

Handbook of Magma Functions

Part XIV: Commutative 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

115 Gröbner Bases **1**
- 115.1 Introduction . . . 1
- 115.2 Representation and Monomial Orders . . . 1
- 115.3 Polynomial Rings and Ideals . . . 3
- 115.4 Gröbner Bases . . . 9
- 115.5 Changing Coefficient Ring . . . 29
- 115.6 Changing Monomial Order . . . 30
- 115.7 Hilbert-driven Gröbner Basis Construction . . . 31
- 115.8 SAT solver . . . 32
- 115.9 Bibliography . . . 33

116 Polynomial Ring Ideal Operations **35**
- 116.1 Introduction . . . 35
- 116.2 Creation of Polynomial Rings and their Ideals . . . 35
- 116.3 First Operations on Ideals . . . 36
- 116.4 Computation of Varieties . . . 41
- 116.5 Multiplicities . . . 42
- 116.6 Elimination . . . 43
- 116.7 Variable Extension of Ideals . . . 48
- 116.8 Homogenization of Ideals . . . 48
- 116.9 Extension and Contraction of Ideals . . . 49
- 116.10 Dimension of Ideals . . . 50
- 116.11 Radical and Decomposition of Ideals . . . 50
- 116.12 Normalisation and Noether Normalisation . . . 58
- 116.13 Hilbert Series and Hilbert Polynomial . . . 62
- 116.14 Syzygies . . . 64
- 116.15 Maps between Rings . . . 64
- 116.16 Symmetric Polynomials . . . 65
- 116.17 Functions for Polynomial Algebra and Module Generators . . . 66
- 116.18 Bibliography . . . 68

117 Local Polynomial Rings **70**
- 117.1 Introduction . . . 70
- 117.2 Elements and Local Monomial Orders . . . 70
- 117.3 Local Polynomial Rings and Ideals . . . 71
- 117.4 Standard Bases . . . 73
- 117.5 Operations on Ideals . . . 76
- 117.6 Changing Coefficient Ring . . . 78
- 117.7 Changing Monomial Order . . . 78
- 117.8 Dimension of Ideals . . . 78
- 117.9 Bibliography . . . 79

118 Affine Algebras **80**
- 118.1 Introduction . . . 80
- 118.2 Creation of Affine Algebras . . . 80
- 118.3 Operations on Affine Algebras . . . 81
- 118.4 Maps between Affine Algebras . . . 84

118.5 Finite Dimensional Affine Algebras . . . 84
118.6 Affine Algebras which are Fields . . . 85
118.7 Rings and Fields of Fractions of Affine Algebras . . . 88

119 Modules over Multivariate Rings 91
119.1 Introduction . . . 91
119.2 Module Basics: Embedded and Reduced Modules . . . 91
119.3 Monomial Orders . . . 93
119.4 Basic Creation and Access . . . 94
119.5 The Homomorphism Type . . . 100
119.6 Submodules and Quotient Modules . . . 102
119.7 Basic Module Constructions . . . 105
119.8 Predicates . . . 106
119.9 Module Operations . . . 106
119.10 Changing Ring . . . 108
119.11 Hilbert Series . . . 108
119.12 Free Resolutions . . . 110
119.13 The Hom Module and Ext . . . 121
119.14 Tensor Products and Tor . . . 123
119.15 Cohomology Of Coherent Sheaves . . . 125
119.16 Bibliography . . . 128

120 Invariant Theory 130
120.1 Introduction . . . 130
120.2 Invariant Rings of Finite Groups . . . 131
120.3 Group Actions on Polynomials . . . 131
120.4 Permutation Group Actions on Polynomials . . . 131
120.5 Matrix Group Actions on Polynomials . . . 132
120.6 Algebraic Group Actions on Polynomials . . . 132
120.7 Verbosity . . . 133
120.8 Construction of Invariants of Specified Degree . . . 133
120.9 Construction of G-modules . . . 136
120.10 Molien Series . . . 137
120.11 Primary Invariants . . . 138
120.12 Secondary Invariants . . . 138
120.13 Fundamental Invariants . . . 139
120.14 The Module of an Invariant Ring . . . 144
120.15 The Algebra of an Invariant Ring and Algebraic Relations . . . 145
120.16 Properties of Invariant Rings . . . 147
120.17 Steenrod Operations . . . 148
120.18 Minimalization and Homogeneous Module Testing . . . 149
120.19 Attributes of Invariant Rings and Fields . . . 152
120.20 Invariant Rings of Linear Algebraic Groups . . . 153
120.21 Invariant Fields . . . 159
120.22 Invariants of the Symmetric Group . . . 162
120.23 Bibliography . . . 163

121 Differential Rings 164
121.1 Introduction . . . 164
121.2 Differential Rings and Fields . . . 165
121.3 Structure Operations on Differential Rings . . . 167
121.4 Element Operations on Differential Ring Elements . . . 171
121.5 Changing Related Structures . . . 174
121.6 Ring and Field Extensions . . . 177
121.7 Ideals and Quotient Rings . . . 180
121.8 Wronskian Matrix . . . 181
121.9 Differential Operator Rings . . . 182
121.10 Structure Operations on Differential Operator Rings . . . 183
121.11 Element Operations on Differential Operators . . . 185

121.12 Related Maps . . . 189
121.13 Changing Related Structures . . . 190
121.14 Euclidean Algorithms, GCDs, and LCMs . . . 193
121.15 Related Matrices . . . 195
121.16 Singular Places and Indicial Polynomials . . . 196
121.17 Rational Solutions . . . 198
121.18 Newton Polygons . . . 199
121.19 Symmetric Powers . . . 201
121.20 Differential Operators of Algebraic Functions . . . 201
121.21 Factorisation of Operators over Differential Laurent Series Rings . . . 201
121.22 Bibliography . . . 211

References **212**

Index **214**

Chapter 115

Gröbner Bases

115.1 Introduction

This chapter describes the basics for configuring Magma's powerful *Gröbner basis* machinery, which lies at the heart of computations with ideals and modules over multivariate polynomial rings. Later chapters will describe the many functions and operations available to the user for working with ideal and modules. Gröbner bases were introduced by Bruno Buchberger [Buc65] and at the heart of the theory is the *Buchberger algorithm* which computes a Gröbner basis of an ideal starting from an arbitrary basis (generating set) of the ideal. The two books *Ideals, Varieties and Algorithms* [CLO96] and *Gröbner Bases* [BW93] have also inspired much of the design and presentation of ideals of multivariate polynomial rings in Magma. Since V2.11 (May 2004), Magma also contains a highly optimized implementation of the Faugère F_4 algorithm [Fau99], based on sparse linear algebra techniques, which usually performs dramatically better than the Buchberger algorithm (see [Ste04]). Chapter Multivariate Polynomial Rings deals with the basics of multivariate polynomial rings and their elements (for which there are very many functions), so it is recommended that that chapter be perused before reading this one. Permutation and matrix groups have a natural action on multivariate polynomial rings. This leads to the subject of invariant rings of finite groups, which is covered in Chapter *Invariant Theory*. See also the chapters on affine algebras (Chapter *Affine Algebras*) and on modules over affine algebras (Chapter *Modules over Multivariate Rings*), and the chapter on algebraically closed fields (Chapter Algebraically Closed Fields), which allows one to compute the variety of an ideal over the algebraic closure of the base field.

115.2 Representation and Monomial Orders

Let P be the polynomial ring $R[x_1, \ldots, x_n]$ of rank n over a ring R. A *monomial* (or *power product*) of P is a product of powers of the variables (or indeterminates) of P, that is, an expression of the form $x_1^{e_1} \cdots x_n^{e_n}$ with $e_i \geq 0$ for $1 \leq i \leq n$. Multivariate polynomials in Magma are stored efficiently in distributive form, using arrays of coefficient-monomial pairs, where the coefficient is in the base ring R. The word 'term' will always refer to a coefficient multiplied by a monomial. Monomial orders are of critical importance when dealing with Gröbner bases. Let M be the set of all monomials of P. A *monomial ordering* on M is a total order $<$ on M such that $1 \leq s$ for all $s \in M$, $s \leq t$ implies $su \leq tu$ for all $s, t, u \in M$, and M is a well-ordering (every non-empty subset of M possesses a minimal element w.r.t. $<$). Monomial orders can be naturally specified in terms of *weight vectors*: a vector W from $\mathbb{Q}^n$ with non-negative entries is called a weight vector since it weights a monomial s by the product $s.W$ (defined to be the dot product of the exponent vector of s with W); any sequence of n linearly-independent weight vectors determines a monomial order on M (see the weight order below [subsection *Weight: weight*]). All monomial orderings can in fact be represented in terms of weight vectors. Multivariate polynomial rings are constructed in Magma such that the monomials of any polynomial are sorted with respect to a specified monomial order, with the greatest monomial first. Gröbner basis computations are dramatically affected by the choice of monomial order. Magma provides an extensive choice of monomial orders. Currently, the intrinsic functions `PolynomialRing` (or `PolynomialAlgebra`), `ChangeOrder`, and `VariableExtension` expect a monomial order; it is specified by a string giving the name, optionally followed by extra arguments for that order. We now describe each of the monomial orders available in Magma. We suppose that s and

t are monomials from P which has rank n. Any order on the monomials is then fully defined by just specifying exactly when $s < t$ with respect to that order. In the following, the argument(s) are described for an order as a list of expressions; that means that the expressions (without the parentheses) should be appended to any base arguments when any particular intrinsic function is called which expects a monomial order. See also [CLO96] (Chap. 2, §2) for more details about the first three orders.

115.2.1 Lexicographical: lex

Definition: $s < t$ iff there exists $1 \leq i \leq n$ such that the first $i-1$ exponents of s and t are equal but the i-th exponent of s is less than the i-th exponent of t. The order is specified by the argument (`"lex"`). The order is called "lexicographical" since it orders the monomials as if they were words in a dictionary. The i-th variable is greater than the $(i+1)$-th variable for $1 \leq i < n$ so the first variable is the greatest variable. A Gröbner basis of an ideal with respect to the lexicographical order usually represents the most information about the ideal but can be hard to compute.

115.2.2 Graded Lexicographical: glex

Definition: $s < t$ iff the total degree of s is less than the total degree of t or the total degree of s is equal to the total degree of t and $s < t$ with respect to the lexicographical order. The order is specified by the argument (`"glex"`). The order is called "graded lexicographical" since it first grades the monomials by total degree, and then decides ties by the lexicographical order. The i-th variable is greater than the $(i+1)$-th variable for $1 \leq i < n$ so the first variable is the greatest variable. This order is rarely used because the `grevlex` order below is usually a better degree order (i.e., yields smaller Gröbner bases).

115.2.3 Graded Reverse Lexicographical: grevlex

Definition: $s < t$ iff the total degree of s is less than the total degree of t or the total degree of s is equal to the total degree of t and $s > t$ with respect to the lexicographical order applied to the exponents of s and t in reverse order. The order is specified by the argument (`"grevlex"`). The order is called "graded reverse lexicographical" since it first grades the monomials by total degree, and then decides ties by the negation of the lexicographical order applied to the variables in reverse order. Again, the i-th variable is greater than the $(i+1)$-th variable for $1 \leq i < n$ so the first variable is the greatest variable. A Gröbner basis of an ideal with respect to the graded reverse lexicographical order is usually the easiest to compute so it is recommended that this order be used when just any Gröbner basis for an ideal is desired.

115.2.4 Graded Reverse Lexicographical (Weighted): grevlexw

Definition (given a sequence W of n positive integer weights): $s < t$ iff the total weighted degree d_s of s w.r.t. W is less than the total degree d_t of t w.r.t. W or $d_s = d_t$ and $s > t$ with respect to the lexicographical order applied to the exponents of s and t in reverse order. The order is specified by the arguments (`"grevlexw", W`). The order is called "graded reverse lexicographical (weighted)" since it first grades the monomials by weighted degree w.r.t. W, and then decides ties by the negation of the lexicographical order applied to the variables in reverse order. If $W = [1, 1, \ldots, 1]$, then this order is equal to the `grevlex` order. Again, the i-th variable is greater than the $(i+1)$-th variable for $1 \leq i < n$ so the first variable is the greatest variable. This order is similar to the `grevlex` order, but is useful if an ideal is homogeneous with respect to the grading given by W, since the Gröbner basis of the ideal will tend to be smaller with this order.

115.2.5 Elimination (k): elim

Definition (given k with $1 \leq k \leq n-1$): $s < t$ iff $s_k < t_k$ with respect to the `grevlex` order or $s_k = t_k$ and $s_{k'} < t_{k'}$ with respect to the `grevlex` order where m_k denotes the monomial consisting of the first k exponents of m and $m_{k'}$ denotes the monomial consisting of the last $n-k$ exponents of m (this order is thus the concatenation of two block `grevlex` orders). The order is specified by the arguments (`"elim", k`). The order is called "elimination" since the first k variables are "eliminated": if G is a Gröbner basis of an ideal I of the polynomial ring $K[x_1, \ldots, x_n]$ with respect to this order, then $G \cap K[x_{k+1}, \ldots, x_n]$ is a Gröbner basis of the k-th elimination ideal $I \cap K[x_{k+1}, \ldots, x_n]$. (It is usually easier to compute a Gröbner basis with respect to this order for any k than with respect to the full lexicographical order.)

Again, the i-th variable is greater than the $(i+1)$-th variable for $1 \leq i < n$ so the first variable is the greatest variable.

115.2.6 Elimination List: elim

Definition (given sequences U and V such that U and V contain distinct integers in the range 1 to n and the sum of the lengths of U and V is n and U and V are disjoint): $s < t$ iff $s_U < t_U$ with respect to the `grevlex` order or $s_U = t_U$ and $s_V < t_V$ with respect to the `grevlex` order where m_L denotes the monomial consisting of the exponents of m corresponding to the entries of L in order. The order is specified by the arguments (`"elim", U, V`). V may be omitted if desired so the arguments are just (`"elim", U`); in this case V is chosen to be an appropriate sequence to complement U. The order is called "elimination" since the variables in U are "eliminated". The order of the elements in U and V are significant since the ordering on the variables makes $U[1]$ greatest, then $U[2]$, etc., then $V[1]$, $V[2]$, etc.

115.2.7 Inverse Block: invblock

Definition (given sequences U and V such that U and V contain distinct integers in the range 1 to n and the sum of the lengths of U and V is n and U and V are disjoint): $s < t$ iff $s_V < t_V$ with respect to the `grevlex` order or $s_V = t_V$ and $s_U < t_U$ with respect to the `grevlex` order. The order is specified by the arguments (`"invblock", U, V`). V may be omitted if desired so the arguments are just (`"invblock", U`); in this case V is chosen to be an appropriate sequence to complement U. The order is called "inverse block" since it applies a block ordering on the exponents on V then U which inverts the lists supplied to the elimination list order. Thus this is the same as the elimination order except that the lists U and V are swapped. See [BW93, p. 390] for the motivation for this order.

115.2.8 Univariate: univ

Definition (given i with $1 \leq i \leq n$): $s < t$ iff $s_L < t_L$ with respect to the `grevlex` order or $s_L = t_L$ and the i-th exponent of s is less than the i-th exponent of t, where L is the sequence `[1 .. n]` with i deleted. The order is specified by the arguments (`"univ", i`). The order is called "univariate" since when monomials are compared, any monomial not containing the i-th variable is greater than any monomial containing the i-th variable. Thus all variables but the i-th are "eliminated" so that a Gröbner basis of a zero-dimensional ideal I with this ordering will contain the unique monic generator of the elimination ideal consisting of all the polynomials in I containing the i-th variable alone. The j-th variable is greater than the $(j+1)$-th variable for $1 \leq j < i$ and $i < j \leq n$ and the j-th variable is greater than the i-th variable for any $j \neq i$.

115.2.9 Weight: weight

Definition (given n weight vectors $W_1, \ldots W_n$ from $\mathbb{Q}^n$): $s < t$ iff there exists $1 \leq i \leq n$ such that $s.W_j = t.W_j$ for $1 \leq j < i$ and $s.W_i < t.W_i$. The order is specified by the arguments (`"weight", Q`) where Q is a sequence of n^2 non-negative integers or rationals describing the n weight vectors of length n (in row major order). The n weight vectors must describe a vector space basis of $\mathbb{Q}^n$ (i.e., be linearly-independent), since otherwise this would not yield a total ordering on the monomials. The weight order allows one to specify any possible monomial order; any of the monomial orders mentioned above can be specified by an appropriate choice of weight vectors. However, using the in-built versions of the specialized orders above is much faster than constructing versions of them based on weight vectors. The next section contains an example in which a polynomial ring is constructed with a weight order for the monomials.

115.3 Polynomial Rings and Ideals

115.3.1 Creation of Polynomial Rings and Accessing their Monomial Orders

Multivariate polynomial rings are created from a coefficient ring, the number of variables, and a monomial order. If no order is specified, the monomial order is taken to be the lexicographical order. This section is briefly repeated from Section Creation of Polynomial Rings in the multivariate polynomial rings chapter, so as to show how one can set up the polynomial ring in which to create an ideal. Please note that the

Gröbner basis of an ideal with respect to the lexicographical order is often much more complicated and difficult to compute than the Gröbner basis of the same ideal with respect to other monomial orders (e.g. the `grevlex` order), so it may be preferable to use another order if the Gröbner basis with respect to any order is desired (see also the function *EasyIdeal* below). Yet the lexicographical order is the most natural order and is often the desired order so that is why it is used by default if no specific order is given.

`PolynomialRing(R, n): Rng, RngIntElt -> RngMPol`

`PolynomialAlgebra(R, n): Rng, RngIntElt -> RngMPol`

Global: BoolElt	Default: false

Create a multivariate polynomial ring in $n > 0$ variables over the ring R. The ring is regarded as an R-algebra via the usual identification of elements of R and the constant polynomials. The lexicographical ordering on the monomials is used for this default construction (see next function). By default, a *non-global* polynomial ring will be returned; if the parameter `Global` is set to `true`, then the unique global polynomial ring over R with n variables will be returned. This may be useful in some contexts, but a non-global result is returned by default since one often wishes to have several rings with the same numbers of variables but with different variable names (and create mappings between them, for example). Explicit coercion is always allowed between polynomial rings having the same number of variables (and suitable base rings), whether they are global or not, and the coercion maps the i-variable of one ring to the i-th variable of the other ring.

`PolynomialRing(R, n, order): Rng, RngIntElt, MonStgElt, ... -> RngMPol`

`PolynomialAlgebra(R, n, order): Rng, RngIntElt, MonStgElt, ... -> RngMPol`

Create a multivariate polynomial ring in $n > 0$ variables over the ring R with the given order order on the monomials. See the section on monomial orders for the valid values for the argument order.

`PolynomialRing(R, n, T): Rng, RngIntElt, Tup -> RngMPol`

`PolynomialAlgebra(R, n, T): Rng, RngIntElt, Tup -> RngMPol`

Create a multivariate polynomial ring in $n > 0$ variables over the ring R with the order given by the tuple T on the monomials. T must be a tuple whose components match the valid arguments for the monomial orders in Section *Representation and Monomial Orders* (or a tuple returned by the following function *MonomialOrder*).

`MonomialOrder(P): RngMPol -> Tup`

Given a polynomial ring P (or an ideal thereof), return a description of the monomial order of P. This is returned as a tuple which matches the relevant arguments listed for each possible order in Section *Representation and Monomial Orders*, so may be passed as the third argument to the function *PolynomialRing* above.

`MonomialOrderWeightVectors(P): RngMPol -> [ [ FldRatElt ] ]`

Given a polynomial ring P of rank n (or an ideal thereof), return the weight vectors of the underlying monomial order as a sequence of n sequences of n rationals. See, for example, [CLO98, p. 153] for more information.

`SetSparseMonomialMinRank(R): RngIntElt`

(Procedure.) Set the limit R so that when a multivariate polynomial ring P is created with at least R variables (i.e., has rank at least R), the sparse monomial representation is used for the monomials ($R = 0$ means that the sparse representation will never be used). The initial value of R is currently 20. The sparse representation saves time and memory for computations where there are large numbers of variables and the monomials are relatively sparse (have small degree).

`GetSparseMonomialMinRank() -> RngIntElt`

Return the sparse monomial rank (as set by `SetSparseMonomialMinRank`).

Example: Polynomial Rings with Different Orders `(ex-7d923a)`

We show how one can construct different polynomial rings with different orders.

```
> Z := IntegerRing();
> // Construct polynomial ring with DEFAULT lex order
> P<a,b,c,d> := PolynomialRing(Z, 4);
> MonomialOrder(P);
<"lex">
> MonomialOrderWeightVectors(P);
[
    [ 1, 0, 0, 0 ],
    [ 0, 1, 0, 0 ],
    [ 0, 0, 1, 0 ],
    [ 0, 0, 0, 1 ]
]
> // Construct polynomial ring with grevlex order
> P<a,b,c,d> := PolynomialRing(Z, 4, "grevlex");
> MonomialOrder(P);
<"grevlex">
> MonomialOrderWeightVectors(P);
[
    [ 1, 1, 1, 1 ],
    [ 1, 1, 1, 0 ],
    [ 1, 1, 0, 0 ],
    [ 1, 0, 0, 0 ]
]
> // Construct polynomial ring with block elimination and a > d > b > c
> P<a,b,c,d> := PolynomialRing(Z, 4, "elim", [1, 4], [2, 3]);
> MonomialOrder(P);
<"elim", [ 1, 4 ], [ 2, 3 ]>
> MonomialOrderWeightVectors(P);
[
    [ 1, 0, 0, 1 ],
    [ 1, 0, 0, 0 ],
    [ 0, 1, 1, 0 ],
    [ 0, 1, 0, 0 ]
]
> a + b + c + d;
a + d + b + c
> a + d^10 + b + c^10;
d^10 + a + c^10 + b
> a + d^10 + b + c;
d^10 + a + b + c
> // Construct polynomial ring with weight order and x > y > z
> P<x, y, z> := PolynomialRing(Z, 3, "weight", [100,10,1, 1,10,100, 1,1,1]);
> MonomialOrder(P);
<"weight", [ 100, 10, 1, 1, 10, 100, 1, 1, 1 ]>
> MonomialOrderWeightVectors(P);
[
    [ 100, 10, 1 ],
    [ 1, 10, 100 ],
    [ 1, 1, 1 ]
]
> x + y + z;
x + y + z
> (x+y^2+z^3)^4;
x^4 + 4*x^3*y^2 + 4*x^3*z^3 + 6*x^2*y^4 + 12*x^2*y^2*z^3 +
    6*x^2*z^6 + 4*x*y^6 + 12*x*y^4*z^3 + 12*x*y^2*z^6 +
    4*x*z^9 + y^8 + 4*y^6*z^3 + 6*y^4*z^6 +
    4*y^2*z^9 + z^12
```

115.3.2 Creation of Graded Polynomial Rings

It is possible within Magma to assign weights to the variables of a multivariate polynomial ring. This means that monomials of the ring then have a *weighted degree* with respect to the weights of the variables. Such a multivariate polynomial ring is called *graded* or *weighted*. A polynomial of the ring whose monomials all have the same weighted degree is called *homogeneous*. The polynomial ring can be decomposed as the direct sum of graded homogeneous components. Suppose a polynomial ring P has n variables $x_1, \ldots, x_n$ and the weights for the variables are $d_1, \ldots, d_n$ respectively. Then for a monomial $m = x_1^{e_1} \cdots x_n^{e_n}$ of P (with $e_i \geq 0$ for $1 \leq i \leq n$), the *weighted degree* of m is defined to be $\sum_{i=1}^{n} e_i d_i$. A polynomial ring created without a specific weighting (using the default version of the *PolynomialRing* function or similar) has weight 1 for each variable so the weighted degree coincides with the total degree. The following functions allow one to create and operate on elements of polynomial rings with specific weights for the variables.

`PolynomialRing(R, Q): Rng, [ RngIntElt ] -> RngMPol`

`PolynomialAlgebra(R, Q): Rng, [ RngIntElt ] -> RngMPol`

Given a ring R and a non-empty sequence Q of positive integers, create a multivariate polynomial ring in $n = \#Q$ variables over the ring R with the weighted degree of the i-th variable set to be $Q[i]$ for each i. The rank n of the polynomial is determined by the length of the sequence Q. (The angle bracket notation can be used to assign names to the variables, just like in the usual invocation of the `PolynomialRing` function.) As of V2.15, the default monomial order chosen is the `grevlexw` order with weights given by Q, since the Gröbner basis of an ideal w.r.t. this order tends to be smaller if the ideal is homogeneous w.r.t. the grading.

`Grading(P): RngMPol -> [ RngIntElt ]`

`VariableWeights(P): RngMPol -> [ RngIntElt ]`

Given a graded polynomial ring P (or an ideal thereof), return the variable weights of P as a sequence of n integers where n is the rank of P. If P was constructed without specific weights, the sequence containing n copies of the integer 1 is returned.

`HomogeneousWeightsSearch(S): [ RngMPol ] -> BoolElt, [ RngIntElt ]`

Given a set or sequence S of polynomials from $R[x_1, \ldots, x_n]$, try to find a sequence of positive weights W w.r.t. which S is homogeneous. If found, this function returns `true` and the sequence W of n weights; otherwise it returns `false`. If the parameter `Approximate` is set to `true` and no exact grading is found, the function attempts to find an sequence W of positive weights w.r.t. which S is approximately homogeneous (with the constant terms the polynomials in S ignored).

115.3.3 Element Operations Using the Grading

`Degree(f): RngMPolElt -> RngIntElt`

`WeightedDegree(f): RngMPolElt -> RngIntElt`

Given a polynomial f of the graded polynomial ring P, this function returns the weighted degree of f, which is the maximum of the weighted degrees of all monomials that occur in f. The weighted degree of a monomial m depends on the weights assigned to the variables of the polynomial ring P — see the introduction of this section for details. Note that this is different from the natural total degree of f which ignores any weights.

`LeadingWeightedDegree(f): RngMPolElt -> RngIntElt`

Given a polynomial f of the graded polynomial ring P, this function returns the leading weighted degree of f, which is the weighted degree of the leading monomial of f. The weighted degree of a monomial m depends on the weights assigned to the variables of the polynomial ring P — see the introduction of this section for details.

`IsHomogeneous(f): RngMPolElt -> BoolElt`

Given a polynomial f of the graded polynomial ring P, this function returns whether f is homogeneous with respect to the weights on the variables of P (i.e., whether the weighted degrees of the monomials of f are all equal).

`HomogeneousComponent(f, d): RngMPolElt, RngIntElt -> RngMPolElt`

Given a polynomial f of the graded polynomial ring P, this function returns the *weighted* degree-d homogeneous component of f which is the sum of all the terms of f whose monomials have weighted degree d. d must be greater than or equal to 0. If f has no terms of weighted degree d, then the result is 0.

`HomogeneousComponents(f): RngMPolElt -> [ RngMPolElt ]`

Given a polynomial f of the graded polynomial ring P, this function returns the *weighted* degree-d homogeneous component of f which is the sum of all the terms of f whose monomials have weighted degree d. d must be greater than or equal to 0. If f has no terms of weighted degree d, then the result is 0.

`MonomialsOfDegree(P, d): RngMPolElt, RngIntElt -> {@ RngMPolElt @}`

Given a polynomial ring P and a non-negative integer d, return an indexed set consisting of all monomials in P with total degree d. If P is graded, the grading is ignored.

`MonomialsOfWeightedDegree(P, d): RngMPolElt, RngIntElt -> {@ RngMPolElt @}`

Given a graded polynomial ring P and a non-negative integer d, return an indexed set consisting of all monomials in P with weighted degree d. If P has the trivial grading, then this function is equivalent to the function *MonomialsOfDegree*.

Example: Operations in a Graded Polynomial Ring (ex-fffb08)

We create a simple graded polynomial ring and perform various simple operations on it.

```
> P<x, y, z> := PolynomialRing(RationalField(), [1, 2, 4]);
> P;
Graded Polynomial ring of rank 3 over Rational Field
Order: Grevlex with weights [1, 2, 4]
Variables: x, y, z
Variable weights: [1, 2, 4]
> VariableWeights(P);
[ 1, 2, 4 ]
> Degree(x);
1
> Degree(y);
2
> Degree(z);
4
> Degree(x^2*y*z^3); // Weighted total degree
16
> TotalDegree(x^2*y*z^3); // Natural total degree
6
> IsHomogeneous(x);
true
> IsHomogeneous(x + y);
false
> IsHomogeneous(x^2 + y);
true
> I := ideal<P | x^2*y + z, (x^4 + z)^2, y^2 + z>;
> IsHomogeneous(I);
true
> MonomialsOfDegree(P, 4);
{@
    x^4,
    x^3*y,
    x^3*z,
    x^2*y^2,
    x^2*y*z,
```

(continues on next page)

(continued from previous page)

```
    x^2*z^2,
    x*y^3,
    x*y^2*z,
    x*y*z^2,
    x*z^3,
    y^4,
    y^3*z,
    y^2*z^2,
    y*z^3,
    z^4
@}
> MonomialsOfWeightedDegree(P, 4);
{@
    x^4,
    x^2*y,
    y^2,
    z
@}
```

115.3.4 Creation of Ideals and Accessing their Bases

Within the general context of ideals of polynomial rings, the term "basis" will refer to an *ordered* sequence of polynomials which generate an ideal. (Thus a basis can contain duplicates and zero elements so is not like a basis of a vector space.) One normally creates an ideal by the *ideal<P | L>* constructor or *Ideal* function, described below. But it is also possible to create an ideal with a specific basis U and then find the coordinates of polynomials from the polynomial ring with respect to U (see the function *Coordinates* below). This is done by specifying a *fixed basis* with the *IdealWithFixedBasis* intrinsic function. In this case, when Magma computes the Gröbner basis of the ideal (see below), extra information is stored so that polynomials of the ideal can be rewritten in terms of the original fixed basis. However, the use of this feature makes the Gröbner basis computation much more expensive so an ideal should usually **not** be created with a fixed basis.

`ideal<P | L>: RngMPol, List -> RngMPol`

Given a multivariate polynomial ring P, return the ideal of P generated by the elements of P specified by the list L. Each term of the list L must be an expression defining an object of one of the following types:

(a) An element of P;

(b) A set or sequence of elements of P;

(c) An ideal of P;

(d) A set or sequence of ideals of P.

`Ideal(B): [ RngMPolElt ] -> RngMPol`

`Ideal(B): { RngMPolElt } -> RngMPol`

Given a set or sequence B of polynomials from a polynomial ring P, return the ideal of P generated by the elements of B with the given basis B. This is equivalent to the above `ideal` constructor, but is more convenient when one simply has a set or sequence of polynomials.

`Ideal(f): RngMPolElt -> RngMPol`

Given a polynomial f from a polynomial ring P, return the principal ideal of P generated by f.

`IdealWithFixedBasis(B): [ RngMPolElt ] -> RngMPol`

Given a sequence B of polynomials from a polynomial ring P, return the ideal of P generated by the elements of B with the given fixed basis B. When the function *Coordinates* is called, its result will be with respect to the entries of B instead of the Gröbner basis of I. **WARNING:** this

function should *only* be used when it is desired to express polynomials of the ideal in terms of the elements of B, as the computation of the Gröbner basis in this case is *very* expensive, so it should be avoided if these expressions are not wanted.

`Basis(I): RngMPol -> [ RngMPolElt ]`

Given an ideal I, return the current basis of I. If I has a fixed basis, that is returned; otherwise the current basis of I (whether it has been converted to a Gröbner basis or not – see below) is returned.

`BasisElement(I, i): RngMPol, RngIntElt -> RngMPolElt`

Given an ideal I together with an integer i, return the i-th element of the current basis of I. This the same as `Basis(I)[i]`.

115.4 Gröbner Bases

Computation in ideals of multivariate polynomial rings is possible because of the construction of Gröbner bases of such ideals. In Magma, it is possible to create ideals and compute their Gröbner bases for polynomial rings defined not only over fields but also over general Euclidean rings. Different monomial orderings give different Gröbner bases for a fixed ideal. When an ideal I is created from a polynomial ring P or another ideal J, then the monomial order of I is taken to be the monomial order of P or J. Ideals can only be compatible if they have the same monomial order.

115.4.1 Gröbner Bases over Fields

Gröbner bases of ideals defined over fields have been studied for some time now, and there is a large literature concerning them. Let I be an ideal of a polynomial ring defined over a field, and choose a fixed monomial ordering on the ring. A finite subset G of $I \setminus \{0\}$ is called a Gröbner basis for I if, for every nonzero $f \in I$, there exists $g \in G$ such that the leading monomial of g divides the leading monomial of f. Every Gröbner basis for I is a set of generators of I [AL94, Cor. 1.6.3]. A Gröbner basis is called **reduced** if each polynomial in it is monic and, for every monomial of each polynomial in the basis, that monomial is not divisible by the leading monomial of any other polynomial in the basis (equivalently, each leading monomial does not divide any monomial in any of the other polynomials). For a given fixed monomial ordering, every nonzero ideal of a polynomial ring over a field possesses a **unique** reduced Gröbner basis [AL94, Thm. 1.8.7]. This unique Gröbner basis (with respect to the order defined by the user) is the Gröbner basis computed by Magma, and is represented by Magma as a sequence of polynomials sorted by the monomial ordering (with decreasing order). This sorting is unique because any two polynomials in a reduced Gröbner basis have distinct leading monomials. This Gröbner basis will be computed automatically when needed by Magma. Before this happens, an ideal will usually possess a basis which is not a Gröbner basis, but when the Gröbner basis is computed, the original basis will be discarded and will be replaced by the unique Gröbner basis. See the procedure *Groebner* below for details on the algorithms available.

115.4.2 Gröbner Bases over Euclidean Rings

Since V2.8 (July 2001), Magma provides facilities for computing with Gröbner bases of ideals of polynomial rings over Euclidean rings (including the important case of the integer ring $\mathbb{Z}$). Such Gröbner bases are computed in Magma by an extension, due to Allan Steel (unpublished), of Jean-Charles Faugère's F_4 algorithm [Fau99], which uses sparse linear algebra.

The current Euclidean rings in Magma supported are: the integer ring $\mathbb{Z}$, the integer residue class rings $\mathbb{Z}_m$, the univariate polynomial rings $K[x]$ over any field K, Galois rings, p-adic quotient rings, and discrete valuation rings. All of these have a Euclidean division algorithm with unique quotient/remainder output.

We first outline some of the things which are peculiar to Gröbner bases defined over a Euclidean ring. Let I be an ideal of a polynomial ring defined over a Euclidean ring R. A subset G of I is called a *Gröbner basis* for I in Magma if, for every $f \in I$, there exists a $g \in G$ such that the leading **term** of g divides the leading **term** of f. Recall that "leading term" here means the leading coefficient times the leading monomial, so the leading coefficient of g must divide the leading coefficient of f in the base ring

R. If R were a field, then obviously the leading coefficients would be insignificant and the Gröbner basis elements could be normalized (made monic) to yield an equivalent Gröbner basis. But if R is not a field, the leading coefficients are quite significant. For example, over the ring $\mathbb{Z}$, the set $\{x^2, 2x\}$ is a Gröbner basis and the polynomial x^2 is not redundant since 2 does not divide 1, but over $\mathbb{Q}$, the polynomial x^2 would be redundant.

Note that the definition here for a Gröbner basis in Magma is actually what some authors (e.g., [AL94, Def. 4.5.6]) call a *strong* Gröbner basis. *Weak* Gröbner bases have also been defined, but strong Gröbner bases satisfy stronger conditions, yield a simple effective normal form algorithm, provide more information about the ideal, are easier to get into a unique form, and are no more difficult to compute using the algorithm implemented in Magma. Thus Magma **always** computes a strong Gröbner basis, so the distinction between weak and strong is ignored. Magma also effectively computes a D-Gröbner basis as defined in [BW93] (Def. 10.4, Table 10.1), although Magma also allows Euclidean rings which are not integral domains (i.e., which have zero divisors).

Over Euclidean rings, a basis is called *reduced* if each polynomial in it is normalized and if, for every term $c \cdot s$ of every polynomial in the basis (where c is the coefficient and s is the monomial), then if some other polynomial in the basis has leading term $d \cdot t$, with t dividing s, then the Euclidean quotient of c by d must be zero (the remainder will be non-zero of course). Informally, this means that each polynomial is reduced modulo all the other polynomials, where each coefficient must be reduced modulo all other appropriate leading coefficients. As an example, suppose $f_1 = x^2 + 14xy$ and $f_2 = 5y + 9$ are in $\mathbb{Z}[x, y]$. Then $\{f_1, f_2\}$ is not reduced, since the second term of f_1 can be reduced by f_2 (y divides xy and the Euclidean quotient of 14 by 5 is 2, with remainder 4). But if we were to replace f_1 by $f_1 - 2xf_2 = x^2 + 4xy - 18x$, then $\{f_1, f_2\}$ would now be reduced.

Just as for fields, every ideal of a polynomial ring over a Euclidean ring possesses a **unique** reduced Gröbner basis (with respect to some fixed monomial ordering), which can be proved by adapting the proof over fields. This unique Gröbner basis (with respect to the order defined by the user) is the Gröbner basis computed by Magma, and is represented by Magma as a sequence of polynomials sorted by the monomial ordering (with decreasing order). This sorting is unique because any two polynomials in a reduced Gröbner basis have distinct leading monomials. This Gröbner basis will be computed automatically when needed by Magma. Before this happens, an ideal will usually possess a basis which is not a Gröbner basis; but when the Gröbner basis is computed, the original basis will be discarded and will be replaced by the unique Gröbner basis.

The uniqueness of the Gröbner basis also ensures that the normal form of an element with respect to an ideal for a fixed monomial order is always unique. All of this holds even for Euclidean rings which have zero divisors.

See the examples below for illustrations of the points made above, and also how one can effectively compute with Gröbner bases of ideals defined over some rings which are not even Euclidean.

115.4.3 Construction of Gröbner Bases

The following functions and procedures allow one to construct Gröbner bases. Note that a Gröbner basis for an ideal will be automatically generated when necessary; the `Groebner` procedure below simply allows control of the algorithms used to compute the Gröbner basis. **NOTE:** Magma applies a special monomial representation and a special variant of the F_4 algorithm if the ideal I is defined over $\mathbf{F}_2$ and the polynomials ${x_i}^2 + x_i$ for all i are present in the input basis of the ideal I. So if one wishes to solve a system of equations over $\mathbf{F}_2$, then one should include these polynomials in the input basis (they can be at any place and in any order; as long as there is at least one copy of ${x_i}^2 + x_i$ present for each i). Alternatively (since V2.15), one can create a boolean polynomial ring (via the function *BooleanPolynomialRing* below) and construct the ideal within this. See also Example *Solving a Boolean Polynomial System* below.

`Groebner(I: parameters): RngMPol`

```
Al                : MonStgElt                    Default: "Default"
Faugere           : BoolElt                      Default: true
Dense             : BoolElt                      Default:
```

(continues on next page)

(continued from previous page)

```
HomogeneousWeights: BoolElt                    Default: true
Homogenize        : BoolElt                    Default: true
DegreeStart       : RngIntElt                  Default: true
GlobalModular     : BoolElt                    Default: true
AllPairs          : BoolElt                    Default: false
PairsLimit        : BoolElt                    Default: 0
ReversePairs      : BoolElt                    Default: false
HFE               : BoolElt                    Default: false
Boolean           : BoolElt                    Default: false
DynamicStrategy   : BoolElt                    Default: false
Nthreads          : RngIntElt                  Default: 1
ReduceInitial     : BoolElt                    Default: true
RemoveRedundant   : BoolElt                    Default: true
ReduceByNew       : BoolElt                    Default: true
SigmaEpsilon      : FldRatElt                  Default: 1/2
TauEpsilon        : FldRatElt                  Default: 1/n
SigmaVectors      : RngIntElt                  Default: n
TauVectors        : RngIntElt                  Default: Ceiling(n/2)
```

(Procedure.) Explicitly force a Gröbner basis (GB) for the ideal I to be constructed. This procedure is normally not necessary, as Magma will automatically compute the GB when needed, but it does allow one to control how the GB is constructed by various parameters. By default, the parameters are set to default values which tend to work best for the particular kinds of inputs which are given, but there exist many inputs for which setting at least one of the parameters to a non-default value will lead to a dramatic improvement. (A general strategy for the computation of GBs is very difficult to design.) If I is defined over a Euclidean ring, then Magma always uses the extension of the Faugère algorithm directly, and of the parameters given below, only `Homogenize` is applicable. So the rest of this description assumes that I is defined over a field. We call a GB algorithm **direct** if it takes the initial basis of the ideal I (with no structure) and computes the unique minimal reduced GB of I with respect to some monomial order. Since V2.11 (May 2004), Magma has two direct algorithms for computing GBs over fields:

(1) The Faugère F_4 algorithm [Fau99], which works by specialized sparse linear algebra and is applicable to ideals defined over a finite field or the rational field;

(2) The Buchberger algorithm [CLO96] (Chap. 2, §7) for ideals defined over any field.

Both direct algorithms use the advanced criteria for eliminating useless pairs in [Mol88]. Magma also uses two **order change** algorithms which both change the GB of an ideal with respect to one monomial order to the GB with respect to another monomial order:

(1) The FGLM algorithm [FGLM93], which works by efficient linear algebra and is only applicable if I is zero-dimensional;

(2) The Gröbner Walk algorithm [CKM97].

This parameter affects the main strategy:

The parameter `Al` may be set to one of: `"Default"`, `"Direct"`, `"FGLM"`, or `"Walk"`. The value `"Direct"` specifies that Magma should compute the GB of I (with respect to the order of I) by a direct algorithm alone, so that an order-conversion algorithm is not used (the parameter `Faugere` below controls which direct algorithm is used). The alternative strategy is to compute the GB first with respect to an "easy" order, and then to convert this to the GB with respect to the order of I. Setting `Al` to the values `"FGLM"` or `"Walk"` will cause this strategy to be used, where the order change algorithm will be the FGLM algorithm or Gröbner Walk algorithm, respectively. If no algorithm is specified, or if `"Default"` is specified, an appropriate strategy is chosen by Magma, which is usually the FGLM method if the ideal is zero-dimensional and over a finite field or the rational field, and the Walk method otherwise.

The following parameters affect the direct algorithms:

If the parameter `Faugere` is set to `true`, then the Faugère F_4 algorithm will be used (if the field is a finite field or the rational field); otherwise the Buchberger algorithm is used. The current implementation of the Faugère algorithm is usually very much faster than the Buchberger algorithm and usually does not take much more memory, so that it is why it is now selected by default. However, there may be examples for which it may be more desirable to use the Buchberger algorithm (particularly to save some memory). Since V2.20, the `Dense` parameter controls the dense variant of F_4; see the next subsection. Since V2.12, if the input basis is not homogeneous, then Magma first attempts to find a weight vector W with respect to which the ideal is homogeneous; if such a W is found, then the "easy" order used internally for the direct algorithm (accessed by *`EasyIdeal`*) is taken to be the `grevlexw` order with respect to W (see subsection *Graded Reverse Lexicographical (Weighted): grevlexw*), since the GB is likely to be smaller with respect to this order. The selection of such an order may be suppressed by setting the parameter `HomogeneousWeights` to `false`. If no appropriate `grevlexw` order is used, then setting `Homogenization` to `true` specifies that the ideal should first be *homogenized*: a GB of the homogenization of the ideal is computed and then the homogenization variable is removed and the final basis reduced. This parameter has the default value of `true` over the rational field and `false` over all other fields, since most computations are improved by these defaults. If the parameter `DegreeStart` is set to an integer d, then any S-polynomial pairs of degree less than d will be ignored. When computing the GB of an ideal I defined over $K = \mathbb{Q}$ or an algebraic number field $K = \mathbb{Q}(\alpha)$, Magma by default uses a Monte-Carlo 'global modular' algorithm where the complete GB of I is computed modulo successive primes and the GB of I over K is reconstructed from these modular GBs. The probability of incorrectness for this algorithm is less than 1 in 2^{140} (approx 1.39×10^{42}). This algorithm can be also turned off by setting the parameter `GlobalModular` to `false`, so that the computation will revert to using the previous 'step by step' modular F_4 algorithm where there is only one full GB computation and the GB polynomials are constructed over K in each matrix step (see also *`SetGBGlobalModular`* below, which has the same effect).

The following parameters affect the Faugère F_4 algorithm:

By default, the Faugère F_4 algorithm includes all pairs of the next degree at each step (see [Fau99, Sec.2.5]), since this usually produces the best performance. However, setting the parameter `AllPairs` to `true` will cause the algorithm to include *all* pairs currently in the queue at each new step; this generally makes the matrix larger and is usually less efficient, but for some inputs (e.g., inhomogeneous ideals where there are only a small number of pairs for each degree at each step) this option may yield a significant improvement. Alternatively, setting the parameter `PairsLimit` to a positive integer n will cause the algorithm to include at most n pairs from the queue at each step; this will usually make the matrix smaller, thus saving memory, but will often also make the running time longer. Setting also the parameter `ReversePairs` to `true` will reverse the list of pairs of the current degree from which the restricted set of pairs is taken: this may help a lot for certain types of input, since this may lead to new polynomials of lower degree being found more quickly. (If there is no pairs limit, then the value of `ReversePairs` is irrelevant since all pairs of the current degree are taken at each step.) If the input basis is an HFE system over $\mathbf{F}_2$ such that the secret degree d is less than or equal to 127, then one should set the `HFE` parameter to `true`. In this case, Magma can apply various optimizations which save memory and time (only pairs of degree of most 4 are considered, as this is sufficient for systems for which $d \leq 127$). Since V2.18, if the base ring is the finite field $\mathbf{F}_p$, where p is a prime with $2 < p < 2^{23.5}$, then a multi-threaded version of the algorithm is available if POSIX threads are enabled in the current Magma version. In this case, setting the parameter `Nthreads` to a positive integer n will cause the F_4 algorithm to use n threads within the linear algebra phase of each step. One can alternatively use the procedure `SetNthreads(n): RngIntElt` to set the global number of threads to a value n so that n threads are always used by default in this algorithm (unless overridden by the `Nthreads` parameter). If the parameter `DynamicStrategy` is set to `true` and the algorithm is running in the dense mode only, then a strategy is used to try to speed up the computation for certain inputs in the case that when a step with a pairs limit has non-trivial reduction to zero, then most or all the remaining pairs of the current degree will probably reduce to zero.

The following parameters affect the Buchberger algorithm:

Setting `ReduceInitial` to `true` specifies that the basis of the ideal should be first reduced (see

the function `Reduce`) before any S-polynomial pairs are considered. Setting `RemoveRedundant` to `true` specifies that redundant polynomials in the input (which reduce to zero with respect to the other polynomials) should first be removed. Setting `ReduceByNew` to `true` specifies that when a new polynomial f is inserted into the current GB being constructed, the current basis should be reduced by f (thus the basis stays close to being fully reduced throughout the algorithm). Each of these control parameters usually have the default values of `true` (it depends on the coefficient ring).

The following parameters affect the Walk algorithm:

The parameters `SigmaEpsilon` and `TauEpsilon` control the factor ϵ which is used in the Walk algorithm to perturb the initial weight vector σ and the final weight vector τ respectively. The parameters `SigmaVectors` and `TauVectors` determine how many weight vectors of the initial and final orders are used to perturb the initial weight vector σ and the final weight vector τ respectively. By default, the ϵ factor and number of weight vectors for σ are determined dynamically to be "optimal", while the ϵ factor for τ is taken to be $1/n$ and the number of weight vectors for τ is taken to be $\lceil n/2 \rceil$, where n is the rank of I.

`GroebnerBasis(I: parameters): RngMPol -> [ RngMPolElt ], [ RngIntElt ]`

Given an ideal I, force the Gröbner basis of I to be computed, and then return that as a sorted sequence of polynomials. The second return value is an integer sequence D which gives the sequence of step degrees encountered in each step of the F_4 algorithm, if that is used to compute the 'easy' GB before a conversion (see *EasyIdeal*) or in a direct GB computation (without order change), when such a use of the algorithm F_4 is non-trivial. The parameters are the same as those for the procedure *Groebner*. See also the function *GroebnerBasis*`(S,d)` below, which creates a truncated degree-d Gröbner basis.

`GroebnerBasis(S: parameters): [ RngMPolElt ] -> [ RngMPolElt ], [ RngIntElt ], []`

`GroebnerBasis(S: parameters): { RngMPolElt } -> [ RngMPolElt ]`

```
ReturnDenominators: BoolElt                    Default: false:1
```

Given a set or sequence S of polynomials, return the unique Gröbner basis of the ideal generated by S as a sorted sequence of polynomials. This function is useful for computing Gröbner bases without the need to construct ideals. The second return value is the integer sequence D of F_4 step degrees, just as in the previous function (see above). The parameters are the same as those for the procedure *Groebner*, except that there is also the parameter `ReturnDenominators`: if this is set to `true` and the input basis is defined over $\mathbb{Q}$, then there is third return value S which is an integer sequence containing the denominators which arise in the algorithm (by which coefficients are possibly divided). As a result, for each prime p such that the leading monomials of the GB of the original ideal reduced modulo p do not equal the leading monomials of the GB over $\mathbb{Q}$, it must be that p divides an element of S. Thus the sequence S can be used to determine such 'bad primes' for the ideal. See also the function *GroebnerBasis*`(S,d)` below, which creates a truncated degree-d Gröbner basis.

`GroebnerBasisUnreduced(S: parameters): [ RngMPolElt ] -> [ RngMPolElt ]`

`GroebnerBasisUnreduced(S: parameters): { RngMPolElt } -> [ RngMPolElt ]`

```
Homogenize   : BoolElt                 Default: true
ReduceInitial: BoolElt                 Default: true
ReduceByNew  : BoolElt                 Default: true
```

Given a set or sequence S of polynomials, return an unreduced Gröbner basis of the ideal generated by S as a sorted sequence. This function is useful for computing Gröbner bases without the need to construct ideals and when the reduction of the Gröbner basis is very expensive. The parameters behave the same as for the procedure *Groebner*.

`GroebnerBasis(S, d: parameters): [ RngMPol ], RngInt -> RngMPolElt`

Given a set or sequence S of polynomials, return the degree-d Gröbner basis of the ideal generated

by S, which is the truncated Gröbner basis obtained by ignoring S-polynomial pairs whose total degree is greater than d.

If the ideal is homogeneous, then it is guaranteed that the result G_d is equal to the set of all polynomials in the full Gröbner basis of the ideal whose total degree is less than or equal to d, and thus a polynomial whose total degree is less than or equal to d is in the ideal iff its normal form with respect to the degree-d Gröbner basis G_d is zero. But if the ideal is not homogeneous, these last properties may not hold, but it may be still useful to construct the truncated basis.

The parameters are the same as those for the procedure *Groebner*. See the section on graded polynomial rings below for an example. See also [BW93, section 10.2], for further discussion.

`SetGBGlobalModular(f): BoolElt`

(Procedure.) Set to f a global flag which controls whether the global modular algorithm may be used when computing Groebner bases over $\mathbb{Q}$ or an algebraic number field. This is equivalent to the `GlobalModular` parameter to the procedure `Groebner` above, but affects all calls to the internal GB machinery. The related function `GetGBGlobalModular` gives the current value of the flag.

`SetFaugereModular(f): BoolElt`

(Procedure.) Set to f a global flag which controls whether modular methods may be used in the Faugere F4 algorithm when computing Groebner bases over $\mathbb{Q}$ or an algebraic number field. The related function `GetFaugereModular` gives the current value of the flag.

115.4.4 The Dense Variant of the F_4 algorithm

Since V2.20, Magma includes a new dense variant of the F_4 algorithm. The dense variant is **currently only practically applicable** to input systems over a finite field where the input polynomials are considered "dense"; that is, if the input polynomials are written as a matrix with columns labelled by the monomials, then the input matrix should be dense. Equivalently: if the field size is q and the set of all monomials occurring in the input has size m, then the number of monomials in each input polynomial should be reasonably close to $(1 - 1/q) \cdot m$. Dense input systems which are applicable include various cryptographic inputs such as HFE, MQQ, and Minrank systems, but systems which do not involve the solving of simultaneous equations are also applicable. (The algorithm does work correctly on sparser input systems but is usually not as efficient as the standard F_4 algorithm for such inputs.) Some key features of the dense variant of the F_4 algorithm are as follows:

(1) This variant essentially uses the same matrices as in the standard F_4 algorithm (i.e., uses the same S-polynomials and reductors as usual) but exploits dense portions of the matrices where possible in the linear algebra phase, leading to significant speedups for larger examples.

(2) There is often also significant savings in memory usage over the standard F_4 algorithm.

(3) If an NVIDIA GPU is available, then this is also exploited (in a special CUDA-enabled Magma executable), yielding greater speedups for larger examples.

(4) Finally, there is a new experimental optional heuristic which can be selected for the algorithm when solving systems of equations over GF(2), called the **Reduction Heuristic** which can give an even greater reduction in time and memory usage for some large examples. When computing a Gröbner basis, Magma will by default automatically select the dense variant of the F_4 algorithm if it considers that the input system is sufficiently dense. But one can force the dense variant to be used or not by setting the parameter `Dense` to `true` or `false` respectively. If the dense variant is turned off, Magma will use the standard F_4 algorithm. The **Reduction Heuristic** is a new experimental heuristic which can be selected in the dense variant of F_4 when attempting to solve certain kinds of systems of equations over $\mathbf{F}_2$ where it is assumed that there is a very small number of solutions, so the Groebner basis will consist of mostly linear polynomials or collapse to $\{1\}$ when there is no solution. The heuristic attempts to reduce the size of the matrices involved in the linear algebra phase of each F_4 step. When the heuristic is selected, the run may simply fail, but when it succeeds, it often saves significant time and memory usage. The kinds of systems for which the saving in time and memory usage tends to be greatest are those such that in the F_4 steps of maximal degree D, the number of S-polynomials is relatively small compared to the total number of monomials of degree D. Currently the Reduction Heuristic depends on a manual choice by the user of a numerical parameter M. Thus if B is the sequence of input polynomials, to select the

Reduction Heuristic with parameter M, one should currently invoke the algorithm with something like the following:

```
GroebnerBasis(B, D: ReductionHeuristic := M);
```

where M is the expected maximal degree reached in the computation. If M is large enough, then the computation will finish correctly and hopefully in considerably less time than the default algorithm without the heuristic. If M is too small, then the algorithm will fail: a runtime error is usually triggered at the point that the algorithm realizes that M was too small. Also, conceivably the algorithm could fail to discover that M is too small and so a wrong Groebner basis could be returned (thus making the algorithm Monte-Carlo). However: (1) a value of M which is too small is usually detected by Magma (and a runtime error is triggered); (2) the algorithm can actually be considered Las Vegas when solving a typical system known to have at least one solution since potential solutions can be trivially checked at the end (so coupled with checking, the algorithm either finds correct solutions or simply fails). At the moment, it is impossible to suggest what is a suitable choice for M for any given system. But it is currently recommended in practice that one start with $M = 500$ or 1000, and if there is failure, one should increase M successively by 500 or 1000 and restart from scratch and repeat this until the algorithm succeeds. Note that making M much larger than necessary reduces the effectiveness of the heuristic, so one should not just set M to a very large value. It may seem inefficient if one has to try several runs with potential failure, but this is often alleviated as follows:

(1) Since the algorithm often runs much faster with a small value for M (whether it succeeds or not), the total time taken for all runs including the ones which fail may still be significantly less than running without the heuristic selected.

(2) Each run with the heuristic may use much less memory than the default run without the heuristic would use, so no matter how much time is taken, one will eventually be able to solve systems with the heuristic which are not solvable in the available memory without the heuristic.

(3) When solving a whole class of systems of the same type, one can determine a suitable M for one system and then use that for all the others. This includes multiple systems arising from evaluating a number of variables in some original system.

For some timing examples of the dense variant of the F_4 algorithm (including uses of the Reduction Heuristic) and further information, please see the webpage:

`tinyurl.com/DenseF4`

115.4.5 Related Functions

The following functions and procedures perform operations related to Gröbner bases.

`HasGroebnerBasis(I): RngMPol -> BoolElt`

Given an ideal I, return whether the Gröbner basis of I can be computed. This depends on the type of base ring of I: the base ring must currently be a field or a Euclidean ring.

`EasyIdeal(I): RngMPol -> RngMPol, [ RngIntElt ]`

Given an ideal I, return the ideal E which is mathematically equal to I but whose basis is the Gröbner basis of I with respect to an "easy" monomial order. This order is automatically chosen by Magma and is usually the `grevlex` order or `grevlexw` order with suitable weights, and the easy basis (the Gröbner basis of the easy ideal) of I is used extensively by Magma in many of its internal algorithms. So this function allows one to access this "easy" Gröbner basis directly. The second return value is an integer sequence D which gives the sequence of step degrees encountered in each step of the F_4 algorithm, if that is used to compute the 'easy' GB, while the third return value is an isomorphism f from I onto E.

`EasyBasis(I): RngMPol -> [ RngMPolElt ]`

Given an ideal I, return the Gröbner basis of the easy ideal of I.

`SmallBasis(I): RngMPol -> [ RngMPolElt ]`

Given an ideal I, return the basis of I with shortest length which is currently known. This may be the original basis with which I was constructed, or a Gröbner basis, but the result always has the same monomial order as the main monomial order of I.

`MarkGroebner(I): RngMPol`

(Procedure.) Given an ideal I, mark the current basis of I to be *the* Gröbner basis of the ideal w.r.t. the monomial order of the ideal. Note that the current basis must exactly equal the unique (reverse) sorted minimal reduced Gröbner basis for the ideal, as returned by the function *GroebnerBasis*. This procedure is useful when one creates an ideal with a basis known to be the Gröbner basis of the ideal from a previous computation or for other reasons. If the basis is not the unique Gröbner basis, the results are unpredictable.

`IsGroebner(S): { RngMPolElt } -> BoolElt`

`IsGroebner(S): [ RngMPolElt ] -> BoolElt`

Given a set or sequence S of polynomials describing a basis of an ideal, return whether the basis is itself a (not necessarily minimal or reduced) Gröbner basis of the ideal.

`Coordinates(I, f): RngMPol, RngMPolElt -> [ RngMPolElt ]`

Given an ideal I of a polynomial ring P, together with a polynomial f in I, and supposing that I has basis $b_1, \ldots, b_k$, return a sequence $[g_1, \ldots, g_k]$ of elements of P so that $f = g_1 * b_1 + \ldots + g_k * b_k$. If I was created by *IdealWithFixedBasis*(B), then the fixed basis B is used as the basis $b_1, \ldots, b_k$; otherwise the (unique) Gröbner basis of I is used as the basis $b_1, \ldots, b_k$. The resulting sequence is not necessarily unique.

`CoordinateMatrix(I): RngMPol -> Matrix`

Given an ideal I such that I has a fixed basis (i.e., such that I was created via the function *IdealWithFixedBasis*), return the coordinate matrix C of I. The i-th row of C gives the coordinates of the i-th element of the Gröbner basis of I w.r.t. the fixed basis of I. The Gröbner basis of I is first computed if it has not been already.

`NormalForm(f, I): RngMPolElt, RngMPol -> RngMPolElt`

Given a polynomial f from a polynomial ring P, together with an ideal I of P, return the unique normal form of f with respect to (the Gröbner basis of) I. The normal form of f is zero if and only if f is in I.

`NormalForm(f, S): RngMPolElt, [ RngMPolElt ] -> RngMPolElt, [ RngMPolElt ]`

Given a polynomial f from a polynomial ring P, together with a set or sequence S of polynomials from P, return a normal form g of f with respect to S. (This is not unique in general. If the normal form of f is zero then f is in the ideal generated by S, but the converse is false in general. In fact, the normal form is unique if and only if S forms a Gröbner basis.) If S is a sequence, one may also assign a second return value C which gives the coordinates of the reduction, so that $C[i] \cdot S[i]$ is subtracted from f for each i to yield g.

`SPolynomial(f, g): RngMPolElt, RngMPolElt -> RngMPolElt`

Given elements f and g from a polynomial ring P, return the S-polynomial of f and g.

`Reduce(S): [ RngMPolElt ] -> [ RngMPolElt ]`

`Reduce(S): { RngMPolElt } -> [ RngMPolElt ]`

Given a set or sequence S of polynomials, return the sequence consisting of the reduction of S. The reduction is obtained by reducing to normal form each element of S with respect to the other elements and sorting the resulting non-zero elements left. Note that all Gröbner bases returned by Magma are automatically reduced so that this function would usually only be used just to simplify a set or sequence of polynomials which is not a Gröbner basis.

`ReduceGroebnerBasis(S): [ RngMPolElt ] -> [ RngMPolElt ]`

`ReduceGroebnerBasis(S): { RngMPolElt } -> [ RngMPolElt ]`

Given a set or sequence S of polynomials which is assumed to be a (not necessarily minimal or reduced) Gröbner basis for an ideal, return the sequence consisting of the reduction of S. The reduction is obtained by first removing each redundant polynomial whose leading term is a multiple of another leading term and then reducing the remaining polynomials as in the function *Reduce*. This function would usually only be used to reduce a set or sequence of polynomials which is known to be a non-reduced Gröbner basis (created in some way other than by one of Magma's internal Gröbner basis construction algorithms).

115.4.6 Gröbner Bases of Boolean Polynomial Rings

Since V2.15, a special type of polynomial ring is available: the **boolean polynomial ring** in n variables. Such a ring is a multivariate polynomial ring defined over $\mathbf{F}_2$ but such that all monomials are reduced modulo the *field relations* $x_i^2 = x_i$ for each i (so a bit vector representation can be used for monomials). Technically, the ring is thus the quotient algebra

$$\mathbf{F}_2[x_1, \ldots, x_n]/\langle x_1^2 + x_1, \ldots, x_n^2 + x_n \rangle.$$

Besides the basic creation and access functions for elements and ideals of such a ring, the main interest is to compute and examine a Gröbner basis of an ideal. Since the field relations are always present, an ideal represents a zero-dimensional system of multivariate polynomial equations over $\mathbf{F}_2$ with the solution components always lying in $\mathbf{F}_2$; these are particularly of interest for algebraic attacks on cryptosystems. Otherwise, there are not many other operations applicable to such rings and their elements. Note that if one creates an ideal I of $\mathbf{F}_2[x_1, \ldots, x_n]$ such that the basis of I includes the *field polynomials* ($x_i^2 + x_i$ for each i), then Magma automatically uses the boolean polynomial ring representation internally, so this is basically equivalent to using the boolean polynomial ring type, except that Magma will have to move back to the original ring $\mathbf{F}_2[x_1, \ldots, x_n]$ at the end, and this may take much more time and memory. So it is preferable to use the boolean polynomial ring from the outset if one wishes to create the Gröbner basis of such an ideal and examine it (particularly if it does not collapse down to a sequence of linear polynomials). See example *Solving a Boolean Polynomial System* below for simple uses of boolean polynomial rings.

`BooleanPolynomialRing(n): RngIntElt -> RngMPolBool`

Create the boolean polynomial ring with n variables (whose coefficients lie in $\mathbf{F}_2$). The default monomial order chosen is the lexicographical (`lex`) order.

`BooleanPolynomialRing(n, order): RngIntElt, MonStgElt -> RngMPolBool`

Create the boolean polynomial ring with n variables (whose coefficients lie in $\mathbf{F}_2$) and with the given order order on the monomials. Currently, order must be one of the following strings: `"lex"`, `"grevlex"`, `"glex"`.

`BooleanPolynomialRing(B, Q): RngMPolBool, [RngIntElt] -> RngMPolBoolElt`

Given a boolean polynomial ring B of rank n and a sequence Q of integers, create the boolean polynomial in B whose monomials are given by the entries of Q: each integer must be in the range $[0 \ldots 2^n - 1]$ and its binary expansion gives the exponents of the monomial in order (the resulting monomials are sorted w.r.t. the monomial order of B, so may be given in any order and duplicate monomials are added). This function is simply provided so that boolean polynomials may be stored and read back in a compact form; otherwise, one can create a boolean polynomial in the usual way from the generators of B after B is created. Note also that if one prints B, an ideal of B, or an element of B with the `Magma` print level, then this function will be used to print the elements in a compact form.

115.4.7 Construction of Input Systems

`MinRankSystem(K, n, k, r): FldFin, RngIntElt, RngIntElt, RngIntElt -> [ RngMPolBool ]`

For a finite field K and positive integer parameters n, k, r, the square (n, k, r)-*MinRank* problem is as follows: given a square linear matrix of size n with k variables (i.e., a matrix whose entries are k-variate polynomials of degree 1 over K), the goal is to find the locus of the points (assignment to the k variables) such that the evaluated matrix has rank less than or equal to r. This function returns the sequence of minors of the relevant symbolic matrix as multivariate polynomials with k variables, so solving the corresponding multivariate system solves the MinRank problem instance. See Example `Solving MinRank Problems (ex-cdcd67)` in Chapter Parallelism on parallelism for example systems constructed by this function.

115.4.8 Verbosity

This subsection describes the verbose flags available for the Gröbner basis algorithms. There are separate verbose flags for each algorithm (`Buchberger`, etc.), but the all-encompassing verbose flag `Groebner` includes all these flags implicitly.

For each procedure provided for setting one of these flags, the value `false` is equivalent to level 0 (nothing), and `true` is equivalent to level 1 (minimal verbosity). For each `Set-` procedure, there is also a corresponding `Get-` function to return the value of the corresponding flag.

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

(Procedure.) Change the verbose printing level for all Gröbner basis algorithms to be v. This includes all of the algorithms whose verbosity is controlled by flags subsequently listed, as well as some other minor related algorithms. Currently the legal levels are 0, 1, 2, 3, or 4. One would normally set this flag to 1 for minimal verbosity for Gröbner basis-type computations, and possibly also set one or more of the following flags to levels higher than 1 for more verbosity.

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

(Procedure.) Change the verbose printing level for the Buchberger algorithm to be v. Currently the legal levels are 0, 1, 2, 3, or 4. If the value w of the `Groebner` verbose flag is greater than v, then w is taken to be the current value of this flag.

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

(Procedure.) Change the verbose printing level for the Faugère algorithm to be v. Currently the legal levels are 0, 1, 2, or 3. If the value w of the `Groebner` verbose flag is greater than v, then w is taken to be the current value of this flag.

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

(Procedure.) Change the verbose printing level for the FGLM order change algorithm to be v. Currently the legal levels are 0, 1, 2, or 3. If the value w of the `Groebner` verbose flag is greater than v, then w is taken to be the current value of this flag.

