithms for rigorously proving whether a matrix over the ring of integers $\mathbb{Z}$, the rational field $\mathbb{Q}$, an algebraic number field, a cyclotomic field, or a quadratic field has finite order or not, and for determining the order if it is finite.

**HasFiniteOrder(A): Mtrx -> BoolElt**

Given a square invertible matrix $A$ over a ring $R$, return **true** iff $A$ has finite order, i.e., iff there exists a positive integer $n$ such that $A^n = 1$. The coefficient ring $R$ is currently restricted to being either a finite field, the ring of integers $\mathbb{Z}$, the rational field $\mathbb{Q}$, an algebraic number field, a cyclotomic field, or a quadratic field. For matrices over any of these rings, the function rigorously proves its result (over other rings, an error results).

**Order(A): AlgMatElt -> RngIntElt**

```
Proof: BoolElt                    Default: true
```

Given a square invertible matrix $A$ over any commutative ring, return the order of $A$. If $R$ is a ring for which a finite order proof exists (see **HasFiniteOrder** above), then an error results if $A$ has infinite order. Over other rings, if $A$ has infinite order then the function may loop indefinitely since it may not be able to prove the infinitude of the order.

**FactoredOrder(A): AlgMatElt -> [ <RngIntElt, RngIntElt> ]**

```
Proof: BoolElt                    Default: true
```

Given a square invertible matrix $A$ over a finite field, return the order of $A$ in factored form. This returns the same value as **Factorization(Order(A))**, but since the order computation must

compute the factorization of the order anyway, it involves no more effort to have it return the factorization. The conditions on the ring are as for `Order`.

`ProjectiveOrder(A): AlgMatElt -> RngIntElt, RngElt`

| `Proof: BoolElt` | `Default: true` |
|---|---|

Given a square invertible matrix $A$ over a finite field $K$, return the projective order $n$ of $A$ and a scalar $s \in K$ such that $A^n = sI$. The projective order of $A$ is the smallest $n$ such that $A^n$ is a scalar matrix (not just the identity matrix), and it always divides the true order of $A$. The parameter `Proof` is as for `Order`.

`FactoredProjectiveOrder(A): AlgMatElt -> [ <RngIntElt, RngIntElt> ], RngElt`

| `Proof: BoolElt` | `Default: true` |
|---|---|

Given a square invertible matrix $A$ over a finite field $K$, return the projective order $n$ of $A$ in factored form and a scalar $s \in K$ such that $A^n = sI$. The parameter `Proof` is as for `FactoredOrder`.

## 27.14 Numerical Linear Algebra

Magma has some limited capabilities with numerical linear algebra, that is, linear algebra over a real or complex field. The basis of these is the `RQDecomposition` (appropriate as Magma is row-based), and its companion `QLDecomposition`. Internally, Magma uses these functions to compute (e.g.) matrix inverses and determinants over $\mathbf{R}$ and $\mathbf{C}$.

All of the intrinsics starting with `Numeric` can also be accessed with this word omitted (by extended types on a real/complex matrix), though for some purposes it is useful to remind oneself that numerical linear algebra is not always the same.

`RQDecomposition(M): Mtrx[RngReCom] -> Mtrx, AlgMatElt`

Given a matrix $M$ over a real or complex field, write $M = RQ$ where $R$ is of triangular form and $Q$ orthogonal (or unitary in the complex case). In particular, $R$ has the same dimensions as $M$; when $M$ has more columns $c$ than rows $r$, then $R$ is of the form $[Z|U]$ where $Z$ is a zero matrix that has $(c-r)$ columns and $U$ is upper triangular. When $r > c$ then the bottom $c$ rows of $R$ form an upper triangular matrix. No normalisations are made except that $Q$ has determinant 1.

`QLDecomposition(M): Mtrx[RngReCom] -> AlgMatElt, Mtrx`

Given a matrix $M$ over a real or complex field, write $M = QL$ where $L$ is of triangular form and $Q$ orthogonal (or unitary in the complex case). This function works simply by applying `RQDecomposition` to the transpose.

`NumericalInverse(M): Mtrx[RngReCom] -> AlgMatElt`

| `Epsilon: FldReElt` | `Default:` |
|---|---|

Given a square positive-dimension matrix over a real or complex field, compute its inverse. The `Epsilon` parameter allows one to specify what size of values should be considered to be zero (resulting in an inversion failure). By default, the value of `Epsilon` is computed in a way that is relative to the size of the entries and their precision.

**Example: Numerical Matrix Decompositions** (ex-5a8543)

We give some examples of these decompositions.

```
> RR := RealField(10);
> r := 4;
> c := 5;
> A := [RR!Random([-2^10..2^10])/2^9 : i in [1..r*c]];
> M := Matrix(r,c,A);
```

(continues on next page)

(continued from previous page)

```
> M;
[-0.1796875000 -1.394531250 -1.345703125 -0.4414062500 0.0000000000]
[1.396484375 -1.496093750 -0.2617187500 -1.882812500 -1.679687500]
[-1.169921875 -0.9062500000 1.707031250 -0.6582031250 1.480468750]
[0.8867187500 -0.3359375000 0.5839843750 0.9707031250 1.896484375]
> R,Q := RQDecomposition(M); R, Q;
[0.0000000000 1.767478576 0.8447303787 -0.05853296288 0.3765439489]
[0.0000000000 0.0000000000 2.924272555 -0.2048205303 1.424771353]
[0.0000000000 0.0000000000 0.0000000000 2.589389831 -1.011949267]
[0.0000000000 0.0000000000 0.0000000000 0.0000000000 -2.403971597]

[-0.1278237197 -0.4893124575 0.3070157953 0.6619247514 -0.4602513884]
[-0.3369945746 -0.5416048144 -0.7236111740 0.05012569151 0.2587917417]
[0.6155218516 -0.6003864917 0.06838441636 -0.4759773097 -0.1715752984]
[-0.5959647961 -0.2953736282 0.5643042371 -0.4119965068 0.2634387749]
[-0.3688557516 0.1397427076 -0.2429248231 -0.4037914284 -0.7888963320]
> Max([Abs(x) : x in Eltseq(R*Q-M)]);
3.492459655E-10 10
> Max([Abs(x) : x in Eltseq(Transpose(Q)*Q-1)]);
1.527951099E-10 14
> //
> CC<i> := ComplexField(5);
> r := 5;
> c := 3;
> A := [CC!Random([-2^10..2^10])/2^9 : j in [1..r*c]];
> B := [CC!Random([-2^10..2^10])/2^9 : j in [1..r*c]];
> M := Matrix(r,c,A) + i*Matrix(r,c,B);
> R,Q := RQDecomposition(M); R, Q;
[-1.1471 + 1.0971*i      -2.6950 - 0.38148*i   0.47294 - 0.95428*i]
[-0.25971 - 0.0084722*i  0.61157 + 0.49147*i   1.5717 - 0.61096*i]
[0.30902 + 0.59626*i     0.79478 + 1.0406*i    -1.1157 - 1.6800*i]
[0.00000                 1.2132 + 1.4143*i    -0.95447 + 0.14362*i]
[0.00000                 0.00000               3.0078 - 0.24966*i]

[0.42713 - 0.0038340*i   0.51414 - 0.56570*i     -0.39550 + 0.27712*i]
[0.54343 + 0.60175*i    -0.087580 + 0.0038354*i   0.56113 + 0.14178*i]
[0.33511 + 0.21874*i    -0.62188 + 0.14580*i     -0.65721]
> Max([Abs(x) : x in Eltseq(R*Q-M)]);
0.00020472 3
> Max([Abs(x) : x in Eltseq(ConjugateTranspose(Q)*Q-1)]);
0.00012207 9
> //
> r := 5;
> c := 5;
> A := [RR!Random([-2^10..2^10])/2^9 : j in [1..r*c]];
> M := Matrix(r,c,A);
> MI := NumericalInverse(M); MI;
[0.5488038571 -0.3734925483 0.005386589048 0.4034292318 -0.6410457874]
[0.3082898339 -0.5126616665 0.1612965318 -0.2971166681 0.07521900340]
[-0.4446389269 0.4962564367 0.1565165202 -0.4722494193 0.2177560581]
[-0.05815013416 -0.3372341073 0.03559373548 0.5402886617 -0.7996494168]
[-0.9780538200 1.225635935 -0.4735623930 -1.060353816 1.356842221]
> Max([Abs(x) : x in Eltseq(M*MI-1)]);
4.656612873E-10 4
```

### 27.14.1 Rank, Kernel, Solution, and Pseudoinverse

To compute the kernel (thus also rank) of a matrix $M$ or a solution to a system $vM = w$, the implementation first performs a QL decomposition internally. When $L$ is of a strict triangular form, namely $L = \left(\frac{Z}{A|T}\right)$ where $Z$ is zero and $T$ is not only lower triangular but also invertible, then the information can be read off immediately. In particular, the rank $r$ is the dimension of $T$, the kernel is the first $k$ rows of $Q$ where $k$ is number of rows in $Z$, while $vM = w$ requires $vQL = w$ so $v = w_r T^{-1} Q_r^T$ where $w_r$ and $Q_r$ are the rightmost $r$ columns of $w$ and $Q$; one checks $vQ_rA$ is (numerically) equal to the other columns of $w$.

When $L$ is not of this form, then an arbitrary orthogonal/unitary transformation is performed on the input (on the right), and the algorithm is re-tried. This case occurs when the $r$ rightmost columns of $M$ are not linearly independent (Magma is internally working on the transpose of $M$ in the $RQ$ decomposition, so that the rightmost columns become the lowermost rows). Similarly, if a condition number for $T$ is too large, such a randomising transform can be profitably applied.

Note that the kernel is returned as a matrix, and indeed has orthonormal rows. The same is true for the image.

The computation of the `NumericalPseudoinverse` is similar, being based on the `QLDecomposition` and possibly also transforming the columns on the right, to ensure that $L$ is of the nice triangular form (this is internally called `QLCDecomposition`). The pseudoinverse of the row-independent (bottom) part $B$ of $L$ is then computed as the solution $\tilde{P}$ of $\tilde{P}BB^\star = B^\star$, and one row-extends $\tilde{P}$ to the left (by zero columns) to get the full (numerical) pseudoinverse $P$.

`NumericalRank(M): Mtrx[RngReCom] -> RngIntElt`

| `Epsilon: FldReElt` | `Default:` |
|---|---|

Given a matrix over a real or complex field, compute its (numerical) rank. The `Epsilon` parameter allows one to specify what size of values should be considered to be zero. By default, the value of `Epsilon` is computed in a way that is relative to the size of the entries and their precision.

`NumericalKernel(M): Mtrx[RngReCom] -> Mtrx`

| `Epsilon: FldReElt` | `Default:` |
|---|---|

Given a matrix over a real or complex field, compute its (numerical) kernel. The `Epsilon` parameter allows one to specify what size of values should be considered to be zero. By default, the value of `Epsilon` is computed in a way that is relative to the size of the entries and their precision.

Exceptionally, `Kernel` returns a space, while `NumericalKernel` returns a basis matrix of it.

`NumericalImage(M): Mtrx[RngReCom] -> Mtrx`

| `Epsilon: FldReElt` | `Default:` |
|---|---|

Given a matrix over a real or complex field, compute its (numerical) image, that is, its row space. The `Epsilon` parameter allows one to specify what size of values should be considered to be zero. By default, the value of `Epsilon` is computed in a way that is relative to the size of the entries and their precision.

Exceptionally, `Image` returns a space, while `NumericalImage` returns a basis matrix of it.

`NumericalSolution(M, w): Mtrx[RngReCom], Mtrx[RngReCom] -> Mtrx, Mtrx`

| `Epsilon: FldReElt` | `Default:` |
|---|---|

Given a matrix $M$ and a compatible matrix (or vector) $w$ over a real or complex field, solve numerically for $vM = w$. This gives an error if there is no solution. The second return value is the kernel of $M$. The `Epsilon` parameter allows one to specify what size of values should be considered to be zero. By default, the value of `Epsilon` is computed in a way that is relative to the size of the entries and their precision.

Exceptionally, the second return value is a matrix for `NumericalSolution` and a space for `Solution`.

`NumericalIsConsistent(M, w): Mtrx[RngReCom], Mtrx[RngReCom] -> BoolElt, Mtrx, Mtrx`

| Epsilon: FldReElt | Default: |
|---|---|

Given a matrix $M$ and a compatible matrix (or vector) $w$ over a real or complex field, determine whether $vM = w$ is numerically solvable. The second return value is a solution (when one exists)(, and the third return value is the (numerical) kernel of $M$. The `Epsilon` parameter allows one to specify what size of values should be considered to be zero. By default, the value of `Epsilon` is computed in a way that is relative to the size of the entries and their precision.

Exceptionally, the third return value is a matrix for `NumericalIsConsistent` and a space for `IsConsistent`.

`NumericalPseudoinverse(M): Mtrx[RngReCom] -> Mtrx`

| Epsilon: FldReElt | Default: |
|---|---|

Given a matrix $M$ over a real or complex field, compute its pseudoinverse $P$ such that $PMP = P$, $MPM = M$, $(MP)^\star = MP$, and $(PM)^\star = PM$. The `Epsilon` parameter allows one to specify what size of values should be considered to be zero. By default, the value of `Epsilon` is computed in a way that is relative to the size of the entries and their precision. Note that the Magma intrinsic `PseudoInverse` is quite different (largely applicable for matrices over Euclidean domains).

**Example: Numerical Rank, Kernel, and Pseudoinverse** (ex-b3a16a)

We give some examples for ranks, kernels, solutions, and pseudoinverses.

```
> RR := RealField(10);
> r := 3;
> c := 5;
> S := [-2^10..2^10];
> A := [Vector([RR!Random(S)/2^9 : i in [1..c]]) : j in [1..r]];
> M := Matrix(A cat [A[1]+A[2]-A[3]]);
> M;
[-1.347656250 -1.980468750 0.2070312500 0.2734375000 -1.273437500]
[0.6464843750 1.726562500 -0.3769531250 0.2011718750 -1.646484375]
[-0.1660156250 1.236328125 -1.500000000 1.119140625 -1.195312500]
[-0.5351562500 -1.490234375 1.330078125 -0.6445312500 -1.724609375]
> NumericalRank(M);
3
> NumericalKernel(M); // scaled to norm 1
[-0.5000000001 -0.4999999998 0.4999999999 0.5000000002]
> v := Vector([RR!Random(S)/2^9 : i in [1..c]]);
> NumericalIsConsistent(M,v);
false
> w := &+[M[i] * Random(S)/2^9 : i in [1..r]];
> NumericalIsConsistent(M,w);
true (1.174316406 0.4165039062 0.7807617187 0.8100585938)
> PI := NumericalPseudoinverse(M); PI;
[-0.1991067080 0.1335911770 -0.1240987264 0.05858319545]
[-0.1885126384 0.1881125824 0.02268111094 -0.02308116681]
[-0.1331315985 0.1012317424 -0.2422989451 0.2103990890]
[0.1423299946 -0.08451596130 0.1982404871 -0.1404264538]
[-0.07615077666 -0.2393032832 -0.09708672579 -0.2183673341]
> Max([Abs(x) :  x in Eltseq(PI*M*PI-PI)]);
7.275957614E-11 9
> //
```

(continues on next page)

(continued from previous page)

```
> CC<i> := ComplexField(30);
> r := 5;
> c := 4;
> A := [CC!Random([-2^30..2^30])/2^28 : j in [1..r*c]];
> B := [CC!Random([-2^30..2^30])/2^28 : j in [1..r*c]];
> M := Matrix(r,c,A) + i*Matrix(r,c,B);
> NumericalRank(M);
4
> NumericalKernel(M);
[0.20896555976959584519167865 4168 + 0.033437330830817586109199 4295635*i
 -0.022704563806390026727980 2631268 - 0.354081499548224953341852663885*i
 -0.220008165304096296179634 856382 - 0.815082824683314746916890536889*i
 0.014650533172374537934129 9824850 - 0.155641405430086914216739561265*i
 0.209604845631814749141329 584762 - 0.219520964338492435702429703559*i]
> PI := NumericalPseudoinverse(M);
> Max([Abs(x) :  x in Eltseq(PI*M*PI-PI)]);
1.60521564392120817260990102530E-30 1
> Max([Abs(x) :  x in Eltseq(M*PI*M-M)]);
2.54399997817916409183980874314E-29 18
> Max([Abs(x) :  x in Eltseq(M*PI-ConjugateTranspose(M*PI))]);
7.88860905221011805411728565283E-30 25
> Max([Abs(x) :  x in Eltseq(PI*M-ConjugateTranspose(PI*M))]);
5.03961120551682881427899748034E-30 8
```

### 27.14.2 Eigenvalues and the Singular Value Decomposition

Computing the eigenvalues of a real or complex square matrix is done by first computing the Hessenberg form (almost upper-triangular, but with possible nonzero entries on the immediate subdiagonal), and then applying iterative methods to achieve the Schur form (upper-triangular except for 2-by-2 blocks with complex eigenvalues in the real case). Special methods are used in the case where the matrix is symmetric. The computation of the singular value decomposition works by first computing the bidiagonal form, and then similarly iterating.

**NumericalHessenbergForm**(M): Mtrx[RngReCom] -> Mtrx, Mtrx

Given a square matrix $M$ over a real/complex field, compute a Hessenberg matrix $H$ that is upper-triangular except for possible nonzero entries on the immediate subdiagonal, and an orthogonal/unitary transformation matrix $Q$ such that $H = QMQ^\star$ and $QQ^\star = I$ (both equalities being numerically approximate). A procedure similar to the RQ decomposition is used, working on both rows and columns.

**NumericalSchurForm**(M): Mtrx[RngReCom] -> Mtrx, Mtrx

```
Transform: BoolElt                    Default: true
```

Given a square matrix $M$ over a real/complex field, compute the Schur form $S$ that is upper-triangular except for possible nonzero entries on the immediate subdiagonal corresponding to 2-by-2 real blocks with complex eigenvalues, and an orthogonal/unitary transformation matrix $Q$ such that $S = QMQ^\star$ and $QQ^\star = I$ (both equalities being numerically approximate). First the Hessenberg form is calculated, and then the unwanted subdiagonal terms are removed by a iterative process.

**NumericalEigenvalues**(M): Mtrx[RngReCom] -> SeqEnum

```
Balance: BoolElt                      Default: true
```

Given a square matrix $M$ over a real/complex field, compute numerical approximations to its eigenvalues. This operates by calculating the Schur form, and then returning the relevant eigenvalues. A form of Parlett-Reinsch balancing is used by default, to increase the precision of the answer.

Exceptionally, `NumericalEigenvalues` returns a sequence of reals or complexes, while `Eigenvalues` on real/complex matrix returns a tuple of eigenvalues and multiplicities, for reasons of backwards compatibility (the multiplicities really cannot be guaranteed in the numerical case). Note that `NumericalEigenvalues` always returns a full sequence of eigenvalues for a real input, while `Eigenvalues` will return only those which are (approximately) real.

`NumericalEigenvectors(M, e): Mtrx, FldComElt -> SeqEnum`

Given a square matrix $M$ that is coercible into the complexes and an approximation $e$ to an eigenvalue of it, attempt to find eigenvectors. This function uses `NumericalKernel`, though returns the result as a sequence of (row) vectors.

`NumericalBidiagonalForm(M): Mtrx[RngReCom] -> Mtrx, Mtrx, Mtrx`

Given a matrix $M$ over a real/complex field, compute a bidiagonal form $B$ and orthogonal/unitary transformations $U, V$ such that $B = UMV^\star$. Here $B$ is upper bidiagonal if the number of columns is at least as large as the number of rows, and else is lower bidiagonal.

`NumericalSingularValueDecomposition(M): Mtrx[RngReCom] -> Mtrx, Mtrx, Mtrx`

Given a matrix $M$ over a real/complex field, compute the singular value decomposition $S$ and orthogonal/unitary transformations $U, V$ such that $S = UMV^\star$. Here $S$ is diagonal in its upper-left corner, with the diagonal entries being nonnegative and sorted in increasing size.

**Example: Numerical Hessenberg Form and Eigenvalues** (ex-604853)

We give some examples for eigenvalues.

```
> RR := RealField(10);
> r := 5;
> c := 5;
> A := [RR!Random([-2^10..2^10])/2^9 : i in [1..r*c]];
> M := Matrix(r,c,A);
> H := NumericalHessenbergForm(M); H;
[-1.387758248 0.2244522681 0.1050196798 -0.6292371533 1.500649611]
[-1.737800256 -0.6560018402 -0.1292093616 -0.4045870847 -0.1501588456]
[-0.0000000000 3.503611888 -0.4200132582 -1.521474700 -0.4771033714]
[-0.0000000000 0.0000000000 1.241929396 -1.887789155 -1.398580785]
[-0.0000000000 0.0000000000 0.0000000000 2.063240612 -1.882812500]
> S := NumericalSchurForm(M); S; // need not be diag in real case
[-0.5276818045 2.970316193 -0.8183434354 -1.227178569 1.207265137]
[-0.6560286404 0.1374330601 0.7474073924 -0.7405697671 0.4496312767]
[0.0000000000 0.0000000000 -2.733003988 0.5326130259 -0.2389927500]
[0.0000000000 0.0000000000 0.0000000000 -1.878704188 2.041990652]
[0.0000000000 0.0000000000 0.0000000000 -2.574033133 -1.232418078]
> NumericalEigenvalues(M);
[ -2.733003988,
  -1.555561133 - 2.269742311*$.1, -1.555561133 + 2.269742311*$.1,
  -0.1951243722 - 1.355735243*$.1, -0.1951243722 + 1.355735243*$.1 ]
> M := M + Transpose(M); // make symmetric
> NumericalEigenvalues(M); // all real
[ -5.927297871, -4.230690093, -3.248785626, -1.661515789, 2.599539376 ]
> //
> // example with companion matrix of a polynomial
> //
> f:=Polynomial([Random(10^(12-j)) : j in [0..8]] cat [1]); f;
x^9 + 8569*x^8 + 98753*x^7 + 533140*x^6 + 3583980*x^5 + 52066941*x^4 +
    183889636*x^3 + 5743452355*x^2 + 22923098485*x + 725907276071
> NumericalEigenvalues(ChangeRing(CompanionMatrix(f),RR));
[ -8557.467295,
  -10.76902934 - 4.819656295*$.1, -10.76902934 + 4.819656295*$.1,
  -4.602845057 - 8.104590519*$.1, -4.602845057 + 8.104590519*$.1,
```

(continues on next page)

(continued from previous page)

```
   2.007741523 - 8.923992632*$.1, 2.007741523 + 8.923992632*$.1,
   7.597780247 - 5.110282521*$.1, 7.597780247 + 5.110282521*$.1 ]
> Sort([r[1] : r in Roots(f,ComplexField(10))],func<x,y|Real(x)-Real(y)>);
[ -8557.467295,
  -10.76902933 - 4.819656325*$.1, -10.76902933 + 4.819656325*$.1,
  -4.602845059 - 8.104590521*$.1, -4.602845059 + 8.104590521*$.1,
   2.007741527 - 8.923992640*$.1, 2.007741527 + 8.923992640*$.1,
   7.597780251 - 5.110282532*$.1, 7.597780251 + 5.110282532*$.1 ]
```

**Example: Numerical Singular Value Decomposition** (ex-4e7c97)

We give some examples for bidiagonal form and singular value decomposition.

```
> CC<i> := ComplexField(5);
> r := 4;
> c := 5;
> A := [CC!Random([-2^10..2^10])/2^9 : i in [1..r*c]];
> B := [CC!Random([-2^10..2^10])/2^9 : i in [1..r*c]];
> M := Matrix(r,c,A) + i*Matrix(r,c,B);
> X := NumericalBidiagonalForm(M); X;
> NumericalBidiagonalForm(M); // matrix over CC
[2.1983 3.7789 0.00000 0.00000 0.00000]
[0.00000 2.0496 1.8677 0.00000 0.00000]
[0.00000 0.00000 2.6879 2.4315 0.00000]
[0.00000 0.00000 0.00000 3.5035 0.00000]
> NumericalSingularValueDecomposition(M);
[4.9265 0.00000 0.00000 0.00000 0.00000]
[0.00000 4.5508 0.00000 0.00000 0.00000]
[0.00000 0.00000 2.5876 0.00000 0.00000]
[0.00000 0.00000 0.00000 0.73134 0.00000]
> [Sqrt(Real(x)) : x in NumericalEigenvalues(M*ConjugateTranspose(M))];
[ 0.73122, 2.5876, 4.5506, 4.9265 ]
```

## 27.15 Miscellaneous Operations on Matrices

`FrobeniusImage(A, e): Mtrx, RngIntElt -> Mtrx`

Given a matrix $A$ over a finite field $K$ of characteristic $p$, return the matrix obtained from $A$ by mapping each entry $A_{i,j}$ to $(A_{i,j})^{p^e}$.

## 27.16 Bibliography


- [ABM99] John Abbott, Manuel Bronstein, and Thom Mulders. Fast Deterministic Computation of Determinants of Dense Matrices. In Sam Dooley, editor, *Proceedings ISSAC'99*, 197–204. New York, 1999. ACM Press.
- [CLG97b] Frank Celler and Charles R. Leedham-Green. Calculating the Order of an Invertible Matrix. In Larry Finkelstein and William M. Kantor, editors, *Groups and Computation II*, volume 28 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 55–60. AMS, 1997.
- [CC82] T.W.J. Chou and G.E. Collins. Algorithms for the solution of systems of linear Diophantine equations. *SIAM J. Computing*, 11(4):687–708, 1982.
- [HHR93] George Havas, Derek F. Holt, and Sarah Rees. Recognizing badly presented Z-modules. *Linear Algebra and its Applications*, 192:137–164, 1993.
- [KB79] R. Kannan and A. Bachem. Polynomial algorithms for computing the Smith and Hermite normal forms of an integer matrix. *SIAM J. Computing*, 9:499–507, 1979.

- [Lub02] F. Lübeck. On the computation of elementary divisors of integer matrices. *J. Symbolic Comp.*, 33:57–65, 2002.
- [Ste97] Allan Steel. A New Algorithm for the Computation of Canonical Forms of Matrices over Fields. *J. Symbolic Comp.*, 24(3):409–432, 1997.

# Chapter 28

# Sparse Matrices

## 28.1 Introduction

A separate type is provided for sparse matrices to allow the user to construct such matrices and apply algorithms which take advantage of sparsity. Sparse matrices are distinct from normal matrices in Magma, which have a dense representation. This chapter describes the operations available for creating and working with sparse matrices.

The operations provided for sparse matrices include dynamic construction, simple properties, and the calculation of a number of non-trivial and important invariants (such as rank, determinant, or computing a non-zero vector in the nullspace). In particular, this datatype supports the class of *index-calculus* algorithms which involve generating a very large sparse system and then solving the system or finding the elementary divisors of the corresponding matrix (for example, to compute the abelian group structure). An extended example presented at the end of the chapter (*Discrete Logarithms by Linear Sieve*) illustrates how an index-calculus method may be implemented in practice in the Magma language using the sparse matrix facilities.

The type name for the category of sparse matrices is `MtrxSprs`. All sparse matrices over a given ring $R$ lie in the same *sparse matrix structure*, whose type name is `MtrxSprsStr`. The user will, in practice, rarely need to refer explicitly to the parent structure.

## 28.2 Creation of Sparse Matrices

This section describes the constructs provided for creating sparse matrices.

### 28.2.1 Construction of Initialized Sparse Matrices

`SparseMatrix(R, m, n, Q): Rng, RngIntElt, RngIntElt, [ <RngIntElt, RngIntElt, RngElt> ] -> MtrxSprs`

`SparseMatrix(m, n, Q): RngIntElt, RngIntElt, [ <RngIntElt, RngIntElt, RngElt> ] -> MtrxSprs`

Given a ring $R$ (optional), integers $m, n \geq 0$, and a sequence $Q$, return the $m \times n$ sparse matrix over $R$ whose non-zero entries are those specified by $Q$, coerced into $R$. If $R$ is not given, it is derived from the entries in $Q$. Either of $m$ and $n$ may be 0, in which case $Q$ must have length 0 (and may be null if $R$ is given), and the $m \times n$ zero sparse matrix over $R$ is returned. There are two possibilities for $Q$:

**(a)** The sequence $Q$ is a sequence of tuples, each of the form `<i, j, x>`, where $1 \leq i \leq m$, $1 \leq j \leq n$, and $x \in S$ for some ring $S$. Such a tuple specifies that the $(i, j)$-th entry of the matrix is $x$. If $R$ is given, then $x$ is coerced into $R$; otherwise the matrix is created over $S$. If an entry position is not given then its value is zero, while if an entry position is repeated then the last value overrides any previous values.

**(b)** The sequence $Q$ is a "flat" sequence of integers, giving the entries of the matrix in compact form. To be precise, $Q$ begins with the number of non-zero entries $n$ for the first row, then $2 \cdot n$ integers giving column-entry pairs for the first row, and this format is immediately followed for the second row and so on. A zero row is specified by a zero value for $n$. If $R$ is given, the integer entries are coerced into $R$; otherwise the matrix is defined over $\mathbb{Z}$. (Thus this method will not allow elements of $R$ which cannot be created by coercing integers into $R$ alone; another way of saying this is that the entries must all lie in the prime ring of $R$). This allows a very compact way to create (and store) sparse matrices. The examples below illustrate this format.

`SparseMatrix(R, m, n): Rng, RngIntElt, RngIntElt -> MtrxSprs`

Given a ring $R$, and integers $m, n \geq 0$, create the $m \times n$ sparse matrix over $R$.

`SparseMatrix(m, n): RngIntElt, RngIntElt -> MtrxSprs`

Given integers $m, n \geq 0$, create the $m \times n$ sparse matrix over the integer ring $\mathbb{Z}$.

### 28.2.2 Construction of Trivial Sparse Matrices

`SparseMatrix(R): Rng -> MtrxSprs`

`SparseMatrix() -> MtrxSprs`

Create the $0 \times 0$ sparse matrix over $R$. If $R$ is omitted (so there are no arguments), then $R$ is taken to be the integer ring $\mathbb{Z}$. These functions will usually be called when the user wishes to create a sparse matrix whose final dimensions are initially unknown, and for which the *SetEntry* procedure below will be used to extend the matrix automatically, as needed.

**Example: Creating Sparse Matrices** (ex-a5eec8)

This example demonstrates simple ways of creating matrices using the general `SparseMatrix(R, m, n, Q)` function. Sparse matrices may be displayed in the sparse representation using the `Magma` print-format. Also, the function *Matrix* (described below) takes a sparse matrix and returns a normal (dense-representation) matrix, so this function provides a means of printing a small sparse matrix as a normal matrix.

(a) A sparse $2 \times 2$ matrix is defined over $\mathbb{Z}$, using the first (sequence of tuples) method. We specify that there is a 3 in the $(1, 2)$ position and a $-1$ in the $(2, 3)$ position. The ring need not be given since the entries are in $\mathbb{Z}$ already.

```
> A := SparseMatrix(2, 3, [<1,2,3>, <2,3,-1>]);
> A;
Sparse matrix with 2 rows and 3 columns over Integer Ring
> Matrix(A);
[ 0  3  0]
[ 0  0 -1]
```

(b) The same matrix is now defined over $\mathbf{F}_{23}$. We could also coerce the 3rd component of each tuple into $\mathbf{F}_{23}$ and thus omit the first argument.

```
> A := SparseMatrix(GF(23), 2, 3, [<1,2,3>, <2,3,-1>]);
> A;
Sparse matrix with 2 rows and 3 columns over GF(23)
> Matrix(A);
[ 0  3  0]
[ 0  0 22]
```

(c) A similar sparse matrix is defined over $\mathbf{F}_{2^4}$. When $A$ is printed in `Magma` format, the sequence-of-tuples form is used (because the entries cannot be printed as integers).

```
> K<w> := GF(2^4);
> A := SparseMatrix(K, 2, 3, [<1,2,3>, <2,3,w>]);
> A;
```

(continues on next page)

(continued from previous page)

```
Sparse matrix with 2 rows and 3 columns over GF(2^4)
> Matrix(A);
[   0    1    0]
[   0    0    w]
> A: Magma;
SparseMatrix(GF(2, 4), 2, 3, [
    <1, 2, 1>,
    <2, 3, w>
])
```

(d) A sparse $4 \times 5$ matrix $A$ is defined over $\mathbb{Z}$, using the second (flat integer sequence) method. Here row 1 has one non-zero entry: -1 at column 3; then row 2 has three non-zero entries: 9 at column 2, 7 at column 3, and $-3$ at column 4; row 3 has no non-zero entries (so we give a 0 at this point); and finally row 4 has one non-zero entry 3 at column 4. Note that when $A$ is printed in `Magma` format, this time the compact "flat" sequence of integers form is used, since this is possible.

```
> A := SparseMatrix(4,5, [1,3,-1, 3,2,9,3,7,4,-3, 0, 1,4,3]);
> A;
Sparse matrix with 4 rows and 5 columns over Integer Ring
> Matrix(A);
[ 0  0 -1  0  0]
[ 0  9  7 -3  0]
[ 0  0  0  0  0]
[ 0  0  0  3  0]
> A: Magma;
SparseMatrix(4, 5, \\[
    1, 3,-1,
    3, 2,9, 3,7, 4,-3,
    0,
    1, 4,3
])
```

### 28.2.3 Construction of Structured Matrices

**`IdentitySparseMatrix(R, n): Rng, RngElt -> MtrxSprs`**

Given a ring $R$, and an integer $n \geq 0$, return the $n \times n$ identity sparse matrix over $R$.

**`ScalarSparseMatrix(n, s): RngIntElt, RngElt -> MtrxSprs`**

Given an integer $n \geq 0$ and an element $s$ of a ring $R$, return the $n \times n$ scalar sparse matrix over $R$ which has $s$ on the diagonal and zeros elsewhere. The argument $n$ may be 0, in which case the $0 \times 0$ sparse matrix over $R$ is returned.

**`ScalarSparseMatrix(R, n, s): Rng, RngIntElt, RngElt -> MtrxSprs`**

Given a ring $R$, an integer $n \geq 0$, and an element $s$ of a ring $S$, return the $n \times n$ scalar sparse matrix over $R$ which has $s$, coerced into $R$, on the diagonal and zeros elsewhere. $n$ may be 0, in which case in which case the $0 \times 0$ sparse matrix over $R$ is returned.

**`DiagonalSparseMatrix(R, n, Q): Rng, RngIntElt, [ RngElt ] -> MtrxSprs`**

Given a ring $R$, an integer $n \geq 0$, and a sequence $Q$ of $n$ ring elements, return the $n \times n$ diagonal sparse matrix over $R$ whose diagonal entries correspond to the entries of $Q$, coerced into $R$.

**`DiagonalSparseMatrix(R, Q): Rng, [ RngElt ] -> MtrxSprs`**

Given a ring $R$ and a sequence $Q$ of $n$ ring elements, return the $n \times n$ diagonal sparse matrix over $R$ whose diagonal entries correspond to the entries of $Q$, coerced into $R$.

**`DiagonalSparseMatrix(Q): [ RngElt ] -> MtrxSprs`**

Given a sequence $Q$ of $n$ elements from a ring $R$, return the $n \times n$ diagonal sparse matrix over $R$ whose diagonal entries

`SparseMonomialMatrix(g): GrpPermElt -> MtrxSprs`

Given a degree-$(2d)$ permutation $g$, return the $d \times d$ monomial matrix corresponding to $g$.

### 28.2.4 Parents of Sparse Matrices

`SparseMatrixStructure(R): [ Rng ] -> MtrxSprsStr`

Create the structure containing all sparse matrices (of any shape) over ring $R$. This structure does not need to be created explicitly by the user (it will be the parent of any sparse matrix over $R$), but it may be useful to create it in this way occasionally.

## 28.3 Accessing Sparse Matrices

The following functions access basic properties of sparse matrices.

### 28.3.1 Elementary Properties

`BaseRing(A): MtrxSprs -> Rng`

`CoefficientRing(A): MtrxSprs -> Rng`

Given a sparse matrix $A$ with entries lying in a ring $R$, return $R$.

`NumberOfRows(A): MtrxSprs -> RngIntElt`

`Nrows(A): MtrxSprs -> RngIntElt`

Given an $m \times n$ sparse matrix $A$, return $m$, the number of rows of $A$.

`NumberOfColumns(A): MtrxSprs -> RngIntElt`

`Ncols(A): MtrxSprs -> RngIntElt`

Given an $m \times n$ sparse matrix $A$, return $n$, the number of columns of $A$.

`ElementToSequence(A): MtrxSprs -> [ <RngIntElt, RngIntElt, RngElt> ]`

`Eltseq(A): MtrxSprs -> [ <RngIntElt, RngIntElt, RngElt> ]`

Given a sparse matrix $A$ over the ring $R$ having $m$ rows and $n$ columns, return the entries of $A$ as a sequence of all tuples of the form `<i, j, x>` such that the $[i, j]$-th entry of $A$ equals $x$ and $x$ is non-zero. It is always true that `SparseMatrix(Nrows(A), Ncols(A), Eltseq(A))` equals $A$.

`NumberOfNonZeroEntries(A): MtrxSprs -> RngIntElt`

`NNZEntries(A): MtrxSprs -> RngIntElt`

Given a sparse matrix $A$, return the number of non-zero entries in $A$.

`Density(A): MtrxSprs -> FldRe`

Given a sparse matrix $A$, return the density of $A$ as a real number, which is the number of non-zero entries in $A$ divided by the product of the number of rows of $A$ and the number of columns of $A$ (or zero if $A$ has zero rows or columns).

`Support(A, i): MtrxSprs, RngIntElt -> [RngIntElt]`

Given a sparse matrix $A$ having $r$ rows, and an integer $i$ such that $1 \leq i \leq r$, return the support of row $i$ of $A$; i.e., the column numbers of the non-zero entries of row $i$ of $A$.

`Support(A): MtrxSprs -> [ <RngIntElt, RngIntElt, RngElt> ]`

Given a sparse matrix $A$, return the sequence of all pairs `<i, j>` such that the $[i, j]$-th entry of $A$ is non-zero.

### 28.3.2 Weights

`RowWeight(A, i): MtrxSprs, RngIntElt -> RngIntElt`

Given a sparse matrix $A$ with $m$ rows and an integer $i$ such that $1 \leq i \leq m$, return the weight (number of non-zero entries) of the $i$-th row of $A$.

**RowWeights(A): MtrxSprs -> [RngIntElt]**

Given a sparse matrix $A$ with $m$ rows, return the length $m$ sequence of integers whose $i$-th entry is the weight of the $i$-th row of $A$.

**ColumnWeight(A, j): MtrxSprs, RngIntElt -> RngIntElt**

Given a sparse matrix $A$ with $n$ columns and an integer $j$ such that $1 \leq j \leq n$, return the weight (number of non-zero entries) of the $j$-th column of $A$.

**ColumnWeights(A): MtrxSprs -> [RngIntElt]**

Given a sparse matrix $A$ with $n$ columns, return the length $n$ sequence of integers whose $j$-th entry is the weight of the $j$-th column of $A$.

## 28.4 Accessing or Modifying Entries

The following functions and procedures enable the user to access or set individual entries of sparse matrices.

**A[i]: MtrxSprs, RngIntElt -> ModTupRngElt**

Given a sparse matrix $A$ over a ring $R$ having $m$ rows and $n$ columns, and an integer $i$ such that $1 \leq i \leq m$, return the $i$-th row of $A$, as a dense vector of length $n$ (lying in $R^n$).

**A[i, j]: MtrxSprs, RngIntElt, RngIntElt -> RngElt**

Given a sparse matrix $A$ over a ring $R$ having $m$ rows and $n$ columns, integers $i$ and $j$ such that $1 \leq i \leq m$ and $1 \leq j \leq n$, return the $(i, j)$-th entry of $A$, as an element of the ring $R$.

**A[i, j] := x: MtrxSprs, RngIntElt, RngIntElt, RngElt**

Given a sparse matrix $A$ over a ring $R$ having $m$ rows and $n$ columns, integers $i$ and $j$ such that $1 \leq i \leq m$ and $1 \leq j \leq n$, and a ring element $x$ coercible into $R$, modify the $(i, j)$-th entry of $A$ to be $x$. Here $i$ and $j$ must be within the ranges given by the current dimensions of $A$; see *SetEntry* below for a procedure to automatically extend $A$ if necessary.

**SetEntry(~A, i, j, x): MtrxSprs, RngIntElt, RngIntElt, RngElt**

(Procedure.) Given a sparse matrix $A$ over a ring $R$, integers $i, j \geq 1$, and a ring element $x$ coercible into $R$, modify the $(i, j)$-th entry of $A$ to be $x$. The entry specified by $i$ and $j$ is allowed to be beyond the current dimensions of $A$; if so, $A$ is automatically extended to have at least $i$ rows and $j$ columns.

This procedure will be commonly used in situations where the final size of the matrix is not known as an algorithm proceeds (e.g., in index-calculus methods). One can create the $0 \times 0$ sparse matrix over $\mathbb{Z}$, say, and then call `SetEntry` to build up the matrix dynamically. See the example *Discrete Logarithms by Linear Sieve* below, which uses this technique.

Note that extending the dimensions of $A$ with a very large $i$ or $j$ will not in itself consume much memory, but if $A$ then becomes dense or is passed to some algorithm, then the memory needed may of course be proportional to the dimensions of $A$.

**Example: Accessing Sparse Matrix Entries** (ex-167588)

This example demonstrates simple ways of accessing the entries of sparse matrices.

```
> A := SparseMatrix(2, 3, [<1,2,3>, <2,3,-1>]);
> A;
Sparse matrix with 2 rows and 3 columns over Integer Ring
> Matrix(A);
[ 0  3  0]
[ 0  0 -1]
> A[1];
(0 3 0)
> A[1, 3]:=5;
> A[1];
(0 3 5)
```

We next extend $A$ using the procedure `SetEntry`.

```
> SetEntry(~A, 1, 5, -7);
> A;
Sparse matrix with 2 rows and 5 columns over Integer Ring
> Matrix(A);
[ 0  3  5  0 -7]
[ 0  0 -1  0  0]
```

A common situation is to start with the empty $0\times 0$ matrix over $\mathbb{Z}$ and then to extend it dynamically.

```
> A := SparseMatrix();
> A;
Sparse matrix with 0 rows and 0 columns over Integer Ring
> SetEntry(~A, 1, 4, -2);
> A;
Sparse matrix with 1 row and 4 columns over Integer Ring
> SetEntry(~A, 2, 3, 8);
> A;
Sparse matrix with 2 rows and 4 columns over Integer Ring
> Matrix(A);
[ 0  0  0 -2]
[ 0  0  8  0]
> SetEntry(~A, 200, 319, 1);
> SetEntry(~A, 200, 3876, 1);
> A;
Sparse matrix with 200 rows and 3876 columns over Integer Ring
> Nrows(A);
200
> Ncols(A);
3876
> NNZEntries(A);
4
> Density(A);
0.00000515995872033023735810113519 0
> Support(A, 200);
[ 319, 3876 ]
```

### 28.4.1 Extracting and Inserting Blocks

The following functions enable the extraction of certain rows, columns, or general submatrices, or the replacement of a block by another sparse matrix.

**Submatrix**(A, i, j, p, q): MtrxSprs, RngIntElt, RngIntElt, RngIntElt, RngIntElt -> MtrxSprs

**ExtractBlock**(A, i, j, p, q): MtrxSprs, RngIntElt, RngIntElt, RngIntElt, RngIntElt -> MtrxSprs

Given an $m \times n$ sparse matrix $A$ and integers $i$, $j$, $p$, and $q$ such that $1 \leq i \leq i+p \leq m+1$, and $1 \leq j \leq j+q \leq n+1$, return the $p \times q$ submatrix of $A$ rooted at $(i, j)$. Either or both of $p$ and $q$ may be zero, while $i$ may be $m+1$ if $p$ is zero and $j$ may be $n+1$ if $q$ is zero.

**SubmatrixRange**(A, i, j, r, s): MtrxSprs, RngIntElt, RngIntElt, RngIntElt, RngIntElt -> MtrxSprs

**ExtractBlockRange**(A, i, j, r, s): MtrxSprs, RngIntElt, RngIntElt, RngIntElt, RngIntElt -> MtrxSprs

Given an $m \times n$ sparse matrix $A$ and integers $i$, $j$, $r$, and $s$ such that $1 \leq i$, $i-1 \leq r \leq m$, $1 \leq j$, and $j-1 \leq s \leq n$, return the $r-i+1 \times s-j+1$ submatrix of $A$ rooted at the $(i, j)$-th entry and

extending to the $(r, s)$-th entry, inclusive. $r$ may equal $i - 1$ or $s$ may equal $j - 1$, in which case a sparse matrix with zero rows or zero columns, respectively, will be returned.

`Submatrix(A, I, J): MtrxSprs, [RngIntElt], [RngIntElt] -> MtrxSprs`

Given an $m \times n$ sparse matrix $A$ and integer sequences $I$ and $J$, return the submatrix of $A$ given by the row indices in $I$ and the column indices in $J$.

`InsertBlock(A, B, i, j): MtrxSprs, MtrxSprs, RngIntElt, RngIntElt -> MtrxSprs`

`InsertBlock(~A, B, i, j): MtrxSprs, MtrxSprs, RngIntElt, RngIntElt -> MtrxSprs`

Given an $m \times n$ sparse matrix $A$ over a ring $R$, a $p \times q$ sparse matrix $B$ over $R$, and integers $i$ and $j$ such that $1 \leq i \leq i + p \leq m + 1$ and $1 \leq j \leq j + q \leq n + 1$, insert $B$ at position $(i, j)$ in $A$. In the functional version ($A$ is a value argument), this function returns the new sparse matrix and leaves $A$ untouched, while in the procedural version ($\sim A$ is a reference argument), $A$ is modified in place so that the $p \times q$ submatrix of $A$ rooted at $(i, j)$ is now equal to $B$.

`RowSubmatrix(A, i, k): MtrxSprs, RngIntElt, RngIntElt -> MtrxSprs`

Given an $m \times n$ sparse matrix $A$ and integers $i$ and $k$ such that $1 \leq i \leq i + k \leq m + 1$, return the $k \times n$ submatrix of $X$ consisting of rows $[i \dots i + k - 1]$ inclusive. The integer $k$ may be zero and $i$ may also be $m + 1$ if $k$ is zero, but the result will always have $n$ columns.

`RowSubmatrix(A, i): MtrxSprs, RngIntElt -> MtrxSprs`

Given an $m \times n$ sparse matrix $A$ and an integer $i$ such that $0 \leq i \leq m$, return the $i \times n$ submatrix of $X$ consisting of the first $i$ rows. The integer $i$ may be 0, but the result will always have $n$ columns.

`RowSubmatrixRange(A, i, j): MtrxSprs, RngIntElt, RngIntElt -> MtrxSprs`

Given an $m \times n$ sparse matrix $A$ and integers $i$ and $j$ such that $1 \leq i$ and $i - 1 \leq j \leq m$, return the $j - i + 1 \times n$ submatrix of $X$ consisting of rows $[i \dots j]$ inclusive. The integer $j$ may equal $i - 1$, in which case a sparse matrix with zero rows and $n$ columns will be returned.

`ColumnSubmatrix(A, i, k): MtrxSprs, RngIntElt, RngIntElt -> MtrxSprs`

Given an $m \times n$ sparse matrix $A$ and integers $i$ and $k$ such that $1 \leq i \leq i + k \leq n + 1$, return the $m \times k$ submatrix of $X$ consisting of columns $[i \dots i + k - 1]$ inclusive. The integer $k$ may be zero and $i$ may also be $n + 1$ if $k$ is zero, but the result will always have $m$ rows.

`ColumnSubmatrix(A, i): MtrxSprs, RngIntElt -> MtrxSprs`

Given an $m \times n$ sparse matrix $A$ and an integer $i$ such that $0 \leq i \leq n$, return the $m \times i$ submatrix of $X$ consisting of the first $i$ columns. The integer $i$ may be 0, but the result will always have $m$ rows.

`ColumnSubmatrixRange(A, i, j): MtrxSprs, RngIntElt, RngIntElt -> MtrxSprs`

Given an $m \times n$ sparse matrix $A$ and integers $i$ and $j$ such that $1 \leq i$ and $i - 1 \leq j \leq n$, return the $m \times j - i + 1$ submatrix of $X$ consisting of columns $[i \dots j]$ inclusive. The integer $j$ may equal $i - 1$, in which case a sparse matrix with zero columns and $n$ rows will be returned.

## 28.4.2 Row and Column Operations

The following functions and procedures provide elementary row or column operations on sparse matrices. For each operation, there is a corresponding function which creates a new sparse matrix for the result (leaving the input sparse matrix unchanged), and a corresponding procedure which modifies the input sparse matrix in place.

`SwapRows(A, i, j): MtrxSprs, RngIntElt, RngIntElt -> MtrxSprs`

`SwapRows(~A, i, j): MtrxSprs, RngIntElt, RngIntElt`

Given an $m \times n$ sparse matrix $A$ and integers $i$ and $j$ such that $1 \leq i \leq m$ and $1 \leq j \leq m$, swap the $i$-th and $j$-th rows of $A$.

`SwapColumns(A, i, j): MtrxSprs, RngIntElt, RngIntElt -> MtrxSprs`

`SwapColumns(~A, i, j): MtrxSprs, RngIntElt, RngIntElt`

Given an $m \times n$ sparse matrix $A$ and integers $i$ and $j$ such that $1 \leq i \leq n$ and $1 \leq j \leq n$, swap the $i$-th and $j$-th columns of $A$.

`ReverseRows(A): MtrxSprs -> MtrxSprs`
`ReverseRows(~A): MtrxSprs`

Given a sparse matrix $A$, reverse all the rows of $A$.

`ReverseColumns(A): MtrxSprs -> MtrxSprs`
`ReverseColumns(~A): MtrxSprs`

Given a sparse matrix $A$, reverse all the columns of $A$.

`AddRow(A, c, i, j): MtrxSprs, RngElt, RngIntElt, RngIntElt -> MtrxSprs`
`AddRow(~A, c, i, j): MtrxSprs, RngElt, RngIntElt, RngIntElt`

Given an $m \times n$ sparse matrix $A$ over a ring $R$, a ring element $c$ coercible into $R$, and integers $i$ and $j$ such that $1 \leq i \leq m$ and $1 \leq j \leq m$, add $c$ times row $i$ of $A$ to row $j$ of $A$.

`AddColumn(A, c, i, j): MtrxSprs, RngElt, RngIntElt, RngIntElt -> MtrxSprs`
`AddColumn(~A, c, i, j): MtrxSprs, RngElt, RngIntElt, RngIntElt`

Given an $m \times n$ sparse matrix $A$ over a ring $R$, a ring element $c$ coercible into $R$, and integers $i$ and $j$ such that $1 \leq i \leq n$ and $1 \leq j \leq n$, add $c$ times column $i$ of $A$ to column $j$.

`MultiplyRow(A, c, i): MtrxSprs, RngElt, RngIntElt -> MtrxSprs`
`MultiplyRow(~A, c, i): MtrxSprs, RngElt, RngIntElt`

Given an $m \times n$ sparse matrix $A$ over a ring $R$, a ring element $c$ coercible into $R$, and an integer $i$ such that $1 \leq i \leq m$, multiply row $i$ of $A$ by $c$ (on the left).

`MultiplyColumn(A, c, i): MtrxSprs, RngElt, RngIntElt -> MtrxSprs`
`MultiplyColumn(~A, c, i): MtrxSprs, RngElt, RngIntElt`

Given an $m \times n$ sparse matrix $A$ over a ring $R$, a ring element $c$ coercible into $R$, and an integer $i$ such that $1 \leq i \leq n$, multiply column $i$ of $A$ by $c$ (on the left).

`RemoveRow(A, i): MtrxSprs, RngIntElt -> MtrxSprs`
`RemoveRow(~A, i): MtrxSprs, RngIntElt`

Given an $m \times n$ sparse matrix $A$ and an integer $i$ such that $1 \leq i \leq m$, remove row $i$ from $A$ (leaving an $(m-1) \times n$ sparse matrix).

`RemoveColumn(A, j): MtrxSprs, RngIntElt -> MtrxSprs`
`RemoveColumn(~A, j): MtrxSprs, RngIntElt`

Given an $m \times n$ sparse matrix $A$ and an integer $j$ such that $1 \leq j \leq n$, remove column $j$ from $A$ (leaving an $m \times (n-1)$ sparse matrix).

`RemoveRowColumn(A, i, j): MtrxSprs, RngIntElt -> MtrxSprs`
`RemoveRowColumn(~A, i, j): MtrxSprs, RngIntElt`

Given an $m \times n$ sparse matrix $A$ and integers $i$ and $j$ such that $1 \leq i \leq m$ and $1 \leq j \leq n$, remove row $i$ and column $j$ from $A$ (leaving an $(m-1) \times (n-1)$ sparse matrix).

`RemoveZeroRows(A): MtrxSprs -> MtrxSprs`
`RemoveZeroRows(~A): MtrxSprs`

Given a sparse matrix $A$, remove all the zero rows of $A$.

## 28.5 Building Block Matrices

`HorizontalJoin(A, B): MtrxSprs, MtrxSprs -> MtrxSprs`

Given a sparse matrix $A$ with $r$ rows and $c$ columns, and a sparse matrix $B$ with $r$ rows and $d$ columns, both over the same coefficient ring $R$, return the sparse matrix over $R$ with $r$ rows and $(c+d)$ columns obtained by joining $A$ and $B$ horizontally (placing $B$ to the right of $A$).

`VerticalJoin(A, B): MtrxSprs, MtrxSprs -> MtrxSprs`

Given a sparse matrix $A$ with $r$ rows and $c$ columns, and a sparse matrix $B$ with $s$ rows and $c$ columns, both over the same coefficient ring $R$, return the sparse matrix with $(r+s)$ rows and $c$ columns over $R$ obtained by joining $A$ and $B$ vertically (placing $B$ underneath $A$).

`DiagonalJoin(A, B): MtrxSprs, MtrxSprs -> MtrxSprs`

Given matrices $A$ with $a$ rows and $b$ columns and $B$ with $c$ rows and $d$ columns, both over the same coefficient ring $R$, return the sparse matrix with $(a+c)$ rows and $(b+d)$ columns over $R$ obtained by joining $A$ and $B$ diagonally (placing $B$ diagonally to the right of and underneath $A$, with zero blocks above and below the diagonal).

## 28.6 Conversion to and from Dense Matrices

The following functions convert between sparse matrices and normal (dense-representation) matrices.

`Matrix(A): MtrxSprs -> Mtrx`

Given a sparse matrix $A$, return the normal (dense-representation) matrix equal to $A$. This function should only be used if the size of $A$ is reasonably small since otherwise the amount of memory needed to represent the dense-representation matrix may be huge. Printing the result of this operation is a convenient way to display $A$ as a normal (dense) matrix if $A$ is small.

`SparseMatrix(A): Mtrx -> MtrxSprs`

Given a normal (dense-representation) matrix $A$, return the sparse matrix equal to $A$. Note that if there is a fast algorithm available for the sparse matrix type, there is no need to convert a dense-representation matrix to the sparse matrix type first and then to use that algorithm, since Magma does this automatically.

## 28.7 Changing Ring

`ChangeRing(A, R): MtrxSprs, Rng -> MtrxSprs`

`SparseMatrix(R, A): Rng, MtrxSprs -> MtrxSprs`

Given a sparse matrix $A$ over a ring $S$ having $m$ rows and $n$ columns, and another ring $R$, return the $m \times n$ sparse matrix over $R$ obtained by coercing the entries of $A$ from $S$ into $R$.

The argument order to `ChangeRing(A, R)` here is consistent with other forms of `ChangeRing`, while the `SparseMatrix(R, A)` form of this function is provided to be consistent with the sparse matrix creation functions above, for which the destination ring is the first argument, if supplied.

## 28.8 Predicates

The functions in this section test various properties of sparse matrices.

`A eq B: MtrxSprs, MtrxSprs -> BoolElt`

Given sparse matrices $A$ and $B$, return `true` if and only if $A$ and $B$ are equal.

`IsZero(A): MtrxSprs -> BoolElt`

Given an $m \times n$ sparse matrix $A$ over the ring $R$, return `true` iff $A$ is the $m \times n$ zero sparse matrix.

`IsOne(A): MtrxSprs -> BoolElt`

Given a square $m \times m$ sparse matrix $A$ over the ring $R$, return `true` iff $A$ is the $m \times m$ identity sparse matrix.

`IsMinusOne(A): MtrxSprs -> BoolElt`

Given a square $m \times m$ sparse matrix $A$ over the ring $R$, return `true` iff $A$ is the negation of the $m \times m$ identity sparse matrix.

`IsScalar(A): MtrxSprs -> BoolElt`

Given a square $m \times m$ sparse matrix $A$ over the ring $R$, return `true` iff $A$ is scalar, i.e., iff $A$ is the product of some element of $R$ and the $m \times m$ identity sparse matrix.

`IsDiagonal(A): MtrxSprs -> BoolElt`

Given a square sparse matrix $A$ over the ring $R$, return `true` iff $A$ is diagonal, i.e., iff the only non-zero entries of $A$ are on the diagonal.

`IsSymmetric(A): MtrxSprs -> BoolElt`

Given a square sparse matrix $A$ over the ring $R$, return `true` iff $A$ is symmetric, i.e., iff $A$ equals its transpose.

`IsUpperTriangular(A): MtrxSprs -> BoolElt`

Given a sparse matrix $A$ over the ring $R$, return `true` iff $A$ is upper triangular, i.e., iff the only non-zero entries of $A$ are on or above the diagonal.

`IsLowerTriangular(A): MtrxSprs -> BoolElt`

Given a sparse matrix $A$ over the ring $R$, return `true` iff $A$ is lower triangular, i.e., iff the only non-zero entries of $A$ are on or below the diagonal.

## 28.9 Elementary Arithmetic

`A + B: MtrxSprs, MtrxSprs -> MtrxSprs`

Given $m \times n$ sparse matrices $A$ and $B$ over a ring $R$, return $A + B$.

`A - B: MtrxSprs, MtrxSprs -> MtrxSprs`

Given $m \times n$ sparse matrices $A$ and $B$ over a ring $R$, return $A - B$.

`A * B: MtrxSprs, MtrxSprs -> MtrxSprs`

Given an $m \times n$ sparse matrix $A$ over a ring $R$ and an $n \times p$ sparse matrix $B$ over $R$, return the $m \times p$ sparse matrix $A \cdot B$ over $R$.

`x * A: RngElt, MtrxSprs -> MtrxSprs`

`A * x: MtrxSprs, RngElt -> MtrxSprs`

Given an $m \times n$ sparse matrix $A$ over a ring $R$ and a ring element $x$ coercible into $R$, return the scalar product $x \cdot A$.

`-A: MtrxSprs -> MtrxSprs`

Given a sparse matrix $A$, return $-A$.

`A ^-1: MtrxSprs, RngIntElt -> MtrxSprs`

Given a invertible square sparse matrix $A$ over a ring $R$, return the inverse $B$ of $A$ so that $A \cdot B = B \cdot A = 1$. The coefficient ring $R$ must be either a field, a Euclidean domain, or a ring with an exact division algorithm and having characteristic equal to zero or greater than $m$ (this includes most commutative rings).

`A ^ n: MtrxSprs, RngIntElt -> MtrxSprs`

Given a square sparse matrix $A$ over a ring $R$ and an integer $n$, return the matrix power $A^n$. $A^0$ is defined to be the identity matrix for any square matrix $A$ (even if $A$ is zero). If $n$ is negative, $A$ must be invertible (see the previous function), and the result is $(A^{-1})^{-n}$.

`Transpose(A): MtrxSprs -> MtrxSprs`

Given an $m \times n$ sparse matrix $A$ over a ring $R$, return the transpose of $A$, which is simply the $n \times m$ sparse matrix over $R$ whose $(i, j)$-th entry is the $(j, i)$-th entry of $A$.