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

(Procedure.) Change verbose printing for the Gröbner Walk order change algorithm to be v. Currently the legal levels are 0, 1, 2, or 3. If the value w of the `Groebner` verbose flag is greater than v, then w is taken to be the current value of this flag.

Example: Groebner Basis of the Cyclic-6 Ideal (ex-a7ca30)

We compute the Gröbner basis of the "Cyclic-6" ideal with respect to the lexicographical order. The ideal is an ideal of the polynomial ring $\mathbb{Q}(x, y, z, t, u, v)$. We also note that the last polynomial in the Gröbner basis is univariate (since, in fact, the ideal is zero-dimensional and the monomial order is lexicographical) and observe that it has a nice factorization. Note especially that in this example, homogenizing at first and keeping the Gröbner basis reduced makes this computation very fast; without using these features (i.e., if the parameters `Homogenize := false` or `ReduceByNew := false` are given), the computation is much more expensive (takes hundreds of seconds on the same computer).

```
> Q := RationalField();
> P<x, y, z, t, u, v> := PolynomialRing(Q, 6);
> I := ideal<P |
>     x + y + z + t + u + v,
>     x*y + y*z + z*t + t*u + u*v + v*x,
>     x*y*z + y*z*t + z*t*u + t*u*v + u*v*x + v*x*y,
>     x*y*z*t + y*z*t*u + z*t*u*v + t*u*v*x + u*v*x*y + v*x*y*z,
>     x*y*z*t*u + y*z*t*u*v + z*t*u*v*x + t*u*v*x*y + u*v*x*y*z + v*x*y*z*t,
>     x*y*z*t*u*v - 1>;
> time B := GroebnerBasis(I);
Time: 1.140
> #B;
17
> B[17];
v^48 - 2554*v^42 - 399710*v^36 - 499722*v^30 + 499722*v^18 + 399710*v^12 +
    2554*v^6 - 1
> time Factorization(B[17]);
[
```

(continues on next page)

(continued from previous page)

```
    <v - 1, 1>,
    <v + 1, 1>,
    <v^2 + 1, 1>,
    <v^2 - 4*v + 1, 1>,
    <v^2 - v + 1, 1>,
    <v^2 + v + 1, 1>,
    <v^2 + 4*v + 1, 1>,
    <v^4 - v^2 + 1, 1>,
    <v^4 - 4*v^3 + 15*v^2 - 4*v + 1, 1>,
    <v^4 + 4*v^3 + 15*v^2 + 4*v + 1, 1>,
    <v^8 + 4*v^6 - 6*v^4 + 4*v^2 + 1, 1>,
    <v^8 - 6*v^7 + 16*v^6 - 24*v^5 + 27*v^4 - 24*v^3 +
        16*v^2 - 6*v + 1, 1>,
    <v^8 + 6*v^7 + 16*v^6 + 24*v^5 + 27*v^4 + 24*v^3 +
        16*v^2 + 6*v + 1, 1>
]
Time: 0.060
```

Example: Runge-Kutta 2 System (ex-a39450)

We solve the system of equations Runge-Kutta 2 from the paper "Some Examples for Solving Systems of Algebraic Equations by Calculating Groebner Bases" by Boege, Gebauer, and Kredel (J. Symbolic Computation (1986) **1**, 83–98). The coefficient field K is the rational function field $\mathbb{Q}(c2, c3)$, and the polynomial ring $K[c4, b4, b3, b2, b1, a21, a31, a32, a41, a42, a43]$ has 11 variables with the lexicographical ordering on monomials. The resulting Gröbner basis contains a linear polynomial for each variable so there is exactly one solution to the system.

```
> K<c2, c3> := FunctionField(IntegerRing(), 2);
> P<c4, b4, b3, b2, b1, a21, a31, a32, a41, a42, a43> := PolynomialRing(K, 11);
> I := ideal<P |
>    b1 + b2 + b3 + b4 - 1,
>    b2*c2 + b3*c3 + b4*c4 - 1/2,
>    b2*c2^2 + b3*c3^2 + b4*c4^2 - 1/3,
>    b3*a32*c2 + b4*a42*c2 + b4*a43*c3 - 1/6,
>    b2*c2^3 + b3*c3^3 + b4*c4^3 - 1/4,
>    b3*c3*a32*c2 + b4*c4*a42*c2 + b4*c4*a43*c3 - 1/8,
>    b3*a32*c2^2 + b4*a42*c2^2 + b4*a43*c3^2 - 1/12,
>    b4*a43*a32*c2 - 1/24,
>    c2 - a21,
>    c3 - a31 - a32,
>    c4 - a41 - a42 - a43>;
> time Groebner(I);
Time: 0.110
> I;
Ideal of Polynomial ring of rank 11 over Multivariate rational function field
    of rank 2 over Integer Ring
Order: Lexicographical
Variables: c4, b4, b3, b2, b1, a21, a31, a32, a41, a42, a43
Inhomogeneous, Dimension 0
Groebner basis:
[
    c4 - 1,
    b4 + (-6*c2*c3 + 4*c2 + 4*c3 - 3)/(12*c2*c3 - 12*c2 - 12*c3 + 12),
    b3 + (2*c2 - 1)/(12*c2*c3^2 - 12*c2*c3 - 12*c3^3 + 12*c3^2),
    b2 + (-2*c3 + 1)/(12*c2^3 - 12*c2^2*c3 - 12*c2^2 + 12*c2*c3),
    b1 + (-6*c2*c3 + 2*c2 + 2*c3 - 1)/(12*c2*c3),
```

(continues on next page)

(continued from previous page)

```
    a21 - c2,
    a31 + (-4*c2^2*c3 + 3*c2*c3 - c3^2)/(4*c2^2 - 2*c2),
    a32 + (-c2*c3 + c3^2)/(4*c2^2 - 2*c2),
    a41 + (-12*c2^2*c3^2 + 12*c2^2*c3 - 4*c2^2 + 12*c2*c3^2 - 15*c2*c3 + 6*c2 -
        4*c3^2 + 5*c3 - 2)/(12*c2^2*c3^2 - 8*c2^2*c3 - 8*c2*c3^2 + 6*c2*c3),
    a42 + (-c2^2 + 4*c2*c3^2 - 5*c2*c3 + 3*c2 - 4*c3^2 + 5*c3 - 2)/(12*c2^3*c3 -
        8*c2^3 - 12*c2^2*c3^2 + 6*c2^2 + 8*c2*c3^2 - 6*c2*c3),
    a43 + (-2*c2^2*c3 + 2*c2^2 + 3*c2*c3 - 3*c2 - c3 + 1)/(6*c2^2*c3^2 -
        4*c2^2*c3 - 6*c2*c3^3 + 3*c2*c3 + 4*c3^3 - 3*c3^2)
]
```

Example: Solving a Boolean Polynomial System `(ex-5d686d)`

We demonstrate how one can solve a system of multivariate equations over $\mathbf{F}_2$. We construct a sequence B of 4 polynomials in 5 variables, and note that the Gröbner basis of B contains monomials having degrees greater than 1.

```
> P<a,b,c,d,e> := PolynomialRing(GF(2), 5);
> B := [a*b + c*d + 1, a*c*e + d*e, a*b*e + c*e, b*c + c*d*e + 1];
> GroebnerBasis(B);
[
    a + c^2*d + c + d^2*e,
    b*c + d^3*e^2 + d^3*e + d^2*e^2 + d*e + e + 1,
    b*e + d*e^2 + d*e + e,
    c*e + d^3*e^2 + d^3*e + d^2*e^2 + d*e,
    d^4*e^2 + d^4*e + d^3*e + d^2*e^2 + d^2*e + d*e + e
]
```

If one wanted to consider solutions over an algebraic closure of $\mathbf{F}_2$, then one would have to work with this ideal. But to solve over $\mathbf{F}_2$ itself, one can add the *field polynomials* $a^2 + a$, $b^2 + b$, etc. Magma recognizes these extra polynomials and uses an optimized representation; this makes the computation much faster for larger examples. The resulting polynomials (besides any remaining field polynomials) will always have degree at most 1 in each variable. In this example, we see that there are 2 solutions over $\mathbf{F}_2$ for the system.

```
> L := [P.i^2 + P.i: i in [1 .. Rank(P)]];
> BB := B cat L;
> BB;
[
    a*b + c*d + 1,
    a*c*e + d*e,
    a*b*e + c*e,
    b*c + c*d*e + 1,
    a^2 + a,
    b^2 + b,
    c^2 + c,
    d^2 + d,
    e^2 + e
]
> GroebnerBasis(BB);
[
    a + d + 1,
    b + 1,
    c + 1,
    d^2 + d,
    e
]
```

(continues on next page)

(continued from previous page)

```
> I := ideal<P|BB>;
> Variety(I);
[ <0, 1, 1, 1, 0>, <1, 1, 1, 0, 0> ]
```

Since V2.15, an alternative way to solve the system over $\mathbf{F}_2$ is to use the boolean polynomial ring type as follows.

```
> P<a,b,c,d,e> := BooleanPolynomialRing(5, "grevlex");
> B := [a*b + c*d + 1, a*c*e + d*e, a*b*e + c*e, b*c + c*d*e + 1];
> I := Ideal(B);
> I;
Ideal of Boolean polynomial ring of rank 5 over GF(2)
Order: Graded Reverse Lexicographical (bit vector word)
Variables: a, b, c, d, e
Basis:
[
    a*b + c*d + 1,
    a*c*e + d*e,
    a*b*e + c*e,
    c*d*e + b*c + 1
]
> GroebnerBasis(I);
[
    a + d + 1,
    b + 1,
    c + 1,
    e
]
> Variety(I);
[ <0, 1, 1, 1, 0>, <1, 1, 1, 0, 0> ]
```

In general, if one wishes to solve a system over $\mathbf{F}_2$ from the outset, it is best to use the boolean polynomial ring type so as to save memory (and to avoid internal conversion to and from the bit vector representation for monomials). Note also that because of the implicit field relations, the Gröbner basis of an ideal generated by only one polynomial may have several polynomials. In the following example, the Gröbner basis of an ideal generated by just one polynomial has linear polynomials alone.

```
> R<[x]> := BooleanPolynomialRing(10, "grevlex");
> R;
Boolean polynomial ring of rank 10 over GF(2)
Order: Graded Reverse Lexicographical (bit vector word)
Variables: x[1], x[2], x[3], x[4], x[5], x[6], x[7], x[8], x[9], x[10]
> f := x[2]*x[3]*x[5]*x[7] + x[2]*x[4]*x[5]*x[8] + x[3]*x[4]*x[5]*x[9] +
>    x[3]*x[6]*x[7]*x[9] + x[2]*x[3]*x[5] + x[2]*x[4]*x[5] + x[2]*x[3]*x[7] +
>    x[2]*x[5]*x[7] + x[3]*x[5]*x[7] + x[3]*x[6]*x[7] + x[2]*x[4]*x[8] +
>    x[2]*x[5]*x[8] + x[4]*x[5]*x[8] + x[3]*x[6]*x[9] + x[3]*x[7]*x[9] +
>    x[6]*x[7]*x[9] + x[2]*x[3] + x[2]*x[4] + x[3]*x[5] + x[4]*x[5] +
>    x[3]*x[6] + x[2]*x[7] + x[5]*x[7] + x[6]*x[7] + x[2]*x[8] + x[4]*x[8] +
>    x[5]*x[8] + x[3]*x[9] + x[6]*x[9] + x[7]*x[9] + x[1]*x[10] + x[1] + x[4]
>    + x[6] + x[8] + x[9] + x[10];
> I := Ideal([f]);
> G := GroebnerBasis(I);
> #G;
38
> [Length(f): f in G];
```

(continues on next page)

(continued from previous page)

```
[ 188, 50, 80, 82, 26, 22, 20, 26, 20, 20, 26, 32, 8, 8, 8, 8, 32, 32, 8, 8, 8,
8, 8, 8, 8, 8, 8, 32, 8, 8, 8, 8, 40, 5, 8, 8, 8, 8 ]
> G[38];
x[1]*x[4]*x[7]*x[10] + x[1]*x[5]*x[7]*x[10] + x[1]*x[4]*x[7] + x[1]*x[5]*x[7] +
    x[4]*x[7]*x[10] + x[5]*x[7]*x[10] + x[4]*x[7] + x[5]*x[7]
```

Example: Groebner Bases over the Integers (ex-daf567)

This simple example illustrates some of the peculiarities of Gröbner bases over Euclidean rings. We first create a simple ideal I in $\mathbb{Z}[x, y, z]$ and compute its Gröbner basis.

```
> P<x, y, z> := PolynomialRing(IntegerRing(), 3);
> I := ideal<P| x^2 - 1, y^2 - 1, 2*x*y - z>;
> GroebnerBasis(I);
[
    x^2 - 1,
    x*z - 2*y,
    2*x - y*z,
    y^2 - 1,
    z^2 - 4
]
```

Notice that the Gröbner basis contains polynomials whose leading terms are x^2, xz, and $2x$, but the third cannot eliminate the first two since the leading coefficient 2 does not divide the other leading coefficients 1 and 1.

When we compute normal forms modulo I, x is clearly not reducible by any polynomial, while $2x$ can be reduced by the $2x - yz$ polynomial.

```
> NormalForm(x, I);
x
> NormalForm(2*x, I);
y*z
```

If we compute the normal form of $(-x)$ modulo I, then even though the x monomial cannot be reduced, the result is **not** the negative of the normal form of x, since one can use the $2x - yz$ polynomial and the fact that $((-1) \bmod 2)$ is 1 to reduce the polynomial to a unique normal form. This behaviour differs from that for ideals defined over fields, where the normal form of $-f$ will always be the negative of the normal form of f.

```
> NormalForm(-x, I);
x - y*z
```

If we reduce the Gröbner basis modulo various primes, we obtain familiar Gröbner bases over fields:

```
> GroebnerBasis(ChangeRing(I, GF(2)));
[
    x^2 + 1,
    y^2 + 1,
    z
]
> GroebnerBasis(ChangeRing(I, GF(3)));
[
    x + y*z,
    y^2 + 2,
    z^2 + 2
]
```

But if we reduce modulo 4, using the ring of integers modulo 4, then the Gröbner basis still has a structure not encountered when working over fields:

```
> GroebnerBasis(ChangeRing(I, IntegerRing(4)));
[
    x^2 + 3,
    x*z + 2*y,
    2*x + y*z,
    y^2 + 3,
    z^2,
    2*z
]
```

In fact, the new polynomial $2z$ has been included in this Gröbner basis.

Example: Finding Primes via Groebner Bases (ex-9e862d)

This example shows how one can use Gröbner bases over the integers to find the primes modulo which a system of equations has a solution, when the system has no solutions over the rationals.

We first form a certain ideal I in $\mathbb{Z}[x, y, z]$, and note that the Gröbner basis of I over $\mathbb{Q}$ contains 1, so there are no solutions over $\mathbb{Q}$ or an algebraic closure of it (this is not surprising as there are 4 equations in 3 unknowns).

```
> P<x, y, z> := PolynomialRing(IntegerRing(), 3);
> I := ideal<P | x^2 - 3*y, y^3 - x*y, z^3 - x, x^4 - y*z + 1>;
> GroebnerBasis(ChangeRing(I, RationalField()));
[
    1
]
```

However, when we compute the Gröbner basis of I (defined over $\mathbb{Z}$), we note that there is a certain integer in the ideal which is not 1.

```
> GroebnerBasis(I);
[
    x + 170269749119,
    y + 2149906854,
    z + 170335012540,
    282687803443
]
```

Now for each prime p dividing this integer 282687803443, the Gröbner basis of I modulo p will be non-trivial and will thus give a solution of the original system modulo p.

```
> Factorization(282687803443);
[ <101, 1>, <103, 1>, <27173681, 1> ]
> GroebnerBasis(ChangeRing(I, GF(101)));
[
    x + 19,
    y + 48,
    z + 68
]
> GroebnerBasis(ChangeRing(I, GF(103)));
[
    x + 39,
    y + 8,
    z + 85
]
```

(continues on next page)

(continued from previous page)

```
> GroebnerBasis(ChangeRing(I, GF(27173681)));
[
    x + 26637654,
    y + 3186055,
    z + 10380032
]
```

Of course, modulo any other prime the Gröbner basis is trivial so there are no other solutions. For example:

```
> GroebnerBasis(ChangeRing(I, GF(3)));
[
    1
]
```

Note that the problem can also be solved by using resultants, but this may yield many extraneous potential primes, while the Gröbner basis technique yields the exact list of primes for which there are modular solutions.

Example: Groebner Bases over a Quadratic Order (ex-6747d7)

This example shows how one can effectively compute in Magma with Gröbner bases over a ring which is not Euclidean (and may not even be a principal ideal ring), by starting with $\mathbb{Z}$ and adding appropriate defining relations. The input for this example is based on [AL94, Ex. 4.2.13].

Let $R = \mathbb{Z}[\sqrt{-5}]$. R is the maximal order of $\mathbb{Q}(\sqrt{-5})$ and is **NOT** a PIR. We consider the ideal I of $R[x, y]$ generated by $f_1 = 2xy + \sqrt{-5}y$ and $f_2 = (1 + \sqrt{-5})x^2 - xy$. To work over R, we simply compute over $\mathbb{Z}$, introduce a new variable S to represent $\sqrt{-5}$, make sure that S is less than both x and y in the monomial order, and include the polynomial $(S^2 + 5)$ in the ideal I. We then print out the Gröbner basis of I.

```
> P<x, y, S> := PolynomialRing(IntegerRing(), 3);
> f1 := 2*x*y + S*y;
> f2 := (1 + S)*x^2 - x*y;
> I := ideal<P | f1, f2, S^2 + 5>;
> GroebnerBasis(I);
[
    x^2*S + x^2 + 5*y^3 + 13*y*S - 25*y,
    6*x^2 + 5*y^2 + 3*y*S - 10*y,
    x*y + 5*y^3 + 13*y*S - 25*y,
    y^2*S + 5*y^2 - 15*y,
    10*y^2 + 5*y*S - 25*y,
    S^2 + 5
]
```

In [AL94, p. 224], a (weak) Gröbner basis for the ideal is given as $\{f_2, f_5, f_7, f_9\}$, where $f_5 = (5 + \sqrt{-5})y^2 - 15y$, $f_7 = -2\sqrt{-5}y^2 + 5(1 + \sqrt{-5})y$, and $f_9 = xy + \sqrt{-5}y^3 - 5\sqrt{-5}y^2 + 8\sqrt{-5}y$. We can easily verify that the ideal J generated by these 4 polynomials describes the same ideal as I (and so has the same Gröbner basis in Magma).

```
> f5 := (5 + S)*y^2 - 15*y;
> f7 := -2*S*y^2 + (5 + 5*S)*y;
> f9 := x*y + S*y^3 - 5*S*y^2 + 8*S*y;
> J := ideal<P |  f2, f5, f7, f9, S^2 + 5>;
> I eq J;
true
> GroebnerBasis(I) eq GroebnerBasis(J);
true
```

We can even write f_5, f_7, and f_9 as combinations of the Gröbner basis elements of I, as follows.

```
> Coordinates(I, f5);
[
    0, 0, 0, 1, 0, 0
]
> Coordinates(I, f7);
[
    0, 0, 0, -2, 1, 0
]
> Coordinates(I, f9);
[
    0, 0, 1, y, -y - 1, 0
]
```

We can see that these elements are fairly trivially derived from the Gröbner basis which Magma computes for I. But if we now create J again using the *IdealWithFixedBasis* function and the sequence $Q = [f_2, f_5, f_7, f_9, S^2 + 5]$, then we can see the coordinates of any element of $I = J$ as a linear combination of the elements of Q. We find the coordinates of the second element of Magma's original Gröbner basis of I with respect to Q. The resulting coordinates are rather non-trivial.

```
> Q := [f2, f5, f7, f9, S^2 + 5];
> J := IdealWithFixedBasis(Q);
> J eq I;
true
> g := GroebnerBasis(I)[2];
> g;
6*x^2 + 5*y^2 + 3*y*S - 10*y
> C := Coordinates(J, g);
> C;
[
    -S + 1,
    -5*y + 1,
    -x - y^2*S + 7*y*S - 2*y - 7*S - 2,
    -2*y*S + 4*S + 6,
    x^2 + 5*y^3 - 13*y^2 + 3*y
]
```

We check that multiplying out the expression recovers g.

```
> &+[C[i]*Q[i]: i in [1 .. #C]] eq g;
true
```

Note that in the terminology of Adams and Loustaunau, Magma is here computing a "strong" Gröbner basis (for this representation which uses an extra variable for $\sqrt{-5}$), while these authors show that $\{f_2, f_5, f_7, f_9\}$ constitutes a "weak" Gröbner basis for I over the ring $\mathbb{Z}[\sqrt{-5}]$. The fact that the coordinates of g with respect to Q are rather non-trivial shows that Magma's strong Gröbner basis computation has computed a lot more information than the weak Gröbner basis (i.e., g, which must be included in the strong Gröbner basis, is not trivially derived from Q).

Most importantly of all, the fact that we have done all this by defining things over $\mathbb{Z}$ with the extra variable S has been no less powerful: we can still do full membership testing, normal forms, coordinate computations, etc. with this representation. Also, see below for an elimination computation which continues this example.

Gröbner bases over very many other general rings can be effectively handled in just the same way as that presented in this example. For example, if we need $\alpha = (1 + \sqrt{5})/2$, we can introduce a variable new A and the polynomial $(2A - 1)^2 - 5$.

Example: Coordinates with Respect to an Ideal Basis (ex-51a96f)

We construct an ideal I of the polynomial ring $P = \mathbb{Q}[x, y]$ with a specific fixed basis S, determine that I is the full polynomial ring P, and then find coordinates of the polynomial 1 of P with respect to S. Note that we use the function *IdealWithFixedBasis* to construct the ideal so that the fixed basis will be remembered.

```
> P<x, y> := PolynomialRing(RationalField(), 2);
> S := [x^2 - y, x^3 + y^2, x*y^3 - 1];
> I := IdealWithFixedBasis(S);
> 1 in I;
true
> C := Coordinates(I, P!1);
> C;
[
    -1/2*x^2*y^3 - 1/2*x^2*y^2 + 1/2*x^2*y + 1/2*x^2 + 1/2*x*y^3 +
        1/2*x*y^2 - 1/2*x*y - 1/2*y^4 - 1/2*y^3 + 1/2*y^2 + 1/2*y,
    1/2*x*y^3 + 1/2*x*y^2 - 1/2*x*y - 1/2*x - 1/2*y^3 - 1/2*y^2 + 1/2*y,
    -1/2*y^2 + 1
]
```

Now we check that multiplying out by the coordinates gives 1.

```
> C[1]*S[1] + C[2]*S[2] + C[3]*S[3];
1
```

Now we move the problem to being over the integer ring $\mathbb{Z}$.

```
> P<x, y> := PolynomialRing(IntegerRing(), 2);
> S := [x^2 - y, x^3 + y^2, x*y^3 - 1];
> I := IdealWithFixedBasis(S);
> 1 in I;
false
> GroebnerBasis(I);
[
    x + 1,
    y + 1,
    2
]
```

We note that 1 is not in the ideal this time, but 2 is! So we compute the coordinates of 2 with respect to I this time.

```
> C := Coordinates(I, P!2);
> C;
[
    x^2*y^2 - x^2*y - x^2 - x*y^2 + x*y + x + y^4 + y^3 - y^2 - y - 1,
    -x*y^2 + x*y + x + y^3 + y^2 - y - 1,
    -x^2 - x*y + y - 2
]
```

Note that C is the same as above, except that each polynomial has been scaled by 2 to make it integral. Finally we check again that multiplying out by the coordinates gives 2.

```
> C[1]*S[1] + C[2]*S[2] + C[3]*S[3];
2
```

Incidentally, we can see from the Gröbner basis of I over $\mathbb{Z}$ that the only solution to the system of equations described by S is the local solution $x = y = 1$ over $\mathbf{F}_2$.

Example: Groebner Bases over a Valuation Ring (ex-eaa5db)

Gröbner bases can be constructed over any exact Euclidean ring in Magma, not just the ring of integers and its residue class rings.

We construct an ideal I of the polynomial ring $P = \mathbb{Q}[x, y]$ with a specific fixed basis S, determine that I is the full polynomial ring P, and then find coordinates of the polynomial 1 of P with respect to S. Note that we use the function *IdealWithFixedBasis* to construct the ideal so that the fixed basis will be remembered.

```
> P<x, y> := PolynomialRing(RationalField(), 2);
> S := [x^2 - y, x^3 + y^2, x*y^3 - 1];
> I := IdealWithFixedBasis(S);
> 1 in I;
true
> C := Coordinates(I, P!1);
> C;
[
    -1/2*x^2*y^3 - 1/2*x^2*y^2 + 1/2*x^2*y + 1/2*x^2 + 1/2*x*y^3 +
        1/2*x*y^2 - 1/2*x*y - 1/2*y^4 - 1/2*y^3 + 1/2*y^2 + 1/2*y,
    1/2*x*y^3 + 1/2*x*y^2 - 1/2*x*y - 1/2*x - 1/2*y^3 - 1/2*y^2 + 1/2*y,
    -1/2*y^2 + 1
]
```

Now we check that multiplying out by the coordinates gives 1.

```
> C[1]*S[1] + C[2]*S[2] + C[3]*S[3];
1
```

Now we move the problem to being over the integer ring $\mathbb{Z}$.

```
> P<x, y> := PolynomialRing(IntegerRing(), 2);
> S := [x^2 - y, x^3 + y^2, x*y^3 - 1];
> I := IdealWithFixedBasis(S);
> 1 in I;
false
> GroebnerBasis(I);
[
    x + 1,
    y + 1,
    2
]
```

We note that 1 is not in the ideal this time, but 2 is! So we compute the coordinates of 2 with respect to I this time.

```
> C := Coordinates(I, P!2);
> C;
[
    x^2*y^2 - x^2*y - x^2 - x*y^2 + x*y + x + y^4 + y^3 - y^2 - y - 1,
    -x*y^2 + x*y + x + y^3 + y^2 - y - 1,
    -x^2 - x*y + y - 2
]
```

Note that C is the same as above, except that each polynomial has been scaled by 2 to make it integral. Finally we check again that multiplying out by the coordinates gives 2.

```
> C[1]*S[1] + C[2]*S[2] + C[3]*S[3];
2
```

Incidentally, we can see from the Gröbner basis of I over $\mathbb{Z}$ that the only solution to the system of equations described by S is the local solution $x = y = 1$ over $\mathbf{F}_2$.

115.4.9 Degree-d Gröbner Bases

`GroebnerBasis(S, d : parameters): [ RngMPolElt ], RngInt -> RngMPolElt`

Given a set or sequence S of polynomials from a graded polynomial ring P, return the weighted degree-d Gröbner basis of the ideal generated by S, which is the truncated Gröbner basis obtained by ignoring S-polynomial pairs whose weighted degree (with respect to the grading on P) is greater than d.

If the ideal is homogeneous, then it is guaranteed that the result is equal to the set of all polynomials in the full Gröbner basis of the ideal whose weighted degree is less than or equal to d, and a polynomial whose weighted degree is less than or equal to d is in the ideal iff its normal form with respect to this truncated basis is zero. But if the ideal is not homogeneous, these last properties may not hold, but it may be still useful to construct the truncated basis.

The parameters are the same as those for the procedure *Groebner*. See also [BW93, section 10.2] for further discussion. Note that the base ring may be a field or Euclidean ring.

Example: Degree-d Groebner Bases (ex-054264)

We create a graded polynomial ring and compute the degree-d Gröbner basis of a sequence L of homogeneous polynomials for various d. Since the polynomials are homogeneous (with respect to the grading), we check that the result for each d contains the set of all polynomials in the full Gröbner basis of L having weighted degree less than or equal to d.

```
> P<a,b,c,d> := PolynomialRing(RationalField(), [4,3,2,1]);
> L := [a*b - c^2*d^3, b*c*d + c^3, c^2*d - d^5, a*d - b*c];
> [IsHomogeneous(f): f in L];
[ true, true, true, true ]
> [Degree(f): f in L];
[ 7, 6, 5, 5 ]
> G:=GroebnerBasis(L);
> G;
[
    a*b - d^7,
    a*c^3 + d^10,
    a*d - b*c,
    b^2*c - d^8,
    b*c^3 + d^9,
    b*c*d + c^3,
    b*d^5 + c^4,
    c^5 - d^10,
    c^2*d - d^5,
    c*d^7 - d^9
]
> #G;
10
> [Degree(f): f in G];
[ 7, 10, 5, 8, 9, 6, 8, 10, 5, 9 ]
> for D := 1 to 10 do
>     T := GroebnerBasis(L, D);
>     printf "D = %o, #GB = %o, contains all degree-D polynomials: %o\\n",
>         D, #T, {f: f in G | Degree(f) le D} subset T;
> end for;
D = 1, #GB = 4, contains all degree-D polynomials: true
D = 2, #GB = 4, contains all degree-D polynomials: true
D = 3, #GB = 4, contains all degree-D polynomials: true
D = 4, #GB = 4, contains all degree-D polynomials: true
D = 5, #GB = 4, contains all degree-D polynomials: true
D = 6, #GB = 4, contains all degree-D polynomials: true
```

(continues on next page)

(continued from previous page)

```
D = 7, #GB = 4, contains all degree-D polynomials: true
D = 8, #GB = 6, contains all degree-D polynomials: true
D = 9, #GB = 8, contains all degree-D polynomials: true
D = 10, #GB = 10, contains all degree-D polynomials: true
> GroebnerBasis(L, 5);
[
    a*b - d^7,
    a*d - b*c,
    b*c*d + c^3,
    c^2*d - d^5
]
> GroebnerBasis(L, 8);
[
    a*b - d^7,
    a*d - b*c,
    b^2*c - d^8,
    b*c*d + c^3,
    b*d^5 + c^4,
    c^2*d - d^5
]
```

115.5 Changing Coefficient Ring

The `ChangeRing` function enables the changing of the coefficient ring of a polynomial ring or ideal.

`ChangeRing(I, S): RngMPol, Rng -> RngMPol`

Given an ideal I of a polynomial ring $P = R[x_1, \ldots, x_n]$ of rank n with coefficient ring R, together with a ring S, construct the ideal J of the polynomial ring $Q = S[x_1, \ldots, x_n]$ obtained by coercing the coefficients of the elements of the basis of I into S. It is necessary that all elements of the old coefficient ring R can be automatically coerced into the new coefficient ring S. If R and S are fields and R is known to be a subfield of S and the current basis of I is a Gröbner basis, then the basis of J is marked automatically to be a Gröbner basis of J.

Example: Groebner Basis over a Smaller Field `(ex-d2d66e)`

It is better to find the Gröbner basis of an ideal over the smallest subfield possible (e.g. $\mathbb{Q}$), then use *ChangeRing* to create the equivalent ideal over a splitting field to find the variety.

```
> P<x, y, z, t, u> := PolynomialRing(RationalField(), 5);
> I := ideal<P |
>     x + y + z + t + u,
>     x*y + y*z + z*t + t*u + u*x,
>     x*y*z + y*z*t + z*t*u + t*u*x + u*x*y,
>     x*y*z*t + y*z*t*u + z*t*u*x + t*u*x*y + u*x*y*z,
>     x*y*z*t*u - 1>;
> Groebner(I);
> K<W> := CyclotomicField(5);
> J := ChangeRing(I, K);
> V := Variety(J);
> #V;
70
```

115.6 Changing Monomial Order

Often one wishes to change the monomial order of an ideal. Magma allows one to do this by use of the `ChangeOrder` function.

`ChangeOrder(I, Q): RngMPol, RngMPol -> RngMPol, Map`

Given an ideal I of the polynomial ring $P = R[x_1, \ldots, x_n]$, together with a polynomial ring Q of rank n (with possibly a different order to that of P), return the ideal J of Q corresponding to I and the isomorphism f from P to Q. The map f simply maps $P.i$ to $Q.i$ for each i.

The point of the function is that one can change the order on monomials of I to be that of Q. When a Gröbner basis of J is needed to be calculated, Magma uses a conversion algorithm starting from a Gröbner basis of I if possible—this usually makes order conversion much more efficient than by computing a Gröbner basis of J from scratch.

`ChangeOrder(I, order): RngMPol, ... -> RngMPol, Map`

Given an ideal I of the polynomial ring $P = R[x_1, \ldots, x_n]$, together with a monomial order order (see Section *Representation and Monomial Orders*), construct the polynomial ring $Q = R[x_1, \ldots, x_n]$ with order order, and then return the ideal J of Q corresponding to I and the isomorphism f from P to Q. See the section on monomial orders for the valid values for the argument order. The map f simply maps $P.i$ to $Q.i$ for each i.

`ChangeOrder(I, T): RngMPol, Tup -> RngMPol`

Given an ideal I of the polynomial ring $P = R[x_1, \ldots, x_n]$, together with a tuple T, construct the polynomial ring $Q = R[x_1, \ldots, x_n]$ with the monomial order given by the tuple T on the monomials, and then return the ideal J of Q corresponding to I and the isomorphism f from P to Q. T must be a tuple whose components match the valid arguments for the monomial orders in Section *Representation and Monomial Orders* (or a tuple returned by the function *`MonomialOrder`*).

Example: Univariate Generators by Change of Order `(ex-6f12cc)`

We write a function `univgen` which, given a zero-dimensional ideal defined over a field, computes the univariate elimination ideal generator for a particular variable by changing order to the appropriate univariate order. Note that this function is the same as (and is in fact implemented in exactly the same way as) the intrinsic function `UnivariateEliminationIdealGenerator`. We then find the appropriate univariate polynomials for a particular ideal.

```
> function univgen(I, i)
>    // Make sure I has a Groebner basis so that
>    // the conversion algorithm will be used when
>    // constructing a Groebner basis of J
>    Groebner(I);
>    J := ChangeOrder(I, "univ", i);
>    Groebner(J);
>    return rep{f: f in Basis(J) | IsUnivariate(f, i)};
> end function;
>
> P<x, y, z> := PolynomialRing(RationalField(), 3, "grevlex");
> I := ideal<P |
>     1 - x + x*y^2 - x*z^2,
>     1 - y + y*x^2 + y*z^2,
>     1 - z - z*x^2 + z*y^2 >;
>
> univgen(I, 1);
x^21 - x^20 - 2*x^19 + 4*x^18 - 5/2*x^17 - 5/2*x^16 + 4*x^15 -
    15/2*x^14 + 129/16*x^13 + 11/16*x^12 - 103/8*x^11 +
    131/8*x^10 - 49/16*x^9 - 171/16*x^8 + 12*x^7 - 3*x^6 -
    29/8*x^5 + 15/4*x^4 - 17/16*x^3 - 5/16*x^2 + 5/16*x - 1/16
> univgen(I, 2);
```

(continues on next page)

(continued from previous page)

```
y^14 - y^13 - 13/2*y^12 + 8*y^11 + 53/4*y^10 - 97/4*y^9 -
    45/8*y^8 + 33*y^7 - 25/2*y^6 - 18*y^5 + 107/8*y^4 + 5/8*y^3 -
    27/8*y^2 + 9/8*y - 1/8
> univgen(I, 3);
z^21 - z^20 - 2*z^19 + 4*z^18 - 5/2*z^17 - 5/2*z^16 + 4*z^15 -
    15/2*z^14 + 129/16*z^13 + 11/16*z^12 - 103/8*z^11 +
    131/8*z^10 - 49/16*z^9 - 171/16*z^8 + 12*z^7 - 3*z^6 -
    29/8*z^5 + 15/4*z^4 - 17/16*z^3 - 5/16*z^2 + 5/16*z - 1/16
```

115.7 Hilbert-driven Gröbner Basis Construction

Magma incorporates an implementation of the *Hilbert-driven Buchberger Algorithm* [Tra96]. This algorithm constructs the Gröbner basis of an homogeneous ideal I whose Hilbert series is known. The algorithm is often much more efficient than the conventional Buchberger algorithm since knowledge of the Hilbert series eliminates many unnecessary reductions of S-polynomials. The algorithm can also be used as an alternative to the Gröbner Walk algorithm for changing order since one can compute the Hilbert series of the ideal with respect to an easy monomial order, and then start again with the Hilbert-driven algorithm to compute the Gröbner basis with respect to the desired final order. Furthermore, the algorithm can sometimes be used to test whether an ideal has a particular Hilbert series and abort early if this is proven to be false. The algorithm is also used extensively internally in the Invariant Theory algorithms of Magma.

`HilbertGroebnerBasis(S, H): [ RngMPolElt ], FldFunRatUElt -> BoolElt, [ RngMPolElt ]`

`HilbertGroebnerBasis(S, N): [ RngMPolElt ], RngUPolElt -> BoolElt, [ RngMPolElt ]`

Let S be a set or sequence of homogeneous polynomials from the multivariate polynomial ring $P = K[x_1, \ldots, x_n]$, where K is a field, and let I be the ideal of P generated by S. Let either H be the Hilbert series $H_{P/I}(t)$ of I (as a rational function in $\mathbb{Z}(t)$) or let $N \in \mathbb{Z}[t]$ be a univariate integer polynomial such that the weighted numerator of the Hilbert series of I is N. This function attempts to construct the (reduced) Gröbner basis of I using the given Hilbert series. The weighted numerator of the Hilbert series of I is the Hilbert series $H_{P/I}(t)$ of I, multiplied by the denominator $\prod_{i=1}^{n} 1-t^{d_i}$, where d_i is the weighted degree of the i-th variable x_i (this denominator is thus $(1-t)^n$ if P has the default grading). If the function returns `false`, then H (or N) cannot be the correct Hilbert series (or weighted numerator of the Hilbert series) of I. Otherwise, the function returns `true` and a sequence B of polynomials which generates the same ideal as S; if H or N is correct, B will be the (reduced) Gröbner basis of I. In more detail, let f_H be the power series corresponding to the true Hilbert series of I and let f_N be the power series corresponding to $N/(\prod_{i=1}^{n} 1 - t^{d_i})$. If $f_H = f_N$, then the function returns `true` and the correct (reduced) Gröbner basis of I. Otherwise, consider the first term at which f_N and f_H differ: if the coefficient of f_N is greater than that of f_H, then the function returns `false` (since it will not be able to construct the extra Gröbner basis polynomials needed), otherwise the function will return `true` with a partial Gröbner basis (since it concludes that it has enough Gröbner basis polynomials when it hasn't). Consequently, the algorithm is usually used when the correct Hilbert series or weighted numerator of the Hilbert series is known, or when there is a weighted numerator which is known to be greater than or equal to the correct weighted numerator of the Hilbert series.

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

Change verbose printing for the Hilbert-driven Buchberger algorithm to be v. Currently the legal values for v are `true`, `false`, 0, or 1.

Example: Hilbert-Driven Groebner Basis for Invariants (ex-0965e4)

We illustrate a subalgorithm of the Invariant Theory module of Magma which uses the Hilbert-driven Buchberger Algorithm. Let R be the invariant ring of the (permutation) cyclic group G of order 4 over the field $K = \mathbf{F}_2$. Suppose we have a sequence L of 4 homogeneous invariants of degrees $1, 2, 2$, and 4 respectively. We wish to determine efficiently whether the polynomials of L constitute primary invariants for R. To check this, the ideal generated by L must be zero-dimensional and the elements of L must be algebraically independent. This is equivalent to the condition that the

weighted numerator of the Hilbert series of the ideal is the product $(1-t)(1-t^2)^2(1-t^4)$. If that is not the correct weighted numerator, it will be less than the correct weighted numerator so the algorithm will return whether the polynomials L do constitute primary invariants for R.

```
> K := GF(2);
> P<a,b,c,d> := PolynomialRing(K, 4);
> L := [
>     a + b + c + d,
>     a*b + a*d + b*c + c*d,
>     a*c + b*d,
>     a*b*c*d
> ];
> // Form potential Hilbert series weighted numerator
> T<t> := PolynomialRing(IntegerRing());
> N := &*[1 - t^Degree(f): f in L];
> N;
t^9 - t^8 - 2*t^7 + 2*t^6 + 2*t^3 - 2*t^2 - t + 1
> time l, B := HilbertGroebnerBasis(L, N);
Time: 0.000
> l;
true
> // Examine Groebner basis B of L:
> B;
[
    a + b + c + d,
    b^2 + d^2,
    b*c + b*d + c^2 + c*d,
    c^3 + c^2*d + c*d^2 + d^3,
    d^4
]
```

115.8 SAT solver

Magma V2.16 contains an interface to the MiniSat satisfiability (SAT) solver. Such a solver is given a system of boolean expressions in conjunctive normal form and determines whether there is an assignment in the variables such that all the expressions are satisfied. Magma supplies a function by which one may transform a system of boolean polynomial equations into an equivalent boolean system, and solve this via the SAT solver. To use the interface function, the MiniSat program must currently be installed as a command external to Magma. At the time of writing (November 2009), the latest version of MiniSat can be installed as follows on most Unix/Linux systems:

(1) Download `http://minisat.se/downloads/minisat2-070721.zip` from the MiniSat website (`minisat.se`).

(2) Use the command `unzip minisat2-070721.zip` or equivalent to unzip the files.

(3) Change directory into `minisat/core` and run `make` there.

(4) Copy the produced executable `minisat` into a place which is in the current path when Magma is run.

SAT(B): [RngMPolBoolElt] -> BoolElt, [FldFinElt]

SAT(B): [RngMPolElt] -> BoolElt, [FldFinElt]

```
Exclude: [ RngMPolElt ]                    Default: []
Verbose: BoolElt                           Default: true
```

Given a sequence B of boolean polynomials in a rank-n boolean polynomial ring (or a rank-n polynomial ring over $\mathbf{F}_2$), call MiniSat on the associated boolean system and return whether the

system is satisfiable, and if so, return also a solution S as a length-n sequence of elements of $\mathbf{F}_2$. (This assumes that MiniSat is in the executable path of external commands; see above for instructions for installing MiniSat). The parameter `Exclude` may be set to a sequence $[e_1, \ldots e_k]$, where each e_i is a sequence of n elements of $\mathbf{F}_2$, specifying that the potential solutions in e_i are to be excluded (this is done by adding new relations to the system to exclude the e_i). The verbose information printed by MiniSat may be controlled by the parameter `Verbose`.

Example: Solving a Boolean System with SAT (ex-a37e10)

In Example *Solving a Boolean Polynomial System*, we solved a boolean polynomial system via the standard Gröbner basis method (which the function `Variety` uses). Here we solve the same system via the SAT solver. Each time we obtain a solution, we can call the function again, but excluding the solution(s) already found. We can thus find all the solutions to the system. Of course, this is not worth doing when there are large numbers of solutions, but it may be of interest to find all solutions when it is expected there is a small number of solutions.

```
> P<a,b,c,d,e> := BooleanPolynomialRing(5, "grevlex");
> B := [a*b + c*d + 1, a*c*e + d*e, a*b*e + c*e, b*c + c*d*e + 1];
> l, S := SAT(B);
> l;
true
> S;
[ 1, 1, 1, 0, 0 ]
> Universe(S);
Finite field of size 2
> [Evaluate(f, S): f in B];
[ 0, 0, 0, 0 ]
> l, S2 := SAT(B: Exclude := [S]);
> l;
true
> S2;
[ 0, 1, 1, 1, 0 ]
> [Evaluate(f, S2): f in B];
[ 0, 0, 0, 0 ]
> l, S3 := SAT(B: Exclude := [S, S2]);
> l;
false
```

115.9 Bibliography

- [AL94] William Adams and Philippe Loustaunau. *An introduction to Gröbner bases.* Volume 3 of Graduate studies in mathematics. American Mathematical Society, Providence, R.I., 1994.
- [BW93] Thomas Becker and Volker Weispfenning. *Gröbner Bases.* Graduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 1993.
- [Buc65] Bruno Buchburger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal.* PhD thesis, University of Innsbruck, Austria, 1965.
- [CKM97] Stephane Collart, Michael Kalkbrener, and Daniel Mall. Converting Bases with the Gröbner Walk. *J. Symbolic Comp.*, 24(3):465–469, 1997.
- [CLO96] David Cox, John Little, and Donal O'Shea. *Ideals, Varieties and Algorithms.* Undergraduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 2nd edition, 1996.
- [CLO98] David Cox, John Little, and Donal O'Shea. *Using Algebraic Geometry.* Graduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 1998.
- [Fau99] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases (F_4). *Journal of Pure and Applied Algebra*, 139 (1-3):61–88, 1999.

- [FGLM93] Jean-Charles Faugère, Patrizia Gianni, Daniel Lazard, and Teo Mora. Efficient computations of zero-dimensional Gröbner bases by change of ordering. *J. Symbolic Comp.*, 16:329–344, 1993.
- [Mol88] H.M. Möller. On the construction of Gröbner bases using syzygies. *J. Symbolic Comp.*, 6:345–359, 1988.
- [Ste04] Allan Steel. Gröbner Basis Timings Page. http://magma.maths.usyd.edu.au/users/allan/gb/, 2004.
- [Tra96] Carlo Traverso. Hilbert Functions and the Buchberger Algorithm. *J. Symbolic Comp.*, 22(4):355–376, 1996.

Chapter 116

Polynomial Ring Ideal Operations

116.1 Introduction

This chapter describes the Magma functionality for ideals over polynomial rings. For the basics on multivariate polynomial rings and their elements, see Chapter Multivariate Polynomial Rings. Most of the significant operations with ideals construct or utilise a previously-constructed Gröbner basis. The monomial ordering used for this basis can greatly affect the speed and memory usage of these operations. This ordering is attached to the polynomial ring in which the ideals are created. For information on Gröbner bases and the creation of polynomial rings with specified orders, see Chapter *Gröbner Bases*. That chapter also tells the user how to compute and return a Gröbner basis, or just to compute it internally for later use in the operations described below, with many additional configuration parameters to optimise the computation. Users may ignore the issue when creating the ambient polynomial rings by allowing Magma to make default choices. It is, however, highly recommended that users who wish to work with complicated ideals thoroughly acquaint themselves with the options available. Magma has an extremely powerful Gröbner basis engine and often makes sophisticated choices internally of alternative monomial orders for particular computations. Ultimately, however, the user may significantly speed up his work by a judicious choice of order. We note here that the default order is the lexicographical one, a total elimination order well suited to finding solutions of zero-dimensional systems of polynomial equations but tending to produce very large bases that can take much time and memory to compute. For homogeneous ideals of rings with the standard weighting (all variables have weight one), the grevlex order is usually the best in practice and there is theoretical justification for this. In the case that the ring has a different weighting and the ideal is homogeneous with respect to that, the weighted grevlex order is the best choice. In any case, the `EasyIdeal` and `EasyBasis` intrinsics of the Gröbner basis chapter return to the user a basis for an internally chosen good order and these "easy" bases are used in many internal functions if a basis with respect to the polynomial ring order has not already been computed and stored. The functions and operations described here cover a wide range of commutative algebra functionality. This includes sums and intersections, colon ideals and saturations, elimination, radicals and primary decompositions, Noether normalisations and computation of Hilbert polynomials and Hilbert series. Related chapters including other polynomial ring functionality relying on Gröbner bases are the chapter on invariant rings of finite group actions, Chapter *Invariant Theory*, and the chapters on affine algebras (Chapter *Affine Algebras*) and on modules over affine algebras (Chapter *Modules over Multivariate Rings*). The chapter on algebraically closed fields (Chapter Algebraically Closed Fields) describes functions that allows one to compute the variety of an ideal over the algebraic closure of the base field. And, of course, the Algebraic Geometry component of Magma and parts of the Arithmetic Geometry are built upon the commutative algebra here.

116.2 Creation of Polynomial Rings and their Ideals

As noted in the introduction, for the basics on multivariate polynomial rings and their elements, including their creation, the user should refer to Chapter Multivariate Polynomial Rings. For creation of polynomial rings with non-default (currently lexicographic) monomial ordering, the user should refer to Chapter *Gröbner Bases*. Similarly, the basic creation functions for ideals and additional basis options are described

in Chapter *Gröbner Bases*. The commonest creation methods are the *ideal<P | L>* constructor and the *Ideal* function.

116.3 First Operations on Ideals

In the following, note that since ideals of a full polynomial ring P are regarded as subrings of P, the ring P itself is a valid ideal as well (the ideal containing 1).

116.3.1 Simple Ideal Constructions

The following basic constructions involve no Gröbner basis computation.

`I + J: RngMPol, RngMPol -> RngMPol`

Given ideals I and J of the same polynomial ring P, return the sum of I and J, which is the ideal generated by the generators of I and those of J.

`I * J: RngMPol, RngMPol -> RngMPol`

Given ideals I and J of the same polynomial ring P, return the product of I and J, which is the ideal generated by the products of the generators of I and those of J.

`I ^ k: RngMPol, RngIntElt -> RngMPol`

Given an ideal I of the polynomial ring P, and an integer k, return the k-th power of I.

`I / J: RngMPol, RngMPol -> RngMPolRes`

Given an ideal I of a polynomial ring P over a field and an ideal J of P, such that $J \subset I$, return the affine algebra I/J.

116.3.2 Basic Commutative Algebra Operations

The following important basic operations on ideals involve Gröbner basis computation and use the standard algorithms as described in Chapter 1.8 of [GP02], for example, unless otherwise stated.

`QuotientDimension(I): RngMPol -> RngIntElt`

Given an ideal I of a polynomial ring P over a field K, return the dimension of P/I as a K-vector space. Note that this is quite different from the function *Dimension* below (which returns the Krull dimension of an ideal). If I is not of Krull dimension 0 then the vector space is infinite and `Infinity` is returned.

`ColonIdeal(I, J): RngMPol, RngMPol -> RngMPol`

`IdealQuotient(I, J): RngMPol, RngMPol -> RngMPol`

Given ideals I and J of the same polynomial ring P, return the colon ideal $I : J$ (or ideal quotient of I by J), consisting of the polynomials f of P such that $f * g$ is in I for all g in J.

`ColonIdeal(I, f): RngMPol, RngMPolElt -> RngMPol, RngIntElt`

`IdealQuotient(I, f): RngMPol, RngMPolElt -> RngMPol, RngIntElt`

Given an ideal I and an element f of a polynomial ring P, return the saturation (colon) ideal $I : f^\infty$, consisting of the polynomials g of P such that there exists an $i \geq 1$ with $f^i * g \in I$. An integer s with $s \geq 1$ is also returned such that $I : f^\infty = I : f^s$. Note that if s is not needed, only one return value of the function should be expected which increases the efficiency enormously. Note also that this function is *not* equivalent to taking the ideal quotient of I by the ideal of P generated by f. It is in some ways a more natural operation mathematically, corresponding to taking the full inverse image of the localised ideal I_f under the localisation map $P \to P_f$, and can be faster than the $I : f$ computation, if s is not required. In this case, the computation goes by the elimination of extra variable t from the ideal $< I, 1 - f * t >$.

`ColonIdealEquivalent(I, f): RngMPol, RngMPolElt -> RngMPol, RngMPolElt`

Saturation(I, f): RngMPol, RngMPolElt -> RngMPol, RngMPolElt

Given an ideal I and an element f of a polynomial ring P, return the saturation (colon) ideal $C = I : f^\infty$, and a polynomial $g \in P$ such that $C = I : \langle g \rangle$ and g is of minimal degree. The irreducible factors of g will be a subset of the irreducible factors of f (and the corresponding multiplicities may be greater or lesser, depending on how often an irreducible factor divides the ideal I).

Saturation(I, J): RngMPol, RngMPol -> RngMPol

Given ideals I and J of some polynomial ring P, return the saturation $(I : J^\infty)$: that is, the ideal $\{f \in P : \exists n > 0, fJ^n \subseteq I\}$.

Saturation(I): RngMPol -> RngMPol

Given an ideal I of a polynomial ring P, return the saturation of I with respect to the irrelevant ideal of P – that is, the ideal of all elements of P having positive degree.

Generic(I): RngMPol -> RngMPol

Given an ideal I of a generic polynomial ring P, return P.

LeadingMonomialIdeal(I): RngMPol -> RngMPol

Given an ideal I, return the leading monomial ideal of I; that is, the ideal generated by all the leading monomials of I.

I meet J: RngMPol, RngMPol -> RngMPol

Given ideals I and J of the same polynomial ring P, return the intersection of I and J.

&meet S: [RngMPol] -> RngMPol

Given a set or sequence S of ideals of the same polynomial ring P, return the intersection of all the ideals of S.

RegularSequence(I): RngMPol -> SeqEnum

Homogeneous: BoolElt	Default: true

Given an ideal I of a polynomial ring P over a field, computes and returns a maximal regular sequence in I. The algorithm used is that of Eisenbud and Sturmfels ([ES94]) that tries to construct a regular sequence of fairly sparse polynomials. If parameter **Homogeneous** is **true** (the default), and I is a homogeneous ideal with respect to the variable weights, then the regular sequence constructed will also consist of homogeneous polynomials.

ReesIdeal(P, I): RngMPol, RngMPol -> RngMPol, Map

ReesIdeal(P, J, I): RngMPol, RngMPol, RngMPol -> RngMPol, Map

a: RngMPolElt	Default: 1

ReesIdeal(R, I): RngMPolRes, RngMPol -> RngMPol, Map

a: RngMPolElt	Default: 1

In each case P is a multivariate polynomial ring and I is an ideal of P. In the third case R is an affine quotient algebra of the form P/J. In the second case J is another ideal of P and we write R for the affine algebra P/J. In the first case, let $R = P$.

The Rees algebra $R(I)$ is the finitely-generated, graded polynomial algebra isomorphic to the algebra

$$R \oplus I \oplus I^2 \oplus I^3 \oplus \dots$$

where I gives the first graded part, I^2 the second etc. and the multiplication is the obvious one. Here I is thought of as an ideal of R, rather than P for the second and third signatures. *Proj* of this algebra represents the blow-up of the affine scheme $Spec(R)$ along the closed subscheme defined by I (see Chapter 2, Section 7 of [Har77]).

The function returns the *Rees ideal*, K, such that, if R_1 is the generic polynomial ring of K, then R_1/K is an affine algebra isomorphic to $R(I)/ < a - torsion >$, where a is an element of P (or R in the third case) that gives a non-zero divisor in R and is 1 by default. In the first case, any such a remains a non-zero divisor in $R(I)$, so is redundant. However, in the second and third cases, a can be specified to be not equal to 1 by use of the `a` parameter. Geometrically, dividing out by a-torsion gives the coordinate ring of the maximal closed subscheme of the blow-up that is flat over the generic point and the codimension one points defined by the vanishing of a, if these points are regular.

116.3.3 Ideal Predicates

`I eq J: RngMPol, RngMPol -> BoolElt`

Given two ideals I and J of the same polynomial ring P, return whether I and J are equal. Involves the use of a Gröbner basis for each ideal.

`I ne J: RngMPol, RngMPol -> BoolElt`

Given two ideals I and J of the same polynomial ring P, return whether I and J are not equal. Involves the use of a Gröbner basis for each ideal.

`I notsubset J: RngMPol, RngMPol -> BoolElt`

Given two ideals I and J in the same polynomial ring P return whether I is not contained in J. Involves the use of a Gröbner basis for J.

`I subset J: RngMPol, RngMPol -> BoolElt`

Given two ideals I and J in the same polynomial ring P return whether I is contained in J. Involves the use of a Gröbner basis for J.

`IsZero(I): RngMPol -> BoolElt`

Given an ideal I of the polynomial ring P, return whether I is the zero ideal (contains zero alone).

`IsProper(I): RngMPol -> BoolElt`

Given an ideal I of the polynomial ring P, return whether I is proper; that is, whether I is strictly contained in P, or whether the Gröbner basis of I does not contain 1 alone.

`IsHomogeneous(I): RngMPol -> BoolElt`

Given an ideal I of the polynomial ring P, this function returns whether I is homogeneous with respect to the weights on the variables of P (i.e., whether I possesses a basis consisting of homogeneous polynomials alone). Checks whether the current basis of I consists of homogeneous polynomials and, if not and the current basis isn't Gröbner, then whether an easy Gröbner basis consists of homogeneous elements.

`IsPrincipal(I): RngMPol -> BoolElt, RngMPolElt`

Given an ideal I of the polynomial ring P, return whether I is principal, and if so, return also a generator of I. This will be true if and only if an arbitrary Gröbner basis consists of a single (generating) element.

`IsPrimary(I): RngMPol -> BoolElt`

Given an ideal I of the polynomial ring P, return whether I is primary. An ideal I is primary if and only if for all $ab \in I$, either $a \in I$ or $b^n \in I$ for some $n \geq 1$. The restrictions on I are the same as for the function *PrimaryDecomposition*—see the description of that function. In general, this function computes or retrieves the primary decomposition and checks whether it has a unique element.

`IsPrime(I): RngMPol -> BoolElt`

Given an ideal I of the polynomial ring P, return whether I is prime. An ideal I is prime if and only if for all $ab \in I$, either $a \in I$ or $b \in I$. The restrictions on I are the same as for the function *PrimaryDecomposition*—see the description of that function. Again, this function computes the primary decomposition or uses the already stored one.

`IsMaximal(I): RngMPol -> BoolElt`

Given an ideal I of the polynomial ring P, return whether I is maximal. The restrictions on I are the same as for the function *PrimaryDecomposition*—see the description of that function. Checks first whether I is zero-dimensional (see below) and, if so, then checks whether it is prime. NB: given that I is of dimension 0, the prime/primary decomposition computation is relatively fast.

`IsRadical(I): RngMPol -> BoolElt`

Given an ideal I of the polynomial ring P, return whether I is radical; that is, whether the radical of I is I itself. The restrictions on I are the same as for the function *Radical*—see the description of that function. The function computes the radical or uses the already stored one.

`IsZeroDimensional(I): RngMPol -> BoolElt`

Given an ideal I of the polynomial ring P, defined over a field, return whether I is zero-dimensional (so the quotient of P by I has non-zero finite dimension as a vector space over the coefficient field – see the section on dimension for further details). Note that the full polynomial ring P as an ideal of itself has dimension -1, so it is not zero-dimensional.

`HasGrevlexOrder(I): RngMPol -> BoolElt`

Given an ideal I of the polynomial ring P, return whether the monomial order of I is the `grevlex` order.

Example: Ideal Arithmetic (ex-d1b8ee)

We construct some ideals in $\mathbb{Q}[x, y, z]$ and perform basic arithmetic on them.

```
> P<x,y,z> := PolynomialRing(RationalField(), 3);
> I := ideal<P | x*y - 1, x^3*z^2 - y^2, x*z^3 - x - 1>;
> J := ideal<P | x*y - 1, x^2*z - y, x*z^3 - x - 1>;
> A := I * J;
> A;
Ideal of Polynomial ring of rank 3 over Rational Field
Order: Lexicographical
Variables: x, y, z
Basis:
[
    x^2*y^2 - 2*x*y + 1,
    x^3*y*z - x^2*z - x*y^2 + y,
    x^2*y*z^3 - x^2*y - x*y - x*z^3 + x + 1,
    x^4*y*z^2 - x^3*z^2 - x*y^3 + y^2,
    x^5*z^3 - x^3*y*z^2 - x^2*y^2*z + y^3,
    x^4*z^5 - x^4*z^2 - x^3*z^2 - x*y^2*z^3 + x*y^2 + y^2,
    x^2*y*z^3 - x^2*y - x*y - x*z^3 + x + 1,
    x^3*z^4 - x^3*z - x^2*z - x*y*z^3 + x*y + y,
    x^2*z^6 - 2*x^2*z^3 + x^2 - 2*x*z^3 + 2*x + 1
]
> M := I meet J;
> M;
Ideal of Polynomial ring of rank 3 over Rational Field
Order: Lexicographical
Variables: x, y, z
Basis:
[
    x^4 + x^3 - x*z^2 + z^12 - 4*z^9 + 6*z^6 - z^4 - 4*z^3 + z + 1,
    x^5 + x^4 - x^2*z^2 + z^9 - 3*z^6 + 3*z^3 - z - 1,
    x*z^3 - x - 1,
    y - z^3 + 1
]
> A eq M;
true
```

(continues on next page)

(continued from previous page)

```
> QuotientDimension(A);
24
> ColonIdeal(I, J);
Ideal of Polynomial ring of rank 3 over Rational Field
Order: Lexicographical
Variables: x, y, z
Inhomogeneous, Dimension 0
Basis:
[
    x*y - 1,
    x^3*z^2 - y^2,
    x*z^3 - x - 1
]
```

116.3.4 Element Operations with Ideals

`f in I: RngMPolElt, RngMPol -> BoolElt`

Given a polynomial f from a polynomial ring P, together with an ideal I of P, return whether f is in I. The function computes the normal form of f relative to some Gröbner basis of I and checks if this is zero.

`f notin I: RngMPolElt, RngMPol -> BoolElt`

Given a polynomial f from a polynomial ring P, together with an ideal I of P, return whether f is not in I. As with `in`, this performs a normal form computation.

`IsInRadical(f, I): RngMPolElt, RngMPol -> BoolElt`

Given a polynomial f from a polynomial ring P, together with an ideal I of P, return whether f is in the radical of I. Note that using this function is much quicker in general than actually computing the radical of I. It uses the algorithm described in section 1.8.6 of [GP02].

`JacobianIdeal(f): RngMPolElt -> RngMPol`

Return the ideal generated by all first partial derivatives of the polynomial f.

Example: Element Operations with Ideals `(ex-273d4e)`

We demonstrate the element operations with respect to an ideal of $\mathbb{Q}[x, y, z]$.

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> I := ideal<P | (x + y)^3, (y - z)^2, y^2*z + z>;
> NormalForm(y^2*z + z, I);
0
> NormalForm(x^3, I);
-3*x^2*y - 3*x*z^4 - 6*x*z^2 + 1/2*z^3 + 3/2*z
> NormalForm(z^4 + y^2, I);
2*z^4 + 2*z^2
> x + y in I;
false
> IsInRadical(x + y, I);
true
> IsInRadical((x + y)^2, I);
true
> IsInRadical(z, I);
false
> SPolynomial(x^4 + y - z, x^2 + y - z);
-x^2*y + x^2*z + y - z
```

116.4 Computation of Varieties

The slightly non-standard term variety in this section refers to the (finite) solution set of the system of polynomials that make up an ideal. It can also be thought of as the set of points of a zero-dimensional affine scheme over a specified field extension of the polynomial ring base field. For more general functionality for schemes of arbitrary dimension, see the chapters on Algebraic and Arithmetic Geometry. The functions here also work for higher-dimensional ideals if the base field is finite, when the solution set is again finite (over the base or a finite extension of the base).

The functions compute solutions over the base field of the polynomial ring or over an extension field L. Magma's algebraically closed fields (see Chapter Algebraically Closed Fields) may be used to get all solutions if so desired when an explicit splitting field is not known for the system. L should be an exact field over which Magma has a root-finding algorithm for univariate polynomials or a real or complex field.

For the corresponding functions with argument a zero-dimensional scheme which may not be affine, see the Section Rational Points and Point Sets in the Schemes chapter.

`Variety(I): RngMPol -> [ ModTupFldElt ]`

`Variety(I, L): RngMPol, Rng -> [ ModTupFldElt ]`

```
Al  : MonStgElt                    Default: "Default"
Eval: RngIntElt                    Default:
```

Given an ideal I of a polynomial ring P, return the affine variety of I over its coefficient field K as a sequence of tuples. Each tuple is of length n, where n is the rank of P, and corresponds to an assignment of the n variables of P (in order) such that all polynomials in I vanish with this assignment.

If K is not a finite field then the ideal must be the full polynomial ring or be zero-dimensional so that the variety is known to be finite. If a superfield L of K is also given, the variety is computed over L instead, so the entries of the tuples lie in L.

`Al := "Wiedemann"`: This parameter selects the Wiedemann algorithm and is applicable to computing the variety of a generic zero-dimensional ideal I over K, where K is a moderately sized finite field. The *degree* D of I is the dimension of the quotient algebra R/I, and as D grows large (particularly above 10000), computing the variety of I becomes very difficult since it involves computing a minimal polynomial of degree D. The Wiedemann algorithm is effective when it is impractical to store a fully dense $D \times D$ matrix in memory. See `http://magma.maths.usyd.edu.au/users/allan/densef4/#CC` for a very large example.

`Al := "KellerGehrig"`: This parameter selects the Keller-Gehrig algorithm, which is similar to the Wiedemann algorithm in that it finds the minimal polynomial of a matrix to compute a variety of a zero-dimensional ideal, but it uses the Keller-Gehrig algorithm to compute this minimal polynomial by mapping the problem to several matrix multiplications. This method typically takes more memory than the Wiedemann algorithm but may be faster for dimensions in the thousands, particularly when a GPU is available.

`Al := "ExhaustiveSearch"`: This parameter selects the Exhaustive Search algorithm for solving quadratic multivariate polynomial systems over $\mathbf{F}_2$. For a system with n variables, the algorithm simply enumerates all 2^n possible assignments of the variables and determines the set of all such assignments which simultaneously satisfy all the polynomial equations. Despite the obvious exponential complexity of this brute force approach, the algorithm is highly optimised: for a quadratic input system, the algorithm covers 10^{10} possible solutions in 1.47 seconds on a typical 3.2GHz Intel Xeon processor and an arbitrary quadratic system with 40 variables is solved in 162 seconds. The algorithm is preferable to other algorithms for certain types of input system. For example, in the case of an arbitrary generic dense quadratic system with no structure (such as a random system), the exhaustive search algorithm is typically faster than both the Gröbner basis and SAT algorithms by a factor of more than 100. This is because all algorithms have exponential complexity for this type of input, but the associated constant is much better for the exhaustive search.

`Eval := e`: If `Eval` is set to a small positive integer e, then the function works by evaluating the input polynomials at the last e variables to all possible constants in K (so there are q^e cases enumerated) and then computing the variety of each simplified ideal and combining all the results. This algorithm can be parallelised by the use of `SetNthreads` and `StartWorkers`.

`VarietySequence(I): RngMPol -> [ [ RngElt ] ]`

`VarietySequence(I, L): RngMPol, Rng -> [ [ RngElt ] ]`

Given a zero-dimensional ideal I of a polynomial ring P whose order is of lexicographic type, return the variety of I over its coefficient field K as a sequence of sequences of elements of K. Each inner sequence is of length n, where n is the rank of P, and corresponds to an assignment of the n variables of P (in order) such that all polynomials in I vanish with this assignment. The parameters and other conditions are the same as for `Variety` above.

`VarietySize(I): RngMPol -> RngIntElt`

Given an ideal I of a polynomial ring P over a field K, return the size of the variety of I over K. The parameters are the same as for the function `Variety`.

Example: Variety over a Finite Field `(ex-eb5dde)`

We construct an ideal I of the polynomial ring $\mathbf{F}_{27}[x, y]$, and then find the variety $V = V(I)$. We then check that I vanishes on V.