## 28.10 Multiplying Vectors or Matrices by Sparse Matrices

These functions allow the multiplication of a normal (dense-representation) vector by a sparse matrix.

`v * A: ModTupRng, MtrxSprs -> ModTupRng`

`V * A: Mtrx, MtrxSprs -> Mtrx`

Given a dense-representation vector $v$ or dense-representation matrix $V$ with $c$ columns, together with a sparse $c \times n$ matrix $A$, both over a ring $R$, return the product $v \cdot A$ or $V \cdot A$.

This is generally fast if $A$ is sparse and uses minimal memory.

`MultiplyByTranspose(v, A): ModTupRng, MtrxSprs -> ModTupRng`

`MultiplyByTranspose(V, A): Mtrx, MtrxSprs -> Mtrx`

Given a dense-representation vector $v$ or dense-representation matrix $V$ with $c$ columns, together with a sparse $n \times c$ matrix $A$, both over a ring $R$, return the product of $v$ or $V$ by the transpose of $A$.

This is generally fast if $A$ is sparse, and is much faster than computing the transpose of $A$ first. For example, if the vector-matrix product $v \cdot A \cdot A^{tr}$ is required, then the function call `MultiplyByTranspose(v*A, A)` should be used to avoid forming the matrix $A \cdot A^{tr}$ which is usually dense. This product occurs in iterative algorithms such as Lanzcos.

## 28.11 Non-trivial Properties

The following functions compute non-trivial properties of sparse matrices.

### 28.11.1 Nullspace and Rowspace

The following functions compute nullspaces (solving equations of the form $V \cdot A = 0$) or rowspaces of sparse matrices.

`Nullspace(A): MtrxSprs -> ModTupRng`

`Kernel(A): MtrxSprs -> ModTupRng, Map`

Given an $m \times n$ sparse matrix $A$ over a ring $R$, return the nullspace of $A$ (or the kernel of $A$, considered as a linear transformation or map), which is the $R$-space consisting of all vectors $v$ of length $m$ such that $v \cdot A = 0$. Since the result will be given in the dense representation, both the nullity of $A$ and the number of rows of $A$ must both be reasonably small.

The algorithm first performs sparse elimination using Markowitz pivoting ([DEJ84, Sec. 9.2]) to obtain a smaller dense matrix, then the nullspace algorithm for dense-representation matrices is applied to this matrix.

`NullspaceMatrix(A): MtrxSprs -> Mtrx`

`KernelMatrix(A): MtrxSprs -> Mtrx`

Given an $m \times n$ sparse matrix $A$ over a ring $R$, return a (dense-representation) basis matrix of the nullspace of $A$. This is a matrix $N$ having $m$ columns and the maximal number of independent rows subject to the condition that $N \cdot A = 0$. This function has the advantage that the nullspace is not returned as a $R$-space, so echelonization of the resulting nullspace may be avoided.

`NullspaceOfTranspose(A): MtrxSprs -> ModTupRng`

This function is equivalent to `Nullspace(Transpose(A))`, but will be more efficient in space for large matrices, since the transpose may not have to be explicitly constructed to compute the nullspace.

`Rowspace(A): MtrxSprs -> ModTupRng`

Given an $m \times n$ sparse matrix $A$ over a ring $R$, return the rowspace of $A$, which is the $R$-space generated by the rows of $A$. Since the result will be given in the dense representation, the rank and the number of columns of $A$ must both be reasonably small.

### 28.11.2 Rank

`Rank(A): MtrxSprs -> RngIntElt`

Given an $m \times n$ sparse matrix $A$ over a ring $R$, return the rank of $A$. The algorithm first performs sparse elimination using Markowitz pivoting ([DEJ84, Sec. 9.2]) to obtain a smaller dense matrix, then the rank algorithm for dense-representation matrices is applied to this matrix.

## 28.12 Determinant and Other Properties

**Determinant**(A: parameters): MtrxSprs -> RngElt

| | |
|---|---|
| MonteCarloSteps: RngIntElt | Default: |

Given a square sparse matrix $A$ over the ring $R$, return the determinant of $A$ as an element of $R$. $R$ may be any commutative ring. The algorithm first performs sparse elimination using Markowitz pivoting ([DEJ84, Sec. 9.2]) to obtain a smaller dense matrix, then the determinant algorithm for dense-representation matrices is applied to this matrix. If the parameter **MonteCarloSteps** is given, then this is passed to the dense algorithm for the dense matrix.

## 28.13 Minimal and Characteristic Polynomials and Eigenvalues

The functions in this section deal with minimal and characteristic polynomials and related operations and they match the functions defined for dense matrices; see Section *Minimal and Characteristic Polynomials and Eigenvalues* for more information on the functions and the possible parameters.

**MinimalPolynomial**(A: parameters): MtrxSprs -> RngUPolElt

Given a square sparse matrix $A$ over a ring $R$, return the minimal polynomial of $A$.

**CharacteristicPolynomial**(A: parameters): MtrxSprs -> RngUPolElt

| | |
|---|---|
| Al : MonStg | Default: "Modular" |
| Proof: BoolElt | Default: true |

Given a square sparse matrix $A$ over a ring $R$, return the characteristic polynomial of $A$.

**MinimalAndCharacteristicPolynomials**(A: parameters): MtrxSprs -> RngUPolElt, RngUPolElt

**MCPolynomials**(A): MtrxSprs -> RngUPolElt, RngUPolElt

Given a square sparse matrix $A$ over a ring $R$, return the minimal and characteristic polynomials of $A$.

**FactoredMinimalPolynomial**(A: parameters): MtrxSprs -> [ <RngUPolElt, RngIntElt>]

Given a square sparse matrix $A$ over a ring $R$, return the factorization of the minimal polynomial of $A$.

**FactoredCharacteristicPolynomial**(A: parameters): MtrxSprs -> [ <RngUPolElt, RngIntElt>]

Given a square sparse matrix $A$ over a ring $R$, return the factorisation of the characteristic polynomial of $A$.

**FactoredMinimalAndCharacteristicPolynomials**(A: parameters): MtrxSprs -> [<RngUPolElt, RngIntElt>], [<RngUPolElt, RngIntElt>]

**FactoredMCPolynomials**(A: parameters): MtrxSprs -> [<RngUPolElt, RngIntElt>], [<RngUPolElt, RngIntElt>]

Given a square matrix sparse $A$ over a ring $R$, return the factorizations of the minimal and characteristic polynomials of $A$, respectively.

**Eigenvalues**(A): MtrxSprs -> { <FldElt, RngIntElt> }

Given a square matrix $A$ over a ring $R$, return the eigenvalues of $A$ as a set of pairs, each of which gives the value of a distinct eigenvalue and its multiplicity.

**Eigenspace**(A, e): AlgMatElt, FldElt -> ModTup

Given a square matrix $A$ over a ring $R$, and an element $e$ of $R$, return the eigenspace of $A$ corresponding to $e$.

### 28.13.1 Elementary Divisors (Smith Form)

`ElementaryDivisors(A): MtrxSprs -> [RngElt]`

Given an $m \times n$ matrix $A$ over the Euclidean ring or field $R$, return the elementary divisors of $A$. These are simply the non-zero diagonal entries of the Smith form of $A$, in order.

The divisors are returned as a sequence $Q = [e_1, \ldots, e_d]$, $e_i | e_{i+1}$ $(i = 1, \ldots, d-1)$ of $d$ elements of $R$ (which may include ones), where $d$ is the rank of $A$. If $R$ is a field, the result is always a sequence of $r$ ones, where $r$ is the rank of $A$.

A function for computing the Smith normal form is not supplied for sparse matrices since the form may be trivially derived from the elementary divisors, and the sequence $Q$ containing the divisors is often more convenient (and takes less memory). As transformation matrices are dense in general, they are not supported for the sparse representation.

The algorithm first performs sparse elimination using Markowitz pivoting to obtain a smaller dense matrix ([DEJ84, Sec. 9.2]; this is similar to the techniques described in [HHR93]). Then it invokes the dense Smith normal form algorithm for normal (dense-representation) matrices (*SmithForm*).

### 28.13.2 Verbosity

`SetVerbose("SparseMatrix", v): MonStgElt, RngIntElt`

(Procedure.) Set the verbose printing level for all sparse matrix algorithms to be $v$. Currently the legal values for $v$ are `true`, `false`, 0, 1, 2, and 3 (`false` has the same effect as 0, and `true` has the same effect as 1).

## 28.14 Linear Systems (Structured Gaussian Elimination)

`ModularSolution(A, M): MtrxSprs, RngIntElt -> ModTupRng`
`ModularSolution(A, L): MtrxSprs, RngIntEltFact -> ModTupRng`

| `Lanczos: BoolElt` | `Default: false` |
|---|---|

Given a sparse $m \times n$ matrix $A$, defined over the integer ring $\mathbb{Z}$, and a positive integer $M$, compute a vector $v$ such that $v$ satisfies the equation $v \cdot A^{tr} = 0$ modulo $M$. $v$ will be non-zero with high probability.

This function is designed for index-calculus-type algorithms where a large sparse linear system defined by the matrix $A$ is first constructed and then a vector satisfying the above equation modulo $M$ is required. For this reason it is natural that the transpose of $A$ appears in this equation. The example *Discrete Logarithms by Linear Sieve* below illustrates such a situation in detail.

The first version of the function takes the actual integer $M$ as the second argument given and so must be factored as part of the calculation, while the second version of the function takes the factorization sequence $L$ of $M$. If possible, the solution vector is multiplied by a unit modulo $M$ so that its first entry is 1.

The function uses the *Structured Gaussian Elimination* algorithm [LO91b, Sec. 5]. This reduces the linear system to be solved to a much smaller but denser system. By default, the function recurses on the smaller system until it is almost completely dense, and then this dense system is solved using the fast dense modular nullspace algorithm of Magma.

If the parameter `Lanczos` is set to `true`, then the *Lanczos* algorithm [LO91b, Sec. 3] will be used instead. This is generally *very much* slower than the default method (it is often 10 to 50 times slower), but it will take considerably less memory, so may be preferable in the case of extremely large matrices.

For typical matrices arising in index-calculus problems, and for most machines, the default method (reducing to a completely dense system) should solve a linear system of size roughly $100,000 \times 100,000$ using about 500MB of memory while a linear system of size roughly $200,000 \times 200,000$ should require about 1.5GB to 2.0GB of memory.

**Example: Discrete Logarithms by Linear Sieve** (ex-a75220)

In this extended example, we demonstrate the application of the function *ModularSolution* to a sparse matrix arising in an index-calculus algorithm. We present Magma code which performs the first stage of the basic linear sieve [COS86, LO91a] for computing discrete logarithms in a prime finite field $\mathbf{F}_p$. This first stage determines most of the logarithms of the elements of the *factor basis* (which is the set of small primes up to a given limit) by using a *sieve* to compute a sparse linear system which is then solved modulo $p-1$.

Even though the following sieving code is written in the Magma language and so is *much* slower than a serious C implementation, it is sufficiently powerful to be able to compute the first stage logarithms in less than a minute for fields $\mathbf{F}_p$ where $p$ is about $10^{20}$ and $(p-1)/2$ is prime. In comparison, the standard Pohlig-Hellman algorithm based on the Pollard-Rho method would take many hours for such fields. This Magma code can also be adapted for other index-calculus methods.

Suppose $p$ is an odd prime and let $K = \mathbf{F}_p$. Let $Q$ be the *factor base*, an ordered set consisting of all positive primes from 2 to a given limit `qlimit`. Fix a primitive element $\alpha$ of $K$ which is also *prime* as an integer, so $\alpha$ is in $Q$. We wish to compute the discrete logarithms of most of the elements of $Q$ with respect to $\alpha$; i.e., we wish to compute $l_q$ with $\alpha^{l_q} = q$ for most $q \in Q$.

The algorithm uses a *sieve* to search for linear relations involving the logarithms of the elements of $Q$. Let $H = \lfloor \sqrt{p} \rfloor + 1$ and $J = H^2 - p$. We search for pairs of integers $(c_1, c_2)$ with $1 \leq c_1, c_2 \leq$ `climit` (where `climit` is a given limit which is much less than $H$) such that

$$[(H + c_1)(H + c_2)] \bmod p = J + (c_1 + c_2)H + c_1 c_2$$

is *smooth* with respect to $Q$ (i.e., all of its prime factors are in $Q$). If we include these $H + c_i$ in the factor base, then this gives a linear relation modulo $(p-1)$ among the logarithms of the elements of the extended factor base.

Fix $c_1$ with $1 \leq c \leq$ `climit` and suppose we initialize a sieve array (to be indexed by $c_2$) to have zero in each position. For each prime power $q^h$ with $q \in Q$ and $h$ sufficiently small, we compute

$$d = [(J + c_1 H)(H + c_1)^{-1}] \bmod q^h.$$

Then for all $c_2 \equiv d \pmod{q^h}$, it turns out that

$$(H + c_1)(H + c_2) \equiv 0 \pmod{q^h}.$$

So for each $c_2$ with $c_1 \leq c_2 \leq$ `climit` and $c_2 \equiv d \pmod{q^h}$, we add $\log(q)$ to the position of the sieve corresponding to $c_2$. (Ensuring that $c_2 \geq c_1$ avoids redundant relations.) When we have done this for each $q \in Q$, we perform trial division to obtain relations for each of the $c_2$ whose sieve values are beneath a suitable threshold.

We repeat this with a new $c_1$ value until we have more relations than elements of the factor base (typically we make the ratio be 1.1 or 1.2), then we solve the corresponding linear system modulo $M = p - 1$ to obtain the desired logarithms. We ensure that $\alpha$ is the first element of $Q$ so that when the vector is normalized modulo $M$ (so that its first entry is 1), the logarithms will be with respect to $\alpha$. For derivations of the above equations and for further details concerning the sieving, see [LO91a].

In practice, one first writes $M = p - 1 = M_1 \cdot M_2$ where $M_1$ contains the maximal powers of all the small primes dividing $M$ (say, for primes $\leq 10000$). The solution space modulo $M_1$ will often have high dimension, so the logarithms modulo $M_1$ usually cannot be correctly computed from the linear system alone. So we only compute the solution of the linear system modulo $M_2$. It is still possible that some logarithms cannot be determined modulo $M_2$ (e.g., if 2 unknowns occur only in one equation), but usually most of the logarithms will be correctly computed modulo $M_2$. Then the logarithm of each factor basis element can be easily computed modulo $M_1$ by the Pohlig-Hellman algorithm, and the Chinese Remainder Theorem can be used to combine these with the correct modulo-$M_2$ logarithms to compute the logarithms modulo $M$ of most elements of the factor basis.

Similar index-calculus-type algorithms should have techniques for handling small prime divisors of the modulus $M$ when the solution of the linear system has high nullity modulo these small primes.

The following function `Sieve` has been developed by Allan Steel, based on code by Benjamin Costello. Its arguments are the field $K = \mathbf{F}_p$, the factor base prime limit `qlimit`, the $c_1, c_2$ range limit `climit`, and the desired stopping ratio of relations to unknowns. The function returns the sparse relation matrix $A$ together with an indexed set containing the corresponding extended factor base (the small primes and the $H + c_i$ values which yield relations).

```
> function Sieve(K, qlimit, climit, ratio)
>     p := #K;
>     Z := Integers();
>     H := Iroot(p, 2) + 1;
>     J := H^2 - p;
>
>     // Get factor basis of all primes <= qlimit.
>     fb_primes := [p: p in [2 .. qlimit] | IsPrime(p)];
>
>     printf "Factor base has %o primes, climit is %o\n", #fb_primes, climit;
>
>     // Ensure that the primitive element of K is prime (as an integer).
>     a := rep{x: x in [2..qlimit] | IsPrime(x) and IsPrimitive(K!x)};
>     SetPrimitiveElement(K,K!a);
>
>     // Initialize extended factor base FB to fb_primes (starting with a).
>     FB := {@ Z!a @};
>     for x in fb_primes do
>         Include(~FB, x);
>     end for;
>
>     // Initialize A to 0 by 0 sparse matrix over Z.
>     A := SparseMatrix();
>
>     // Get logs of all factor basis primes.
>     log2 := Log(2.0);
>     logqs := [Log(q)/log2: q in fb_primes];
>
>     for c1 in [1 .. climit] do
>
>         // Stop if ratio of relations to unknowns is high enough.
>         if Nrows(A)/#FB ge ratio then break; end if;
>
>         if c1 mod 50 eq 0 then
>             printf "c1: %o, #rows: %o, #cols: %o, ratio: %o\n",
>                 c1, Nrows(A), #FB, RealField(8)!Nrows(A)/#FB;
>         end if;
>
>         // Initialize sieve.
>         sieve := [z: i in [1 .. climit]] where z := Log(1.0);
>         den := H + c1;                  // denominator of relation
>         num := -(J + c1*H);             // numerator
>
>         for i := 1 to #fb_primes do
>             // For each prime q in factor base...
>             q := fb_primes[i];
>             logq := logqs[i];
>
>             qpow := q;
>             while qpow le qlimit do
>                 // For all powers qpow of q up to qlimit...
```

(continues on next page)

(continued from previous page)

```
>
>                 if den mod qpow eq 0 then break; end if;
>                 c2 := num * Modinv(den, qpow) mod qpow;
>                 if c2 eq 0 then c2 := qpow; end if;
>
>                 nextqpow := qpow*q;
>                 // Ensure c2 >= c1 to remove redundant relations.
>                 while c2 lt c1 do
>                     c2 +:= qpow;
>                 end while;
>
>                 while c2 le #sieve do
>                     // Add logq into sieve for c2.
>                     sieve[c2] +:= logq;
>
>                     // Test higher powers of q if nextqpow is too large.
>                     if nextqpow gt qlimit then
>                         prod := (J + (c1 + c2)*H + c1*c2) mod p;
>                         nextp := nextqpow;
>                         while prod mod nextp eq 0 do
>                             sieve[c2] +:= logq;
>                             nextp *:= q;
>                         end while;
>                     end if;
>                     c2 +:= qpow;
>                 end while;
>                 qpow := nextqpow;
>             end while;
>         end for;
>
>         // Check sieve for full factorizations.
>         rel := den * (H + 1);   // the relation
>         relinc := H + c1;       // add to relation to get next relation
>         count := 0;
>         for c2 in [1 .. #sieve] do
>             n := rel mod p;
>             if Abs(sieve[c2] - Ilog2(n)) lt 1 then
>                 fact, r := TrialDivision(n, qlimit);
>                 if r eq 1 then // n is qlimit-smooth
>                     // Include each H + c_i in extended factor basis.
>                     Include(~FB, H + c1);
>                     Include(~FB, H + c2);
>
>                     // Include relation (H + c1)*(H + c2) = fact.
>                     row := Nrows(A) + 1;
>                     for t in fact do
>                         SetEntry(~A, row, Index(FB, t[1]), t[2]);
>                     end for;
>                     if c1 eq c2 then
>                         SetEntry(~A, row, Index(FB, H + c1), -2);
>                     else
>                         SetEntry(~A, row, Index(FB, H + c1), -1);
>                         SetEntry(~A, row, Index(FB, H + c2), -1);
>                     end if;
>                 end if;
>             end if;
```

(continues on next page)

(continued from previous page)

```
>             rel +:= relinc;
>         end for;
>     end for;
>
>     // Check matrix by multiplying out relations in field.
>     assert {&*[(K!FB[j])^A[k, j]: j in Support(A, k)]: k in [1..Nrows(A)]}
>             eq {1};
>
>     return A, FB;
> end function;
```

We first apply the function to a trivial example to illustrate the main points. We let $K$ be $\mathbf{F}_{103}$, and we make the factor basis include primes up to 35, the $c_i$ range be up to 27, and the stopping ratio be 1.1. We first compute the relation matrix $A$ and the extended factor basis $F$.

```
> K := GF(103);
> A, F := Sieve(K, 35, 27, 1.1);
Factor base has 11 primes, climit is 27
> A;
Sparse matrix with 33 rows and 30 columns over Integer Ring
```

We examine a few rows of $A$ and the extended factor basis $F$. Note that 5 is the smallest prime which is primitive in $K$, so it has been inserted first in $F$.

```
> A[1]; A[2]; A[30];
(1 0 0 0 0 1 0 0 0 0 0 -1 -1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 1 1 0 0 0 0 0 0 -1 0 -1 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 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 0 0 -1 0 0 0 0 0)
> F;
{@ 5, 2, 3, 7, 11, 13, 17, 19, 23, 29, 31, 12, 14, 15, 16, 21, 38,
20, 26, 35, 22, 33, 34, 24, 25, 28, 30, 37, 32, 36 @}
```

Next we compute a solution vector $v$ such that $v \cdot A^{tr} = 0$ modulo $M = \#K - 1$.

```
> Factorization(#K-1);
[ <2, 1>, <3, 1>, <17, 1> ]
> v := ModularSolution(A, #K - 1);
> v;
(1 44 39 4 61 72 70 80 24 86 57 25 48 40 74 43 22 0 1 5 3 100 12 69 2
    92 84 93 16 64)
```

Notice that 0 occurs in $v$, so the corresponding logarithm is not known. The vector is normalized so that the logarithm of 5 is 1, as desired. We finally compute the powers of the primitive element of $K$ by each element of $v$ and check that all of these match the entries of $F$ except for the one missed entry. Note also that because $M$ has small prime divisors, it is fortunate that the logarithms are all correct, apart from the missed one.

```
> a := PrimitiveElement(K);
> a;
5
> Z := IntegerRing();
> [a^Z!x: x in Eltseq(v)];
[ 5, 2, 3, 7, 11, 13, 17, 19, 23, 29, 31, 12, 14, 15, 16, 21, 38, 1,
5, 35, 22, 33, 34, 24, 25, 28, 30, 37, 32, 36 ]
> F;
{@ 5, 2, 3, 7, 11, 13, 17, 19, 23, 29, 31, 12, 14, 15, 16, 21, 38, 20,
26, 35, 22, 33, 34, 24, 25, 28, 30, 37, 32, 36 @}
```

Finally, we take $K = \mathbf{F}_p$, where $p$ is the smallest prime above $10^{20}$ such that $(p-1)/2$ is prime. The sparse matrix constructed here is close to $1000 \times 1000$, but the solution vector is still found in less than a second.

```
> K := GF(100000000000000000763);
> Factorization(#K - 1);
[ <2, 1>, <50000000000000000381, 1> ]
> time A, F := Sieve(K, 2200, 800, 1.1);
Factor base has 327 primes, climit is 800
c1: 50, #rows: 222, #cols: 555, ratio: 0.4
c1: 100, #rows: 444, #cols: 738, ratio: 0.60162602
c1: 150, #rows: 595, #cols: 836, ratio: 0.71172249
c1: 200, #rows: 765, #cols: 921, ratio: 0.83061889
c1: 250, #rows: 908, #cols: 973, ratio: 0.9331963
c1: 300, #rows: 1014, #cols: 1011, ratio: 1.003
c1: 350, #rows: 1105, #cols: 1023, ratio: 1.0802
Time: 3.990
> A;
Sparse matrix with 1141 rows and 1036 columns over Integer Ring
> time v := ModularSolution(A, #K - 1);
Time: 0.170
```

We observe that the list consisting of the primitive element powered by the computed logarithms agrees with the factor basis for at least the first 30 elements.

```
> a := PrimitiveElement(K);
> a;
2
> v[1], v[2];
1 71610399209536789314
> a^71610399209536789314;
3
> P := [a^x: x in Eltseq(v)];
> [P[i]: i in [1 .. 30]];
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113 ]
> [F[i]: i in [1 .. 30]];
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113 ]
```

There are in fact 8 logarithms which could not be computed correctly.

```
> [i: i in [1..#F] | P[i] ne F[i]];
[ 671, 672, 673, 737, 738, 947, 1024, 1025 ]
```

A zero value appearing in the place of a logarithm indicates that the nullity of the system was larger than 1, even considered modulo $(\#K - 1)/2$.

```
> [v[i]: i in [1..#F] | P[i] ne F[i]];
[ 0, 22076788376647522787, 73252391663364176895, 0,
33553634905886614528, 42960107136526083388, 0, 57276316725691590267 ]
> [P[i]: i in [1..#F] | P[i] ne F[i]];
[ 1, 7480000052677, 7960000056517, 1, 5220000041437,
8938028430619715763, 1, 11360000275636 ]
```

However, we have found the correct logarithms for nearly all factor basis elements. As pointed out above, the Pollard-Rho method applied to an element of the factor basis for this field would take many hours!

## 28.15 Bibliography


- [COS86] D. Coppersmith, A. M. Odlyzko, and R. Schroeppel. Discrete logarithms in GF($p$). *Algorithmica*, 1:1–15, 1986.
- [DEJ84] I.S. Duff, A.M. Erisman, and J.K.Reid. *Direct methods for sparse matrices*. Monographs on Numerical Analysis. Oxford University Press, 1984.
- [HHR93] George Havas, Derek F. Holt, and Sarah Rees. Recognizing badly presented Z-modules. *Linear Algebra and its Applications*, 192:137–164, 1993.
- [LO91a] B. A. LaMacchia and A. M. Odlyzko. Computation of Discrete Logarithms in Prime Fields. In A.J. Menezes and S. Vanstone, editors, *Advances in Cryptology—CRYPTO 1990*, volume 537 of INCS, 616–618. Springer-Verlag, 1991.
- [LO91b] B. A. LaMacchia and A. M. Odlyzko. Solving Large Sparse Linear Systems over Finite Fields. In A.J. Menezes and S. Vanstone, editors, *Advances in Cryptology—CRYPTO 1990*, volume 537 of LNCS, 109–133. Springer-Verlag, 1991.

# Chapter 29

# Vector Spaces

## 29.1 Introduction

In this chapter we will discuss vector spaces and their linear transformations. Let $K$ be a field. In Magma, the standard $K$-vector space is taken to be the set of $n$-tuples over the field $K$, which we shall write as $K^{(n)}$. A rectangular matrix over a field $K$ is considered to be an element of the vector space consisting of all $m \times n$ matrices over $K$. This vector space will be written as $K^{(m\times n)}$. Let $U$ and $V$ be $K$-vector spaces of dimensions $m$ and $n$, respectively. The set of all linear transformations with domain $U$ and codomain $V$ will be denoted by $\mathrm{Hom}_K(U, V)$. Once bases have been chosen for $U$ and $V$, we may identify $\mathrm{Hom}_K(U, V)$ with $K^{(m\times n)}$. Thus, $K^{(m\times n)}$ is first of all a vector space and all the normal vector space operations apply. However, since it is also a set of mappings, some additional operations arising from this characterization apply. We shall use the term *vector space* or *$K$-vector space* (if we wish to emphasize the coefficient field) to refer to both the space $K^{(n)}$ and the space $K^{(m\times n)}$. If we wish to differentiate between the two, we shall use the term *tuple space* when referring to $K^{(n)}$ and the term *matrix space* referring to $K^{(m\times n)}$.

### 29.1.1 Vector Space Categories

The family of all finite dimensional vector spaces over a given field $K$ forms a category, while the set of all finite dimensional vector spaces forms a family of categories indexed by the field $K$. In this family of categories, objects are vector spaces and the morphisms are linear transformations. The (indexed family of) categories consisting of vector spaces of $n$-tuples has the name `ModTupFld`, while the (indexed family of) categories consisting of vector spaces of $m \times n$-matrices has the name `ModMatFld`.

### 29.1.2 The Construction of a Vector Space

Every vector space $V$ defined over a field $K$ is created either as a subspace of the row space $K^{(n)}$ (tuple spaces) or as a subspace of $K^{(m\times n)}$ (matrix modules). Thus, the construction of a general vector space is a two step process:

**(i)** The appropriate row space $K^{(n)}$, is constructed;

**(ii)** The required vector space $V$ is then defined as a subspace or quotient space of $K^{(n)}$.

## 29.2 Creation of Vector Spaces and Arithmetic with Vectors

### 29.2.1 Construction of a Vector Space

`VectorSpace(K, n): Fld, RngIntElt -> ModTupFld`

`KSpace(K, n): Fld, RngIntElt -> ModTupFld`

Given a field $K$ and a non-negative integer $n$, create the $n$-dimensional vector space $V = K^{(n)}$, consisting of all $n$-tuples over $K$. The vector space is created with respect to the standard basis,

$e_1, \ldots, e_n$, where $e_i$ $(i = 1, \ldots, n)$ is the vector containing a 1 in the $i$-th position and zeros elsewhere.

Use of the functions `VectorSpace` and `KSpace` ensures that subspaces of $V$ will be presented in embedded form.

`KModule(K, n): Fld, RngIntElt -> ModFld`

Given a field $K$ and a non-negative integer $n$, create the $n$-dimensional vector space $V = K^{(n)}$, consisting of all $n$-tuples over $K$. The vector space is created with respect to the standard basis, $e_1, \ldots, e_n$, where $e_i$ $(i = 1, \ldots, n)$ is the vector containing a 1 in the $i$-th position and zeros elsewhere.

Use of the function `KModule` ensures that subspaces of $V$ will be presented in reduced form. In all other respects, a vector space created by this function is identical to one created by `KSpace`.

`KMatrixSpace(K, m, n): Fld, RngIntElt, RngIntElt -> ModMatFld`

Given a field $K$ and integers $m$ and $n$ greater than one, create the vector space $K^{(m \times n)}$, consisting of all $m \times n$ matrices over $K$. The vector space is created with the standard basis, $\{E_{ij} \mid i = 1 \ldots, m, j = 1 \ldots, n\}$, where $E_{ij}$ is the matrix having a 1 in the $(i, j)$-th position and zeros elsewhere. Note that for a matrix space, subspaces will always be presented in embedded form, i.e. there is no reduced mode available for matrix spaces.

`Hom(V, W): ModTupFld, ModTupFld -> ModMatFld`

If $V$ is the vector space $K^{(m)}$ and $W$ is the vector space $K^{(n)}$, create the matrix space $\mathrm{Hom}_K(V, W)$ as the vector space $K^{(m \times n)}$, represented as the set of all $m \times n$ matrices over $K$. The vector space is created with the standard basis, $\{E_{ij} \mid i = 1 \ldots, m, j = 1 \ldots, n\}$, where $E_{ij}$ is the matrix having a 1 in the $(i, j)$-th position and zeros elsewhere.

**Example: Rational Vector Space of Dimension 6** (ex-d5b27e)

We construct the vector space $V$ consisting of 6-tuples over the rational field.

```
> Q := RationalField();
> V := VectorSpace(Q, 6);
> V;
Vector space of dimension 6 over Rational Field
```

**Example: Matrix Space over a Quadratic Field** (ex-4fca7f)

We construct the matrix space $M$ consisting of $3 \times 5$ matrices over the field $\mathbb{Q}(\sqrt{5})$.

```
> K<w> := QuadraticField(5);
> V := KMatrixSpace(K, 3, 5);
> V;
Full Vector Space of 3 by 5 matrices over Quadratic Field Q(w)
```

### 29.2.2 Construction of a Vector Space with Inner Product Matrix

`VectorSpace(K, n, F): Fld, RngIntElt, Mtrx -> ModTupFld`

`KSpace(K, n, F): Fld, RngIntElt, Mtrx -> ModTupFld`

Given a field $K$, a non-negative integer $n$, and a square $n \times n$ symmetric matrix $F$, create the $n$-dimensional vector space $V = K^{(n)}$ (in embedded form), with inner product matrix $F$. This is the same as `VectorSpace(K, n)`, except that the functions `Norm` and `InnerProduct` (see below) will be with respect to the inner product matrix $F$.

### 29.2.3 Construction of a Vector

`elt<V | L>: ModTupFld, List -> ModTupFldElt`

**(1)** Suppose $V$ is a subspace of the vector space $K^{(n)}$. Given elements $a_1, \ldots, a_n$ belonging to $K$, construct the vector $v = (a_1, \ldots, a_n)$ as a vector of $V$. Note that if $v$ is not an element of $V$, an error will result.

**(2)** Suppose $V$ is a subspace of the matrix space $K^{(m\times n)}$. Given elements $a_1, \ldots, a_{mn}$ belonging to $K$, construct the matrix $m = (a_1, \ldots, a_{mn})$ as an element of $V$. Note that if $m$ is not an element of $V$, an error will result.

`V ! Q: ModTupFld, [RngElt] -> ModTupFldElt`

**(1)** Suppose $V$ is a subspace of the vector space $K^{(n)}$. Given elements $a_1, \ldots, a_n$ belonging to $K$, construct the vector $v = (a_1, \ldots, a_n)$ as a vector of $V$. Note that if $v$ is not an element of $V$, an error will result.

**(2)** Suppose $V$ is a subspace of the matrix space $K^{(m\times n)}$. Given elements $a_1, \ldots, a_{mn}$ belonging to $K$, construct the matrix $m = (a_1, \ldots, a_{mn})$ as an element of $V$. Note that if $m$ is not an element of $V$, an error will result.

`CharacteristicVector(V, S): ModTupFld, { RngElt } -> ModTupFldElt`

Given a subspace $V$ of the vector space $K^{(n)}$ together with a set $S$ of integers lying in the interval $[1, n]$, return the characteristic number of $S$ as a vector of $V$.

`V ! 0: ModTupFld, RngIntElt -> ModTupFldElt`

`Zero(V): ModTupFld -> ModTupFldElt`

The zero element for the vector space $V$.

`Random(V): ModTupFld -> ModTupFldElt`

Given a vector space $V$ defined over a finite field, return a random vector.

**Example: Vector over a Finite Field** (ex-df9b72)

We create the 5-dimensional vector space $V$ over $\mathbf{F}_4$ and define the vector $u = (1, w, 1 + w, 0, 0)$, where $w$ is a primitive element of $\mathbf{F}_4$.

```
> K<w> := GaloisField(4);
> V    := VectorSpace(K, 5);
> u    := V ! [1, w, 1+w, 0, 0];
> u;
(1 w w + 1 0 0)
> zero := V ! 0;
> zero;
(0 0 0 0 0)
r := Random(V);
(1 0 w 1 w + 1)
```

**Example: Matrix over a Number Field** (ex-d17116)

We create an element belonging to the space of $3 \times 4$ matrices over the number field $\mathbb{Q}(w)$, where $w$ is a root of $x^7 - 7x + 3$.

```
> R<x> := PolynomialRing(RationalField());
> L<w> := NumberField(x^7 - 7*x + 3);
> L34 := KMatrixSpace(L, 3, 4);
> a := L34 ! [ 1, w, 0, -w, 0, 1+w, 2, -w^3, w-w^3, 2*w, 1/3, 1 ];
> a;
[1    w    0    -1 * w]
[0    w + 1    2    -1 * w^3]
[-1 * w^3 + w    2 * w    1/3    1]
```

## 29.2.4 Deconstruction of a Vector

`ElementToSequence(u): ModTupFldElt -> [RngElt]`

`Eltseq(u): ModTupFldElt -> [RngElt]`

Given an element $u$ belonging to the $K$-vector space $V$, return $u$ in the form of a sequence $Q$ of elements of $V$. Thus, if $u$ is an element of $K^{(n)}$, then $Q[i] = u[i]$, $1 \leq i \leq n$.

### 29.2.5 Arithmetic with Vectors

For the following operations the vectors $u$ and $v$ must belong to the same vector space i.e. the same tuple space $K^{(n)}$ or the same matrix space $K^{(m\times n)}$. The scalar $a$ must belong to the field $K$.

`u + v: ModTupFldElt, ModTupFldElt -> ModTupFldElt`

Sum of the vectors $u$ and $v$, where $u$ and $v$ lie in the same vector space.

`- u: ModTupFldElt -> ModTupFldElt`

Additive inverse of the vector $u$.

`u - v: ModTupFldElt, ModTupFldElt -> ModTupFldElt`

Difference of the vectors $u$ and $v$, where $u$ and $v$ lie in the same vector space.

`x * u: FldElt, ModTupFldElt -> ModTupFldElt`

`u * x: ModTupFldElt, FldElt -> ModTupFldElt`

The scalar product of the vector $u$ belonging to the $K$-vector space and the field element $x$ belonging to $K$.

`u / x: ModTupFldElt, FldElt -> ModTupFldElt`

The scalar product of the vector $u$ belonging to the $K$-vector space and the field element $1/x$ belonging to $K$ where $x$ is non-zero.

`NumberOfColumns(u): ModTupFldElt -> RngIntElt`

`Ncols(u): ModTupFldElt -> RngIntElt`

The number of columns in the vector $u$.

`Depth(u): ModTupRngElt -> RngIntElt`

The index of the first non-zero entry of the vector $u$ (0 if none such).

`(u, v): ModTupFldElt, ModTupFldElt -> FldElt`

`InnerProduct(u, v): ModTupFldElt, ModTupFldElt -> FldElt`

Return the inner product of the vectors $u$ and $v$ with respect to the inner product defined on the space. If an inner product matrix $F$ is given when the space is created, then this is defined to be $u \cdot F \cdot v^{tr}$. Otherwise, this is simply $u \cdot v^{tr}$.

`IsZero(u): ModElt -> BoolElt`

Returns `true` iff the vector $u$ belonging to a vector space is the zero element.

`Norm(u): ModTupFldElt -> FldElt`

Return the norm product of the vector $u$ with respect to the inner product defined on the space. If an inner product matrix $F$ is given when the space is created, then this is defined to be $u \cdot F \cdot u^{tr}$. Otherwise, this is simply $u \cdot u^{tr}$.

`Normalise(u): ModTupFldElt -> ModTupFldElt`

`Normalize(u): ModTupFldElt -> ModTupFldElt`

Given an element $u$, not the zero element, belonging to the $K$-vector space $V$, return $\frac{1}{a} * u$, where $a$ is the first non-zero component of $u$. If $u$ is the zero vector, it is returned. The net effect is that `Normalize(u)` always returns a vector $v$ in the subspace generated by $u$, such that the first non-zero component of $v$ (if existent) is $K!1$.

`Rotate(u, k): ModTupFldElt, RngIntElt -> ModTupFldElt`

Given a vector $u$, return the vector obtained from $u$ by rotating by k coordinate positions.

`Rotate(~u, k): ModTupFldElt, RngIntElt`

Given a vector $u$, destructively rotate $u$ by k coordinate positions.

`NumberOfRows(u): ModTupFldElt -> RngIntElt`

`Nrows(u): ModTupFldElt -> RngIntElt`

The number of rows in the vector $u$ (1 of course; included for completeness).

**Support(u): ModTupFldElt -> { RngElt }**

A set of integers giving the positions of the non-zero components of the vector $u$.

**TensorProduct(u, v): ModTupFldElt, ModTupFldElt -> FldElt**

The tensor (Kronecker) product of the vectors $u$ and $v$. The resulting vector has degree equal to the product of the degrees of $u$ and $v$.

**Trace(u, F): ModTupFldElt, Fld -> ModTupFldElt**

**Trace(u): ModTupFldElt -> ModTupFldElt**

Given a vector belonging to the space $K^{(n)}$, and a subfield $F$ of $K$, return the vector obtained by replacing each component of $u$ by its trace over the subfield $F$. If $F$ is the prime field of $K$, it may be omitted.

**Weight(u): ModTupFldElt -> RngIntElt**

The number of non-zero components of the vector $u$.

**Example: Vector Arithmetic over a Cyclotomic Field** (ex-ca5d9a)

We illustrate the use of the arithmetic operators for module elements by applying them to elements of the 4-dimensional vector space over the field $\mathbb{Q}(w)$, where $w$ is an 8-th root of unity.

```
> K<w> := CyclotomicField(8);
> V := VectorSpace(K, 4);
> x := V ! [ w, w^2, w^4, 0];
> y := V ! [1, w, w^2, w^4];
> x + y;
(  w + 1 w^2 + w w^2 - 1      -1)
> -x;
(  -w -w^2    1    0)
> x - y;
(   w - 1  w^2 - w -w^2 - 1        1)
> w * x;
(w^2 w^3  -w   0)
> y * w^ -4;
(  -1   -w -w^2    1)
> Normalize(x);
(  1   w w^3   0)
> InnerProduct(x, y);
w^2 + 2*w
> z := V ! [1, 0, w, 0 ];
> z;
(1 0 w 0)
> Support(z);
{ 1, 3 }
```

**Example: Defining a Nontrivial Inner Product** (ex-e5eb13)

We illustrate how one can define a non-trivial inner product on a space.

```
> Q := RationalField();
> F := SymmetricMatrix(Q, [1, 0,2, 0,0,3, 1,2,3,4]);
> F;
[1 0 0 1]
[0 2 0 2]
[0 0 3 3]
[1 2 3 4]
> V := VectorSpace(Q, 4, F);
> V;
Full Vector space of degree 4 over Rational Field
```

(continues on next page)

(continued from previous page)

```
Inner Product Matrix:
[1 0 0 1]
[0 2 0 2]
[0 0 3 3]
[1 2 3 4]
> v := V![1,0,0,0];
> Norm(v);
1
> w := V![0,1,0,0];
> Norm(w);
2
> InnerProduct(v, w);
0
> z := V![0,0,0,1];
> Norm(z);
4
> InnerProduct(v, z);
1
> InnerProduct(w, z);
2
```

### 29.2.6 Indexing Vectors and Matrices

The indexing operations have a different meaning depending upon whether they are applied to a tuple space or a matrix space.

`u[i]: ModTupFldElt, RngIntElt -> RngElt`

`u[i]: ModTupFldElt, RngIntElt -> ModTupFldElt`

`u[i, j]: ModTupFldElt, RngIntElt, RngIntElt -> ModTupFldElt`

Given a vector $u$ belonging to a $K$-vector space $V$, the result of this operation depends upon whether $V$ is a tuple or matrix space. If $V$ is a subspace of $K^{(n)}$, and $i$, $1 \leq i \leq n$, is a positive integer, the $i$-th component of the vector $u$ is returned (as an element of the field $K$). If $V$ is a subspace of $K^{(m \times n)}$, and $i$, $1 \leq i \leq m$, is a positive integer, $u[i]$ will return the $i$-th row of the matrix $u$ (as an element of the vector space $K^{(n)}$). Similarly, if $i$ and $j$, $1 \leq i \leq m$, $1 \leq j \leq n$, are positive integers, $u[i, j]$ will return the $(i, j)$-th component of the matrix $u$ (as an element of $K$).

`u[i] := x: ModTupFldElt, RngIntElt, RngElt -> ModTupFldElt`

`u[i] := x: ModTupFldElt, RngIntElt, ModTupFldElt -> ModTupFldElt`

`u[i, j] := x: ModTupFldElt, RngIntElt, RngIntElt, ModTupFldElt -> ModTupFldElt`

Given a vector $u$ belonging to a $K$-vector space $V$, and an element $x$ of $K$, the result of this operation depends upon whether $V$ is a tuple or matrix space. If $V$ is a subspace of $K^{(n)}$, and $i$, $1 \leq i \leq n$, is a positive integer, the $i$-th component of the vector $u$ is redefined to be $x$. If $V$ is a subspace of $K^{(m \times n)}$ and $1 \leq i \leq m$ is a positive integer and $x$ is an element of $K^{(n)}$, `u[i] := x` will redefine the $i$-th row of the matrix $u$ to be the vector $x$, where $x$ must be an element of $K^{(n)}$. Similarly, if $1 \leq i \leq m$, $1 \leq j \leq n$, are positive integers, `u[i, j] := x` will redefine the $(i, j)$-th component of the matrix $u$ to be $x$, where $x$ must be an element of $K$.

**Example: Indexing Tuple and Matrix Spaces** `(ex-876f3e)`

We illustrate the use of the indexing operators for vector space elements by applying them to a 3-dimensional tuple space and a $2 \times 3$ matrix space over the field $\mathbb{Q}(w)$, where $w$ is an 8-th root of unity.

```
> K<w> := CyclotomicField(8);
> V := VectorSpace(K, 3);
> u := V ! [ 1 + w, w^ 2, w^ 4];
```

(continues on next page)

(continued from previous page)

```
> u;
((1 + w)     (w^2)           -1)
> u[3];
-1
> u[3] := 1 + w - w^7;
> u;
((1 + w) (w^2) (1 + w + w^3))
> // We now demonstrate indexing a matrix space
> W := KMatrixSpace(K, 2, 3);
> l := W ! [ 1 - w, 1 + w, 1 + w + w^2, 0, 1 - w^7, 1 - w^3 + w^6 ];
> l;
[(1 - w) (1 + w) (1 + w + w^2)]
[0 (1 + w^3) (1 - w^2 - w^3)]
> l[2];
(0 (1 + w^3) (1 - w^2 - w^3))
> l[2,2];
(1 + w^3)
> m := l[2];
> m;
(0 (1 + w^3) (1 - w^2 - w^3))
> l[2] := u;
> l;
[(1 - w) (1 + w) (1 + w + w^2)]
[(1 + w) (w^2) -1]
> l[2, 3] := 1 + w - w^7;
> l;
[(1 - w) (1 + w) (1 + w + w^2)]
[(1 + w) (w^2) (1 + w + w^3)]
```

## 29.3 Subspaces, Quotient Spaces, and Homomorphisms

### 29.3.1 Construction of Subspaces

The conventions defining the presentations of subspaces and quotient spaces are as follows:

- If $V$ has been created using the function `VectorSpace` or `MatrixSpace`, then every subspace and quotient space of $V$ is given in terms of a basis consisting of elements of $V$, i.e. by means of an embedded basis.
- If $V$ has been created using the function `RModule`, then every subspace and quotient space of $V$ is given in terms of a reduced basis.

**`sub<V | L>: ModTupFld, List -> ModTupFld`**

Given a $K$-vector space $V$, construct the subspace $U$ generated by the elements of $V$ specified by the list $L$. Each term $L_i$ of the list $L$ must be an expression defining an object of one of the following types:

**(a)** A sequence of $n$ elements of $K$ defining an element of $V$;

**(b)** A set or sequence whose terms are elements of $V$;

**(c)** A subspace of $V$;

**(d)** A set or sequence whose terms are subspaces of $V$.

The generators stored for $U$ consist of the vectors specified by terms $L_i$ together with the stored generators for subspaces specified by terms of $L_i$. Repetitions of a vector and occurrences of the zero vector are removed (unless $U$ is the trivial subspace).

The constructor returns the subspace $U$ and the inclusion homomorphism $f : U \to V$. If $V$ is of embedded type, the basis constructed for $U$ consists of elements of $V$. If $V$ is of standard type, a standard basis is constructed for $U$.

`Morphism(U, V): ModTupFld, ModTupFld -> RModMatElt`

Assuming the vector space $U$ has been created as a subspace of $V$, the function returns the matrix defining the embedding of $U$ into $V$.

**Example: Golay Code in a Vector Space** (ex-466045)

The ternary Golay code is a six-dimensional subspace of the vector space $K^{(11)}$, where $K$ is $\mathbf{F}_3$. This subspace is first constructed in the space constructed by the `VectorSpace` function.

```
> K11 := VectorSpace(FiniteField(3), 11);
> G3 := sub< K11  |
>     [1,0,0,0,0,0,1,1,1,1,1], [0,1,0,0,0,0,0,1,2,2,1],
>     [0,0,1,0,0,0,1,0,1,2,2], [0,0,0,1,0,0,2,1,0,1,2],
>     [0,0,0,0,1,0,2,2,1,0,1], [0,0,0,0,0,1,1,2,2,1,0] >;
> G3;
Vector space of degree 11, dimension 6 over GF(3)
Generators:
(1 0 0 0 0 0 1 1 1 1 1)
(0 1 0 0 0 0 0 1 2 2 1)
(0 0 1 0 0 0 1 0 1 2 2)
(0 0 0 1 0 0 2 1 0 1 2)
(0 0 0 0 1 0 2 2 1 0 1)
(0 0 0 0 0 1 1 2 2 1 0)
Echelonized basis:
(1 0 0 0 0 0 1 1 1 1 1)
(0 1 0 0 0 0 0 1 2 2 1)
(0 0 1 0 0 0 1 0 1 2 2)
(0 0 0 1 0 0 2 1 0 1 2)
(0 0 0 0 1 0 2 2 1 0 1)
(0 0 0 0 0 1 1 2 2 1 0)
```

**Example: Golay Code in an R-Module** (ex-1f57d1)

We now construct the ternary Golay code starting with the vector space constructed using the `RModule` function. In this case the subspace is presented on a reduced basis.

```
> K11 := RModule(FiniteField(3), 11);
> G3 := sub< K11 |
>     [1,0,0,0,0,0,1,1,1,1,1], [0,1,0,0,0,0,0,1,2,2,1],
>     [0,0,1,0,0,0,1,0,1,2,2], [0,0,0,1,0,0,2,1,0,1,2],
>     [0,0,0,0,1,0,2,2,1,0,1], [0,0,0,0,0,1,1,2,2,1,0] >;
> G3;
KModule G3 of dimension 6 with base ring GF(3)
> Basis(G3);

[
    G3: (1 0 0 0 0 0),
    G3: (0 1 0 0 0 0),
    G3: (0 0 1 0 0 0),
    G3: (0 0 0 1 0 0),
    G3: (0 0 0 0 1 0),
    G3: (0 0 0 0 0 1)
]
> f := Morphism(G3, K11);
> f;
```

(continues on next page)

(continued from previous page)

```
[1 0 0 0 0 0 1 1 1 1 1]
[0 1 0 0 0 0 0 1 2 2 1]
[0 0 1 0 0 0 1 0 1 2 2]
[0 0 0 1 0 0 2 1 0 1 2]
[0 0 0 0 1 0 2 2 1 0 1]
[0 0 0 0 0 1 1 2 2 1 0]
```

### 29.3.2 Construction of Quotient Vector Spaces

`quo<V | L>: ModTupFld, List -> ModTupFld, Map`

Given a $K$-vector space $V$, construct the quotient vector space $W = V/U$, where $U$ is the subspace generated by the elements of $V$ specified by the list $L$. Each term $L_i$ of the list $L$ must be an expression defining an object of one of the following types:

**(a)** A sequence of $n$ elements of $K$ defining an element of $V$;

**(b)** A set or sequence whose terms are elements of $V$;

**(c)** A subspace of $V$;

**(d)** A set or sequence whose terms are subspaces of $V$.

The generators constructed for $U$ consist of the elements specified by terms $L_i$ together with the stored generators for subspaces specified by terms of $L_i$.

The constructor returns the quotient space $W$ and the natural homomorphism $f : V \to W$.

`V / U: ModTupFld, ModTupFld -> ModTupFld, Map`

Given a subspace $U$ of the vector space $V$, construct the quotient space $W$ of $V$ by $U$. If $r$ is defined to be $\dim(V) - \dim(U)$, then $W$ is created as an $r$-dimensional vector space relative to the standard basis.

The constructor returns the quotient space $W$ and the natural homomorphism $f : V \to W$.

**Example: Quotient by the Golay Code** `(ex-4d668d)`

We construct the quotient of $K^{(11)}$ by the Golay code.

```
> K11 := VectorSpace(FiniteField(3), 11);
> Q3, f := quo< K11 |
>  [1,0,0,0,0,0,1,1,1,1,1], [0,1,0,0,0,0,0,1,2,2,1],
>  [0,0,1,0,0,0,1,0,1,2,2], [0,0,0,1,0,0,2,1,0,1,2],
>  [0,0,0,0,1,0,2,2,1,0,1], [0,0,0,0,0,1,1,2,2,1,0] >;
> Q3;
Full Vector space of degree 5 over GF(3)
> f;
Mapping from: ModTupFld: K11 to ModTupFld: Q3
```

**Example: Golay Quotient as a Complement** `(ex-e0a05b)`

If we wished to construct this quotient of $K^{(11)}$ as a subspace of the original space, we could do so using the `Complement` function.

```
> K11 := VectorSpace(FiniteField(3), 11);
> S := sub< K11 |
>     [1,0,0,0,0,0,1,1,1,1,1], [0,1,0,0,0,0,0,1,2,2,1],
>     [0,0,1,0,0,0,1,0,1,2,2], [0,0,0,1,0,0,2,1,0,1,2],
>     [0,0,0,0,1,0,2,2,1,0,1], [0,0,0,0,0,1,1,2,2,1,0] >;
> Complement(K11, S);
Vector space of degree 11, dimension 5 over GF(3)
```

(continues on next page)

(continued from previous page)

```
Echelonized basis:
(0 0 0 0 0 0 1 0 0 0 0)
(0 0 0 0 0 0 0 1 0 0 0)
(0 0 0 0 0 0 0 0 1 0 0)
(0 0 0 0 0 0 0 0 0 1 0)
(0 0 0 0 0 0 0 0 0 0 1)
```

**Example: Quotient of a Rational Matrix Space** (ex-aceeb2)

We construct a subspace and its quotient space in $\mathbb{Q}^{(3\times 4)}$.

```
> Q := RationalField();
> Q3 := VectorSpace(Q, 3);
> Q4 := VectorSpace(Q, 4);
> H34 := Hom(Q3, Q4);
> a := H34 ! [ 2, 0, 1, -1/2,  1, 0, 3/2, 4,  4/5, 6/7, 0, -1/3];
> b := H34 ! [ 1/2, -3, 0, 5,  1/3, 2, 4/5, 0,  5, -1, 5, 7];
> c := H34 ! [ -1, 4/9, 1, -4,  5, -5/6, -3/2, 0,  4/3, 7, 0, 7/9];
> d := H34 ! [ -3, 5, 1/3, -1/2,  2/3, 4, -2, 0,  0, 4, -1, 0];
> a, b, c, d;
[   2    0    1 -1/2]
[   1    0  3/2    4]
[ 4/5  6/7    0 -1/3]

[1/2  -3   0   5]
[1/3   2 4/5   0]
[  5  -1   5   7]

[  -1  4/9    1   -4]
[   5 -5/6 -3/2    0]
[ 4/3    7    0  7/9]

[  -3    5  1/3 -1/2]
[ 2/3    4   -2    0]
[   0    4   -1    0]
> U := sub< H34 | a, b, c, d >;
> U:Maximal;
KMatrixSpace of 3 by 4 GHom matrices and dimension 4 over Rational Field
Echelonized basis:

[1    0    0    0]
[-33872/30351    -5164/10117    42559/50585    11560/10117]
[-10514/10117    -121582/70819    -8476/10117    -48292/30351]

[           0            1            0            0]
[ -7797/10117   4803/10117 12861/101170   5940/10117]
[ -7818/10117 -38214/70819  -7821/10117 -10967/10117]

[           0            0            1            0]
[ 31261/10117  28101/20234  -2157/20234  18552/10117]
[161802/50585 291399/70819  20088/10117  33419/10117]

[          0           0           0           1]
[-8624/30351  7445/10117  7696/50585  2408/10117]
[32388/50585 -3562/10117  6272/10117 27580/30351]
> W := H34/U;
```

(continues on next page)

(continued from previous page)

```
> W;
Full Vector space of degree 8 over Rational Field
```

## 29.4 Changing the Coefficient Field

The standard constructions described in Standard Constructions for $R$-modules may be applied to vector spaces. In addition, we may extend or restrict the field of scalars, using the functions described here.

**ExtendField(V, L): ModTupFld, Fld -> ModTupFld, MapHom**

Given a $K$-vector space $V$, with $K$ a field and $L$ an extension of $K$, construct the $L$-vector space $U = V \otimes_K L$. The function returns

**(a)** the vector space $U$; and

**(b)** the inclusion homomorphism $\phi : V \to U$.

**RestrictField(V, L): ModTupFld, Fld -> ModTupFld, MapHom**

Given a $K$-vector space $V$, with $K$ a field and $L$ a subfield of $K$, construct the $L$-vector space $U$ consisting of those vectors of $V$ having all of their components lying in the subfield $L$. The function returns

**(a)** the vector space $U$; and

**(b)** the restriction homomorphism $\phi : V \to U$.

**VectorSpace(V, F): ModTupFld, Fld -> ModTupFld, Map**

**KSpace(V, F): ModTupFld, Fld -> ModTupFld, Map**

**KMatrixSpace(V, F): ModTupFld, Fld -> ModTupFld, Map**

**KModule(V, F): ModTupFld, Fld -> ModTupFld, Map**

Given an $n$-dimensional $K$-vector space $V$, and a subfield $F$ of a finite field or cyclotomic field $K$ such that $K$ has degree $m$ over $F$, construct a vector space $U$ of dimension $mn$ over the field $F$. The function returns

**(a)** the vector space $U$; and

**(b)** a mapping $\phi : V \to U$ such that a vector $(v_1, \ldots, v_i, \ldots, v_n)$ of $V$ is mapped into the vector

$$(u_{11}, \ldots, u_{1n}, \ldots, u_{i1}, \ldots, u_{in}, \ldots, u_{n1}, \ldots u_{nn}),$$

where $(u_{i1}, \ldots, u_{in})$ is the field element $v_i$ written as a vector over the subfield $F$.

## 29.5 Basic Operations

### 29.5.1 Accessing Vector Space Invariants

**V . i: ModTupFld, RngIntElt -> ModTupFldElt**

Given a vector space $V$ and a positive integer $i$, return the $i$-th generating element of $V$.

**CoefficientField(V): ModTupFld -> Fld**

**BaseField(V): ModTupFld -> Fld**

Given a $K$-vector space $V$, return the field $K$.

**Degree(V): ModTupFld -> RngIntElt**

Given a $K$-vector space $V$ which is a subspace of $K^{(n)}$, return $n$.

**Degree(u): ModTupFldElt -> RngIntElt**

Given a vector $u$ belonging to a subspace of the vector space $K^{(n)}$, return $n$.

**Dimension(V): ModTupFld -> RngIntElt**

The dimension of the vector space $V$.

**Generators(V): ModTupFld -> { ModElt}**

The generators for the vector space $V$, returned as a set.

**NumberOfGenerators(M): ModTupFld -> RngIntElt**

**Ngens(M): ModTupFld -> RngIntElt**

The number of generators for the vector space $V$.

**OverDimension(V): ModTupFld -> RngIntElt**

Given a $K$-vector space $V$ which is a subspace of $K^{(n)}$, return $n$.

**OverDimension(u): ModTupFldElt -> RngIntElt**

Given a vector $u$ belonging to a subspace of the vector space $K^{(n)}$, return $n$.

**Generic(V): ModFld -> ModFld**

The generic vector space containing $V$, i.e. the full vector space in which $V$ is naturally embedded.

**Parent(V): ModFld -> SetPow**

The power structure for the vector space $V$ (the set consisting of all finite dimensional vector spaces).

### 29.5.2 Membership and Equality

**v in V: ModTupFldElt, ModTupFld -> BoolElt**

Returns `true` if the element $v$ lies in the vector space $V$, where $v$ and $V$ belong to a common space.

**v notin V: ModTupFldElt, ModTupFld -> BoolElt**

Returns `true` if the element $v$ does not lie in the vector space $V$, where $v$ and $V$ belong to a common space.

**U subset V: ModTupFld, ModTupFld -> BoolElt**

Returns `true` if the $K$-vector space $U$ is contained in the $K$-vector space $V$, where $U$ and $V$ are subspaces of some common vector space.

**U notsubset V: ModTupFld, ModTupFld -> BoolElt**

Returns `true` if the $K$-vector space $U$ is not contained in the $K$-vector space $V$, where $U$ and $V$ are subspaces of some common vector space.

**U eq V: ModTupFld, ModTupFld -> BoolElt**

Returns `true` if the subspaces $U$ and $V$ are equal, where $U$ and $V$ belong to a common vector space.

**U ne V: ModTupFld, ModTupFld -> BoolElt**

Returns `true` if the subspaces $U$ and $V$ are not equal, where $U$ and $V$ belong to a common vector space.

### 29.5.3 Operations on Subspaces

**U + V: ModTupFld, ModTupFld -> ModTupFld**

Sum of the subspaces $U$ and $V$, where $U$ and $V$ must be subspaces of a common vector space.

**U meet V: ModTupFld, ModTupFld -> ModTupFld**

Intersection of the subspaces $U$ and $V$, where $U$ and $V$ must be subspaces of a common vector space.

`U meet:= V: ModTupFld, ModTupFld -> ModTupFld`

Replace $U$ with the intersection of the subspaces $U$ and $V$, where $U$ and $V$ must be subspaces of a common vector space.

`&meet S: [ ModTupFld ] -> ModTupFld`

Intersection of the subspaces of the set or sequence $S$, which must be subspaces of a common vector space.