```
> K<w> := GF(27);
> P<x, y> := PolynomialRing(K, 2);
> I := ideal<P | x^8 + y + 2, y^6 + x*y^5 + x^2>;
> Groebner(I);
> I;
Ideal of Polynomial ring of rank 2 over GF(3^3)
Order: Lexicographical
Variables: x, y
Inhomogeneous, Dimension 0
Groebner basis:
[
    x + 2*y^47 + 2*y^45 + y^44 + 2*y^43 + y^41 + 2*y^39 + 2*y^38 + 2*y^37 +
        2*y^36 + y^35 + 2*y^34 + 2*y^33 + y^32 + 2*y^31 + y^30 + y^28 + y^27 +
        y^26 + y^25 + 2*y^23 + y^22 + y^21 + 2*y^19 + 2*y^18 + 2*y^16 + y^15 +
        y^13 + y^12 + 2*y^10 + y^9 + y^8 + y^7 + 2*y^6 + y^4 + y^3 + y^2 + y +
        2,
    y^48 + y^41 + 2*y^40 + y^37 + 2*y^36 + 2*y^33 + y^32 + 2*y^29 + y^28 +
        2*y^25 + y^24 + y^2 + y + 1
]
> V := Variety(I);
> V;
[ <w^14, w^12>, <w^16, w^10>, <w^22, w^4> ]
> // Check that the original polynomials vanish:
> [
>    <x^8 + y + 2, y^6 + x*y^5 + x^2> where x is v[1] where y is v[2]: v in V
> ];
[ <0, 0>, <0, 0>, <0, 0> ]
> // Note that the variety of I would be larger over an extension field of K:
> VarietySizeOverAlgebraicClosure(I);
48
```

116.5 Multiplicities

This section contains some useful invariants for an isolated singularity at the origin of a hypersurface given by a multivariate polynomial f.

`MilnorNumber(f): RngMPolElt -> RngElt`

Given a polynomial $f \in K[x_1, \ldots, x_n]$, where K is a field, return the Milnor number of f at the origin. This is the dimension of the quotient by the ideal generated by the partials of f in the localization of $K[x_1, \ldots, x_n]$ at the origin. See [CLO98, p. 147] or [DL06, Remark 9.37].

`TjurinaNumber(f): RngMPolElt -> RngElt`

Given a polynomial $f \in K[x_1, \ldots, x_n]$, where K is a field, return the Tjurina number of f at the origin. This is the dimension of the quotient by the ideal generated by f and the partials of f in the localization of $K[x_1, \ldots, x_n]$ at the origin. See [CLO98, p. 148] or [DL06, Def. 9.35].

Example: Milnor and Tjurina Numbers `(ex-e4a04f)`

We compute some Milnor and Tjurina numbers, based on Exercise 12 of [CLO98, p. 177].

```
> P<x,y> := PolynomialRing(RationalField(), 2);
> MilnorNumber((x^2 + y^2)^3 - 4*x^2*y^2); // 4-leaved rose
13
> [MilnorNumber(y^2 - x^n): n in [1 .. 5]];
[ 0, 1, 2, 3, 4 ]
> P<x,y,z> := PolynomialRing(RationalField(), 3);
> [MilnorNumber(x*y*z + x^n + y^n + z^n): n in [1 .. 10]];
[ 0, 1, 8, 11, 14, 17, 20, 23, 26, 29 ]
> [TjurinaNumber(x*y*z + x^n + y^n + z^n): n in [1 .. 10]];
[ 0, 1, 8, 10, 13, 16, 19, 22, 25, 28 ]
```

A much larger example is given in [DL06, p. 254].

```
> P<x,y> := PolynomialRing(RationalField(), 2);
> f := y^2 - 2*x^28*y - 4*x^21*y^17 + 4*x^14*y^33 - 8*x^7*y^49 +
>      x^56 + 20*y^65 + 4*x^49*y^16;
> time TjurinaNumber(f);
2260
Time: 0.010
```

116.6 Elimination

Elimination theory plays an important role when working with ideals of multivariate polynomial rings. Magma provides an assortment of functions to perform various kinds of elimination easily. Elimination of variables is accomplished by computing a Gröbner basis with respect to a suitable elimination order (for more information about elimination orders, see Section *Representation and Monomial Orders* as well as comments in the function descriptions below). All of the functions in this section may be applied to ideals over general Euclidean rings, not just over fields.

116.6.1 Construction of Elimination Ideals

`EliminationIdeal(I, k: parameters): RngMPol, RngIntElt -> RngMPol`

Given an ideal I of a polynomial ring P of rank n with $P = R[x_1, \ldots, x_n]$, together with an integer k with $0 \leq k \leq n$, return the k-th elimination ideal I_k of I, which is defined to be $I \cap R[x_{k+1}, \ldots, x_n]$. Thus I_k consists of all polynomials of I which have the first k variables eliminated. If the elimination ideals I_k are to be computed for several different k, it is recommended first that a Gröbner basis with respect to lexicographical order for I first be computed as then the elimination ideals can be determined trivially. If I does not have a Gröbner basis stored with respect to lexicographical order, then a Gröbner basis computation will be necessary each time an elimination ideal is desired. If $k = n$, then $I \cap R$ is returned, which, if R is a field, is always the full ring P or the empty ideal, according to whether I is the full polynomial ring or not. But if R is not a field, then this intersection will yield the ideal generated by the normalized smallest element of R which is in I (according to the Euclidean norm), which could be neither 0 nor 1. The parameters are as for the *Groebner* procedure. Note that setting `Al := "Direct"` occasionally produces much better

performance since the relevant elimination order may yield a better Gröbner basis than the default method of going via the `grevlex` order.

`EliminationIdeal(I, S): RngMPol, { RngIntElt } -> RngMPol`

`EliminationIdeal(I, S): RngMPol, { RngMPolElt } -> RngMPol`

Given an ideal I of a polynomial ring P of rank n with $P = R[x_1, \ldots, x_n]$, together with a set S describing a subset U of the variables $\{x_1, \ldots x_n\}$, return the elimination ideal I_U of I, which is defined to be $I \cap R[U]$. Thus I_U consists of all polynomials of I which contain variables only found in U. U can be specified in two ways: either as a set S of integers in the range $1 \ldots n$ such the integer i corresponds to the i-th variable x_i, or as a set S of variables lying in P. S may be the empty set, in which case this is equivalent to `EliminationIdeal(I, n)`; see above.

Example: Elimination over a Quadratic Order `(ex-0217ea)`

This example continues the example above which computed a Gröbner basis over a ring which was not even a PIR.

As before, $R = \mathbb{Z}[\sqrt{-5}]$ and I is the ideal of $R[x, y]$ generated by $f_1 = 2xy + \sqrt{-5}y$ and $f_2 = (1 + \sqrt{-5})x^2 - xy$. As before, we compute over $\mathbb{Z}$, introduce a new variable S and include $(S^2 + 5)$ in I, so we can effectively work over R.

```
> P<x, y, S> := PolynomialRing(IntegerRing(), 3);
> f1 := 2*x*y + S*y;
> f2 := (1 + S)*x^2 - x*y;
> I := ideal<P | f1, f2, S^2 + 5>;
> GroebnerBasis(I);
[
    x^2*S + x^2 + 5*y^3 + 13*y*S - 25*y,
    6*x^2 + 5*y^2 + 3*y*S - 10*y,
    x*y + 5*y^3 + 13*y*S - 25*y,
    y^2*S + 5*y^2 - 15*y,
    10*y^2 + 5*y*S - 25*y,
    S^2 + 5
]
```

In [AL94, Ex. 4.3.8], the elimination ideal $E_y = I \cap (\mathbb{Z}[\sqrt{-5}])[y]$ is shown to be generated by $f_5 = (5 + \sqrt{-5})y^2 - 15y$ and $f_7 = -2\sqrt{-5}y^2 + 5(1 + \sqrt{-5})y$. We can compute E_y in Magma easily using `EliminationIdeal`. We must be careful to include S in the second argument (the set of variables which we want), since S should be considered a 'constant' (member of R) in this context.

```
> Ey := EliminationIdeal(I, {y, S});
> GroebnerBasis(Ey);
[
    y^2*S + 5*y^2 - 15*y,
    10*y^2 + 5*y*S - 25*y,
    S^2 + 5
]
```

Obviously, the polynomials yielded are simply the last 3 polynomials of the full Gröbner basis given above. We check also that the ideal generated by f_5 and f_7 over R is the same as that given by Magma.

```
> f_5 := y^2*S + 5*y^2 - 15*y;
> f_7 := -2*y^2*S + 5*y*S + 5*y;
> E := ideal<P | f_5, f_7, S^2 + 5>;
> E eq Ey;
true
```

Finally, we also compute $E_x = I \cap (\mathbb{Z}[\sqrt{-5}])[x]$, which requires more effort this time, since it cannot be read off the Gröbner basis.

```
> Ex := EliminationIdeal(I, {x, S});
> GroebnerBasis(Ex);
[
    2*x^3*S + 2*x^3 + x^2*S - 5*x^2,
    12*x^3 + 6*x^2*S,
    S^2 + 5
]
```

From this, we see that E_x is generated by $(2+2\sqrt{-5})x^3 + (-5+\sqrt{-5})x^2$ and $12x^3 + 6\sqrt{-5}x^2$.

116.6.2 Univariate Elimination Ideal Generators

`UnivariateEliminationIdealGenerator(I, i): RngMPol, RngIntElt -> RngMPolElt`

Given a zero-dimensional ideal I of a polynomial ring P of rank n with $P = K[x_1, \ldots, x_n]$, together with an integer i with $1 \leq i \leq n$, return the unique monic generator of the univariate elimination ideal $I \cap K[x_i]$.

`UnivariateEliminationIdealGenerators(I): RngMPol -> [ RngMPolElt ]`

Given a zero-dimensional ideal I of a polynomial ring P of rank n with $P = K[x_1, \ldots, x_n]$, return the sequence of length n whose i-th element is the unique monic generator of the univariate elimination ideal $I \cap K[x_i]$.

Example: Elimination Ideals from Neural Network Theory `(ex-09af2f)`

We construct an ideal I (derived from Neural networks theory) of the polynomial ring $\mathbb{Q}[x, y, z]$, and then find various elimination ideals of I.

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> I := ideal<P |
>     1 - x + x*y^2 - x*z^2,
>     1 - y + y*x^2 + y*z^2,
>     1 - z - z*x^2 + z*y^2 >;
> UnivariateEliminationIdealGenerator(I, 1);
x^21 - x^20 - 2*x^19 + 4*x^18 - 5/2*x^17 - 5/2*x^16 + 4*x^15 - 15/2*x^14 +
    129/16*x^13 + 11/16*x^12 - 103/8*x^11 + 131/8*x^10 - 49/16*x^9 -
    171/16*x^8 + 12*x^7 - 3*x^6 - 29/8*x^5 + 15/4*x^4 - 17/16*x^3 - 5/16*x^2
    + 5/16*x - 1/16
> UnivariateEliminationIdealGenerator(I, 2);
y^14 - y^13 - 13/2*y^12 + 8*y^11 + 53/4*y^10 - 97/4*y^9 - 45/8*y^8 + 33*y^7 -
    25/2*y^6 - 18*y^5 + 107/8*y^4 + 5/8*y^3 - 27/8*y^2 + 9/8*y - 1/8
> E :=  EliminationIdeal(I, {y, z});
> E;
Ideal of Polynomial ring of rank 3 over Rational Field
Order: Lexicographical
Variables: x, y, z
Basis:
[
    z^21 - z^20 - 2*z^19 + 4*z^18 - 5/2*z^17 - 5/2*z^16 + 4*z^15 - 15/2*z^14 +
        129/16*z^13 + 11/16*z^12 - 103/8*z^11 + 131/8*z^10 - 49/16*z^9 -
        171/16*z^8 + 12*z^7 - 3*z^6 - 29/8*z^5 + 15/4*z^4 - 17/16*z^3 - 5/16*z^2
        + 5/16*z - 1/16,
    y + 141944208/7806653*z^20 - 42803108/7806653*z^19 - 290535348/7806653*z^18
        + 309392460/7806653*z^17 - 164881460/7806653*z^16 -
        331099258/7806653*z^15 + 203830442/7806653*z^14 - 894960798/7806653*z^13
        + 622205873/7806653*z^12 + 1352184655/31226612*z^11 -
        4746138097/31226612*z^10 + 5122044359/31226612*z^9 +
        991547639/31226612*z^8 - 830598655/7806653*z^7 + 1472712995/15613306*z^6
        - 59983627/15613306*z^5 - 486698319/15613306*z^4 + 174173263/7806653*z^3
```

(continues on next page)

(continued from previous page)

```
        - 30672252/7806653*z^2 - 735083/664396*z + 30093391/31226612
]
```

Example: Radical of a Zero-Dimensional Ideal (ex-84cd6f)

We write a simple function ZRadical to compute the radical of a zero dimensional ideal defined over a field using the univariate elimination ideal generators. See [BW93, p. 345].

```
> function ZRadical(I)
>     // Find radical of zero dimensional ideal I
>     P := Generic(I);
>     n := Rank(P);
>     G := UnivariateEliminationIdealGenerators(I);
>     N := {};
>
>     for i := 1 to n do
>         // Set FF to square-free part of the i-th univariate
>         // elimination ideal generator
>         F := G[i];
>         FF := F;
>         while true do
>             D := GCD(FF, Derivative(FF, 1, i));
>             if D eq 1 then
>                 break;
>             end if;
>             FF := FF div D;
>         end while;
>         // Include FF in N if FF is a proper divisor of F
>         if FF ne F then
>             Include(~N, FF);
>         end if;
>     end for;
>
>     // Return the sum of I and N
>     if #N eq 0 then
>       return I;
>     else
>       return ideal<P | I, N>;
>     end if;
> end function;
```

We now apply ZRadical to an ideal of $\mathbb{Q}[x, y, z]$.

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> I := ideal<P | (x+1)^3*y^4, x*(y-z)^2+1, z^3-z^2>;
> R := ZRadical(I);
> Groebner(I);
> Groebner(R);
> I;
Ideal of Polynomial ring of rank 3 over Rational Field
Order: Lexicographical
Variables: x, y, z
Inhomogeneous, Dimension 0, Non-radical
Groebner basis:
[
    x - 4*y^9 + 21*y^8 - 32*y^7 + 7*y^6 + 432*y^5*z^2 - 546*y^5*z + 120*y^5 -
        137*y^4*z^2 + 288*y^4*z - 146*y^4 - 956*y^3*z^2 + 1088*y^3*z - 128*y^3 +
```

(continues on next page)

(continued from previous page)

```
        393*y^2*z^2 - 576*y^2*z + 186*y^2 + 498*y*z^2 - 540*y*z + 44*y - 220*z^2
        + 288*z - 67,
    y^10 - 6*y^9 + 12*y^8 - 8*y^7 + 288*y^5*z^2 - 348*y^5*z + 60*y^5 -
        110*y^4*z^2 + 192*y^4*z - 82*y^4 - 624*y^3*z^2 + 696*y^3*z - 72*y^3 +
        273*y^2*z^2 - 384*y^2*z + 111*y^2 + 322*y*z^2 - 348*y*z + 26*y - 150*z^2
        + 192*z - 42,
    y^6*z - y^6 - 6*y^5*z^2 + 6*y^5*z - 3*y^4*z + 3*y^4 + 12*y^3*z^2 - 12*y^3*z
        + 3*y^2*z - 3*y^2 - 6*y*z^2 + 6*y*z - z + 1,
    z^3 - z^2

> R;
Ideal of Polynomial ring of rank 3 over Rational Field
Order: Lexicographical
Variables: x, y, z
Inhomogeneous, Dimension 0, Radical
Groebner basis:
[
    x + 1,
    y^2 - 2*y*z + z - 1,
    z^2 - z
]
> I subset R;
true
> R subset I;
false
> IsInRadical(x + 1, I);
true
```

116.6.3 Relation Ideals

`RelationIdeal(Q): [ RngMPol ] -> RngMPol`

`RelationIdeal(Q, T): [ RngMPol ], RngMPol -> RngMPol`

Given a sequence Q of k polynomials of a polynomial ring P over a ring S (not necessarily a field), return the relation ideal U of Q which is an ideal of the polynomial ring of rank k over S containing all algebraic relations between the elements of Q. That is, U consists of all polynomials $r \in S[y_1, \ldots, y_k]$ such that $r(Q[1], \ldots, Q[k]) = 0$. If U is desired to be an ideal of a particular polynomial ring T of rank k (to obtain predetermined names of variables, for example), then T may be passed as a second argument.

The computation is the same as that for the image of an affine polynomial map, which this basically is, thinking of the polynomials in Q as giving a map from n-dimensional affine space (n = rank of P) to k-dimensional affine space. k new variables y_i and relations $y_i - Q[i]$ are added and then the original variables x_i of P are eliminated in the usual way.

Example: Expressing One in Terms of Generators (ex-d17522)

We construct an ideal I of the polynomial ring $\mathbf{F}_2[x, y, z]$, and discover that the ideal is the full polynomial ring. Suppose we then wish to write $1 \in I$ as an (algebraic) expression in terms of the original generators. We use `RelationIdeal` to find that expression.

```
> P<x, y, z> := PolynomialRing(GF(2), 3, "grevlex");
> S := [(x + y + z)^2, (x^2 + y^2 + z^2)^3 + x + y + z + 1];
> I := ideal<P | S>;
> Groebner(I);
> I;
Ideal of Polynomial ring of rank 3 over GF(2)
Graded Reverse Lexicographical Order
```

(continues on next page)

(continued from previous page)

```
Variables: x, y, z
Groebner basis:
[
    1
]
> Q<a, b> := PolynomialRing(GF(2), 2);
> U := RelationIdeal(S, Q);
> U;
Ideal of Polynomial ring of rank 2 over GF(2)
Order: Lexicographical
Variables: a, b
Inhomogeneous, Dimension >0
Basis:
[
    a^6 + a + b^2 + 1
]
```

Finally, we check the algebraic expression, evaluating it at the original polynomials:

```
> S[1]^6 + S[1] + S[2]^2;
1
```

116.7 Variable Extension of Ideals

Often one wishes to introduce new variables temporarily to a polynomial ring. Magma allows one to do this by use of the `VariableExtension` function, and also to restrict again to the original ring with elimination performed automatically.

`VariableExtension(I, k, b): RngMPol, RngIntElt, BoolElt -> RngMPol, Map`

`VariableExtension(I, k, b, order): RngMPol, RngIntElt, BoolElt, ... -> RngMPol, Map`

Given an ideal I of the polynomial ring $P = R[x_1, \ldots, x_n]$, create a polynomial ring Q as a k-variable extension of P, the ideal J of Q corresponding to I, and the embedding map $f : P \to Q$, and return J and f.

If the argument b (standing for "before") is `true`, the k variables are inserted before the current variables of P, so Q is defined to be $R[y_1, \ldots, y_k, x_1, \ldots, x_n]$ and f maps $P.i$ to $Q.(k + i)$ (so the x_i variables of P are mapped to the x_i variables of Q).

If the argument b is `false`, the k variables are inserted after the current variables of P, so Q is defined to be $R[x_1, \ldots, x_n, y_1, \ldots, y_k]$ and f maps $P.i$ to $Q.i$ (so the x_i variables of P are mapped to the x_i variables of Q).

If the argument order is given, then Q is constructed with the specified order; otherwise, the `grevlex` order is used for Q by default. See the section on monomial orders (Section *Representation and Monomial Orders*) for the valid values for the argument order.

The image under f of a polynomial of P is the corresponding polynomial of Q, while the image under f of an ideal of P is the corresponding ideal of Q. The inverse image under f of a polynomial of Q is only defined if none of the extension variables of Q occur in that polynomial, in which case the inverse image is just the restriction back to P, while the inverse image under f of an ideal H of Q is always defined and is the restriction back to P of the elimination ideal $H \cap R[x_1, \ldots, x_n]$.

116.8 Homogenization of Ideals

Magma allows one to homogenize a polynomial ring or ideal by use of the `Homogenization` function, and also to restrict again to the original ring with elimination performed automatically.

`Homogenization(I, b): RngMPol, RngIntElt, BoolElt -> RngMPol, Map`

`Homogenization(I, b, order): RngMPol, RngIntElt, BoolElt, ... -> RngMPol, Map`

`Homogenization(I): RngMPol, RngIntElt, BoolElt -> RngMPol, Map`

`Homogenization(I, order): RngMPol, RngIntElt, BoolElt, ... -> RngMPol, Map`

Given an ideal I of the polynomial ring $P = R[x_1, \ldots, x_n]$, create a polynomial ring H as a single variable extension of P, the homogenized ideal J of H corresponding to I, and the homogenization map $f : P \to H$, and return J and f.

If the argument b (standing for "before") is `true`, the homogenization variable is inserted before the current variables of P, so H is defined to be $R[h, x_1, \ldots, x_n]$ and f maps $P.i$ to $H.(k + i)$ (so the x_i variables of P are mapped to the x_i variables of H).

If the argument b is `false`, the homogenization variable is inserted after the current variables of P, so H is defined to be $R[x_1, \ldots, x_n, h]$ and f maps $P.i$ to $H.i$ (so the x_i variables of P are mapped to the x_i variables of H).

If the argument b is omitted, it is taken to be `false`, so the homogenization variable is introduced after the current variables of P.

If the argument order is given, then H is constructed with the specified order; otherwise, the `grevlex` order is used for H by default. See the section on monomial orders (Section *Representation and Monomial Orders*) for the valid values for the argument order.

The image under f of a polynomial of P is the homogenization of f in H, while the image under f of an ideal of P is the homogenization ideal I^h in H. The inverse image under f of a polynomial of H is the restriction back to P (obtained by setting the homogenization variable to 1), while the inverse image under f of an ideal J of H is the restriction back to P of the ideal obtained by setting the homogenization variable to 1.

116.9 Extension and Contraction of Ideals

Magma allows the extension to and contraction from the ring of quotients of an ideal, defined over a field, with respect to certain variables. See [BW93, pp. 54--58 and 388--397] for the relevant definitions and theory.

`Extension(I, U): RngMPol, [ RngIntElt ] -> RngMPol, Map`

Given an ideal I of the polynomial ring $P = K[x_1, \ldots, x_n]$, where K is a field, together with a sequence U of integers each between 1 and n, create the (ring of quotients) extension Q of P, and return the ideal J of Q, together with the map $f : P \to Q$.

If U has length k and the values (in order) of U are $u_1, \ldots, u_k$, then first the rational function field $F = K(x_{u_1}, \ldots, x_{u_k})$ is constructed, then the list $v_1, \ldots, v_{n-k}$ is constructed as the list $1, \ldots, n$ with the u_i removed, and finally the extension Q of P is defined to be the polynomial ring $F[x_{v_1}, \ldots, x_{v_{n-k}}] = K(x_{u_1}, \ldots, x_{u_k})[x_{v_1}, \ldots, x_{v_{n-k}}]$.

The map f is constructed in the obvious way so that x_i is mapped to the appropriate variable in F if i is in U, or the appropriate variable in Q otherwise. The image under f of an ideal of P is just the appropriate ideal of Q whose basis is obtained by taking the image under f of each of the polynomials in the basis of I.

The inverse image under f of a polynomial of Q is obtained by first making the polynomial monic, then multiplying by the LCM of the denominators ("clearing the denominators"), then mapping each variable back to the appropriate one in P—this is possible since there are no proper denominators. The inverse image under f of an ideal H of Q is defined to be the ideal of P generated by the inverse images under f of the polynomials in the basis of H (note that this is not always equal to the contraction of H—see [BW93, p. 389], for a simple algorithm to compute the contraction of an ideal of Q).

116.10 Dimension of Ideals

Let I be an ideal of the polynomial ring $P = K[x_1, \ldots, x_n]$, where K is a field. Let X be the set $\{x_1, \ldots, x_n\}$ of variables of P. A subset U of X is called *independent modulo* I if $I \cap K[U] = \emptyset$. A subset U of X is called *maximally independent modulo* I if U is independent modulo I, and no proper superset of U is independent modulo I. The *dimension* of I is defined to be the maximum of the cardinalities of all the independent sets modulo I. It is not too hard to see in this case that this coincides with the more abstract commutative algebra definition of the Krull dimension of the *quotient algebra* P/I as the maximal length of a chain of prime ideals. Note that the definition given above of zero-dimensionality (as the case when the quotient of P by I has finite dimension as a vector space over the coefficient field) coincides with the definition of zero-dimensionality as dimension 0.

`Dimension(I): RngMPol -> RngIntElt, [ RngIntElt ]`

Given an ideal I of a polynomial ring P defined over a field, return the dimension d of I, together with a (sorted) sequence U of integers of length d such that the variables of P corresponding to the integers of U constitute a maximally independent set modulo I. If I is the full polynomial ring P, the dimension is defined to be -1, and the second return value is not set. The algorithm implemented is that given in [BW93, p. 449].

116.11 Radical and Decomposition of Ideals

Magma has algorithms for computing the full radical and the primary decomposition of ideals. See [BW93, chapter 8], for the relevant definitions and theory. The implementation of the algorithms presented here in Magma was based on the algorithms presented in that chapter. Currently these algorithms work only for ideals of polynomial rings over fields (Euclidean rings will be supported in the future).

There are also functions for some easier decompositions than the full primary decomposition: radical decompositions, equidimensional decompositions, and triangular decompositions for zero-dimensional ideals. The theory behind these is discussed in the relevant function description.

116.11.1 Radical

`Radical(I): RngMPol -> RngMPol`

Given an ideal I of a polynomial ring P over a field, return the radical of I. The radical R of I is defined to be the set of all polynomials $f \in P$ such that $f^n \in I$ for some $n \geq 1$. The radical R is also an ideal of P, containing I. The function works for an ideal defined over any field over which polynomial factorization is available.

Example: Radical of an Ideal `(ex-964567)`

We compute the radical of an ideal of $\mathbb{Q}[x, y, z, t, u]$ (which is not zero-dimensional).

```
> P<x, y, z, t, u> := PolynomialRing(RationalField(), 5);
> I := ideal<P |
> x + y + z + t + u,
> x*y + y*z + z*t + t*u + u*x,
> x*y*z + y*z*t + z*t*u + t*u*x + u*x*y,
> x*y*z*t + y*z*t*u + z*t*u*x + t*u*x*y + u*x*y*z,
> x*y*z*t*u>;
> R := Radical(I);
> Groebner(R);
> R;
Ideal of Polynomial ring of rank 5 over Rational Field
Order: Lexicographical
Variables: x, y, z, t, u
Homogeneous, Dimension >0, Radical
Groebner basis:
[
```

(continues on next page)

(continued from previous page)

```
    x + y + z + t + u,
    y^2 + y*t - z*u - u^2,
    y*z,
    y*u + z*u + u^2,
    z^2*u + z*u^2,
    z*t,
    t*u
]
> // Check that t*u is in the radical of I by another method:
> IsInRadical(t*u, I);
true
```

116.11.2 Primary Decomposition

`PrimaryDecomposition(I): RngMPol -> [ RngMPol ], [ RngMPol ]`

Given an ideal I of a polynomial ring over a field, return the primary decomposition of I, and also the sequence of associated prime ideals. See *IsPrimary* for the definition of a primary ideal. The primary decomposition of I is returned as two parallel sequences Q and P of ideals, both of length k, having the following properties:

(a) The ideals of Q are primary.

(b) The intersection of the ideals of Q is I.

(c) The ideals of P are the associated primes of Q; i.e., $P[i]$ is the radical of $Q[i]$ (so $P[i]$ is prime) for $1 \leq i \leq k$.

(d) Q is minimal: no ideal of Q contains the intersection of the rest of the ideals of Q and the associated prime ideals in P are distinct.

(e) Q and P are sorted so that P is always unique and Q is unique if I is zero-dimensional. If I is not zero-dimensional, then an embedded component of Q (one whose associated prime contains another associated prime from P) will not be unique in general. Yet Magma will always return the same values for Q and P, given the same I.

The function works for an ideal defined over any field over which polynomial factorization is available (inseparable field extensions are handled by an algorithm due to Allan Steel [Ste05]). NB: if one only wishes to compute the prime components P, then the next function *RadicalDecomposition* should be used, since this may be much more efficient.

`RadicalDecomposition(I): RngMPol -> [ RngMPol ]`

Given an ideal I of a polynomial ring over a field, return the prime decomposition of the radical of I. This is equivalent to applying the function *PrimaryDecomposition* to the radical of I, but may be a much more efficient than using that method. The (prime) radical decomposition of I is returned as a sequence P of ideals of length k having the following properties:

(a) The ideals of P are prime.

(b) The intersection of the ideals of P is the radical of I.

(c) P is minimal: no ideal of P contains the intersection of the rest of the ideals of P.

(e) P is sorted so that P is always unique. Thus Magma will always return the same values for P, given the same I.

The function works for an ideal defined over any field over which polynomial factorization is available.

`ProbableRadicalDecomposition(I): RngMPol -> [ RngMPol ]`

Given an ideal I of a polynomial ring P over a field, return a probabilistic prime decomposition of the radical of I as a sequence of ideals of P. This function is like the function *RadicalDecomposition* except that the ideals returned may not be prime, but the time taken may be much less.

`MinimalDecomposition(S): [ RngMPol ] -> [ RngMPol ]`

Given a set or sequence S of ideals of a polynomial ring over a field, with $I = \cap_{J \in S} J$ (so that S describes a decomposition of I), return a minimal decomposition M of I as a subset of S such that $I = \cap_{J \in M} J$ also (so none of the ideals in the decomposition M are redundant).

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

Change verbose printing for the (Primary/Radical) Decomposition algorithm to be v. Currently the legal values for v are `true`, `false`, 0, 1, or 2.

Example: Primary Decomposition (ex-5b6487)

We compute the primary decomposition of the same ideal of $\mathbb{Q}[x, y, z, t, u]$ (which is not zero-dimensional).

```
> P<x, y, z, t, u> := PolynomialRing(RationalField(), 5);
> I := ideal<P |
> x + y + z + t + u,
> x*y + y*z + z*t + t*u + u*x,
> x*y*z + y*z*t + z*t*u + t*u*x + u*x*y,
> x*y*z*t + y*z*t*u + z*t*u*x + t*u*x*y + u*x*y*z,
> x*y*z*t*u>;
> IsZeroDimensional(I);
false
> Q, P := PrimaryDecomposition(I);
```

We next print out the primary components Q and associated primes P.

```
> Q;
[
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
    Variables: x, y, z, t, u
    Homogeneous, Dimension 1, Non-radical, Primary, Non-prime
    Groebner basis:
    [
        x + 1/2*z + 1/2*u,
        y + 1/2*z + 1/2*u,
        z^2 + 2*z*u + u^2,
        t
    ],
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
    Variables: x, y, z, t, u
    Homogeneous, Dimension 1, Non-radical, Primary, Non-prime
    Groebner basis:
    [
        x + 2*z + t,
        y - z,
        z^2,
        u
    ],
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
```

(continues on next page)

(continued from previous page)

```
Variables: x, y, z, t, u
Homogeneous, Dimension 1, Non-radical, Primary, Non-prime
Groebner basis:
[
    x + z + 2*u,
    y,
    t - u,
    u^2
],
Ideal of Polynomial ring of rank 5 over Rational Field
Order: Lexicographical
Variables: x, y, z, t, u
Homogeneous, Dimension 1, Non-radical, Primary, Non-prime
Groebner basis:
[
    x - u,
    y + t + 2*u,
    z,
    u^2
],
Ideal of Polynomial ring of rank 5 over Rational Field
Order: Lexicographical
Variables: x, y, z, t, u
Homogeneous, Dimension 1, Non-radical, Primary, Non-prime
Groebner basis:
[
    x,
    y + 2*t + u,
    z - t,
    t^2
],
Ideal of Polynomial ring of rank 5 over Rational Field
Order: Lexicographical
Variables: x, y, z, t, u
Homogeneous, Dimension 0, Non-radical, Primary, Non-prime
Size of variety over algebraically closed field: 1
Groebner basis:
[
    x + y + z + t + u,
    y^2 + y*t + 2*y*u - z*t + z*u + u^2,
    y*z^2 - y*z*t + y*t*u - y*u^2 + z^2*t - z^2*u + z*t*u - 2*z*u^2 + t^2*u
        + t*u^2 - u^3,
    y*z*t^2 - 2*y*z*u^2 + 3*y*t*u^2 - 2*y*u^3 + z^3*t - z^3*u - z^2*t^2 +
        4*z^2*t*u - 4*z^2*u^2 + z*t^2*u + 2*z*t*u^2 - 5*z*u^3 + 3*t^2*u^2 +
        2*t*u^3 - 2*u^4,
    y*z*t*u + y*z*u^2 - y*t^2*u - 4*y*t*u^2 + 3*y*u^3 - z^3*t + z^3*u +
        z^2*t^2 - 3*z^2*t*u + 4*z^2*u^2 - 2*z*t*u^2 + 6*z*u^3 - t^3*u -
        5*t^2*u^2 - 3*t*u^3 + 3*u^4,
    y*z*u^3 - 5/2*y*t*u^3 + 3/2*y*u^4 + 1/4*z^3*t^2 + 1/2*z^3*u^2 -
        3/4*z^2*t^3 + 5/4*z^2*t^2*u - 1/4*z^2*t*u^2 + 9/4*z^2*u^3 -
        9/4*z*t^3*u + 1/4*z*t^2*u^2 - 3/4*z*t*u^3 + 13/4*z*u^4 - t^3*u^2 -
        5/2*t^2*u^3 - 7/4*t*u^4 + 3/2*u^5,
    y*t^3*u - 17/4*y*t*u^3 + 13/4*y*u^4 + 1/8*z^3*t^2 + 5/4*z^3*u^2 -
        19/8*z^2*t^3 + 13/8*z^2*t^2*u - 5/8*z^2*t*u^2 + 33/8*z^2*u^3 -
        33/8*z*t^3*u - 7/8*z*t^2*u^2 - 31/8*z*t*u^3 + 49/8*z*u^4 + t^4*u +
        1/2*t^3*u^2 - 15/4*t^2*u^3 - 31/8*t*u^4 + 13/4*u^5,
```

(continues on next page)

(continued from previous page)

```
        y*t^2*u^2 - 3/4*y*t*u^3 - 1/4*y*u^4 + 3/8*z^3*t^2 - 1/4*z^3*u^2 -
            1/8*z^2*t^3 + 7/8*z^2*t^2*u + 1/8*z^2*t*u^2 - 5/8*z^2*u^3 -
            3/8*z*t^3*u + 11/8*z*t^2*u^2 - 5/8*z*t*u^3 - 5/8*z*u^4 + 1/2*t^3*u^2
            + 3/4*t^2*u^3 - 5/8*t*u^4 - 1/4*u^5,
        y*t*u^4 - 2/3*z^2*t^4 + 13/15*z^2*t^2*u^2 - 1/5*z^2*t*u^3 -
            31/15*z*t^4*u + 3/5*z*t^3*u^2 - 2/5*z*t^2*u^3 + 23/15*z*t*u^4 -
            3/5*t^4*u^2 + 2/15*t^3*u^3 - 1/3*t^2*u^4 + t*u^5,
        y*u^5 - 1/2*z^2*t^4 - 1/2*z^2*t^2*u^2 + 1/2*z^2*t*u^3 + 1/2*z^2*u^4 -
            3/2*z*t^4*u - 3*z*t^3*u^2 + 5/2*z*t*u^4 + 3/2*z*u^5 - 1/2*t^4*u^2 -
            2*t^3*u^3 - 2*t^2*u^4 + 1/2*t*u^5,
        z^7,
        z^4*t - z^4*u - z^3*t^2 - 3*z^3*u^2 + 2*z^2*t^3 + 2*z^2*t^2*u -
            9*z^2*t*u^2 - 3*z^2*u^3 + 7*z*t^3*u + 2*z*t^2*u^2 - z*u^4 +
            2*t^3*u^2 - t^2*u^3 + t*u^4,
        z^4*u^2 + 7/3*z^2*t^4 - 40/3*z^2*t^2*u^2 + 8*z^2*t*u^3 - 3*z^2*u^4 +
            22/3*z*t^4*u - 20*z*t^3*u^2 + 2*z*t^2*u^3 + 31/3*z*t*u^4 - 2*z*u^5 +
            t^4*u^2 - 41/3*t^3*u^3 - 10/3*t^2*u^4 + 2*t*u^5,
        z^3*t^3 + 1/3*z^2*t^4 + 2/3*z^2*t^2*u^2 + z^2*t*u^3 + 1/3*z*t^4*u -
            2*z*t^3*u^2 - z*t^2*u^3 + 1/3*z*t*u^4 - 2/3*t^3*u^3 - 1/3*t^2*u^4,
        z^3*t*u - z^2*t^3 + 3*z^2*t*u^2 - 3*z*t^3*u + z*t*u^3 - t^3*u^2,
        z^3*u^3 - 1/3*z^2*t^4 + 7/3*z^2*t^2*u^2 - 2*z^2*t*u^3 + 2*z^2*u^4 -
            4/3*z*t^4*u + 7*z*t^3*u^2 - 2*z*t^2*u^3 - 13/3*z*t*u^4 + z*u^5 +
            14/3*t^3*u^3 + 4/3*t^2*u^4 - t*u^5,
        z^2*t^5 - 3*z*t*u^5 + 17/2*t^5*u^2 + 33/2*t^4*u^3 + 9*t^3*u^4 +
            15/2*t^2*u^5,
        z^2*t^3*u - z^2*t^2*u^2 + z*t^3*u^2,
        z^2*t^2*u^3 - 4/5*z*t*u^5 + 16/5*t^5*u^2 + 59/10*t^4*u^3 -
            11/10*t^3*u^4,
        z^2*t*u^4 - 4/5*z*t*u^5 + 47/10*t^5*u^2 + 42/5*t^4*u^3 - 31/10*t^3*u^4 -
            1/2*t^2*u^5,
        z^2*u^5 + 6*z*t*u^5 - 2*t^5*u^2 - 4*t^4*u^3 - 4*t^3*u^4 - 7*t^2*u^5,
        z*t^5*u + z*t*u^5 - 5/2*t^5*u^2 - 11/2*t^4*u^3 - 3*t^3*u^4 -
            5/2*t^2*u^5,
        z*t^4*u^2 + 2/5*z*t*u^5 - 11/10*t^5*u^2 - 17/10*t^4*u^3 - 1/5*t^3*u^4 -
            1/2*t^2*u^5,
        z*t^3*u^3 + 1/5*z*t*u^5 - 3/10*t^5*u^2 - 3/5*t^4*u^3 - 1/10*t^3*u^4 -
            1/2*t^2*u^5,
        z*t^2*u^4 + 2/5*z*t*u^5 - 8/5*t^5*u^2 - 16/5*t^4*u^3 - 1/5*t^3*u^4,
        t^6,
        t^5*u^3,
        t^4*u^4,
        t^3*u^5,
        u^6
    ]
]
> P;
[
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
    Variables: x, y, z, t, u
    Homogeneous, Dimension 1, Radical, Prime
    Groebner basis:
    [
        x,
        y,
        z + u,
```

(continues on next page)

(continued from previous page)

```
        t
    ],
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
    Variables: x, y, z, t, u
    Homogeneous, Dimension 1, Radical, Prime
    Groebner basis:
    [
        x + t,
        y,
        z,
        u
    ],
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
    Variables: x, y, z, t, u
    Homogeneous, Dimension >0, Radical, Prime
    Groebner basis:
    [
        x + z,
        y,
        t,
        u
    ],
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
    Variables: x, y, z, t, u
    Homogeneous, Dimension 1, Radical, Prime
    Groebner basis:
    [
        x,
        y + t,
        z,
        u
    ],
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
    Variables: x, y, z, t, u
    Homogeneous, Dimension 1, Radical, Prime
    Groebner basis:
    [
        x,
        y + u,
        z,
        t
    ],
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
    Variables: x, y, z, t, u
    Homogeneous, Dimension 0, Radical, Prime
    Size of variety over algebraically closed field: 1
    Groebner basis:
    [
        x,
        y,
        z,
```

(continues on next page)

(continued from previous page)

```
        t,
        u
    ]
]
```

Notice that $P[6]$ contains other ideals of P so $Q[6]$ is an embedded primary component of I. Thus the first 5 ideals of Q would the same be in any primary decomposition of I, while $Q[6]$ could be different in another primary decomposition of I. Finally, notice that the prime decomposition of the radical of I is the same as P except for the removal of $P[6]$ to satisfy the uniqueness condition. The structure of the variety of I can be easily understood by examining the prime decomposition of the radical.

```
> RP := RadicalDecomposition(I);
> #RP;
5
> Set(RP) eq { P[i]: i in [1 .. 5] };
true
```

116.11.3 Triangular Decomposition

Let T be a zero-dimensional ideal of the polynomial ring $K[x_1, \ldots, x_n]$, where K is a field. T is called *triangular* if its Gröbner basis G has length n and the initial term of the i-th polynomial of G is of the form $x_i^{e_i}$ for each i. Any *radical* zero-dimensional ideal has a decomposition as an intersection of triangular ideals. The algorithm in Magma for primary decomposition now (since V2.4) first computes a triangular decomposition and then decomposes each triangular component to primary ideals since the computation of a triangular decomposition is usually fast. See [Laz92] for further discussion of triangular ideals.

`TriangularDecomposition(I): RngMPol -> [ RngMPol ], BoolElt`

Given a zero-dimensional ideal I of a polynomial ring over a field with *lexicographical* order, return a triangular decomposition of I as a sequence Q of ideals such that the intersection of the ideals of Q equals I and for each ideal J of Q which is radical, J is triangular (see above for the definition of a triangular ideal). A second return value indicates whether I is proven to be radical. If I is radical, all entries of Q are triangular. Computing a triangular decomposition will often be faster than computing the full primary decomposition and may yield sufficient information for a specific problem. The algorithm implemented is that given in [Laz92].

Example: Triangular Decomposition of Cyclic-5 `(ex-765e71)`

We compute the triangular decomposition of the (radical) Cyclic-5 roots ideal and compare it with the full primary decomposition of the same ideal.

```
> R<x, y, z, t, u> := PolynomialRing(RationalField(), 5);
> I := ideal<R |
>     x + y + z + t + u,
>     x*y + y*z + z*t + t*u + u*x,
>     x*y*z + y*z*t + z*t*u + t*u*x + u*x*y,
>     x*y*z*t + y*z*t*u + z*t*u*x + t*u*x*y + u*x*y*z,
>     x*y*z*t*u - 1>;
> IsRadical(I);
true
> time T := TriangularDecomposition(I);
Time: 0.000
> time Q, P := PrimaryDecomposition(I);
Time: 0.010
> #T;
9
```

(continues on next page)

(continued from previous page)

```
> #Q;
20
```

So we notice that although I decomposes into 9 triangular ideals, some of these ideals must decompose further since the primary decomposition consists of 20 prime ideals. We examine the first entry of T. Notice that it is at least triangular (it has 5 polynomials and for each variable there is a polynomial whose leading monomial is a power of that variable).

```
> T[1];
Ideal of Polynomial ring of rank 5 over Rational Field
Order: Lexicographical
Variables: x, y, z, t, u
Inhomogeneous, Dimension 0
Groebner basis:
[
    x - 6/5*t^5 - 4*t^4 - 3*t^3 - 3*t^2 - 3*t - 9/5,
    y - 2/5*t^5 - 2*t^4 - 3*t^3 - 2*t^2 - 2*t - 8/5,
    z + 8/5*t^5 + 6*t^4 + 6*t^3 + 5*t^2 + 6*t + 22/5,
    t^6 + 4*t^5 + 5*t^4 + 5*t^3 + 5*t^2 + 4*t + 1,
    u - 1
]
> IsPrimary(T[1]);
false
> D := PrimaryDecomposition(T[1]);
> #D;
2
> D;
[
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
    Variables: x, y, z, t, u
    Inhomogeneous, Dimension 0, Radical, Prime
    Size of variety over algebraically closed field: 2
    Groebner basis:
    [
        x - 1,
        y - 1,
        z + t + 3,
        t^2 + 3*t + 1,
        u - 1
    ],
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
    Variables: x, y, z, t, u
    Inhomogeneous, Dimension 0, Radical, Prime
    Size of variety over algebraically closed field: 4
    Groebner basis:
    [
        x + t^3 + t^2 + t + 1,
        y - t^3,
        z - t^2,
        t^4 + t^3 + t^2 + t + 1,
        u - 1
    ]
]
```

116.11.4 Equidimensional Decomposition

`EquidimensionalPart(I): RngMPol -> RngMPol`

`EquidimensionalDecomposition(I): RngMPol -> [ RngMPol ]`

`FineEquidimensionalDecomposition(I): RngMPol -> SeqEnum`

Let I be an ideal of a polynomial ring P over a field. Currently for the two decomposition functions, it is *assumed* that I has no embedded associated primes (e.g., when I is radical). In this case, it can be much faster to compute an equidimensional decomposition rather than a full primary or radical one. The equidimensional decomposition is the set of ideals which are the intersections of all primary components of I associated to primes of the same dimension. This decomposition (often trivial) is useful for certain constructions involving the Jacobian ideal.

The first function just computes the highest-dimensional decomposition component. The second performs the straight decomposition. The third gives a slightly finer decomposition for the convenience of some applications. In it, each equidimensional component is possibly further split so that, for each final equidimensional factor there is a single set of variables which constitute a maximally independent set of every primary component of the factor (*cf* *Dimension*). A sequence of pairs consisting of each factor and the indices of its set of variables is returned.

The algorithm from [GP02] is used in the general case. When I is homogeneous, a faster, more module-theoretic method is employed for the first two functions. This involves first expressing P/I as a finite module M over a linear Noether Normalisation (described in the next section) S of I. Then if $E(I)$ is the equidimensional part of I, $E(I)/I$ as a submodule of M is equal to the kernel of the natural map of M to its double dual over S, $Hom_S(Hom_S(M,S),S)$. Working with modules over S rather than over P here allows the "reduction to dimension 0". We could directly over P, doing a similar computation but with Hom_S replaced by some Ext^i_P (see [EHV92]).

Example: Equidimensional Decomposition (ex-1ff599)

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> P1 := ideal<P|x*y+y*z+z*x>; // dimension 2 prime
> P2 := ideal<P|x^2+y,y*z+2>; // dimension 1 prime
> P3 := ideal<P|x*y-1,y+z>;   // dimension 1 prime
> I := P1 meet P2 meet P3;
> time rd := RadicalDecomposition(I);
Time: 3.720
> time ed := EquidimensionalDecomposition(I);
Time: 0.070
> #ed;
2
> ed[1] eq P1;
true
> ed[2] eq (P2 meet P3);
true
```

116.12 Normalisation and Noether Normalisation

Suppose I is an ideal of $P = K[x_1, \ldots, x_n]$ with K a field, and I has dimension d. A Noether normalisation of I is given by a set of d polynomials $f_1, \ldots, f_d$ of P, algebraically independent over K, for which $K[f_1, \ldots, f_d] \cap I = 0$ and $K[f_1, \ldots, f_d] \to P/I$ is an integral extension. These always exist and if K is an infinite field, the f_i can be chosen to be linear expressions in the x_i. If I is radical, then the normalisation of I here will refer to the integral closure of the affine ring P/I in its total ring of fractions. If $I = \bigcap P_i$ with P_i prime, then the normalisation is equal to the finite direct product of the normalisations of the P_i as affine rings. It will be specified by a list of pairs (I_i, ϕ_i) where I_i is a prime ideal with generic ring G_i, a multivariate polynomial ring over K, and ϕ_i a homomorphism from P to G_i. The pairs represent the normalisation of each P_i and the inclusion $P/I \to \prod G_i/I_i$ induced by the ϕ_i makes the RHS the integral closure of P/I.

116.12.1 Noether Normalisation

`NoetherNormalisation(I): RngMPol -> [RngMPolElt], Map, Map`

`NoetherNormalization(I): RngMPol -> [RngMPolElt], Map, Map`

This function attempts to compute a Noether Normalisation for I, as described above, using *linear* combinations of the variables. The function is guaranteed to work if K has characteristic zero but may fail in unlucky cases in small characteristic.

The algorithm followed is basically that given in [GP02] but with a simpler test for homogeneous ideals I, which gives a speed-up in that case. Also, subsets of the full sets of variables are considered before more general linear combinations.

The return values are

1) the sequence $[f_1, \ldots, f_d]$.
2) h, an automorphism $P \to P$ given by a linear change of variables which maps the f_i to the last d variables of P. Thus $x_{n-d+1}, \ldots, x_n$ are a corresponding Noether normalising set of polynomials for $h(I)$.
3) the inverse of h.

Example: Noether Normalisation (ex-998b7c)

```
> P<x,y> := PolynomialRing(RationalField(),2);
> I := ideal<P | x*y+x+2>;
> fs,h,hinv := NoetherNormalisation(I);
> fs;
[
    x + y
]
> J := ideal<P | [h(b) : b in Basis(I)]>; J;
Ideal of Polynomial ring of rank 2 over Rational Field
Order: Lexicographical
Variables: x, y
Basis:
[
    -x^2 + x*y + x + 2
]
> // clearly x is integral over the last variable y in P/J
```

116.12.2 Normalisation

`Normalisation(I): RngMPol -> List`

`Normalization(I): RngMPol -> List`

```
UseFF     : BoolElt                    Default: true
FFMin     : BoolElt                    Default: true
FFMinBound: RngIntElt                  Default: 50
UseMax    : BoolElt                    Default: false
```

This function computes the normalisation of the ideal I and returns the result as a list of pairs as described above. The ideal I must be radical - this is not checked in the function. Also the base field K must be perfect.

There are several options. The general algorithm used is that of De Jong as described in [GP02]. However, if the generic polynomial ring P of I has rank 2 then Magma's powerful function field machinery can be applied to give a generally much faster algorithm. This is the default behaviour but can be bypassed by setting the parameter `UseFF` to `false`. For each curve component, this method forms the associated algebraic function field, computes the finite maximal order (the integral closure over a univariate polynomial subring; see Section Construction of Orders of Algebraic

Function Fields), and converts a basis of the order and its multiplication table into an affine algebra presentation.

When the function field machinery is used, a correct result can be obtained extremely quickly, but the generic spaces of the solution ideals can be of quite high dimension. The default behaviour, controlled by the parameter `FFMin`, is to use a filtration by Riemann-Roch spaces to try to find a roughly minimal number of generators of the algebras G_i/I_i and return the corresponding ideal in the smaller number of variables as a more optimal presentation of the solution. This takes more time but is still usually faster than the general algorithm and tends to produce much nicer results. In some cases, the minimised solution is the same as the basic one but takes longer to generate. The minimising stage can be cut out by setting `FFMin` to `false`.

The Riemann-Roch filtration used by the minimising stage can become very expensive for curves of large degree or genus. To avoid this, the stage is automatically skipped (falling back to the basic, non-minimal presentation) when an estimate of its cost exceeds the parameter `FFMinBound`. The estimate is the degree of the largest Riemann-Roch space that would be computed, and is cheap to evaluate. Setting `FFMinBound` to a larger value forces the minimisation to be attempted on harder examples, while a value of 0 always skips it (equivalent to `FFMin := false`).

The general algorithm can avoid doing some work if it is known that certain conditions on I hold. One standard condition is that $I \subseteq M =< x_1, \ldots, x_n >$ and that P/I is locally normal away from M. This holds, for example, if the affine variety defined by I in K^n is non-singular except at the origin. If this is known, then parameter `UseMax` can be set to `true` which will usually speed up the general algorithm (it has no effect if the function field method is used). However, if P/I is locally non-normal at other primes then this will produce an incorrect result.

Example: Normalisation of Plane Curves (ex-7977fa)

```
> P<x,y> := PolynomialRing(RationalField(),2);
> // we begin with a very simple example (prime ideal)
> I := Ideal((x - y^2)^2 - x*y^3);
> time Js := Normalisation(I); // function field method
Time: 0.010
> #Js;
1
> N := Js[1][1];
> N<[a]> := N;
> N;
Ideal of Polynomial ring of rank 2 over Rational Field
Order: Lexicographical
Variables: a[1], a[2]
Basis:
[
    -a[1]*a[2] + a[2]^2 - 2*a[2] + 1
]
> // Now try the basic function field method
> time Js := Normalisation(I: FFMin:=false);
Time: 0.010
> //get the same result here either way
> N := Js[1][1];
> N<[a]> := N;
> N;
Ideal of Polynomial ring of rank 2 over Rational Field
Order: Lexicographical
Variables: a[1], a[2]
Basis:
[
    -a[1]*a[2] + a[2]^2 - 2*a[2] + 1
]
```

(continues on next page)

(continued from previous page)

```
> time Js := Normalisation(I:UseFF:=false); // try the general method
Time: 0.120
> J := Js[1][1];
> Groebner(J);
> J;
Ideal of Polynomial ring of rank 4 over Rational Field
Lexicographical Order
Variables: $.1, $.2, $.3, $.4
Groebner basis:
[
    $.1^2 + 2*$.1 + $.2 - 1,
    $.1*$.2 - 2*$.1 + 2*$.2 - $.3 + $.4 - 4,
    $.1*$.3 + $.3*$.4 + 2*$.3 - $.4^2,
    $.1*$.4 - $.2 + 2,
    $.2^2 - 4*$.2 - $.3*$.4 - 2*$.3 + $.4^2 + 4,
    $.2*$.3 + $.3*$.4^2 + 2*$.3*$.4 - 2*$.3 - $.4^3,
    $.2*$.4 + $.3 - 2*$.4,
    $.3^2 - $.3*$.4^3 - 2*$.3*$.4^2 + $.4^4
]
> // try the general method with UseMax (which applies here)
> time Js := Normalisation(I:UseFF:=false,UseMax:=true);
Time: 0.040
> J := Js[1][1];
> Groebner(J);
> J;
Ideal of Polynomial ring of rank 3 over Rational Field
Lexicographical Order
Variables: $.1, $.2, $.3
Groebner basis:
[
    $.1^2 - 4*$.1 - $.2*$.3 - 2*$.2 + $.3^2 + 4,
    $.1*$.2 + $.2*$.3^2 + 2*$.2*$.3 - 2*$.2 - $.3^3,
    $.1*$.3 + $.2 - 2*$.3,
    $.2^2 - $.2*$.3^3 - 2*$.2*$.3^2 + $.3^4
]
> // now try a harder case - a singular affine form of modular curve X1(11)
> I := ideal<P | (x-y)*x*(y+x^2)^3-y^3*(x^3+x*y-y^2)>;
> time Js := Normalisation(I: FFMin := false);
Time: 0.110
> #Js;
1
> J := Js[1][1];
> Groebner(J);
> J;
Ideal of Polynomial ring of rank 5 over Rational Field
Lexicographical Order
Variables: $.1, $.2, $.3, $.4, $.5
Groebner basis:
[
    $.1*$.3 - $.1 - 6*$.3 + $.4*$.5^2 - 4*$.4*$.5 + 6*$.4 - $.5^5 + $.5^4 +
        11*$.5^3 - 16*$.5^2 + 2*$.5 + 6,
    $.1*$.4 + 2*$.3 - $.4*$.5^2 + 2*$.4*$.5 - 2*$.4 + $.5^4 - 4*$.5^3 + 4*$.5^2
        - 2,
    $.1*$.5 - 2*$.3 + $.4 + $.5^3 - 2*$.5^2 + $.5 + 1,
    $.2 - $.3 + $.5^3 - $.5^2,
    $.3^2 + 3*$.3 - 2*$.4*$.5^2 + 4*$.4*$.5 - 4*$.4 - $.5^6 + 2*$.5^5 + $.5^4 -
```

(continues on next page)

(continued from previous page)

```
        10*$.5^3 + 10*$.5^2 - 4,
    $.3*$.4 - $.3 - $.4*$.5^3 + $.4*$.5^2 - $.4*$.5 + $.4 - $.5^4 + 2*$.5^3 -
        2*$.5^2 + 1,
    $.3*$.5 + $.3 - $.4 - $.5^4 + 2*$.5^2 - $.5 - 1,
    $.4^2 - 2*$.4*$.5^2 + $.4*$.5 + $.4 - $.5^5
]
> time Js := Normalisation(I);
Time: 1.110
> J := Js[1][1];
> Groebner(J);
> J;
Ideal of Polynomial ring of rank 2 over Rational Field
Lexicographical Order
Variables: $.1, $.2
Groebner basis:
[
    $.1^2*$.2 + 2*$.1*$.2 + $.1 - $.2^2 + 2*$.2 + 1
]
> // Minimised result is a cubic equation in K[x,y] - as good as we could get!
> // This example takes MUCH longer with the general method - even setting
> // UseMax := true.
```

116.13 Hilbert Series and Hilbert Polynomial

Let I be a *homogeneous* ideal of the graded polynomial ring $P = K[x_1, \ldots, x_n]$, where K is a field. Then the quotient ring P/I is a *graded* vector space in the following way: P/I is the direct sum of the vector spaces V_d for $d = 0, 1, \ldots$ where V_d is the K-vector space consisting of all homogeneous polynomials in P/I (i.e., reduced residues of polynomials of P with respect to I) of weighted degree d. The *Hilbert Series* of the graded vector space P/I is the generating function

$$H_{P/I}(t) = \sum_{d=0}^{\infty} \dim(V_d) t^d.$$

The Hilbert series can be written as a rational function in the variable t. If the weights on the variables of P are all 1, then there also exists the Hilbert polynomial $F_{P/I}(d)$ corresponding to the Hilbert series $H_{P/I}(t)$ which is a univariate polynomial in $\mathbb{Q}[d]$ such that $F_{P/I}(i)$ is equal to the coefficient of t^i in the Hilbert series for all $i \geq k$ for some fixed k.

`HilbertSeries(I): RngMPol -> FldFunUElt`

Given an homogeneous ideal I of a polynomial ring P over a field, return the Hilbert series $H_{P/I}(t)$ of the quotient ring P/I as an element of the univariate function field $\mathbb{Z}(t)$ over the ring of integers. The algorithm implemented is that given in [BS92]. Note that this is equivalent to *HilbertSeries*(QuotientModule(I)), while if one wishes the Hilbert series of I considered as a P-module, one should call *HilbertSeries*(Submodule(I)).

`HilbertSeries(I, p): RngMPol, RngIntElt -> RngSerLaurElt`

Given an homogeneous ideal I of a polynomial ring P over a field, return the Hilbert series $H_{P/I}(t)$ of the quotient ring P/I as a power series to precision p.

`HilbertDenominator(I): RngMPol -> RngUPol`

Given an homogeneous ideal I of a polynomial ring P over a field, return the unreduced Hilbert denominator D of P/I (as a univariate polynomial over the ring of integers). The denominator D equals $\prod_{i=1}^{n}(1 - t^{w_i})$, where n is the rank of P and w_i is the weight of the i-th variable (1 by default).

`HilbertNumerator(I): RngMPol -> RngUPol`

Given an homogeneous ideal I of a polynomial ring P over a field, return the unreduced Hilbert

numerator N of P/I (as a univariate polynomial over the ring of integers). The numerator N equals $D \times \times H_{P/I}(t)$, where D is the unreduced Hilbert denominator above. Computing with the unreduced numerator is often more convenient.

`HilbertPolynomial(I): RngMPol -> RngUPolElt, RngIntElt`

Given an homogeneous ideal I of a polynomial ring P over a field with weight 1 for each variable, return the Hilbert polynomial $H(d)$ of the quotient ring P/I as an element of the univariate polynomial ring $Q[d]$, together with the index of regularity of P/I (the minimal integer $k \geq 0$ such that $H(d)$ agrees with the Hilbert function of P/I at d for all $d \geq k$).

Example: Hilbert Series of a Matrix Square Ideal `(ex-d1966b)`

We compute the Hilbert series and Hilbert polynomial for an ideal corresponding to the square of a matrix (see [BS92]).

```
> MatSquare := function(n)
>     P := PolynomialRing(RationalField(), n * n, "grevlex");
>     AssignNames(
>         ~P,
>         ["x" cat IntegerToString(i) cat IntegerToString(j): i, j in [1..n]]
>     );
>     M := MatrixRing(P, n);
>     X := M ! [P.((i - 1) * n + j): i, j in [1 .. n]];
>     Y := X^2;
>     return ideal<P | [Y[i][j]: i, j in [1 .. n]]>;
> end function;
> I := MatSquare(4);
> I;
Ideal of Polynomial ring of rank 16 over Rational Field
Order: Graded Reverse Lexicographical
Variables: x11, x12, x13, x14, x21, x22, x23, x24, x31, x32,
    x33, x34, x41, x42, x43, x44
Homogeneous
Basis:
[
    x11^2 + x12*x21 + x13*x31 + x14*x41,
    x11*x12 + x12*x22 + x13*x32 + x14*x42,
    x11*x13 + x12*x23 + x13*x33 + x14*x43,
    x11*x14 + x12*x24 + x13*x34 + x14*x44,
    x11*x21 + x21*x22 + x23*x31 + x24*x41,
    x12*x21 + x22^2 + x23*x32 + x24*x42,
    x13*x21 + x22*x23 + x23*x33 + x24*x43,
    x14*x21 + x22*x24 + x23*x34 + x24*x44,
    x11*x31 + x21*x32 + x31*x33 + x34*x41,
    x12*x31 + x22*x32 + x32*x33 + x34*x42,
    x13*x31 + x23*x32 + x33^2 + x34*x43,
    x14*x31 + x24*x32 + x33*x34 + x34*x44,
    x11*x41 + x21*x42 + x31*x43 + x41*x44,
    x12*x41 + x22*x42 + x32*x43 + x42*x44,
    x13*x41 + x23*x42 + x33*x43 + x43*x44,
    x14*x41 + x24*x42 + x34*x43 + x44^2
]
> S<t> := HilbertSeries(I);
> S;
(t^12 - 7*t^11 + 20*t^10 - 28*t^9 + 14*t^8 + 15*t^7 - 20*t^6 +
    19*t^5 - 22*t^4 + 7*t^3 + 20*t^2 + 8*t + 1)/(t^8 - 8*t^7 +
    28*t^6 - 56*t^5 + 70*t^4 - 56*t^3 + 28*t^2 - 8*t + 1)
> H<d>, k := HilbertPolynomial(I);
> H, k;
```

(continues on next page)

(continued from previous page)

```
1/180*d^7 + 7/90*d^6 + 293/360*d^5 + 61/36*d^4 + 1553/360*d^3 +
    851/180*d^2 + 101/30*d + 1
5
> // Check that evaluations of H for d >= 5 match coefficients of S:
> L<u> := LaurentSeriesRing(IntegerRing());
> L;
Laurent Series Algebra over Integer Ring
> L ! S;
1 + 16*u + 120*u^2 + 575*u^3 + 2044*u^4 + 5927*u^5 + 14832*u^6 +
    33209*u^7 + 68189*u^8 + 130642*u^9 + 236488*u^10 + 408288*u^11 +
    677143*u^12 + 1084929*u^13 + 1686896*u^14 + 2554659*u^15 +
    3779609*u^16 + 5476772*u^17 + 7789144*u^18 + 10892530*u^19 +
    O(u^20)
> Evaluate(H, 5);
5927
> Evaluate(H, 6);
14832
> Evaluate(H, 19);
10892530
```

116.14 Syzygies

The main functions to compute syzygies work with or return modules. See Chapter *Modules over Multivariate Rings* for these. This section contains a variant that returns a basis of syzygies of a polynomial sequence as rows of a matrix.

SyzygyMatrix(Q): [RngMPolElt] -> ModMatRngElt

Given a sequence Q of polynomials from a multivariate polynomial ring P, return the module of syzygies of Q as a matrix S. This an r by k matrix, where k is the length of Q, whose rows span the space of all vectors v such that the sum of $v[i] * Q[i]$ for $i = 1, \ldots k$ is zero. The algorithm used is the standard one, computing a module Gröbner basis with respect to a particular elimination order (see section 2.5 of [GP02], for example). The base ring may be a field or Euclidean ring.

Example: Syzygy Matrix (ex-41f6f0)

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> SyzygyMatrix([x + y, x - y, x*z + y*z]);
[             z              0             -1]
[ 1/2*x - 1/2*y -1/2*x - 1/2*y              0]
```

116.15 Maps between Rings

Magma includes functions for working with maps between multivariate polynomial rings. Let $R = K_1[x_1, \ldots, x_n]$ and $S = K_2[y_1, \ldots, y_m]$ be a polynomial rings over the fields K_1, K_2, and $f : R \to S$ a ring homomorphism.

PolyMapKernel(f): Map -> RngMPol

Return the kernel of the map f as an ideal in the domain R, i.e., the set $\{a \in R | f(a) = 0\}$. This is basically the computation of the relation ideal for the polynomials defining the map and is as described in *RelationIdeal*.

IsInImage(f, p): Map, RngMPolElt -> [BoolElt]

Given a polynomial p in S, return whether p is in the image of the map f. The algorithm is the one described on p. 82 of [AL94].

IsSurjective(f): Map -> [BoolElt]

Return whether the map f is surjective. Uses the function above to check whether each codomain variable lies in the image.

Extension(phi, I): Map, RngMPol -> RngMPol

The extension of the ideal I by ϕ, where ϕ is a homomorphism from the generic of I. That is, the ideal generated by the image of I under ϕ.

Implicitization(phi): Map -> RngMPol

Suppose the polynomial map $\phi : K^n \to K^m$ is a parametrization of a variety V, i.e., V is the image of ϕ in K^m. This function constructs the ideal of S corresponding to V. The map ϕ maps $(z_1, \ldots, z_n) \mapsto (f_1(z_1), \ldots, f_m(z_m))$ where the z_i are the coordinates of K^n. Let $f : S \to R$ be the map of polynomial rings defined by $(y_1, \ldots, y_m) \mapsto (f_1(y_1), \ldots, f_m(y_m))$. Then `Implicitization(f)` is the ideal of S corresponding to V. If V is not a true variety, the function returns the smallest variety containing V (the Zariski closure of V). The algorithm used is given on p. 97 of [CLO96]

Example: Implicitization of a Monomial Map (ex-a72e07)

We demonstrate the use of the function `Implicitization` for the variety defined by $\phi : Q[x, y] \to Q[r, u, v, w]$, $(x, y) \mapsto (x^4, x^3y, xy^3, y^4)$. This example is taken from [AL94, Ex. 2.5.4].