`TensorProduct(U, V): ModTupFld, ModTupFld -> FldElt`

The tensor (Kronecker) product of the vector spaces $U$ and $V$, generated by all the tensor products of elements of $U$ by elements of $V$. The resulting vector space has degree equal to the product of the degrees of $U$ and $V$.

`Complement(V, U): ModTupFld, ModTupFld -> ModTupFld`

Given a subspace $U$ of the vector space $V$, construct a complement for $U$ in $V$ (a subspace of $V$).

Note that this complement is always with respect to the standard inner product, and thus need not be orthogonal with respect to a given one.

`Transversal(V, U): ModTupFld, ModTupFld -> { ModTupFldELt}`

Given a subspace $U$ of the vector space $V$ over a finite field, return a transversal for $U$ in $V$ as a set of vectors.

## 29.6 Reducing Vectors Relative to a Subspace

`ReduceVector(W, v): ModTupRng, ModTupRngElt -> ModTupRngElt`

(Function.) Given a vector $v$ from a tuple module $V$ and a submodule $W$ of $V$, return the reduction of $v$ with respect to $W$ (that is, the canonical representative of the coset $v + W$).

Note that the reduction is done with respect to the standard inner product.

`ReduceVector(W, ~v): ModTupRng, ModTupRngElt`

(Procedure.) Given a vector $v$ from a tuple module $V$ and a submodule $W$ of $V$, replace $v$ with its reduction of with respect to $W$ (that is, the canonical representative of the coset $v + W$).

Note that the reduction is done with respect to the standard inner product.

`DecomposeVector(U, v): ModTupRng, ModTupRngElt -> ModTupRngElt, ModTupRngElt`

Given a vector $v$ from a tuple module $V$ and a submodule $U$ of $V$, return the unique $u$ in $U$ and $w$ in the complement to $U$ in $U+ < v >$ such that $v = u + w$.

Note that the reduction is done with respect to the standard inner product, and the complement noted above need not be orthogonal.

## 29.7 Bases

This section is concerned with the construction of bases for vector spaces.

`VectorSpaceWithBasis(Q): [ModTupFldElt] -> ModTupFld`
`VectorSpaceWithBasis(a): AlgMatElt -> ModTupFld`
`VectorSpaceWithBasis(a): ModMatFldElt -> ModTupFld`
`KSpaceWithBasis(Q): [ModTupFldElt] -> ModTupFld`
`KSpaceWithBasis(a): AlgMatElt -> ModTupFld`
`KSpaceWithBasis(a): ModMatFldElt -> ModTupFld`
`KModuleWithBasis(Q): [ModTupFldElt] -> ModTupFld`

Create a vector space having as basis the terms of $B$ (rows of $a$).

`Basis(V): ModTupFld -> [ModTupFldElt]`

The current basis for the vector space $V$, returned as a sequence of vectors.

`BasisElement(V, i): ModTupFld, RngIntElt -> ModTupFldElt`

The $i$-th basis element for the vector space $V$.

`BasisMatrix(V): ModTupFld -> ModMatElt`

The current basis for the vector space $V$, returned as the rows of a matrix belonging to the matrix space $K^{(m\times n)}$, where $m$ is the dimension of $V$ and $n$ is the over-dimension of $V$.

`Coordinates(V, v): ModTupFld, ModTupFldElt -> [FldElt]`

Given a vector $v$ belonging to the $r$-dimensional $K$-vector space $V$, with basis $v_1, \ldots, v_r$, return a sequence $[a_1, \ldots, a_r]$ of elements of $K$ giving the coordinates of $v$ relative to the $V$-basis: $v = a_1 * v_1 + \cdots + a_r * v_r$.

`Dimension(V): ModTupFld -> RngIntElt`

The dimension of the vector space $V$.

`ExtendBasis(Q, U): [ModTupFldElt], ModTupFld -> [ModTupFldElt]`

Given a sequence $Q$ containing $r$ linearly independent vectors belonging to the vector space $U$, extend the vectors of $Q$ to a basis for $U$. The basis is returned in the form of a sequence $T$ such that $T[i] = Q[i], i = 1, \ldots r$.

`ExtendBasis(U, V): ModTupFld, ModTupFld -> [ModTupFldElt]`

Given an $r$-dimensional subspace $U$ of the vector space $V$, return a basis for $V$ in the form of a sequence $T$ of elements such that the first $r$ elements correspond to the given basis vectors for $U$.

`IsIndependent(S): { ModTupFldElt} -> BoolElt`

Given a set $S$ of elements belonging to the vector space $V$, return `true` if the elements of $S$ are linearly independent.

`IsIndependent(Q): [ ModTupFldElt ] -> BoolElt`

Given a sequence $Q$ of elements belonging to the vector space $V$, return `true` if the terms of $Q$ are linearly independent.

**Example: Bases of the Ternary Golay Code** (ex-a4fb0c)

These operations will be illustrated in the context of the subspace $G3$ of the 11-dimensional vector space over $\mathbf{F}_3$ defining the ternary Golay code.

```
> V11 := VectorSpace(FiniteField(3), 11);
> G3  := sub< V11 |  [1,0,0,0,0,0,1,1,1,1,1], [0,1,0,0,0,0,0,1,2,2,1],
>                    [0,0,1,0,0,0,1,0,1,2,2], [0,0,0,1,0,0,2,1,0,1,2],
>                    [0,0,0,0,1,0,2,2,1,0,1], [0,0,0,0,0,1,1,2,2,1,0] >;
> Dimension(G3);
   6
> Basis(G3);
[
    (1 0 0 0 0 0 1 1 1 1 1),
    (0 1 0 0 0 0 0 1 2 2 1),
    (0 0 1 0 0 0 1 0 1 2 2),
    (0 0 0 1 0 0 2 1 0 1 2),
    (0 0 0 0 1 0 2 2 1 0 1),
    (0 0 0 0 0 1 1 2 2 1 0)
]
> S := ExtendBasis(G3, V11);
> S;
[
    (1 0 0 0 0 0 1 1 1 1 1),
    (0 1 0 0 0 0 0 1 2 2 1),
    (0 0 1 0 0 0 1 0 1 2 2),
    (0 0 0 1 0 0 2 1 0 1 2),
    (0 0 0 0 1 0 2 2 1 0 1),
```

(continues on next page)

(continued from previous page)

```
    (0 0 0 0 0 1 1 2 2 1 0),
    (0 0 0 0 0 0 1 0 0 0 0),
    (0 0 0 0 0 0 0 1 0 0 0),
    (0 0 0 0 0 0 0 0 1 0 0),
    (0 0 0 0 0 0 0 0 0 1 0),
    (0 0 0 0 0 0 0 0 0 0 1)
]
> C3:= Complement(V11, G3);
> C3;
Vector space of degree 11, dimension 5 over GF(3)
Echelonized basis:
(0 0 0 0 0 0 1 0 0 0 0)
(0 0 0 0 0 0 0 1 0 0 0)
(0 0 0 0 0 0 0 0 1 0 0)
(0 0 0 0 0 0 0 0 0 1 0)
(0 0 0 0 0 0 0 0 0 0 1)
> G3 + C3;
Full Vector space of degree 11 over GF(3)
> G3 meet C3;
Vector space of degree 11, dimension 0 over GF(3)
> x := Random(G3);
> x;
(1 1 2 0 0 1 1 1 1 2 0)
> c := Coordinates(G3, x);
> c;
[ 1, 1, 2, 0, 0, 1 ]
> G3 ! &+[ c[i] * G3.i : i in [1 .. Dimension(G3)]];
(1 1 2 0 0 1 1 1 1 2 0)
```

## 29.8 Operations with Linear Transformations

Throughout this section, $V$ is a subspace of $K^{(m)}$, $W$ is a subspace of $K^{(n)}$ and $a$ is a linear transformation belonging to $\mathrm{Hom}_K(V, W)$. See also the chapter on general matrices for many other functions applicable to such matrices (e.g., `EchelonForm`).

`v * a: ModTupFldElt, ModMatFldElt -> ModTupFldElt`

`a(v): ModMatFldElt, ModTupFldElt -> ModTupFldElt`

Given an element $v$ belonging to the vector space $V$, and an element $a$ belonging to $\mathrm{Hom}_K(V, W)$, return the image of $v$ under the linear transformation $a$ as an element of the vector space $W$.

`a * b: ModMatRngElt, ModMatRngElt -> ModMatRngElt`

Given a matrix $a$ belonging to $K^{(m\times n)}$ and a matrix $b$ belonging to $K^{(n\times p)}$, for some integers $m$, $n$, $p$, form the product of $a$ and $b$ as an element of $K^{(m\times p)}$.

`Domain(a): ModMatRngElt -> ModTupRng`

The domain of the linear transformation $a$ belonging to $\mathrm{Hom}_K(V, W)$, returned as a subspace of $V$.

`Codomain(a): ModMatRngElt -> ModTupRng`

The codomain of the linear transformation $a$ belonging to $\mathrm{Hom}_K(V, W)$, returned as a subspace of $W$.

`Image(a): ModMatRngElt -> ModTupRng, Map, Map`

The image of the linear transformation $a$ belonging to $\mathrm{Hom}_K(V, W)$, returned as a subspace of $W$.

`Rank(a): ModMatRngElt -> RngIntElt`

The dimension of the image of the linear transformation $a$, i.e., the rank of the matrix $a$.

`Kernel(a): ModMatRngElt -> ModTupFld, Map`

`NullSpace(a): ModMatRngElt -> ModTupFld, Map`

The kernel of the linear transformation $a$ belonging to $\mathrm{Hom}_K(V, W)$, returned as a subspace of $V$.

`Cokernel(a): ModMatRngElt -> ModTupFld, Map`

The cokernel of the linear transformation $a$ belonging to $\mathrm{Hom}_K(V, W)$.

**Example: Operations on Linear Transformations** (ex-135cf0)

We illustrate the map operations for matrix spaces in the following example:

```
> Q := RationalField();
> Q2 := VectorSpace(Q, 2);
> Q3 := VectorSpace(Q, 3);
> Q4 := VectorSpace(Q, 4);
> H23 := Hom(Q2, Q3);
> H34 := Hom(Q3, Q4);
> x := Q2 ! [ -1, 2 ];
> a := H23 ! [ 1/2, 3, 0,  2/3, 4/5, -1 ];
> a;
[1/2   3   0]
[2/3 4/5  -1]
> Domain(a);
Full Vector space of degree 2 over Rational Field
> Codomain(a);
Full Vector space of degree 3 over Rational Field
> x*a;
( 5/6 -7/5   -2)
> b := H34 ! [ 2, 0, 1, -1/2,  1, 0, 3/2, 4,  4/5, 6/7, 0, -9/7];
> b;
[   2    0    1 -1/2]
[   1    0  3/2    4]
[ 4/5  6/7    0 -9/7]
> c := a*b;
> c;
[      4       0       5    47/4]
[    4/3    -6/7   28/15 436/105]
> x*c;
(     -4/3     -12/7    -19/15 -1447/420)
> Image(c);
Vector space of degree 4, dimension 2 over Rational Field
Echelonized basis:
(     1      0    5/4  47/16)
(     0      1  -7/30 -11/40)
> Kernel(c);
Vector space of degree 2, dimension 0 over Rational Field
> Rank(c);
2
> EchelonForm(c);
[     1      0    5/4  47/16]
[     0      1  -7/30 -11/40]
```

# Chapter 30

# Polar Spaces

## 30.1 Introduction

This chapter describes Magma functions for working with quadratic, bilinear, and sesquilinear forms defined on vector spaces. The emphasis is on vector spaces defined over finite fields but in some instances the functions apply more widely. For quadratic forms defined on lattices see Chapter Quadratic Forms. For the interpretation of reflexive forms as algebras with involution see Chapter Algebras with Involution. General references for this material are [Bou07, Tay92].

## 30.2 Reflexive Forms

Let $V$ be a vector space of dimension $n$ over a field $K$. If $\sigma$ is an automorphism of $K$, a $\sigma$-sesquilinear form on the vector space $V$ over $K$ is a map $\beta : V \times V \to K$ such that

$$\begin{aligned}\beta(u_1 + u_2, v) &= \beta(u_1, v) + \beta(u_2, v),\\ \beta(u, v_1 + v_2) &= \beta(u, v_1) + \beta(u, v_2)\\ &\text{and}\\ \beta(au, bv) &= a\sigma(b)\beta(u, v).\end{aligned}$$

for all $u, u_1, u_2, v, v_1, v_2 \in V$ and all $a, b \in K$. If $\sigma$ is the identity, the form is said to be *bilinear*.

A linear transformation $g$ of $V$ is an *isometry* if $g$ preserves $\beta$; it is a *similarity* if it preserves $\beta$ up to a non-zero scalar multiple.

A $\sigma$-sesquilinear form $\beta$ is *reflexive* if for all $u, v \in V$, $\beta(u, v) = 0$ implies $\beta(v, u) = 0$. Any non-zero multiple of a reflexive form is again reflexive with the same group of isometries. By a theorem of Brauer [Bra36] (but sometimes referred to as the Birkhoff–von Neumann theorem), up to a non-zero scalar multiple, there are three types of *non-degenerate* reflexive forms:

**Alternating.** In this case $\sigma$ is the identity, $\beta(u, u) = 0$ for all $u \in V$ and consequently $\beta(u, v) = -\beta(v, u)$ for all $u, v \in V$. The group of isometries is a *symplectic* group.

**Symmetric.** In this case $\sigma$ is the identity and $\beta(u, v) = \beta(v, u)$ for all $u, v \in V$. If the characteristic of $K$ is not two, the group of isometries is an *orthogonal* group. If the characteristic is two, the form is either alternating or pseudo-alternating (see below).

**Hermitian.** In this case $\sigma$ is an automorphism of order two and $\beta(u, v) = \sigma\beta(v, u)$ for all $u, v \in V$. The group of isometries is a *unitary* group.

If $V$ is a vector space $V$ and if $\beta$ is a reflexive form defined on $V$, the partially ordered set of totally isotropic subspaces with respect to $\beta$ is often referred to as a . Similarly, there are polar spaces associated with quadratic forms (see Section *Quadratic Forms*). But throughout this chapter by polar space we shall simply mean a vector space furnished with either a reflexive $\sigma$-sesquilinear form or a quadratic form. See [Bue95, Chap. 2] for an account of polar spaces in a more general context.

Let $K_0$ be the fixed field of $\sigma$. Multiplying an alternating, symmetric, or hermitian form by a non-zero element of $K_0$ leaves the type of the form unchanged.

However, multiplying an hermitian form by a non-zero element of $K$ produces a sesquilinear form $\xi$ and an element $\varepsilon \in K$ such that for all $u, v \in V$, $\xi(v,u) = \varepsilon\sigma\xi(u,v)$, where $\varepsilon\sigma(\varepsilon) = 1$. In this case $\xi$ is said to be $\varepsilon$-hermitian.

**Skew-hermitian.** A reflexive $\sigma$-sesquilinear form is skew-hermitian if the order of $\sigma$ is two and $\xi(v,u) = -\sigma\xi(v,u)$ for all $u, v \in V$. If $\beta$ is hermitian and if $d \in K$ is chosen so that $d \neq \sigma(d)$, then $e = d - \sigma(d)$ satisfies $\sigma(e) = -e$. Thus $\xi(u,v) = e\beta(u,v)$ is skew-hermitian. The group of isometries of $\xi$ coincides with the group of isometries of $\beta$ and it is therefore a *unitary* group.

In the case of fields of characteristic two there is no distinction between hermitian and skew-hermitian forms and moreover, every alternating form is symmetric.

**Pseudo-alternating.** A symmetric form (in characteristic two) which is not alternating is said to be *pseudo-alternating*.

The three types of forms—alternating, symmetric, and hermitian—correspond to the three types of classical groups of isometries: symplectic, orthogonal, and unitary. But this is not quite the whole story because it does not include orthogonal groups over fields of characteristic two. In order to include these groups it is necessary to consider quadratic forms in addition to symmetric bilinear forms.

### 30.2.1 Quadratic Forms

If $\beta$ is a bilinear form, a *quadratic form* with *polar form* $\beta$ is a function $Q : V \to K$ such that

$$Q(av) = a^2 Q(v)$$
and
$$\beta(u,v) = Q(u+v) - Q(u) - Q(v)$$

for all $u, v \in V$ and all $a \in K$. We have $\beta(v,v) = 2Q(v)$ and therefore, if the characteristic of $K$ is not two, $\beta$ determines $Q$.

We extend the notion of polar space to include vector spaces $V$ with an associated quadratic form $Q$. The pair $(V, Q)$ is an orthogonal geometry and $V$ is a *quadratic* space.

## 30.3 Inner Products

Every vector space $V$ in Magma created via the `VectorSpace` intrinsic (or its synonym `KSpace`) has an associated bilinear form which is represented by a matrix and which can be accessed via `InnerProductMatrix(V)` or via the attribute `ip_form`. By default the inner product matrix is the identity. If the dimension of $V$ is $n$, then any $n \times n$ matrix defined over the base field of $V$ can serve as the inner product matrix by passing it to `VectorSpace` as an additional parameter.

If $e_1$, $e_2$, ..., $e_n$ is a basis for $V$, the matrix of the form $\beta$ with respect to this basis is $J := \big(\beta(e_i, e_j)\big)$.

**Example: Vector Space with a General Form** `(ex-2f3d6e)`

```
> K := GF(11);
> J := Matrix(K,3,3,[1,2,3, 4,5,6, 7,8,9]);
> V := VectorSpace(K,3,J);
> InnerProductMatrix(V);
[ 1  2  3]
[ 4  5  6]
[ 7  8  9]
```

A vector space $V$ may also have an associated quadratic form. This can be assigned by attaching a matrix to $V$ via the function `QuadraticSpace` described in Section *Quadratic Spaces*. If assigned, the matrix can be accessed as the return value of `QuadraticFormMatrix`.

The quadratic form defined by the matrix $A$ is $q(v) = vAv^{\mathrm{tr}}$, and the matrix of its polar form is $A + A^{\mathrm{tr}}$. It is always possible to represent a quadratic form by an upper triangular matrix. Furthermore, if the characteristic of the field is not two, the quadratic form may be represented by the symmetric matrix $\frac{1}{2}J$, where $J$ is the matrix of its polar form.

In order to accommodate hermitian forms, a vector space of type `ModTupFld` has an attribute `Involution`. This attribute is intended to hold an automorphism (of order two) of the base field.

**`EnsureUpperTriangular(A): AlgMatElt -> AlgMatElt`**

Given an $n \times n$ matrix $A$ this function returns the upper triangular matrix $Q$ which represents the same quadratic form as $A$. That is, for all $n$-tuples $v$ we have $vAv^{\mathrm{tr}} = vQv^{\mathrm{tr}}$.

**`DotProduct(u, v): ModTupFldElt, ModTupFldElt -> FldElt`**

If $V$ is the generic space of the parent of $u$ and $v$, let $\sigma$ be the field automorphism `V`Involution` if this attribute is assigned or the identity automorphism if `V`Involution` is not assigned. If $J$ is the inner product matrix of $V$, the expression `DotProduct(u,v)` evaluates to $uJ\sigma(v^{\mathrm{tr}})$. That is, it returns $\beta(u, v)$, where $\beta$ is a bilinear or sesquilinear form on $V$.

**`DotProductMatrix(S): SeqEnum[ModTupFldElt] -> AlgMatElt`**

The matrix of inner products of the vectors in the sequence $S$. The inner products are calculated using `DotProduct` and therefore take into account any field automorphism attached to the `Involution` attribute of the generic space of the universe of $S$.

**`GramMatrix(V): ModTupRng -> AlgMatElt`**

If $B$ is the basis matrix of $V$ and if $J$ is the inner product matrix, this function returns $BJB^{\mathrm{tr}}$. In this case the `Involution` attribute is ignored.

**`InnerProductMatrix(V): ModTupRng -> AlgMatElt`**

The inner product matrix attached to the generic space of $V$. This is the attribute `V`ip_form`.

**Example: Gram Matrix Versus Inner Product Matrix** (ex-9d71da)

This example illustrates the difference between `GramMatrix` and `InnerProductMatrix`. The function `GramMatrix` uses the echelonised basis of the subspace $W$. To obtain the matrix of inner products between a given list of vectors, use `DotProductMatrix`.

```
> K<a> := QuadraticField(-2);
> J := Matrix(K,3,3,[1,2,1, 2,1,0, 1,0,2]);
> V := VectorSpace(K,3,J);
> W := sub<V| [a,a,a], [1,2,3]>;
> InnerProductMatrix(W);
[1 2 1]
[2 1 0]
[1 0 2]
> GramMatrix(W);
[1 0]
[0 9]
> DotProductMatrix([W.1,W.2]);
[ -20 19*a]
[19*a   37]
```

**Example: Symmetric and Hermitian Products** (ex-af15e4)

Continuing the previous example, the vector space $V$ does not have the attribute `Involution` assigned and therefore `DotProduct` uses the *symmetric* bilinear form represented by the inner product matrix $J$. However, the field $K$ has a well-defined operation of complex conjugation and so `InnerProduct` uses the *hermitian* form represented by $J$.

```
> u := W.1+W.2;
> DotProduct(u,u);
38*a + 17
```

(continues on next page)

(continued from previous page)

```
> InnerProduct(u,u);
57
```

### 30.3.1 Orthogonality

If $\beta$ is any bilinear or sesquilinear form, the vectors $u$ and $v$ are *orthogonal* if $\beta(u, v) = 0$. The *left orthogonal complement* of a subset $X$ of $V$ is the subspace

$$^{\perp}X := \{ u \in V \mid \beta(u, x) = 0 \text{ for all } x \in X \}$$

and the *right orthogonal complement* of $W$ is

$$X^{\perp} := \{ u \in V \mid \beta(x, u) = 0 \text{ for all } x \in X \}.$$

If $\beta$ is reflexive, then $^{\perp}X = X^{\perp}$.

**OrthogonalComplement(V, X : parameters): ModTupFld, ModTupFld -> ModTupFld**

| Right: BoolElt | Default: false |
|---|---|

The default value is the left orthogonal complement of $X$ in $V$. To obtain the right orthogonal complement set `Right` to `true`.

**Radical(V : parameters): ModTupFld -> ModTupFld**

| Right: BoolElt | Default: false |
|---|---|

The left radical of the inner product space $V$, namely $^{\perp}V$. To obtain the right radical set `Right` to `true`.

A bilinear or sesquilinear form $\beta$ is *non-degenerate* if $\text{rad}(V) = 0$, where $V$ is the polar space of $\beta$.

**IsNondegenerate(V): ModTupFld -> BoolElt**

Returns `true` if the determinant of the matrix of inner products of the basis vectors of $V$ is non-zero, otherwise `false`. This function takes into account the field automorphism, if any, attached to the `Involution` attribute of the generic space of $V$.

**IsDegenerate(V): ModTupFld -> BoolElt**

The opposite of the above.

If $V$ is a quadratic space over a perfect field of characteristic 2, the restriction of the quadratic form $Q$ to the radical is a semilinear functional (with respect to $x \mapsto x^2$) whose kernel is the *singular radical* of $V$. A quadratic space is *non-singular* if its singular radical is zero.

**SingularRadical(V): ModTupFld -> ModTupFld**

The kernel of the restriction of the quadratic form of the quadratic space $V$ to the radical of $V$.

**IsNonsingular(V): ModTupFld -> BoolElt**

Returns `true` if $V$ is a non-singular quadratic space, otherwise `false`.

## 30.4 Isotropic and Singular Vectors and Subspaces

Let $\beta$ be a reflexive bilinear or a sesquilinear form on the vector space $V$. A non-zero vector $v$ is *isotropic* (with respect to $\beta$) if $\beta(v, v) = 0$. If $Q$ is a quadratic form, a non-zero vector $v$ is *singular* if $Q(v) = 0$.

**HasIsotropicVector(V): ModTupFld -> BoolElt, ModTupFldElt**

Determine whether the polar space $V$ contains an isotropic vector; if it does, the second return value is a representative.

`HasSingularVector(V): ModTupFld -> BoolElt, ModTupFldElt`

Determine whether the quadratic space $V$ contains a singular vector; if it does, the second return value is a representative.

A subspace $W$ of a polar space is *totally isotropic* if every non-zero vector of $W$ is isotropic.

`IsTotallyIsotropic(V): ModTupFld -> BoolElt`

Returns `true` if the polar space $V$ is totally isotropic, otherwise `false`.

A subspace $W$ of a quadratic space defined by a quadratic form $Q$ is *totally singular* if $Q(w) = 0$ for all $w \in W$.

`IsTotallySingular(V): ModTupFld -> BoolElt`

Returns `true` if the quadratic space $V$ is totally singular, otherwise `false`.

`MaximalTotallyIsotropicSubspace(V): ModTupFld -> ModTupFld`

A representative maximal totally isotropic subspace of the polar space $V$.

`MaximalTotallySingularSubspace(V): ModTupFld -> ModTupFld`

A representative maximal totally singular subspace of the quadratic space $V$.

The *Witt index* of a polar space $V$ that is not a quadratic space is the dimension of a maximal totally isotropic space. The *Witt index* of a quadratic space is the dimension of a maximal totally singular subspace.

If the characteristic of the field is not 2 and if $\beta$ is the polar form of $Q$, a subspace is totally singular if and only if it is totally isotropic with respect to $\beta$; in this case the Witt index of $Q$ coincides with the Witt index of $\beta$.

`WittIndex(V): ModTupFld -> RngIntElt`

The Witt index of the polar space $V$.

An ordered pair of vectors $(u, v)$ such that $u$ and $v$ are isotropic and $\beta(u, v) = 1$ is a *hyperbolic pair*. If $V$ is a quadratic space, $u$ and $v$ are required to be singular. The subspace spanned by a hyperbolic pair is a *hyperbolic plane*.

If $V$ is a pseudo-symplectic space defined by a symmetric bilinear form $\beta$ over a finite field of characteristic 2, define the *pseudo-radical* of $V$ to be the radical of the hyperplane $\{v \in V \mid \beta(v, v) = 0\}$.

`HyperbolicPair(V, u): ModTupFld, ModTupFldElt -> ModTupFldElt`

Given a singular or isotropic vector $u$ which is not in the radical or pseudo-radical, return a vector $v$ such that $(u, v)$ is a hyperbolic pair.

**Example: Isotropic Vectors in a Pseudo-Symplectic Space** (ex-ced2e7)

The vector space of dimension 2 over $\mathbf{F}_2$ is pseudo-symplectic (the form is the identity matrix). It has three non-zero elements only one of which is isotropic. This confirms that not every isotropic vector in a non-degenerate pseudo-symplectic space belongs to a hyperbolic pair.

```
> V := VectorSpace(GF(2),2);
> IsPseudoSymplecticSpace(V);
true
> IsNondegenerate(V);
true
> { v : v in V | v ne V!0 and DotProduct(v,v) eq 0};
{
    (1 1)
}
```

A polar space $V$ has a *hyperbolic splitting*; namely, a direct sum decomposition

$$V = L_1 \perp L_2 \perp \cdots \perp L_m \perp W$$

where the $L_i$ are *hyperbolic planes* and $m$ is maximal.

The polar space is *hyperbolic* if $W = 0$; i.e., it is an orthogonal sum of hyperbolic planes. In Bourbaki [Bou07, p. 66] the corresponding form is said to be *neutral*.

If the form defining the polar space is non-degenerate and not pseudo-alternating, then every isotropic (resp. singular) vector belongs to a hyperbolic pair. Therefore if the charactersistic is not 2, $W$ is *anisotropic*; i.e., it does not contain any isotropic vectors. In this case the integer $m$ is the *Witt index* of the form and $W$ is called the *anisotropic component* of the splitting.

If the characteristic is 2 and $V$ is a quadratic space, $W$ does not contain singular vectors but it may contain isotropic vectors.

**Example: Hyperbolic Splitting in Characteristic 2** (ex-b0692a)

The last term of a hyperbolic splitting of a quadratic space in characteristic 2 can contain an isotropic vector.

```
> Q := StandardQuadraticForm(6,4 : Minus);
> V := QuadraticSpace(Q);
> WittIndex(V);
2
> H := HyperbolicSplitting(V);
> W := sub< V | H[2] >;
> HasSingularVector(W);
false
> HasIsotropicVector(W);
true (    0     0     1     0     0     0)
```

`HyperbolicSplitting(V): ModTupFld -> Tup`

A pair $(M, B)$, where $M$ is a maximal list of pairwise orthogonal hyperbolic pairs and $B$ is a basis for the orthogonal complement of the subspace they span. This function requires the form to be non-degenerate and, except for symplectic spaces, the base ring of $V$ must be a finite field.

**Example: Hyperbolic Splitting of a Symmetric Form** (ex-7e8c7f)

Find the hyperbolic splitting of a polar space defined by a symmetric bilinear form. In this example $W$ is a non-degenerate subspace of the polar space $V$.

```
> K<a> := GF(7,2);
> J := Matrix(K,3,3,[1,2,1, 2,1,0, 1,0,2]);
> V := VectorSpace(K,3,J);
> W := sub<V| [a,a,a], [1,2,3]>;
> IsNondegenerate(W);
true
> HyperbolicSplitting(W);
<[
    [
        (a^20    1 a^39),
        (a^12    2    a)
    ]
], []>
```

**Example: Splitting Off the Radical** (ex-621e11)

The polar space $V$ of the previous example is degenerate and so `HyperbolicSplitting` cannot be applied directly. Instead, we first split off the radical.

```
> IsNondegenerate(V);
false
> R := Radical(V);
> H := (Dimension(R) eq 0) select V else
>   sub<V|[e : e in ExtendBasis(B,V) | e notin B] where B is Basis(R)>;
> HyperbolicSplitting(H);
```

(continues on next page)

(continued from previous page)

```
<[
    [
        (   0 a^20    1),
        (   0 a^12    2)
    ]
], []>
```

A non-degenerate polar space $V$ of dimension $2m$ which is the direct sum of two totally isotropic subspaces is hyperbolic and it has a *symplectic basis*; i.e., a basis $e_1$, $f_1$, ..., $e_m$, $f_m$ such that the pairs $(e_i, f_i)$, $1 \leq i \leq m$ are mutually orthogonal hyperbolic pairs.

**SymplecticBasis(V, U, W): ModTupFld, ModTupFld, ModTupFld -> [ModTupFldElt]**

Given totally isotropic subspaces $U$ and $W$ of a non-degnerate polar space $V$ such that $V = U \oplus W$, return a symplectic basis for $V$ such that $e_1$, $e_2$, ..., $e_m$ is a basis for $U$ and $f_1$, $f_2$, ..., $f_m$ is a basis for $W$.

Let $V = L_1 \perp \cdots \perp L_m \perp W \perp \mathrm{rad}(V)$ be a hyperbolic splitting of the polar space $V$ where the $L_i$ are hyperbolic planes spanned by hyperbolic pairs $(e_i, f_i)$ for $1 \leq i \leq m$. The subspaces $P = \langle e_1, \ldots, e_m \rangle$ and $N = \langle f_1, \ldots, f_m \rangle$ are totally isotropic (resp. totally singular) and we call the 4-tuple $(\mathrm{rad}(V), P, N, W)$ a *Witt decomposition* of $V$.

**WittDecomposition(V): ModTupFld -> SeqEnum[ModTupFld]**

The Witt decomposition of the space $V$.

**WittDecomposition(M, a): AlgMatElt[FldFin], FldAut -> AlgMatElt[FldFin], AlgMatElt[FldFin]**

Given a field automorphism $a$, and a matrix $M$ which is hermitian with respect to $a$, returns the Gram matrix with respect to a basis of a Witt decomposition of the polar space of $M$, in the order $(P, N, W, \mathrm{rad}(V))$. Also returns the basis matrix.

A quadratic space is *metabolic* if it is a direct sum $E \oplus F$ of totally singular subspaces $E$ and $F$ such that $E = E^\perp$.

Given a quadratic space with quadratic form $q : V \to F$, the *metabolic space* based on $V$ is the quadratic space $M = V \oplus V^*$ with quadratic form $Q : M \to F$ defined by $Q(v, f) = q(v) + vf$.

**MetabolicSpace(V): ModTupFld -> ModTupFld**

The metabolic space based on the quadratic space $V$.

## 30.5 The Standard Forms

This section describes the "standard" alternating, hermitian, quadratic, and symmetric forms defined on a finite dimensional vector space over a field. These are forms of maximal Witt index together with the quadratic forms of non-maximal Witt index over finite fields (see Section *Isotropic and Singular Vectors and Subspaces*). The standard forms are preserved by the Magma implementation of the classical groups over finite fields.

If $J$ is the matrix of a bilinear form, $X$ preserves the form if $XJX^{\mathrm{tr}} = J$. If $J$ represents an hermitian form and if $\overline{X}$ is obtained from $X$ by application of the field automorphism, then $X$ preserves the form if $XJ\overline{X}^{\mathrm{tr}} = J$.

If $\beta$ is a non-degenerate alternating form, then $\mathrm{rad}(V)$ and the anisotropic component of a hyperbolic splitting are zero. Thus the dimension of $V$ must be even and $V$ has a basis of mutually orthogonal hyperbolic pairs. In particular, up to equivalence, there is only one non-degenerate alternating form on $V$.

**StandardAlternatingForm(n, R): RngIntElt, Rng -> AlgMatElt**

**StandardAlternatingForm(n, q): RngIntElt, RngIntElt -> AlgMatElt**

If $n = 2m$, this function returns the $n \times n$ matrix of a non-degenerate alternating form over the

ring $R$ (or the field of $q$ elements) such that if $e_1$, $e_2$, ..., $e_{2m}$ is the standard basis, then $(e_1, e_{2m})$, $(e_2, e_{2m-1})$, ..., $(e_m, e_{m+1})$ are mutually orthogonal hyperbolic pairs.

The group of isometries of this form is the symplectic group $\mathrm{Sp}(2m, R)$.

**Example: Standard Alternating Form** (ex-3c60cd)

Create a symplectic geometry with the standard alternating form and then check that every non-zero vector is isotropic.

```
> K := GF(5);
> J := StandardAlternatingForm(4,K);
> J;
[0 0 0 1]
[0 0 1 0]
[0 4 0 0]
[4 0 0 0]
> V := VectorSpace(K,4,J);
> forall{ v : v in V | DotProduct(v,v) eq 0 };
true
```

**StandardPseudoAlternatingForm(n, K): RngIntElt, Fld -> AlgMatElt**

**StandardPseudoAlternatingForm(n, q): RngIntElt, RngIntElt -> AlgMatElt**

The matrix of the standard pseudo-alternating form of degree $n$ over the field $K$ (or the finite field of order $q$), which must have characteristic 2; that is, a symmetric form which is not alternating.

**StandardHermitianForm(n, K): RngIntElt, Fld -> AlgMatElt, Map**

**StandardHermitianForm(n, q): RngIntElt, RngIntElt -> AlgMatElt, Map**

The first return value of this function is the $n \times n$ anti-diagonal matrix $(\delta_{i,n-i+1})$ over the field $K$ (or the field of $q^2$ elements). If $K$ is the finite field of $q^2$ elements, the second return value is the field involution $K \to K : x \mapsto x^q$. If $K$ is a field which admits the operation of complex conjugation, the second return value is the field automorphism which sends each element to its complex conjugate.

If $\beta$ is a non-degenerate hermitian form over a finite field, then $\mathrm{rad}(V)$ is zero and the dimension of the anisotropic component of a hyperbolic splitting is either 1 or 0.

In the finite field case, the group of isometries of this form is $\mathrm{GU}(n, q)$.

**StandardQuadraticForm(n, K : parameters): RngIntElt, Fld -> AlgMatElt**

**StandardQuadraticForm(n, q : parameters): RngIntElt, RngIntElt -> AlgMatElt**

| | | |
|---|---|---|
| Minus : | BoolElt | Default: false |
| Variant: | MonStgElt | Default: "Default" |

An $n \times n$ upper triangular matrix representing a quadratic form over the field $K$ (or the field of order $q$). The default option is to return a form of maximal Witt index, namely the upper triangular matrix whose non-zero entries are $\delta_{i,n-i+1}$, where $1 \leq i \leq (n+1)/2$.

If `Minus` is `true` and $n = 2m$, this function returns a form whose Witt index is $m - 1$. If `Minus` is `true`, $n = 2m + 1$ and $K$ is a finite field of odd characteristic, the return value is a form of Witt index $m$ which is similar to but not congruent to the default form. Use the `Revised` option of `Variant` to obtain the form preserved by the orthogonal groups `AltGOMinus(n,q)` and subgroups.

If $K$ is a finite field of order $q$ and $W$ is the anisotropic component of a hyperbolic splitting of a form of even rank and `Minus` type, then $W$ has basis vectors $e$ and $f$ such that $Q(e) = -1$, $\beta(e, f) = -\xi - \xi^q$, and $Q(f) = -\xi\xi^q$, where $\xi$ is a primitive element of $\mathbf{F}_{q^2}$. If the characteristic of $K$ is odd, the `Revised` option returns a form corresponding to an orthonormal basis for $W$.

**Example: Standard Quadratic Form of Minus Type** (ex-78f890)

Construct a standard quadratic form of minus type.

```
> K<z> := GF(7,2);
> Q := StandardQuadraticForm(4,49 : Minus);
> Q;
[   0    0     0    1]
[   0    6     5    0]
[   0    0  z^29    0]
[   0    0     0    0]
> _<x> := PolynomialRing(K);
> a := Q[2,2] * Q[3,3];
> IsIrreducible(x^2+x+a);
true
```

**Example: Revised Minus-Type Quadratic Form** (ex-29b8f5)

Compare the revised form with the standard form: the forms `Q` above and `QR` below have different entries in the central $2 \times 2$ block.

```
> QR := StandardQuadraticForm(4,49 : Minus, Variant := "Revised");
> QR;
[   0    0     0    1]
[   0    4     0    0]
[   0    0  z^11    0]
[   0    0     0    0]
```

**StandardSymmetricForm(n, K): RngIntElt, Fld -> AlgMatElt**

**StandardSymmetricForm(n, q : parameters): RngIntElt, RngIntElt -> AlgMatElt**

```
Minus   : BoolElt                       Default: false
Variant: MonStgElt                      Default: "Default"
```

In all cases this is $Q + Q^{\mathrm{tr}}$, where $Q$ is the corresponding standard quadratic form, as defined above.

## 30.6 Constructing Polar Spaces

If $J$ is an $n \times n$ matrix, the command `VectorSpace(K,n,J)` creates a vector space of dimension $n$ over $K$ with a bilinear form whose matrix is $J$.

The default form attached to every vector space in Magma is the symmetric form whose matrix is the identity matrix.

A vector space $V$ is recognised as a polar space if any of the following conditions apply. (There is no check to ensure that the inner product matrix is non-degenerate.)

1.There is a quadratic form attached to $V$.

2.There is a field involution attached to $V$ and the inner product matrix of $V$ is hermitian or skew-hermitian with respect to this involution.

3.The inner product matrix of $V$ is symmetric or alternating.

Thus a vector space with a symmetric inner product matrix but no quadratic form attached is a polar space. If the characteristic of the field is 2 and the form is not alternating it is a pseudo-symplectic space, otherwise we shall call it an orthogonal space to distinguish it from quadratic spaces.

**PolarSpace(F, a): AlgMatElt[FldFin], FldAut -> ModTupFld[FldFin]**

```
Symbolic: BoolElt                       Default: true
```

The polar space associated to the Gram matrix $F$, hermitian with repsect to the field automorphism $a$. `Symbolic` determines the method of finding isotropic subspaces.

`TrivialPolarSpace(F, n): FldFin, RngIntElt -> ModTupFld[FldFin]`

Polar space over $F$ of dimension $n$ with the zero bilinear form.

`IsPolarSpace(V): ModTupFld -> BoolElt`

Check if the vector space $V$ is a quadratic space or if the Gram matrix of $V$ is a reflexive form.

`PolarSpaceType(V): ModTupFld -> MonStgElt`

The type of the polar space $V$, returned as a string.

**Example: Testing for a Polar Space** `(ex-1d3f36)`

Create the standard vector space of dimension 4 over the rational field and check if it is a polar space.

```
> V := VectorSpace(Rationals(),4);
> IsPolarSpace(V);
true
> PolarSpaceType(V);
orthogonal space
```

### 30.6.1 Isotropic Subspaces

`FirstIsotropicSubspace(V, k): ModTupFld[FldFin], RngIntElt -> ModTupFld[FldFin]`

Starts an enumeration process of $k$-dimensional isotropic subspaces in $V$, and returns the first non-singular one.

`NextIsotropicSubspace(V, k): ModTupFld[FldFin], RngIntElt -> ModTupFld[FldFin]`

The next $k$-dimensional isotropic subspace in $V$.

`AllIsotropicSubspaces(V, k): ModTupFld[FldFin], RngIntElt -> [ ModTupFld[FldFin] ]`

All $k$-dimensional isotropic subspaces in $V$.

`NumberOfIsotropicSubspaces(V, k): ModTupFld[FldFin], RngIntElt -> RngIntElt`

The number of $k$-dimensional isotropic subspaces in $V$.

### 30.6.2 Symplectic Spaces

`SymplecticSpace(J): AlgMatElt -> ModTupRng`

The symplectic space of dimension $n$ defined by the $n \times n$ matrix $J$. This function checks to ensure that $J$ is alternating.

The symplectic inner product on this space is obtained from `InnerProduct`, which evaluates the alternating form $J$ and returns a value in the base ring. This is distinct from the intrinsic `SymplecticInnerProduct` described in Chapter Quantum Codes: that intrinsic computes the coding-theory product $(a|b) * (c|d) = ad - bc$ in compact or extended format and returns its field trace down to the prime field (see that chapter for the precise source field of the trace in each format).

`IsSymplecticSpace(W): ModTupFld -> BoolElt`

Returns `true` if the `Involution` attribute of the generic vector space $W$ is not assigned and the space carries an alternating form, otherwise `false`.

Note that a quadratic space over a field of characteristic 2 satisfies these conditions and consequently this function will return `true` for these spaces.

`IsPseudoSymplecticSpace(W): ModTupFld -> BoolElt`

Given a vector space $W$ over a finite field, this intrinsic returns `true` if the base field has characteristic 2, the `Involution` attribute is not assigned to the generic space and the form is symmetric but not alternating, otherwise `false`.

`DirectSum(V, W): ModTupRng, ModTupRng -> ModTupRng, Map, Map`

If $V$ and $W$ are symplectic spaces this function returns the orthogonal direct sum $V \perp W$ and linear embeddings $V \to V \perp W$ and $W \to V \perp W$.

### 30.6.3 Unitary Spaces

In order to accommodate hermitian forms it is necessary to assign a field automorphism of order two to the `Involution` attribute of the vector space.

Thus a unitary space is characterised as a vector space $V$ whose ambient space, `Generic(V)`, has the attribute `Involution` and whose inner product matrix is either hermitian or skew hermitian.

`UnitarySpace(J, sigma): AlgMatElt, Map -> ModTupFld`

The $n$-dimensional unitary space over the base field $K$ of $J$, where $\sigma$ is an automorphism of $K$ of order 2 and where $J$ is an $n \times n$ matrix which is hermitian or skew-hermitian with respect to $\sigma$.

`IsUnitarySpace(W): ModTupFld -> BoolElt, RngIntElt`

Return `true` if the `Involution` attribute of the generic space of $W$ is assigned and the form is either hermitian or skew-hermitian when restricted to $W$. The second return value is 1 if the form is hermitian and $-1$ if the form is skew-hermitian.

**Example: Isotropic Vector in a Unitary Space** (ex-b9c1f1)

Create a unitary geometry with the standard hermitian form and check that the given vector is isotropic. Note that the function `DotProduct` takes both the form and the field involution into account when calculating its values. For finite fields, the function `InnerProduct` *ignores* the field involution.

```
> K<z> := GF(25);
> J, sigma := StandardHermitianForm(5,K);
> J;
[   0    0    0    0    1]
[   0    0    0    1    0]
[   0    0    1    0    0]
[   0    1    0    0    0]
[   1    0    0    0    0]
> sigma(z);
z^5
> V := UnitarySpace(J,sigma);
> u := V![1,z,0,z^2,-1];
> DotProduct(u,u);
0
> InnerProduct(u,u);
z^20
```

`ConjugateTranspose(M, sigma): Mtrx, Map -> Mtrx`

The transpose of the matrix $\sigma(M)$, where $\sigma$ is an automorphism of the base field of the matrix $M$.

### 30.6.4 Quadratic Spaces

A vector space $V$ with an attached quadratic form is called a *quadratic space*. The polar form of a quadratic space is the inner product matrix $J$ of the space. If the characteristic of the field is not 2, the value of the quadratic form on a row vector $v$ is $\frac{1}{2} v * J * v^{\mathrm{tr}}$.

`QuadraticSpace(Q): AlgMatElt -> ModTupRng`

The quadratic space of dimension $n$ defined by the quadratic form represented by an $n \times n$ matrix $Q$. The inner product matrix of the space is $Q + Q^{\mathrm{tr}}$.

`QuadraticSpace(f): RngMPolElt -> ModTupRng`

The quadratic space of dimension $n$ whose quadratic form is given by the quadratic polynomial $f$

in $n$ variables. If the variables are $x_1$, ..., $x_n$, then for $i \leq j$, the $(i, j)$-th entry of the matrix of the form is the coefficient of $x_i x_j$ in $f$.

`SymmetricToQuadraticForm(J): AlgMatElt -> AlgMatElt`

Provided the characteristic of the field is not two, this is the upper triangular matrix which represents the quadratic form whose polar form is the symmetric matrix $J$.

`QuadraticFormMatrix(V): ModTupRng -> ModAlgElt`

The matrix which defines the quadratic form of the quadratic space $V$.

`QuadraticNorm(v): ModTupFldElt -> FldElt`

The value $Q(v)$, where $Q$ is the quadratic form attached to the generic space of the parent of the vector $v$.

`QuadraticFormPolynomial(V): ModTupRng -> RngPolElt`

The polynomial $\sum_{i,j} q_{ij} x_i x_j$, where $Q = (q_{ij})$ is the matrix of the quadratic form of the quadratic space $V$.

`QuadraticFormPolynomial(Q): AlgMatElt -> RngPolElt`

The polynomial $\sum_{i,j} q_{ij} x_i x_j$, where $Q = (q_{ij})$ is a matrix representing a quadratic form.

**Example: Quadratic Space from a Polynomial** (ex-474785)

The quadratic space defined by a polynomial.

```
> _<x,y,z> := PolynomialRing(Rationals(),3);
> f := x^2 + x*y +3*x*z - 2*y*z + y^2 +z^2;
> V := QuadraticSpace(f);
> PolarSpaceType(V);
quadratic space
> IsNonsingular(V);
true
> QuadraticFormMatrix(V);
[ 1  1  3]
[ 0  1 -2]
[ 0  0  1]
```

`OrthogonalSum(V, W): ModTupFld, ModTupFld -> ModTupFld, Map, Map`

The orthogonal direct sum $V$ of the quadratic spaces $V$ and $W$ together with linear embeddings $V \to V \perp W$ and $W \to V \perp W$.

`OrthogonalTensorProduct(V, W): ModTupFld, ModTupFld -> ModTupFld`

The tensor product of the quadratic spaces $V$ and $W$. If $\beta$ and $\gamma$ are the symmetric bilinear forms on $V$ and $W$, respectively, and if $\xi$ is the bilinear form on the tensor product, then $\xi(v_1 \otimes w_1, v_2 \otimes w_2) = \beta(v_1, v_2)\gamma(w_1, w_2)$.

`TotallySingularComplement(V, U, W): ModTupFld, ModTupFld, ModTupFld -> ModTupFld`

Given totally singular subspaces $U$ and $W$ of the quadratic space $V$ such that $U^\perp \cap W = 0$ this function returns a totally singular subspace $X$ such that $V = X \oplus U^\perp$ and $W \subseteq X$.

`Discriminant(V): ModTupFld -> RngIntElt`

If $V$ is a vector space over the finite field $K$ and if $J$ is the Gram matrix of $V$, the *discriminant* of $V$ is the determinant $\Delta$ of $J$ modulo squares. That is, the discriminant is 0 if $\Delta$ is a square in $K$, 1 if it is a non-square. The form $J$ is required to be non-degenerate.

`ArfInvariant(V): ModTupFld -> RngIntElt`

The Arf invariant of the quadratic space $V$.

Currently this is available only for quadratic spaces of even dimension $2m$ over a finite field $F$ of characteristic 2. In this case there are two possibilities: either the Witt index of the form is $m$ and the Arf invariant is 0, or the Witt index is $m - 1$ and the Arf invariant is 1.

**DicksonInvariant(V, f): ModTupFld, Mtrx -> RngIntElt**

The Dickson invariant of the isometry $f$ of the quadratic space $V$ is the rank (mod 2) of $1 - f$. If the polar form of $Q$ is non-degenerate, the Dickson invariant defines a homomorphism from the orthogonal group $O(V)$ onto the additive group of order 2.

**SpinorNorm(V, f): ModTupFld, Mtrx -> RngIntElt**

The spinor norm of the isometry $f$ of the quadratic space $V$. This is the discriminant of the Wall form (Section *Wall Forms*) of $f$.

**SpinorNorm(Q, g): AlgMatElt[Fld], AlgMatElt[Fld] -> FldElt**

Given a symmetric matrix $Q$ over a field $F$, and a matrix $g$ satisfying $gQg^{\mathrm{tr}} = Q$, returns an element $x \in F$ representing in $F/F^{\times 2}$ the spinor norm of the isometry $g$ with repsect to the quadratic form represented by $Q$.

**Example: Spinor Norm over a General Field** (ex-ff51ba)

An example for a spinor norm computation over a general field.

```
> Q := SymmetricMatrix(Rationals(), [2,0,2,0,0,2,1,0,0,2,0,0,1,1,52]);
> g := Matrix(Rationals(), 5, 5,
>                     [1,0,0,0,0,0,1,0,0,0,0,0,1,0,0,1,0,0,-1,0,0,0,1,0,-1]);
> g * Q * Transpose(g) eq Q;
true
> SpinorNorm(Q,g);
2745
> FactorizationOfQuotient(SpinorNorm(Q, g));
[ <3, 2>, <5, 1>, <61, 1> ]
```

**CharacterQQModSquares(d, r): RngIntElt, FldRatElt -> RngIntElt**

Let $\chi_d : \mathbb{Q}^\times/\mathbb{Q}^{\times 2} \to \mathbb{Z}^\times$ be the multiplicative character defined by $\chi_d(p) = -1$ iff $p \mid d$. Returns $\chi_d(r)$.

**SpinorNormRho(d, g, Q): RngIntElt, GrpMatElt, AlgMatElt -> RngIntElt**

Let $Q$ be an integral symmetric matrix, $g$ an isometry of the quadratic space, and $d \mid \mathrm{Disc}(Q)$. Returns the value of $\chi_d$ (as in **CharacterQQModSquares**) at the spinor norm of $g$ with respect to $\mathrm{O}(Q)$.

**HyperbolicBasis(U, B, W): ModTupFld, SeqEnum, ModTupFld -> SeqEnum**

Given complementary totally singular subspaces $U$ and $W$ of a quadratic space and a basis $B$ for $U$, return a sequence of pairwise orthogonal hyperbolic pairs whose second components form a basis for $W$.

**OrthogonalReflection(a): ModTupFldElt -> AlgMatElt**

The reflection determined by a non-singular vector of a quadratic space.

**RootSequence(V, f): ModTupFld, Mtrx -> SeqEnum**

Given a matrix $f$ representing an isometry of the quadratic space $V$, return a sequence of vectors such that the product of the corresponding orthogonal reflections is $f$. The empty sequence is returned if $f$ is the identity matrix.

**ReflectionFactors(V, f): ModTupFld, Mtrx -> SeqEnum**

Given a matrix $f$ representing an isometry of the quadratic space $V$, return a sequence of reflections whose product is $f$. The empty sequence corresponds to the identity matrix.

Given a quadratic space $V$ defined by a quadratic form $Q$ with polar form $\beta$ and non-zero vectors $u, v \in V$ such that $u$ is singular and $\beta(u, v) = 0$, the *Siegel transformation* (also called an Eichler transformation) is the isometry $\rho_{u,v}$ defined by

$$x\rho_{u,v} = x + \beta(x, v)u - \beta(x, u)v - Q(v)\beta(x, u)u.$$

**SiegelTransformation(u, v): ModTupFldElt, ModTupFldElt -> AlgMatElt**

The Siegel transformation defined by a singular vector $u$ and a vector $v$ orthogonal to $u$. The common parent of $u$ and $v$ must be a quadratic space.

**Example: Isometries from Siegel Transformations** (ex-717e94)

A group of isometries generated by Siegel transformations.

```
> Q := StandardQuadraticForm(4,3);
> V := QuadraticSpace(Q);
> u := V.1;
> QuadraticNorm(u);
0
> X := { v : v in V | DotProduct(u,v) eq 0 and QuadraticNorm(v) ne 0 };
> #X;
12
> H := sub< GL(V) | [SiegelTransformation(u,v) : v in X]>;
> #H;
9
```

## 30.7 Isometries and Similarities

If $\beta$ is a bilinear or sesquilinear form on the vector space $V$, a linear transformation $g$ of $V$ is an *isometry* if $g$ preserves $\beta$; it is a *similarity* if it preserves $\beta$ up to a non-zero scalar multiple.

### 30.7.1 Isometries

**IsIsometry(U, V, f): ModTupFld, ModTupFld, Map -> BoolElt**

Returns **true** if the map $f$ is an isometry from $U$ to $V$ with respect to the attached forms.

**IsIsometry(f): Map -> BoolElt**

Returns **true** if the map $f$ is an isometry from its domain to its codomain.

**IsIsometry(V, g): ModTupFld, Mtrx -> BoolElt**

Returns **true** if the matrix $g$ is an isometry of $V$ with respect to the attached form.

**IsIsometric(V, W): ModTupFld, ModTupFld -> BoolElt, Map**

Determines whether the polar spaces $V$ and $W$ are isometric; if they are, an isometry is returned (as a map).

**Example: Isometric Polar Spaces** (ex-9909e6)

A vector space is always equipped with a bilinear form and the default value is the identity matrix. However the "standard" symmetric form is $Q + Q^{\mathrm{tr}}$, where $Q$ is the standard quadratic form. In the following example the polar spaces defined by these forms are similar but not isometric.

```
> F := GF(5);
> V1 := VectorSpace(F,5);
> PolarSpaceType(V1);
orthogonal space
> WittIndex(V1);
2
> J2 := StandardSymmetricForm(5,F);
> J2;
[0 0 0 0 1]
[0 0 0 1 0]
[0 0 2 0 0]
[0 1 0 0 0]
[1 0 0 0 0]
```

(continues on next page)

(continued from previous page)

```
> V2 := VectorSpace(F,5,J2);
> IsIsometric(V1,V2);
false
> V3 := VectorSpace(F,5,2*J2);
> flag, f := IsIsometric(V1,V3); flag;
true
> IsIsometry(f);
true
```

If $J$ is an $n \times n$ matrix which represents a bilinear form and if $M$ is a non-singular $n \times n$ matrix, then $J$ and $MJM^{\mathrm{tr}}$ are said to be *congruent* and they define isometric polar spaces.

Conversely, given bilinear forms $J_1$ and $J_2$ the following example shows how to use the `IsIsometric` function to determine whether $J_1$ and $J_2$ are congruent and if so how to find a matrix $M$ such that $J_1 = MJ_2M^{\mathrm{tr}}$.

**Example: Isometry Between Symplectic Spaces** (ex-5a3a59)

Begin with alternating forms $J_1$ and $J_2$ over $\mathbf{F}_{25}$, construct the corresponding symplectic spaces and then use the isometry to define the matrix $M$.

```
> F<x> := GF(25);
> J1 := Matrix(F,4,4,[ 0, x^7, x^14, x^13, x^19, 0, x^8, x^5,
>   x^2, x^20, 0, x^17, x, x^17, x^5, 0 ]);
> J2 := Matrix(F,4,4,[ 0, x^17, 2, x^23, x^5, 0, x^15, x^5,
>   3, x^3, 0, 4, x^11, x^17, 1, 0 ]);
> V1 := SymplecticSpace(J1);
> V2 := SymplecticSpace(J2);
> flag, f := IsIsometric(V1,V2); assert flag;
> f;
Mapping from: ModTupFld: V1 to ModTupFld: V2 given by a rule
> M := Matrix(F,4,4,[f(V1.i) : i in [1..4]]);
> J1 eq M*J2*Transpose(M);
true
```

**Example: Transforming Forms with TransformForm** (ex-6727c3)

Another way to obtain a matrix with the same effect as $M$ is to use the function `TransformForm`.

```
> M1 := TransformForm(J1,"symplectic");
> M2 := TransformForm(J2,"symplectic");
> M_alt := M1*M2^-1;
> J1 eq M_alt*J2*Transpose(M_alt);
true
```

**`CommonComplement(V, U, W): ModTupFld, ModTupFld, ModTupFld -> ModTupFld`**

A common complement to the subspaces $U$ and $W$ in the vector space $V$. (The subspaces must have the same dimension.) This is used by the following function, which implements Witt's theorem.

**`ExtendIsometry(V, U, f): ModTupFld, ModTupFld, Map -> Map`**

An extension of the isometry $f : U \to V$ to an isometry $V \to V$, where $U$ is a subspace of the polar space $V$.

This is an implementation of Witt's theorem on the extension of an isometry defined on a subspace of a symplectic, unitary, or quadratic space. The isometry $f$ must satisfy $f(U \cap \mathrm{rad}(V)) = f(U) \cap \mathrm{rad}(V)$.

If the characteristic is two and the form $J$ of $V$ is symmetric, then $J$ must be alternating.

`IsometryGroup(V): ModTupFld -> GrpMat`

The group of isometries of the polar space $V$. This includes degenerate polar spaces as well as polar spaces defined by a quadratic form over a field of characteristic two.

Given a reflexive form $J$, the function `IsometryGroup(J)` defined in Chapter Algebras with Involution returns the isometry group of $J$. More generally, if $S$ is a sequence of reflexive forms, the function `IsometryGroup(S)` returns the group of isometries of the system.

**Example: Isometry Group of a Degenerate Quadratic Space** `(ex-833a77)`

We give an example of an isometry group of a degenerate quadratic space over a field of characteristic 2.

```
> F := GF(4);
> Q1 := StandardQuadraticForm(4,F : Minus);
> Q := DiagonalJoin(Q1,ZeroMatrix(F,2,2));
> V := QuadraticSpace(Q);
> G := IsometryGroup(V);
> [ IsIsometry(V,g) : g in Generators(G) ];
[ true, true, true, true, true, true, true ]
> #G;
96259276800
```

**Example: Conjugating Isometry Groups** `(ex-86d774)`

The matrix $M$ constructed in Example *Isometry Between Symplectic Spaces* can be used to conjugate the isometry group of $J_1$ to the isometry group of $J_2$.

```
> F<x> := GF(25);
> J1 := Matrix(F,4,4,[ 0, x^7, x^14, x^13, x^19, 0, x^8, x^5,
>   x^2, x^20, 0, x^17, x, x^17, x^5, 0 ]);
> J2 := Matrix(F,4,4,[ 0, x^17, 2, x^23, x^5, 0, x^15, x^5,
>   3, x^3, 0, 4, x^11, x^17, 1, 0 ]);
> V1 := SymplecticSpace(J1);
> V2 := SymplecticSpace(J2);
> flag, f := IsIsometric(V1,V2); assert flag;
> M := Matrix(F,4,4,[f(V1.i) : i in [1..4]]);
> G1 := IsometryGroup(V1);
> G2 := IsometryGroup(V2);
> M^-1*G1.1*M in G2;
true
> M^-1*G1.2*M in G2;
true
```

## 30.7.2 Similarities

If $\beta$ is a bilinear or sesquilinear form, a linear transformation $f$ is a *similarity* with *multiplier* $\lambda$ if $\beta(uf, vf) = \lambda\beta(u, v)$ for all $u, v$.