```
> R<x, y> := PolynomialRing(Rationals(), 2);
> S<r, u, v, w> := PolynomialRing(Rationals(), 4);
> f := hom<S -> R |x^4, x^3*y, x*y^3, y^4>;
> Implicitization(f);
Ideal of Polynomial ring of rank 4 over Rational Field
Lexicographical Order
Variables: r, u, v, w
Basis:
[
    -r^2*v + u^3,
    r*v^2 - u^2*w,
    -u*w^2 + v^3,
    -r*w + u*v
]
```

116.16 Symmetric Polynomials

Magma includes functions for working with symmetric polynomials.

ElementarySymmetricPolynomial(P, k): RngMPol, RngIntElt -> RngMPolElt

Given a polynomial ring P of rank n, and an integer k with $1 \leq k \leq n$, return the k-th elementary symmetric polynomial of P.

IsSymmetric(f): RngMPolElt -> BoolElt, RngMPolElt

IsSymmetric(f, S): RngMPolElt, RngMPol -> BoolElt, RngMPolElt

Given a polynomial f from a polynomial ring P of rank n, return whether f is a symmetric polynomial of P (i.e., is symmetric in all the n variables of P). If the answer is true, a polynomial g from a new polynomial ring of rank n is returned such that $f = g(e_1, \ldots, e_n)$, where e_i is the i-th elementary symmetric polynomial of P. If g is desired to be a member of a particular polynomial ring S of rank n (to obtain predetermined names of variables, for example), then S may also be passed.

Example: Symmetric Polynomial in Elementary Terms (ex-6e4642)

We create a symmetric polynomial from $\mathbb{Q}[a, b, c, d]$ and express it in terms of the elementary symmetric polynomials.

```
> P<a, b, c, d> := PolynomialRing(RationalField(), 4, "grevlex");
> f :=
> a^2*b^2*c*d + a^2*b*c^2*d + a*b^2*c^2*d + a^2*b*c*d^2 + a*b^2*c*d^2 +
>     a*b*c^2*d^2 - a^2*b^2*c - a^2*b*c^2 - a*b^2*c^2 - a^2*b^2*d -
>     3*a^2*b*c*d - 3*a*b^2*c*d - a^2*c^2*d - 3*a*b*c^2*d - b^2*c^2*d -
>     a^2*b*d^2 - a*b^2*d^2 - a^2*c*d^2 - 3*a*b*c*d^2 - b^2*c*d^2 -
>     a*c^2*d^2 - b*c^2*d^2 + a + b + c + d;
> // Check orbit under Sym(4) has size one:
> #(f^Sym(4));
1
> Q<e1, e2, e3, e4> := PolynomialRing(RationalField(), 4);
> l, E := IsSymmetric(f, Q);
> l;
true
> E;
e1 - e2*e3 + e2*e4
```

In the following example, we use a rational function field to define parameters a and b which occur as coefficients of the symmetric polynomial f.

```
> F<a,b> := FunctionField(RationalField(), 2);
> P<x1,x2,x3,x4,x5> := PolynomialRing(F, 5, "grevlex");
> y1 := x1^4 + x1^2*a + x1*b;
> y2 := x2^4 + x2^2*a + x2*b;
> y3 := x3^4 + x3^2*a + x3*b;
> y4 := x4^4 + x4^2*a + x4*b;
> y5 := x5^4 + x5^2*a + x5*b;
> f := y1*y2 + y1*y3 + y1*y4 + y1*y5 + y2*y3 + y2*y4 +
>      y2*y5 + y3*y4 + y3*y5 + y4*y5;
> Q<e1,e2,e3,e4,e5> := PolynomialRing(F,  5);
> l,E := IsSymmetric(f, Q);
> l, E;
true b*e1^3*e2 - 2*a*e1^3*e3 - 4*e1^3*e5 + a*e1^2*e2^2 +
    4*e1^2*e2*e4 + 2*e1^2*e3^2 - b*e1^2*e3 + 2*a*e1^2*e4 -
    4*e1*e2^2*e3 - 3*b*e1*e2^2 + 4*a*e1*e2*e3 + 8*e1*e2*e5 +
    a*b*e1*e2 - 8*e1*e3*e4 - 2*a^2*e1*e3 + b*e1*e4 - 6*a*e1*e5 +
    e2^4 - 2*a*e2^3 - 4*e2^2*e4 + a^2*e2^2 + 4*e2*e3^2 +
    5*b*e2*e3 + 2*a*e2*e4 + b^2*e2 - 3*a*e3^2 - 4*e3*e5 -
    3*a*b*e3 + 6*e4^2 + 2*a^2*e4 - 5*b*e5
```

116.17 Functions for Polynomial Algebra and Module Generators

The following functions work with collections of polynomials which are considered as generators for subalgebras or submodules of a polynomial ring. They have particular use in invariant theory.

MinimalAlgebraGenerators(L): [RngMPol] -> [RngMPol]

MinimalAlgebraGenerators(L): { RngMPol } -> [RngMPol]

Let $R = K[x_1, \ldots, x_n]$ be a polynomial ring of rank n over the field K. Suppose L is a set or sequence of k polynomials $f_1, \ldots, f_k$ in R. Let $A = K[f_1, \ldots, f_k]$ be the subalgebra (*not* ideal) of R generated by L. This function returns a minimal generating set of the algebra A as a (sorted) sequence of elements taken from L.

HomogeneousModuleTest(P, S, F): [RngMPol], [RngMPol], RngMPol -> BoolElt, [RngMPol]

Let $R = K[x_1, \ldots, x_n]$ be a polynomial ring of rank n over the field K. Suppose P is a sequence of k homogeneous polynomials $p_1, \ldots, p_k$ in R and suppose S is a sequence of r homogeneous polynomials $s_1, \ldots, s_r$ in R. Let $A = K[p_1, \ldots, p_k]$ be the subalgebra (*not* ideal) of R generated

by P and let $M = A[s_1, \ldots, s_r]$ be the A-module generated by S over A. Finally, suppose F is an element of R. This function returns whether F is in the module M (considered as a submodule of R). If the result is `true`, the function also returns a sequence $C = [c_1, \ldots, c_r]$ of length r with $c_i \in K[t_1, \ldots, t_k]$ such that $F = \sum_{i=1}^{r} c_i(p_1, \ldots, p_k) \cdot s_i$. (The polynomial ring $K[t_1, \ldots, t_r]$ is constructed separately but automatically with the print names `t1`, `t2`, etc.) The grading of the polynomial ring R is used to determine the (weighted) degrees of all the polynomials in P, S, and the polynomial F. The function works as follows: it first splits F into its homogeneous components, and then, for each homogeneous component of (weighted) degree d, it constructs a basis for the K-space of all polynomials of the module M of degree d and then determines by linear algebra whether the component lies in that space. The function is most often used with an invariant ring: P is the sequence of primary invariants, S is the sequence of secondary invariants, and F is a general invariant which one wishes to express in terms of the module generators S over the algebra generated by P. Also, if one wishes to test only for membership in the algebra $A = K[p_1, \ldots, p_k]$, then the sequence `[R!1]` should be passed for S.

`HomogeneousModuleTest(P, S, L): [ RngMPol ], [ RngMPol ], [ RngMPol ] -> [ BoolElt ], [ [ RngMPol ] ]`

Let $R = K[x_1, \ldots, x_n]$ be a polynomial ring of rank n over the field K. Suppose P is a sequence of k homogeneous polynomials $p_1, \ldots, p_k$ in R and suppose S is a sequence of r homogeneous polynomials $s_1, \ldots, s_r$ in R. Let $A = K[p_1, \ldots, p_k]$ be the subalgebra (*not* ideal) of R generated by P and let $M = A[s_1, \ldots, s_r]$ be the A-module generated by S over A. Finally, suppose L is a sequence of length l of elements of R which are all homogeneous of (weighted) degree d. This function returns parallel sequences B and V with the following properties:

(a) B is sequence of length l of booleans such that for $1 \leq i \leq l$, $B[i]$ is true iff $L[i]$ is in the module M.

(b) V is a sequence of length l consisting of sequences of length r and consisting of polynomials in the polynomial ring $T = K[t_1, \ldots, t_r]$. (The polynomial ring $T = K[t_1, \ldots, t_r]$ is constructed separately but automatically with the print names `t1`, `t2`, etc.) If $B[i]$ is false (so $L[i]$ is not in M), $V[i]$ is a sequence of r zero polynomials. Otherwise $V[i]$ is a sequence of r polynomials $c_{i,1}, \ldots, c_{i,r}$ in T such that $L[i] = \sum_{j=1}^{r} c_{i,j}(p_1, \ldots, p_k) \cdot s_j$.

The grading of the polynomial ring R is used to determine the (weighted) degrees of all the polynomials in P, S, and L.

The function works as follows: it constructs a basis for the K-space of all polynomials of the module M of degree d and then, for each i with $1 \leq i \leq l$, determines by linear algebra whether $L[i]$ lies in the space. Only one echelonization of the space is needed to determine all the values of B and V so it is *much* more efficient to use this function if possible with many polynomials in L of the same homogeneous degree instead of calling the previous function separately for each polynomial since that will need to construct the basis for the homogeneous space and perform an echelonization each time.

Again, this function is most often used with an invariant ring: P is the sequence of primary invariants, S is the sequence of secondary invariants, and L is a sequence of general invariants which one wishes to express in terms of the module generators S over the algebra generated by P. Also, if one wishes to test only for membership in the algebra $A = K[p_1, \ldots, p_k]$, then the sequence $[R!1]$ should be passed for S.

`HomogeneousModuleTestBasis(P, S, L): [ RngMPol ], [ RngMPol ], [ RngMPol ] -> [ BoolElt ], [ [ RngMPol ] ]`

Let $R = K[x_1, \ldots, x_n]$ be a polynomial ring of rank n over the field K. Suppose P is a sequence of k homogeneous polynomials $p_1, \ldots, p_k$ in R and suppose S is a sequence of r homogeneous polynomials $s_1, \ldots, s_r$ in R. Let $A = K[p_1, \ldots, p_k]$ be the subalgebra (*not* ideal) of R generated by P and let $M = A[s_1, \ldots, s_r]$ be the A-module generated by S over A. Finally, suppose L is a sequence of length l of elements of R which are all homogeneous of (weighted) degree d. Let U be the K-subspace of R consisting of all polynomials of the module M of (weighted) degree d and let V be the K-subspace of R generated by the elements of L. This function returns a sequence I of integer indices such that the sequence elements of L corresponding to the indices in I forms

a basis for a K-subspace W of R such that $U + V = U \oplus W$. That is, I selects a subsequence of L which yields an *extension* of any basis of U to a basis of $U + V$. Using this function, one can extend a minimal module generating set in S to include new elements of increasing degree, while ensuring that the module generators are minimalized (i.e., there is no redundancy amongst them). The grading of the polynomial ring R is used to determine the (weighted) degrees of all the polynomials in P, S, and L.

Example: Homogeneous Module Test `(ex-882819)`

We demonstrate simple uses of the function `HomogeneousModuleTest`. See also the example `HomogeneousModuleTest2` in the Invariant Rings chapter which demonstrates the use of the function `HomogeneousModuleTest` in invariant theory.

```
> R<x, y, z> := PolynomialRing(RationalField(), 3);
> P := [x^2 + y^2, z];
> S := [1, x + y + z];
> L := [x^2 + y^2, (x+y+z)^2-z^2-2*x*y, x*y];
> B, V := HomogeneousModuleTest(P, S, L);
> B;
[ true, true, false ]
> V;
[
    [
        t1,
        0
    ],
    [
        t1 - 2*t2^2,
        2*t2
    ],
    [
        0,
        0
    ]
]
> // Thus L[1] is P[1]*S[1] and
> // L[2] is (P[1] - 2*P[2]^2)*S[1] + 2*P[2]*S[2].
> L[1] eq P[1]*S[1];
true
> (P[1] - 2*P[2]^2)*S[1] + 2*P[2]*S[2] eq L[2];
true
> // Determine subsequence of [x^3, y^3, z^3] which forms
> // extension basis of module generated by P and S.
> L := [x^3, y^3, z^3];
> HomogeneousModuleTestBasis(P, S, L);
[ 1, 2 ]
> // Thus x^2 and y^2 could be appended to S to preserve
> // minimality.
```

116.18 Bibliography

- [AL94] William Adams and Philippe Loustaunau. *An introduction to Gröbner bases.* Volume 3 of Graduate studies in mathematics. American Mathematical Society, Providence, R.I., 1994.
- [BS92] David Bayer and Michael Stillman. Computation of Hilbert Functions. *J. Symbolic Comp.*, 14(1):31–50, 1992.
- [BW93] Thomas Becker and Volker Weispfenning. *Gröbner Bases.* Graduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 1993.

- [CLO96] David Cox, John Little, and Donal O'Shea. *Ideals, Varieties and Algorithms*. Undergraduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 2nd edition, 1996.
- [CLO98] David Cox, John Little, and Donal O'Shea. *Using Algebraic Geometry*. Graduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 1998.
- [DL06] Wolfram Decker and Christoph Lossen. *Computing in Algebraic Geometry*. Volume 16 of Algorithms and Computation in Mathematics. Springer, New York–Berlin–Heidelberg, 2006.
- [ES94] Eisenbud and Sturmfels. Finding sparse systems of parameters. *Journal of Pure and Applied Algebra*, 94:143–157, 1994.
- [EHV92] D. Eisenbud, C. Huneke, and W. Vasconcelas. Direct Methods for Primary Decomposition. *Inv. math.*, 110:207–235, 1992.
- [GP02] G.-M. Greuel and G. Pfister. *A Singular Introduction to Commutative Algebra*. Springer-Verlag, Berlin–Heidelberg–New York, 2002. ISBN 3-540-42897-6.
- [Har77] Robin Hartshorne. *Algebraic Geometry, GTM 52*. Springer, ASpringer, 1977.
- [Laz92] Daniel Lazard. Solving Zero-dimensional Algebraic Systems. *J. Symbolic Comp.*, 13(2):117–131, 1992.
- [Ste05] Allan Steel. Conquering Inseparability: Primary Decomposition and Multivariate Factorization over Algebraic Function Fields of Positive Characteristic. *J. Symbolic Comp.*, 40(3):1053–1075, 2005.

Chapter 117

Local Polynomial Rings

117.1 Introduction

This chapter describes local polynomial rings. Let R be the multivariate polynomial ring $K[x_1, \ldots, x_n]$, where K is a field. We denote by

$$K[x_1, \ldots, x_n]_{\langle x_1, \ldots, x_n \rangle}$$

the collection of all rational functions f/g of $x_1, \ldots, x_n$ with $g(p) \neq 0$, where $p = (0, \ldots, 0)$. Such a ring is **local** (has a unique maximal ideal) and we will call it a **local polynomial ring** in Magma. Such a ring is always multivariate and is related to the corresponding multivariate polynomial ring $K[x_1, \ldots, x_n]$ which will we will call **global** (when distinguishing it from the local case). We will also call $K[x_1, \ldots, x_n]_{\langle x_1, \ldots, x_n \rangle}$ the **localization** of $K[x_1, \ldots, x_n]$ (this is always understood to be at the prime ideal generated by $x_1, \ldots, x_n$, corresponding to the origin). Much of the theory for multivariate polynomial rings and their ideals carry over to local polynomial rings, so the reader should first be familiar with multivariate polynomial rings and their ideals (see Chapters Multivariate Polynomial Rings and *Gröbner Bases*). Corresponding to a Gröbner basis of an ideal of a global multivariate ring is a **standard basis** of an ideal of a local polynomial ring. See [CLO98, Chapter 4] or [GP02, Chapter 1] for the basics of the theory and algorithms. The other facilities are currently basic but will be expanded in coming versions. But note that computations with $R-$modules, where R is a local polynomial ring, are fully supported: see Chapter *Modules over Multivariate Rings*.

117.2 Elements and Local Monomial Orders

Elements of a local polynomial ring are multivariate polynomials just like the usual (global) multivariate polynomials, except that the monomials are sorted (again with the greatest first) with respect to a **local monomial order**, which is in general the negation of a standard global monomial order. Thus the monomial 1 is less than all other monomials and the polynomials are like multivariate formal power series (written, for example, as $1 + x + x^2y + y^4$). But most arithmetic-like operations allowed for global polynomials also carry over automatically for elements of a local polynomials so we will not list them in detail in this chapter (see Chapter Multivariate Polynomial Rings). Note that in the strict mathematical definition of $R = K[x_1, \ldots, x_n]_{\langle x_1, \ldots, x_n \rangle}$, elements of R may have non-trivial denominators, but this is currently not supported in Magma: the elements in Magma must always be strict polynomials. The main purpose of supporting such rings is for standard bases of ideals (see below), and this restriction does not matter there, since units are automatically removed from the elements of a standard basis. We now describe the current local monomial orders available in Magma. First the reader should see Section *Representation and Monomial Orders* for the fundamental points about (global) monomial orders for multivariate polynomial rings. The fundamental difference in the local case is that for a local polynomial ring R of rank n, the monomial order is the negation of a global monomial order. More precisely, let M be the monomials of R. A **local monomial ordering** on M is a total order $<$ on M such that $s \leq 1$ for all $s \in M$, $s \leq t$ implies $su \leq tu$ for all $s, t, u \in M$, and M is a well-ordering (every non-empty subset of M possesses a minimal element w.r.t. $<$). See [CLO98, Sec. 4.3], [DL06, Sec. 9.1], or [GP02,

Sec. 1.2] for more information. We now list each of the monomial orders available in Magma (these will be expanded in future versions). As in the global case, we suppose that s and t are monomials from a ring R of rank n. Any order on the monomials is then fully defined by just specifying exactly when $s < t$ with respect to that order. In the following, the argument(s) are described for an order as a list of expressions; that means that the expressions (without the parentheses) should be appended to any base arguments when any particular intrinsic function is called which expects a monomial order.

117.2.1 Local Lexicographical: llex

Definition: $s < t$ iff there exists $1 \leq i \leq n$ such that all of the j-th exponents of s and t are equal for $i < j \leq n$, but the i-th exponent of s is greater than the i-th exponent of t. The order is specified by the argument (`"llex"`). This order is the negation of the global lexicographical order, but with the reverse order for the variables. Thus the i-th variable is greater than the $(i+1)$-th variable for $1 \leq i < n$ so the first variable is the greatest variable.

117.2.2 Local Graded Lexicographical: lglex

Definition: $s < t$ iff the total degree of s is greater than the total degree of t or the total degree of s is equal to the total degree of t and $s > t$ with respect to the (glocal) lexicographical order. The order is specified by the argument (`"lglex"`). This order is the negation of the global `glex` order.

117.2.3 Local Graded Reverse Lexicographical: lgrevlex

Definition: $s < t$ iff the total degree of s is greater than the total degree of t or the total degree of s is equal to the total degree of t and $s < t$ with respect to the (global) lexicographical order applied to the exponents of s and t in reverse order. The order is specified by the argument (`"grevlex"`). This order is the negation of the global `grevlex` order.

117.3 Local Polynomial Rings and Ideals

117.3.1 Creation of Local Polynomial Rings and Accessing their Monomial Orders

Local polynomial rings are created from a coefficient field, the number of variables, and a monomial order. If no order is specified, the monomial order is taken to be the local lexicographical order.

`LocalPolynomialRing(K, n): Rng, RngIntElt -> RngMPolLoc`

Create a local polynomial ring in $n > 0$ variables over the field K. The *local lexicographical* ordering on the monomials is used for this default construction.

`LocalPolynomialRing(K, n, order): Rng, RngIntElt, MonStgElt, ... -> RngMPolLoc`

`LocalPolynomialAlgebra(K, n, order): Rng, RngIntElt, MonStgElt, ... -> RngMPolLoc`

Create a local polynomial ring in $n > 0$ variables over the ring R with the given order order on the monomials. See the above section on local monomial orders for the valid values for the argument order.

`LocalPolynomialRing(K, n, T): Rng, RngIntElt, Tup -> RngMPolLoc`

Create a local polynomial ring in $n > 0$ variables over the field K with the order given by the tuple T on the monomials. T must be a tuple whose components match the valid arguments for the monomial orders in Section *Elements and Local Monomial Orders*. Such a tuple is also returned by the next function.

`MonomialOrder(R): RngMPolLoc -> Tup`

Given a local polynomial ring R (or an ideal thereof), return a description of the monomial order of R. This is returned as a tuple which matches the relevant arguments listed for each possible order in Section *Elements and Local Monomial Orders*, so may be passed as the third argument to the function *LocalPolynomialRing* above.

`MonomialOrderWeightVectors(R): RngMPol -> [ [ FldRatElt ] ]`

Given a polynomial ring R of rank n (or an ideal thereof), return the weight vectors of the underlying monomial order as a sequence of n sequences of n rationals. See, for example, [CLO98, p. 153] for more information.

`Localization(R): RngMPol -> RngMPolLoc`

`Localization(I): RngMPol -> RngMPolLoc`

Given a (global) multivariate polynomial ring $R = K[x_1, \ldots, x_n]$ (or an ideal I of such an R), return the localization $K[x_1, \ldots, x_n]_{\langle x_1, \ldots, x_n \rangle}$ of R (or the ideal of the localization of R which corresponds to I). The print names for the variables of R are carried over.

Example: Local Polynomial Rings with Different Orders (ex-ab71f5)

We show how one can construct local polynomial rings with different orders. Note the order on the monomials for elements of the rings.

```
> K := RationalField();
> R<x,y,z> := LocalPolynomialRing(K, 3);
> R;
Localization of Polynomial Ring of rank 3 over Rational Field
Order: Local Lexicographical
Variables: x, y, z
> MonomialOrder(R);
<"llex">
> MonomialOrderWeightVectors(R);
[
    [ 0, 0, -1 ],
    [ 0, -1, 0 ],
    [ -1, 0, 0 ]
]
> 1 + x + y + z + x^7 + x^8*y^7 + y^5 + z^10;
1 + x + x^7 + y + y^5 + x^8*y^7 + z + z^10
> R<x,y,z> := LocalPolynomialRing(K, 3, "lgrevlex");
> R;
Localization of Polynomial Ring of rank 3 over Rational Field
Order: Local Graded Reverse Lexicographical
Variables: x, y, z
> MonomialOrder(R);
<"lgrevlex">
> MonomialOrderWeightVectors(R);
[
    [ -1, -1, -1 ],
    [ -1, -1, 0 ],
    [ -1, 0, 0 ]
]
> 1 + x + y + z + x^7 + x^8*y^7 + y^5 + z^10;
1 + z + y + x + y^5 + x^7 + z^10 + x^8*y^7
```

117.3.2 Creation of Ideals and Accessing their Bases

As for global polynomial rings, within the general context of ideals of local polynomial rings, the term "basis" will refer to an *ordered* sequence of polynomials which generate an ideal. (Thus a basis can contain duplicates and zero elements so is not like a basis of a vector space.)

`ideal<R | L>: RngMPolLoc, List -> RngMPolLoc`

Given a local polynomial ring R, return the ideal of R generated by the elements of R specified by the list L. Each term of the list L must be an expression defining an object of one of the following types:

(a) An element of R;

(b) A set or sequence of elements of R;

(c) An ideal of R;

(d) A set or sequence of ideals of R.

`Ideal(B): [ RngMPolLocElt ] -> RngMPolLoc`

`Ideal(B): { RngMPolLocElt } -> RngMPolLoc`

Given a set or sequence B of polynomials from a local polynomial ring R, return the ideal of R generated by the elements of B with the given basis B. This is equivalent to the above `ideal` constructor, but is more convenient when one simply has a set or sequence of polynomials.

`Ideal(f): RngMPolLocElt -> RngMPolLoc`

Given a polynomial f from a local polynomial ring R, return the principal ideal of R generated by f.

`IdealWithFixedBasis(B): [ RngMPolLocElt ] -> RngMPolLoc`

Given a sequence B of polynomials from a local polynomial ring R, return the ideal of R generated by the elements of B with the given fixed basis B. When the function *Coordinates* is called, its result will be with respect to the entries of B instead of the Gröbner basis of I. **WARNING:** this function should *only* be used when it is desired to express polynomials of the ideal in terms of the elements of B, as the computation of the Gröbner basis in this case is *very* expensive, so it should be avoided if these expressions are not wanted.

`Basis(I): RngMPolLoc -> [ RngMPolLocElt ]`

Given an ideal I, return the current basis of I. This will be the standard basis of I if it is computed; otherwise it will be the original basis.

`BasisElement(I, i): RngMPolLoc, RngIntElt -> RngMPolLocElt`

Given an ideal I together with an integer i, return the i-th element of the current basis of I. This the same as `Basis(I)[i]`.

117.4 Standard Bases

Computation in ideals of local polynomial rings is possible because of the construction of **standard bases** of such ideals. These are the counterpart to Gröbner bases for ideals of global polynomial rings. Currently, standard bases may only be computed for ideals defined over fields. Magma computes a standard basis of an ideal using the Mora normal form and standard basis algorithms (with the homogenization technique): see [CLO98, Sec. 4.4] for an overview. In contrast to the global case, for a given fixed monomial ordering a standard basis of an ideal is not unique in general because it can be difficult to get the lower order terms of polynomials in the standard basis into a unique form. But the **leading monomials** of a standard basis are always sorted in Magma and **are unique**.

117.4.1 Construction of Standard Bases

The following functions and procedures allow one to construct standard bases. Note that a standard basis for an ideal will be automatically generated when necessary; the `Groebner` procedure below simply allows control of the algorithms used to compute the standard basis. The verbose flags are shared with those for global Gröbner basis construction (since the standard basis algorithms reduce to those), so see Section *Verbosity* for details on these.

`StandardBasis(I): RngMPolLoc -> RngMPolLocElt`

Given an ideal I, force the standard basis of I to be computed, and then return that.

`StandardBasis(S): [ RngMPolLocElt ] -> [ RngMPolLocElt ]`

`StandardBasis(S): { RngMPolLocElt } -> [ RngMPolLocElt ]`

Given a set or sequence S of polynomials of a local polynomial ring R, return a standard basis of the ideal generated by S as a sorted sequence.

`Coordinates(I, f): RngMPolLoc, RngMPolLocElt -> [ RngMPolLocElt ]`

Given an ideal I of a local polynomial ring R, together with a polynomial f in I, and supposing that I has basis $b_1, \ldots, b_k$, return a sequence $[g_1, \ldots, g_k]$ of elements of R so that $f = g_1 * b_1 + \ldots + g_k * b_k$. If I was created by *IdealWithFixedBasis*(B), then the fixed basis B is used as the basis $b_1, \ldots, b_k$; otherwise the Gröbner basis of I is used as the basis $b_1, \ldots, b_k$. The resulting sequence is not necessarily unique.

`CoordinateMatrix(I): RngMPolLoc -> Matrix`

Given an ideal I such that I has a fixed basis (i.e., such that I was created via the function *IdealWithFixedBasis*), return the coordinate matrix C of I. The i-th row of C gives the coordinates of the i-th element of the Gröebner basis of I w.r.t. the fixed basis of I. The Gröebner basis of I is first computed if it has not been already.

Example: Standard Basis of an Ideal (ex-37d3dd)

We compute the standard basis of the ideal given in [CLO98, p.167].

```
> Q := RationalField();
> R<x,y,z> := LocalPolynomialRing(Q, 3);
> I := Ideal([x^5 - x*y^6 + z^7, x*y + y^3 + z^3, x^2 + y^2 - z^2]);
> I;
Ideal of Localization of Polynomial Ring of rank 3 over Rational Field
Order: Local Lexicographical
Variables: x, y, z
Basis:
[
    x^5 - x*y^6 + z^7,
    x*y + y^3 + z^3,
    x^2 + y^2 - z^2
]
> StandardBasis(I);
[
    x^2 + y^2 - z^2,
    x*y + y^3 + z^3,
    y^3 - x*y^3 - y*z^2 - x*z^3,
    x*z^4 + 3*y^2*z^4 + 4*x*y^4*z^4 + y*z^5 + 5*x*y^3*z^5 - 2*z^6 + 4*y^2*z^6 +
        x*y^2*z^6 + z^7 - 2*x*z^7 + 7*y*z^7 + 4*x*y*z^7 - y^2*z^7 + 3*z^8 +
        4*x*z^8,
    y^2*z^4 + 3*y^5*z^5 - z^6 + 3/2*x*z^6 + 2*y^2*z^6 + y^4*z^6 - x*z^7 +
        7/2*y*z^7 - x*y^2*z^7 - y^3*z^7 + 3/2*z^8 - 3/2*x*z^8 - 2*y*z^8 +
        2*x*y*z^8 + 3/2*y^2*z^8,
    y*z^7 + 1/2*x*z^8 + 23/4*y*z^8 - 9/4*x*y^3*z^8 + 3/2*y^4*z^8,
    z^9
]
```

We note that no elements of the standard basis have factors which are units in R.

```
> [Factorization(f): f in $1];
[
    [
        <x^2 + y^2 - z^2, 1>
    ],
    [
        <x*y + y^3 + z^3, 1>
    ],
    [
        <y^2 - x*y^2 - y*z + x*y*z - x*z^2, 1>,
        <y + z, 1>
    ],
```

(continues on next page)

(continued from previous page)

```
    [
        <z, 4>,
        <x + 3*y^2 + 4*x*y^4 + y*z + 5*x*y^3*z - 2*z^2 + 4*y^2*z^2 + x*y^2*z^2 +
            z^3 - 2*x*z^3 + 7*y*z^3 + 4*x*y*z^3 - y^2*z^3 + 3*z^4 + 4*x*z^4, 1>
    ],
    [
        <z, 4>,
        <y^2 + 3*y^5*z - z^2 + 3/2*x*z^2 + 2*y^2*z^2 + y^4*z^2 - x*z^3 +
            7/2*y*z^3 - x*y^2*z^3 - y^3*z^3 + 3/2*z^4 - 3/2*x*z^4 - 2*y*z^4 +
            2*x*y*z^4 + 3/2*y^2*z^4, 1>
    ],
    [
        <z, 7>,
        <y + 1/2*x*z + 23/4*y*z - 9/4*x*y^3*z + 3/2*y^4*z, 1>
    ],
    [
        <z, 9>
    ]
]
```

Example: Standard Basis Versus Groebner Basis (ex-47561b)

We note that starting from an ideal I of a global polynomial ring, the standard basis of the localization of I may be much simpler than the Gröbner basis of I.

```
> Q := RationalField();
> R<x,y,z> := PolynomialRing(Q, 3);
> I := Ideal([x^2 - x*y^3 + z^3, x*y + y^2 + z, x + y^2 - z^2]);
> Groebner(I); I;
Ideal of Polynomial ring of rank 3 over Rational Field
Order: Lexicographical
Variables: x, y, z
Inhomogeneous, Dimension 0
Groebner basis:
[
    x - y*z + 127/1052*z^9 + 585/1052*z^8 + 233/263*z^7 + 1273/1052*z^6 +
        695/526*z^5 + 223/1052*z^4 + 569/526*z^3 + 35/1052*z^2 - z,
    y^2 + y*z - 127/1052*z^9 - 585/1052*z^8 - 233/263*z^7 - 1273/1052*z^6 -
        695/526*z^5 - 223/1052*z^4 - 569/526*z^3 - 1087/1052*z^2 + z,
    y*z^2 + 51/263*z^9 + 208/263*z^8 + 308/263*z^7 + 476/263*z^6 + 465/263*z^5 +
        278/263*z^4 + 691/263*z^3 + 217/263*z^2,
    z^10 + 5*z^9 + 10*z^8 + 15*z^7 + 16*z^6 + 9*z^5 + 12*z^4 + 13*z^3 + 2*z^2
]
> QuotientDimension(I);
12
>
> IL := Localization(I);
> StandardBasis(IL);
[
    x + y^2,
    y^2 - y^3 + z,
    z^2
]
> QuotientDimension(IL);
4
```

117.5 Operations on Ideals

In the following, note that since ideals of a full polynomial ring P are regarded as subrings of P, the ring P itself is a valid ideal as well (the ideal containing 1).

117.5.1 Basic Operations

`I + J: RngMPolLoc, RngMPolLoc -> RngMPolLoc`

Given ideals I and J of the same polynomial ring P, return the sum of I and J, which is the ideal generated by the generators of I and those of J.

`I * J: RngMPolLoc, RngMPolLoc -> RngMPolLoc`

Given ideals I and J of the same polynomial ring P, return the product of I and J, which is the ideal generated by the products of the generators of I and those of J.

`I ^ k: RngMPolLoc, RngIntElt -> RngMPolLoc`

Given an ideal I of the polynomial ring P, and an integer k, return the k-th power of I.

`QuotientDimension(I): RngMPol -> RngIntElt`

Given an ideal I of a local polynomial ring R over a field K, return the dimension of P/I as a K-vector space. Note that this is quite different from the function *Dimension* below (which returns the Krull dimension of an ideal).

`Generic(I): RngMPolLoc -> RngMPolLoc`

Given an ideal I of a generic local polynomial ring R, return R.

`LeadingMonomialIdeal(I): RngMPolLoc -> RngMPolLoc`

Given an ideal I, return the leading monomial ideal of I; that is, the ideal generated by all the leading monomials of I.

`I meet J: RngMPolLoc, RngMPolLoc -> RngMPolLoc`

Given ideals I and J of the same polynomial ring P, return the intersection of I and J.

`&meet S: [ RngMPolLoc ] -> RngMPolLoc`

Given a set or sequence S of ideals of the same local polynomial ring R, return the intersection of all the ideals of S.

117.5.2 Ideal Predicates

`I eq J: RngMPolLoc, RngMPolLoc -> BoolElt`

Given two ideals I and J of the same polynomial ring P, return whether I and J are equal.

`I ne J: RngMPolLoc, RngMPolLoc -> BoolElt`

Given two ideals I and J of the same polynomial ring P, return whether I and J are not equal.

`I notsubset J: RngMPolLoc, RngMPolLoc -> BoolElt`

Given two ideals I and J in the same polynomial ring P return whether I is not contained in J.

`I subset J: RngMPolLoc, RngMPolLoc -> BoolElt`

Given two ideals I and J in the same polynomial ring P return whether I is contained in J.

`IsZero(I): RngMPolLoc -> BoolElt`

Given an ideal I of the local polynomial ring R, return whether I is the zero ideal (contains zero alone).

`IsProper(I): RngMPolLoc -> BoolElt`

Given an ideal I of the local polynomial ring R, return whether I is proper; that is, whether I is strictly contained in R (or whether the standard basis of I does not contain 1 alone).

`IsZeroDimensional(I): RngMPolLoc -> BoolElt`

Given an ideal I of the local polynomial ring R, return whether I is zero-dimensional (so the quotient of P by I has non-zero finite dimension as a vector space over the coefficient field – see the section on dimension for further details). Note that the ring R has dimension -1, so it is not zero-dimensional.

Example: Ideal Arithmetic in a Local Ring (ex-b95527)

We construct some ideals in $\mathbb{Q}[x, y, z]$ and perform basic arithmetic on them.

```
> R<x,y,z> := LocalPolynomialRing(RationalField(), 3);
> I := ideal<R | x*y - z, x^3*z^2 - y^2, x*z^3 - x - y>;
> J := ideal<R | x*y - z, x^2*z - y, x*z^3 - x - y>;
> A := I * J;
> _ := StandardBasis(A);
> A;
Ideal of Localization of Polynomial Ring of rank 3 over Rational Field
Order: Local Lexicographical
Variables: x, y, z
Inhomogeneous, Dimension 0
Standard basis:
[
    x^2 - y^2 + 2*x^3*z,
    x*y + y^2 - x^3*z,
    y^3,
    x*z + y*z,
    y*z,
    z^2
]
> M := I meet J;
> M;
Ideal of Localization of Polynomial Ring of rank 3 over Rational Field
Order: Local Lexicographical
Variables: x, y, z
Homogeneous
Basis:
[
    x + y,
    y^2,
    z
]
> A eq M;
false
> A subset M;
true
```

117.5.3 Operations on Elements of Ideals

`f in I: RngMPolLocElt, RngMPolLoc -> BoolElt`

Given a polynomial f from a local polynomial ring R, together with an ideal I of R, return whether f is in I.

`NormalForm(f, I): RngMPolLocElt, RngMPolLoc -> RngMPolLocElt`

Given a polynomial f from a local polynomial ring R, together with an ideal I of R, return a normal form of f with respect to (the standard basis of) I. The normal form of f is zero if and only if f is in I.

`f notin I: RngMPolLocElt, RngMPolLoc -> BoolElt`

Given a polynomial f from a polynomial ring P, together with an ideal I of P, return whether f is not in I.

Example: Element Operations with Local Ideals (ex-e2cb48)

We demonstrate the element operations with respect to an ideal of the localization of $\mathbb{Q}[x, y, z]$.

```
> R<x,y,z> := LocalPolynomialRing(RationalField(), 3);
> I := ideal<R | (x + y)^3, (y - z)^2, y^2*z + z>;
> NormalForm(y^2*z + z, I);
0
> NormalForm(x^3, I);
-3*x^2*y
> x + y in I;
false
```

117.6 Changing Coefficient Ring

The `ChangeRing` function enables the changing of the coefficient ring of a local polynomial ring or ideal.

`ChangeRing(I, L): RngMPolLoc, Rng -> RngMPolLoc`

Given an ideal I of a local polynomial ring $R = K[x_1, \ldots, x_n]$ of rank n with coefficient ring K, together with a field L, construct the ideal J of the polynomial field $S = L[x_1, \ldots, x_n]$ obtained by coercing the coefficients of the elements of the basis of I into L. It is necessary that all elements of the old coefficient field K can be automatically coerced into the new coefficient field L. If K and L are fields and K is known to be a subfield of L and the current basis of I is a standard basis, then the basis of J is marked automatically to be a standard basis of J.

117.7 Changing Monomial Order

Often one wishes to change the monomial order of an ideal. Magma allows one to do this by use of the `ChangeOrder` function.

`ChangeOrder(I, Q): RngMPolLoc, RngMPolLoc -> RngMPolLoc, Map`

Given an ideal I of the local polynomial ring $R = K[x_1, \ldots, x_n]$, together with a local polynomial ring S of rank n (with possibly a different order to that of R), return the ideal J of S corresponding to J and the isomorphism f from R to S. The map f simply maps $R.i$ to $S.i$ for each i.

`ChangeOrder(I, order): RngMPolLoc, ... -> RngMPolLoc, Map`

Given an ideal I of the polynomial ring $P = R[x_1, \ldots, x_n]$, together with a monomial order order (see Section *Elements and Local Monomial Orders*), construct the polynomial ring $Q = R[x_1, \ldots, x_n]$ with order order, and then return the ideal J of Q corresponding to I and the isomorphism f from P to Q. See the section on monomial orders for the valid values for the argument order. The map f simply maps $P.i$ to $Q.i$ for each i.

117.8 Dimension of Ideals

Let I be an ideal of the local polynomial ring $K[x_1, \ldots, x_n]_{\langle x_1, \ldots, x_n \rangle}$, where K is a field. As for polynomial rings, the dimension of the ideal I can be defined as the maximum of the cardinalities of all the independent sets modulo I (see Section *Dimension of Ideals* for details).

`Dimension(I): RngMPolLoc -> RngIntElt, [ RngIntElt ]`

Given an ideal I of a local polynomial ring R defined over a field, return the dimension d of I, together with a (sorted) sequence U of integers of length d such that the variables of P corresponding to the integers of U constitute a maximally independent set modulo I. If I is the full local

polynomial ring R, the dimension is defined to be -1, and the second return value is not set. The algorithm implemented is that given in [BW93, p. 449].

117.9 Bibliography

- [BW93] Thomas Becker and Volker Weispfenning. *Gröbner Bases*. Graduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 1993.
- [CLO98] David Cox, John Little, and Donal O'Shea. *Using Algebraic Geometry*. Graduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 1998.
- [DL06] Wolfram Decker and Christoph Lossen. *Computing in Algebraic Geometry*. Volume 16 of Algorithms and Computation in Mathematics. Springer, New York–Berlin–Heidelberg, 2006.
- [GP02] G.-M. Greuel and G. Pfister. *A Singular Introduction to Commutative Algebra*. Springer-Verlag, Berlin–Heidelberg–New York, 2002. ISBN 3-540-42897-6.

Chapter 118

Affine Algebras

118.1 Introduction

An affine algebra in Magma is simply the quotient ring of a multivariate polynomial ring $P = R[x_1, \ldots, x_n]$ by an ideal J of P. Such rings arise commonly in commutative algebra and algebraic geometry. They can also be viewed as generalizations of number fields and algebraic function fields, when R is a field.

The elements of affine algebras are simply multivariate polynomials which are always kept reduced to normal form modulo the ideal J of "relations". Practically all operations which are applicable to multivariate polynomials are also applicable in Magma to elements of affine algebras (when meaningful).

If the ideal J of relations defining an affine algebra $A = R[x_1, \ldots, x_n]/J$ is *maximal* and R is a field, then A is a field and may be used with any algorithms in Magma which work over fields. Factorization of polynomials over such affine algebras is also supported (including fields of small characteristic, since V2.10).

If an affine algebra defined over a field has finite dimension considered as a vector space over the coefficient field, extra special operations are available on its elements.

Currently the base ring R may be a field or a Euclidean ring. Further operations for affine algebras over Euclidean rings will be supported in the future.

An affine algebra has type `RngMPolRes` and its elements type `RngMPolResElt`.

118.2 Creation of Affine Algebras

One can create an affine algebra simply by forming the quotient of a multivariate polynomial ring by an ideal (`quo` constructor or `/` function). A special constructor `AffineAlgebra` is also provided to remove the need to create the base polynomial ring.

`quo< P | J >: RngMPol, RngMPol -> RngMPolRes`

`quo< P | a`$_1$`, ..., a`$_r$` >: RngMPol, RngMPolElt, ..., RngMPolElt -> RngMPolRes, Map`

Given a multivariate polynomial ring P and an ideal J of P, return the quotient ring P/J. The ideal J may either be specified as an ideal or by a list a_1, a_2, ..., a_r, of generators which all lie in P. The angle bracket notation can be used to assign names to the indeterminates: `Q<q, r> := quo< I | I.1 + I.2, I.2^2 - 2, I.3^2 + I.4 >;`.

`P / J: RngMPol, RngMPol -> RngMPolRes`

Given a multivariate polynomial ring P and an ideal J of P, return the quotient affine algebra P/J.

`AffineAlgebra< R, X | L >: Fld, List, List -> RngMPolRes`

Given a ring R, a list X of n identifiers, and a list L of polynomials (relations) in the n variables

X, create the affine algebra of rank n with base ring R with given quotient relations; i.e., return $R[X]/\langle L \rangle$. The angle bracket notation can be used to assign names to the indeterminates.

Example: Number Field Extensions as Affine Algebras (ex-3c15de)

One can create a relative extension of an algebraic number field as an affine algebra. The multivariate representation will often be more efficient than an absolute representation because of the sparsity of the elements in the field.

```
> Q := RationalField();
> A<x, y> := AffineAlgebra<Q, x, y | x^2 - y^2 + 2, y^3 - 5>;
> A;
Affine Algebra of rank 2 over Rational Field
Lexicographical Order
Variables: x, y
Quotient relations:
[
    x^2 - y^2 + 2,
    y^3 - 5
]
> x^2;
y^2 - 2
> x^-1;
2/17*x*y^2 + 5/17*x*y + 4/17*x
> P<z> := PolynomialRing(Q);
> MinimalPolynomial(x);
z^6 + 6*z^4 + 12*z^2 - 17
> MinimalPolynomial(x^-1);
z^6 - 12/17*z^4 - 6/17*z^2 - 1/17
> MinimalPolynomial(y);
z^3 - 5
```

Another important construction is to create an affine algebra over a rational function field to obtain an algebraic function field:

```
> F<t> := FunctionField(IntegerRing());
> A<x, y> := AffineAlgebra<F, x, y | t*x^2 - y^2 + t + 1, y^3 - t>;
> P<z> := PolynomialRing(F);
> x^-1;
(-t^2 - t)/(t^3 + 2*t^2 + 3*t + 1)*x*y^2 - t^2/(t^3 + 2*t^2 + 3*t + 1)*x*y
    + (-t^3 - 2*t^2 - t)/(t^3 + 2*t^2 + 3*t + 1)*x
> MinimalPolynomial(x);
z^6 + (3*t + 3)/t*z^4 + (3*t^2 + 6*t + 3)/t^2*z^2 + (t^3 + 2*t^2 + 3*t +
    1)/t^3
> MinimalPolynomial(x^-1);
z^6 + (3*t^3 + 6*t^2 + 3*t)/(t^3 + 2*t^2 + 3*t + 1)*z^4 + (3*t^3 +
   3*t^2)/(t^3 + 2*t^2 + 3*t + 1)*z^2 + t^3/(t^3 + 2*t^2 + 3*t + 1)
```

In this example we can consider y as a cube root of the transcendental indeterminate t. Note that in general the (Krull) dimension of the ideal defining the relations may be anything; it need not be 0 or 1 as it is in these examples.

118.3 Operations on Affine Algebras

This section describes operations on affine algebras. Most of the operations are very similar to those for multivariate polynomial rings; such operations are done by mapping the computation to the preimage ideal and then by mapping the result back into the affine algebra. See the corresponding functions for the multivariate polynomial rings for details.

`Q . i: RngMPolRes, RngIntElt -> RngMPolResElt`

Given an affine algebra Q, return the i-th indeterminate of Q as an element of Q.

`CoefficientRing(Q): RngMPolRes -> Rng`

Return the coefficient ring of the affine algebra Q.

`Rank(Q): RngMPolRes -> RngIntElt`

Return the rank of the affine algebra Q (the number of indeterminates of Q).

`DivisorIdeal(I): RngMPolRes -> RngMPol`

Suppose Q is an affine algebra Q which equals the quotient ring P/J, where P is a polynomial ring and J is an ideal of P. Given I such that I an ideal of Q, or $I = Q$ itself, return the ideal J of P.

`PreimageIdeal(I): RngMPolRes -> RngMPol`

Suppose Q is an affine algebra Q which equals the quotient ring P/J, where P is a polynomial ring and J is an ideal of P. Given I such that I an ideal of Q, or $I = Q$ itself, return the ideal I' of P such that the image of I' under the natural epimorphism $P \to Q$ is I.

`PreimageRing(Q): RngMPolRes -> RngMPol`

Given an affine algebra Q which is the quotient ring P/J, where P is a polynomial ring and J an ideal of P, return the polynomial ring P.

`OriginalRing(Q): RngMPolRes -> Rng`

Return the generic polynomial ring P such that Q is P/J for some ideal J of P.

`I eq J: RngMPolRes, RngMPolRes -> BoolElt`

Given two ideals I and J of the same affine algebra Q, return `true` if and only if I and J are equal.

`I subset J: RngMPolRes, RngMPolRes -> BoolElt`

Given two ideals I and J of the same affine algebra Q, return `true` if and only if I is contained in J.

`I + J: RngMPolRes, RngMPolRes -> RngMPolRes`

Given two ideals I and J of the same affine algebra Q, return the sum $I + J$.

`I * J: RngMPolRes, RngMPolRes -> RngMPolRes`

Given two ideals I and J of the same affine algebra Q, return the product $I * J$.

`I ^ n: RngMPolRes, RngIntElt -> BoolElt`

Given an ideal I of an affine algebra Q and an integer n, return the power I^n.

`I meet J: RngMPolRes, RngMPolRes -> RngMPolRes`

Given two ideals I and J of the same affine algebra Q, return the intersection $I \cap J$.

`IsProper(I): RngMPolRes -> BoolElt`

Given an ideal I of the affine algebra Q, return whether I is proper; that is, whether I is strictly contained in Q.

`IsZero(I): RngMPolRes -> BoolElt`

Given an ideal I of the affine algebra Q, return whether I is the zero ideal. Note that this is equivalent to whether the preimage ideal of I is the divisor ideal of Q.

`IsPrime(I): RngMPolRes -> BoolElt`

Given an ideal I of the affine algebra Q, return whether I is a prime ideal.

`IsPrimary(I): RngMPolRes -> BoolElt`

Given an ideal I of the affine algebra Q, return whether I is a primary ideal.

`IsRadical(I): RngMPolRes -> BoolElt`

Given an ideal I of the affine algebra Q, return whether I is a radical ideal.

`PrimaryDecomposition(I): RngMPolRes -> [ RngMPolRes ], [ RngMPolRes ]`

Given an ideal I of the affine algebra Q, return the primary decomposition of I, together with the associated primes.

`RadicalDecomposition(I): RngMPolRes -> [ RngMPolRes ]`

Given an ideal I of the affine algebra Q, return the (prime) decomposition of the radical of I.

Example: Ideals of Affine Algebras (ex-f9694a)

We illustrate the operations on ideals of affine algebras.

```
> Q := RationalField();
> A<x,y,z> := AffineAlgebra<Q,x,y,z | x^2 - y + 1, y^3 + z - 1>;
> A;
Affine Algebra of rank 3 over Rational Field
Lexicographical Order
Variables: x, y, z
Quotient relations:
[
    x^2 - y + 1,
    y^3 + z - 1
]
> I := ideal<A | x^3*y*z^2>;
> IsRadical(I);
false
> Radical(I);
Affine Algebra of rank 3 over Rational Field
Lexicographical Order
Variables: x, y, z
Quotient relations:
[
    x^2 - y + 1,
    y^3 + z - 1
]
Generating basis:
[
    x*y^2 + x*y - x*z + x,
    y*z,
    z^2 - z
]
> PQ, PP := PrimaryDecomposition(I);
> #PQ;
3
> PQ[1];
Affine Algebra of rank 3 over Rational Field
Lexicographical Order
Variables: x, y, z
Quotient relations:
[
    x^2 - y + 1,
    y^3 + z - 1
]
Generating basis:
[
    y + 5/81*z^3 + 1/9*z^2 + 1/3*z - 1,
    x*z^3,
    y + 5/81*z^3 + 1/9*z^2 + 1/3*z - 1,
    z^4
```

(continues on next page)

(continued from previous page)

```
]
> PP[1];
Affine Algebra of rank 3 over Rational Field
Lexicographical Order
Variables: x, y, z
Quotient relations:
[
    x^2 - y + 1,
    y^3 + z - 1
]
Generating basis:
[
    x,
    y - 1,
    z
]
```

118.4 Maps between Affine Algebras

Magma includes functions for working with maps between affine algebras.

AffineAlgebraMapKernel(phi): Map -> MPol

Return the kernel of the homomorphism ϕ of affine algebras.

118.5 Finite Dimensional Affine Algebras

If an affine algebra is defined over a field and has finite dimension considered as a vector space over its coefficient field, extra special operations are available on its elements. Similar operations for affine algebras defined over general Euclidean rings will be supported in the future.

HasFiniteDimension(Q): RngMPolRes -> BoolElt

Given an affine algebra Q defined over a field, return whether Q has finite dimension.

Dimension(Q): RngMPolRes -> RngIntElt

Given a finite dimensional affine algebra Q defined over a field, return the dimension of Q.

VectorSpace(Q): RngMPolRes -> ModTupFld, Map

Given a finite dimensional affine algebra Q defined over a field, construct the vector space V isomorphic to Q, and return V together with the isomorphism f from Q onto V.

MonomialBasis(Q): RngMPolRes -> [RngMPolResElt]

Given a finite dimensional affine algebra Q defined over a field, return the basis B of monomials of Q. This is a sequence of monomials in Q of length d, such that the image $f(B[i]) = V.i$ where V and f are the return values of `VectorSpace` above.

MatrixAlgebra(Q): RngMPolRes -> AlgMat, Map

Given a finite dimensional affine algebra Q defined over a field, construct the matrix algebra A isomorphic to Q, and return A together with the isomorphism f from Q onto A.

RepresentationMatrix(f): RngMPolResElt -> AlgMatElt

Given an element f of a finite dimensional affine algebra Q defined over a field, return the representation matrix of f, which is a d by d matrix over the coefficient field of Q (where d is the dimension of Q) which represents f.

IsUnit(f): RngMPolResElt -> BoolElt

Given an element f of a finite dimensional affine algebra Q defined over a field, return whether f is a unit.

`IsNilpotent(f): RngMPolResElt -> BoolElt, RngIntElt`

Given an element f of a finite dimensional affine algebra Q defined over a field, return whether f is nilpotent, and if so, return also the smallest q such that $f^q = 0$.

`MinimalPolynomial(f): RngMPolResElt -> RngUPol`

Given an element f of a finite dimensional affine algebra Q defined over a field, return the minimal polynomial of f as a univariate polynomial over the coefficient field of Q.

Example: Minimal Polynomial of an Algebraic Number (ex-caec0e)

Suppose we wish to find the minimal polynomial of $\theta = \sqrt{2} + \sqrt[3]{5}$ over $\mathbb{Q}$. To do this we can just compute the minimal polynomial of (the coset of) $x + y$ over $\mathbb{Q}$ in the affine algebra $Q[x, y]/(x^2 - 2, y^3 - 5)$.

```
> Q := RationalField();
> A<x, y> := AffineAlgebra<Q, x, y | x^2 - 2, y^3 - 5>;
> UP<z> := PolynomialRing(Q);
> MinimalPolynomial(x + y);
z^6 - 6*z^4 - 10*z^3 + 12*z^2 - 60*z + 17
```

118.6 Affine Algebras which are Fields

If the ideal J of relations defining an affine algebra $A = K[x_1, \ldots, x_n]/J$, where K is a field, is *maximal*, then A is a field and may be used with any algorithms in Magma which work over fields. Factorization of polynomials over such affine algebras is also supported (in any characteristic, since V2.10). The examples below will demonstrate some of the applications available.

Note that an affine algebra defined over a field which itself is a field also has finite dimension when considered as a vector space over its coefficient field, so all of the operations in the previous section are also available.

Example: Elliptic Curve Function Field (ex-b17854)

We create the function field $F = \mathbb{Q}(a, b, x)$ and then the affine algebra $A = F[y]/ < y^2 - (x^3 + ax + b) >$ (which is also equivalent to an algebraic function field). This then allows us to create a generic elliptic curve E over A and compute the coordinates of multiples of a generic point easily.

```
> Q := RationalField();
> F<x, a, b> := FunctionField(Q, 3);
> A<y> := AffineAlgebra<F, y | y^2 - (x^3 + a*x + b)>;
> IsField(A);
true
> y^2;
x^3 + x*a + b
> y^-1;
1/(x^3 + x*a + b)*y
> E := EllipticCurve([A | a, b]);
> E;
Elliptic Curve defined by y^2 = x^3 + a*x + b over Affine Algebra of rank 1 over
    Rational function field of rank 3 over Rational Field
    Variables: x, a, b
> p := E ! [x, y];
> p;
(x : y : 1)
> q := 2*p;
> q;
((1/4*x^4 - 1/2*x^2*a - 2*x*b + 1/4*a^2)/(x^3 + x*a + b) : (1/8*x^6 +
5/8*x^4*a + 5/2*x^3*b - 5/8*x^2*a^2 - 1/2*x*a*b - 1/8*a^3 - b^2)/(x^6
+ 2*x^4*a + 2*x^3*b + x^2*a^2 + 2*x*a*b + b^2)*y : 1)
```

(continues on next page)

(continued from previous page)

```
> c := LeadingCoefficient(q[2]);
> Denominator(c);
x^6 + 2*x^4*a + 2*x^3*b + x^2*a^2 + 2*x*a*b + b^2
> Factorization($1);
[
    <x^3 + x*a + b, 2>
]
```

Example: Factorization over an Affine Algebra (ex-701393)

Starting with the same affine algebra $A = \mathbb{Q}(a, b, x)F[y]/ < y^2 - (x^3 + ax + b) >$ as in the last example, we factor some univariate polynomials over A. A is of course isomorphic to an absolute field, but the presentation given may be much more convenient to the user.

```
> Q := RationalField();
> F<x, a, b> := FunctionField(Q, 3);
> A<y> := AffineAlgebra<F, y | y^2 - (x^3 + a*x + b)>;
> P<z> := PolynomialRing(A);
> f := z^2 - (x^3 + a*x + b);
> f;
z^2 + -x^3 - x*a - b
> time Factorization(f);
[
    <z - y, 1>,
    <z + y, 1>
]
Time: 0.019
```

Example: Multiple Field Extension (ex-c88a64)

In this final example, A is isomorphic to an algebraic number field, but its presentation may be more convenient than an absolute presentation (and may lead to sparser expressions for elements).

```
> Q := RationalField();
> A<a,b,c> := AffineAlgebra<Q, a,b,c | a^2 - b*c + 1, b^2 - c + 1, c^2 + 2>;
> P<x> := PolynomialRing(A);
> time Factorization(x^2 + 2);
[
    <x - c, 1>,
    <x + c, 1>
]
Time: 0.080
> time Factorization(x^2 - b*c + 1);
[
    <x - a, 1>,
    <x + a, 1>
]
Time: 0.090
> MinimalPolynomial(a);
x^8 + 4*x^6 + 2*x^4 - 4*x^2 + 9
> time Factorization(P ! $1);
[
    <x - a, 1>,
    <x + a, 1>,
    <x - 1/3*a*b*c - 2/3*a*b + 1/3*a*c - 1/3*a, 1>,
    <x + 1/3*a*b*c + 2/3*a*b - 1/3*a*c + 1/3*a, 1>,
    <x^4 + 2*x^2 - 2*c - 1, 1>
```

(continues on next page)

(continued from previous page)

```
]
Time: 2.809
```

NumberField(A): RngMPolRes -> FldNum, Map

```
Absolute: BoolElt                    Default: false
```

Given an affine algebra $A = K[x_1, \ldots, x_n]/J$ over the rational field or a number field K, where the ideal J is maximal (so that A is a field), return a number field N isomorphic to A, together with the isomorphism from A to N.

By default, N is a *relative* number field, presented as a tower with one extension for each variable. But if the parameter `Absolute` is set to `true`, then N is instead an absolute number field, obtained by computing a primitive element of A.

Example: Number Field of Degree 8 (ex-7885e5)

If the defining ideal of an affine algebra A is maximal, then A is a field; here it is isomorphic to a number field of degree 8. By default the field is presented as a relative tower, with one extension for each of the variables. We use the returned isomorphism to move elements between the two presentations.

```
> Q := RationalField();
> A<a,b,c> := AffineAlgebra<Q, a,b,c |
>     a^2 - b*c + 1, b^2 - c + 1, c^2 + 2>;
> IsField(A);
true
> N<w,v,u>, m := NumberField(A);
> P<x>:=PolynomialRing(BaseRing(N));
> N;
Number Field with defining polynomial x^2 - u*v + 1 over its
    ground field
> AbsoluteDegree(N);
8
> m(a);
w
> m(a*b) eq m(a)*m(b);
true
> assert $1;
> ma := m(a);
> ma @@ m eq a;
true
> assert $1;
```

We may instead ask for an absolute number field, presented by a single defining polynomial, and apply some basic number-field machinery to it.

```
> Na<v>, ma := NumberField(A : Absolute);
> Na;
Number Field with defining polynomial x^8 + 4*x^6 + 2*x^4 -
    4*x^2 + 9 over the Rational Field
> ma(a*b) eq ma(a)*ma(b);
true
> assert $1;
> Signature(Na);
0 4
```

118.7 Rings and Fields of Fractions of Affine Algebras

Given any affine algebra $Q = K[x_1, \ldots, x_n]/J$, where K is a field, one may create the *ring of fractions* R of Q. This is the set of fractions a/b, where $a, b \in Q$ and b is invertible, and it forms a ring. The defining ideal J does not need to be zero-dimensional. The ring of fractions R is itself represented internally by an affine algebra over an appropriate rational function field, but has the appearance to the user of the set of fractions, so one may access the numerator and denominator of elements of R, for example. If the ideal J is prime, then R is the field of fractions of A and may be used with any algorithms in Magma which work over fields. For example, factorization of polynomials over such fields of fractions is supported (in any characteristic).

Rings of fractions have type `RngFunFrac` and their elements `RngFunFracElt`.

`RingOfFractions(Q): RngMPolRes -> RngFunFrac`

`FieldOfFractions(Q): RngMPolRes -> RngFunFrac`

Given an affine algebra Q over a field K, return the ring of fractions of Q. The only difference between the two functions is that for `FieldOfFractions`, the defining ideal of Q must be prime.

`Numerator(a): RngFunFracElt -> RngMPolResElt`

`Denominator(a): RngFunFracElt -> RngMPolResElt`

Given an element a from the ring of fractions of an affine algebra Q, return the numerator (resp. denominator) of a as an element of Q.

Example: Field of Fractions (ex-a76faa)

We create the field of fractions of an affine algebra and note the basic operations.

```
> A<x,y> := AffineAlgebra<RationalField(), x,y | y^2 - x^3 - 1>;
> IsField(A);
false
> F<a,b> := FieldOfFractions(A);
> F;
Ring of Fractions of Affine Algebra of rank 2 over Rational Field
Lexicographical Order
Variables: x, y
Quotient relations:
[
    x^3 - y^2 + 1
]
> a;
a
> b;
b
> a^-1;
> a^-1;
1/(b^2 - 1)*a^2
> b^-1;
1/b
> c := b/a;
> c;
b/(b^2 - 1)*a^2
> Numerator(c);
x^2*y
> Denominator(c);
y^2 - 1
> P<X> := PolynomialRing(F);
> time Factorization(X^3 - b^2 + 1);
[
    <X - a, 1>,
```

(continues on next page)

(continued from previous page)

```
    <X^2 + a*X + a^2, 1>
]
Time: 0.000
> P<X,Y> := PolynomialRing(F, 2);
> time Factorization((X + Y)^3 - b^2 + 1);
[
    <X + Y - a, 1>,
    <X^2 + 2*X*Y + a*X + Y^2 + a*Y + a^2, 1>
]
Time: 0.030
> time Factorization((b*X^2 - a)*(a*Y^3 - b + 1)*(X^3 - b^2 + 1));
[
    <Y^3 - 1/(b + 1)*a^2, 1>,
    <X - a, 1>,
    <X^2 - 1/b*a, 1>,
    <X^2 + a*X + a^2, 1>
]
Time: 0.010
```

Example: Extension and Contraction of Ideals (ex-d7e7e8)

This example shows the internal operations underlying the method of constructing the field of fractions. If the ideal of relations has dimension d, then the sequence L of d maximally independent variables is passed to the extension/contraction construction, which creates a rational function field with d variables such that the ideal of relations over this field now becomes zero dimensional. Appropriate maps are set up, too.