`IsSimilarity(U, V, f): ModTupFld, ModTupFld, Map -> BoolElt, FldElt`

Returns `true` if the map $f$ is a similarity from $U$ to $V$ with respect to the attached forms. The second return value is the multiplier.

`IsSimilarity(f): Map -> BoolElt, FldElt`

Returns `true` if the map $f$ is a similarity from its domain to its codomain. The second return value is the multiplier.

`IsSimilarity(V, g): ModTupFld, Mtrx -> BoolElt, FldElt`

Returns `true` if the matrix $g$ is a similarity of $V$ with respect to the attached form. The second return value is the multiplier.

**IsSimilar(V, W): ModTupFld, ModTupFld -> BoolElt, Map**

Determines whether the polar spaces $V$ and $W$ are similar; if they are, a similarity is returned (as a map).

**Example: Similar Hermitian and Skew-Hermitian Spaces** (ex-86e68e)

An example of two unitary spaces, the first defined by an hermitian form, the second by a skew-hermitian form. The spaces are similar but not isometric.

```
> F<z> := GF(25);
> sigma := hom< F -> F | x :-> x^5 >;
> J1 := Matrix(F,4,4,[
>    0, z^3, z^14, z^9,   z^15, 2, z^21, z^5,
>    z^22, z^9, 1, z^7,   z^21, z, z^11, 4]);
> J2 := Matrix(F,4,4,[
>    z^15, z^10, z^17, z^7,  z^14, z^15, z^14, z^9,
>    z, z^10, z^3, z^20,     z^23, z^9, z^16, z^21]);
> V1 := UnitarySpace(J1,sigma);
> V2 := UnitarySpace(J2,sigma);
> IsUnitarySpace(V1);
true 1
> IsUnitarySpace(V2);
true -1
> IsIsometric(V1,V2);
false
> flag, f := IsSimilar(V1,V2);
> flag;
true
> IsSimilarity(V1,V2,f);
true z^3
```

**SimilarityGroup(V): ModTupFld -> GrpMat**

The group of similarities of the polar space $V$. This includes degenerate polar spaces as well as polar spaces defined by a quadratic form over a field of characteristic two.

### 30.7.3 Gram-Schmidt Normalisation

Suppose that $\beta$ is a bilinear or sesquilinear form on the vector space $V$ and that $e_1$, ..., $e_n$ is an ordered basis for $V$. The Gram matrix $J$ of $\beta$ with respect to this basis is $(\beta(e_i, e_j))$. Changing the basis to $e_1A$, ..., $e_nA$ where $A$ is an invertible matrix changes the Gram matrix to the congruent matrix $AJA^{\sigma t}$, where $\sigma$ is a field automorphism. (When the form is symmetric or alternating $\sigma$ is the identity.)

For a positive definite inner product on a real vector space it is possible to choose $A$ so that $AJA^t$ is the identity matrix. The algorithm which achieves this is the *Gram–Schmidt process*.

In general it is possible to choose $A$ so that $D = AJA^{\sigma t}$ is almost diagonal; that is, $D$ is a block diagonal matrix where each block is either a $1 \times 1$ block or a $2 \times 2$ block $\begin{pmatrix} 0 & \alpha \\ s\alpha & 0 \end{pmatrix}$, where $s$ is $-1$ if the form is alternating.

The following intrinsic uses the algorithm of Wilson [Wil13b].

**GramSchmidtPair(J): AlgMatElt -> AlgMatElt, AlgMatElt**

Given the matrix $J$ of a reflexive form, apply Wilson's Gram-Schmidt algorithm to return a pair of matrices $D$ and $A$ such that $AJA' = D$, where $D$ is almost diagonal and $A'$ is either the transposed conjugate of $A$ (when $J$ is hermitian) or the transpose of $A$.

**Example: Gram-Schmidt for a Hermitian Form** (ex-97668f)

For the case of an hermitian matrix over the Gaussian rationals the algorithm is able to diagonalise the form.

```
> F<i> := QuadraticField(-1);
> B := Matrix(F,4,4,[1,0,3-i,2, 0,3,1+i,1+i, 3+i,1-i,0,2, 2,1-i,2,1]);
> B;
[     1      0 -i + 3      2]
[     0      3  i + 1  i + 1]
[ i + 3 -i + 1      0      2]
[     2 -i + 1      2      1]
> D, A := GramSchmidtPair(B);
> D;
[        1         0         0         0]
[        0         3         0         0]
[        0         0    -32/3         0]
[        0         0         0      -5/4]
> A;
[             1              0              0              0]
[             0              1              0              0]
[        -i - 3   1/3*(i - 1)              1              0]
[  1/8*(-i - 4)   1/8*(i - 2) 1/16*(3*i - 7)              1]
```

**Example: Gram-Schmidt for an Alternating Form** (ex-7c8257)

Alternating forms cannot be diagonalised.

```
> B := Matrix(Rationals(),4,4,[0,0,3,2, 0,0,1,1, -3,-1,0,2, -2,-1,-2,0]);
> B;
[ 0  0  3  2]
[ 0  0  1  1]
[-3 -1  0  2]
[-2 -1 -2  0]
> D, A := GramSchmidtPair(B);
> D;
[ 0  1  0  0]
[-1  0  0  0]
[ 0  0  0  1]
[ 0  0 -1  0]
> A;
[ 1 -2  0  0]
[ 2 -6  1  0]
[-1  3  0  0]
[ 0  0  0  1]
> A*B*Transpose(A) eq D;
true
```

## 30.8 Classical Groups

Given the matrix $F$ of a bilinear or sesquilinear form, the functions described in the previous section can be used to construct classical groups which preserve $F$ or preserve $F$ up to a scalar.

**Example: Groups Fixing an Alternating Form** (ex-1225d3)

For example the following code constructs the symplectic group $G$ and the conformal symplectic group $C$ preserving the non-degenerate alternating form $F$.

```
> F := Matrix(GF(5),4,4,[0,-1,-1,1, 1,0,1,-1, 1,-1,0,1, -1,1,-1,0]);
> V := SymplecticSpace(F);
> G := IsometryGroup(V);
> C := SimilarityGroup(V);
```

(continues on next page)

(continued from previous page)

```
> f1, _ := IsIsomorphic(G,Sp(4,5));
> f2, _ := IsIsomorphic(C,CSp(4,5));
> f1, f2;
true true
```

If only the symplectic group is needed this can be obtained on one line

```
> G := IsometryGroup(SymplecticSpace(F));
```

Similar code can be used to construct the orthogonal group and the conformal orthogonal group preserving a quadratic form or the unitary group and conformal unitary group preserving an hermitian form.

However, to construct a group strictly between the isometry group and the similarity group of a form some more work is needed. The following example illustrates how this may be carried out for an hermitian form.

**Example: Special Unitary Group of a Hermitian Form** (ex-e06cca)

This example constructs the group of unitary matrices of determinant 1 which preserve an hermitian form $F$. The first step is to define a Magma function which takes the form as its only argument. The base field and dimension of $F$ can be recovered from $F$ itself and then an isometry $\varphi$ is constructed from the unitary space of the standard form to the unitary space of $F$. This is used to transform the generators of a standard copy of the special unitary group to the group preserving the form.

```
> specialUnitaryGrp := function(F)
>   K := BaseRing(F); n := NumberOfRows(F);
>   J, sigma := StandardHermitianForm(n,K);
>   V := UnitarySpace(F,sigma);
>   S := UnitarySpace(J,sigma);
>   flag, phi := IsIsometric(S,V);
>   assert flag;
>   T := Matrix(K,n,n,[phi(S.i) : i in [1..n]]);
>   H := SpecialUnitaryGroup(n,K);
>   return sub<IsometryGroup(V) | [T^-1*H.i*T : i in␣
↪[1..NumberOfGenerators(H)]]>;
> end function;
```

To complete the example we define an hermitian form $F$ and then use the above function to find the special unitary group that preserves $F$.

```
> K<z> := GF(9);
> F := Matrix(K,4,4,[1,z^7,z^6,2, z^5,2,z^6,z^6,  z^2,z^2,1,z^3, 2,z^2,z,0]);
> G := specialUnitaryGrp(F);
> flag where flag is IsIsomorphic(G,SU(4,3));
true
```

## 30.9 Lie Algebras and Bilinear Forms

If $J$ is the matrix of a bilinear form, the Lie algebra of derivations of $J$ consists of the matrices $X$ such that $XJ + JX^{\mathrm{tr}} = 0$.

`DerivationAlgebra(J): AlgMatElt -> AlgLie`

```
Rep  : MonStgElt                    Default: "Sparse"
Check: BoolElt                      Default: false
```

The Lie algebra of derivations of the bilinear form with matrix $J$. The possible values for `Rep` are `"Dense"`, `"Sparse"`, and `"Partial"` with the default being `"Sparse"`.

**Example: Lie Algebra of Type C3** (ex-3caa04)

Construct the Lie algebra preserving the standard alternating form of rank 6 over the field of 7 elements and check that it is a simple algebra of type $C_3$.

```
> J := StandardAlternatingForm(6,7);
> L := DerivationAlgebra(J);
> IsSimple(L);
true
> SemisimpleType(L);
C3
```

**Example: G2 Ideal in Characteristic 2** (ex-196405)

Over a field of characteristic 2, the Lie algebra of an alternating form is no longer simple. In this example the Lie algebra $L$ of the standard alternating form of rank 6 over $\mathbf{F}_8$ is of symplectic type $C_3$ but has an ideal $I$ of type $G_2$. The dimension of the centre of $L$ is 1 and so the ideal $I$ will appear as either the first or second composition factor.

```
> J := StandardAlternatingForm(6,8);
> L := DerivationAlgebra(J);
> SemisimpleType(L);
C3
> Dimension(Centre(L));
1
> CF := CompositionFactors(L);
> CF;
[
    Lie Algebra of dimension 1 with base ring GF(2^3),
    Lie Algebra of dimension 14 with base ring GF(2^3),
    Lie Algebra of dimension 1 with base ring GF(2^3),
    Lie Algebra of dimension 1 with base ring GF(2^3),
    Lie Algebra of dimension 1 with base ring GF(2^3),
    Lie Algebra of dimension 1 with base ring GF(2^3),
    Lie Algebra of dimension 1 with base ring GF(2^3),
    Lie Algebra of dimension 1 with base ring GF(2^3)
]
> exists(I){I : I in CF | Dimension(I) eq 14 };
true
> IsSimple(I);
true
> SemisimpleType(I);
G2
```

Another way to construct a Lie algebra from an alternating form $\beta$ with matrix $J$ defined on a vector space $V$ of dimension $n$ over a field $F$ is to set $L = V \oplus F$ and define the multiplication by $[e_i, e_j] = \beta(e_i, e_j)$, where $e_1$, $e_2$, ..., $e_n$ is a basis for $V$. (All other structure constants are 0.) This is the (generalised) Heisenberg algebra.

`HeisenbergAlgebra(J): AlgMatElt -> AlgLie`

```
Rep   : MonStgElt                    Default: "Sparse"
Check: BoolElt                       Default: false
```

The nilpotent Lie algebra whose structure constants are obtained from the alternating form with matrix $J$ as described above. The possible values for `Rep` are `"Dense"`, `"Sparse"`, and `"Partial"` with the default being `"Sparse"`.

**Example: Heisenberg Algebra of a Form** (ex-aa6842)

Continuing the previous example we construct an explicit isomorphism between the quotient of $L$ by its $G_2$ ideal and the Heisenberg algebra of the form.

```
> Q := CS[1];
> W := L/Q;
> W;
Lie Algebra of dimension 7 with base ring GF(2^3)
> Z := Centre(W);
> z := Z.1;
> exists(u1,v1){ <u,v> : u,v in W | u*v ne 0 and u*v eq z };
true
> W1 := Centraliser(W,sub<W|u1,v1>);
true
> exists(u2,v2){ <u,v> : u,v in W1 | u*v ne 0 and u*v eq z };
true
> W2:= Centraliser(W1,sub<W1|u2,v2>);
> exists(u3,v3){ <u,v> : u,v in W2 | u*v ne 0 and u*v eq z };
> H := HeisenbergAlgebra(J);
> f := hom< H -> W | u1,u2,u3,v3,v2,v1,z >;
> forall{ <u,v> : u,v in Basis(H) | f(u*v) eq f(u)*f(v) };
true
> Kernel(f);
Lie Algebra of dimension 0 with base ring GF(2^3)
> Image(f) eq W;
true
```

## 30.10 Wall Forms

Given an isometry $f$ of a quadratic, symplectic, or unitary space $V$ with bilinear or sesquilinear form $\beta$, the Wall form of $f$ is the form $\theta$ defined on the image $I$ of $1-f$ by $\theta(u,v) = \beta(w,v)$, where $u = w(1-f)$. In general, the Wall form is not reflexive.

`WallForm(V, f): ModTupFld, Mtrx -> ModTupFld, Map`

The space of the Wall form of the isometry $f$ and its embedding in $V$.

`WallIsometry(V, I, mu): ModTupFld, ModTupFld, Map -> Mtrx`

The inverse of `WallForm`. This is an isometry corresponding to the embedding $\mu : I \to V$, where $V$ is a quadratic, symplectic, or unitary space.

`WallDecomposition(V, f): ModTupFld, Mtrx -> Mtrx, Mtrx`

An isometry $f$ of a quadratic or symplectic space $V$ is *Wall-regular* if the restriction of $1-f$ to the image of $1-f$ is invertible. If $f$ is any isometry of $V$ this function returns a Wall-regular element $f_r$ and a unipotent element $f_u$ such that $f = f_r f_u = f_u f_r$.

`SemiOrthogonalBasis(V): ModTupFld -> SeqEnum`

If $V$ is a vector space with a bilinear form $\beta$, a basis $e_1$, $e_2$, ..., $e_n$ for $V$ is semi-orthogonal if $\beta(e_i, e_j) = 0$ for $i < j$. This function returns a semi-orthogonal basis with respect to the non-degenerate, non-alternating form attached to $V$. If the base field is $\mathbf{F}_2$, the form should be symmetric.

`GeneralisedWallForm(V, f): ModTupFld, Mtrx -> ModTupFld, Map`

This function returns the space of the generalised Wall form of the similarity $f$ and its embedding in the quadratic space $V$. Suppose that the quadratic form $Q$ of $V$ is nondegenerate and let $\beta$ be its polar form. Then $Q(vf) = \eta Q(v)$ for some $\eta$. Suppose that $\eta = \zeta^2$ and let $V(f,\zeta)$ denote the $\zeta$-eigenspace of $f$. The generalised Wall form $\theta$ of $f$ is defined on the orthogonal complement of $V(f,\zeta)$ by $\theta(u,v) = \beta(w,v)$, where $u = \zeta w - wf$.

## 30.11 Invariant Forms

Given a group $G$ which acts on a vector space $V$ over a finite field $F$, the space of all $G$-invariant bilinear forms is isomorphic to $\mathrm{Hom}_G(V, V^*)$, where $V^*$ is the dual space of $V$. The isomorphism associates the form $\beta$ to $\theta \in \mathrm{Hom}_G(V, V^*)$, where $\beta(u, v) = \langle v, u\theta \rangle$ and where $\langle v, \varphi \rangle$ denotes the action of $\varphi$ on $v$. If $v_1$, $v_2$, ..., $v_n$ is a basis for $V$ with dual basis $\omega_1$, $\omega_2$, ..., $\omega_n$, the matrix of $\theta$ with respect to these bases is $J = (\beta(e_i, e_j))$.

A linear transformation with matrix $A$ preserves the form if and only if $AJA^{\mathrm{tr}} = J$.

If the characteristic of the field is not 2, then $J = \frac{1}{2}(J + J^{\mathrm{tr}}) + \frac{1}{2}(J - J^{\mathrm{tr}})$. Therefore, in this case, every $G$-invariant form is the sum of a $G$-invariant symmetric form and a $G$-invariant alternating form. If the characteristic of the field is 2, every alternating form is symmetric. Thus in this case the space of $G$-invariant alternating forms is a subspace of the space of $G$-invariant symmetric forms. If the $G$-module is irreducible these two spaces coincide.

`InvariantBilinearForms(G): GrpMat -> SeqEnum[AlgMatElt], SeqEnum[AlgMatElt]`

Given a matrix group $G$ this function returns two sequences: a basis for the space of $G$-invariant symmetric forms and a basis for the space of $G$-invariant alternating forms.

**Example: Invariant Form of a Reducible Group** (ex-246b09)

In this example the group $G$ is reducible but (up to a scalar multiple) there is a unique $G$-invariant bilinear form.

```
> F<x> := GF(25);
> G := MatrixGroup< 4, F |
>    [ 1, 0, 0, 0,  0, 1, 0, 0,  0, x^14, 1, 0,  0, 0, 0, 1 ],
>    [ 3, x^23, x^20, x^10,  2, 3, 0, x^13,  4, x^10, x^13, x^23,
>      x^5, x^11, x, x^17 ] >;
> IsIrreducible(G);
false
> InvariantBilinearForms(G);
[]
[
    [   0    0    0    1]
    [   0    0    1    0]
    [   0    4    0    0]
    [   4    0    0    0]
]
```

If $G$ acts irreducibly on a vector space $V$ of dimension $n$ over a (finite) field $F$ and if $\theta_0 : V \to V^*$ is a $G$-invariant isomorphism, then $D \to \mathrm{Hom}_G(V, V^*) : \theta \mapsto \theta\theta_0$ is an isomorphism of vector spaces, where $D = \mathrm{End}_G(V)$. The algebra $D$ is a division ring and hence a field (since $F$ is finite). Thus $V$ becomes a vector space of dimension $m$ over $D$, where $n = m|D : F|$ and $G$ is isomorphic to a subgroup of $\mathrm{GL}(m, D)$.

**Example: Forms of a Non-Absolutely Irreducible Group** (ex-8559f1)

If $G$ acts irreducibly on $V$, the spaces of symmetric and alternating $G$-invariant forms are isomorphic (as vector spaces) to subfields of $\mathrm{End}_G(V)$ and therefore their dimensions are either 0 or divide $\dim_F(V)$.

```
> F<a> := GF(25);
> G := MatrixGroup< 4, F |
>   [ a^10, a^21, a^4, 4,
>     a^16, 4, a^9, a^8,
>     a^20, 4, 4, a^13,
>     0, a^2, a^11, a ] >;
> IsIrreducible(G), #G;
true 626
```

(continues on next page)

(continued from previous page)

```
> sym, alt := InvariantBilinearForms(G);
> #sym,#alt;
2 2
```

If the characteristic of the field is not two and if $J$ is a symmetric bilinear form there is a unique upper triangular matrix $Q$ such that $J = Q + Q^{\mathrm{tr}}$.

On the other hand, if the characteristic is two and $J$ is alternating, the upper triangular matrices $Q$ such that $J = Q + Q^{\mathrm{tr}}$ form an affine space of dimension $\dim V$.

Suppose that the characteristic is two. If $G$ preserves a symmetric bilinear form which is not alternating, then $G$ is reducible. Conversely, if $G$ is irreducible and if $J$ is the matrix of a symmetric form preserved by $G$, then the form must be alternating and there is a unique $G$-invariant quadratic form $Q$ such that $J = Q + Q^{\mathrm{tr}}$.

`InvariantQuadraticForms(G): GrpMat -> SeqEnum[AlgMatElt]`

A basis for the space of quadratic forms preserved by the irreducible matrix group $G$.

**Example: Invariant Quadratic Forms of a Cyclic Group** (ex-e24e49)

In the following example the quadratic forms which are invariant under the action of a cyclic group $H$ of order 13 form a vector space of dimension 3 over $\mathbf{F}_4$.

```
> F<z> := GF(4);
> H := MatrixGroup<6,F |
>     [ z, 0, z^2, z, z, 1,
>       1, z, 0, z, z, z,
>       0, z^2, z, 1, z^2, z^2,
>       z, 1, z, 1, 1, 0,
>       1, z^2, z, z, 0, 1,
>       1, 0, 1, 0, z^2, 1 ] >;
>
> InvariantQuadraticForms(H);
[
    [  1   1   0   1   0   0]
    [  0   0   1 z^2 z^2   z]
    [  0   0   1   0 z^2   z]
    [  0   0   0   0   0   z]
    [  0   0   0   0 z^2   z]
    [  0   0   0   0   0 z^2],

    [  1   0   1 z^2   1   0]
    [  0   z   1   1   1   z]
    [  0   0   z   0   1   0]
    [  0   0   0 z^2 z^2 z^2]
    [  0   0   0   0 z^2   1]
    [  0   0   0   0   0   1],

    [  0   0   0   0   0   1]
    [  0   0   0   0   1   0]
    [  0   0   1   1   0   0]
    [  0   0   0   z   0   0]
    [  0   0   0   0   0   0]
    [  0   0   0   0   0   0]
]
```

Given a group $G$ which acts on a vector space $V$ over a finite field $F$ with an automorphism $F \to F : a \mapsto \overline{a}$ of order 2, the space of $G$-invariant sesquilinear forms is isomorphic to the space of $G$-invariant semilinear

maps from $V$ to $V^*$; equivalently it is isomorphic to $\mathrm{Hom}_G(V, \overline{V}^*)$, where $\overline{V}^*$ is the semilinear dual of $V$, namely the space of all semilinear maps from $V$ to $F$.

If $\theta \in \mathrm{Hom}_G(V, \overline{V}^*)$, the corresponding sesquilinear form $\beta$ is defined by $\beta(u, v) = \langle v, u\theta \rangle$ where, as before, $\langle v, \varphi \rangle$ denotes the action of $\varphi$ on $v$.

**SemilinearDual(M, mu): ModGrp, Map -> ModGrp**

The semilinear dual of the $G$-module $M$ with respect to the field automorphism mu.

**InvariantSesquilinearForms(G): GrpMat -> SeqEnum[AlgMatElt]**

A basis for the space of *hermitian* forms preserved by the matrix group $G$.

**Example: Invariant Hermitian Forms** (ex-17b876)

Let $F_0$ be the fixed field of the involution. The set $\mathcal{H}$ of $G$-invariant hermitian forms is a vector space over $F_0$ and if the characteristic of $F$ is not 2, then $\mathrm{Hom}_G(V, \overline{V}^*) \simeq \mathcal{H} \otimes_{F_0} F$.

```
> F<x> := GF(5,2);
> mu := hom< F->F | x :-> x^5 >;
> H := MatrixGroup< 5, F |
>    [ 0, x^3, 0, 1, x^9, x^8, 1, 0, x^11, x^7, x^20, x^16, 1,
>     x^11, x^3, x^21, 4, 1, x^3, x^23, x^4, x^3, x, x^3, 2 ] >;
> M := GModule(H);
> D := SemilinearDual(M,mu);
> E := AHom(M,D);
> Dimension(E);
5
> herm := InvariantSesquilinearForms(H);
> #herm;
5
```

**Example: Preserving Bilinear and Sesquilinear Forms** (ex-9dd62c)

If an irreducible group preserves both a bilinear and a sesquilinear form then it is realisable over a subfield of its base field. Conversely, this observation can be used to construct an example:

```
> F<x> := GF(81);
> H := MatrixGroup< 4, F | [ChangeRing(g,F) : g in Generators(Sp(4,9))]>;
> InvariantBilinearForms(H);
[]
[
    [   0    0    0    1]
    [   0    0    1    0]
    [   0    2    0    0]
    [   2    0    0    0]
]
> InvariantSesquilinearForms(H);
[
    [   0    0    0 x^45]
    [   0    0 x^45    0]
    [   0  x^5    0    0]
    [ x^5    0    0    0]
]
```

**InvariantFormBases(G): GrpMat -> SeqEnum[AlgMatElt], SeqEnum[AlgMatElt], SeqEnum[AlgMatElt], SeqEnum[AlgMatElt]**

This function returns four sequences: bases for the spaces of symmetric, alternating, hermitian, and quadratic forms preserved by the matrix group $G$.

### 30.11.1 Semi-invariant Forms

Given a vector space $V$ over a finite field $F$ and a group $G$ which acts on $V$, a bilinear form $\beta : V \times V \to F$ is *semi-invariant* if for all $g \in G$ there is a scalar $\lambda(g)$ such that $\beta(ug, vg) = \lambda(g)\beta(u, v)$ for all $u, v \in V$. The function $\lambda : G \to F^\times$ is a homomorphism and its kernel contains the derived group of $G$. The *twisted dual* $V^*_\lambda$ of the $G$-module $V$ is the dual space of $V$ with $G$-action given by $\langle v, \varphi g\rangle = \lambda(g)\langle vg^{-1}, \varphi\rangle$; thus if $A$ is the matrix of $g$ acting on $V$ the matrix of the action on $V^*_\lambda$ with respect to the dual basis is $\lambda(g)A^{-\mathrm{tr}}$.

The space of all semi-invariant bilinear forms is isomorphic to $\mathrm{Hom}_G(V, V^*_\lambda)$. The isomorphism associates the form $\beta$ to $\theta \in \mathrm{Hom}_G(V, V^*_\lambda)$, where $\beta(u, v) = \langle v, u\theta\rangle$.

If $\beta$ is a bilinear form with matrix $J$, then the linear transformation $g$ with matrix $A$ preserves the form up to multiplication by $\lambda(g)$ if and only if $AJA^{\mathrm{tr}} = \lambda(g)J$.

`TwistedDual(M, lambda): ModGrp, Map -> ModGrp`

The twisted dual of the $G$-module M with respect to the linear character `lambda`.

`SemiInvariantBilinearForms(G): GrpMat -> SeqEnum`

A sequence of triples $\langle L, S, A\rangle$ where $L$ is a sequence of field elements (one for each generator) which define a homomorphism from the matrix group $G$ to its base field and $S$ and $A$ are bases for the spaces of symmetric and alternating forms preserved by $G$ (up to multiplication by scalars).

`SemiInvariantQuadraticForms(G): GrpMat -> SeqEnum`

A sequence of pairs $\langle L, Q\rangle$ where $L$ is a sequence of field elements (one for each generator) which define a homomorphism from the matrix group $G$ to its base field and $Q$ is a basis for the space of quadratic forms preserved by $G$ (up to multiplication by scalars).

`TwistedSemilinearDual(M, lambda, mu): ModGrp, Map, Map -> ModGrp`

The twisted semilinear dual of the $G$-module $M$ with respect to the linear character $\lambda$ and the field automorphism $\mu$.

`SemiInvariantSesquilinearForms(G): GrpMat -> SeqEnum`

A sequence of pairs $\langle L, H\rangle$ where $L$ is a sequence of field elements (one for each generator) which define a homomorphism from the matrix group $G$ to the field $F_0$, where the base field of $G$ is a quadratic extension of $F_0$, and $H$ is a basis for the space of hermitian forms preserved by $G$ (up to multiplication by scalars).

**Example: Semi-Invariant Forms of a Normal Subgroup** `(ex-f6ba6b)`

In this example $H$ is a normal subgroup of the absolutely irreducible group $N$ an $H$ is irreducible but not absolutely irreducible.

```
> F<x> := GF(3,2);
> H := MatrixGroup<3,F|
>   [x^2,x^7,x^3, x,0,1, x^3,x^6,2],
>   [x^3, 0, 0, 0, x^3, 0, 0, 0, x^3 ] >;
> N := MatrixGroup<3,F|H.1,H.2,[x^5,x^5,2, 0,x^2,x^6, x^7,x^7,2]>;
> IsNormal(N,H);
true
> IsIrreducible(H), IsAbsolutelyIrreducible(H);
true false
> IsIrreducible(N), IsAbsolutelyIrreducible(N);
true true
> SemiInvariantSesquilinearForms(H);
[
    <[ 1, 2 ],
    [
        [  1   x   x]
        [x^3   0 x^3]
        [x^3   x   1],
```

(continues on next page)

(continued from previous page)

```
        [  0 x^3   x]
        [  x   0 x^5]
        [x^3 x^7   0],

        [  0   0   1]
        [  0   1   0]
        [  1   0   0]
    ]>
]
> SemiInvariantSesquilinearForms(N);
[
    <[ 1, 2, 1 ],
    [
        [  0   0   1]
        [  0   1   0]
        [  1   0   0]
    ]>
]
```

## 30.12 Polar Spaces More Generally

In V2.29 (September 2025), in order to support more functionality for polar spaces, a category `SpcPlr` has been added, and one can create a polar spaces also in this category.

### 30.12.1 Creation of PolarSpaces

**`AmbientPolarSpace(J): AlgMatElt -> SpcPlr`**

**`AmbientPolarSpace(J, a): AlgMatElt, FldAut -> SpcPlr`**

Builds a polar space with respect to the matrix $J$ and the field automorphism $a$.

**`PolarSpace(L): LatNF -> SpcPlr`**

**`PolarSpace(L): Lat -> SpcPlr`**

**`AmbientSpace(L): LatNF -> SpcPlr`**

The polar space associated to the lattice $L$.

**`ChangeRing(V, R): SpcPlr, Rng -> SpcPlr`**

The polar space obtained from $V$ by base change to $R$.

### 30.12.2 Properties of Polar Spaces

**`BaseField(V): SpcPlr -> Fld`**

**`BaseRing(V): SpcPlr -> Fld`**

**`CoefficientField(V): SpcPlr -> Fld`**

**`CoefficientRing(V): SpcPlr -> Fld`**

The field over which $V$ is defined.

**`VectorSpace(V): SpcPlr -> ModTupFld`**

**`KSpace(V): SpcPlr -> ModTupFld`**

The underlying vector space.

**`Dimension(V): SpcPlr -> RngIntElt`**

The dimension of $V$.

`InnerForm(V): SpcPlr -> AlgMatElt`

The inner form associated to $V$.

`Involution(V): SpcPlr -> FldAut`

The involution with respect to which $V$ is hermitian or skew-hermitian.

`SpaceType(V): SpcPlr -> MonStgElt`

The type of the polar space $V$, returned as a string.

`Diagonal(V): SpcPlr -> AlgMatElt`

The coefficients of the diagonalized form.

### 30.12.3 Predicates on Polar Spaces

`IsDefinite(V): SpcPlr -> BoolElt`

Whether $V$ space is totally positive definite, or totally negative definite.

## 30.13 Bibliography


- [Bou07] N. Bourbaki. *Éléments de mathématique. Algèbre. Chapitre 9.* Springer-Verlag, Berlin, 2007. ISBN 978-3-540-35338-6; 3-540-35338-0. Reprint of the 1959 original.
- [Bra36] Richard Brauer. A characterization of null systems in projective space. *Bull. Amer. Math. Soc.*, 42(4):247–254, 1936.
- [Bue95] Francis Buekenhout. *Handbook of incidence geometry.* North-Holland, Amsterdam, 1995.
- [Tay92] Donald E. Taylor. *The geometry of the classical groups.* Volume 9 of Sigma Series in Pure Mathematics. Heldermann Verlag, Berlin, 1992. ISBN 3-88538-009-9.
- [Wil13b] James B. Wilson. Optimal algorithms of Gram-Schmidt type. *Linear Algebra Appl.*, 438(12):4573–4583, 2013. URL: https://doi.org/10.1016/j.laa.2013.02.026, doi:10.1016/j.laa.2013.02.026.

# References


[ABM99] John Abbott, Manuel Bronstein, and Thom Mulders. Fast Deterministic Computation of Determinants of Dense Matrices. In Sam Dooley, editor, *Proceedings ISSAC'99*, 197–204. New York, 1999. ACM Press.

[Bou07] N. Bourbaki. *Éléments de mathématique. Algèbre. Chapitre 9*. Springer-Verlag, Berlin, 2007. ISBN 978-3-540-35338-6; 3-540-35338-0. Reprint of the 1959 original.

[Bra36] Richard Brauer. A characterization of null systems in projective space. *Bull. Amer. Math. Soc.*, 42(4):247–254, 1936.

[Bue95] Francis Buekenhout. *Handbook of incidence geometry*. North-Holland, Amsterdam, 1995.

[CLG97b] Frank Celler and Charles R. Leedham-Green. Calculating the Order of an Invertible Matrix. In Larry Finkelstein and William M. Kantor, editors, *Groups and Computation II*, volume 28 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 55–60. AMS, 1997.

[CC82] T.W.J. Chou and G.E. Collins. Algorithms for the solution of systems of linear Diophantine equations. *SIAM J. Computing*, 11(4):687–708, 1982.

[COS86] D. Coppersmith, A. M. Odlyzko, and R. Schroeppel. Discrete logarithms in GF($p$). *Algorithmica*, 1:1–15, 1986.

[DEJ84] I.S. Duff, A.M. Erisman, and J.K.Reid. *Direct methods for sparse matrices*. Monographs on Numerical Analysis. Oxford University Press, 1984.

[HHR93] George Havas, Derek F. Holt, and Sarah Rees. Recognizing badly presented Z-modules. *Linear Algebra and its Applications*, 192:137–164, 1993.

[KB79] R. Kannan and A. Bachem. Polynomial algorithms for computing the Smith and Hermite normal forms of an integer matrix. *SIAM J. Computing*, 9:499–507, 1979.

[LO91a] B. A. LaMacchia and A. M. Odlyzko. Computation of Discrete Logarithms in Prime Fields. In A.J. Menezes and S. Vanstone, editors, *Advances in Cryptology—CRYPTO 1990*, volume 537 of INCS, 616–618. Springer-Verlag, 1991.

[LO91b] B. A. LaMacchia and A. M. Odlyzko. Solving Large Sparse Linear Systems over Finite Fields. In A.J. Menezes and S. Vanstone, editors, *Advances in Cryptology—CRYPTO 1990*, volume 537 of LNCS, 109–133. Springer-Verlag, 1991.

[Lub02] F. Lübeck. On the computation of elementary divisors of integer matrices. *J. Symbolic Comp.*, 33:57–65, 2002.

[Ste97] Allan Steel. A New Algorithm for the Computation of Canonical Forms of Matrices over Fields. *J. Symbolic Comp.*, 24(3):409–432, 1997.

[Tay92] Donald E. Taylor. *The geometry of the classical groups*. Volume 9 of Sigma Series in Pure Mathematics. Heldermann Verlag, Berlin, 1992. ISBN 3-88538-009-9.

[Wil13b] James B. Wilson. Optimal algorithms of Gram-Schmidt type. *Linear Algebra Appl.*, 438(12):4573–4583, 2013. URL: https://doi.org/10.1016/j.laa.2013.02.026, doi:10.1016/j.laa.2013.02.026.

# Index

## Symbols


(u, v) (*literal*), 56
&meet S (*operation*), 65
- A (*operation*), 14
- u (*operation*), 56
-A (*operation*), 43


## A


A * B (*operation*), 14, 43
a * b (*operation*), 67
A * x (*operation*), 14, 43
A + B (*operation*), 14, 43
A - B (*operation*), 14, 43
A ^ -1 (*operation*), 14
A ^ n (*operation*), 14, 43
A ^-1 (*operation*), 43
A eq B (*operation*), 42
a(v) (*literal*), 67
Accessing Matrix Entries (*example*), 8
Accessing Sparse Matrix Entries (*example*), 38
AddColumn(~A, c, i, j) (*procedure*), 11, 41
AddColumn(A, c, i, j) (*function*), 11, 41
AddRow(~A, c, i, j) (*procedure*), 11, 41
AddRow(A, c, i, j) (*function*), 11, 41
AddScaledMatrix(~A, s, B) (*procedure*), 15
AddScaledMatrix(A, s, B) (*function*), 15
Adjoint(A) (*function*), 21
AllIsotropicSubspaces(V, k) (*function*), 78
AmbientPolarSpace(J) (*function*), 94
AmbientPolarSpace(J, a) (*function*), 94
AmbientSpace(L) (*function*), 94
AntisymmetricMatrix(Q) (*function*), 5
AntisymmetricMatrix(R, Q) (*function*), 5
ArfInvariant(V) (*function*), 80
A[i .. j] (*indexed*), 8
A[i, j] (*indexed*), 8, 38
A[i, j] := x (*literal*), 8, 38
A[i] (*indexed*), 8, 38
A[i] := v (*literal*), 8
A[Q] (*indexed*), 8


## B


BaseField(V) (*function*), 63, 94
BaseRing(A) (*function*), 8, 37
BaseRing(V) (*function*), 94
Bases of the Ternary Golay Code (*example*), 66
Basis(V) (*function*), 65
BasisElement(V, i) (*function*), 65
BasisMatrix(V) (*function*), 66
BlockMatrix(m, n, blocks) (*function*), 13
BlockMatrix(m, n, rows) (*function*), 13
BlockMatrix(rows) (*function*), 13


## C


CanChangeRing(A, R) (*function*), 14
Canonical Forms over a Finite Field (*example*), 22
ChangeRing(A, f) (*function*), 14
ChangeRing(A, R) (*function*), 14, 42
ChangeRing(A, R, f) (*function*), 14
ChangeRing(V, R) (*function*), 94
CharacteristicPolynomial(A: parameters) (*function*), 19, 45
CharacteristicVector(V, S) (*function*), 55
CharacterQQModSquares(d, r) (*function*), 81
Codomain(a) (*function*), 67
CoefficientField(V) (*function*), 63, 94
CoefficientRing(A) (*function*), 8, 37
CoefficientRing(V) (*function*), 94
Cofactor(M, i, j) (*function*), 19
Cofactors(M) (*function*), 19
Cofactors(M, r) (*function*), 19
Cokernel(a) (*function*), 68
ColumnSubmatrix(A, i) (*function*), 10, 40
ColumnSubmatrix(A, i, k) (*function*), 10, 40
ColumnSubmatrixRange(A, i, j) (*function*), 10, 40
ColumnWeight(A, j) (*function*), 38
ColumnWeights(A) (*function*), 38
CommonComplement(V, U, W) (*function*), 83
Complement(V, U) (*function*), 65
ConjugateTranspose(M, sigma) (*function*), 79
Conjugating Isometry Groups (*example*), 84
Coordinates(V, v) (*function*), 66
Creating Matrices with Matrix (*example*), 2
Creating Sparse Matrices (*example*), 35


## D


DecomposeVector(U, v) (*function*), 65
Defining a Nontrivial Inner Product (*example*), 57
Degree(u) (*function*), 63
Degree(V) (*function*), 63
Density(A) (*function*), 8, 37
Depth(u) (*function*), 56
DerivationAlgebra(J) (*function*), 87
Determinant(A: parameters) (*function*), 18, 45

Diagonal(V) (*function*), 95
DiagonalJoin(A, B) (*function*), 41
DiagonalJoin(Q) (*function*), 13
DiagonalJoin(T) (*function*), 13
DiagonalJoin(X, Y) (*function*), 13
DiagonalMatrix(Q) (*function*), 4
DiagonalMatrix(R, n, Q) (*function*), 4
DiagonalMatrix(R, Q) (*function*), 4
DiagonalSparseMatrix(Q) (*function*), 36
DiagonalSparseMatrix(R, n, Q) (*function*), 36
DiagonalSparseMatrix(R, Q) (*function*), 36
DicksonInvariant(V, f) (*function*), 80
Dimension(V) (*function*), 64, 66, 94
DirectSum(V,W) (*function*), 78
Discrete Logarithms by Linear Sieve (*example*), 46
Discriminant(V) (*function*), 80
Domain(a) (*function*), 67
DotProduct(u, v) (*function*), 71
DotProductMatrix(S) (*function*), 71

## E

Echelon Forms over a Field and the Integers (*example*), 24
EchelonForm(A) (*function*), 20
Eigenspace(A, e) (*function*), 20, 45
Eigenvalues(A) (*function*), 20, 45
ElementaryDivisors(A) (*function*), 24, 46
ElementToSequence(A) (*function*), 8, 37
ElementToSequence(u) (*function*), 55
elt<V | L> (*constructor*), 54
Eltseq(A) (*function*), 8, 37
Eltseq(u) (*function*), 55
EnsureUpperTriangular(A) (*function*), 71
Enumerating All Solutions of a System (*example*), 16
ExtendBasis(Q, U) (*function*), 66
ExtendBasis(U, V) (*function*), 66
ExtendField(V, L) (*function*), 63
ExtendIsometry(V, U, f) (*function*), 83
ExtractBlock(A, i, j, p, q) (*function*), 9, 39
ExtractBlockRange(A, i, j, r, s) (*function*), 9, 39
Extracting Submatrices (*example*), 10

## F

FactoredCharacteristicPolynomial(A: parameters) (*function*), 20, 45
FactoredMCPolynomials(A: parameters) (*function*), 20, 45
FactoredMinimalAndCharacteristic Polynomials(A: parameters) (*function*), 20, 45
FactoredMinimalPolynomial(A: parameters) (*function*), 20, 45
FactoredOrder(A) (*function*), 25
FactoredProjectiveOrder(A) (*function*), 26
FirstIsotropicSubspace(V, k) (*function*), 78
Forms of a Non-Absolutely Irreducible Group (*example*), 90
FrobeniusFormAlternating(A) (*function*), 22
FrobeniusImage(A, e) (*function*), 32

## G

G2 Ideal in Characteristic 2 (*example*), 88
GeneralisedWallForm(V, f) (*function*), 89
Generators(V) (*function*), 64
Generic(V) (*function*), 64
Golay Code in a Vector Space (*example*), 60
Golay Code in an R-Module (*example*), 60
Golay Quotient as a Complement (*example*), 61
Gram Matrix Versus Inner Product Matrix (*example*), 71
Gram-Schmidt for a Hermitian Form (*example*), 85
Gram-Schmidt for an Alternating Form (*example*), 86
GramMatrix(V) (*function*), 71
GramSchmidtPair(J) (*function*), 85
Groups Fixing an Alternating Form (*example*), 86

## H

HasFiniteOrder(A) (*function*), 25
HasIsotropicVector(V) (*function*), 72
HasSingularVector(V) (*function*), 72
Heisenberg Algebra of a Form (*example*), 88
HeisenbergAlgebra(J) (*function*), 88
HermiteForm(A) (*function*), 23
HessenbergForm(A) (*function*), 22
Hom(V, W) (*function*), 54
HorizontalJoin(A, B) (*function*), 41
HorizontalJoin(Q) (*function*), 13
HorizontalJoin(T) (*function*), 13
HorizontalJoin(X, Y) (*function*), 13
Hyperbolic Splitting in Characteristic 2 (*example*), 74
Hyperbolic Splitting of a Symmetric Form (*example*), 74
HyperbolicBasis(U, B, W) (*function*), 81
HyperbolicPair(V, u) (*function*), 73
HyperbolicSplitting(V) (*function*), 74

## I

IdentityMatrix(R, n) (*function*), 4
IdentitySparseMatrix(R, n) (*function*), 36
Image(a) (*function*), 67
Indexing Tuple and Matrix Spaces (*example*), 58
InnerForm(V) (*function*), 94
InnerProduct(u, v) (*function*), 56
InnerProductMatrix(V) (*function*), 71
InsertBlock(~A, B, i, j) (*function*), 9, 40
InsertBlock(A, B, i, j) (*function*), 9, 40
Invariant Form of a Reducible Group (*example*), 90

Invariant Hermitian Forms (*example*), 92
Invariant Quadratic Forms of a Cyclic Group (*example*), 91
InvariantBilinearForms(G) (*function*), 90
InvariantFactors(A) (*function*), 21
InvariantFormBases(G) (*function*), 92
InvariantQuadraticForms(G) (*function*), 91
InvariantSesquilinearForms(G) (*function*), 92
Involution(V) (*function*), 95
IsConsistent(A, Q) (*function*), 15
IsConsistent(A, W) (*function*), 15
IsDefinite(V) (*function*), 95
IsDegenerate(V) (*function*), 72
IsDiagonal(A) (*function*), 17, 42
IsHermitian(A,f) (*function*), 17
IsIndependent(Q) (*function*), 66
IsIndependent(S) (*function*), 66
IsIsometric(V, W) (*function*), 82
IsIsometry(f) (*function*), 82
IsIsometry(U, V, f) (*function*), 82
IsIsometry(V, g) (*function*), 82
IsLowerTriangular(A) (*function*), 17, 43
IsMinusOne(A) (*function*), 17, 42
IsNondegenerate(V) (*function*), 72
IsNonsingular(V) (*function*), 72
Isometric Polar Spaces (*example*), 82
Isometries from Siegel Transformations (*example*), 82
Isometry Between Symplectic Spaces (*example*), 83
Isometry Group of a Degenerate Quadratic Space (*example*), 84
IsometryGroup(V) (*function*), 83
IsOne(A) (*function*), 17, 42
Isotropic Vector in a Unitary Space (*example*), 79
Isotropic Vectors in a Pseudo-Symplectic Space (*example*), 73
IsPolarSpace(V) (*function*), 78
IsPseudoSymplecticSpace(W) (*function*), 78
IsScalar(A) (*function*), 17, 42
IsSimilar(A, B) (*function*), 22
IsSimilar(V, W) (*function*), 84
IsSimilarity(f) (*function*), 84
IsSimilarity(U, V, f) (*function*), 84
IsSimilarity(V, g) (*function*), 84
IsSingular(A) (*function*), 17
IsSymmetric(A) (*function*), 17, 42
IsSymplecticMatrix(A) (*function*), 18
IsSymplecticSpace(W) (*function*), 78
IsTotallyIsotropic(V) (*function*), 73
IsTotallySingular(V) (*function*), 73
IsUnit(A) (*function*), 17
IsUnitarySpace(W) (*function*), 79
IsUpperTriangular(A) (*function*), 17, 43
IsZero(A) (*function*), 17, 42
IsZero(u) (*function*), 56

## J

JordanForm(A) (*function*), 21

## K

Kernel(A) (*function*), 15, 44
Kernel(a) (*function*), 68
KernelMatrix(A) (*function*), 15, 44
KMatrixSpace(K, m, n) (*function*), 54
KMatrixSpace(V, F) (*function*), 63
KModule(K, n) (*function*), 54
KModule(V, F) (*function*), 63
KModuleWithBasis(Q) (*function*), 65
KroneckerProduct(A, B) (*function*), 13
KSpace(K, n) (*function*), 53
KSpace(K, n, F) (*function*), 54
KSpace(V) (*function*), 94
KSpace(V, F) (*function*), 63
KSpaceWithBasis(a) (*function*), 65
KSpaceWithBasis(Q) (*function*), 65

## L

Lie Algebra of Type C3 (*example*), 88
LowerTriangularMatrix(Q) (*function*), 4
LowerTriangularMatrix(R, Q) (*function*), 4

## M

Matrix Construction Shortcuts (*example*), 3
Matrix over a Number Field (*example*), 55
Matrix Space over a Quadratic Field (*example*), 54
Matrix(A) (*function*), 4, 42
Matrix(m, n, Q) (*function*), 3
Matrix(n, Q) (*function*), 3
Matrix(Q) (*function*), 3
Matrix(R, A) (*function*), 14
Matrix(R, m, n, Q) (*function*), 1
Matrix(R, n, Q) (*function*), 3
Matrix(R, Q) (*function*), 3
MaximalTotallyIsotropicSubspace(V) (*function*), 73
MaximalTotallySingularSubspace(V) (*function*), 73
MCPolynomials(A) (*function*), 19, 45
MetabolicSpace(V) (*function*), 75
MinimalAndCharacteristicPolynomials(A: parameters) (*function*), 19, 45
MinimalPolynomial(A: parameters) (*function*), 19, 45
Minor(M, I, J) (*function*), 18
Minor(M, i, j) (*function*), 18
Minors(M, r) (*function*), 18
ModularSolution(A, L) (*function*), 46
ModularSolution(A, M) (*function*), 46
Morphism(U, V) (*function*), 60
MultiplyByTranspose(V, A) (*function*), 43
MultiplyByTranspose(v, A) (*function*), 43
MultiplyColumn(~A, c, i) (*procedure*), 11, 41

MultiplyColumn(A, c, i) (*function*), 11, 41
MultiplyRow(~A, c, i) (*procedure*), 11, 41
MultiplyRow(A, c, i) (*function*), 11, 41

## N

Ncols(A) (*function*), 8, 37
Ncols(u) (*function*), 56
NextIsotropicSubspace(V, k) (*function*), 78
Ngens(M) (*function*), 64
NNZEntries(A) (*function*), 8, 37
Norm(u) (*function*), 56
Normalise(u) (*function*), 56
Normalize(u) (*function*), 56
Nrows(A) (*function*), 7, 37
Nrows(u) (*function*), 56
Nullspace of a Random Integer Matrix (*example*), 16
NullSpace(a) (*function*), 68
Nullspace(A) (*function*), 15, 44
NullspaceMatrix(A) (*function*), 15, 44
NullspaceOfTranspose(A) (*function*), 15, 44
NumberOfColumns(A) (*function*), 8, 37
NumberOfColumns(u) (*function*), 56
NumberOfGenerators(M) (*function*), 64
NumberOfIsotropicSubspaces(V, k) (*function*), 78
NumberOfNonZeroEntries(A) (*function*), 8, 37
NumberOfRows(A) (*function*), 7, 37
NumberOfRows(u) (*function*), 56
Numerical Hessenberg Form and Eigenvalues (*example*), 31
Numerical Matrix Decompositions (*example*), 26
Numerical Rank, Kernel, and Pseudoinverse (*example*), 29
Numerical Singular Value Decomposition (*example*), 32
NumericalBidiagonalForm(M) (*function*), 31
NumericalEigenvalues(M) (*function*), 30
NumericalEigenvectors(M, e) (*function*), 31
NumericalHessenbergForm(M) (*function*), 30
NumericalImage(M) (*function*), 28
NumericalInverse(M) (*function*), 26
NumericalIsConsistent(M,w) (*function*), 29
NumericalKernel(M) (*function*), 28
NumericalPseudoinverse(M) (*function*), 29
NumericalRank(M) (*function*), 28
NumericalSchurForm(M) (*function*), 30
NumericalSingularValueDecomposition(M) (*function*), 31
NumericalSolution(M,w) (*function*), 28

## O

Operations on Linear Transformations (*example*), 68
Order(A) (*function*), 25
OrthogonalComplement(V, X : parameters) (*function*), 72
OrthogonalReflection(a) (*function*), 81
OrthogonalSum(V, W) (*function*), 80
OrthogonalTensorProduct(V, W) (*function*), 80
OverDimension(u) (*function*), 64
OverDimension(V) (*function*), 64

## P

Parent(V) (*function*), 64
PermutationMatrix(R, Q) (*function*), 5
PermutationMatrix(R, x) (*function*), 5
Pfaffian(M) (*function*), 19
Pfaffian(M, I, J) (*function*), 19
Pfaffians(M, r) (*function*), 19
PolarSpace(F, a) (*function*), 77
PolarSpace(L) (*function*), 94
PolarSpaceType(V) (*function*), 78
Preserving Bilinear and Sesquilinear Forms (*example*), 92
PrimaryInvariantFactors(A) (*function*), 21
PrimaryRationalForm(A) (*function*), 21
ProjectiveOrder(A) (*function*), 26

## Q

QLDecomposition(M) (*function*), 26
Quadratic Space from a Polynomial (*example*), 80
QuadraticFormMatrix(V) (*function*), 80
QuadraticFormPolynomial(Q) (*function*), 80
QuadraticFormPolynomial(V) (*function*), 80
QuadraticNorm(v) (*function*), 80
QuadraticSpace(f) (*function*), 79
QuadraticSpace(Q) (*function*), 79
quo<V | L> (*constructor*), 61
Quotient by the Golay Code (*example*), 61
Quotient of a Rational Matrix Space (*example*), 62

## R

Radical(V : parameters) (*function*), 72
Random(V) (*function*), 55
RandomGLnZ(n, k, l) (*function*), 7
RandomMatrix(R, m, n) (*function*), 7
RandomPositiveDefiniteSymmetricMatrix(n, M) (*function*), 7
RandomSLnZ(n, k, l) (*function*), 7
RandomSymmetricMatrix(R, n) (*function*), 7
RandomSymmetricMatrix(R, n, M) (*function*), 7
RandomSymplecticMatrix(g, m) (*function*), 7
RandomUnimodularMatrix(n, M) (*function*), 7
Rank(A) (*function*), 18, 44
Rank(a) (*function*), 67
Rational Vector Space of Dimension 6 (*example*), 54
RationalForm(A) (*function*), 21
ReduceVector(W, ~v) (*procedure*), 65
ReduceVector(W, v) (*function*), 65
ReflectionFactors(V, f) (*function*), 81
RemoveColumn(~A, j) (*procedure*), 12, 41

RemoveColumn(A, j) (*function*), 12, 41
RemoveRow(~A, i) (*procedure*), 11, 41
RemoveRow(A, i) (*function*), 11, 41
RemoveRowColumn(~A, i, j) (*procedure*), 12, 41
RemoveRowColumn(A, i, j) (*function*), 12, 41
RemoveZeroRows(~A) (*procedure*), 12, 41
RemoveZeroRows(A) (*function*), 12, 41
RestrictField(V, L) (*function*), 63
ReverseColumns(~A) (*procedure*), 11, 41
ReverseColumns(A) (*function*), 11, 41
ReverseRows(~A) (*procedure*), 11, 40
ReverseRows(A) (*function*), 11, 40
Revised Minus-Type Quadratic Form (*example*), 77
RootSequence(V, f) (*function*), 81
Rotate(~u, k) (*procedure*), 56
Rotate(u, k) (*function*), 56
Row and Column Operations (*example*), 12
RowSequence(A) (*function*), 8
Rowspace(A) (*function*), 44
RowSubmatrix(A, i) (*function*), 10, 40
RowSubmatrix(A, i, k) (*function*), 9, 40
RowSubmatrixRange(A, i, j) (*function*), 10, 40
RowWeight(A, i) (*function*), 37
RowWeights(A) (*function*), 37
RQDecomposition(M) (*function*), 26

## S

Saturation(A) (*function*), 24
Scalar and Structured Matrices (*example*), 5
ScalarMatrix(n, s) (*function*), 4
ScalarMatrix(R, n, s) (*function*), 4
ScalarSparseMatrix(n, s) (*function*), 36
ScalarSparseMatrix(R, n, s) (*function*), 36
Semi-Invariant Forms of a Normal Subgroup (*example*), 93
SemiInvariantBilinearForms(G) (*function*), 93
SemiInvariantQuadraticForms(G) (*function*), 93
SemiInvariantSesquilinearForms(G) (*function*), 93
SemilinearDual(M, mu) (*function*), 92
SemiOrthogonalBasis(V) (*function*), 89
SetEntry(~A, i, j, x) (*procedure*), 38
SetVerbose("SparseMatrix", v) (*procedure*), 46
SiegelTransformation(u, v) (*function*), 81
Similar Hermitian and Skew-Hermitian Spaces (*example*), 85
SimilarityGroup(V) (*function*), 85
SingularRadical(V) (*function*), 72
SmithForm(A) (*function*), 24
Solution(A, Q) (*function*), 16
Solution(A, W) (*function*), 15
SpaceType(V) (*function*), 95
SparseMatrix() (*function*), 35
SparseMatrix(A) (*function*), 42
SparseMatrix(m, n) (*function*), 35
SparseMatrix(m, n, Q) (*function*), 34
SparseMatrix(R) (*function*), 35
SparseMatrix(R, A) (*function*), 42
SparseMatrix(R, m, n) (*function*), 35
SparseMatrix(R, m, n, Q) (*function*), 34
SparseMatrixStructure(R) (*function*), 37
SparseMonomialMatrix(g) (*function*), 37
Special Unitary Group of a Hermitian Form (*example*), 87
Spinor Norm over a General Field (*example*), 81
SpinorNorm(Q, g) (*function*), 81
SpinorNorm(V, f) (*function*), 81
SpinorNormRho(d, g, Q) (*function*), 81
Splitting Off the Radical (*example*), 74
Standard Alternating Form (*example*), 76
Standard Quadratic Form of Minus Type (*example*), 76
StandardAlternatingForm(n,q) (*function*), 75
StandardAlternatingForm(n,R) (*function*), 75
StandardHermitianForm(n,K) (*function*), 76
StandardHermitianForm(n,q) (*function*), 76
StandardPseudoAlternatingForm(n,K) (*function*), 76
StandardPseudoAlternatingForm(n,q) (*function*), 76
StandardQuadraticForm(n, K : parameters) (*function*), 76
StandardQuadraticForm(n, q : parameters) (*function*), 76
StandardSymmetricForm(n, K) (*function*), 77
StandardSymmetricForm(n, q : parameters) (*function*), 77
sub<V | L> (*constructor*), 59
Submatrix(A, I, J) (*function*), 9, 40
Submatrix(A, i, j, p, q) (*function*), 9, 39
SubmatrixRange(A, i, j, r, s) (*function*), 9, 39
Support(A) (*function*), 37
Support(A, i) (*function*), 37
Support(u) (*function*), 56
SwapColumns(~A, i, j) (*procedure*), 11, 40
SwapColumns(A, i, j) (*function*), 11, 40
SwapRows(~A, i, j) (*procedure*), 11, 40
SwapRows(A, i, j) (*function*), 11, 40
Symmetric and Hermitian Products (*example*), 71
SymmetricMatrix(Q) (*function*), 5
SymmetricMatrix(R, Q) (*function*), 5
SymmetricToQuadraticForm(J) (*function*), 80
SymplecticBasis(V,U,W) (*function*), 75
SymplecticSpace(J) (*function*), 78

## T

TensorProduct(U, V) (*function*), 65
TensorProduct(u, v) (*function*), 57
Testing for a Polar Space (*example*), 78
TotallySingularComplement(V, U, W) (*function*), 80
Trace(A) (*function*), 18

Trace(u) (*function*), 57
Trace(u, F) (*function*), 57
TraceOfProduct(A, B) (*function*), 18
Transforming Forms with TransformForm (*example*), 83
Transpose(A) (*function*), 15, 43
Transversal(V, U) (*function*), 65
TrivialPolarSpace(F, n) (*function*), 77
TwistedDual(M, lambda) (*function*), 93
TwistedSemilinearDual(M, lambda, mu) (*function*), 93

## U

u * x (*operation*), 56
U + V (*operation*), 64
u + v (*operation*), 56
u / x (*operation*), 56
u - v (*operation*), 56
U eq V (*operation*), 64
U meet V (*operation*), 64
U meet:= V (*operation*), 64
U ne V (*operation*), 64
U notsubset V (*operation*), 64
U subset V (*operation*), 64
UnitarySpace(J, sigma) (*function*), 79
UpperTriangularMatrix(Q) (*function*), 5
UpperTriangularMatrix(R, Q) (*function*), 5
u[i, j] (*indexed*), 58
u[i, j] := x (*literal*), 58
u[i] (*indexed*), 58
u[i] (*literal*), 58
u[i] := x (*literal*), 58

## V

V ! 0 (*operation*), 55
V ! Q (*operation*), 55
V * A (*operation*), 43
v * A (*operation*), 43
v * a (*operation*), 67
V . i (*operation*), 63
V / U (*operation*), 61
v in V (*operation*), 64
v notin V (*operation*), 64
Vector Arithmetic over a Cyclotomic Field (*example*), 57
Vector over a Finite Field (*example*), 55
Vector Space with a General Form (*example*), 70
Vector(n, Q) (*function*), 7
Vector(Q) (*function*), 7
Vector(R, n, Q) (*function*), 7
Vector(R, Q) (*function*), 7
VectorSpace(K, n) (*function*), 53
VectorSpace(K, n, F) (*function*), 54
VectorSpace(V) (*function*), 94
VectorSpace(V, F) (*function*), 63
VectorSpaceWithBasis(a) (*function*), 65
VectorSpaceWithBasis(Q) (*function*), 65
VerticalJoin(A, B) (*function*), 41
VerticalJoin(Q) (*function*), 13
VerticalJoin(T) (*function*), 13
VerticalJoin(X, Y) (*function*), 13

## W

WallDecomposition(V, f) (*function*), 89
WallForm(V, f) (*function*), 89
WallIsometry(V, I, mu) (*function*), 89
Weight(u) (*function*), 57
WittDecomposition(M, a) (*function*), 75
WittDecomposition(V) (*function*), 75
WittIndex(V) (*function*), 73

## X

x * A (*operation*), 14, 43
x * u (*operation*), 56

## Z

Zero(V) (*function*), 55
ZeroMatrix(R, m, n) (*function*), 4