```
> Q := RationalField();
> A<x,y> := AffineAlgebra<RationalField(), x,y | y^2 - x^3 - 1>;
> IsField(A);
false
> I := DivisorIdeal(A);
> I;
Ideal of Polynomial ring of rank 2 over Rational Field
Lexicographical Order
Variables: x, y
Groebner basis:
[
    x^3 - y^2 + 1
]
> d, L := Dimension(I);
> d;
1
> L;
[ 2 ]
> E, f := Extension(I, L);
> E;
Ideal of Polynomial ring of rank 1 over Multivariate rational function
    field of rank 1 over Integer Ring
Graded Reverse Lexicographical Order
Variables: x
Basis:
[
    x^3 - y^2 + 1
]
> F := Generic(E)/E;
```

(continues on next page)

(continued from previous page)

```
Affine Algebra of rank 1 over Multivariate rational function field of
    rank 1 over Integer Ring
Graded Reverse Lexicographical Order
Variables: x
Quotient relations:
[
    x^3 - y^2 + 1
]
> g := map<A -> F | x :-> F!f(x)>;
>
> g(x);
x
> g(y);
y
> g(x)^-1;
1/(y^2 - 1)*x^2
> g(y)^-1;
1/y
> g(x^2 + x*y);
x^2 + y*x
> g(x^2 + x*y)^-1;
y^2/(y^5 + y^4 - y^3 - 2*y^2 + 1)*x^2 + 1/(y^3 + y^2 - 1)*x - y/
    (y^3 + y^2 - 1)
> $1 * $2;
1
```

Chapter 119

Modules over Multivariate Rings

119.1 Introduction

This chapter describes modules over multivariate polynomial rings and related rings. The fundamental tool for computing with such modules is the construction of Gröbner bases for modules, since these rings are not principal ideal rings in general (so standard matrix echelonization algorithms are not applicable). In this chapter, unless otherwise indicated, a **ring** R will refer to one of the following:

(a) Multivariate Polynomial Ring (Chapters Multivariate Polynomial Rings and *Gröbner Bases*). Currently the coefficient ring of such a ring may be a field or Euclidean ring (even operations such as syzygy modules or free resolutions work over modules whose coefficient rings are Euclidean but not fields).

(b) Local Polynomial Ring (Localization of a Multivariate Polynomial Ring: Chapter *Local Polynomial Rings*; new in V2.15). Currently the coefficient ring of such a ring must be a field.

(c) Affine Algebra (Chapter *Affine Algebras*). Currently the coefficient ring of such a ring must be a field.

(d) Exterior Algebra (Chapter Finitely Presented Algebras; new in V2.15). Currently the coefficient ring of such a ring must be a field. Strictly speaking, this is a skew-commutative ring, so is not a commutative ring, and the associated modules are left R-modules, but the operations on R-modules in this chapter are practically all applicable if R is such an algebra also, so the term 'a ring R' will include such an algebra in this chapter.

In this chapter, the term "**module**" will always refer to an R-module, where R is one of the above types of ring, and such a module will have type `ModMPol` (or may have type `ModMPolGrd` if graded; see below). So we assume that the reader is generally familiar with such base rings and their ideals in Magma; see the relevant chapters for background. Many of the concepts and tools of Gröbner basis theory carry over from these types of rings.

119.2 Module Basics: Embedded and Reduced Modules

All of the modules considered in this chapter are ambient modules or embedded in such a module. We call an R-module **ambient** if it has the **explicit** presentational form $R^k/\langle relations \rangle$, where the relations are elements of R^k (and they may be zero or not even determined, initially). Elements of an ambient R-module M are represented explicitly as vectors in R^k, and M is always generated by the k unit vectors. The **degree** of M is k.

An arbitrary module S may have a representation as a submodule of such an ambient A, which is referred to as its **ambient module**. Hence the most general definition of a module is as a sub-quotient of a free module. If A has no relations then S is just a submodule of a free module (namely, A). However, in this case, S will often also have an *internal* representation in presentational form that is essential for much of its fundamental functionality. In any case, the primary representation of elements of such an embedded module S is as vectors in the ambient.

As with vector spaces, there are two basic ways that modules can be defined in Magma: as embedded or reduced modules. A general subquotient as described above is in embedded form, but ambients may also be defined of either reduced or embedded type. The type primarily affects the way submodules and quotient modules are created. Briefly, submodules and quotient modules of embedded modules stay in embedded form (as generally proper submodules of an ambient) whereas submodules or quotients of reduced modules are always returned in presentational form as ambients, with connecting homomorphisms to link them explicitly to the original module. The two types are described in a bit more detail below. For illustration, see the examples at the end of Section *Submodules and Quotient Modules*.

Embedded modules are created in general via the function *EModule*, which returns a free embedded module, and in principle mimic the embedded R-spaces (as created by the function `RSpace(R, n): Rng, RngIntElt → ModTupRng(R, k)` in Chapter Free Modules). Such modules are always presented with their elements and bases lying in an ambient module $R^k/\langle$ *relations*$\rangle$. The modules are basically implemented as extensions of the multivariate polynomial ideal type (or affine algebra type if non-zero relations are present), where columns are internally added to monomials in a polynomial to represent a vector. Many operations applicable to ideals, including various Gröbner basis operations, naturally extend to such modules.

Starting with an ambient embedded module $M = R^k/\langle$ *relations*$\rangle$, when a submodule S of M is created, the ambient module of S is still M, so the elements of S are represented as elements of R^k (modulo the relations if present); this therefore also applies to elements of any basis of S, including the Gröbner basis of S. Thus S itself may be not ambient and this is the only situation in which non-ambients can occur. Similarly, when a quotient module Q of M is constructed, the elements of Q appear as elements of R^k, while Q simply gains more relations than M, but its generators are usually not minimally reduced.

Reduced modules are created in general via the function *RModule*, which returns a free reduced module, and are more abstract and mimic the reduced modules with action over fields and Euclidean rings (as created by the function `RModule(R, n): Rng, RngIntElt → ModRng(R, k)` in Chapter Free Modules). Such modules are **always ambient**, so always have the abstract form $R^n/\langle$*relations*$\rangle$, and the relationships between such modules are managed by **morphisms** lying in the background. The Gröbner basis techniques and properties are also hidden from the user in general.

Starting from a reduced module $M = R^k/\langle$*relations*$\rangle$, when a submodule S (having s generators $v_1, \ldots, v_s$) of M is created, S is generally created as $R^s/\langle$ *relations*$_S\rangle$ (where the relations for S are initially unknown and are only computed when needed) and a morphism is stored from S to M, which maps the i-th unit vector of S to v_i in M. Similarly, a quotient module Q of M is constructed as another ambient module, usually with minimal generators, and a morphism from M onto Q is stored in the background. All morphisms between modules can be accessed via the function *Morphism*.

For any module M, there exists an isomorphic reduced **presentation module** P, which is always ambient, since P is reduced. If M is embedded, then P is a reduced module equivalent to M (and morphisms in the background allow automatic coercion between M and P). Otherwise, M is already reduced so P is simply identical to M. Some functions (such as *FreeResolution*) always move to the presentation of M, since it is more natural to work only with ambient modules in that context.

Embedded modules are generally preferable when one wishes to work explicitly with Gröbner bases at a very low level, while reduced modules are generally preferable for homological computations since the ambient presentation form is more convenient (particularly for the relevant maps).

Technically, there is little difference in practice between an ambient embedded module and a reduced module, if each module is considered in isolation. The concepts basically refer to how submodules and quotient modules are derived from a given module (and the fact that embedded modules allow non-ambient submodules).

Finally, there is a subclass of reduced modules with the special type `ModMPolGrd`: these are **graded**, which means that they are always generated by homogeneous elements (with respect to the relevant grading). The main distinctive of this type is simply that when one creates a submodule or quotient module of a module of type `ModMPolGrd`, then the generators must be homogeneous, thus ensuring that the new derived module is also graded so will be of type `ModMPolGrd` also. In the future, more functions will be developed which will take modules of type `ModMPolGrd` explicitly. Note also that since the type `ModMPolGrd` ISA `ModMPol` via the type 'ISA' relation, any operation applicable to a module of type `ModMPol` is also applicable to a module of type `ModMPolGrd`.

119.3 Monomial Orders

In this section we describe each of the **module monomial orders** available in Magma. If the user wishes to work with reduced modules only (particularly for homology computations), then the underlying monomial orders and Gröbner bases will probably be rarely of interest to the user, so this section may be skipped. The monomial orders are mostly of interest if one wishes to work with embedded modules with special orders so that the relevant Gröbner bases have special properties. In either case, elements of the module are represented by vectors in an ambient and we refer to the vector component positions as *columns* in analogy to matrix terminology: a presentational module is often just defined by a matrix of relations, the rows giving vectors generating the relation module and the column numbering labelling the components of the vectors. For our modules there can be a non-trivial column weighting, which we think of as applying a shift to the degree of a homogeneous polynomial that occurs as the corresponding vector component of the module element. This is used to define homogeneity and degree of the overall vector. Given an R-module M, suppose that the underlying monomial order of R is $<_R$. A **module monomial** of M is a monomial-column pair consisting of a monomial s of R and a column number c (with $c \geq 1$), written as $s[c]$ in the following. Monomial-column pairs give an (infinite) basis for the elements in a free module R^k and a vector representing an element of M can be decomposed into a sum of scalar multiples of monomial-column pairs just as elements of the polynomial ring R can be written as a sum of scalar multiples of plain monomials. Now suppose that $s_1[c_1]$ and $s_2[c_2]$ are module monomials from M. Any order on the pairs is then fully defined by just specifying exactly when $s_1[c_1] < s_2[c_2]$ with respect to that order. As for multivariate polynomial rings, in the following the argument(s) are described for an order as a list of expressions; that means that the expressions (without the parentheses) should be appended to any base arguments when any particular intrinsic function is called which expects a module monomial order. See [AL94, Sec. 3.5] and [CLO98, Def. 2.4] for motivation and further discussion.

119.3.1 Term Over Position: TOP

Definition: $s_1[c_1] < s_2[c_2]$ iff $s_1 <_R s_2$ or $s_1 = s_2$ and $c_2 > c_1$. The order is specified by the argument (`"top"`). This order is called "TOP" (term over position) since it first compares the underlying monomials (terms with the coefficients ignored (Some authors apply the terms 'monomial' and 'term' in opposite senses to how we do here, so that is why there are the established names 'TOP' and 'POT'; we follow this instead of using 'MOP' and 'POM'!)) and then compares the columns (the positions). The column comparison is ordered so that the first column is the greatest. A Gröbner basis of a module with respect to the `TOP` order is usually the easiest to compute, and corresponds to the `grevlex` order for polynomial rings in a certain way (i.e., the order favours the 'size' of monomials and only gives priority to the columns in a secondary way).

119.3.2 Term Over Position (Weighted): TOPW

Definition (given a sequence W of k integer weights, where k is the degree of the ambient module): write $d_i = \text{Degree}_W(s_i[c_i]) = \text{Degree}(s_i) + W[c_i]$; then $s_1[c_1] < s_2[c_2]$ iff $d_1 < d_2$ or $d_1 = d_2$ and $s_1 <_R s_2$ or $d_1 = d_2$, $s_1 = s_2$, and $c_2 > c_1$. The order is specified by the arguments (`"topw", W`). The weights need not be positive (but must be small integers). This order first compares the degrees of the monomial-coefficient pairs using both the weights of the underlying ring R and the weights on the columns given by W and then proceeds as for the `TOP` order. If there is a natural grading W on the columns of the module, then it is preferable to use this order with W, particularly if submodules of interest are homogeneous or graded w.r.t. W, since then the GB w.r.t. this order will tend to be smaller and easier to compute. Normally one would also make the base order $<_R$ to be one of the `grevlex` or `grevlexw` degree orders (see Subsections *Graded Reverse Lexicographical: grevlex*, *Graded Reverse Lexicographical: grevlex*), so that the order $<$ extends the degree order $<_R$ to a degree order on the module.

119.3.3 Position Over Term: POT

Definition: $s_1[c_1] < s_2[c_2]$ iff $c_2 > c_1$ or $c_1 = c_2$ and $s_1 <_R s_2$. The order is specified by the argument (`"pot"`). This order is called "POT" (position over term) since it first compares the columns and then compares the underlying monomials. The column comparison is ordered so that the first column is the greatest. A Gröbner basis of a module with respect to the `POT` order is like an echelon form of a matrix, since the order gives priority to the columns but this is in general rather harder to compute than the

GB w.r.t. the TOP order.

119.3.4 Position Over Term (Permutation): POTPERM

Definition (given a sequence P of k integers describing a permutation of $[1..k]$, where k is the degree of the ambient module): $s_1[c_1] < s_2[c_2]$ iff $P[c_2] > P[c_1]$ or $c_1 = c_2$ and $s_1 <_R s_2$. The order is specified by the arguments ("potperm", P). This order first compares the columns using the given permutation, and then compares the underlying monomials.

119.3.5 Block TOP-TOP: TOPTOP

Definition (given a integer k): say that a column c is in the 1st block if $c \leq k$ and in the 2nd block if $c > k$; then $s_1[c_1] < s_2[c_2]$ iff c_2 is in the 1st block and c_1 is in the 2nd block, or if the columns are in the same block and $s_1[c_1] < s_2[c_2]$ w.r.t. the TOP order. This order is a block order, like an elimination order for polynomial rings: comparison is first made on the blocks in which the columns lie, and then the TOP order is applied within each block. A GB w.r.t. this order is easier in general to compute than the POT order and so is useful when one wishes to 'eliminate' the first k columns only in a GB.

119.3.6 Block TOP-POT: TOPPOT

Definition (given a integer k): say that a column c is in the 1st block if $c \leq k$ and in the 2nd block if $c > k$; then $s_1[c_1] < s_2[c_2]$ iff c_2 is in the 1st block and c_1 is in the 2nd block, or if the columns are in the same block and $s_1[c_1] < s_2[c_2]$ w.r.t. the TOP/POT order (respective to the 1st/2nd blocks). This order is a block order, like an elimination order for polynomial rings: comparison is first made on the blocks in which the columns lie, and then the TOP order is applied within the 1st block and the POT order is applied within the 2nd block. This is similar to the TOPTOP order, but it may be preferable to order the 2nd block w.r.t. the POT order. Note: POTPOT would equal to POT, and POTTOP does not seem to be useful.

119.4 Basic Creation and Access

An ambient free module $M = R^k$ is created by giving the base ring R (see introduction above), the degree r or a sequence W of r integers for the column weights, and, optionally, an argument specifying the type of module monomial order.

119.4.1 Creation of Ambient Embedded Modules

The following functions create ambient embedded modules.

`EModule(R, k): Rng, RngIntElt -> ModMPol`

Given a ring R, create the ambient embedded module R^k with the default TOP module monomial order.

`EModule(R, k, order): Rng, RngIntElt, MonStgElt, ... -> ModMPol`

Given a ring R, create the ambient embedded module R^k with the module monomial order described by the given order order. See Section *Monomial Orders* for the valid values for order.

`EModule(R, W): Rng, [ RngIntElt ] -> ModMPol`

Given a ring R and a sequence W of k integers, create the ambient embedded module R^k with column weights given by W and with the TOPW module monomial order with weights W.

`EModule(R, W, order): Rng, [ RngIntElt ], MonStgElt, ... -> ModMPol`

Given a ring R and a sequence W of k integers, create the ambient embedded module R^k with column weights given by W and with the module monomial order described by the given order order. See Section *Monomial Orders* for the valid values for order.

119.4.2 Creation of Reduced Modules

The following functions create reduced modules, which are always ambient.

`RModule(R, k): Rng, RngIntElt -> ModMPol`

Given a ring R, create the reduced module R^k with zero column weights.

`RModule(R, W): Rng, [ RngIntElt ] -> ModMPol`

Given a ring R and a sequence W of k integers, create the reduced module R^k with column weights given by W.

`GradedModule(R, k): Rng, RngIntElt -> ModMPol`

Given a ring R, create the reduced graded module R^k with zero column weights. The resulting module has type `ModMPolGrd`, so submodules and quotient modules of it may only be generated by homogeneous elements. Note also that in general it is preferable if possible that the base ring R has a degree ordering (such as the `grevlex` or `grevlexw` orders) so that associated Gröbner bases of derived modules will be easier to compute.

`GradedModule(R, W): Rng, [ RngIntElt ] -> ModMPol`

Given a ring R and a sequence W of k integers, create the reduced graded module R^k with column weights given by W. The resulting module has type `ModMPolGrd`, so submodules and quotient modules of it may only be generated by homogeneous elements.

119.4.3 Localization

`Localization(M): ModMPol -> ModMPol`

Given an R-module M, where $R = K[x_1, \ldots, x_n]$ for a field K, return the corresponding S-module $M_{\langle x_1,\ldots,x_n\rangle}$, where $S = K[x_1, \ldots, x_n]_{\langle x_1,\ldots,x_n\rangle}$ is the localization of R. See Chapter *Local Polynomial Rings* for more information.

119.4.4 Basic Invariants

The following functions access simple defining invariants of a module M.

`Ambient(M): ModMPol -> ModMPol`

`Generic(M): ModMPol -> ModMPol`

Given a module M, return the ambient (or generic) module A in which M is embedded. The only case in which A differs from M is when M is a proper submodule of an ambient embedded module. So if M is reduced, A will always equal M.

`IsAmbient(M): ModMPol -> BoolElt`

Given a module M, return whether M is ambient.

`IsEmbedded(M): ModMPol -> BoolElt`

Given a module M, return whether M is embedded.

`IsReduced(M): ModMPol -> BoolElt`

Given a module M, return whether M is reduced.

`IsRoot(M): ModMPol -> BoolElt`

Given a module M, return whether M is a root (an independent module, not derived via sub- or quotient constructions from another module).

`CoefficientRing(M): ModMPol -> ModMPol`

`BaseRing(M): ModMPol -> Rng`

Given an R-module M, return the base ring R over which M is defined. Note that one can then call `BaseRing(R)` to obtain the underlying ring S in which the base coefficients of elements R lie.

`Degree(M): ModMPol -> RngIntElt`

Given an R-module M, return the degree of M, which is the k such that the ambient module of M equals $R^k/\langle relations \rangle$. Note that if M is free and ambient, then the degree of M equals the rank of M, but otherwise in general the rank of M may be less than the degree of M (see the function *Rank* below).

`ColumnWeights(M): ModMPol -> [ RngIntElt ]`

`Grading(M): ModMPol -> [ RngIntElt ]`

Given a module M of degree k, return the grading of M, which is a sequence of k integers giving the grading on the columns of M.

`RelationModule(M): ModMPol -> [ ModMPol ]`

Given an R-module M of degree k, return the submodule of the embedded module R^k which is generated by the defining relations of M.

`Relations(M): ModMPol -> [ ModMPol ]`

Given an R-module M of degree k, return the defining relations of M as a sorted sequence of elements of the embedded module R^k.

`RelationMatrix(M): ModMPol -> ModMatRngElt`

Given a module M, return the relation matrix of M, which is the matrix whose rows are the defining relations of M.

`Presentation(M): ModMPol -> [ ModMPol ]`

Given an R-module M, return the presentation module P of M. This is a reduced module isomorphic to M (and such that automatic coercion between M and P is allowed). If M is reduced, then P is identical to M.

`IsGraded(M): ModMPol -> BoolElt`

`IsHomogeneous(M): ModMPol -> BoolElt`

Given a module M, return whether M is graded (or equivalently, homogeneous), w.r.t. the grading of M (given by the weights on the columns of M and the variables of the base ring of M). This is true iff the Gröbner basis of M consists of homogeneous elements only (always true if M is reduced) and the Gröbner basis of the relation module of M consists of homogeneous elements alone. Note that a module of type `ModMPolGrd` is always graded.

119.4.5 Creation of Module Elements

Module elements (internally, multivariate polynomials with columns attached to the monomials) are constructed in general by giving a sequence or vector of elements from the coefficient ring R.

`M ! Q: ModMPol, [ RngElt ] -> ModMPolElt`

Suppose M is an R-module of degree r. Given a sequence $Q = [a_1, \ldots, a_r]$ of ring elements such that the a_i are coercible into R, construct the element of M corresponding to Q.

`M ! v: ModMPol, ModTupRngElt -> ModMPolElt`

Suppose M is an R-module of degree r. Given a vector v from the R-space R^r, construct the element of M corresponding to v.

`M ! 0: ModMPol, RngIntElt -> ModMPolElt`

`Zero(M): ModMPol -> ModMPolElt`

Create the zero element of the module M.

`UnitVector(M, i): ModMPol, RngIntElt -> ModMPolElt`

Suppose M is an R-module of degree r. Given an integer i in the range $[1..r]$, construct the i-th unit vector of M (the vector with 1 in the i-th column and 0 elsewhere) whose parent is the ambient module of M (since it may not lie in M itself). Note that this *not* the same as the function `BasisElement` (below) which depends on the current basis of M.

119.4.6 Element Operations

The following functions allow simple access and operations on module elements. Some of them use the module structure and refer to the column structure of an element; others use the polynomial structure and ignore the column structure.

Access

`Eltseq(f): ModMPolElt -> [ RngMPolElt ]`

Given an element f of the R-module of degree r, return the sequence $[f_1, \ldots, f_r]$ of r elements from R corresponding to f.

`Vector(f): ModMPolElt -> ModTupRngElt`

Given an element f of the module M over R and of degree r, return the element of the R-space of degree r over R corresponding to f.

`f[i]: ModMPolElt, RngIntElt -> RngMPolElt`

Given an element f of the R-module of degree r, together with an integer i in the range $[1..r]$, return the i-th component of f as an element of R.

Arithmetic

The following functions act on elements of R-modules. The operations are similar to those for multivariate polynomials or vectors, whenever meaningful. For the binary operations, the elements must be **compatible**; that is, their parents must have the same ambient module. Note that if quotient relations for M are present, then the result is reduced to the unique normal form modulo the quotient relations, but if the determination of the relations is delayed, then an element may have a non-unique representation, but all the predicates on elements below do not depend on the representation.

`f + g: ModMPolElt, ModMPolElt -> ModMPolElt`

`f - g: ModMPolElt, ModMPolElt -> ModMPolElt`

`- f: ModMPolElt -> ModMPolElt`

`r * f: RngElt, ModMPolElt -> ModMPolElt`

`f * r: ModMPolElt, RngElt -> ModMPolElt`

Basic arithmetic operations. The element r lies in the base ring R.

`f div s: ModMPolElt, RngMPolElt -> ModMPolElt`

Given a scalar ring element s and an element f of the module M, such that s is coercible into R s divides all components of f, return the quotient of f by s.

`SPolynomial(f, g): ModMPolElt, ModMPolElt -> ModMPolElt`

Given elements f and g of the module M such that the leading module monomials of f and g have the same column, return the S-polynomial of f and g. Note that the result is always reduced to the unique normal form modulo the quotient relations of M.

`Normalize(f): ModMPolElt -> ModMPolElt`

Given an element f of the module M, return the normalized form of f (so that the leading module monomial of f is normalized).

`NormalForm(f, S): ModMPolElt, ModMPol -> ModMPolElt`

Given an element f of the module M, together with a compatible module S, return the normal form of f with respect to S. This is unique if the base ring R is not local. In general, S will be a non-ambient embedded module for this to be useful (otherwise any f would already be in S so the result would always be zero).

`Coordinates(M, f): ModMPolElt, ModMPol -> [ RngMPolElt ]`

Given a R-module M and an element f of the R-module S, such that S is compatible with M and f is in M, return the coordinates of f with respect to the Gröbner basis of M (whose components lie in R).

Accessing the Underlying Representation

The following functions access simple properties of module elements which are to do with the underlying representation.

`Coefficients(f): ModMPolElt -> [RngElt]`

`Monomials(f): ModMPolElt -> [ModMPolElt]`

`Terms(f): ModMPolElt -> [ModMPolElt]`

`LeadingCoefficient(f): ModMPolElt -> RngElt`

`LeadingMonomial(f): ModMPolElt -> ModMPolElt`

`LeadingTerm(f): ModMPolElt -> ModMPolElt`

`CoefficientsAndMonomials(f): ModMPolElt -> [RngElt], [ModMPolElt]`

These functions are equivalent to the access functions for multivariate polynomials and access the underlying distributed polynomial representation (with columns added to the monomials); see Section Coefficients, Monomials, and Terms for details.

`Column(f): ModMPolElt -> RngIntElt`

Given a single-term element f of a module M, return the column c of the single monomial-column pair (module monomial) $s[c]$ which f has.

`Degree(f): ModMPolElt -> RngIntElt`

`WeightedDegree(f): ModMPolElt -> RngIntElt`

Given an element f of a module M, return the weighted degree (abbreviated to 'degree' in this chapter) of f, which is the maximum of the weighted degrees of the monomial-column pairs of f. The weighted degree of a monomial-column $s[c]$ is the weighted degree of s (in the base ring R) plus the degree of column c in the grading of M.

`IsHomogeneous(f): ModMPolElt -> BoolElt`

Given an element f of a module M, return whether f is homogeneous; that is, whether the weighted degrees of all the monomial-columns of f are equal. (Note that the grading of M is thus significant.)

Predicates

`IsZero(f): ModMPolElt -> BoolElt`

Given an element f of the module M, return whether f is the zero element of M. Note that if the relations of M are non-zero this operation may be non-trivial (especially if the relations are not yet computed, but they will be automatically computed if needed).

`f eq g: ModMPolElt, ModMPolElt -> BoolElt`

Given elements f and g of the module M, return whether f and g are equal. Note that this may be non-trivial (see the remarks above).

`f lt g: ModMPolElt, ModMPolElt -> BoolElt`

Given elements f and g of the module M, return whether $f < g$ w.r.t. the underlying module monomial order. The operators `le`, `gt`, `ge` are similarly defined.

`f in M: ModMPolElt, ModMPol -> BoolElt`

Given an element f of a module S together with a compatible module M, return whether f is in M.

Example: Embedded Modules over a Polynomial Ring (ex-ce53cf)

We illustrate simple modules over a multivariate polynomial ring. We construct simple ambient embedded modules over $\mathbb{Q}[x, y, z]$. The first module has default weights 0 on its columns, while the second has weights 1, 2, and 3 respectively on its columns.

```
> R<x,y,z> := PolynomialRing(RationalField(), 3, "grevlex");
> M := EModule(R, 3);
> M;
```

(continues on next page)

(continued from previous page)

```
Free Embedded Module R^3
Order: Module TOP: Graded Reverse Lexicographical
> f := M![x, y, z^2];
> g := M![z, y^3, x + 1];
> f;
[x, y, z^2]
> g;
[z, y^3, x + 1]
> f + g;
[x + z, y^3 + y, z^2 + x + 1]
> Terms(f);
[
    [0, 0, z^2],
    [x, 0, 0],
    [0, y, 0]
]
> Degree(f);
2
> [Degree(m): m in Monomials(f)];
[ 2, 1, 1 ]
> LeadingMonomial(f);
[0, 0, z^2]
> M2 := EModule(R, [10, 5, 1]);
Free Embedded Module R^3 with grading [10, 5, 1]
Order: Module TOP with column weights [10, 5, 1]: Graded Reverse Lexicographical
> f := M2![x, y, z^2];
> f;
[x, y, z^2]
> Terms(f);
[
    [x, 0, 0],
    [0, y, 0],
    [0, 0, z^2]
]
> Degree(f);
11
> [Degree(m): m in Monomials(f)];
[ 11, 6, 3 ]
```

Similar operations can be done with reduced modules. There is no difference for the elements.

```
> M := RModule(R, 3);
> M;
Free RModule R^3
> M := GradedModule(R, [10, 5, 1]);
> M;
Free Graded Module R^3 with grading [10, 5, 1]
> Grading(M);
[ 10, 5, 1 ]
> f := M![x, y^6, z^10];
> f;
[x, y^6, z^10]
> IsHomogeneous(f);
true
```

119.5 The Homomorphism Type

Magma has a special type for a homomorphism between two R-modules. The type of such a homomorphism is `ModMPolHom`. In general, functions such as *Morphism* return a homomorphism of type `ModMPolHom`, while the boundary maps of complexes are also of type `ModMPolHom` (see the function *FreeResolution*). A homomorphism $f : M \to N$ is represented by a matrix A. There are two ways in which A can be defined:

(a) A is an **ambient** matrix: in this case, A gives the explicit map on the *ambient* modules of M and N. Thus A is $m \times n$, where $m =$`Degree(M)`, $n =$`Degree(N)`.

(b) A is a **presentation** matrix: in this case, A gives the explicit map on the *presentation* modules of M and N. Thus A is $m \times n$, where $m =$`Degree(Presentation(M))`, $n =$`Degree(Presentation(N))`.

If M and N are reduced (a common case), then they equal their respective presentation modules, so there is no difference between the above two cases (the ambient matrix and the presentation matrix are identical). So the only difference between (a) and (b) occurs when at least one of M and N is a non-ambient (proper) submodule of an embedded module.

When M and N are graded - that is, generated by elements homogeneous with respect to the ambient column weightings and with a relation module that is also generated by homogeneous elements - all homomorphisms as non-graded modules are still allowed. However there are functions to test if a given homomorphism preserves the gradings on the domain and codomain up to a constant degree shift. See `IsHomogeneous` and `Degree` below.

`Homomorphism(M, N, A): ModMPol, ModMPol, Mtrx -> ModMPolHom`

`Presentation: BoolElt`	`Default: true`

Given R-modules M and N and an $m \times n$ matrix A over R, construct the homomorphism $f : M \to N$ (with type `ModMPolHom`) defined by A. By default, A is assumed to be a presentation matrix (see the comments above), in which case m and n must equal the degrees of the presentation modules of M and N, respectively. Alternatively, setting the parameter `Presentation` to `false` specifies that A is an ambient matrix; in this case, m and n must equal the degrees of M and N, respectively.

`Domain(f): ModMPolHom -> ModMPol`

Given a module homomorphism $f : M \to N$, return the domain M.

`Codomain(f): ModMPolHom -> ModMPol`

Given a module homomorphism $f : M \to N$, return the codomain N.

`PresentationMatrix(f): ModMPolHom -> ModMatRngElt`

`Matrix(f): ModMPolHom -> ModMatRngElt`

Given a module homomorphism $f : M \to N$, return the presentation matrix A_P of f as an $m \times n$ matrix corresponding to the presentation modules of M and N, respectively. This presentation matrix is always well-defined and computed, even if f is constructed via an ambient matrix.

`AmbientMatrix(f): ModMPolHom -> ModMatRngElt`

`Matrix(f): ModMPolHom -> ModMatRngElt`

Given a module homomorphism $f : M \to N$, return the ambient matrix A_A of f as an $m \times n$ corresponding to the ambient modules of M and N, respectively. If M and N are reduced (as commonly happens), this will be the same as the presentation matrix above. But if M and N are not reduced and f is constructed via a presentation matrix, then an error may result (since it may be impossible to give a matrix over the base ring R which gives the mapping for the ambient modules).

`f(v): ModMPolHom, RngMPolElt -> RngMPolElt`

`v * f: ModMPolHom, RngMPolElt -> RngMPolElt`

Given a module homomorphism $f : M \to N$ and an element v of M, return the image of v under f, as an element of N.

`f[i]: ModMPolHom, RngIntElt -> RngMPolElt`

Given a module homomorphism $f : M \to N$ and an integer i, return the element of N corresponding to the i-th row of the ambient matrix of f.

`Image(f): ModMPolHom -> ModMPol`

Given a module homomorphism $f : M \to N$, return the image of f as a submodule of N (which will be reduced iff N is).

`Kernel(f): ModMPolHom -> ModMPol`

Given a module homomorphism $f : M \to N$, return the kernel of f as a submodule of M (which will be reduced iff M is).

`Cokernel(f): ModMPolHom -> ModMPol`

Given a module homomorphism $f : M \to N$, return the cokernel of f as a quotient module of N (which will be reduced iff N is).

`IsZero(f): ModMPolHom -> BoolElt`

Given a module homomorphism $f : M \to N$, return whether f is the zero map. Note that f may be the zero map even if the presentation or ambient matrices of f are non-zero.

`IsInjective(f): ModMPolHom -> BoolElt`

Given a module homomorphism $f : M \to N$, return whether f is injective (whether the kernel of f is the zero module).

`IsSurjective(f): ModMPolHom -> BoolElt`

Given a module homomorphism $f : M \to N$, return whether f is surjective (whether the image of f equals N).

`IsBijective(f): ModMPolHom -> BoolElt`

Given a module homomorphism $f : M \to N$, return whether f is bijective (injective and surjective).

`IsGraded(f): ModMPolHom -> BoolElt`

`IsHomogeneous(f): ModMPolHom -> BoolElt`

Given a module homomorphism $f : M \to N$, where M and N are graded modules, return whether f is homogeneous of some degree d; that is, whether for every pure degree element $v \in M$, $f(v) = 0$ or `Degree`$(f(v))$ equals `Degree`$(v) + d$.

`Degree(f): ModMPolHom -> RngIntElt`

Given a module homomorphism $f : M \to N$, return the degree of f, which is the maximum d such that an element of M of degree e is mapped via f to zero or an element of degree $e + d$. If f is homogeneous, then the 'maximum' concept is unnecessary, since the degree will be consistent for all elements of M (see the previous function).

Example: Inclusion Homomorphism of Submodules (ex-ce742b)

We illustrate some homomorphism functionality by looking at the explicit inclusion homomorphism between two submodules of a rank 3 free module over $\mathbb{Q}[x, y]$. We define this in non-presentational form by the identity matrix. Then we can retrieve the corresponding defining matrix for the map between the internal presentations of the two submodules. The two submodules being graded submodules, we check that the inclusion is indeed homogeneous of degree 0 (as it must be, obviously preserving degrees of elements).

```
> R<x,y> := PolynomialRing(RationalField(), 2, "grevlex");
> F := EModule(R, 3);
> // get a submodule M1 generated by a single non-zero element of F
> M1 := sub<F|[x^2,y^2,x*y]>;
> // and a second submodule M2 containing M1
> M2 := sub<F|[x,0,y],[0,y,0]>;
> incl_hm :=  Homomorphism(M1,M2,IdentityMatrix(R,3) :
>               Presentation := false);
```

(continues on next page)

(continued from previous page)

```
> incl_hm;
Module homomorphism (3 by 3)
Ambient matrix:
[1 0 0]
[0 1 0]
[0 0 1]
```

Now the corresponding presentation matrix of the inclusion map is the obvious one coming from the expression of the natural generator of M1 in terms of the two natural generators of M2

```
>  PresentationMatrix(incl_hm);
[y x]
> // check homogeneity of incl_hm
> IsHomogeneous(incl_hm);
true
> Degree(incl_hm);
0
```

119.6 Submodules and Quotient Modules

The following functions allow the construction of submodules and quotient modules and access to essential properties.

119.6.1 Creation

sub<M | L>: ModMPol, List -> ModMPol

Given a module M over a ring R, return the submodule of M (with the same quotient relations as M) generated by the elements of M specified by the list L. Each term of the list L must be an expression defining an object of one of the following types:

(a) An element of M;

(b) A set or sequence of elements of M;

(c) A submodule of M;

(d) A set or sequence of submodules of M. A morphism is stored from the resulting submodule S into M, such that $S.i$ is mapped to the i-th generator given in the above list.

quo<M | L>: ModMPol, List -> ModMPol

Given a module M over a ring R, return the quotient module of M by the elements of M specified by the list L. Each term of the list L must be an expression defining an object of one of the following types:

(a) An element of M;

(b) A set or sequence of elements of M;

(c) A submodule of M;

(d) A set or sequence of submodules of M. A morphism is stored from M onto the resulting quotient module Q.

Morphism(M, N): ModMPol, ModMPol -> ModMPolHom

Given modules M and N, related by a chain of stored sub and quo morphisms as mentioned above, returns the resulting morphism matrix map from M to N. If no known sub/quo relationship chain exists between M and N then an error is returned.

Submodule(I): RngMPol -> ModMPol

Given an ideal I of a polynomial ring R, return the submodule of R^1 generated by I.

QuotientModule(I): RngMPol -> ModMPol

Given an ideal I of a polynomial ring R, return the quotient module R^1/I.

GradedModule(I): RngMPol -> ModMPol

Given a homogeneous ideal I of a ring R, return the graded quotient module R^1/I.

119.6.2 Module Bases

The following functions allow one to manipulate the bases of modules. Note that a Gröbner basis for a module will be automatically generated when necessary; the **Groebner** procedure just allows explicit immediate construction of the Gröbner basis.

Basis(M): ModMPol -> RngMPolElt

Given a module M, return the current basis (whether it has been converted to a Gröbner basis or not) of M.

BasisElement(M, i): ModMPol, RngIntElt -> RngMPolElt

Given a module M together with an integer i, return the i-th element of the current basis of M. Note that this is *not* the same as $M.i$.

BasisMatrix(M): ModMPol -> ModMatRngElt

Given a module M, return the basis matrix of M, which is a k by r matrix over R, where k is the length of the basis of M and r is the degree of M.

Groebner(M): ModMPol

(Procedure.) Explicitly force a Gröbner basis for the module M to be constructed.

Example: Submodules and Quotients of an Embedded Module (ex-df76cb)

We construct simple submodules and quotient modules of an embedded module and consider some of their basic properties.

```
> R<x, y, z> := PolynomialRing(RationalField(), 3);
> M := EModule(R, 3);
> S := sub<M | [1, x, x^2+y], [z, y, x*y^2+1],
>              [y, z, x+z]>;
> Groebner(S);
> S;
Embedded Submodule of R^3
Order: Module TOP: Lexicographical
Groebner basis:
[ -x*z + y^2 + y, x*y^2 - x*y + z,         y^3 + z],
[  x*y - y*z - 1,   x*z - x - z^2,        -y - z^2],
[              y,               z,           x + z],
[        y^3 - z,       y^2*z - y,       y^2*z - 1]
> a := M ! [y, z, x+z];
> a;
[y, z, x + z]
> a in S;
true
> BasisElement(S, 1);
[-x*z + y^2 + y, x*y^2 - x*y + z, y^3 + z]
> Q := quo<M | [x, y, z]>;
> Q;
Embedded Module R^3/<relations>
Order: Module TOP: Lexicographical
Relations (Groebner basis):
```

(continues on next page)

(continued from previous page)

```
[x, y, z]
> a := Q![x, y, 0];
> b := Q![0, 0, z];
> a;
[0, 0, -z]
> b;
[0, 0, z]
> a+b;
[0, 0, 0]
> Q ! [x,y,z];
[0, 0, 0]
> QQ := quo<Q | [x^2, 0, y+z]>;
> QQ;
Embedded Module R^3/<relations>
Order: Module TOP: Lexicographical
Relations (Groebner basis):
[          0,          x*y, x*z - y - z],
[          x,            y,           z]
> SL := Localization(S);
> SL;
Embedded Submodule of R^3 (local)
Order: Module TOP: Local Lexicographical
Basis:
[        1,         x,   x^2 + y],
[        z,         y, 1 + x*y^2],
[        y,         z,     x + z]
```

Example: Submodules and Quotients of a Reduced Module (ex-c177b3)

We construct simple submodules and quotient modules of a reduced module and consider some of their basic properties.

```
> R<x,y,z> := PolynomialRing(RationalField(), 3);
> M := RModule(R, 3);
> S := sub<M | [1, x, x^2+y], [z, y, x*y^2+1]>;
> M;
Free Reduced Module R^3
> S;
Reduced Module R^2/<relations>
> Morphism(S, M);
Module homomorphism (2 by 3)
Ambient matrix:
[        1         x   x^2 + y]
[        z         y x*y^2 + 1]
> RelationMatrix(S);
Matrix with 0 rows and 2 columns
> S;
Free Reduced Module R^2
> M.1;
[1, 0, 0]
> M!S.1;
[1, x, x^2 + y]
> M!S.2;
[z, y, x*y^2 + 1]
> M.1 in S;
false
```

(continues on next page)

(continued from previous page)

```
> Q := quo<M | [1, x^2, y]>;
> Q;
Free Reduced Module R^2
> RelationMatrix(Q);
Matrix with 0 rows and 2 columns
> Morphism(M, Q);
Module homomorphism (3 by 2)
Ambient matrix:
[-x^2   -y]
[   1    0]
[   0    1]
> Morphism(S, Q);
Module homomorphism (2 by 2)
Ambient matrix:
[       -x^2 + x             x^2]
[     -x^2*z + y x*y^2 - y*z + 1]
> Q!M.1;
[-x^2, -y]
> M!Q.1;
[0, 1, 0]
> M!Q.2;
[0, 0, 1]
> Q!M!Q.2;
[0, 1]
```

119.7 Basic Module Constructions

The following functions give some fundamental basic constructions with modules.

`M + N: ModMPol, ModMPol -> ModMPol`

Given compatible modules M and N (ie, embedded in the same ambient module), return the sum of M and N; that is, the submodule of the ambient generated by M and N.

`M meet N: ModMPol, ModMPol -> ModMPol`

Given compatible modules M and N (ie, embedded in the same ambient module), return the intersection of M and N in the ambient. This uses the standard algorithm for intersecting two modules of a free module (see Section 2.8.3 of [GP02]). If the ambient is the quotient of a free module F by non-trivial relations, the intersection performed is effectively that of the inverse images of M and N in F.

`f * M: ModMPol, RngElt -> ModMPol`

`M * f: ModMPol, RngElt -> ModMPol`

Given an R-module M and an element $f \in R$, return the submodule of M generated by $\{f \cdot v : v \in M\}$ or $\{v \cdot f : v \in M\}$, respectively.

`I * M: RngMPol, ModMPol -> ModMPol`

`M * I: ModMPol, RngMPol -> ModMPol`

Given an R-module M and an ideal I of R, return the submodule of M generated by $\{f \cdot v : f \in I, v \in M\}$ or $\{v \cdot f : f \in I, v \in M\}$, respectively.

`M / N: ModMPol, ModMPol -> ModMPol`

Given compatible modules M and N (ie, embedded in the same ambient module), return the quotient module $M/(M \cap N)$. This has the same effect as using the `quo` constructor.

`DirectSum(M, N): ModMPol, ModMPol -> ModMPol, [ModMPolHom], [ModMPolHom]`

Given R-modules M and N, return the direct sum $D = M \oplus N$ and two sequences of corresponding homomorphisms giving the injections into and projections from D, respectively.

`DirectSum(S): [ModMPol] -> ModMPol, [ModMPolHom], [ModMPolHom]`

`DirectSum(S): [* ModMPol *] -> ModMPol, [ModMPolHom], [ModMPolHom]`

A sequence or list L of R-modules, return their direct sum D and two sequences of corresponding homomorphisms giving the injections into and projections from D, respectively.

`Twist(M, d): ModMPol, RngIntElt -> [ ModMPolElt ], ModMPolHom`

Given a graded module M, and an integer d, return the Serre twist $M(d)$ and an isomorphism $f : M \to M(d)$. The twisted module is simply an isomorphic copy of M, but with the grading twisted by d (so d is subtracted from each weight of M). f has degree $-d$.

119.8 Predicates

`IsZero(M): ModMPol -> ModMPol`

Given a module M, return whether M is the zero module.

`M subset N: ModMPol, ModMPol -> BoolElt`

Given compatible modules M and N (ie, embedded in the same ambient module), return whether M is a submodule of N. This will generally involve module Gröbner basis and normal form computations to check that the generators of M lie in N.

`M eq N: ModMPol, ModMPol -> BoolElt`

Given compatible modules M and N (ie, embedded in the same ambient module), return whether M equals N. The function checks that appropriate module Gröbner bases of M and N are equal.

`IsFree(M): ModMPol -> BoolElt`

Given an R-module M, return whether M is free. M is free iff M is isomorphic to the module R^k for some k. Such a k need not equal the degree of M but will equal the rank of M (as defined in the next section) if M is free. The function checks whether a minimised presentation of M has trivial relations or not.

119.9 Module Operations

The following functions perform some fundamental module operations.

`MinimalBasis(M): ModMPol -> [ ModMPolElt ]`

Given an R-module M, return a minimal basis B of M. If M is graded, or if R is a local ring, then the cardinality of B (the rank) is guaranteed to be unique (so is the absolutely minimal number of elements needed to generate M).

Otherwise the cardinality of B is not unique: B will only satisfy the rule that the i-th element of B is not in the submodule generated by elements 1 to $i-1$ of B.

In the graded case or local cases, a minimal basis is computed in the usual way starting from any basis B consisting of homogeneous elements. B gives a particular presentation whose relation matrix R consists of homogeneous polynomials. If R contains a non-zero constant term (or more generally a unit in the local case), an element of B can be eliminated and R recalculated. This can be continued until all non-zero terms of R have positive degree.

`MinimalBasis(S): [ ModMPolElt ] -> [ ModMPolElt ]`

`MinimalBasis(S): { ModMPolElt } -> [ ModMPolElt ]`

Given a set or sequence S of homogeneous module elements from a module M, return a minimal basis of the submodule of M generated by S.

`Rank(M): ModMPol -> RngIntElt`

Given an R-module M, return the rank of M. This is simply defined to be the cardinality of the minimal basis of M, returned by the function `MinimalBasis`. Thus if M is graded, or if R is a local ring, then the rank is guaranteed to be unique (and is the absolutely minimal number of elements needed to generate M). Otherwise the result is not an invariant of M, but simply reflects the minimum as found by the `MinimalBasis` algorithm.

`ColonModule(M, J): ModMPol, RngMPol -> ModMPol`

Given an R-module M and an ideal J of R, return the colon module $M : J$ which is the submodule of the ambient module A of M consisting of all $f \in A$ such that $f \cdot g \in M$ for all $g \in J$. When J is generated by a single element, this easily reduces to a syzygy computation in A and in the general case, we intersect the colon modules for a set of generators of J.

`ColonIdeal(M, N): ModMPol, ModMPol -> RngMPol`

Given an R-modules M and N which are both submodules of a common supermodule, return the colon ideal $M : N$, which is the ideal of R consisting of all $f \in R$ such that $f \cdot N \subset M$. The algorithm used is as described in section 2.8.4 of [GP02].

`Annihilator(M): ModMPol -> RngMPol`

Given an R-module M, return the annihilator ideal of M. This is the ideal I of R consisting of all $f \in R$ such that $f \cdot M = 0$ (which can be seen to equal the ideal $0_M : M$, where 0_M is the zero submodule of M, so is a special case of `ColonIdeal`).

`FittingIdeal(M, i): ModMPol, RngIntElt -> RngMPol`

Given an R-module M of degree r and an integer $i \geq 0$, return the i-th Fitting ideal of M, which is the ideal of R generated by the $(r-i)$-th minors of the presentation matrix of M, where r is the degree of M. See [CLO98, p.229] or [Eis95, Sec. 20.2].

`FittingIdeals(M): ModMPol -> RngMPol`

Given an R-module M of degree r, return the Fitting ideals (for from 0 to r) as a sequence of ideals of R.

`SyzygyModule(M): ModMPol -> [ ModMPolElt ]`

Given a module M, return the syzygy module S of M. If the basis B of M has length k, the syzygy module S has degree k and elements of S express a syzygy amongst the k elements of the basis B. Note that the degree of the resulting module thus depends on the current basis of M.

`MinimalSyzygyModule(M): ModMPol -> [ ModMPolElt ]`

Given a homogeneous module M, return the syzygy module S of the minimal basis of M. If the minimal basis B of M has length k, the syzygy module S has degree k and elements of S express a syzygy amongst the k elements of the minimal basis B.

`SyzygyModule(Q): [ RngMPolElt ] -> ModTupRng`

Given a sequence Q of polynomials from a multivariate polynomial ring P, return the module of syzygies of Q. This is a module over P of degree k, where k is the length of Q, consisting of all vectors v such that the sum of $v[i] * Q[i]$ for $i = 1, \ldots k$ is zero.

Example: Rank of a Module and Its Localization `(ex-e30655)`

In this example we note that a certain module M has rank 3 (equal to its degree 3), since no generator is redundant. If we move to the localization of M, then $(1 + x - z)$ becomes a unit, so the first generator becomes redundant.

```
> R<x,y,z> := PolynomialRing(RationalField(), 3, "grevlex");
> F := RModule(R, 3);
> M := quo<F | [x + 1, y, z], [z, y, 0]>;
> M;
Reduced Module R^3/<relations>
Relations:
[x + 1,     y,     z],
[    z,     y,     0]
> Degree(M);
3
> Rank(M);
3
> ML := Localization(M);
> ML;
```

(continues on next page)

(continued from previous page)

```
Reduced Module R^3/<relations> (local)
Relations:
[1 + x - z,         0,         z],
[        z,         y,         0]
> Rank(ML);
2
> MinimalBasis(ML);
[
    [0, 1, 0],
    [0, 0, 1]
]
```

119.10 Changing Ring

The `ChangeRing` function enables the changing of the polynomial ring over which a module is defined.

`ChangeRing(M, S): RngMPol, Rng -> RngMPol`

Given an R-module M, where R is a polynomial ring, and another polynomial ring S, construct the S-module N obtained by coercing the coefficients of the elements of the basis and relations of M into S. It is necessary that all elements of the old coefficient ring R can be automatically coerced into the new coefficient ring S. Note that S itself must be polynomial ring having the same rank as R, so S does not specify the new ring for the underlying coefficients (one can use *ChangeRing* for polynomial rings to do that first).

119.11 Hilbert Series

The following functions compute the Hilbert series information of graded or (homogeneous) modules. This depends on the column weights, just as in graded polynomial rings.

`HilbertSeries(M): ModMPol -> FldFunElt`

Given a graded R-module M, return the Hilbert series $H_M(t)$ of M (as an element of the univariate function field over the ring of integers). The i-th coefficient of the series gives the vector-space dimension of the degree-i graded piece of M. The algorithm implemented is that given in [BS92]. Note that if I is an ideal of the ring R, then the corresponding function for ideals *HilbertSeries* applied to I gives the Hilbert series of the affine algebra (quotient) R/I, so this is equivalent to `HilbertSeries(QuotientModule(I))`.

`HilbertSeries(M, p): ModMPol, RngIntElt -> RngSerLaurElt`

Given a graded R-module M, return the Hilbert series $H_M(t)$ of M (as a Laurent series to precision p). (A Laurent series is required in general, since negative powers may occur when there are negative values in the grading of M.)

`HilbertDenominator(M): ModMPol -> RngUPol`

Given a graded R-module M, return the unreduced Hilbert denominator D of the Hilbert series $H_M(t)$ of M (as a univariate polynomial over the ring of integers). The denominator D equals *HilbertDenominator*`(R)` which is simply $\prod_{i=1}^{n}(1-t^{w_i})$, where n is the rank of R and w_i is the weight of the i-th variable (1 by default).

`HilbertNumerator(M): ModMPol -> RngUPolElt, RngIntElt`

Given a graded R-module M, return the unreduced Hilbert numerator N of the Hilbert series $H_M(t)$ of M (as a univariate polynomial over the ring of integers) and a valuation shift s. The numerator N equals $D \times t^s \times H_M(t)$, where D is the unreduced Hilbert denominator above. Computing with the unreduced numerator is often more convenient. Note that s will only be non-zero when M has negative weights in its grading.

`HilbertPolynomial(I): ModMPol -> RngUPolElt, RngIntElt`

Given a graded R-module M, return the Hilbert polynomial $H(d)$ of M as an element of the univariate polynomial ring $\mathbb{Q}[d]$, together with the index of regularity of M (the minimal integer $k \geq 0$ such that $H(d)$ agrees with the Hilbert function of M at d for all $d \geq k$).

Example: Hilbert Series of a Quotient Module `(ex-ffdbad)`

We apply the Hilbert series functions to a simple quotient module.

```
> R<x,y,z> := PolynomialRing(RationalField(), 3);
> F := GradedModule(R, 3);
> M := quo<F | [x,0,0], [0,y^2,0]>;
> M;
Graded Module R^3/<relations>
Relations:
[  x,   0,   0],
[  0, y^2,   0]
> HilbertSeries(M);
(t^2 + t - 3)/(t^3 - 3*t^2 + 3*t - 1)
> HilbertSeries(M, 10);
3 + 8*s + 14*s^2 + 21*s^3 + 29*s^4 + 38*s^5 + 48*s^6 + 59*s^7 + 71*s^8 + 84*s^9
    + O(s^10)
> HilbertNumerator(M);
-x^2 - x + 3
0
> HilbertDenominator(M);
-x^3 + 3*x^2 - 3*x + 1
> HilbertPolynomial(M);
1/2*x^2 + 9/2*x + 3
0
> [Evaluate(HilbertPolynomial(F), i): i in [0..10]];
[ 3, 9, 18, 30, 45, 63, 84, 108, 135, 165, 198 ]
```

If the module has negative weights, then denominator may include extra powers of t, so the shift for the numerator will be non-zero.

```
> F := GradedModule(R, [-1]);
> F;
Free Graded Module R^1 with grading [-1]
> HilbertSeries(F);
-1/(t^4 - 3*t^3 + 3*t^2 - t)
> HilbertSeries(F, 10);
s^-1 + 3 + 6*s + 10*s^2 + 15*s^3 + 21*s^4 + 28*s^5 + 36*s^6 + 45*s^7 + O(s^8)
> HilbertNumerator(F);
1
1
> HilbertDenominator(F);
-x^3 + 3*x^2 - 3*x + 1
> HilbertPolynomial(F);
1/2*x^2 + 5/2*x + 3
-1
> [Evaluate(HilbertPolynomial(F), i): i in [-1..10]];
[ 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78 ]
```

119.12 Free Resolutions

The functions in this section deal with free resolutions and associated properties. Free resolutions are returned as chain complexes (see Chapter Chain Complexes).

119.12.1 Constructing Free Resolutions

`FreeResolution(M): ModMPol -> ModCpx, ModMPolHom`

```
Minimal   : BoolElt                    Default: true
Limit     : RngIntElt                  Default: 0
Homogenize: BoolElt                    Default: true
Al        : MonStgElt                  Default: "LaScala"
```

Given an R-module M, return a free resolution M as a complex C, and a comparison homomorphism $f : C_0 \to M$ (where C_0 is the term of C of degree 0).

By default, the free resolution will be **minimal**. Setting the parameter `Minimal` to `false` will construct a non-minimal resolution (which is constructed via a sequence of successive syzygy modules, with no minimization).

Magma has two algorithms for computing resolutions:

(1) The **La Scala** (LS) [SS98] algorithm, which works with a homogeneous module. The Magma implementation involves an extension of this algorithm which uses techniques from the Faugère F_4 [Fau99] algorithm to compute many normal forms together in a block.

(2) The **Iterative** algorithm, which simply computes successive syzygy modules progressively (minimizing as it goes if and only a minimal resolution is desired).

By default the LS algorithm is used if M is homogeneous and the coefficient ring of R is a finite field or the rational field, since this tends to be faster in general. But for some inputs the iterative algorithm may be significantly faster, particularly for some modules over the rationals. So one may set the parameter `Al` to `"Iterative"` to select the iterative algorithm. Uniqueness of the terms in the resolution is as follows.

(1) If M is homogeneous or defined over a local ring R, then the resulting complex C is guaranteed to be minimal, so the ranks of the terms in C and the associated Betti numbers will be unique.

(2) If M is non-homogeneous and over a global ring R, then the boundary maps of C will not have any entries which are units, but C cannot be guaranteed to be an absolutely minimal free resolution, so the ranks of the terms and the associated Betti numbers will not be unique in general. Also, Magma may choose to compute C by computing the free resolution C_H of a homogenization M_H of M, and then specializing C_H to yield C, since this method is usually faster (since the LS algorithm can then be used). One may set the parameter `Homogenize` to `true` or `false` to force Magma to use this homogenization technique or not.

If the parameter `Limit` is set to a non-zero value l, then at most l terms (plus the term corresponding to the free module) are computed. If R is an affine algebra or exterior algebra of rank n, then by default the limit is set to n, since the resolution is not finite in general.

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

(Procedure.) Change the verbose printing level for the free resolution algorithm and related functions to be v.

Example: Minimal Free Resolution of a Module `(ex-2a8a2e)`

We construct the module $M = R^1/I$ where I is a homogeneous ideal of $\mathbb{Q}[x, y, z, t]$ and then construct a minimal free resolution of M and note simple properties of this.

```
> R<x,y,z,t> := PolynomialRing(RationalField(), 4, "grevlex");
> B := [
>     -x^2 + y*t, -y*z + x*t, x*z - t^2,
>     x*y - t^2, -y*z + x*t, -x^2 + z*t
> ];
> M := GradedModule(Ideal(B));
> M;
Graded Module R^1/<relations>
Relations:
[-x^2 + y*t],
[-y*z + x*t],
[ x*z - t^2],
[ x*y - t^2],
[-y*z + x*t],
[-x^2 + z*t]
> C := FreeResolution(M);
> C;
Chain complex with terms of degree 4 down to -1
Dimensions of terms: 0 1 5 5 1 0
> Terms(C);
[
    Free Graded Module R^0,
    Free Graded Module R^1 with grading [5],
    Free Graded Module R^5 with grading [3, 3, 3, 3, 3],
    Free Graded Module R^5 with grading [2, 2, 2, 2, 2],
    Free Graded Module R^1,
    Free Graded Module R^0
]
> B := BoundaryMaps(C);
> B;
[*
    Graded module homomorphism (0 by 1),
    Graded module homomorphism (1 by 5) of degree 0
    Ambient matrix:
    [ x*z - t^2  x^2 - z*t -y*t + z*t  y*z - x*t -x*y + t^2],
    Graded module homomorphism (5 by 5) of degree 0
    Ambient matrix:
    [-y  x  0 -t  0]
    [ 0 -z  y  0  t]
    [ t -z  0  x  0]
    [ 0 -t  t  0  x]
    [-z  0  x -t  z],
    Graded module homomorphism (5 by 1) of degree 0
    Ambient matrix:
    [x^2 - z*t]
    [x*y - t^2]
    [x*z - t^2]
    [y*z - x*t]
    [y*t - z*t],
    Graded module homomorphism (1 by 0)
*]
> B[2]*B[3];
Module homomorphism (1 by 5)
Ambient matrix:
[0 0 0 0 0]
> B[3]*B[4];
Module homomorphism (5 by 1)
```

(continues on next page)

(continued from previous page)

```
Ambient matrix:
[0]
[0]
[0]
[0]
[0]
> Image(B[3]) eq Kernel(B[4]);
true
```

Example: Free Resolution of Six Points (ex-999f8c)

Following [CLO98, p.248], we compute the ideal I of $\mathbb{Q}[x, y]$ whose affine variety is a certain list of 6 pairs.

```
> R<x,y> := PolynomialRing(RationalField(), 2, "grevlex");
> L := [<0, 0>, <1, 0>, <0, 1>, <2, 1>, <1, 2>, <3, 3>];
> I := Ideal(L, R);
> I;
Ideal of Polynomial ring of rank 2 over Rational Field
Graded Reverse Lexicographical Order
Variables: x, y
Inhomogeneous, Dimension 0
Groebner basis:
[
    x^3 - 5*x^2 + 2*x*y - 2*y^2 + 4*x + 2*y,
    x^2*y - 5*x^2 + 3*x*y - 4*y^2 + 5*x + 4*y,
    x*y^2 - 4*x^2 + 3*x*y - 5*y^2 + 4*x + 5*y,
    y^3 - 2*x^2 + 2*x*y - 5*y^2 + 2*x + 4*y
]
```

I is not homogeneous, and we compute a non-minimal free resolution of the module R/I.

```
> M := QuotientModule(I);
> M;
Reduced Module R^1/<relations>
Relations:
[  x^3 - 5*x^2 + 2*x*y - 2*y^2 + 4*x + 2*y],
[x^2*y - 5*x^2 + 3*x*y - 4*y^2 + 5*x + 4*y],
[x*y^2 - 4*x^2 + 3*x*y - 5*y^2 + 4*x + 5*y],
[  y^3 - 2*x^2 + 2*x*y - 5*y^2 + 2*x + 4*y]
> C := FreeResolution(M: Minimal := false);
> C;
Chain complex with terms of degree 3 down to -1
Dimensions of terms: 0 3 4 1 0
> B := BoundaryMaps(C);
> B;
[*
    Module homomorphism (0 by 3),
    Module homomorphism (3 by 4)
    Ambient matrix:
    [-y + 5  x - 8      6     -2]
    [     4 -y - 8  x + 8     -4]
    [     2     -6 -y + 8  x - 5],
    Module homomorphism (4 by 1)
    Ambient matrix:
    [  x^3 - 5*x^2 + 2*x*y - 2*y^2 + 4*x + 2*y]
    [x^2*y - 5*x^2 + 3*x*y - 4*y^2 + 5*x + 4*y]
```

(continues on next page)

(continued from previous page)

```
    [x*y^2 - 4*x^2 + 3*x*y - 5*y^2 + 4*x + 5*y]
    [  y^3 - 2*x^2 + 2*x*y - 5*y^2 + 2*x + 4*y],
    Module homomorphism (1 by 0)
*]
> IsZero(B[2]*B[3]);
true
```

As noted in [CLO98], the 3 by 3 minors of the boundary map from R^3 to R^4 generate the ideal I again, and this is due to the Hilbert-Burch Theorem.

```
> U := Minors(Matrix(B[2]), 3);
> U;
[
    y^3 - 2*x^2 + 2*x*y - 5*y^2 + 2*x + 4*y,
    x*y^2 - 4*x^2 + 3*x*y - 5*y^2 + 4*x + 5*y,
    x^2*y - 5*x^2 + 3*x*y - 4*y^2 + 5*x + 4*y,
    x^3 - 5*x^2 + 2*x*y - 2*y^2 + 4*x + 2*y
]
> Ideal(U) eq I;
true
```

119.12.2 Betti Numbers and Related Invariants

Each of the functions in this section compute numerical properties of a free resolution of a module M. Each function takes the same parameters as the function *FreeResolution* (not repeated here), thus allowing control of the construction of the underlying resolution. In particular, by default the **minimal** free resolution of M is used (so the Betti numbers correspond to that), so the relevant invariant is guaranteed to be unique if M is graded or over a local ring R. Otherwise, one may set the parameter `Minimal` to `false` to give the Betti numbers for a non-minimal resolution. **Note:** If M is graded and the LS algorithm is used (which will be the case by default), then computing any of the invariants to do with Betti numbers in this section may be quicker than computing the full resolution (since minimization of the actual resolution is needed for the latter). Thus it is preferable just to use one of the following functions instead of *FreeResolution* if only the numerical invariants are desired.

`BettiNumbers(M): ModMPol -> [ RngIntElt ]`

Given a module M, return the Betti numbers of M, which is simply the sequence of integers consisting of the degrees of the non-zero terms of the free resolution of M. See the discussion above concerning the parameters. Since the underlying resolution is minimal by default, if M is graded or over a local ring, then the result is unique.

`BettiNumber(M, i, j): ModMPol, RngIntElt, RngIntElt -> RngIntElt`

Given a module M and integers $i, j \geq 0$, return the graded Betti number $\beta_{i,j}$ of M as an integer. This is the number of generators of degree j in the i-th term F_i of the free resolution of M.

`MaximumBettiDegree(M, i): ModMPol, RngIntElt -> RngIntElt`

Given a module M and an integer $i \geq 0$, return the maximum degree of the generators in the i-th term of the free resolution of M. Equivalently, this is the maximum j such that `BettiNumber(M, i, j)` is non-zero.

`BettiTable(M): ModMPol -> [[ RngIntElt ]], RngIntElt`

Given a module M, return the Betti table of M as a sequence S of sequences of integers, and a shift s. This is designed so that if M is non-zero, then `S[1, 1]` is always non-zero and `S[i, j]` equals `BettiNumber(M, i, j - i + s)`. (So the degrees are shifted by s.)

`Regularity(M): ModMPol -> RngInt`

Given an R-module M which is either graded or over a local ring, return the Castelnuovo-Mumford regularity. This is the least r such that in a minimal free resolution of M, the maximum of the

degrees of the generators of the i-th term F_i is at most $i + r$. A simple consequence of this is that M is generated by elements of degree at most r. See [Eis95, Sec. 20.5] or [DL06, p. 167].

`HomologicalDimension(M): ModMPol -> RngInt`

Given a module M, return the homological dimension of M. This is just the length of a minimal free resolution of M (the number of non-zero boundary maps).

Example: Koszul Complex `(ex-289091)`

For an integer n, we can construct a Koszul complex as the free resolution of R/I, where I is the ideal of $R = K[x_1, \ldots, x_n]$ generated by the n variables.

```
> Q := RationalField();
> n := 3;
> R<[x]> := PolynomialRing(Q, n);
> I := Ideal([R.i: i in [1 .. n]]);
> M := QuotientModule(I);
> M;
Graded Module R^1/<relations>
Relations:
[x[1]],
[x[2]],
[x[3]]
> C := FreeResolution(M);
> C;
Chain complex with terms of degree 4 down to -1
Dimensions of terms: 0 1 3 3 1 0
> BoundaryMaps(C);
[*
    Module homomorphism (0 by 1),
    Module homomorphism (1 by 3)
    Ambient matrix:
    [ x[3] -x[2]  x[1]],
    Module homomorphism (3 by 3)
    Ambient matrix:
    [-x[2]  x[1]     0]
    [-x[3]     0  x[1]]
    [    0 -x[3]  x[2]],
    Module homomorphism (3 by 1)
    Ambient matrix:
    [x[1]]
    [x[2]]
    [x[3]],
    Module homomorphism (1 by 0)
*]
```

In general, the i-th Betti number is $\binom{n}{i}$. We can see this for $n = 10$. Each boundary map consists of linear relations alone, so the regularity is zero.

```
> n := 10;
> R<[x]> := PolynomialRing(Q, n);
> I := Ideal([R.i: i in [1 .. n]]);
> M := QuotientModule(I);
> time C := FreeResolution(M);
Time: 0.060
> C;
Chain complex with terms of degree 11 down to -1
Dimensions of terms: 0 1 10 45 120 210 252 210 120 45 10 1 0
> Terms(C);
```

(continues on next page)

(continued from previous page)

```
[
    Free Graded Module R^0,
    Free Graded Module R^1 with grading [10],
    Free Graded Module R^10 with grading [9, 9, 9, 9, 9, 9, 9, 9, 9, 9],
    Free Graded Module R^45 with grading [8^^45],
    Free Graded Module R^120 with grading [7^^120],
    Free Graded Module R^210 with grading [6^^210],
    Free Graded Module R^252 with grading [5^^252],
    Free Graded Module R^210 with grading [4^^210],
    Free Graded Module R^120 with grading [3^^120],
    Free Graded Module R^45 with grading [2^^45],
    Free Graded Module R^10 with grading [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    Free Graded Module R^1,
    Free Graded Module R^0
]
> B := BoundaryMaps(C);
> B: Minimal;
[*
    Graded module homomorphism (0 by 1),
    Graded module homomorphism (1 by 10) of degree 0,
    Graded module homomorphism (10 by 45) of degree 0,
    Graded module homomorphism (45 by 120) of degree 0,
    Graded module homomorphism (120 by 210) of degree 0,
    Graded module homomorphism (210 by 252) of degree 0,
    Graded module homomorphism (252 by 210) of degree 0,
    Graded module homomorphism (210 by 120) of degree 0,
    Graded module homomorphism (120 by 45) of degree 0,
    Graded module homomorphism (45 by 10) of degree 0,
    Graded module homomorphism (10 by 1) of degree 0,
    Graded module homomorphism (1 by 0)
*]
> [Binomial(n, i): i in [0 .. n]];
[ 1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1 ]
> BettiTable(M);
[
    [ 1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1 ]
]
> $1 eq [[Binomial(n, i): i in [0 .. n]]];
true
> Regularity(M);
0
```

Example: Resolution over an Exterior Algebra (ex-7b4771)

We can construct the same type of ideal and module as in the last example for $n = 3$, but over an exterior algebra. The free resolution is infinite here, but we can construct the resolution partially (by default, a bound is set on the number of terms). In this general construction, the i-th Betti number will be $\binom{i}{n-1}$.

```
> Q := RationalField();
> n := 3;
> R<[x]> := ExteriorAlgebra(Q, n);
> I := Ideal([R.i: i in [1 .. n]]);
> M := QuotientModule(I);
> M;
Reduced Module R^1/<relations>
```

(continues on next page)

(continued from previous page)

```
Relations:
[x[1]],
[x[2]],
[x[3]]
> BettiNumbers(M);
[ 1, 3, 6, 10, 15 ]
> [Binomial(i + n - 1, n - 1): i in [0..4]];
[ 1, 3, 6, 10, 15 ]
> C := FreeResolution(M);
> C;
Chain complex with terms of degree 5 down to -1
Dimensions of terms: 0 15 10 6 3 1 0
> BoundaryMaps(C);
[*
    Graded module homomorphism (0 by 15),
    Graded module homomorphism (15 by 10) of degree 0
    Ambient matrix:
    [x[3]    0    0    0    0    0    0    0    0    0]
    [   0 x[3]    0 x[2]    0    0    0    0    0    0]
    [   0    0 x[2]    0    0    0    0    0    0    0]
    [   0 x[2] x[3]    0    0    0    0    0    0    0]
    [x[2]    0    0 x[3]    0    0    0    0    0    0]
    [   0    0    0    0 x[3]    0    0    0    0 x[1]]
    [   0    0    0    0    0 x[2]    0    0 x[1]    0]
    [   0    0    0    0 x[2] x[3]    0 x[1]    0    0]
    [   0    0    0    0    0    0 x[1]    0    0    0]
    [   0    0    0    0    0 x[1] x[2]    0    0    0]
    [   0    0    0    0 x[1]    0 x[3]    0    0    0]
    [   0    0    0 x[1]    0    0    0 x[3]    0 x[2]]
    [   0    0 x[1]    0    0    0    0    0 x[2]    0]
    [   0 x[1]    0    0    0    0    0 x[2] x[3]    0]
    [x[1]    0    0    0    0    0    0    0    0 x[3]],
    Graded module homomorphism (10 by 6) of degree 0
    Ambient matrix:
    [x[3]    0    0    0    0    0]
    [   0 x[3] x[2]    0    0    0]
    [   0 x[2]    0    0    0    0]
    [x[2]    0 x[3]    0    0    0]
    [   0    0    0 x[3]    0 x[1]]
    [   0    0    0 x[2] x[1]    0]
    [   0    0    0 x[1]    0    0]
    [   0    0 x[1]    0 x[3] x[2]]
    [   0 x[1]    0    0 x[2]    0]
    [x[1]    0    0    0    0 x[3]],
    Graded module homomorphism (6 by 3) of degree 0
    Ambient matrix:
    [x[3]    0    0]
    [   0 x[2]    0]
    [x[2] x[3]    0]
    [   0    0 x[1]]
    [   0 x[1] x[2]]
    [x[1]    0 x[3]],
    Graded module homomorphism (3 by 1) of degree 0
    Ambient matrix:
    [x[3]]
    [x[2]]
```

(continues on next page)

(continued from previous page)

```
    [x[1]],
    Graded module homomorphism (1 by 0)
*]
```

Example: Betti Numbers of a Localized Module (ex-af7b7b)

We construct a non-homogeneous quotient module M of $\mathbb{Q}^3$. As expected, the Betti numbers of the localization of M are smaller than the Betti numbers of M.

```
> R<x,y,z> := PolynomialRing(RationalField(), 3, "grevlex");
> R3 := RModule(R, 3);
> B := [R3 | [x*y, x^2, z], [x*z^3, x^3, y], [y*z, z, x],
>       [z, y*z, x], [y, z, x]];
> M := quo<R3 | B>;
> M;
Reduced Module R^3/<relations>
Relations:
[  x*y,   x^2,     z],
[x*z^3,   x^3,     y],
[  y*z,     z,     x],
[    z,   y*z,     x],
[    y,     z,     x]
> BettiNumbers(M);
[ 3, 5, 4, 2 ]
> BettiNumbers(Localization(M));
[ 3, 5, 3, 1 ]
```

Since M is non-homogeneous, the Betti numbers are not unique. If we create a second module M_2 which is equivalent to M and compute the Betti numbers this time without homogenization (in the internal free resolution algorithm), then we obtain different Betti numbers for M_2. But since the Betti numbers over a local ring are unique, we get the same result for the localization of M_2.

```
> M2 := quo<R3 | B>;
> BettiNumbers(M2: Homogenize :=false);
[ 3, 6, 5, 2 ]
> BettiNumbers(Localization(M2): Homogenize:=false);
[ 3, 5, 3, 1 ]
```

Example: Hilbert Series from Betti Numbers (ex-ebeb1e)

Suppose M is a graded R-module. Given the graded Betti numbers $\beta_{i,j}$ of M, one can compute the Hilbert series $H_M(t)$ of M via the formula ([Eis95, Thm. 1.13] or [DL06, Thm. 1.22]):

$$H_M(t) = \frac{\sum_{i,j}(-1)^i \beta_{i,j} t^j}{D},$$

where D is the Hilbert denominator of M: this depends on the underlying ring R and equals

$$\prod_{i=1}^{n}(1 - t^{w_i}),$$

where n is the rank of R and w_i is the weight of the i-th variable (1 by default). We can thus write a simple function to compute the Hilbert series numerator via this formula.

```
> function HilbertNumeratorBetti(M)
>    P<t> := PolynomialRing(IntegerRing());
>    return &+[
>        (-1)^i*BettiNumber(M, i, j)*t^j:
```

(continues on next page)

(continued from previous page)

```
>           j in [0 .. MaximumBettiDegree(M, i)],
>           i in [0 .. #BettiNumbers(M)]
>    ];
> end function;
```

We then check that this function agrees with the Magma internal function `HilbertNumerator` for some modules. (Since the modules do not have negative gradings, we do not have to worry about the denominator shift which is 0 for these modules.) First we try the Twisted Cubic.

```
> Q := RationalField();
> R<x,y,z,t> := PolynomialRing(Q, 4, "grevlex");
> B := [
>     -x^2 + y*t, -y*z + x*t, x*z - t^2,
>     x*y - t^2, -y*z + x*t, -x^2 + z*t
> ];
> M := GradedModule(Ideal(B));
> HilbertNumeratorBetti(M);
-t^5 + 5*t^3 - 5*t^2 + 1
> HilbertNumerator(M);
-t^5 + 5*t^3 - 5*t^2 + 1
0
```

Now we apply the function to the module $M = R^1/I$ where I is the ideal generated by the 2×2 minors of a generic 4×4 matrix. Computing the Hilbert series numerator via the Betti numbers takes a little time since the resolution is non-trivial. Note the components of the Betti table which contribute to the terms of the Hilbert series numerator.

```
> n := 4;
> R<[x]> := PolynomialRing(Q, n^2, "grevlex");
> A := Matrix(n, [R.i: i in [1 .. n^2]]);
> A;
[ x[1]  x[2]  x[3]  x[4]]
[ x[5]  x[6]  x[7]  x[8]]
[ x[9] x[10] x[11] x[12]]
[x[13] x[14] x[15] x[16]]
> I := Ideal(Minors(A, 2));
> #Basis(I);
36
> M := QuotientModule(I);
> time HilbertNumeratorBetti(M);
-t^12 + 36*t^10 - 160*t^9 + 315*t^8 - 288*t^7 + 288*t^5 - 315*t^4 + 160*t^3 -
    36*t^2 + 1
Time: 0.470
> time HilbertNumerator(M);
-t^12 + 36*t^10 - 160*t^9 + 315*t^8 - 288*t^7 + 288*t^5 - 315*t^4 + 160*t^3 -
    36*t^2 + 1
0
Time: 0.000
> assert $1 eq $2;
> BettiNumbers(M);
[ 1, 36, 160, 315, 388, 388, 315, 160, 36, 1 ]
> BettiTable(M);
[
    [ 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
    [ 0, 36, 160, 315, 288, 100, 0, 0, 0, 0 ],
    [ 0, 0, 0, 0, 100, 288, 315, 160, 36, 0 ],
```

(continues on next page)

(continued from previous page)

```
    [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 ]
]
0
```

Example: Bounding the Regularity of a Module (ex-0f654c)

Given a graded module $M = R^1/I$, one can obtain an upper bound on the regularity of M by computing the regularity of $M_L = R^1/I_L$, where I_L is the leading monomial ideal of I. This will be faster in general (since the associated free resolution will be easier to compute).

```
> wts := [ 1, 5, 9, 13, 17, 5, 1, 1, 1 ];
> K := GF(32003);
> R<x0,x1,x2,x3,x4,y0,y1,u,t> := PolynomialRing(K, wts);
> I := Ideal([
>     x0*y0 - y1^3*u^3 - x1*t,
>     x1*y1 - x0*u^5 - t^6,
>     x1^2 - x0*x2 + y1^2*u^3*t^5,
>     x2^2 - x1*x3 + y0*y1*u^8*t^4,
>     x3^2 - x2*x4 + y0^2*u^13*t^3,
>     x3*y0 - u^18 - x4*t,
>     x4*y1 - x3*u^5 - y0^3*t^3,
>     x1*x2 - x0*x3 + y0*y1^2*u^3*t^4 + y1*u^8*t^5,
>     x2^2 - x0*x4 + y0*y1*u^8*t^4 + u^13*t^5,
>     x2*x3 - x1*x4 + y0^2*y1*u^8*t^3 + y0*u^13*t^4,
>     x1*y0 - y1^2*u^8 - x2*t,
>     x2*y0 - y1*u^13 - x3*t,
>     x2*y1 - x1*u^5 - y0*t^5,
>     x3*y1 - x2*u^5 - y0^2*t^4]);
> IsHomogeneous(I);
true
> M := GradedModule(I);
> time Regularity(M);
67
Time: 3.360
> IL := LeadingMonomialIdeal(I);
> ML := GradedModule(IL);
> time Regularity(ML);
92
Time: 0.530
> BettiNumbers(M);
[ 1, 14, 45, 72, 76, 58, 29, 8, 1 ]
> BettiNumbers(ML);
[ 1, 42, 210, 505, 723, 659, 388, 144, 31, 3 ]
```

Example: Random Space Curve of Genus 11 (ex-8c6c7c)

The following example shows how to explicitly use the resolution and syzygy functions to compute the ideal of a random space curve (in P^3) of genus 11. The construction is described in Section 1.2 of [ST02] and an equivalent form of the following computation is used by Magma's `RandomCurveByGenus` function to produce such curves. We work over the field $GF(101)$, which will be referred to as K and the polynomial ring R will be the 4 variable polynomial ring over K. The construction begins by choosing a random 8 x 3 matrix with entries given by random linear and quadratic polynomials of R in appropriate positions. The minimal free resolution of the reduced module having this as the matrix of relations is computed. The image of the second boundary map of the resolution is the module referred to as $\mathcal{G}^*$ in the above reference. Taking the submatrix of rows of a certain weighting of the matrix defining this map, we multiply by a 6 x 8 matrix with random entries in K. The resulting matrix represents a map from a free module F of rank 6 to $\mathcal{G}^*$, whose kernel is isomorphic to R as a submodule of F. The 6 coordinates of a generator of the

kernel generate the desired ideal I. This kernel is computed with a syzygy computation (note: we could also use `Kernel` for the matrix giving the map). We also check that the quotient module of I has a minimal free resolution of the right form.

```
> K := GF(101);
> R<x,y,z,t> := PolynomialRing(GF(101),4,"grevlex");
> v := [1,1,1,1,1,1,2,2];
> // generate the base random relations with appropriate linear and quadratic
> // entries using the Random function for multivariate polynomials.
> rels := [[Random(i,R,0): j in [1..3]] : i in v];
> Matrix(8,3,[TotalDegree(e) : e in &cat(rels)]);
[1 1 1]
[1 1 1]
[1 1 1]
[1 1 1]
[1 1 1]
[1 1 1]
[2 2 2]
[2 2 2]
> // get the quotient module
> F := RModule(R,3);
> M := quo<F|rels>;
```

Get the minimal free resolution and check that it has the correct Betti table.

```
> res := MinimalFreeResolution(M);
> BettiTable(res);
[
    [ 3, 6, 0, 0, 0 ],
    [ 0, 2, 8, 0, 0 ],
    [ 0, 0, 3, 10, 4 ]
]
0
```

Get the 2nd boundary map matrix and then the 8 x 8 submatrix of linear and quadratic entry rows.

```
> mat :=  Matrix(BoundaryMap(res,2));
> Nrows(mat); Ncols(mat);
11
8
> u := [1,1,2,2,2,2,2,2];
> mat := Matrix(R,[ri : i in [1..11] |
>          &and[(ri[j] eq 0) or (TotalDegree(ri[j]) eq u[j]):
>             j in [1..8]] where ri is Eltseq(mat[i])]);
> Nrows(mat); Ncols(mat);
8
8
> Matrix(8,8,[TotalDegree(m) :m in Eltseq(mat)]);
[1 1 2 2 2 2 2 2]
[1 1 2 2 2 2 2 2]
[1 1 2 2 2 2 2 2]
[1 1 2 2 2 2 2 2]
[1 1 2 2 2 2 2 2]
[1 1 2 2 2 2 2 2]
[1 1 2 2 2 2 2 2]
[1 1 2 2 2 2 2 2]
```

Now generate the random 6 x 8 matrix over K, compute the kernel of the composition via syzygies

and generate the matrix I.

```
> mat1 := Matrix(R,6,8,[Random(K) : i in [1..48]]);
> matc := mat1*mat;
> F1 := EModule(R,[2-x : x in u]);
> syz := SyzygyModule(sub<F1|RowSequence(matc)>);
> B := MinimalBasis(syz);
> #B;
1
> I := ideal<R|Eltseq(B[1])>;
```

Finally, check I has the right dimension (2) and degree (12) and that R/I has the correct minimal free resolution with Betti table as given in [ST02].

```
> Dimension(I); Degree(I);
2 [ 3, 4 ]
12
> OC := QuotientModule(I);
> BettiTable(MinimalFreeResolution(OC));
[
    [ 1, 0, 0, 0 ],
    [ 0, 0, 0, 0 ],
    [ 0, 0, 0, 0 ],
    [ 0, 0, 0, 0 ],
    [ 0, 6, 2, 0 ],
    [ 0, 0, 6, 3 ]
]
0
```

119.13 The Hom Module and Ext

Hom(M, N): ModMPol, ModMPol -> ModMPol, Map

Given $R-$modules M and N, return $H = \mathrm{Hom}_R(M, N)$ as an abstract reduced module and a transfer map $f : H \to S$, where S is the set of all homomorphisms (of type **ModMPolHom**) from M to N. Thus H is a module representing the set of all homomorphisms from M to N, while f maps an element $h \in H$ to an actual homomorphism from M to N (and the inverse image of an element of S under f gives a corresponding element of H). If M and N are graded, then H is graded also, and the degree d_f of an element $f \in H$ is the degree of the corresponding homomorphism (so an element in M of degree d will be mapped by f to zero or an element of degree $d_f + d$ in N).

Hom(C, N): ModCpx, ModMPol -> ModMPol

Given a complex C of R-modules and an R-module N, return $\mathrm{Hom}_R(C, N)$. This is a new complex whose i-th term is $\mathrm{Hom}_R(C_i, N)$ (where C_i is the i-th term of C); the boundary maps are also derived from those of C in the natural way via the functor $\mathrm{Hom}_R(-, N)$ (see [Eis95, p.63]). Note that the direction of arrows in this complex is opposite to that of C.

Ext(i, M, N): RngIntElt, ModMPol, ModMPol -> ModMPol

Given an integer $i \geq 0$ and R-modules M and N, return $\mathrm{Ext}^i(M, N)$. This is the homology at the i-th term of the complex $\mathrm{Hom}_R(C, N)$ where C is a free resolution of M.

Example: Hom Module and Explicit Homomorphisms (ex-7c5089)

We construct a Hom module and explicit homomorphisms derived from it.

```
> R<x,y,z> := PolynomialRing(RationalField(), 3);
> M := quo<GradedModule(R, 3) |
>     [x*y, x*z, y*z], [y, x, y],
>     [0, x^3 - x^2*z, x^2*y - x*y*z], [y*z, x^2, x*y]>;
```

(continues on next page)

(continued from previous page)

```
> N := quo<GradedModule(R, 2) |
>     [x^2, y^2], [x^2, y*z], [x^2*z, x*y^2]>;
> M;
Graded Module R^3/<relations>
Relations:
[          x*y,           x*z,           y*z],
[            y,             x,             y],
[            0,   x^3 - x^2*z, x^2*y - x*y*z],
[          y*z,           x^2,           x*y]
> N;
Graded Module R^2/<relations>
Relations:
[  x^2,   y^2],
[  x^2,   y*z],
[x^2*z, x*y^2]
> H, f := Hom(M, N);
> H;
Graded Module R^7/<relations> with grading [1, 2, 1, 1, 1, 1, 1]
Relations:
[x, 0, 0, -z, 0, x, 0],
[y, 0, x, 0, y, 0, 0],
[y, 0, x, 0, 0, y, 0],
[0, 0, 0, 0, 0, 0, y],
[-y, 0, -x, 0, -z, 0, z],
[x, 0, 0, -y, x, 0, 0],
[x*y, y, 0, 0, 0, 0, x*y],
[-x*y + x*z, -y + z, 0, 0, 0, 0, x*z - z^2],
[x*z, x, 0, 0, 0, 0, 0],
[0, y, 0, y^2, -z^2, 0, z^2],
[0, y - z, 0, 0, 0, 0, z^2]
> h := f(H.1);
> h;
Module homomorphism (3 by 2) of degree 1
Presentation matrix:
[0 z]
[x 0]
[0 0]
> $1 @@ f;
[1, 0, 0, 0, 0, 0, 0]
> Degree(M.1);
0
> h(M.1);
[0, z]
> Degree(h(M.1));
1
> f(Basis(H));
[
    Module homomorphism (3 by 2) of degree 1
    Presentation matrix:
    [0 z]
    [x 0]
    [0 0],
    Module homomorphism (3 by 2) of degree 2
    Presentation matrix:
    [   0 -z^2]
    [   0  y*z]
```

(continues on next page)

(continued from previous page)

```
    [   0    0],
    Module homomorphism (3 by 2) of degree 1
    Presentation matrix:
    [ 0  0]
    [-y  0]
    [ x  0],
    Module homomorphism (3 by 2) of degree 1
    Presentation matrix:
    [ 0  0]
    [ 0 -y]
    [ 0  x],
    Module homomorphism (3 by 2) of degree 1
    Presentation matrix:
    [ 0 -z]
    [ 0  0]
    [ 0  y],
    Module homomorphism (3 by 2) of degree 1
    Presentation matrix:
    [ 0 -z]
    [ 0  0]
    [ 0  z],
    Module homomorphism (3 by 2) of degree 1
    Presentation matrix:
    [    0 y - z]
    [    0     0]
    [    0     0]
]
```

119.14 Tensor Products and Tor

`TensorProduct(M, N): ModMPol, ModMPol -> ModMPol, Map`

Given $R-$modules M and N, return the tensor product $M \otimes_R N$ as an ambient module T, together with the associated map $f : M \times N \to T$. If M and N are graded, then T is graded also.

`TensorProduct(C, N): ModCpx, ModMPol -> ModMPol`

Given a complex C of R-modules and an R-module N, return $C \otimes_R N$. This is a new complex whose i-th term is $C_i \otimes_R N$ (where C_i is the i-th term of C); the boundary maps are also derived from those of C in the natural way via the functor $- \otimes_R N$ (see [Eis95, p.64]).

`Tor(i, M, N): RngIntElt, ModMPol, ModMPol -> ModMPol`

Given an integer $i \geq 0$ and R-modules M and N, return $\mathrm{Tor}^i(M, N)$. This is the homology at the i-th term of the complex $C \otimes_R N$ where C is a free resolution of M.

Example: Tensor Products and Tor Modules `(ex-ff1e8f)`

We construct a tensor product and some Tor modules for the same modules from the previous example.

```
> R<x,y,z> := PolynomialRing(RationalField(), 3);
> M := quo<GradedModule(R, 3) |
>     [x*y, x*z, y*z], [y, x, y],
>     [0, x^3 - x^2*z, x^2*y - x*y*z], [y*z, x^2, x*y]>;
> N := quo<GradedModule(R, 2) |
>     [x^2, y^2], [x^2, y*z], [x^2*z, x*y^2]>;
> T, f := TensorProduct(M, N);
> T;
```

(continues on next page)

(continued from previous page)

```
Graded Module R^6/<relations>
Relations (Groebner basis):
[x^2, y*z, 0, 0, 0, 0],
[0, 0, 0, 0, x^2, y*z],
[0, 0, 0, 0, 0, x*y*z - y*z^2],
[x*y - y*z, 0, 0, 0, 0, 0],
[0, x*y - y*z, 0, 0, 0, 0],
[y*z, 0, 0, -y*z, x*y, 0],
[y, 0, x, 0, y, 0],
[0, y, 0, x, 0, y],
[0, y^2 - y*z, 0, 0, 0, 0],
[0, 0, 0, y^2 - y*z, 0, 0],
[0, 0, 0, 0, 0, y^2 - y*z],
[y*z^2, 0, 0, -y*z^2, 0, -y*z^2],
[0, y*z^2, 0, y*z^2, 0, y*z^2]
```

Note that f maps the cartesian product of M and N into T.

```
> f(<M.1, N.1>);
[1, 0, 0, 0, 0, 0]
> [f(<m, n>): n in Basis(N), m in Basis(M)];
[
    [1, 0, 0, 0, 0, 0],
    [0, 1, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 0],
    [0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 1, 0],
    [0, 0, 0, 0, 0, 1]
]
```

Finally we construct associated Tor modules.

```
> Tor(0, M, N);
Graded Module R^6/<relations>
Relations:
[y, 0, x, 0, y, 0],
[0, y, 0, x, 0, y],
[0, 0, 0, 0, x*y - y*z, 0],
[0, 0, 0, 0, 0, x*y - y*z],
[y*z, x^2, 0, 0, 0, 0],
[x*y*z - y*z^2, 0, 0, 0, 0, 0],
[y^2 - y*z, 0, 0, 0, 0, 0],
[0, 0, y*z, x^2, 0, 0],
[0, 0, x*y*z - y*z^2, 0, 0, 0],
[0, 0, y^2 - y*z, 0, 0, 0],
[0, 0, 0, 0, y*z, x^2],
[0, 0, 0, 0, y^2 - y*z, 0],
[0, 0, 0, 0, x*y*z - y*z^2, 0]
> Tor(1, M, N);
Graded Module R^2/<relations> with grading [3, 3]
Relations:
[y - z,     0],
[    z,    -y],
[  z^2,  -x*y],
[    0,     0]
> Tor(2, M, N);
Free Reduced Module R^0
```

119.15 Cohomology Of Coherent Sheaves

We have implemented functions to compute the dimensions of cohomology groups of coherent sheaves on ordinary projective space over an (exact) field. The sheaves are represented by graded modules over the coordinate ring of the ambient projective space. The sheaf may arise naturally as one supported on a particular closed subscheme (*eg*, the structure sheaf of a projective variety) but it is a matter of indifference whether the sheaf is considered as lying on the subscheme or the entire ambient space (equivalently, whether the representing module is considered as a module over the coordinate ring of the ambient or the quotient coordinate ring of the subscheme) because the cohomology groups are naturally isomorphic. We plan to add a fuller package of functionality for coherent sheaves, but it is convenient to add the cohomology function now as the algorithm we use works equally efficiently (roughly speaking) when applied to any two graded modules that represent the same coherent sheaf.

The algorithm we have implemented is that of Decker, Eisenbud, Floystad, and Schreyer which uses the Beilinson-Gelfand-Gelfand (BGG) correspondence to reduce the computation of the cohomology groups of the sheaf and its Serre twists to that of various graded free modules in the projective resolution of a module over a finite exterior (alternating) algebra.

`CohomologyDimension(M, r, n): ModMPolGrd, RngIntElt, RngIntElt -> RngIntElt`

```
Cohom: Verbose                    Default: 1
```

M is a graded module over $P = k[x_0, .., x_m]$ with k an exact field. Let $\tilde{M}$ be the corresponding coherent sheaf on $Proj(P) = \mathbb{P}^m_k$. The function returns the k-dimension of the cohomology group $H^r(\mathbb{P}^m_k, \tilde{M}(n))$ where $\tilde{M}(n)$ is the nth Serre twist of $\tilde{M}$. n can be any integer and r a non-negative integer.

The algorithm used is based on the BGG correspondence. Details can be found in [EFS03] or see [DE02] for a slightly more computational description. Let A be the finite exterior algebra with $m+1$ generators, which is of dimension 2^{m+1} over k. The *Tate resolution* of $\tilde{M}$ is a doubly infinite exact sequence of graded free A-modules. Each cohomology group of a twist of $\tilde{M}$ is isomorphic as a k vector space to a particular graded piece of a particular term in the Tate resolution. In fact, we never need to explicitly compute the terms of the resolution of index $\geq reg(M)$ (the regularity of M) because they are pure graded of dimension given by the Hilbert polynomial of M.

The algorithm computes two consecutive terms in the Tate resolution at indices $\geq reg(M)$, and the A-homomorphism between them, from two corresponding graded pieces of M and the linear maps between them coming from multiplication by the base variables. Then the resolution is extended backwards as far as necessary by computing the A-projective resolution of the kernel of this A-homomorphism. The projective resolution is efficiently determined by non-commutative Gröbner basis computations. This uses the new Magma machinery for exterior algebras and their modules. The projective resolution information is cached so that repeated calls to the function for the same module M will require either no extra work or only an extension to the part of the resolution already computed.

Example: Cohomology of an Enriques Surface `(ex-f9ee7a)`

We consider a random surface X in a family of Enriques surfaces of degree 9 in $\mathbb{P}^4$. It is defined by 15 degree 5 polynomials and we work over $\mathbb{F}_{17}$ to keep the input of reasonable size (it is still fairly large!).

The surface is non-singular with arithmetic genus (p_a), geometric genus (g) and irregularity q all zero. These are related generally for a non-singular surface by $g = p_a + q$ and p_a can be computed without cohomology machinery (from Hilbert polynomials). But cohomology of the structure sheaf of X and Serre duality is the easiest way to get g or q.

```
> R<x,y,z,t,u> := PolynomialRing(GF(17),5,"grevlex");
> I := ideal<R |[
>  2*x^3*z*t + 5*x^2*y*z*t + 14*x^2*z^2*t + x^3*z*u + 5*x^2*y*z*u +
>   10*x^2*z^2*u + 8*x*y*z^2*u + 15*y^2*z^2*u + 4*x*z^3*u + 2*y*z^3*u +
```

(continues on next page)

(continued from previous page)

```
>   9*x^3*t*u + 14*x^2*y*t*u + 16*x^2*z*t*u + 10*x*y*z*t*u + 10*x*z^2*t*u +
>   y*z^2*t*u + 13*x^3*u^2 + 14*x^2*y*u^2 + 11*x^2*z*u^2 + 15*x*y*z*u^2 +
>   10*y^2*z*u^2 + 8*x*z^2*u^2 + 8*y*z^2*u^2 + x^2*t*u^2 + 11*x*y*t*u^2 +
>   16*x*y*u^3 + 9*y^2*u^3 + 4*x*z*u^3 + 2*y*z*u^3 + 10*x*t*u^3 + y*t*u^3 +
>   8*x*u^4 + 8*y*u^4,
>  5*x^3*z*t + x^2*z^2*t + 5*x^3*z*u + 11*x^2*z^2*u + 15*x*y*z^2*u + 2*x*z^3*u
>   + 14*x^3*t*u + 5*x^2*z*t*u + x*z^2*t*u + 14*x^3*u^2 + 10*x*y*z*u^2 +
>   8*x*z^2*u^2 + 15*x^2*t*u^2 + 11*x^2*u^3 + 9*x*y*u^3 + 2*x*z*u^3 +
>   x*t*u^3 + 8*x*u^4,
>  14*x^3*z*t + x^2*y*z*t + 13*x^2*z^2*t + 7*x^2*z*t^2 + 3*x^3*z*u +
>   16*x^2*y*z*u + 4*x^2*z^2*u + 6*x^3*t*u + 16*x^2*y*t*u + 9*x^2*z*t*u +
>   9*x^2*t^2*u + 11*x^3*u^2 + x^2*y*u^2 + 14*x^2*z*u^2 + 2*x*z^2*u^2 +
>   11*y*z^2*u^2 + 6*z^3*u^2 + 4*x^2*t*u^2 + 4*x*z*t*u^2 + 14*y*z*t*u^2 +
>   6*z^2*t*u^2 + 15*x*t^2*u^2 + 10*z*t^2*u^2 + 3*x^2*u^3 + 11*x*z*u^3 +
>   16*y*z*u^3 + 4*z^2*u^3 + 16*x*t*u^3 + 3*y*t*u^3 + 14*z*t*u^3 + 6*t^2*u^3
>   + 13*x*u^4 + 7*y*u^4 + 16*z*u^4 + 11*t*u^4 + 10*u^5,
>  15*x^3*z^2 + 12*x^2*y*z^2 + 3*x^2*z^3 + 12*x^3*z*u + 8*x^2*y*z*u +
>   11*x^2*z^2*u + x^3*u^2 + 14*x^2*y*u^2 + 3*x^2*z*u^2 + 11*x^2*u^3,
>  12*x^3*z^2 + 16*x^2*z^3 + 8*x^3*z*u + x^2*z^2*u + 14*x^3*u^2 + 16*x^2*z*u^2
>   + x^2*u^3,
>  2*x^3*y*z + 5*x^2*y^2*z + 14*x^2*y*z^2 + 13*x^3*y*u + 12*x^2*y^2*u +
>   8*x^3*z*u + 4*x^2*y*z*u + 12*x^2*z^2*u + 3*x*y*z^2*u + 14*y^2*z^2*u +
>   15*x*z^3*u + 3*y*z^3*u + 15*x^2*y*t*u + 4*x^2*z*t*u + 15*x*y*z*t*u +
>   11*x*z^2*t*u + 10*y*z^2*t*u + 2*x^3*u^2 + 12*x^2*y*u^2 + 3*x^2*z*u^2 +
>   14*x*y*z*u^2 + 13*x*z^2*u^2 + 10*y*z^2*u^2 + x^2*t*u^2 + 15*x*y*t*u^2 +
>   16*y*z*t*u^2 + 12*x*y*u^3 + 3*y^2*u^3 + 15*x*z*u^3 + 4*y*z*u^3 +
>   11*x*t*u^3 + 6*y*t*u^3 + 13*x*u^4 + 14*y*u^4,
>  5*x^3*y*z + x^2*y*z^2 + 10*x^2*z^2*t + 4*x^4*u + 12*x^3*y*u + 5*x^3*z*u +
>   12*x^2*y*z*u + 14*x*y*z^2*u + 3*x*z^3*u + 15*x^3*t*u + 16*x^2*z*t*u +
>   10*x*z^2*t*u + 11*x^3*u^2 + 4*x^2*y*u^2 + 13*x^2*z*u^2 + 10*x*z^2*u^2 +
>   4*x^2*t*u^2 + 16*x*z*t*u^2 + 13*x^2*u^3 + 3*x*y*u^3 + 4*x*z*u^3 +
>   6*x*t*u^3 + 14*x*u^4,
>  10*x^2*z^3 + 8*x^2*z^2*u + 5*x^2*z*u^2 + 11*x^2*u^3,
>  16*x^3*z^2 + 12*x^2*y*z^2 + 7*x^2*z^3 + 9*x*y*z^3 + 2*y^2*z^3 + 13*x*z^4 +
>   15*y*z^4 + 13*x^3*z*t + 12*x^2*y*z*t + 7*x^2*z^2*t + 7*x*y*z^2*t +
>   7*x*z^3*t + 16*y*z^3*t + 4*x^3*z*u + 3*x^2*y*z*u + 6*x^2*z^2*u +
>   2*x*y*z^2*u + 7*y^2*z^2*u + 9*x*z^3*u + 9*y*z^3*u + 16*x^3*t*u +
>   3*x^2*y*t*u + 13*x^2*z*t*u + 6*x*y*z*t*u + x*y*z*u^2 + 8*y^2*z*u^2 +
>   13*x*z^2*u^2 + 15*y*z^2*u^2 + 6*x^2*t*u^2 + 7*x*z*t*u^2 + 16*y*z*t*u^2 +
>   9*x*z*u^3 + 9*y*z*u^3,
>  12*x^3*z^2 + 6*x^2*z^3 + 2*x*y*z^3 + 15*x*z^4 + 12*x^3*z*t + 11*x^2*z^2*t +
>   16*x*z^3*t + 3*x^3*z*u + 7*x*y*z^2*u + 9*x*z^3*u + 3*x^3*t*u +
>   3*x^2*z*t*u + 6*x^2*z*u^2 + 8*x*y*z*u^2 + 15*x*z^2*u^2 + 16*x^2*t*u^2 +
>   16*x*z*t*u^2 + 9*x*z*u^3,
>  x^3*y*z + 5*x^2*y^2*z + 10*x^2*y*z^2 + 8*x*y^2*z^2 + 15*y^3*z^2 + 4*x*y*z^3
>   + 2*y^2*z^3 + 13*x^3*y*t + 2*x^2*y^2*t + 9*x^3*z*t + 12*x^2*y*z*t +
>   10*x*y^2*z*t + 5*x^2*z^2*t + 7*x*y*z^2*t + 4*y^2*z^2*t + 2*x*z^3*t +
>   14*y*z^3*t + 2*x^2*y*t^2 + 13*x^2*z*t^2 + 2*x*y*z*t^2 + 6*x*z^2*t^2 +
>   7*y*z^2*t^2 + 13*x^3*y*u + 14*x^2*y^2*u + 11*x^2*y*z*u + 15*x*y^2*z*u +
>   10*y^3*z*u + 8*x*y*z^2*u + 8*y^2*z^2*u + 15*x^3*t*u + 6*x^2*y*t*u +
>   11*x*y^2*t*u + 14*x^2*z*t*u + 3*x*y*z*t*u + 4*x*z^2*t*u + 7*y*z^2*t*u +
>   16*x^2*t^2*u + 2*x*y*t^2*u + y*z*t^2*u + 16*x*y^2*u^2 + 9*y^3*u^2 +
>   4*x*y*z*u^2 + 2*y^2*z*u^2 + 15*x*y*t*u^2 + 15*y^2*t*u^2 + 2*x*z*t*u^2 +
>   13*y*z*t*u^2 + 6*x*t^2*u^2 + 11*y*t^2*u^2 + 8*x*y*u^3 + 8*y^2*u^3 +
>   4*x*t*u^3 + 3*y*t*u^3,
>  5*x^3*y*z + 11*x^2*y*z^2 + 15*x*y^2*z^2 + 2*x*y*z^3 + 13*x^4*t + 2*x^3*y*t +
```

(continues on next page)

(continued from previous page)

```
>   7*x^3*z*t + 6*x^2*y*z*t + 16*x^2*z^2*t + 4*x*y*z^2*t + 14*x*z^3*t +
>   2*x^3*t^2 + 9*x^2*z*t^2 + 7*x*z^2*t^2 + 14*x^3*y*u + 5*x^3*z*u +
>   4*x^2*y*z*u + 10*x*y^2*z*u + 12*x^2*z^2*u + 8*x*y*z^2*u + 15*x*z^3*u +
>   6*y*z^3*u + 11*z^4*u + 16*x^3*t*u + 15*x^2*y*t*u + 13*x^2*z*t*u +
>   3*x*z^2*t*u + 3*y*z^2*t*u + 11*z^3*t*u + 13*x^2*t^2*u + 3*x*z*t^2*u +
>   7*z^2*t^2*u + 7*x^3*u^2 + 7*x^2*y*u^2 + 9*x*y^2*u^2 + x^2*z*u^2 +
>   2*x*y*z*u^2 + 6*x*z^2*u^2 + y*z^2*u^2 + 13*z^3*u^2 + 12*x^2*t*u^2 +
>   15*x*y*t*u^2 + 14*x*z*t*u^2 + 14*y*z*t*u^2 + 3*z^2*t*u^2 + 11*x*t^2*u^2
>   + 11*z*t^2*u^2 + 9*x^2*u^3 + 8*x*y*u^3 + 4*x*z*u^3 + 10*y*z*u^3 +
>   z^2*u^3 + 3*x*t*u^3 + 6*z*t*u^3 + 7*z*u^4,
>  4*x^4*z + 3*x^3*y*z + 10*x^3*z^2 + x^2*z^3 + 14*x*y*z^3 + 3*x*z^4 +
>   15*x^3*z*t + 8*x^2*z^2*t + 10*x*z^3*t + 14*x^3*y*u + 15*x^3*z*u +
>   11*x^2*z^2*u + 10*x*z^3*u + 16*x^2*z*t*u + 16*x*z^2*t*u + 6*x^3*u^2 +
>   14*x^2*z*u^2 + 3*x*y*z*u^2 + 4*x*z^2*u^2 + 6*x^2*t*u^2 + 6*x*z*t*u^2 +
>   15*x^2*u^3 + 14*x*z*u^3,
>  5*x^3*z^2 + 4*x^2*y*z^2 + 12*x^2*z^3 + 15*x*z^4 + 6*y*z^4 + 11*z^5 +
>   14*x^3*z*t + x^2*y*z*t + 7*x^2*z^2*t + 13*x*z^3*t + 3*y*z^3*t + 11*z^4*t
>   + 12*x^2*z*t^2 + 2*x*z^2*t^2 + 7*z^3*t^2 + 7*x^3*z*u + 13*x^2*y*z*u +
>   x^2*z^2*u + 6*x*z^3*u + y*z^3*u + 13*z^4*u + 6*x^3*t*u + 16*x^2*y*t*u +
>   9*x^2*z*t*u + x*z^2*t*u + 14*y*z^2*t*u + 3*z^3*t*u + 6*x^2*t^2*u +
>   11*z^2*t^2*u + 9*x^2*z*u^2 + 4*x*z^2*u^2 + 10*y*z^2*u^2 + z^3*u^2 +
>   15*x^2*t*u^2 + 6*z^2*t*u^2 + 7*z^2*u^3,
>  13*x^5 + 2*x^4*y + 7*x^4*z + 6*x^3*y*z + 16*x^3*z^2 + 4*x^2*y*z^2 +
>   14*x^2*z^3 + 2*x^4*t + 9*x^3*z*t + 7*x^2*z^2*t + 16*x^4*u + 15*x^3*y*u +
>   x^3*z*u + 11*x^2*z^2*u + 3*x*y*z^2*u + 14*x*z^3*u + 13*x^3*t*u +
>   3*x^2*z*t*u + 7*x*z^2*t*u + 14*x^3*u^2 + 15*x^2*y*u^2 + 6*x^2*z*u^2 +
>   14*x*y*z*u^2 + 10*x*z^2*u^2 + 11*x^2*t*u^2 + 11*x*z*t*u^2 + 3*x*z*u^3]>;
> X := Scheme(Proj(R),I); // define the surface
> // The structure sheaf O_X of X is represented by graded module R/I
> O_X := GradedModule(I); // R/I
```

We first compute the dimension of $H^0(O_X)$. This is fairly uninteresting (it's just 1) but after this computation, the cached data will allow the following cohomology calls to execute practically instantaneously.

```
> CohomologyDimension(O_X,0,0);  // dim H^0(O_X)
1
> // get the geometric genus and irregularity
> time CohomologyDimension(O_X,2,0); // dim H^2(O_X) = g
0
Time: 0.000
> time CohomologyDimension(O_X,1,0); // dim H^1(O_X) = q
0
Time: 0.000
> // => p_a(X)=0. Verify this.
> ArithmeticGenus(X);
0
```

We now compute a module representative of the canonical sheaf K_X of X. We can just take $Ext^2_R(M, R(-5))$. Then we check again that $H^0(K_X) = g$ is 0. Note that here the module representing K_X is maximal so that $H^0(K_X(n))$ is just the dimension of its nth graded part for any n. However, in other cases (where X is again *not* arithmetically Cohen-Macaulay) the Ext computation for K_X may not give the maximal representing module, so its 0th graded piece might have dimension less than g.

```
> K_X := Ext(2,O_X,RModule(R,[5]));
```

(continues on next page)

(continued from previous page)

```
> K_X;
Reduced Module R^6/<relations> with grading [1, 1, 1, 1, 1, 1]
Quotient Relations:
[12*z + 8*t + 7*u, 5*y + 7*z + 8*t + 6*u, 16*z + 4*u, 16*z + 7*u, 15*u,
    7*u],
[14*u, 8*u, 13*x + 2*y + 4*t + 14*u, 4*y + 8*t + 2*u, 16*t + 6*u, 3*u],
[15*z + 9*t + 11*u, 2*y + 13*z + 8*t + 14*u, 7*t + 16*u, 8*x + 14*t + 6*
    11*t + 15*u, 11*z + 14*t + 16*u],
[12*z + 15*t + 6*u, 5*y + 7*z + 4*t + 9*u, t + u, 2*t + 2*u, 7*x + 4*t +
    12*z + t + 2*u],
[12*y + 10*z + 14*t + 14*u, 7*y + 15*z + 8*t, 4*u, 8*u, 16*u, 5*x + 15*z
    + 9*u],
[15*x, 15*x, 4*u, 8*u, 16*u, 0],
[14*z + 15*t + 9*u, 3*y + 11*z + 7*t + 13*u, 0, 11*z + 6*u, 2*z + 14*u,
    10*t + 4*u],
[10*z + 14*t + 14*u, 4*x + 12*y + 15*z + 8*t, 0, 0, 0, 15*z + 16*t + 9*u
[5*z + 16*t + 4*u, 12*y + 10*z + 15*t + 5*u, 0, 2*z + 15*u, 6*u, 15*z +
    7*u],
[13*z + t + 15*u, 4*y + 9*z + 16*t + 16*u, 0, 0, 0, 5*z + 11*t + 3*u]
> CohomologyDimension(K_X,0,0);
0
```

Finally, we verify some more cases of Serre duality which gives

$$\dim\ H^r(O_X(n)) = \dim\ H^{2-r}(K_X(-n)).$$

```
> [CohomologyDimension(K_X,0,i) eq CohomologyDimension(O_X,2,-i) :
>    i in [-1..5]];
[ true, true, true, true, true, true, true ]
```

119.16 Bibliography

- [AL94] William Adams and Philippe Loustaunau. *An introduction to Gröbner bases*. Volume 3 of Graduate studies in mathematics. American Mathematical Society, Providence, R.I., 1994.
- [BS92] David Bayer and Michael Stillman. Computation of Hilbert Functions. *J. Symbolic Comp.*, 14(1):31–50, 1992.
- [CLO98] David Cox, John Little, and Donal O'Shea. *Using Algebraic Geometry*. Graduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 1998.
- [DE02] Wolfram Decker and David Eisenbud. Sheaf algorithms using the Exterior algebra. In Eisenbud and others, editors, *Computations in Algebraic Geometry with Macaulay2*, volume 8 of Springer Algorithms and Computation in Mathematics Series, 215–247. Springer-Verlag, 2002.
- [DL06] Wolfram Decker and Christoph Lossen. *Computing in Algebraic Geometry*. Volume 16 of Algorithms and Computation in Mathematics. Springer, New York–Berlin–Heidelberg, 2006.
- [EFS03] Eisenbud, Floystad, and Schreyer. Sheaf Cohomology and Free Resolutions over Exterior Algebras. *Trans. Am. Maths. Soc.*, 355:4397–4426, 2003.
- [Eis95] David Eisenbud. *Commutative Algebra with a View Toward Algebraic Geometry*. Volume 150 of Graduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 1995.
- [Fau99] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases (F_4). *Journal of Pure and Applied Algebra*, 139 (1-3):61–88, 1999.
- [GP02] G.-M. Greuel and G. Pfister. *A Singular Introduction to Commutative Algebra*. Springer-Verlag, Berlin–Heidelberg–New York, 2002. ISBN 3-540-42897-6.

- [SS98] Roberto La Scala and Michael Stillman. Strategies for Computing Minimal Free Resolutions. *J. Symbolic Comp.*, 26(4):409–431, 1998.
- [ST02] Frank-Olaf Schreyer and Fabio Tonoli. Needles in a Haystack: Special Varieties via Small Fields. In Eisenbud and others, editors, *Computations in Algebraic Geometry with Macaulay2*, volume 8 of Springer Algorithms and Computation in Mathematics Series, 251–277. Springer-Verlag, 2002.

Chapter 120

Invariant Theory

120.1 Introduction

Magma contains a powerful module for computing with invariant rings and fields of finite groups and algebraic groups. The algorithms for invariant theory of finite groups in Magma are based on those in the *Invar* package written in Maple, implemented by G. Kemper [Kem96], but also include many new ideas and improvements which are described in detail in a subsequent paper [KS97]. Since a detailed understanding of the latter paper is useful for better understanding of many of the functions in the chapter, it is recommended the paper be perused by anyone wishing to make serious applications of the functions. Since V2.14, Magma also has algorithms for invariant theory of linear algebraic groups. In particular, Derksen's algorithm [Der99] and the algorithm by Beth and Müller-Quade [MQB99] have been implemented. These additions use code written by G. Kemper. The primary goal of invariant theory in Magma is the computation of generators of the invariant ring or field of a given group, which may be finite or algebraic. The ground field may have arbitrary characteristic. In invariant theory of finite groups, the *modular case*, i.e., the case where the characteristic of the ground field K divides the group order, is of particular interest, since in that case there are still many theoretical questions unanswered. Magma also contains easy algorithms to calculate properties of modular invariant rings, such as the Hilbert series, the Cohen-Macaulay property, depth, and free resolutions. The approach to calculating the invariant ring of a finite group is broken up into two major steps: first a system of *primary invariants* is constructed, i.e., homogeneous invariants $f_1, \ldots, f_n$ which are algebraically independent, such that the invariant ring is a finitely generated module over $A = K[f_1, \ldots, f_n]$. In the next step we calculate *secondary invariants*, which are generators of the invariant ring as an A-module. Throughout this chapter, K will be a field and G is a group acting linearly on the n-dimensional vector space $V \cong K^n$ with basis $x_1, \ldots, x_n$. G may be a linear algebraic group, in which case K is assumed to be algebraically closed, or a finite matrix group, or a permutation group. G also acts on the symmetric algebra $K[V] = S(V)$, which is the multivariate polynomial ring $K[x_1, \ldots, x_n]$ in the variables $x_1, \ldots, x_n$. The invariant ring $R = \{f \in K[V] \mid f^\sigma = f \ \forall \sigma \in G\}$ is denoted by $K[V]^G$. The G-action extends naturally to the rational function field $K(V)$ on V, leading to the analogous definition of the invariants field $K(V)^G$. Sections *Creation* through *Verbosity* describe the general setup of invariant theory in Magma. Section *Construction of Invariants of Specified Degree* is about computing invariants of specified degree. Sections *Construction of G-modules* through *Properties of Invariant Rings* deal with functions for invariant rings of finite groups. The following Sections *Steenrod Operations* and *Minimalization and Homogeneous Module Testing* present some functions whose scope is not limited to the context of invariant theory. Sections *Invariant Rings of Linear Algebraic Groups* and *Invariant Fields* are about functions for invariant rings of algebraic groups and for invariant fields, respectively. The Section *Attributes of Invariant Rings and Fields* gives information about some low-level control of the data structures associated to invariant theory. Finally, since V2.14, the Section *Invariant Rings of Linear Algebraic Groups* deals with invariant rings of algebraic groups and the Section *Invariant Fields* deals with invariant fields.

120.2 Invariant Rings of Finite Groups

120.2.1 Creation

Let G be a finite matrix or permutation group acting on the polynomial ring $P = K[x_1, \ldots, x_n]$ over the field K. Magma allows the construction of the invariant ring $R = K[V]^G$. The invariant ring R is a special structure which contains references to the group G and polynomial ring P. When the invariant ring R is created using the `InvariantRing` function, no explicit calculations are done until specifically invoked (e.g., by the *PrimaryInvariants* function). The elements of R are the polynomials of P which are invariant under the action of G. Note that the parent of such polynomials is still P – the invariant ring R is just a special structure which contains all the information about the invariant ring. The category of invariant rings is `RngInvar`.

`InvariantRing(G): GrpMat -> RngInvar`

`InvariantRing(G, K): GrpPerm, Fld -> RngInvar`

Construct the invariant ring $R = K[V]^G$ of the finite matrix or permutation group G over the field K. For a matrix group G, G alone should be supplied, while for a permutation group G, G should be supplied, together with the field K. The appropriate multivariate polynomial ring P is automatically constructed. No other explicit calculations are done (e.g. computation of primary invariants).

120.2.2 Access

The following functions allow simple access to basic properties of invariant rings.

`Group(R): RngInvar -> Grp`

Given the invariant ring $R = K[V]^G$ of the group G over the field K, return the group G.

`CoefficientRing(R): RngInvar -> Grp`

`CoefficientField(R): RngInvar -> Rng`

Given the invariant ring $R = K[V]^G$ of the group G over the field K, return the coefficient field K.

`PolynomialRing(R): RngInvar -> RngMPol`

Given an invariant ring $R = K[V]^G$ of the group G of degree n over the field K, return the polynomial ring $P = K[x_1, \ldots, x_n]$ in which the invariants of R lie. P has the print names `"x1"`, `"x2"`, etc. – the angle bracket notation or the . operator should be used to assign the variables of P to actual Magma variables.

`f in R: RngMPol, RngInvar -> FldFunUElt, ModMPolElt`

Return whether the polynomial f is in $R = K[V]^G$. Note that the parent of f is always the polynomial ring P, never R, so a `true` result does not mean that the parent of f is R.

120.3 Group Actions on Polynomials

This section describes in detail the actions which groups have on multivariate polynomial rings.

120.4 Permutation Group Actions on Polynomials

If P is a polynomial ring in n indeterminates $x_1, \ldots, x_n$, over any coefficient ring, $\mathrm{Sym}(n)$ acts on P by permuting the indices of the indeterminates. Thus, the polynomial $f(x_1, \ldots, x_n)$ is mapped into the polynomial $f(x_{g(1)}, \ldots, x_{g(n)})$.

`f ^ g: RngMPolElt, GrpPermElt -> RngMPolElt`

Given a polynomial f belonging to a polynomial ring having n indeterminates, and a permutation g belonging to a subgroup of $\mathrm{Sym}(\{1, \ldots, n\})$, return the image of f under g.

`f ^ G: RngMPolElt, GrpPerm -> { RngMPolElt }`

Given a polynomial f belonging to a polynomial ring having n indeterminates, and a permutation group G contained in $\mathrm{Sym}(\{1,\ldots,n\})$, return the orbit of f under G.

`IsInvariant(f, g): RngMPolElt, GrpElt -> BoolElt`

Given a polynomial f belonging to a polynomial ring having n indeterminates, and a permutation g of degree n or an element of a matrix group of degree n whose coefficient ring is the same as that of f, return whether f is an invariant of g, i.e., whether $f^g = f$.

`IsInvariant(f, G): RngMPolElt, Grp -> BoolElt`

Given a polynomial f belonging to a polynomial ring having n indeterminates, and a permutation group G of degree n or a matrix group of degree n whose coefficient ring is the same as that of f, return whether f is an invariant of G, i.e., whether $f^g = f$ for all $g \in G$.

120.5 Matrix Group Actions on Polynomials

If P is a polynomial ring in n indeterminates $x_1,\ldots,x_n$, over the ring S, then $\mathrm{GL}(n,S)$ acts on P as follows: Let $\mathbf{x}$ denote the vector $(x_1,\ldots,x_n)$. Then the image g of a polynomial f of P under the action of a matrix a of $\mathrm{GL}(n,S)$ is defined by $g(\mathbf{x}) = f(\mathbf{x} * a)$.

`f ^ a: RngMPolElt, GrpMatElt -> RngMPolElt`

Given a polynomial f belonging to a polynomial ring having n indeterminates and coefficient ring S, and a matrix a belonging subgroup G of $\mathrm{GL}(n,S)$, return the image of f under a.

`f ^ G: RngMPolElt, GrpMat -> { RngMPolElt }`

Given a polynomial f belonging to a polynomial ring having n indeterminates and coefficient ring S, and a to a subgroup of $\mathrm{GL}(n,S)$, return the orbit of f under G.

Example: Matrix Group Action on Polynomials (ex-df3c71)

We act on the polynomial ring in two indeterminates over the field $K = Q(\sqrt{2})$, by a cyclic subgroup of $\mathrm{GL}(2,K)$.

```
> K := QuadraticField(2);
> Aq := [ x / K.1 : x in [1, 1, -1, 1]];
> G := MatrixGroup<2, K | Aq>;
> P<x, y> := PolynomialRing(K, 2);
> f := x^2 + x * y + y^2;
> g := f^G.1;
> g;
1/2*x^2 + 3/2*y^2
> f^G;
{
   1/2*x^2 + 3/2*y^2,
   x^2 - x*y + y^2,
   x^2 + x*y + y^2,
   3/2*x^2 + 1/2*y^2
}
```

120.6 Algebraic Group Actions on Polynomials

In the invariant theory package of Magma, a linear algebraic group G is given by polynomials, say in variables $t_1,\ldots,t_m$, defined over some field K that is representable in Magma, as the affine variety over the algebraic closure $\bar{K}$ of K given by these polynomials. A G-module is given by a matrix $A \in K[t_1,\ldots,t_m]^{n\times n}$ such that a group element $(\eta_1,\ldots,\eta_m) \in G$ acts on $\bar{K}^n$ by the matrix obtained by substituting $(\eta_1,\ldots,\eta_m)$ into the polynomials occurring in the matrix A.

G then also acts on the ring of polynomials on $\bar{K}^n$ by

$$\sigma(f) = f \circ \sigma^{-1}$$

for $\sigma \in G$ and $f \in \bar{K}[x_1, \ldots, x_n]$. Since the algorithms in Magma do not work with the algebraic closure, single group elements are never dealt with. In fact, all relevant algorithms ony involve field elements of K, the field of definition.

120.7 Verbosity

The following procedure allows verbose information for the Invariant Theory algorithms to be displayed.

SetVerbose("Invariants", v): MonStgElt, RngIntElt

(Procedure.) Set the verbose printing level for the Invariant Theory algorithms of Magma to be v. Currently the legal values for v are `true`, `false`, 0, 1, 2, 3, or 4 (`false` has the same effect as 0, and `true` has the same effect as 1). Level 1 gives a minimal amount of useful information during the running of all the algorithms while higher levels give more detailed information. For the primary invariants computation, the verbose output displays each possible degree list (degrees of the potential primary invariants) before and then tries to find primary invariants corresponding to this degree list. For the secondary invariants computation, in the non-modular case the algorithm loops over the necessary degrees in increasing order and computes the relevant new invariants; in the modular case the algorithm finds secondary invariants with respect to a subgroup and then performs a module syzygy computation. Full details of the algorithms are found in [KS97].

120.8 Construction of Invariants of Specified Degree

Let $R = K[V]^G$ be the invariant ring of the group G over the field K. Let $d \geq 0$ be a fixed integer. The homogeneous invariants in R of degree d form a vector space R_d over K.

There are two ways of explicitly constructing homogeneous invariants in R of degree d: the *Reynolds operator* method and the *linear algebra* method. Both methods are described in detail in [KS97].

The Reynolds operator method only works fro finite groups in the non-modular case. It takes a monomial of degree d and yields either the zero polynomial or a non-zero invariant of degree d. By applying it to several different monomials, a complete basis of R_d can be constructed. If G is a permutation group, a simplified version of the Reynolds operator can always be used which is independent of the field K (and thus whether we are in the modular case or not).

The linear algebra method works in both the modular and non-modular cases and, with appropriate modifications, also for linear algebraic groups. It simply finds a basis for R_d in one step – it is not possible to find a single invariant alone by this method.

Magma provides the function `InvariantsOfDegree` to automatically compute a basis of R_d by a default appropriate method – the method can also be selected by a parameter. The function `InvariantsOfDegree` can also be given a positive integer k which is less than or equal to the dimension of R_d: in such a case, only k linearly independent invariants are computed. See also the functions *MonomialsOfDegree* and *MonomialsOfWeightedDegree* in the Ideal Theory chapter.

ReynoldsOperator(f, G): RngMPolElt, GrpMat -> RngMPolElt

Given a polynomial f and a matrix group G such that G can act on f, return the application of the Reynolds operator of G to f. (f need not be a monomial but may be a non-homogeneous polynomial.)

InvariantsOfDegree(R, d): RngInvar, RngIntElt -> [RngMPolElt]

InvariantsOfDegree(G, d): GrpMat, RngIntElt -> [RngMPolElt]

InvariantsOfDegree(G, K, d): GrpPerm, Fld, RngIntElt -> [RngMPolElt]

InvariantsOfDegree(G, P, d): GrpMat, RngMPol, RngIntElt -> [RngMPolElt]

InvariantsOfDegree(G, P, d): GrpPerm, RngMPol, RngIntElt -> [RngMPolElt]

Invariants: MonStgElt Default: "Both"

Construct a K-basis of the space R_d of the homogeneous invariants of degree d in the invariant ring $R = K[V]^G$ of the group G over the field K as a sequence of polynomials. Either the invariant ring R, the group G (if a matrix group), or the group G (if a permutation group) together with the field K may be passed. A specific polynomial ring P compatible with G and K may be passed so that the returned invariants lie in P. The parameter **Invariants** may be supplied to select the method of the construction of the invariants: "Reynolds" (use the Reynolds operator), "Linear" (use the linear algebra method), or "Both" (use an appropriate combination of both methods). The default is "Both".

InvariantsOfDegree(R, d, k): RngInvar, RngIntElt, RngIntElt -> [RngMPolElt]

InvariantsOfDegree(G, d, k): GrpMat, RngIntElt, RngIntElt -> [RngMPolElt]

InvariantsOfDegree(G, K, d, k): GrpPerm, Fld, RngIntElt, RngIntElt -> [RngMPolElt]

InvariantsOfDegree(G, P, d, k): GrpPerm, RngMPol, RngIntElt, RngIntElt -> [RngMPolElt]

Invariants: MonStgElt Default: "Both"

Construct k linearly independent homogeneous invariants of degree d in the invariant ring $R = K[V]^G$ of the group G over the field K as a sequence of polynomials, where k must be greater than or equal to 1 and less than or equal to the dimension of the space R_d. Either the invariant ring R, the group G (if a matrix group), or the group G (if a permutation group) together with the field K may be passed. A specific polynomial ring P compatible with G and K may be passed so that the returned invariants lie in P. The parameter **Invariants** may be supplied to select the method of the construction of the invariants – see the last function.

Example: Reynolds Operator and Invariants of a Degree (ex-5b0d55)

We demonstrate elementary uses of **ReynoldsOperator** and **InvariantsOfDegree**.

```
> K<z> := CyclotomicField(5);
> w := -z^3 - z^2;
> G := MatrixGroup<3,K |
>    [1,0,-w, 0,0,-1, 0,1,-w],
>    [-1,-1,w, -w,0,w, -w,0,1]>;
> P<x1,x2,x3> := PolynomialRing(K, 3);
> time ReynoldsOperator(x1^4, G);
(-z^3 - z^2 + 1)*x1^4 + (12/5*z^3 + 12/5*z^2 -
    4/5)*x1^3*x2 + (12/5*z^3 + 12/5*z^2 - 4/5)*x1^3*x3
    + (-14/5*z^3 - 14/5*z^2 + 14/5)*x1^2*x2^2 +
    (4/5*z^3 + 4/5*z^2 + 4/5)*x1^2*x2*x3 + (-14/5*z^3 -
    14/5*z^2 + 14/5)*x1^2*x3^2 + (12/5*z^3 + 12/5*z^2 -
    4/5)*x1*x2^3 + (4/5*z^3 + 4/5*z^2 + 4/5)*x1*x2^2*x3
    + (4/5*z^3 + 4/5*z^2 + 4/5)*x1*x2*x3^2 + (12/5*z^3
    + 12/5*z^2 - 4/5)*x1*x3^3 + (-z^3 - z^2 + 1)*x2^4 +
    (12/5*z^3 + 12/5*z^2 - 4/5)*x2^3*x3 + (-14/5*z^3 -
    14/5*z^2 + 14/5)*x2^2*x3^2 + (12/5*z^3 + 12/5*z^2 -
    4/5)*x2*x3^3 + (-z^3 - z^2 + 1)*x3^4
Time: 0.090
> time I20_1 := InvariantsOfDegree(G, 20, 1);
0.259
> time I20 := InvariantsOfDegree(G, 20);
3.589
> [LeadingMonomial(f): f in I20];
[
```

(continues on next page)

(continued from previous page)

```
    x1^20,
    x1^18*x2^2,
    x1^16*x2^4,
    x1^15*x2^5,
    x1^14*x2^6,
    x1^13*x2^7,
    x1^12*x2^8
]
> G := CyclicGroup(4);
> K := GF(2);
> InvariantsOfDegree(G, K, 4);
[
    x1^4 + x2^4 + x3^4 + x4^4,
    x1^3*x2 + x1*x4^3 + x2^3*x3 + x3^3*x4,
    x1^3*x3 + x1*x3^3 + x2^3*x4 + x2*x4^3,
    x1^3*x4 + x1*x2^3 + x2*x3^3 + x3*x4^3,
    x1^2*x2^2 + x1^2*x4^2 + x2^2*x3^2 + x3^2*x4^2,
    x1^2*x2*x3 + x1*x2*x4^2 + x1*x3^2*x4 + x2^2*x3*x4,
    x1^2*x2*x4 + x1*x2^2*x3 + x1*x3*x4^2 + x2*x3^2*x4,
    x1^2*x3^2 + x2^2*x4^2,
    x1^2*x3*x4 + x1*x2^2*x4 + x1*x2*x3^2 + x2*x3*x4^2,
    x1*x2*x3*x4
]
```

`SetAllInvariantsOfDegree(R, d, Q): RngInvar, RngIntElt, [ RngMPolElt ]`

(Procedure.) Given an invariant ring $R = K[V]^G$, an integer $d \geq 0$, and a sequence Q consisting of k degree-d homogeneous invariants of G, set the internal list of all linearly-independent homogeneous invariants of degree d of R to be Q. Thus the elements of Q must describe a basis of the space of all homogeneous invariants of degree d of R. If the Hilbert Series of R is known, it will be used to check that the length of Q (the dimension of the basis) is correct.

Example: Setting All Invariants of a Degree `(ex-dd87c1)`

We demonstrate a simple use of `SetAllInvariantsOfDegree`.

```
> R := InvariantRing(CyclicGroup(4), GF(2));
> P<x1,x2,x3,x4> := PolynomialRing(R);
> L := [
>     x1^2 + x2^2 + x3^2 + x4^2,
>     x1*x2 + x1*x4 + x2*x3 + x3*x4,
>     x1*x3 + x2*x4
> ];
> SetAllInvariantsOfDegree(R, 2, L);
> InvariantsOfDegree(R, 2);
[
    x1^2 + x2^2 + x3^2 + x4^2,
    x1*x2 + x1*x4 + x2*x3 + x3*x4,
    x1*x3 + x2*x4
]
> PrimaryInvariants(R);
[
    x1 + x2 + x3 + x4,
    x1*x2 + x1*x4 + x2*x3 + x3*x4,
    x1*x3 + x2*x4,
    x1*x2*x3*x4
]
```

The following sections *Construction of G-modules* through *Properties of Invariant Rings* all deal with

invariant rings of finite groups.

120.9 Construction of G-modules

This section describes how one can create a finite-dimensional G-module corresponding to the action of a finite group G on a polynomial ring P. There are two ways one can create a finite-dimensional action: the action on the space of homogeneous polynomials of a fixed degree, or the action on the quotient space of polynomials by a zero-dimensional ideal (so the quotient has finite-dimension as a vector space). The functions in this section are also found in the chapter on general modules but are also included here since they are useful in Invariant Theory.

GModule(G, P, d): Grp, RngMPol, RngIntElt -> ModGrp, Map, {@ RngMPolElt @}

Given a finite permutation or matrix group G of degree n, a polynomial ring $P = K[x_1, \ldots, x_n]$ over a field K, and a non-negative integer d, create the $K[G]$-module M corresponding to the action of G on the space of homogeneous polynomials of degree d of the polynomial ring P. The function also returns the isomorphism f between the space of homogeneous polynomials of degree d of P and M, together with an indexed set of monomials of degree d of P which correspond to the columns of M.

GModule(G, I, J): Grp, RngMPol, RngMPol -> ModGrp, Map, {@ RngMPolElt @}

Given a finite permutation or matrix group G of degree n, an ideal I of a multivariate polynomial ring $P = K[x_1, \ldots, x_n]$ over a field K, and a zero-dimensional subideal J of I, create the $K[G]$-module M corresponding to the action of G on the finite-dimensional quotient I/J. The function also returns the isomorphism f between the quotient space I/J and M, together with an indexed set of monomials of P, forming a (vector space) basis of I/J, and which correspond to the columns of M.

GModule(G, Q): Grp, RngMPolRes -> ModGrp, Map, {@ RngMPolElt @}

Given a finite permutation or matrix group G of degree n, and a finite-dimensional quotient ring $Q = I/J$ of a multivariate polynomial ring $P = K[x_1, \ldots, x_n]$ over a field K, create the $K[G]$-module M corresponding to the action of G on the finite-dimensional quotient Q. The function also returns the isomorphism f between the quotient ring Q and M, together with an indexed set of monomials of P, forming a (vector space) basis of Q, and which correspond to the columns of M.

Example: Constructing a G-Module (ex-393820)

We demonstrate simple uses of the GModule function.

```
> q := 5;
> K := GF(q);
> G := GL(3, K);
> P<x, y, z> := PolynomialRing(K, 3);
> I := ideal< P | x^5 - x,y^5 - y, z^5 - z >;
> Q, rho := quo< P | I >;
> f := x^3 + x^2*y + y^3;
> M, phi:= GModule(G, P, I);
> Constituents(M);
[
    GModule of dimension 1 over GF(5),
    GModule of dimension 3 over GF(5),
    GModule of dimension 3 over GF(5),
    GModule of dimension 6 over GF(5),
    GModule of dimension 6 over GF(5),
    GModule of dimension 10 over GF(5),
    GModule of dimension 10 over GF(5),
    GModule of dimension 15 over GF(5),
    GModule of dimension 15 over GF(5),
```

(continues on next page)

(continued from previous page)

```
    GModule of dimension 18 over GF(5),
    GModule of dimension 18 over GF(5),
    GModule of dimension 19 over GF(5)
]
> N := sub<M | phi(f)>;
> N;
GModule N of dimension 10 over GF(5)
> M5 := GModule(G, P, 5);
> M5;
GModule M5 of dimension 21 over GF(5)
> Constituents(M5);
[
    GModule of dimension 3 over GF(5),
    GModule of dimension 18 over GF(5)
]
```

120.10 Molien Series

Let $R = K[V]^G$ be the invariant ring of the finite group G over the field K. If G is a finite matrix group in the non-modular case or a permutation group (in *either* the modular or non-modular case) then the Molien series of G yields the Hilbert Series of R.

`MolienSeries(G): GrpMat -> FldFunUElt`

`MolienSeries(G): GrpPerm -> FldFunUElt`

The Molien series of G, returned as an element of the rational function field $\mathbb{Z}(t)$. If G is a permutation group, the Molien series always exists and equals the Hilbert series of the invariant ring of G for any field. If G is a matrix group, the characteristic of the coefficient field of G must be coprime with the order of G.

`MolienSeriesApproximation(G, n): GrpPerm, RngIntElt -> RngSerLaurElt`

The Molien series of a permutation group G, or more precisely, an approximation to it, as a Laurent series with n known coefficients. In contrast to the `MolienSeries` function above, approximations can be computed for far larger groups.

Example: Molien Series of a Matrix Group `(ex-cbf09f)`

We compute the Molien series of a matrix G and verify that the coefficients of the corresponding power series match the number of independent invariants for each degree.

```
> K<z> := CyclotomicField(5);
> w := -z^3 - z^2;
> G := MatrixGroup<3,K |
>    [1,0,-w, 0,0,-1, 0,1,-w],
>    [-1,-1,w, -w,0,w, -w,0,1]>;
> M<t> := MolienSeries(G);
> M;
(-t^8 - t^7 + t^5 + t^4 + t^3 - t - 1)/(t^11 + t^10 -
    t^9 - 2*t^8 - t^7 + t^4 + 2*t^3 + t^2 - t - 1)
> P<u> := PowerSeriesRing(IntegerRing());
> P ! M;
1 + u^2 + u^4 + 2*u^6 + 2*u^8 + 3*u^10 + 4*u^12 +
    4*u^14 + u^15 + 5*u^16 + u^17 + 6*u^18 + u^19 +
    O(u^20)
> Coefficients(P ! M);
[ 1, 0, 1, 0, 1, 0, 2, 0, 2, 0, 3, 0, 4, 0, 4, 1, 5, 1,
6, 1 ]
```

(continues on next page)

(continued from previous page)

```
> time [#InvariantsOfDegree(G, i): i in [0 .. 19]];
[ 1, 0, 1, 0, 1, 0, 2, 0, 2, 0, 3, 0, 4, 0, 4, 1, 5, 1,
6, 1 ]
```

120.11 Primary Invariants

Let $R = K[V]^G$ be the invariant ring of a finite group G over the field K and suppose the degree of G is n. A set of *primary invariants* of R is a set $\{f_1, \ldots, f_n\}$ of n algebraically independent homogeneous invariants of R such that the invariant ring R is a finitely generated module over $A = K[f_1, \ldots, f_n]$. A set of primary invariants always exists for any invariant ring R. The invocation `PrimaryInvariants(R)` allows automatic construction of primary invariants of R. The primary invariants are stored in R and recalled as necessary in subsequent computations. The latest algorithm in Magma to compute primary invariants, due to G. Kemper [Kem99], now guarantees that the degrees of the primary invariants found by the algorithm are optimal (with respect to their product and then their sum).

`PrimaryInvariants(R): RngInvar -> [ RngMPolElt ]`

Construct optimal primary invariants for the invariant ring $R = K[V]^G$ as a sorted sequence (with increasing degrees) of n polynomials of R where n is the degree of G.

Example: Primary Invariants of an A5 Representation `(ex-93d2b8)`

We compute primary invariants for the "first A_5 in $\mathrm{SL}(\mathbf{F}_2)$", discussed in [AM94, p. 116]. The resulting degrees $3, 5, 8$, and 12 are necessarily optimal (see [Kem96]).

```
> K := GF(2);
> G := MatrixGroup<4, K |
>     [0,1,0,0, 1,1,0,0, 0,0,1,1, 0,0,1,0],
>     [1,0,0,0, 0,1,0,0, 1,0,1,0, 0,1,0,1],
>     [1,0,1,0, 0,1,0,1, 0,0,1,0, 0,0,0,1]>;
> R := InvariantRing(G);
> time p := PrimaryInvariants(R);
Time: 1.399
> [TotalDegree(f): f in p];
[ 3, 5, 8, 12 ]
```

120.12 Secondary Invariants

Let $R = K[V]^G$ be the invariant ring of a finite group G over the field K and suppose the degree of G is n. If $\{f_1, \ldots, f_n\}$ is a set of primary invariants for R then R can be viewed as a finitely generated module over the algebra $A = K[f_1, \ldots, f_n]$. A set of *secondary invariants* for R with respect to these primary invariants is set of module generators over A. The invocation `SecondaryInvariants(R)` allows automatic construction of secondary invariants of R. The secondary invariants are stored in R and recalled as necessary in subsequent computations. Different algorithms are needed for the modular and non-modular cases – see [KS97] for details.

`SecondaryInvariants(R): RngInvar -> [ RngMPolElt ]`

Construct secondary invariants for the invariant ring $R = K[V]^G$ (with respect to the current primary invariants of R, constructed automatically first if necessary) as a sorted sequence (with increasing degrees) of polynomials of R. The secondary invariants are *minimal*; i.e. they are a minimal generating set for R considered as a module over the algebra generated by the primary invariants.

`SecondaryInvariants(R, H): RngInvar, Grp -> [ RngMPolElt ]`

Construct secondary invariants for the *modular* invariant ring $R = K[V]^G$ (with respect to the current primary invariants of R), using the subgroup H. This function can only be used if R is a modular invariant ring. H must be a subgroup of the group G; first, secondary invariants are

computed for $K[V]^H$ using the current primary invariants for G and then these secondary invariants are used in the manner described in [KS97]. The function `SecondaryInvariants(R)` (taking just the invariant ring R) follows a default strategy in which it tries to use this function with the best subgroup H appropriate. Thus usually using this function to specify a particular subgroup is not more helpful than the one-argument function but occasionally it may be.

`IrreducibleSecondaryInvariants(R): RngInvar -> [ RngMPolElt ]`

Return the irreducible secondary invariants of the invariant ring $R = K[V]^G$ (with respect to the current primary invariants of R, constructed automatically first if necessary) as a sequence of polynomials of R. These, together with the primary invariants of R, generate R as an algebra over K. In the modular case, these will be the same as the secondary invariants of R (excluding the polynomial 1) but in the non-modular case they may form a proper subsequence of the secondary invariants. Note that the expression of the secondary invariants in terms of the irreducible secondary invariants is given as the second return value of the function `Algebra` (see the section on the algebra of an invariant ring and algebraic relations below).

Example: Secondary Invariants of a Cyclic Group (ex-4cf4d4)

We construct primary and then secondary invariants for the invariant ring R of the group G over $\mathbf{F}_2$, where G is the (permutation) cyclic group of order 4. Note that in this example Noether's degree bound (which holds for characteristic 0) is violated.

```
> K := GF(2);
> G := CyclicGroup(4);
> R := InvariantRing(G, K);
> time PrimaryInvariants(R);
[
    x1 + x2 + x3 + x4,
    x1*x2 + x1*x4 + x2*x3 + x3*x4,
    x1*x3 + x2*x4,
    x1*x2*x3*x4
]
Time: 0.040
> time SecondaryInvariants(R);
[
    1,
    x1*x2*x3 + x1*x2*x4 + x1*x3*x4 + x2*x3*x4,
    x1^2*x3 + x1^2*x4 + x1*x2^2 + x1*x3^2 + x2^2*x4 +
        x2*x3^2 + x2*x4^2 + x3*x4^2,
    x1^2*x3^2 + x1^2*x3*x4 + x1*x2^2*x4 + x1*x2*x3^2 +
        x2^2*x4^2 + x2*x3*x4^2,
    x1^3*x3*x4 + x1^2*x2^2*x3 + x1^2*x2^2*x4 +
        x1^2*x2*x3^2 + x1^2*x2*x3*x4 + x1^2*x2*x4^2 +
        x1^2*x3^2*x4 + x1^2*x3*x4^2 + x1*x2^3*x4 +
        x1*x2^2*x3^2 + x1*x2^2*x3*x4 + x1*x2^2*x4^2 +
        x1*x2*x3^3 + x1*x2*x3^2*x4 + x1*x2*x3*x4^2 +
        x1*x3^2*x4^2 + x2^2*x3^2*x4 + x2^2*x3*x4^2 +
        x2*x3^2*x4^2 + x2*x3*x4^3
]
Time: 0.080
```

120.13 Fundamental Invariants

Let $R = K[V]^G$ be the invariant ring of the group G over the field K and suppose the degree of G is n. A set of *fundamental invariants* for R is a generating set of R as an algebra over K.

`FundamentalInvariants(R): RngInvar -> [ RngMPolElt ]`

```
Al       : MonStgElt                    Default: "King"
MaxDegree: RngIntElt                    Default: 0
```

Construct fundamental invariants for the invariant ring $R = K[V]^G$ as a sorted sequence (with increasing degrees) of polynomials of R. As of V2.15, if R is non-modular, then by default the fundamental invariants are computed via the algorithm of S.King [Kin07]; the alternative algorithm (always used in the modular case), which computes the fundamental invariants by minimalizing the union of the primary and secondary invariants of R, may be selected by setting the parameter `Al` to `"MinPrimSec"`. If the fundamental invariants are known to be bounded by degree d, then the parameter `MaxDegree` may be set to d to assist the King algorithm with an early stopping condition in the non-modular case.

Example: Fundamental Invariants of a Permutation Group (ex-4519e0)

We construct fundamental invariants for the invariant ring R of the group G over $\mathbb{Q}$, where G is permutation group consisting of two parallel copies of S_3 in degree 6. Notice that the sequence of fundamental invariants is shorter and simpler than the sequence consisting of the primary invariants combined with the secondary invariants.

```
> K := RationalField();
> G := PermutationGroup<6 | (1,2,3)(4,5,6), (1,2)(4,5)>;
> R := InvariantRing(G, K);
> PrimaryInvariants(R);
[
    x1 + x2 + x3,
    x4 + x5 + x6,
    x1^2 + x2^2 + x3^2,
    x4^2 + x5^2 + x6^2,
    x1^3 + x2^3 + x3^3,
    x4^3 + x5^3 + x6^3
]
> SecondaryInvariants(R);
[
    1,
    x1*x4 + x2*x5 + x3*x6,
    x1^2*x4 + x2^2*x5 + x3^2*x6,
    x1*x4^2 + x2*x5^2 + x3*x6^2,
    x1^2*x4^2 + 2*x1*x2*x4*x5 + 2*x1*x3*x4*x6 + x2^2*x5^2 + 2*x2*x3*x5*x6 +
        x3^2*x6^2,
    x1^3*x4^3 + x1^2*x2*x4*x5^2 + x1^2*x3*x4*x6^2 + x1*x2^2*x4^2*x5 +
        x1*x3^2*x4^2*x6 + x2^3*x5^3 + x2^2*x3*x5*x6^2 + x2*x3^2*x5^2*x6 +
        x3^3*x6^3
]
> FundamentalInvariants(R);
[
    x1 + x2 + x3,
    x4 + x5 + x6,
    x1^2 + x2^2 + x3^2,
    x1*x4 + x2*x5 + x3*x6,
    x4^2 + x5^2 + x6^2,
    x1^3 + x2^3 + x3^3,
    x1^2*x4 + x2^2*x5 + x3^2*x6,
    x1*x4^2 + x2*x5^2 + x3*x6^2,
    x4^3 + x5^3 + x6^3
]
```

Example: Transitive Groups of Degree 7 (ex-ea0676)

As in [Kin07], we compute fundamental invariants for the invariant rings for all transitive groups of

degree 7 (in characteristic zero). For each group, we print its order and a summary of the degrees (where the i-th element of the sequence gives the number of fundamental invariants of degree i).

```
> function deg_summary(B)
>     degs := [TotalDegree(f): f in B];
>     return [#[j: j in degs | j eq d]: d in [1 .. Max(degs)]];
> end function;
>
> d := 7;
> time for i := 1 to NumberOfTransitiveGroups(d) do
>     G := TransitiveGroup(d, i);
>     R := InvariantRing(G, RationalField());
>     F := FundamentalInvariants(R);
>     printf "%o: Order: %o, Degrees: %o\n", i, #G, deg_summary(F);
> end for;
1: Order: 7, Degrees: [ 1, 3, 8, 12, 12, 6, 6 ]
2: Order: 14, Degrees: [ 1, 3, 4, 6, 6, 3, 3 ]
3: Order: 21, Degrees: [ 1, 1, 4, 5, 8, 8, 6 ]
4: Order: 42, Degrees: [ 1, 1, 2, 3, 4, 7, 7, 5, 1 ]
5: Order: 168, Degrees: [ 1, 1, 2, 2, 2, 2, 2 ]
6: Order: 2520, Degrees: [ 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 ]
7: Order: 5040, Degrees: [ 1, 1, 1, 1, 1, 1, 1 ]
Time: 1.610
```

Instead of computing over the rational field, for each group G we can instead compute over $\mathbf{F}_p$, where p is the smallest prime which does not divide the order of G. This is faster, and it is conjectured that the resulting degrees are always the same as for the computation over the rationals.

```
> d := 7;
> time for i := 1 to NumberOfTransitiveGroups(d) do
>     G := TransitiveGroup(d, i);
>     p := rep{p: p in [2 .. #G] | IsPrime(p) and #G mod p ne 0};
>     R := InvariantRing(G, GF(p));
>     F := FundamentalInvariants(R);
>     printf "%o: Order: %o, Degrees: %o\n", i, #G, deg_summary(F);
> end for;
1: Order: 7, Degrees: [ 1, 3, 8, 12, 12, 6, 6 ]
2: Order: 14, Degrees: [ 1, 3, 4, 6, 6, 3, 3 ]
3: Order: 21, Degrees: [ 1, 1, 4, 5, 8, 8, 6 ]
4: Order: 42, Degrees: [ 1, 1, 2, 3, 4, 7, 7, 5, 1 ]
5: Order: 168, Degrees: [ 1, 1, 2, 2, 2, 2, 2 ]
6: Order: 2520, Degrees: [ 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 ]
7: Order: 5040, Degrees: [ 1, 1, 1, 1, 1, 1, 1 ]
Time: 0.790
```

Finally, we can do the same for all transitive groups of degree 8 in about 2 minutes.

```
> d := 8;
> time for i := 1 to NumberOfTransitiveGroups(d) do
>     G := TransitiveGroup(d, i);
>     p := rep{p: p in [2 .. #G] | IsPrime(p) and #G mod p ne 0};
>     R := InvariantRing(G, GF(p));
>     F := FundamentalInvariants(R);
>     printf "%o: Order: %o, Degrees: %o\n", i, #G, deg_summary(F);
> end for;
1: Order: 8, Degrees: [ 1, 4, 10, 18, 16, 8, 4, 4 ]
```

(continues on next page)

(continued from previous page)

```
2: Order: 8, Degrees: [ 1, 5, 9, 16, 8 ]
3: Order: 8, Degrees: [ 1, 7, 7, 7 ]
4: Order: 8, Degrees: [ 1, 6, 8, 12, 5 ]
5: Order: 8, Degrees: [ 1, 4, 10, 19, 15, 7 ]
6: Order: 16, Degrees: [ 1, 4, 5, 9, 8, 4, 2, 2 ]
7: Order: 16, Degrees: [ 1, 3, 7, 12, 13, 9, 4, 4 ]
8: Order: 16, Degrees: [ 1, 3, 6, 11, 12, 7, 2, 2 ]
9: Order: 16, Degrees: [ 1, 5, 5, 8, 4 ]
10: Order: 16, Degrees: [ 1, 4, 6, 11, 7, 2 ]
11: Order: 16, Degrees: [ 1, 4, 6, 11, 7, 3 ]
12: Order: 24, Degrees: [ 1, 2, 4, 8, 11, 12, 7 ]
13: Order: 24, Degrees: [ 1, 3, 3, 7, 8, 11, 7 ]
14: Order: 24, Degrees: [ 1, 3, 3, 8, 7, 9, 6, 1, 1 ]
15: Order: 32, Degrees: [ 1, 3, 4, 7, 6, 4, 2, 2 ]
16: Order: 32, Degrees: [ 1, 3, 5, 8, 7, 7, 4, 4 ]
17: Order: 32, Degrees: [ 1, 3, 4, 7, 6, 4, 2, 2 ]
18: Order: 32, Degrees: [ 1, 4, 4, 7, 3 ]
19: Order: 32, Degrees: [ 1, 3, 3, 7, 6, 7, 5, 1 ]
20: Order: 32, Degrees: [ 1, 3, 5, 9, 6, 4, 2, 1 ]
21: Order: 32, Degrees: [ 1, 4, 4, 6, 4, 3, 2, 1 ]
22: Order: 32, Degrees: [ 1, 4, 4, 7, 3, 1 ]
23: Order: 48, Degrees: [ 1, 2, 3, 5, 6, 6, 5, 2 ]
24: Order: 48, Degrees: [ 1, 3, 3, 6, 4, 3, 1 ]
25: Order: 56, Degrees: [ 1, 1, 1, 4, 6, 13, 18, 23, 18, 6 ]
26: Order: 64, Degrees: [ 1, 3, 3, 5, 3, 3, 2, 3, 1 ]
27: Order: 64, Degrees: [ 1, 3, 5, 8, 6, 4, 2, 2 ]
28: Order: 64, Degrees: [ 1, 3, 3, 5, 4, 4, 2, 2 ]
29: Order: 64, Degrees: [ 1, 3, 3, 6, 3, 2, 1 ]
30: Order: 64, Degrees: [ 1, 3, 3, 5, 3, 2, 3, 4, 3, 2, 1, 1 ]
31: Order: 64, Degrees: [ 1, 4, 4, 6, 3, 1 ]
32: Order: 96, Degrees: [ 1, 2, 2, 4, 3, 5, 4, 2, 2, 1, 1, 1 ]
33: Order: 96, Degrees: [ 1, 2, 2, 4, 3, 6, 5, 5, 3 ]
34: Order: 96, Degrees: [ 1, 2, 2, 5, 2, 5, 4, 3, 3 ]
35: Order: 128, Degrees: [ 1, 3, 3, 5, 3, 2, 1, 1 ]
36: Order: 168, Degrees: [ 1, 1, 1, 2, 2, 5, 6, 8, 10, 11, 8 ]
37: Order: 168, Degrees: [ 1, 1, 1, 3, 1, 5, 5, 8, 9, 9, 7 ]
38: Order: 192, Degrees: [ 1, 2, 2, 3, 3, 5, 4, 3, 2, 1, 1, 1 ]
39: Order: 192, Degrees: [ 1, 2, 2, 4, 2, 2, 1 ]
40: Order: 192, Degrees: [ 1, 2, 2, 3, 2, 2, 1, 1, 1, 3, 3, 2, 2, 1, 1, 1 ]
41: Order: 192, Degrees: [ 1, 2, 2, 4, 2, 3, 2, 2, 1 ]
42: Order: 288, Degrees: [ 1, 2, 2, 3, 2, 3, 2, 2, 1, 1 ]
43: Order: 336, Degrees: [ 1, 1, 1, 2, 1, 3, 3, 5, 4, 6, 5, 4, 2 ]
44: Order: 384, Degrees: [ 1, 2, 2, 3, 2, 2, 1, 1 ]
45: Order: 576, Degrees: [ 1, 2, 2, 3, 2, 2, 1, 1, 0, 0, 0, 1 ]
46: Order: 576, Degrees: [ 1, 2, 2, 3, 2, 2, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1 ]
47: Order: 1152, Degrees: [ 1, 2, 2, 3, 2, 2, 1, 1 ]
48: Order: 1344, Degrees: [ 1, 1, 1, 2, 1, 2, 2, 2, 1, 1 ]
49: Order: 20160, Degrees: [ 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 ]
50: Order: 40320, Degrees: [ 1, 1, 1, 1, 1, 1, 1, 1 ]
Time: 128.030
```

Example: Degree-10 Representation of S5 `(ex-b8dda0)`

We compute fundamental invariants of a degree-10 representation of S_5 acting on pairs. See [Kin07, p.11--12]. First we compute the fundamental invariants mod 7 of the permutation representation (very difficult in practice hitherto).

```
> G := PermutationGroup<10 | (2,5)(3,6)(4,7),(1,5,8,10,4)(2,6,9,3,7)>;
> #G;
120
> R := InvariantRing(G, GF(7));
> time F := FundamentalInvariants(R);
Time: 29.310
> {* Degree(f): f in F *};
{* 1, 2^^2, 3^^4, 4^^7, 5^^10, 6^^13, 7^^13, 8^^4, 9^^2 *}
```

Finally, we can compute a matrix representation of G as a direct sum of irreducible representations of degrees 1, 4, and 5. We then compute the fundamental invariants of the invariant ring of this representation mod 7.

```
> Q := RationalField();
> R0 := InvariantRing(G, Q);
> P0 := PolynomialRing(R0);
> M := GModule(G, Q);
> Gl := MatrixGroup(M);
> C := CharacterTable(Gl);
> Pi := [&+[Q!Integers()!c(g)*MatrixAlgebra(Q, 10)!g: g in Gl]/#G: c in C];
> Pi := [p: p in Pi | p ne 0];
> L := [sub<M | Image(p)>: p in Pi];
> G := MatrixGroup(DirectSum(DirectSum(L[1],L[2]),L[3]));
> G;
MatrixGroup(10, Rational Field)
Generators:
    [   1    0    0    0    0    0    0    0    0    0]
    [   0    1  1/3  1/3  1/3    0    0    0    0    0]
    [   0    0  1/3 -2/3 -2/3    0    0    0    0    0]
    [   0    0 -2/3  1/3 -2/3    0    0    0    0    0]
    [   0    0 -2/3 -2/3  1/3    0    0    0    0    0]
    [   0    0    0    0    0    1    0    0    0    0]
    [   0    0    0    0    0    0    0    0    1    0]
    [   0    0    0    0    0    0    0    0    0    1]
    [   0    0    0    0    0    0    1    0    0    0]
    [   0    0    0    0    0    0    0    1    0    0]

    [   1    0    0    0    0    0    0    0    0    0]
    [   0    0  1/3 -2/3 -2/3    0    0    0    0    0]
    [   0    0 -2/3  1/3 -2/3    0    0    0    0    0]
    [   0    0 -2/3 -2/3  1/3    0    0    0    0    0]
    [   0    1  1/3  1/3  1/3    0    0    0    0    0]
    [   0    0    0    0    0   -1   -1    1    1    0]
    [   0    0    0    0    0   -1    0    0    0    1]
    [   0    0    0    0    0   -1    0    1    0    0]
    [   0    0    0    0    0    0   -1    0    0    0]
    [   0    0    0    0    0    0   -1    1    0    0]
> Gp := ChangeRing(G, GF(7));
> #Gp;
120
> Rp := InvariantRing(Gp);
> time Fp := FundamentalInvariants(Rp);
Time: 35.380
> {* Degree(f): f in Fp *};
{* 1, 2^^2, 3^^4, 4^^7, 5^^10, 6^^13, 7^^13, 8^^4, 9^^2 *}
> [Degree(f): f in F] eq [Degree(f): f in Fp];
true
```

120.14 The Module of an Invariant Ring

Let $R = K[V]^G$ be the invariant ring of a finite group G over the field K and suppose the degree of G is n. Suppose also that primary invariants $\{f_1, \ldots, f_n\}$ for R have been constructed, together with minimal secondary invariants $S = \{g_1, \ldots, g_m\}$ for R with respect to these primary invariants. (These secondary invariants may possess non-trivial module syzygies.) Then R can be considered as a module over the algebra $A = K[f_1, \ldots, f_n]$ with the minimal (module) generating set S. To compute with this module structure of R easily, Magma automatically constructs the graded multivariate polynomial algebra $A' = K[t_1, \ldots, t_n]$ (with the weighted degree of the variable t_i defined to be the degree of f_i) which is isomorphic to A, and then constructs the graded module $M = A'^m/Q$ over A' with the quotient relations Q given by the syzygies of the g_i (and with the weighted degree of column i equal to the degree of g_i). The algebra A' is isomorphic to A under the map $t_i \mapsto f_i$, and the module M is isomorphic to R (considered as a module) under the map $M.i \mapsto g_i$ (extended by the isomorphism from A' onto A). (See the chapter on modules over $K[x_1, \ldots, x_n]$ for details on how to compute with the module M and an explanation of quotient relations, the unit vectors $M.i$, etc.) Once the module M is created, together with the isomorphism $f : R \to M$, one can apply f to a general element h of R to obtain the element of M corresponding to h. This effectively yields a representation of h as a sum $\sum_{i=1} ka_i g_i$ with $a_i \in A$ in terms of the primary and secondary invariants. This representation is also unique up to the relations given by the syzygies of the g_i. When creating the module M, the coefficient ring A' of M is assigned the print names `"t1", "t2"`, etc. – the angle bracket notation or the . operator should be used to assign the variables of A' to actual Magma variables.

`Module(R): RngInvar -> ModMPol, Map`

The module M isomorphic to $R = K[V]^G$, together with the isomorphism $f : R \to M$.

Example: Module of an Invariant Ring (ex-822234)

We create the module M corresponding to the invariant ring R of the group G generated by the 4 by 4 Jordan block over $\mathbf{F}_3$.

```
> K := GF(3);
> G := MatrixGroup<4,K | [1,0,0,0, 1,1,0,0, 0,1,1,0, 0,0,1,1]>;
> R := InvariantRing(G);
> P<x1,x2,x3,x4> := PolynomialRing(R);
> p := PrimaryInvariants(R);
> s := SecondaryInvariants(R);
> [TotalDegree(f): f in p];
[ 1, 2, 3, 9 ]
> [TotalDegree(f): f in s];
[ 0, 3, 4, 5, 6, 7, 8, 9 ]
> M, f := Module(R);
> M;
Full Quotient Module of degree 8
TOP Order
Column weights: 0 3 4 5 6 7 8 9
Coefficient ring:
    Graded Polynomial ring of rank 4 over GF(3)
    Lexicographical Order
    Variables: t1, t2, t3, t4
    Variable weights: 1 2 3 9
Quotient Relations:
[
    t1[7] + 2*t2[6] + t3[5],
    t1[4] + 2*t2[3] + t3[2]
]
> h := x1^5*x2 + 2*x1^3*x3^3 + 2*x2^6;
> h;
x1^5*x2 + 2*x1^3*x3^3 + 2*x2^6
```

(continues on next page)

(continued from previous page)

```
> m := f(h);
> m;
t1^4*t2[1] + t1^3[2] + t2^3[1]
> // Evaluate in the primaries and secondaries:
> p[1]^4*p[2]*s[1] + p[1]^3*s[2] + p[2]^3*s[1];
x1^5*x2 + 2*x1^3*x3^3 + 2*x2^6
```

120.15 The Algebra of an Invariant Ring and Algebraic Relations

Let $R = K[V]^G$ be the invariant ring of a finite group G over the field K and suppose the degree of G is n. Suppose also that primary invariants $\{f_1, \ldots, f_n\}$ for R have been constructed, together with minimal secondary invariants $S = \{g_1, \ldots, g_m\}$ for R with respect to these primary invariants. Suppose also that the irreducible secondary invariants for R are $S = \{h_1, \ldots, h_r\}$ so that the g_i are power products of the h_i. We write $g_i = p_i(h_i)$ where the p_i are monomials of the indeterminates $t_1, \ldots, t_r$. Then R is generated as an algebra over K by the primary invariants $f_1, \ldots, f_n$ and the irreducible secondary invariants $h_1, \ldots, h_r$. Magma allows the construction of a polynomial algebra A with indeterminate names `"f1"`, `"f2"`, etc. corresponding to the primary invariants and indeterminate names `"h1"`, `"h2"`, etc. corresponding to the irreducible secondary invariants. Thus R can be regarded as an homomorphic image of A and finding the algebraic relations between these (algebra) generators of R yields a presentation of R as a quotient of a polynomial algebra. The functions in this section construct the algebra A and the algebraic relations for R. When creating the algebra A, the algebra A is assigned the print names `"f1", "f2", "h1", "h2"`, etc. – the angle bracket notation or the . operator should be used to assign the variables of A to actual Magma variables.

`Algebra(R): RngInvar -> RngMPol, [ RngMPolElt ]`

Given an invariant ring $R = K[V]^G$, return the polynomial algebra $A = K[f_1, \ldots, f_n, h_1, \ldots, h_r]$ of which R is an homomorphic image. This function also returns a sequence Q giving the secondary invariants in terms of the irreducible secondary invariants as monomials in A. Thus $Q[i]$ is the monomial $p_i(t_i)$ mentioned in the introduction to this section. Note that the secondary invariant 1 is *not* an irreducible secondary invariant so no h-variable corresponds to it (the polynomial 1 in A simply corresponds to it).

`Relations(R): RngInvar -> [ RngMPolElt ]`

Given an invariant ring $R = K[V]^G$, return a (sorted) sequence L giving the algebraic relations amongst the algebra generators of R as elements of the algebra A corresponding to R. Thus R is isomorphic as an algebra (or ring) to the quotient of A by the ideal of A generated by the relations in L.

`RelationIdeal(R): RngInvar -> RngMPol`

Given an invariant ring $R = K[V]^G$, return the ideal of algebraic relations corresponding to R. This is simply the same as taking the ideal generated by the algebra A by the sequence L returned by the function `Relations(R)`.

`PrimaryAlgebra(R): RngInvar -> RngMPol`

Given an invariant ring $R = K[V]^G$, return the algebra corresponding to the primary invariants of R as a graded polynomial ring (with the weights corresponding to the degrees of the primary invariants).

`PrimaryIdeal(R): RngInvar -> RngMPol`

Given an invariant ring $R = K[V]^G$, return the ideal generated by the primary invariants of R (this is stored in R).

Example: Algebraic Relations of an Invariant Ring `(ex-899101)`

We create the invariant ring $R = K[V]^G$ where G is a degree-6 permutation representation of the direct product $C_3 \times C_3$ of two cyclic groups both of order 3 and K is the rational field. We construct the algebra A and the sequence Q giving the secondary invariants in terms of the

irreducible secondary invariants. We then note that the degree-6 secondary invariant is obtained as the product of two degree-3 irreducible secondary invariants. We then construct the list L of algebraic relations in A for R. Thus R is isomorphic to the quotient ring $A/ < L >$. We then construct an homomorphism h from A onto R and check that the relations in L are correct. Finally, we check that the Hilbert series of the (quotient by the) ideal of A generated by L is the same as the Hilbert series of R as expected.

```
> G := PermutationGroup<6 | (1, 2, 3), (4, 5, 6)>;
> R := InvariantRing(G, RationalField());
> P := PrimaryInvariants(R);
> P;
[
    x1 + x2 + x3,
    x4 + x5 + x6,
    x1^2 + x2^2 + x3^2,
    x4^2 + x5^2 + x6^2,
    x1^3 + x2^3 + x3^3,
    x4^3 + x5^3 + x6^3
]
> S := SecondaryInvariants(R);
> S;
[
    1,
    x1^2*x2 + x1*x3^2 + x2^2*x3,
    x4^2*x5 + x4*x6^2 + x5^2*x6,
    x1^2*x2*x4^2*x5 + x1^2*x2*x4*x6^2 + x1^2*x2*x5^2*x6 +
        x1*x3^2*x4^2*x5 + x1*x3^2*x4*x6^2 + x1*x3^2*x5^2*x6 +
        x2^2*x3*x4^2*x5 + x2^2*x3*x4*x6^2 + x2^2*x3*x5^2*x6
]
> H := IrreducibleSecondaryInvariants(R);
> H;
[
    x1^2*x2 + x1*x3^2 + x2^2*x3,
    x4^2*x5 + x4*x6^2 + x5^2*x6
]
> A, Q := Algebra(R);
> A;
Graded Polynomial ring of rank 8 over Rational Field
Lexicographical Order
Variables: f1, f2, f3, f4, f5, f6, h1, h2
Variable weights: 1 1 2 2 3 3 3 3
> Q;
[
    1,
    h1,
    h2,
    h1*h2
]
> // Thus S[4] must be H[1]*H[2]:
> S[4];
x1^2*x2*x4^2*x5 + x1^2*x2*x4*x6^2 + x1^2*x2*x5^2*x6 +
    x1*x3^2*x4^2*x5 + x1*x3^2*x4*x6^2 + x1*x3^2*x5^2*x6 +
    x2^2*x3*x4^2*x5 + x2^2*x3*x4*x6^2 + x2^2*x3*x5^2*x6
> H[1];
x1^2*x2 + x1*x3^2 + x2^2*x3
> H[2];
x4^2*x5 + x4*x6^2 + x5^2*x6
```

(continues on next page)

(continued from previous page)

```
> H[1]*H[2] eq S[4];
true
> L := Relations(R);
> L;
[
    -1/24*f1^6 + 3/8*f1^4*f3 - 1/3*f1^3*f5 - 9/8*f1^2*f3^2 +
        2*f1*f3*f5 + f1*f3*h1 + 1/8*f3^3 - f5^2 - f5*h1 - h1^2,
    -1/24*f2^6 + 3/8*f2^4*f4 - 1/3*f2^3*f6 - 9/8*f2^2*f4^2 +
        2*f2*f4*f6 + f2*f4*h2 + 1/8*f4^3 - f6^2 - f6*h2 - h2^2
]
> // Construct homomorphism h from A onto (polynomial ring of) R:
> h := hom<A -> PolynomialRing(R) | P cat H>;
> // Check images of L under h are zero so that elements of L are relations:
> h(L);
[
    0,
    0
]
> // Create relation ideal and check its Hilbert series equals that of R:
> I := RelationIdeal(R);
> I;
Ideal of Graded Polynomial ring of rank 8 over Rational Field
Lexicographical Order
Variables: f1, f2, f3, f4, f5, f6, h1, h2
Variable weights: 1 1 2 2 3 3 3 3
Basis:
[
    f1^6 - 9*f1^4*f3 + 8*f1^3*f5 + 27*f1^2*f3^2 - 48*f1*f3*f5 -
        24*f1*f3*h1 - 3*f3^3 + 24*f5^2 + 24*f5*h1 + 24*h1^2,
    f2^6 - 9*f2^4*f4 + 8*f2^3*f6 + 27*f2^2*f4^2 - 48*f2*f4*f6 -
        24*f2*f4*h2 - 3*f4^3 + 24*f6^2 + 24*f6*h2 + 24*h2^2
]
> HilbertSeries(I);
(t^4 - 2*t^3 + 3*t^2 - 2*t + 1)/(t^10 - 4*t^9 + 6*t^8 - 6*t^7 +
    9*t^6 - 12*t^5 + 9*t^4 - 6*t^3 + 6*t^2 - 4*t + 1)
> HilbertSeries(I) eq HilbertSeries(R);
true
```

120.16 Properties of Invariant Rings

The following functions return non-trivial structural properties of invariant rings of finite groups.

`HilbertSeries(R): RngInvar -> FldFunUElt`

The Hilbert series of the invariant ring $R = K[V]^G$, returned as an element of the rational function field $\mathbb{Z}(t)$. The Molien series of G will be used if possible; otherwise (the modular matrix group case) secondary invariants for R will be constructed to determine the result.

`HilbertSeriesApproximation(R, n): RngInvar, RngIntElt -> RngSerLaurElt`

The Hilbert series of the invariant ring $R = K[V]^G$, returned as a Laurent series with n known terms. The conjugacy classes of G will be used to compute the approximation.

`IsCohenMacaulay(R): RngInvar -> BoolElt`

Given the invariant ring $R = K[V]^G$ of the group G over the field K, return `true` iff R is Cohen-Macaulay. This is always true in the non-modular case. Otherwise, secondary invariants for R will be constructed to determine the result.

`FreeResolution(R): RngInvar -> [ ModMPol ]`

Given the invariant ring $R = K[V]^G$ of the group G over the field K, return a free resolution of (the module of) R. This is just the same as the invocation `FreeResolution(Module(R))`. The free resolution is returned as a sequence F such that $F[1]$ is M, $F[i+1]$ is the syzygy module of $F[i]$ for $i < \#F$, and the last element of F is free (its basis has no syzygies).

`MinimalFreeResolution(R): RngInvar -> [ ModMPol ]`

Given the invariant ring $R = K[V]^G$ of the group G over the field K, return a minimal free resolution of (the module of) R. This is just the same as the invocation `MinimalFreeResolution(Module(R))`.

`HomologicalDimension(R): RngInvar -> RngInt`

Given the invariant ring $R = K[V]^G$ of the group G over the field K, return the homological dimension of R. This is just the length of a minimal free resolution of R minus 1 (taking account of the fact that the module M of R is always included in the free resolution).

`Depth(R): RngInvar -> RngIntElt`

Given the invariant ring $R = K[V]^G$ of the group G over the field K, return the depth of R. This is $n - d$ by the Auslander-Buchsbaum formula, where n is the rank of R and d is the homological dimension of R.

Example: Depth of an Invariant Ring (ex-6d79cb)

We construct a minimal free resolution of the invariant ring of the group generated by the degree-5 Jordan block over $\mathbf{F}_2$ and verify that the depth is 3.

```
> K:=GF(2);
> G := MatrixGroup<5,K | [1,0,0,0,0, 1,1,0,0,0, 0,1,1,0,0,
>                         0,0,1,1,0, 0,0,0,1,1]>;
> R := InvariantRing(G);
> time F := MinimalFreeResolution(R);
Time: 0.690
> F;
Chain complex with terms of degree 3 down to -1
Dimensions of terms: 0 1 7 22 0
> Depth(R);
3
> HomologicalDimension(R);
2
```

Sections *Steenrod Operations* and *Minimalization and Homogeneous Module Testing* present functions whose scope is not limited to the context of invariant theory.

120.17 Steenrod Operations

`SteenrodOperation(f, i): RngMPolElt, RngIntElt -> RngMPolElt`

The i-th Steenrod operation $P^i(f)$ of f, which must be a multivariate polynomial with coefficients in a finite field, and i must be a non-negative integer.

Example: Steenrod Operations (ex-dbbce3)

We demonstrate an elementary use of Steenrod operations.

```
> K:=GF(3);
> F4:=MatrixGroup<4,K |
>   [-1,0,0,0, 1,1,0,0, 0,0,1,0, 0,0,0,1],
>   [1,1,0,0, 0,-1,0,0, 0,1,1,0, 0,0,0,1],
>   [1,0,0,0, 0,1,-1,0, 0,0,-1,0, 0,0,1,1],
>   [1,0,0,0, 0,1,0,0, 0,0,1,1, 0,0,0,-1] >;
> R := InvariantRing(F4);
```

(continues on next page)

(continued from previous page)

```
> f2 := InvariantsOfDegree(R, 2)[1];
> f4 := SteenrodOperation(f2, 1);
> f10 := SteenrodOperation(f4, 3);
> f4;
2*x1^4 + x1^3*x3 + 2*x1^3*x4 + x1*x3^3 + 2*x1*x4^3 + 2*x2^3*x3 + x2^3*x4 +
    2*x2*x3^3 + x2*x4^3 + x4^4
> f10;
2*x1^10 + x1^9*x3 + 2*x1^9*x4 + x1*x3^9 + 2*x1*x4^9 + 2*x2^9*x3 + x2^9*x4 +
    2*x2*x3^9 + x2*x4^9 + x4^10
> f4 in R;
true
> f10 in R;
true
```

120.18 Minimalization and Homogeneous Module Testing

The following functions work with collections of polynomials which are considered as generators for subalgebras or submodules of a polynomial ring. They are repeated from the chapter on multivariate polynomials since they are used extremely often in invariant theory to express an invariant in terms of the primary and secondary invariants of an invariant ring. Full descriptions of the functions are not given here. See the descriptions in the chapter on multivariate polynomials.

`MinimalAlgebraGenerators(L): [ RngMPol ] -> [ RngMPol ]`

`MinimalAlgebraGenerators(L): { RngMPol } -> [ RngMPol ]`

Let $R = K[x_1, \ldots, x_n]$ be a polynomial ring of rank n over the field K. Suppose L is a set or sequence of k polynomials $p_1, \ldots, p_k$ in R. Let $A = K[p_1, \ldots, p_k]$ be the subalgebra (*not* ideal) of R generated by L. This function returns a minimal generating set of the algebra A as a (sorted) sequence of elements taken from L.

`HomogeneousModuleTest(P, S, F): [ RngMPol ], [ RngMPol ], RngMPol -> BoolElt, [ RngMPol ]`

Let $R = K[x_1, \ldots, x_n]$ be a polynomial ring of rank n over the field K. Suppose P is a sequence of k homogeneous polynomials $p_1, \ldots, p_k$ in R and suppose S is a sequence of r homogeneous polynomials $s_1, \ldots, s_r$ in R. Let $A = K[p_1, \ldots, p_k]$ be the subalgebra (*not* ideal) of R generated by P and let $M = A[s_1, \ldots, s_r]$ be the A-module generated by S over A. Finally, suppose F is an element of R. This function returns whether F is in the module M (considered as a submodule of R). If the result is `true`, the function also returns a sequence $C = [c_1, \ldots, c_r]$ of length r with $c_i \in K[t_1, \ldots, t_r]$ such that $F = \sum_{i=1}^{r} c_i(p_1, \ldots, p_k) \cdot s_i$.

`HomogeneousModuleTest(P, S, L): [ RngMPol ], [ RngMPol ], [ RngMPol ] -> [ Bool Elt ], [ [ RngMPol ] ]`

Let $R = K[x_1, \ldots, x_n]$ be a polynomial ring of rank n over the field K. Suppose P is a sequence of k homogeneous polynomials $p_1, \ldots, p_k$ in R and suppose S is a sequence of r homogeneous polynomials $s_1, \ldots, s_r$ in R. Let $A = K[p_1, \ldots, p_k]$ be the subalgebra (*not* ideal) of R generated by P and let $M = A[s_1, \ldots, s_r]$ be the A-module generated by S over A. Finally, suppose L is a sequence of length l of elements of R which are all homogeneous of (weighted) degree d. This function returns parallel sequences B and V with the following properties:

(a) B is sequence of length l of booleans such that for $1 \leq i \leq l$, $B[i]$ is `true` iff $L[i]$ is in the module M.

(b) V is a sequence of length l consisting of sequences of length r and consisting of polynomials in the polynomial ring $T = K[t_1, \ldots, t_r]$. (The polynomial ring $T = K[t_1, \ldots, t_r]$ is constructed separately but automatically with the print names `t1`, `t2`, etc.) If $B[i]$ is `false` (so $L[i]$ is not in M), $V[i]$ is a sequence of r zero polynomials. Otherwise $V[i]$ is a sequence of r polynomials $c_{i,1}, \ldots, c_{i,r}$ in T such that $L[i] = \sum_{j=1}^{r} c_{i,j}(p_1, \ldots, p_k) \cdot s_j$.

Example: Minimal Algebra Generators (ex-4ad419)

We demonstrate how the function `MinimalAlgebraGenerators` can be used to compute fundamental invariants (in fact, the Magma function `FundamentalInvariants` does just this).

```
> K := RationalField();
> G := PermutationGroup<6 | (1,2,3)(4,5,6), (1,2)(4,5)>;
> R := InvariantRing(G, K);
> P := PrimaryInvariants(R);
> P;
[
    x1 + x2 + x3,
    x4 + x5 + x6,
    x1^2 + x2^2 + x3^2,
    x4^2 + x5^2 + x6^2,
    x1^3 + x2^3 + x3^3,
    x4^3 + x5^3 + x6^3
]
> S := SecondaryInvariants(R);
> S;
[
    1,
    x1*x4 + x2*x5 + x3*x6,
    x1^2*x4 + x2^2*x5 + x3^2*x6,
    x1*x4^2 + x2*x5^2 + x3*x6^2,
    x1^2*x4^2 + 2*x1*x2*x4*x5 + 2*x1*x3*x4*x6 + x2^2*x5^2 + 2*x2*x3*x5*x6 +
        x3^2*x6^2,
    x1^3*x4^3 + x1^2*x2*x4*x5^2 + x1^2*x3*x4*x6^2 + x1*x2^2*x4^2*x5 +
        x1*x3^2*x4^2*x6 + x2^3*x5^3 + x2^2*x3*x5*x6^2 + x2*x3^2*x5^2*x6 +
        x3^3*x6^3
]
> MinimalAlgebraGenerators(P cat S);
[
    1,
    x1 + x2 + x3,
    x4 + x5 + x6,
    x1^2 + x2^2 + x3^2,
    x1*x4 + x2*x5 + x3*x6,
    x4^2 + x5^2 + x6^2,
    x1^3 + x2^3 + x3^3 + x4*x5*x6,
    x1^2*x4 + x2^2*x5 + x3^2*x6,
    x1*x4^2 + x2*x5^2 + x3*x6^2,
    x4^3 + x5^3 + x6^3
]
```

Example: Homogeneous Module Test for Invariant Rings (ex-4f66d4)

We demonstrate uses of the function `HomogeneousModuleTest` in invariant theory.

```
> // Create invariant ring R with primaries P, secondaries S
> R := InvariantRing(CyclicGroup(4), GF(2));
> P := PrimaryInvariants(R);
> S := SecondaryInvariants(R);
> #S;
5
> S[5];
x1^3*x3^2 + x1^2*x2^2*x3 + x1^2*x2*x3^2 + x1^2*x2*x4^2 +
    x1^2*x3^3 + x1^2*x3^2*x4 + x1*x2^2*x4^2 + x1*x3^2*x4^2 +
```

(continues on next page)

(continued from previous page)

```
    x2^3*x4^2 + x2^2*x3^2*x4 + x2^2*x3*x4^2 + x2^2*x4^3
> // Write S[2] in terms of P and S
> HomogeneousModuleTest(P, S, S[2]^2);
true [
    t1^2*t3^2 + t2^3,
    t1*t2,
    t1^3,
    0,
    0
]
> // Find all invariants I5 of degree 5
> I5 := InvariantsOfDegree(R, 5);
> I5;
[
    x1^5 + x2^5 + x3^5 + x4^5,
    x1^4*x2 + x1*x4^4 + x2^4*x3 + x3^4*x4,
    x1^4*x3 + x1*x3^4 + x2^4*x4 + x2*x4^4,
    x1^4*x4 + x1*x2^4 + x2*x3^4 + x3*x4^4,
    x1^3*x2^2 + x1^2*x4^3 + x2^3*x3^2 + x3^3*x4^2,
    x1^3*x2*x3 + x1*x2*x4^3 + x1*x3^3*x4 + x2^3*x3*x4,
    x1^3*x2*x4 + x1*x2^3*x3 + x1*x3*x4^3 + x2*x3^3*x4,
    x1^3*x3^2 + x1^2*x3^3 + x2^3*x4^2 + x2^2*x4^3,
    x1^3*x3*x4 + x1*x2^3*x4 + x1*x2*x3^3 + x2*x3*x4^3,
    x1^3*x4^2 + x1^2*x2^3 + x2^2*x3^3 + x3^2*x4^3,
    x1^2*x2^2*x3 + x1^2*x2*x4^2 + x1*x3^2*x4^2 + x2^2*x3^2*x4,
    x1^2*x2^2*x4 + x1^2*x3*x4^2 + x1*x2^2*x3^2 + x2*x3^2*x4^2,
    x1^2*x2*x3^2 + x1^2*x3^2*x4 + x1*x2^2*x4^2 + x2^2*x3*x4^2,
    x1^2*x2*x3*x4 + x1*x2^2*x3*x4 + x1*x2*x3^2*x4 + x1*x2*x3*x4^2
]
> // Write all elements of I5 in terms of P and S
> // (the t-variables correspond to elements of P and
> // the "columns" of the inner sequences to elements of S)
> HomogeneousModuleTest(P, S, I5);
[ true, true, true, true, true, true, true, true, true, true,
true, true, true, true ]
[
    [ t1^5 + t1^3*t2 + t1^3*t3 + t1*t2^2 + t1*t3^2 + t1*t4,
        0, t1^2 + t2 + t3, 0, 0 ],
    [ t1^3*t2 + t1^3*t3 + t1*t4, t1^2 + t2, t2 + t3, 0, 0 ],
    [ t1^3*t3 + t1*t3^2 + t1*t4, 0, t1^2 + t2 + t3, 0, 0 ],
    [ t1^3*t3 + t1*t2^2 + t1*t4, t1^2 + t2, t2 + t3, 0, 0 ],
    [ t1*t2^2 + t1*t3^2, t2, t1^2, 0, 1 ],
    [ t1*t2*t3, t3, t2 + t3, 0, 1 ],
    [ t1*t2*t3 + t1*t4, 0, t1^2 + t2, 0, 0 ],
    [ t1*t3^2 + t1*t4, 0, t3, 0, 0 ],
    [ 0, t3, t3, 0, 1 ],
    [ t1*t3^2, t2, t1^2 + t2, 0, 1 ],
    [ t1*t3^2, 0, 0, 0, 1 ],
    [ t1*t3^2, 0, t2, 0, 1 ],
    [ t1*t4, 0, t3, 0, 0 ],
    [ t1*t4, 0, 0, 0, 0 ]
]
```

120.19 Attributes of Invariant Rings and Fields

In this section we list various attributes of invariant rings which can be examined and set by the user. This allows low-level control of information stored in invariant rings or fields. Note that when the user sets an attribute, only minimal testing can be done on the value so if an incorrect value is set, unpredictable results may occur. Note also that if an attribute is not set, referring to it in an expression (using the ` operator) will *not* trigger the calculation of it (while intrinsic functions do); rather an error will ensue. Use the `assigned` operator to test whether an attribute is set.

R`PrimaryInvariants

The attribute for the primary invariants of invariant ring $R = K[V]^G$. If the attribute `R`PrimaryInvariants` is examined, either the current primary invariants of $R$ are set so they are returned or an error results. If the attribute `R`PrimaryInvariants` is set by assignment, it must be sequence of n algebraically-independent invariants of G, where n is the rank of R. Magma will not necessarily check that this condition is met since that may be very time-consuming. If the attribute is already set, the new value must be the same as the old value. Note that this attribute is useful when it is desired to compute secondary invariants of R with respect to some specially constructed primary invariants which would not be constructed by the automatic algorithm in Magma.

R`SecondaryInvariants

The attribute for the secondary invariants of invariant ring $R = K[V]^G$. If the attribute `R`SecondaryInvariants` is examined, either the current primary invariants of $R$ are set so they are returned or an error results. If the attribute `R`SecondaryInvariants` is set by assignment to Q, primary invariants for R must already be defined, and Q must be sequence of secondary invariants with respect to the primary invariants of R. Magma will not necessarily check that this condition is met since that may be very time-consuming. If the attribute is already set, the new value must be the same as the old value.

R`HilbertSeries

The attribute for the Hilbert series of invariant ring $R = K[V]^G$. If the attribute `R`HilbertSeries` is examined, either the Hilbert series of $R$ is computed so it is returned or an error results. If the attribute `R`HilbertSeries` is set by assignment to H, H must be rational function in the function field $Z(t)$ and equal to the Hilbert series of R. Magma will not necessarily check that this condition is met since that may be very time-consuming. If the attribute is already set, the new value must be the same as the old value.

Example: Attributes of Invariant Rings (ex-20324a)

We demonstrate elementary uses of attributes.

```
> // Create group G and subgroup H of G and invariant rings
> // RG and RH of G and H respectively.
> G := CyclicGroup(4);
> H := sub<G|G.1^2>;
> RG := InvariantRing(G, GF(2));
> RH := InvariantRing(H, GF(2));
>
> // Create Hilbert Series S of RG and set it in RG.
> F<t> := FunctionField(IntegerRing());
> S := (t^3 + t^2 - t + 1)/(t^8 - 2*t^7 + 2*t^5 - 2*t^4 +
>     2*t^3 - 2*t + 1);
> RG`HilbertSeries := S;
>
> // Note RG has no primary invariants yet so let Magma compute them as PG.
> RG`PrimaryInvariants;

>> RG`PrimaryInvariants;
     ^
```

(continues on next page)

(continued from previous page)

```
Runtime error in `: Attribute 'PrimaryInvariants' for this structure is valid
but not assigned
> PG := PrimaryInvariants(RG);
> PG;
[
    x1 + x2 + x3 + x4,
    x1*x2 + x1*x4 + x2*x3 + x3*x4,
    x1*x3 + x2*x4,
    x1*x2*x3*x4
]
>
> // Set primary invariants of RH to PG and compute secondary
> // invariants of RH with respect to PG.
> RH`PrimaryInvariants := PG;
> SecondaryInvariants(RH);
[
    1,
    x2 + x4,
    x2*x4,
    x1*x2 + x1*x3 + x2^2 + x2*x4 + x3*x4 + x4^2,
    x1^2*x2 + x1*x2*x3 + x1*x3*x4 + x2^3 + x3^2*x4 + x4^3,
    x1^2*x2 + x1*x2^2 + x1*x2*x3 + x1*x2*x4 + x1*x3*x4 + x1*x4^2
        + x2^3 + x2^2*x3 + x2*x3*x4 + x3^2*x4 + x3*x4^2 + x4^3,
    x1*x2*x4^2 + x2^2*x3*x4 + x2^2*x4^2,
    x1^2*x2*x4^2 + x1*x2^2*x3*x4 + x1*x2^2*x4^2 + x1*x2*x3*x4^2 +
        x2^3*x4^2 + x2^2*x3^2*x4 + x2^2*x3*x4^2 + x2^2*x4^3
]
```

120.20 Invariant Rings of Linear Algebraic Groups

By definition, a linear algebraic group is an affine variety G together with morphisms giving G the structure of a group. In the invariant theory algorithms of Magma, the group structure of G is nowhere required. Therefore an algebraic group will be defined by simply giving polynomials defining G as an affine variety. A G-module of an algebraic group is a finite dimensional vector space K^n together with a morphism $G \to \mathrm{GL}_n$ of algebraic groups. In Magma, such a morphism is given by specifying an n by n matrix whose entries are polynomials in the same variables as the polynomials specifying G (for more details, see section *Algebraic Group Actions on Polynomials*). An invariant ring of a linear algebraic group is constructed by giving a linear algebraic group together with a G-module. Magma makes no checks that the variety defined by the user has a multiplication making it into an algebraic group, or that the morphism $G \to \mathrm{GL}_n(K)$ really provides an action of G. If they do not, the computations will have unpredictable results. Likewise, Magma is unable to decide whether an algebraic group is reductive or linearly reductive. Therefore the user should indicate whether a group has these properties at creation by the options described below. This is important because Derksen's algorithm only works for linearly reductive groups.

120.20.1 Creation

`InvariantRing(I, A): RngMPol, Mtrx -> RngInvar`

```
Reductive        : BoolElt                    Default: false
LinearlyReductive: BoolElt                    Default: false
PolynomialRing   : RngMPol                    Default:
```

Construct the invariant ring R for the algebraic group G defined by the ideal I and representation matrix A.

If the parameter `Reductive` is set to `true`, then G is assumed to be reductive, while if the parameter `LinearlyReductive` is set to `true`, then G is assumed to be linearly reductive.

If the parameter `PolynomialRing` is set to a value P, then P is used as the polynomial ring in which the invariants of R will lie.

`BinaryForms(N, p): [RngIntElt], RngIntElt -> RngMPol, [[RngMPolElt]], RngMPol`

`BinaryForms(n, p): RngIntElt, RngIntElt -> RngMPol, [[RngMPolElt]], RngMPol`

Let $N = [n_1, \ldots, n_k]$ be a sequence of positive integers and let p be a positive prime or zero. Let $G = \mathrm{SL}_2(K)$ with K an algebraically closed field of characteristic p. This function defines the action on a direct sum of spaces of binary forms with degrees given by the n_i. The function returns three items: the ideal I_G defining G as an algebraic group, the representation matrix A (as a sequence of sequences of polynomials), and a polynomial ring on which G acts with appropriate naming of variables. The second version of the function is given an integer n, and takes N to be $[n]$.

120.20.2 Access

`GroupIdeal(R): RngInvar -> RngMPol`

Given an invariant ring R defined over an algebraic group G, return the ideal I defining G.

`Representation(R): RngInvar -> Mtrx`

Given an invariant ring R defined over an algebraic group G, return the representation matrix A for G.

120.20.3 Functions

`InvariantsOfDegree(R, d): RngInvar, RngIntElt -> [ RngMPolElt ]`

Return a K-basis of the space R_d of the homogeneous invariants of degree d in the invariant ring $R = K[V]^G$ of the algebraic group G over the field K as a sequence of polynomials.

`FundamentalInvariants(R): RngInvar -> RngMPol`

```
Optimize       : BoolElt                    Default: true
Minimize       : BoolElt                    Default: true
MinimizeHilbert: BoolElt                    Default: true
Force          : BoolElt                    Default: false
```

Given an invariant ring R defined over an algebraic group, return a sequence of fundamental invariants of R, using Derksen's algorithm.

By default, the computation of homogeneous invariants is optimized by extending at each the degree the basis obtained from multiplying lower-degree invariants by appropriate monomials. This method can be suppressed by setting `Optimize` to `false`. By default the generators will be minimal. By setting the parameter `Minimize` to `false`, no minimization will be attempted. By setting the parameter `MinimizeHilbert` to `false`, the basis of the Hilbert ideal will not be minimized.

By default the group must be linearly reductive. Setting the parameter `Force` to `true` will force the application of Derksen's algorithm even though the group may not be linearly reductive.

`DerksenIdeal(R): RngInvar -> [RngMPolElt]`

Given an invariant ring R defined over an algebraic group, return a sequence of generators of the Derksen ideal of R. The Derksen ideal is an ideal D of $P[y_1, \ldots, y_n]$, where $P = K[x_1, \ldots, x_n]$ is the ambient polynomial ring of R, and the y_i are new indeterminates. By definition, D is the intersection of all the ideals $\langle y_1 - g(x_1), \ldots, y_n - g(x_n)\rangle$ for all $g \in G$, the group of R. Geometrically, D is the vanishing ideal of the subset

$$\{(x, g(x)) | x \in K^n, g \in G\}$$

of the cartesian product $K^n \times K^n$.

`HilbertIdeal(R): RngInvar -> RngMPol`

```
Minimize: BoolElt                    Default: true
Force   : BoolElt                    Default: false
```

Given an invariant ring R defined over a linear algebraic group, return the Hilbert ideal of R. This is the ideal in the polynomial ring generated by all non-constant, homogeneous invariants. The result is a sequence of homogeneous generators (not necessarily invariant).

By default the generators will be minimal. By setting the parameter `Minimize` to `false`, no minimization will be attempted. Also, setting the parameter `Force` to `true` will force the application of Derksen's algorithm even though the group may not be linearly reductive.

Example: Invariant Ring of SL2 over the Rationals (ex-ff4ed2)

We consider invariant ring of the group $G = SL_2(\mathbb{Q})$, which is characterised by the equation $\det(A) = 1$.

```
> Q := RationalField();
> P<[a]>:=PolynomialRing(Q, 4);
> A := MatrixRing(P,2)!a;
> IG := ideal<P | Determinant(A) - 1>;
> IG;
Ideal of Polynomial ring of rank 4 over Rational Field
Lexicographical Order
Variables: a[1], a[2], a[3], a[4]
Basis:
[
    a[1]*a[4] - a[2]*a[3] - 1
]
```

The simultaneous action of G on three vectors is given by the matrix $I_3 \otimes A$:

```
> T := TensorProduct(MatrixRing(P, 3) ! 1, A);
> T;
[a[1] a[2]    0    0    0    0]
[a[3] a[4]    0    0    0    0]
[   0    0 a[1] a[2]    0    0]
[   0    0 a[3] a[4]    0    0]
[   0    0    0    0 a[1] a[2]]
[   0    0    0    0 a[3] a[4]]
```

We create the invariant ring R of G (which is reductive) with this action and compute fundamental invariants.

```
> IR := InvariantRing(IG, T: Reductive);
> FundamentalInvariants(IR);
[
    x3*x6 - x4*x5,
    x1*x6 - x2*x5,
    x1*x4 - x2*x3
]
```

We see that there are three fundamental invariants. It is well known that the invariant ring of the simultaneous action of SL_n on m vectors is generated by the minors of the $n \times m$ matrix formed by the vectors. We can see this in the present case.

```
> R<x1,x2,x3,x4,x5,x6> := PolynomialRing(Q, 6);
> M := Matrix([[x1,x3,x5], [x2,x4,x6]]);
```

(continues on next page)

(continued from previous page)

```
> M;
[x1 x3 x5]
[x2 x4 x6]
> Minors(M, 2);
[
    x1*x4 - x2*x3,
    -x1*x6 + x2*x5,
    x3*x6 - x4*x5
]
```

Example: Invariants of a Triple Tensor Product of SL2 (ex-893b15)

As a second example, we consider the representation of the group $SL_2(\mathbb{Q}) \times SL_2(\mathbb{Q}) \times SL_2(\mathbb{Q})$ given by the tensor product of the canonical representation:

```
> n:=3;
> P<[x]>:=PolynomialRing(RationalField(), n*4, "grevlex");
> L_A := [MatrixRing(P,2)!x[i..i+3]:i in [1..n*4 by 4]];
> IG := ideal<P|[Determinant(A)-1:A in L_A]>;
> IG;
Ideal of Polynomial ring of rank 12 over Rational Field
Graded Reverse Lexicographical Order
Variables: x[1], x[2], x[3], x[4], x[5], x[6], x[7], x[8],
x[9], x[10], x[11], x[12]
Basis:
[
    -x[2]*x[3] + x[1]*x[4] - 1,
    -x[6]*x[7] + x[5]*x[8] - 1,
    -x[10]*x[11] + x[9]*x[12] - 1
]
>
> M:=L_A[1];
> for i:=2 to n do
>     M:=TensorProduct(M,L_A[i]);
> end for;
> M;
[x[1]*x[5]*x[9]   x[1]*x[5]*x[10]   x[1]*x[6]*x[9]   x[1]*x[6]*x[10]
    x[2]*x[5]*x[9]   x[2]*x[5]*x[10]   x[2]*x[6]*x[9]   x[2]*x[6]*x[10]]
[x[1]*x[5]*x[11]   x[1]*x[5]*x[12]   x[1]*x[6]*x[11]   x[1]*x[6]*x[12]
    x[2]*x[5]*x[11]   x[2]*x[5]*x[12]   x[2]*x[6]*x[11]   x[2]*x[6]*x[12]]
[x[1]*x[7]*x[9]   x[1]*x[7]*x[10]   x[1]*x[8]*x[9]   x[1]*x[8]*x[10]
    x[2]*x[7]*x[9]   x[2]*x[7]*x[10]   x[2]*x[8]*x[9]   x[2]*x[8]*x[10]]
[x[1]*x[7]*x[11]   x[1]*x[7]*x[12]   x[1]*x[8]*x[11]   x[1]*x[8]*x[12]
    x[2]*x[7]*x[11]   x[2]*x[7]*x[12]   x[2]*x[8]*x[11]   x[2]*x[8]*x[12]]
[x[3]*x[5]*x[9]   x[3]*x[5]*x[10]   x[3]*x[6]*x[9]   x[3]*x[6]*x[10]
    x[4]*x[5]*x[9]   x[4]*x[5]*x[10]   x[4]*x[6]*x[9]   x[4]*x[6]*x[10]]
[x[3]*x[5]*x[11]   x[3]*x[5]*x[12]   x[3]*x[6]*x[11]   x[3]*x[6]*x[12]
    x[4]*x[5]*x[11]   x[4]*x[5]*x[12]   x[4]*x[6]*x[11]   x[4]*x[6]*x[12]]
[x[3]*x[7]*x[9]   x[3]*x[7]*x[10]   x[3]*x[8]*x[9]   x[3]*x[8]*x[10]
    x[4]*x[7]*x[9]   x[4]*x[7]*x[10]   x[4]*x[8]*x[9]   x[4]*x[8]*x[10]]
[x[3]*x[7]*x[11]   x[3]*x[7]*x[12]   x[3]*x[8]*x[11]   x[3]*x[8]*x[12]
    x[4]*x[7]*x[11]   x[4]*x[7]*x[12]   x[4]*x[8]*x[11]   x[4]*x[8]*x[12]]

> IR:=InvariantRing(IG, M: Reductive);
> time FundamentalInvariants(IR);
[
```

(continues on next page)

(continued from previous page)

```
    x1^2*x8^2 - 2*x1*x2*x7*x8 - 2*x1*x3*x6*x8 - 2*x1*x4*x5*x8 +
        4*x1*x4*x6*x7 + x2^2*x7^2 + 4*x2*x3*x5*x8 - 2*x2*x3*x6*x7 -
        2*x2*x4*x5*x7 + x3^2*x6^2 - 2*x3*x4*x5*x6 + x4^2*x5^2
]
Time: 0.610
> time DerksenIdeal(IR);
[
    y1^2*y8^2 - 2*y1*y2*y7*y8 - 2*y1*y3*y6*y8 - 2*y1*y4*y5*y8 + 4*y1*y4*y6*y7 +
        y2^2*y7^2 + 4*y2*y3*y5*y8 - 2*y2*y3*y6*y7 - 2*y2*y4*y5*y7 + y3^2*y6^2 -
        2*y3*y4*y5*y6 + y4^2*y5^2 - x1^2*x8^2 + 2*x1*x2*x7*x8 + 2*x1*x3*x6*x8 +
        2*x1*x4*x5*x8 - 4*x1*x4*x6*x7 - x2^2*x7^2 - 4*x2*x3*x5*x8 +
        2*x2*x3*x6*x7 + 2*x2*x4*x5*x7 - x3^2*x6^2 + 2*x3*x4*x5*x6 - x4^2*x5^2
]
Time: 0.010
> time HilbertIdeal(IR);
[
    x1^2*x8^2 - 2*x1*x2*x7*x8 - 2*x1*x3*x6*x8 - 2*x1*x4*x5*x8 + 4*x1*x4*x6*x7 +
        x2^2*x7^2 + 4*x2*x3*x5*x8 - 2*x2*x3*x6*x7 - 2*x2*x4*x5*x7 + x3^2*x6^2 -
        2*x3*x4*x5*x6 + x4^2*x5^2
]
Time: 0.000
```

So in this case, we find that the invariant ring is generated by a single polynomial.

Example: Invariants of Binary Forms `(ex-06a17f)`

We compute fundamental invariants for the invariant ring of $G = \mathrm{SL}_2(\mathbb{Q})$ acting on a space of binary forms.

```
> IG, A := BinaryForms([1,1,2,2], 0);
> IG;
Ideal of Polynomial ring of rank 4 over Rational Field
Lexicographical Order
Variables: t1, t2, t3, t4
Basis:
[
    t1*t4 - t2*t3 - 1
]
> A;
[t4   -t3   0   0   0   0   0   0   0   0]
[-t2   t1   0   0   0   0   0   0   0   0]
[0   0   t4   -t3   0   0   0   0   0   0]
[0   0   -t2   t1   0   0   0   0   0   0]
[0   0   0   0   t4^2   -t3*t4   t3^2   0   0   0]
[0   0   0   0   -2*t2*t4   t1*t4 + t2*t3   -2*t1*t3   0   0   0]
[0   0   0   0   t2^2   -t1*t2   t1^2   0   0   0]
[0   0   0   0   0   0   0   t4^2   -t3*t4   t3^2]
[0   0   0   0   0   0   0   -2*t2*t4   t1*t4 + t2*t3   -2*t1*t3]
[0   0   0   0   0   0   0   t2^2   -t1*t2   t1^2]
> R:=InvariantRing(IG,A: LinearlyReductive);
> time FundamentalInvariants(R);
[
    x8*x10 - 1/4*x9^2,
    x5*x10 - 1/2*x6*x9 + x7*x8,
    x5*x7 - 1/4*x6^2,
    x1*x4 - x2*x3,
    x1*x3*x7 - 1/2*x1*x4*x6 - 1/2*x2*x3*x6 + x2*x4*x5,
```

(continues on next page)

(continued from previous page)

```
    x1*x3*x10 - 1/2*x1*x4*x9 - 1/2*x2*x3*x9 + x2*x4*x8,
    x3^2*x10 - x3*x4*x9 + x4^2*x8,
    x3^2*x7 - x3*x4*x6 + x4^2*x5,
    x1^2*x10 - x1*x2*x9 + x2^2*x8,
    x1^2*x7 - x1*x2*x6 + x2^2*x5,
    x1*x2*x5*x10 - x1*x2*x7*x8 - x2^2*x5*x9 + x2^2*x6*x8,
    x1*x4*x5*x10 - 1/2*x1*x4*x6*x9 + x1*x4*x7*x8 - 1/2*x2*x3*x5*x10 +
        1/2*x2*x3*x6*x9 - 3/2*x2*x3*x7*x8 - 1/2*x2*x4*x5*x9 + 1/2*x2*x4*x6*x8,
    x3*x4*x5*x10 - x3*x4*x7*x8 - x4^2*x5*x9 + x4^2*x6*x8
]
Time: 0.650
```

Example: Invariants of a Non-Reductive Algebraic Group (ex-9a36c2)

We do simple computations on an invariant ring of an algebraic group. The group is not reductive, so fundamental invariants cannot be computed, but invariants of specific degrees can be.

```
> K := RationalField();
> Pa<a,b> := PolynomialRing(K, 2);
> IG := ideal<Pa|>;
> A := Matrix(7,
> [1, 0, 0, 0, 0, 0, 0, a, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
> 0, 0, 0, 0, a, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
> a, 1, 0, 0, 0, 0, 0, b, 0, 1 ]);
> A;
[1 0 0 0 0 0 0]
[a 1 0 0 0 0 0]
[0 0 1 0 0 0 0]
[0 0 a 1 0 0 0]
[0 0 0 0 1 0 0]
[0 0 0 0 a 1 0]
[0 0 0 0 b 0 1]
> R:=InvariantRing(IG, A);
> R;
Invariant Ring of algebraic group
Field of definition:
    Rational Field
> InvariantsOfDegree(R, 1);
[
    x1,
    x3,
    x5
]
> InvariantsOfDegree(R, 2);
[
    x1^2,
    x1*x3,
    x1*x4 - x2*x3,
    x1*x5,
    x1*x6 - x2*x5,
    x3^2,
    x3*x5,
    x3*x6 - x4*x5,
    x5^2
]
> FundamentalInvariants(R);
```

(continues on next page)

(continued from previous page)

```
>> FundamentalInvariants(R);
                           ^
Runtime error in 'FundamentalInvariants': Computing fundamental invariants (via
Derksen's algorithm) is only possible for linearly reductive groups
```

120.21 Invariant Fields

If G is a group acting on a polynomial ring $K[x_1, \ldots, x_n]$, it also acts on the rational function field $K(x_1, \ldots, x_n)$ by homomorphic extension. The invariants field $K(x_1, \ldots, x_n)^G$ is the field consisting of all functions which are fixed by G. Magma allows the construction of the invariant field by the function `InvariantField`. All that was said above about possible arguments of `InvariantRing` and access functions for invariant rings carries over to invariant fields. The category of invariant fields is `FldInvar`.

120.21.1 Creation

`InvariantField(G, K): GrpPerm, Fld -> FldInvar`

`InvariantField(G): GrpMat -> RngInvar`

`InvariantField(I, A): RngMPol, Mtrx -> FldInvar`

```
Reductive        : BoolElt                    Default: false
LinearlyReductive: BoolElt                    Default: false
FunctionField    : FldFunRat                  Default:
```

Create the invariant field for the group G over the field K. The arguments and parameters are the same as for the function *InvariantRing*, in the three cases of permutation groups, matrix groups, and algebraic groups.

120.21.2 Access

`FunctionField(F): FldInvar -> FldFunRat`

Given an invariant field F, return the underlying function field of F.

`Group(F): FldInvar -> Grp`

Given an invariant field F, return the underlying group of F.

`GroupIdeal(F): FldInvar -> RngMPol`

Given an invariant field F defined over an algebraic group G, return the ideal I defining G.

`Representation(F): FldInvar -> Mtrx`

Given an invariant field F defined over an algebraic group G, return the representation matrix A for G.

120.21.3 Functions for Invariant Fields

This section describes functions that apply to invariant fields.

`FundamentalInvariants(F): FldInvar -> RngMPol`

```
Al        : MonStgElt                 Default: "BethMuellerQuade"
Minimize  : BoolElt                   Default: true
Min       : RngIntElt                 Default: 0
BottomUpTo: RngIntElt                 Default: 0
```

Given an invariant field F, return a sequence of fundamental invariants of F which generate F as an algebra over the base field of the ambient rational function field of F.

By default this function uses the algorithm of Beth and Müller-Quade [MQB99]. By setting the parameter `Al` to `"FleischmannKemperWoodcock"`, an alternative algorithm of Fleischmann, Kemper, and Woodcock will be used.

By default the returned invariants will be minimal (in the sense of 'non-redundant'). By setting the parameter `Minimize` to `false`, no minimization will be attempted. The other parameters apply to the minimization and are as in the function *MinimizeGenerators* below.

`DerksenIdeal(F): FldInvar -> RngMPol`

Given an invariant field F, return the Derksen ideal of F. This is an ideal D in $K[y_1 \dots y_n]$, where $K = k(x_1 \dots x_n)$ is the ambient rational function field of F, and the y_i are new indeterminates. By definition, D is the intersection of all the ideals

$$< y_1 - g(x_1), \dots, y_n - g(x_n) >$$

for $g \in G$, the group of R. The function returns D as an ideal with a Groebner basis.

`MinimizeGenerators(L): [FldFunRatElt] -> [FldFunRatElt]`

```
Min       : RngIntElt                    Default: 0
BottomUpTo: RngIntElt                    Default: 0
```

Suppose L is a set or sequence of non-constant elements of a rational function field. This function selects a minimal (in the sense of 'irredundant') subset of L which generates the same subfield as L. The function returns a sequence of such minimal generators.

If the parameter `Min` is set to $m > 0$, then the function stops when a generating set with m elements is reached ($m = 0$ is the default and implies no limit).

If the parameter `BottomUpTo` is set to $b > 0$, then the function first tries to eliminate generators by testing if they lie in the subfield generated by a small number of elements from L. This small number is limited by b.

`QuadeIdeal(L): [FldFunRatElt] -> RngMPol`

```
Fy        : BoolElt                      Default:
LargeIdeal: BoolElt                      Default: false
```

Suppose L is a non-empty set or sequence of non-constant elements from a rational function field $F = k(x_1, \dots, x_n)$, generating a subfield $K = k(L)$. The Quade ideal, introduced in [MQS99], is the ideal in $F[y_1, \dots, y_n]$ generated by the kernel of the map $K[y_1, \dots, y_n] \to F$ given by $y_i \mapsto x_i$. This function returns the Quade ideal (with its basis being a Groebner basis).

The parameter `Fy` may be set to a polynomial ring P of rank n over F, so that the result is an ideal of P. If the parameter `LargeIdeal` is set to `true`, then an ideal in a larger polynomial ring is returned, whose intersection with $F[y_1, \dots, y_n]$ is the Quade ideal.

Example: Invariant Field of a Cyclic Group `(ex-41006c)`

This example works with the invariant field of the finite group C_3 over the rational field.

```
> IF := InvariantField(CyclicGroup(3), RationalField());
> time L := FundamentalInvariants(IF);
Time: 1.780
> L;
[
    x1 + x2 + x3,
    (x1^2*x2 - 3*x1*x2*x3 + x1*x3^2 + x2^2*x3)/(x1^2 - x1*x2 - x1*x3 + x2^2 -
        x2*x3 + x3^2),
```

(continues on next page)

(continued from previous page)

```
    (x1^3 - x1^2*x3 - x1*x2^2 + x2^3 - x2*x3^2 + x3^3)/(x1^2 - x1*x2 - x1*x3 +
        x2^2 - x2*x3 + x3^2)
]
> time DerksenIdeal(IF);
Ideal of Polynomial ring of rank 3 over Multivariate rational function field of
    rank 3 over Rational Field
Graded Reverse Lexicographical Order
Variables: y1, y2, y3
Dimension 0
Groebner basis:
[
    y2^2 + (-x1^3 + x1^2*x3 + x1*x2^2 - x2^3 + x2*x3^2 - x3^3)/(x1^2 - x1*x2 -
        x1*x3 + x2^2 - x2*x3 + x3^2)*y2 + (-x1^2*x2 + x1^2*x3 + x1*x2^2 -
        x1*x3^2 - x2^2*x3 + x2*x3^2)/(x1^2 - x1*x2 - x1*x3 + x2^2 - x2*x3 +
        x3^2)*y3 + (x1^3*x2 - x1^2*x2^2 - x1^2*x3^2 + x1*x3^3 + x2^3*x3 -
        x2^2*x3^2)/(x1^2 - x1*x2 - x1*x3 + x2^2 - x2*x3 + x3^2),
    y2*y3 + (-x1^2*x2 + 3*x1*x2*x3 - x1*x3^2 - x2^2*x3)/(x1^2 - x1*x2 - x1*x3 +
        x2^2 - x2*x3 + x3^2)*y2 + (-x1^2*x3 - x1*x2^2 + 3*x1*x2*x3 -
        x2*x3^2)/(x1^2 - x1*x2 - x1*x3 + x2^2 - x2*x3 + x3^2)*y3 + (x1^2*x2^2 -
        x1^2*x2*x3 + x1^2*x3^2 - x1*x2^2*x3 - x1*x2*x3^2 + x2^2*x3^2)/(x1^2 -
        x1*x2 - x1*x3 + x2^2 - x2*x3 + x3^2),
    y3^2 + (x1^2*x2 - x1^2*x3 - x1*x2^2 + x1*x3^2 + x2^2*x3 - x2*x3^2)/(x1^2 -
        x1*x2 - x1*x3 + x2^2 - x2*x3 + x3^2)*y2 + (-x1^3 + x1^2*x2 + x1*x3^2 -
        x2^3 + x2^2*x3 - x3^3)/(x1^2 - x1*x2 - x1*x3 + x2^2 - x2*x3 + x3^2)*y3 +
        (x1^3*x3 - x1^2*x2^2 - x1^2*x3^2 + x1*x2^3 - x2^2*x3^2 + x2*x3^3)/(x1^2
        - x1*x2 - x1*x3 + x2^2 - x2*x3 + x3^2),
    y1 + y2 + y3 - x1 - x2 - x3
]
```

Example: Invariant Field of a Non-Reductive Group (ex-27b1e3)

We can compute with the invariant field of the non-reductive group presented above.

```
> K := RationalField();
> Pa<a,b> := PolynomialRing(K, 2);
> IG := ideal<Pa|>;
> A := Matrix(7,
> [1, 0, 0, 0, 0, 0, 0, a, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
> 0, 0, 0, 0, a, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
> a, 1, 0, 0, 0, 0, 0, b, 0, 1 ]);
> A;
[1 0 0 0 0 0 0]
[a 1 0 0 0 0 0]
[0 0 1 0 0 0 0]
[0 0 a 1 0 0 0]
[0 0 0 0 1 0 0]
[0 0 0 0 a 1 0]
[0 0 0 0 b 0 1]
> IF := InvariantField(IG, A);
> IF;
Invariant field of algebraic group
Field of definition: Rational Field
> time FundamentalInvariants(IF);
[
    x5,
    x3/x5,
```

(continues on next page)

(continued from previous page)

```
    x1,
    (x1*x6 - x2*x5)/x5,
    (x3*x6 - x4*x5)/x5
]
Time: 0.010
> DerksenIdeal(IF);
Ideal of Polynomial ring of rank 7 over Multivariate rational function field of
    rank 7 over Rational Field
Graded Reverse Lexicographical Order
Variables: y1, y2, y3, y4, y5, y6, y7
Groebner basis:
[
    y1 - x1,
    y2 - x1/x5*y6 + (x1*x6 - x2*x5)/x5,
    y3 - x3,
    y4 - x3/x5*y6 + (x3*x6 - x4*x5)/x5,
    y5 - x5
]
```

120.22 Invariants of the Symmetric Group

Magma includes basic functions for working with symmetric polynomials, which are invariants of the symmetric group.

ElementarySymmetricPolynomial(P, k): RngMPol, RngIntElt -> RngMPolElt

Given a polynomial ring P of rank n, and an integer k with $1 \leq k \leq n$, return the k-th elementary symmetric polynomial of P.

IsSymmetric(f): RngMPolElt -> BoolElt, RngMPolElt

IsSymmetric(f, S): RngMPolElt, RngMPol -> BoolElt, RngMPolElt

Given a polynomial f from a polynomial ring P of rank n, return whether f is a symmetric polynomial of P (i.e., is symmetric in all the n variables of P). If the answer is true, a polynomial g from a new polynomial ring of rank n is returned such that $f = g(e_1, \ldots, e_n)$, where e_i is the i-th elementary symmetric polynomial of P. If g is desired to be a member of a particular polynomial ring S of rank n (to obtain predetermined names of variables, for example), then S may also be passed.

Example: Symmetric Polynomial in Elementary Terms (ex-e5b1eb)

We create a symmetric polynomial from $\mathbb{Q}[a, b, c, d]$ and express it in terms of the elementary symmetric polynomials.

```
> P<a, b, c, d> := PolynomialRing(RationalField(), 4, "grevlex");
> f :=
> a^2*b^2*c*d + a^2*b*c^2*d + a*b^2*c^2*d + a^2*b*c*d^2 + a*b^2*c*d^2 +
>     a*b*c^2*d^2 - a^2*b^2*c - a^2*b*c^2 - a*b^2*c^2 - a^2*b^2*d -
>     3*a^2*b*c*d - 3*a*b^2*c*d - a^2*c^2*d - 3*a*b*c^2*d - b^2*c^2*d -
>     a^2*b*d^2 - a*b^2*d^2 - a^2*c*d^2 - 3*a*b*c*d^2 - b^2*c*d^2 -
>     a*c^2*d^2 - b*c^2*d^2 + a + b + c + d;
> // Check orbit under Sym(4) has size one:
> #(f^Sym(4));
1
> Q<e1, e2, e3, e4> := PolynomialRing(RationalField(), 4);
> l, E := IsSymmetric(f, Q);
> l;
true
```

(continues on next page)

(continued from previous page)

```
> E;
e1 - e2*e3 + e2*e4
```

In the following example, we use a rational function field to define parameters a and b which occur as coefficients of the symmetric polynomial f.

```
> F<a,b> := FunctionField(RationalField(), 2);
> P<x1,x2,x3,x4,x5> := PolynomialRing(F, 5, "grevlex");
> y1 := x1^4 + x1^2*a + x1*b;
> y2 := x2^4 + x2^2*a + x2*b;
> y3 := x3^4 + x3^2*a + x3*b;
> y4 := x4^4 + x4^2*a + x4*b;
> y5 := x5^4 + x5^2*a + x5*b;
> f := y1*y2 + y1*y3 + y1*y4 + y1*y5 + y2*y3 + y2*y4 +
>      y2*y5 + y3*y4 + y3*y5 + y4*y5;
> Q<e1,e2,e3,e4,e5> := PolynomialRing(F,  5);
> l,E := IsSymmetric(f, Q);
> l, E;
true b*e1^3*e2 - 2*a*e1^3*e3 - 4*e1^3*e5 + a*e1^2*e2^2 +
    4*e1^2*e2*e4 + 2*e1^2*e3^2 - b*e1^2*e3 + 2*a*e1^2*e4 -
    4*e1*e2^2*e3 - 3*b*e1*e2^2 + 4*a*e1*e2*e3 + 8*e1*e2*e5 +
    a*b*e1*e2 - 8*e1*e3*e4 - 2*a^2*e1*e3 + b*e1*e4 - 6*a*e1*e5 +
    e2^4 - 2*a*e2^3 - 4*e2^2*e4 + a^2*e2^2 + 4*e2*e3^2 +
    5*b*e2*e3 + 2*a*e2*e4 + b^2*e2 - 3*a*e3^2 - 4*e3*e5 -
    3*a*b*e3 + 6*e4^2 + 2*a^2*e4 - 5*b*e5
```

120.23 Bibliography

- [AM94] A. Adem and R.J. Milgram. *Cohomology of Finite Groups*. Grundlehren der Mathematischen Wissenschaften. Springer, Berlin-New York-Heidelberg, 1994.
- [Der99] Harm Derksen. Computation of Invariants for Reductive Groups. *Adv. Math.*, 141:366–384, 1999.
- [Kem96] Gregor Kemper. Calculating Invariant Rings of Finite Groups over Arbitrary Fields. *J. Symbolic Comp.*, 21(3):351–366, 1996.
- [Kem99] Gregor Kemper. An Algorithm to Calculate Optimal Homogeneous Systems of Parameters. *J. Symbolic Comp.*, 27(2):171–184, 1999.
- [KS97] Gregor Kemper and Allan Steel. Some Algorithms in Invariant Theory of Finite Groups. In P. Dräxler, G.O. Michler, and C.M. Ringel, editors, *Computational Methods for Representations of Groups and Algebras, Euroconference in Essen, April 1-5 1997*, number 173 in Progress in Mathematics. Basel, 1997. Birkhäuser.
- [Kin07] Simon King. Minimal generating sets of non-modular invariant rings of finite groups. http://arxiv.org/abs/math/0703035, 2007.
- [MQS99] Jörg Müller-Quade and Rainer Steinwandt. Basic algorithms for rational function fields. *J. Symbolic Comp.*, 27(2):143–170, 1999. doi:http://dx.doi.org/10.1006/jsco.1998.0246.
- [MQB99] Jörg Müller-Quade and Thomas Beth. Calculating Generators for Invariant Fields of Linear Algebraic Groups. In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes (Honolulu, HI, 1999)*, number 1719 in LNCS, 392–403. Berlin, 1999. Springer.

Chapter 121

Differential Rings

121.1 Introduction

The Galois theory of linear differential equations, or differential Galois theory, is the analogue of the classical Galois theory of polynomials for linear differential equations. Generally speaking one studies linear differential equations, that is differential equations of the form

$$L(y) = a_n y^{(n)} + a_{n-1} y^{(n-1)} + \cdots + a_1 y^{(1)} + a_0 y = 0,$$

in which the coefficients a_i are contained in some ring. The natural analogue of a field in the classical case is the notion of a differential field, that is a specific case of a differential ring. A *differential ring* F is equipped with an additive map $\delta_F : F \to F$ called a *derivation*, satisfying the multiplicative rule

$$\delta_F(a \cdot b) = \delta_F(a) \cdot b + a \cdot \delta_F(b), \quad a, b \in F.$$

A classical derivation is the usual derivative. All differential rings have a ring structure and have a map defined on them. A differential ring that is also a field is called a *differential field*.

The differential rings have type `RngDiff` and their elements have type `RngDiffElt`. All differential rings contain a *differential ring of constants* on which the derivation acts as the zero map. The differential rings and their elements inherit all functionality of the rings from which the differential ring is created. We call the ring from which a differential ring F is created the *underlying ring* of F.

A solution of a differential equation is an element of some differential field. It can happen that a solution is not an element of a given differential field F, but is an element of a differential extension of F. By this we mean a differential field (ring) M with $F \subset M$ such that the derivations satisfy $\delta_M|_F = \delta_F$. This is completely analogous to field extensions induced by solutions of a polynomial.

To clearly describe linear differential equations in Magma we formalize the concept of taking the derivative. To a differential field F with derivation δ_F, one associates a non–commutative ring $F[D]$, *the ring of linear differential operators*. An element of $F[D]$ is called a *differential operator*. A differential operator of *degree* $n \in \mathbb{Z}_{\geq 0}$ in $F[D]$ is of the form

$$L = a_n D^n + a_{n-1} D^{n-1} + \cdots + a_1 D + a_0,$$

with $a_n \neq 0$ and all $a_i \in F$. Addition in $F[D]$ is term–wise and the multiplication of elements in $F[D]$ is determined by the rule

$$D * a = aD + \delta_F(a), \quad a \in F.$$

With these concepts $L(y) = 0$ is the linear differential equation

$$a_n \delta_F^n(y) + a_{n-1} \delta_F^{n-1}(y) + \cdots + a_1 \delta_F(y) + a_0 y = 0.$$

For an introduction to the basic concepts in differential Galois theory, one is encouraged to consult [vdPS03]. This book is used as the basis for the implementation of differential rings, fields, and operator rings in Magma.

121.2 Differential Rings and Fields

121.2.1 Creation

There are two ways to create a differential ring. The first creation is a general creation of a differential ring, for which the user specifies the ring and its derivation. The second creates a differential field which has the structure of a rational function field of transcendence degree 1 over its base field. Its derivation is specified by a differential.

Once a differential ring is created one can ask for its ring or field of fractions.

`DifferentialRing(P, f, C): Rng, Map, Rng -> RngDiff`

Given a ring P and derivation f acting on P, return the differential ring isomorphic to P, with induced derivation f acting on it, and ring of constants C. The ring C should be a subring of P on which f is zero.

Example: Creating a General Differential Ring (ex-dba016)

Here we illustrate the creation and printing of a general differential ring.

```
> P := PolynomialRing(Rationals());
> f := map<P->P | a:->5*Derivative(a)>;
> R := DifferentialRing(P, f, Rationals());
> R;
Differential Ring of Univariate Polynomial Ring over
Rational Field with derivation given by Mapping
from: RngUPol: P to RngUPol: P given by a rule [no inverse]
```

`RationalDifferentialField(C): Fld -> RngDiff`

The differential field in one variable over the constant field C. If this field is called F, say, then the derivation on F is given by $d/(1)d(F.1)$, where $F.1$ is the variable of F, and $(1)d(F.1)$ is its differential in the differential space of F. Any exact field with polynomial GCD is valid input for C.

Example: Creating a Rational Differential Field (ex-c5e585)

Here we illustrate the creation and printing of the differential field obtained from the command `RationalDifferentialField`.

```
> F<z> := RationalDifferentialField(Rationals());
> F;
Differential Ring of Algebraic function field defined over
Rational Field by $.2 - 4711 with
derivation given by (1) d(z)
```

`DifferentialLaurentSeriesRing(C): Fld -> RngDiff`

The differential Laurent series ring (in one variable) over the constant field C. If this field is called F, say, then the derivation on F is given by $F.1 \cdot d/d(F.1)$, where $F.1$ is the variable of F.

Example: Creating a Differential Laurent Series Ring (ex-d698ec)

This example illustrates the creation and printing of the differential Laurent series ring obtained from the command `DifferentialLaurentSeriesRing`.

```
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> S;
Differential Ring of Laurent series field in t over Rational Field
with derivation given by Mapping from: Laurent series field in t over Rational
Field to Laurent series field in t over Rational Field given by a rule [no
inverse]
```

`RingOfFractions(R): RngDiff -> RngDiff, Map`

Returns the differential ring $R[r^{-1} : r \in R \text{ not a zero divisor}]$ of fractions of the differential ring R, together with the inclusion map from R to the newly created ring.

`FieldOfFractions(R): RngDiff -> RngDiff, Map`

Returns the differential field of fractions of the differential ring R, together with the inclusion map from R to the newly created field.

`AssignNames(~R, S): RngDiff, [MonStgElt]`

Given a differential ring R with n indeterminates and a sequence S of n strings, assign the elements of S to the names of the variables of R.

This procedure only changes the names used in the printing of the elements of R.

121.2.2 Creation of Differential Ring Elements

The easiest way to create an element in a given ring is to use the angle bracket construction to attach names to the indeterminates of the ring. Others are given below.

`Name(R, i): RngDiff, RngIntElt -> RngDiffElt`

`R . i: RngDiff, RngIntElt -> RngDiffElt`

The i-th indeterminate of the differential ring R, where i is between 1 and the number of generators of R.

`R ! s: RngDiff, RngElt -> RngDiffElt`

Coerce the element s in the differential ring R. Elements that are coercible are elements that are coercible in the underlying ring of the differential ring R.

`Zero(R): RngDiff -> RngDiffElt`

The zero element of the differential ring R.

`One(R): RngDiff -> RngDiffElt`

`Identity(R): RngDiff -> RngDiffElt`

The identity element of the differential ring R.

`SeparatingElement(F): RngDiff -> RngDiffElt`

Returns the separating element of the algebraic differential field F.

Example: Creating Differential Field Elements `(ex-284f4d)`

We construct the differential field $F = \mathbb{Q}(z)$ with derivation d/dz and show some of the elements that can be created.

```
> F<z> := RationalDifferentialField(Rationals());
> F.1;
z
> two := F!2;
> two;
2
> Parent(two) eq F;
true
> Zero(F); One(F);
0
1
> Parent(Zero(F)) eq F and Parent(Identity(F)) eq F;
true
> elt := SeparatingElement(F);
> elt;
z
> ISA(Type(elt),RngDiffElt);
```

(continues on next page)

(continued from previous page)

```
true
> Parent(elt) eq F;
true
> elt eq F!SeparatingElement(UnderlyingRing(F));
true
```

121.3 Structure Operations on Differential Rings

121.3.1 Category and Parent

Differential Rings form the Magma category `RngDiff`. The notional power structures exist as parents of differential rings.

`Category(R): RngDiff -> RngDiff`

`Type(R): RngDiff -> RngDiff`

The category, or type, of the differential ring R.

`Parent(R): RngDiff -> PowStr`

The power structure of the differential ring R.

121.3.2 Related Structures

The underlying ring and constant ring from which the differential ring was created can each be retrieved as described below. There is also the concept of a base ring. If one has created a differential extension M/F in Magma, then F is the *base ring* of M.

`UnderlyingRing(R): RngDiff -> Rng`

The underlying ring of the differential ring R. The type of the underlying ring indicates what ring R inherits from.

`UnderlyingField(R): RngDiff -> Rng`

The underlying ring of the differential ring R, provided it is a field.

`BaseRing(R): RngDiff -> Rng`

The base ring of the differential ring R.

`BaseField(R): RngDiff -> Rng`

The base ring of the differential ring R, provided it is a field.

`ConstantRing(R): RngDiff -> Rng`

The constant ring of the differential ring R. The derivation of R acts trivially on the constant ring. It is therefore contained in the differential ring of constants of R.

`ConstantField(R): RngDiff -> Rng`

The constant ring of the differential ring R, provided it is a field.

`ExactConstantField(F): RngDiff -> RngDiff, Map`

The exact constant field of F, i.e. the algebraic closure in F of the constant field of F, together with the inclusion map to F. The field F must be a function field. The differential field F must have been created with respect to a differential. If the derivation of F has been constructed with respect to a differential, then the exact constant field coincides with the differential field of constants of F.

`Generators(R): RngDiff -> SeqEnum`

The list of generators of the differential ring R. If there is no list assigned to R, one is constructed by default from the underlying ring of R.

Example: Related Structures of a Differential Field (ex-64f77a)

First we construct the differential field $F = \mathbb{Q}(z)$ with derivation d/dz and show what some of the related structures are. Then we construct the field extension $M = \mathbb{Q}(z, \alpha)$, where α is a root of the polynomial $X^2 - 2$. We do this with the usual `ext< >` constructor. For M we again derive some related structures.

```
> F<z> := RationalDifferentialField(Rationals());
> ConstantRing(F);
Rational Field
> UnderlyingRing(F);
Algebraic function field defined over Rational Field by
$.2 - 4711
>
> _<X> := PolynomialRing(F);
> M<alpha> := ext< F | X^2-2 >;
> BaseRing(M);
Differential Ring of Algebraic function field defined over Rational Field by
$.2 - 4711
with derivation given by (1) d(z)
> BaseRing(M) eq F;
true
> ConstantRing(M);
Rational Field
> E := ExactConstantField(M);
> E;
Number Field with defining polynomial $.1^2 - 2 over the Rational Field
> Generators(M);
[ alpha ]
```

Example: Related Structures of a Laurent Series Ring (ex-326abb)

Related structures also exist for differential Laurent series rings.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> UnderlyingRing(S);
Laurent series field in t over Rational Field
> ConstantRing(S);
Rational Field
> Generators(S);
[ t ]
```

121.3.3 Derivation and Differential

The derivation of a differential ring and its differential, whenever applicable, can be retrieved as indicated below.

`Derivation(R): RngDiff -> Map`

The derivation of the differential ring R.

`Differential(F): RngDiff -> DiffFunElt`

The differential belonging to the derivation of the differential field F. The field F must have been constructed in such a way that its derivation is defined by a differential.

Example: Derivation and Differential of a Field (ex-627c5f)

```
> F<z> := RationalDifferentialField(Rationals());
> Derivation(F);
Mapping from: RngDiff: F to RngDiff: F given by a rule [no inverse]
```

(continues on next page)

(continued from previous page)

```
> Differential(F);
(1) d(z)
```

121.3.4 Numerical Invariants

`Ngens(R): RngDiff -> RngIntElt`

The number of indeterminates associated with the differential ring R.

121.3.5 Predicates and Booleans

`R eq F: RngDiff, RngDiff -> BoolElt`

Returns `true` if and only if the differential rings R and F are the same.

`IsIdentical(R, F): RngDiff, RngDiff -> BoolElt`

Returns `true` if and only if the differential rings R and F are identical.

`IsDomain(R): RngDiff -> BoolElt`

Returns `true` if and only if the differential ring R is a domain.

`IsField(R): RngDiff -> BoolElt`

Returns `true` if and only if the differential ring R is field.

`IsDifferentialField(R): Rng -> BoolElt`

Returns `true` if and only if the ring R is a differential field.

`IsAlgebraicDifferentialField(R): Rng -> BoolElt`

Returns `true` if and only if the field structure of the differential ring R is an algebraic function field.

`IsDifferentialSeriesRing(R): Rng -> BoolElt`

Returns `true` if and only if the underlying ring of the differential ring R is a series ring.

`IsDifferentialLaurentSeriesRing(R): Rng -> BoolElt`

Returns `true` if and only if the underlying ring of the differential ring R is a Laurent series ring and R has been created with a known constant ring.

Example: Predicates for Differential Rings `(ex-913ce7)`

This example shows some booleans for various differential rings.

```
> F<z>:=RationalDifferentialField(Rationals());
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> IsAlgebraicDifferentialField(F);
true
> IsDifferentialSeriesRing(F);
false
> IsAlgebraicDifferentialField(S);
false
> IsDifferentialSeriesRing(S);
true
> IsDifferentialLaurentSeriesRing(S);
true
```

`HasProjectiveDerivation(F): RngDiff -> BoolElt`

Returns `true` if and only if F is a differential ring with derivation weakly of the form $(F.1)\cdot d/d(F.1)$.

`HasZeroDerivation(F): RngDiff -> BoolElt`

Returns `true` if and only if the algebraic differential field or differential series ring F has zero derivation. When F is a series ring we relax being zero to being weakly zero.

Example: Predicates on the Derivation (ex-a3014d)

```
> F<z>:=RationalDifferentialField(Rationals());
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> HasProjectiveDerivation(F);
false
> HasProjectiveDerivation(ChangeDerivation(F,z));
true
> HasZeroDerivation(F);
false
> HasProjectiveDerivation(S);
true
> HasProjectiveDerivation(ChangeDerivation(S,S!3));
false
> HasZeroDerivation(S);
false
```

121.3.6 Precision

`RelativePrecision(F): RngDiff -> RngElt`

Returns the relative precision of the underlying series ring of F.

`RelativePrecisionOfDerivation(F): RngDiff -> RngElt`

Given a differential Laurent series ring F, returns the relative precision of the ring derivative of F.1.

Example: Relative Precision of a Series Ring (ex-4674cf)

This example illustrate the relative precision of differential rings.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> Derivative(t);
t
> IsDifferentialLaurentSeriesRing(S);
true
> RelativePrecision(S);
20
> RelativePrecision(UnderlyingRing(S));
20;
> V<w>:=DifferentialLaurentSeriesRing(Rationals():Precision:=30);
> RelativePrecision(V);
30
> RelativePrecision(V) eq RelativePrecision(UnderlyingRing(V));
true
```

Example: Relative Precision of a Derivation (ex-617005)

```
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> RelativePrecisionOfDerivation(S);
Infinity
> V<w> := ChangeDerivation(S,t+O(t^6));
> Derivation(V)(w);
w^2 + O(w^7)
> RelativePrecisionOfDerivation(V);
5
```

`ChangePrecision(F, p): RngDiff, RngElt -> RngDiff, Map`

Returns the differential series ring isomorphic to F with relative precision p. The map returned is the induced map of F to the new field.

Example: Changing the Precision of a Series Ring (ex-d18d60)

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> RelativePrecision(S);
20
> V<w>,mp :=  ChangePrecision(S,10);
> Type(V);
RngDiff
> IsDifferentialLaurentSeriesRing(V);
true
> RelativePrecision(V);
10
> RelativePrecision(1/(w-1)) eq 10;
true
> mp(t) eq w;
true
> w@@mp eq t;
true
> derivt := Derivation(S)(t);
> derivt;
t
> derivw := Derivation(V)(w);
> derivw;
w
> mp(derivt) eq Derivation(V)(w);
true
```

121.4 Element Operations on Differential Ring Elements

121.4.1 Category and Parent

`Category(s): RngDiffElt -> RngDiffElt`

`Type(s): RngDiffElt -> RngDiffElt`

The category, or type, of the differential ring element s.

`Parent(s): RngDiffElt -> RngDiff`

The parent of the differential ring element s.

121.4.2 Arithmetic

All the usual arithmetic operations are possible for differential ring elements.

`s + t: RngDiffElt, RngDiffElt -> RngDiffElt`

The sum of the two differential ring elements s and t.

`- s: RngDiffElt -> RngDiffElt`

The negation of the differential ring element s.

`s - t: RngDiffElt, RngDiffElt -> RngDiffElt`

The difference between the differential ring elements s and t.

`s * t: RngDiffElt, RngDiffElt -> RngDiffElt`

The product of the differential ring elements s and t.

`s ^ n: RngDiffElt, RngIntElt -> RngDiffElt`

Given a differential ring element s and an integer n, return the n-th power of s. If s is invertible, n may be negative.

`s div t: RngDiffElt, RngDiffElt -> RngDiffElt`

Given the differential ring elements s and t, return the exact division of s by t, if s is divisible by t.

`s / t: RngDiffElt, RngDiffElt -> RngDiffElt`

Given the differential field elements s and t, return s divided by t.

121.4.3 Predicates and Booleans

`s eq t: RngDiffElt, RngDiffElt -> BoolElt`

Return `true` iff the differential ring elements s and t are exactly the same.

`IsZero(s): RngDiffElt -> BoolElt`

Return `true` iff the differential ring element s is the zero element of its parent.

`IsOne(s): RngDiffElt -> BoolElt`

Return `true` iff the differential ring element s is the unity element of its parent.

`IsWeaklyEqual(s, t): RngDiffElt, RngDiffElt -> BoolElt`

Return `true` if and only if the differential ring element s is weakly equal to the differential ring element t.

`IsWeaklyZero(s): RngDiffElt -> BoolElt`

Return `true` if and only if the differential ring element s is weakly equal to the zero element of its parent.

`IsOrderTerm(s): RngDiffElt -> BoolElt`

`IsOrderTerm(s): RngSerElt -> BoolElt`

Return `true` if and only if the differential ring element s is purely an order term of a differential series ring.

Example: Predicates for Differential Ring Elements (ex-61c4b6)

This examples shows the booleans for various differential rings.

```
> F<z> := RationalDifferentialField(Rationals());
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> IsOne(F!1);
true
> t eq t+O(t^2);
false
> IsWeaklyEqual(t, t+O(t^2));
true
> IsWeaklyZero(t^(-1));
false
> IsWeaklyZero(O(t));
true
> IsOrderTerm(t+O(t^2));
false
> IsOrderTerm(O(t));
true
```

121.4.4 Coefficients and Terms

`O(s): RngDiffElt -> RngDiffElt`

Creates the order term of the differential series s.

`Truncate(s): RngDiffElt -> RngDiffElt`

The known part of the differential series s.

Eltseq(s): RngDiffElt -> SeqEnum

Returns the coefficients of the differential ring element s.

Exponents(s): RngDiffElt -> SeqEnum

Exponents(s): RngSerElt -> SeqEnum

Returns the interval from the valuation of s to (including) the degree of s.

Example: Coefficients of Algebraic Differential Elements (ex-18d9d7)

```
> F<z> := RationalDifferentialField(Rationals());
> _<X> := PolynomialRing(F);
> K<x>, mp := ext<F|X^2+X+1>;
> seq := Eltseq(x^2);
> seq;
[ -1, -1 ]
> Universe(seq) eq F;
true
```

Example: Coefficients of Differential Series (ex-4eef4b)

```
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> O(t+t^2);
O(t)
> Parent(O(t)) eq S;
true
> trunc := Truncate(t^(-1)+5*t^2 +O(t^4));
> trunc;
t^-1 + 5*t^2
> Parent(trunc) eq S;
true
> seq := Eltseq(trunc);
> seq;
[ 1, 0, 0, 5 ]
> Universe(seq) eq Rationals();
true
> Exponents(trunc);
[ -1 .. 2 ]
```

121.4.5 Conjugates, Norm, and Trace

MinimalPolynomial(s): RngDiffElt -> RngUPolElt

The minimal polynomial of the differential field element s over the base field.

Example: Minimal Polynomial of an Algebraic Element (ex-ecfdbc)

```
> F<z> := RationalDifferentialField(Rationals());
> P<X> := PolynomialRing(F);
> K<x>, mp := ext<F|X^2+X+1>;
> f := MinimalPolynomial(x^2);
> f;
X^2 + X + 1
> Parent(f) eq P;
true
> g := MinimalPolynomial(x+3/2);
> g;
X^2 + -2*X + 7/4
```

121.4.6 Derivatives and Differentials

`Derivative(s): RngDiffElt -> RngDiffElt`

The image of s under the derivation of the parent of s. Notice that it can be different to the "usual" derivative, as it relies on the defined derivation.

`Differential(s): RngDiffElt -> RngDiffElt`

Returns the differential of s in the algebraic differential field F, as a differential in the differential space of the underlying ring of F.

Example: Derivatives and Differentials of Elements (ex-396a0f)

```
> F<z> := RationalDifferentialField(Rationals());
> Derivative(z^2 + 7/z);
(2*z^3 - 7)/z^2
> Differential(z);
(1) d(z)
> Differential(1/z+6+5*z);
((5*z^2 - 1)/z^2) d(z)
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> Derivative(5 + 2*t + 3*t^2);
2*t + 6*t^2
```

121.5 Changing Related Structures

Sometimes whilst working with a differential ring R, one might wish to consider the same ring, but with a different derivation or with a larger constant ring. It is a consequence of the creation of a differential ring, that its constant ring may actually be smaller than its differential ring of constants.

To alter the settings defined by the creation of a differential ring or field the following functions are available.

`ChangeDerivation(R, f): RngDiff, RngElt -> RngDiff, Map`

Returns a differential ring isomorphic to R, but whose derivation is the map $f \cdot$ `Derivation(R)` induced by the isomorphism. The ring element f must be non–zero. The isomorphism of R to the new differential ring is also returned. The new differential ring has the same underlying ring as R.

Example: Changing the Derivation of a Differential Ring (ex-271826)

```
> F<z> := RationalDifferentialField(Rationals());
> Derivative(z^2);
2*z
> K, toK := ChangeDerivation(F, z);
> K;
Differential Ring of Algebraic function field defined over Rational Field by
$.2 - 4711
with derivation given by (1/z) d(z)
> toK;
Mapping from: RngDiff: F to RngDiff: K given by a rule
> Derivative(toK(z^2));
2*z^2
> UnderlyingRing(F) eq UnderlyingRing(K);
true
```

Notice that the differential of K is $(1/z)d(z)$, so that the derivation of K is $z \cdot d/dz$, as requested.

`ChangeDifferential(F, df): RngDiff, DiffFunElt -> RngDiff, Map`

Returns the algebraic differential field, whose underlying ring is the one of F, but with derivation with respect to the differential df. The map returned is the bijective map from F into the new algebraic differential field.

Example: Changing the Differential of a Differential Ring (ex-7c6694)

```
> F<z> := RationalDifferentialField(Rationals());
> df := Differential(1/z);
> df in DifferentialSpace(UnderlyingRing(F));
true
> M<u>, mp := ChangeDifferential(F,df);
> IsAlgebraicDifferentialField(M);
true
> Domain(mp) eq F and Codomain(mp) eq M;
true
> Differential(M);
(-1/u^2) d(u)
> mp(z);
u
> Derivation(M)(u);
u^2
> Derivation(F)(z);
1
> dg := Differential(z^3+5);
> N<v>, mp := ChangeDifferential(F,dg);
> Differential(M);
(3*v^2) d(v)
> mp(z);
v
> Derivation(N)(mp(z));
1/3/v^2
```

ConstantFieldExtension(F, C): RngDiff, Fld -> RngDiff, Map

Returns the differential field isomorphic to the differential field F, but whose constant field is the extension C, and the isomorphism from F to the new field. The differential field F must be an algebraic function field.

Example: Constant Field Extension of a Differential Field (ex-17cbbe)

```
> F<z> := RationalDifferentialField(Rationals());
> _<X> := PolynomialRing(F);
> M := ext< F | X^2-2 >;
> ConstantField(M);
Rational Field
> _<x>:=PolynomialRing(Rationals());
> C := NumberField(x^2-2);
> Mext, toMext := ConstantFieldExtension(M, C);
> ConstantField(Mext);
Number Field with defining polynomial x^2 - 2 over the Rational Field
> toMext;
Mapping from: RngDiff: M to RngDiff: Mext given by a rule
```

Example: Constant Field Extension of a Laurent Series Ring (ex-a878b1)

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> P<T> := PolynomialRing(Rationals());
> Cext := ext<Rationals()|T^2+1>;
> Sext<text>, mp := ConstantFieldExtension(S,Cext);
> IsDifferentialLaurentSeriesRing(Sext);
true
> ConstantRing(Sext) eq Cext;
true
```

(continues on next page)

(continued from previous page)

```
> Derivative(text^(-2)+7+2*text^3+O(text^6));
-2*text^-2 + 6*text^3 + O(text^6);
> mp;
Mapping from: RngDiff: S to RngDiff: Sext given by a rule
> mp(t);
text
```

Completion(F, p): RngDiff, PlcFunElt -> RngDiff, Map

```
Precision: RngIntElt                    Default: Infinity()
```

The completion of the differential field F with respect to the place p. The place p should be an element of the set of places of F. The derivation of the completion is the one naturally induced by the derivation of F. The map returned is the embedding of F into the completion. Upon creation one can set the precision by using **Precision**. If no precision is given, then a default value is taken.

Example: Laurent Series Ring by Completion (ex-5fcb44)

This example illustrates the creation of the differential Laurent series ring by using the command **Completion**.

```
> F<z> := RationalDifferentialField(Rationals());
> pl := Zeros(z)[1];
> S<t>, mp := Completion(F,pl: Precision := 5);
> IsDifferentialLaurentSeriesRing(S);
true
> mp;
Mapping from: RngDiff: F to RngDiff: S given by a rule
> Domain(mp) eq F, Codomain(mp) eq S;
true true
> Derivation(S)(t);
1
> 1/(1-t);
1 + t + t^2 + t^3 + t^4 + O(t^5)
```

Example: Completion of an Elliptic Function Field (ex-512dde)

This example shows that one does not have to restrict to differential fields of genus 0 to use **Completion**.

```
> F<z> := RationalDifferentialField(Rationals());
> P<Y> := PolynomialRing(F);
> K<y> := ext<F|Y^2-z^3+z+1>;
> Genus(UnderlyingRing(K));
1
> pl:=Zeros(K!z)[1];
> Degree(pl);
2
> S<t>, mp := Completion(K,pl);
> IsDifferentialLaurentSeriesRing(S);
true
> C<c> := ConstantRing(S);
> C;
Number Field with defining polynomial $.1^2 + 1 over the Rational Field
> mp(y) + O(t^4);
c - t - 4*t^3 + O(t^4)
```

121.6 Ring and Field Extensions

The first differential ring and field extensions we consider are the ones induced by a differential operator. Given a differential operator

$$L = a_n D^n + a_{n-1} D^{n-1} + \cdots + a_1 D + a_0, \quad a_n \neq 0$$

in a differential operator ring $F[D]$ with coefficients in a differential field F, we construct a ring or field extension of degree n over F, whose indeterminates play the role of a formal solution of $L(y) = 0$ and its derivatives.

Given a differential field F, it is also possible to construct differential extensions of the form $F[X]/f(X)$, where $f(X)$ is an irreducible polynomial over F.

`DifferentialRingExtension(L): RngDiffOpElt -> RngDiff`

Constructs a differential ring extension of the base ring of the differential operator L, by adding a formal solution of L and its formal derivatives as indeterminates.

Let P denote the new differential ring, and F the coefficient ring of L. The ring F is a differential field. If n is the degree of L, the underlying ring of P is a multivariate polynomial ring of degree n over F. We thus have $P = F[Y_1, Y_2, \ldots, Y_n]$, with indeterminates $Y_1, Y_2, \ldots, Y_n$. If L is written as $a_n D^n + a_{n-1} D^{n-1} + \cdots + a_1 D + a_0 \in F[D]$, then the derivation of P is induced by the differential operator L as follows: $\delta_P(Y_i) = Y_{i+1}$, for $i < n$ and $a_n \delta_P(Y_n) = -a_{n-1} Y_{n-1} - \cdots - a_2 Y_2 - a_1 Y_1$. With this construction Y_1 mimics a solution of $L(y) = 0$, and all the others are its derivatives.

`DifferentialFieldExtension(L): RngDiffOpElt -> RngDiff`

Constructs a differential field extension of the base ring of the differential operator L, by adding a formal solution of L and its formal derivatives as indeterminates.

The construction of the new differential field is completely analogous to the differential ring created by `DifferentialRingExtension(L)`. The only difference is that now a differential field $M = F(Y_1, Y_2, \ldots, Y_n)$, with n indeterminates $Y_1, Y_2, \ldots, Y_n$ is created. The action of the derivation of M on $Y_1, Y_2, \ldots, Y_n$ is as described in `DifferentialRingExtension(L)`.

Example: Differential Ring Extension by an Operator (ex-434a22)

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> L := z^2*D^2-z*D+1;
> P<Y1,Y2> := DifferentialRingExtension(L);
> P;
Differential Ring Extension over F
with derivation given by Mapping from: Polynomial ring of rank 2 over F to
Polynomial ring of rank 2 over F given by a rule [no inverse]
> Derivative(Y1);
Y2
> Derivative(Y2);
-1/z^2*Y1 + 1/z*Y2
```

Example: Differential Field Extension by an Operator (ex-5ef2f3)

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> L := z^2*D^2-1;
> M<Y,DY> := DifferentialFieldExtension(L);
> IsDifferentialField(M);
true
> Derivative(Y);
DY
> Derivative(DY);
-1/z^2*Y
```

ext< F | f >: RngDiff, RngUPolElt -> RngDiff

The differential field extension $F(\alpha)$ of the differential field F, where α is a root of the irreducible polynomial f over F. The angle bracket notation may be used to assign the root α to an identifier.

Example: Algebraic Extension of a Differential Field (ex-8a3a0e)

```
> F<z> := RationalDifferentialField(Rationals());
> _<X> := PolynomialRing(F);
> M<alpha> := ext< F | X^2-z >;
> M;
Differential Ring Extension over F by $.1^2 - z
with derivation given by (1) d(z)
> alpha^2;
z
```

The differential of M is the differential dz of the differential space of F lifted to the space of differentials of M.

ExponentialFieldExtension(F, f): RngDiff, RngDiffElt -> RngDiff

Returns the differential field $F(E)$ as an extension of F, such that the derivation of E is $f \cdot E$. The parent of f must be F.

LogarithmicFieldExtension(F, f): RngDiff, RngDiffElt -> RngDiff

Returns the differential field $F(L)$ as an extension of F, such that the derivation of L is $F(L)!f$. The parent of f must be F.

Example: Exponential and Logarithmic Extensions (ex-0b50af)

```
> F<z> := RationalDifferentialField(Rationals());
> K<E> := ExponentialFieldExtension(F, z);
> K;
Differential Ring Extension over F
with derivation given by Mapping from: Multivariate Rational function field of
rank 1 over F to Multivariate Rational function field of rank 1 over F given by
a rule [no inverse]
> Derivative(E);
z*E
> _<L> := LogarithmicFieldExtension(F, 1/z);
> Derivative(L);
1/z
> Parent($1) eq Parent(L);
true
```

PurelyRamifiedExtension(f): RngUPolElt[RngDiff] -> RngDiff, Map

Creates a purely ramified field extension M of the differential field F with respect to the purely ramified polynomial $f \in F[X]$. By definition, such a polynomial f is of the form $X^n - a \cdot (F.1)$ for some constant element a in F and positive integer n. The returned extension field M is of the same type as F. The allowed differential fields are algebraic differential fields and differential Laurent series rings. When F is a differential Laurent series ring, its derivation is required to be weakly of the form $c * (F.1) * d/d(F.1)$ for some constant c. The relative precision of M is then n times the relative precision of F. The second argument returned is the embedding map of F into M. The inverse map acts on elements for which it is defined. Otherwise it returns 0.

Example: Purely Ramified Extension of an Algebraic Differential Field (ex-838a6e)

A purely ramified extension of an algebraic differential field is constructed in this example.

```
> F<z> := RationalDifferentialField(Rationals());
> _<X> := PolynomialRing(F);
```

(continues on next page)

(continued from previous page)

```
> Fext<v>, mp := PurelyRamifiedExtension(X^2-5*z);
> IsAlgebraicDifferentialField(Fext);
true
> mp(z) eq 1/5*v^2;
true
> Parent(mp(z)) eq Fext;
true
> Derivation(Fext)(mp(z));
1
> Derivation(Fext)(v);
1/2/z*v
> Derivation(Fext)(v^2) eq Fext!5;
true
> Inverse(mp)(v^2);
5*z;
```

Example: Purely Ramified Extension of a Laurent Series Ring (ex-efad09)

A differential Laurent series ring with a derivation without an order term is considered in this example.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> _<T>:=PolynomialRing(S);
> pol := T^4-5*t;
> Sext<r>,mp := PurelyRamifiedExtension(pol);
> IsDifferentialLaurentSeriesRing(Sext);
true
> BaseRing(Sext) eq S and ConstantField(Sext) eq ConstantField(S);
true
> RelativePrecision(Sext);
80
> RelativePrecisionOfDerivation(Sext);
Infinity
> Derivation(S)(t);
t
> mp(t);
1/5*r^4
> Derivation(Sext)(mp(t));
1/5*r^4
> mp(Derivation(S)(t));
1/5*r^4
> x := 4+6*t+O(t^6);
> mp(x);
4 + 6/5*r^4 + O(r^24)
> Derivation(Sext)(mp(x));
6/5*r^4 + O(r^24)
> mp(Derivation(S)(x));
6/5*r^4 + O(r^24)
> Inverse(mp)(r^4-r^8);
5*t - 25*t^2
> Inverse(mp)(r^4+O(r^5));
5*t + O(t^2)
> Derivation(Sext)(r);
1/4*r
```

Example: Purely Ramified Extension with an Inexact Derivation (ex-da702b)

The ring in this example has an order term in its derivation. Therefore, taking a derivative of an

element x is of influence on the relative precision of the image of x.

```
> F<z> := RationalDifferentialField(Rationals());
> FF<z>:=ChangeDerivation(RationalDifferentialField(Rationals()),z);
> RR<DD>:=DifferentialOperatorRing(FF);
> RS<DS>, mpRRtoRS :=Completion(RR,Zeros(z)[1]);
> S<t>:=BaseRing(RS);
> IsDifferentialLaurentSeriesRing(S);
true
> _<T> := PolynomialRing(S);
> E<r>, mp := PurelyRamifiedExtension(T^3-5*t);
> IsDifferentialLaurentSeriesRing(E);
true
> RelativePrecision(E);
60
> RelativePrecisionOfDerivation(E);
60
> Derivation(E)(r);
1/3*r + O(r^61);
> mp(t);
1/5*r^3
> Derivation(S)(t);
t + O(t^21)
> Derivation(E)(mp(t));
1/5*r^3 + O(r^63)
> mp(Derivation(S)(t));
1/5*r^3 + O(r^63)
> x:=t^(-2) +7+t^3 +O(t^15);
> Derivation(S)(x);
-2*t^-2 + 3*t^3 + O(t^15)
> Derivation(E)(mp(x));
-50*r^-6 + 3/125*r^9 + O(r^45)
> mp(Derivation(S)(x));
-50*r^-6 + 3/125*r^9 + O(r^45)
> y := 2*t+O(t^25);
> Derivation(S)(y);
2*t + O(t^21)
> Derivation(E)(mp(y)) eq mp(Derivation(S)(y));
true
> Derivation(E)(mp(y));
2/5*r^3 + O(r^63)
```

121.7 Ideals and Quotient Rings

A differential ideal $I \subset R$ of a differential ring R is an ideal of R that is closed under the derivation of R. However, we consider a differential ideal as an ideal of the underlying ring of R. More specifically, ideals of differential rings are restricted to those rings whose underlying rings are multivariate polynomial rings.

121.7.1 Defining Ideals and Quotient Rings

`DifferentialIdeal(L): [RngDiffElt] -> RngMPol`

Given a sequence L with entries in a differential ring R, return the differential ideal generated by the entries of L as an ideal of the underlying ring of R. The underlying ring of R must be of type `RngMPol`. At first the elements of L may generate an ideal which is not closed under the derivation of R. By adding as many derivatives of the elements to the set of generators of the ideal as needed, one obtains a full set of generators for the calculated differential ideal.

`QuotientRing(R, I): RngDiff, RngMPol -> RngDiff, Map`

Given a differential ring R and a differential ideal I, return the differential quotient ring $Q = R/I$. The derivation of Q is induced by the derivation of R. It maps $Q.i$ to $Q!\delta_R(R.i)$, for $i = 1, 2, \ldots, m$ where m is the number of generators of Q (or R). The induced quotient map from R to Q is also returned.

Example: Differential Ideal and Quotient Ring (ex-effe4a)

```
> P := PolynomialRing(Rationals(),1);
> f := map<P->P | a:->a*Derivative(a,1)>;
> R<T> := DifferentialRing(P, f, Rationals());
> L := [T^2+T-1];
> I := DifferentialIdeal(L);
> I;
Ideal of Polynomial ring of rank 1 over Rational Field
Lexicographical Order
Variables: T
Basis:
[
    T^2 + T - 1,
]
> Q<X>, toQ := QuotientRing(R,I);
> Q;
Differential Ring of Affine Algebra of rank 1 over Rational Field
Lexicographical Order
Variables: X
Quotient relations:
[
    X^2 + X - 1
]
with derivation given by Mapping from: Affine Algebra of rank 1 over Rational
    Field to Affine Algebra of rank 1 over Rational Field given by a rule [no
inverse]
> toQ(T);
X
> Derivative(T^2);
2*T^3
> Derivative(X^2);
X
```

121.7.2 Boolean Operations on Ideals

`IsDifferentialIdeal(R, I): RngDiff, RngMPol -> BoolElt`

Returns `true` if and only if I is a differential ideal of the differential ring R.

121.8 Wronskian Matrix

Let R be a differential ring and let $y_1, y_2, \ldots, y_n$ be elements of R. The *wronskian matrix* of $y_1, y_2, \ldots, y_n$ is defined as the $n \times n$ matrix

$$W(y_1, y_2, \ldots, y_n) = \begin{pmatrix} y_1 & y_2 & \cdots & y_n \\ \delta_R(y_1) & \delta_R(y_2) & \cdots & \delta_R(y_n) \\ \vdots & \vdots & \ddots & \vdots \\ \delta_R^{n-1}(y_1) & \delta_R^{n-1}(y_2) & \cdots & \delta_R^{n-1}(y_n) \end{pmatrix}$$

The *wronskian determinant*, or simply the *wronskian*, of $y_1, y_2, \ldots, y_n$ is the determinant of the wronskian matrix $W(y_1, y_2, \ldots, y_n)$.

`WronskianMatrix(L): [RngDiffElt] -> AlgMatElt`

Given a sequence of differential ring elements L, return the Wronskian matrix of L whose entries are elements of the universe of L.

`WronskianDeterminant(L): [RngDiffElt] -> RngDiffElt, AlgMatElt`

Given a sequence of differential ring elements L, return the determinant of the Wronskian matrix of L as well as the matrix itself.

Example: Wronskian Matrix and Determinant (ex-95dc85)

```
> F<z> := RationalDifferentialField(Rationals());
> WronskianMatrix([1,z,z^2]);
[1 z z^2]
[0 1 2*z]
[0 0 2]
> WronskianDeterminant([1,z^2,1/z]);
6/z
[z z^2 1/z]
[1 2*z -1/z^2]
[0 2 2/z^3]
```

121.9 Differential Operator Rings

121.9.1 Creation

`DifferentialOperatorRing(F): RngDiff -> RngDiffOp`

Returns the differential operator ring over the differential field F.

Example: Creating a Differential Operator Ring (ex-42ff41)

```
> F<z> := RationalDifferentialField(Rationals());
> R := DifferentialOperatorRing(F);
> R;
Differential operator ring over Differential Ring of Algebraic function field
defined over Rational Field by
$.2 - 4711
with derivation given by (1) d(z)
```

`AssignNames(~R, S): RngDiffOp, [MonStgElt]`

Given a differential operator ring R with n indeterminates and a sequence S of n strings, assign the elements of S to the names of the variables of R.

This procedure only changes the names used in the printing of the elements of R.

121.9.2 Creation of Differential Operators

The easiest way to create an element in a given ring is to use the angle bracket construction to attach a name to the indeterminate of the differential operator ring. Other constructions are given below.

`Name(R, i): RngDiffOp, RngIntElt -> RngDiffElt`

`R . i: RngDiffOp, RngIntElt -> RngDiffOpElt`

The i-th indeterminate of the differential ring R, where i must be 1.

`R ! s: RngDiffOp, RngElt -> RngDiffOpElt`

Coerce the element s into the differential operator ring R. Elements that are coercible into R are elements coercible into its underlying ring, sequences, and differential operators defined over the base ring of the coefficient ring of R.

When the base ring of R is an algebraic differential field, elements of other differential operator rings over algebraic differential fields can be coerced into R so long as the underlying rings of the differential fields are the same.

`Zero(R): RngDiffOp -> RngDiffOpElt`

The zero element of the differential operator ring R.

`One(R): RngDiffOp -> RngDiffOpElt`

The identity element of the differential operator ring R.

Example: Coercion into an Operator Ring (ex-facb79)

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> R.1;
D
> R!(1/z);
1/z;
> R![1/2,0,5,z];
z*D^3 + 5*D^2 + 1/2
> S<T> := DifferentialOperatorRing(ChangeDerivation(F,z));
> R!T;
z*D
> S!D;
1/z*T
```

121.10 Structure Operations on Differential Operator Rings

121.10.1 Category and Parent

Differential Operator Rings form the Magma category `RngDiffOp`. The notional power structures exist as parents of differential operator rings.

`Category(R): RngDiffOp -> RngDiffOp`

`Type(R): RngDiffOp -> RngDiffOp`

The category, or type, of the differential operator ring R.

`Parent(R): RngDiffOp -> PowStr`

The power structure of the differential operator ring R.

121.10.2 Related Structures

As outlined in the introduction, a differential operator ring R is of the form $F[D]$, for a differential ring F. The ring F is called the *base ring* or *coefficient ring* of R.

`BaseRing(R): RngDiffOp -> Rng`

`CoefficientRing(R): RngDiffOp -> Rng`

The base ring, or coefficient ring, of the differential operator ring R.

`ConstantRing(R): RngDiffOp -> Rng`

The constant ring of the differential ring operator R.

121.10.3 Derivation and Differential

By construction the variable D of a differential operator ring $F[D]$ is related to the derivation δ_F. That is why δ_F is also considered to be the derivation of R.

`Derivation(R): RngDiffOp -> Map`

The derivation of the differential operator ring R.

`Differential(R): RngDiffOp -> DiffFunElt`

The differential belonging to the derivation of the differential operator ring R. The derivation must have been constructed in such a way that it is defined by a differential.

Example: Related Structures of an Operator Ring (ex-fcdd49)

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> BaseRing(R) eq F;
true
> Derivation(R);
Mapping from: RngDiff: F to RngDiff: F given by a rule [no inverse]
> Differential(R);
(1) d(z)
```

121.10.4 Predicates and Booleans

`R eq F: RngDiffOp, RngDiffOp -> BoolElt`

Returns `true` if and only if the differential operator rings R and F are the same.

`IsIdentical(R, F): RngDiffOp, RngDiffOp -> BoolElt`

Returns `true` if and only if the differential operator rings R and F are identical.

`IsDifferentialOperatorRing(R): . -> BoolElt`

Returns `true` if and only if the given argument is a differential operator ring.

`HasProjectiveDerivation(R): RngDiffOp -> BoolElt`

Returns `true` iff R is defined over a ring F with derivation weakly of the form $(F.1) \cdot d/d(F.1)$.

`HasZeroDerivation(R): RngDiffOp -> BoolElt`

Returns `true` iff the base ring of R is an algebraic differential field or a differential series ring F such that the derivation of R acts as a (weak) zero derivation on $F.1$.

Example: Predicates for Operator Rings (ex-74f01d)

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> IsDifferentialOperatorRing(F);
false
> IsDifferentialOperatorRing(R);
true
> Derivation(R)(z);
1
> HasProjectiveDerivation(R);
false
> HasProjectiveDerivation(ChangeDerivation(R,z));
true
> HasZeroDerivation(R);
false
```

Example: Predicates for Operator Rings over Series (ex-f354a2)

```
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> V<W> := DifferentialOperatorRing(S);
> IsDifferentialOperatorRing(V);
true
> Derivation(V)(t);
t
```

(continues on next page)

(continued from previous page)

```
> HasProjectiveDerivation(V);
true
> HasZeroDerivation(V);
false
> P<Q>, mp := ChangeDerivation(V,3/t);
> IsDifferentialOperatorRing(P);
true
> HasProjectiveDerivation(P);
false
> X<y> := BaseRing(P);
> Q*y;
y*Q + 3
```

121.10.5 Precision

RelativePrecisionOfDerivation(R): RngDiffOp -> RngElt

The relative precision of the derivation of an operator ring over a Laurent series ring.

Example: Relative Precision of an Operator Ring (ex-1465b5)

This example illustrates the relative precision of derivations of differential operatorrings.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> RS<DS> := DifferentialOperatorRing(S);
> RelativePrecisionOfDerivation(RS);
Infinity
> RV<DV> := ChangeDerivation(RS, t^2+O(t^8));
> relprec := RelativePrecisionOfDerivation(RV);
> relprec;
6
> RelativePrecisionOfDerivation(BaseRing(RV)) eq relprec;
true
```

121.11 Element Operations on Differential Operators

121.11.1 Category and Parent

Category(L): RngDiffOpElt -> RngDiffOpElt

Type(L): RngDiffOpElt -> RngDiffOpElt

The category, or type, of the differential operator L.

Parent(L): RngDiffOpElt -> RngDiffOp

The parent of the differential operator L.

121.11.2 Arithmetic

All the usual arithmetic operations are possible for differential operators. It follows from the multiplication rule for differential operators that the multiplication of differential operators is non–commutative.

s + t: RngDiffOpElt, RngDiffOpElt -> RngDiffOpElt

s + t: RngDiffOpElt, RngElt -> RngDiffOpElt

s + t: RngElt, RngDiffOpElt -> RngDiffOpElt

The sum of the two differential operators s and t.

`- s: RngDiffOpElt -> RngDiffOpElt`

The negation of the differential operator s.

`s - t: RngDiffOpElt, RngDiffOpElt -> RngDiffOpElt`

`s - t: RngDiffOpElt, RngElt -> RngDiffOpElt`

`s - t: RngElt, RngDiffOpElt -> RngDiffOpElt`

The difference between the differential operators s and t.

`s * t: RngDiffOpElt, RngDiffOpElt -> RngDiffOpElt`

`s * t: RngDiffOpElt, RngElt -> RngDiffOpElt`

`s * t: RngElt, RngDiffOpElt -> RngDiffOpElt`

The product of the differential operators s and t.

`s ^ n: RngDiffOpElt, RngIntElt -> RngDiffElt`

Given a differential operator s and an integer $n \geq 0$, return the n-th power of s.

Example: Arithmetic of Differential Operators (ex-a0ba79)

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> (z*D-1)*(D+1);
z*D^2 + (z - 1)*D + -1
> (D+1)*(z*D-1);
z*D^2 + z*D + -1
> (D-1/z)^2;
D^2 + -2/z*D + 2/z^2
```

121.11.3 Predicates and Booleans

`s eq t: RngDiffOpElt, RngDiffOpElt -> BoolElt`

Return `true` iff the differential operators s and t are exactly the same.

`IsZero(L): RngDiffOpElt -> BoolElt`

Return `true` iff the differential operator L is the zero element of its parent.

`IsOne(L): RngDiffOpElt -> BoolElt`

Return `true` iff the differential operator L is the unity element of its parent.

`IsMonic(L): RngDiffOpElt -> BoolElt`

Return `true` iff the differential operator L is monic.

`IsWeaklyEqual(L, P): RngDiffOpElt, RngDiffOpElt -> BoolElt`

Returns `true` if and only if the differential operator L is weakly equal to the operator P. This means that the i-th coefficients of L and P should be weakly equal to each other for every $i \in [0..\max(\deg(L), \deg(P))]$.

`IsWeaklyZero(L): RngDiffOpElt -> BoolElt`

Returns `true` if and only if the differential operator $L \in R$ is weakly equal to $R!0$.

`IsWeaklyMonic(L): RngDiffOpElt -> BoolElt`

Returns `true` if and only if the leading coefficient of the differential operator L is weakly equal to 1.

121.11.4 Coefficients and Terms

Differential operators look like univariate polynomials with coefficients in a differential ring. Some of the terminology used for polynomial rings is mimicked for differential operators.

`Eltseq(L): RngDiffOpElt -> SeqEnum`

`Coefficients(L): RngDiffOpElt -> SeqEnum`

Given an operator L with coefficients in R, this function returns the sequence of elements in R, that are the coefficients of L. The sequence is ordered from the constant coefficient to the coefficient of the highest order term of L.

`Coefficient(L, i): RngDiffOpElt, RngIntElt -> RngElt`

Given an operator L with coefficients in R, this function returns the coefficient of the monomial of degree i in L, as an element of R.

`LeadingCoefficient(L): RngDiffOpElt -> RngElt`

Given an operator L with coefficients in R, this function returns the coefficient of the highest order term of L.

`LeadingTerm(L): RngDiffOpElt -> RngDiffOpElt`

The leading term of the differential operator L.

`Terms(L): RngDiffOpElt -> SeqEnum`

Given an operator L with coefficients in R, this function returns the sequence of non–zero coefficients of L as elements of R. The sequence is ordered from the lowest order term to the highest order term in L.

Example: Coefficients and Terms of an Operator (ex-d50a45)

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> L := D^3  + (-4*z + 5)*D + (3*z - 4);
> L;
D^3 + (-4*z + 5)*D + 3*z - 4
> Eltseq(L);
[ 3*z - 4, -4*z + 5, 0, 1 ]
> LeadingTerm(L);
D^3
> Terms(L);
[
    3*z - 4,
    (-4*z + 5)*D,
    D^3
]
```

121.11.5 Order and Degree

`Order(L): RngDiffOpElt -> RngIntElt`

`Degree(L): RngDiffOpElt -> RngIntElt`

Returns the order of the differential operator L. In the case that L is identically 0, the order is defined to be -1.

`WeakOrder(L): RngDiffOpElt -> RngIntElt`

`WeakDegree(L): RngDiffOpElt -> RngIntElt`

If the differential operator L is defined over a differential series ring, then the exponent of the highest coefficient of L that is not weakly 0 is returned.

Example: Order and Degree of Operators (ex-873e6e)

```
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> R<D> := DifferentialOperatorRing(S);
> L := D^2 + 2*t;
> P := O(t)*D^3 + (1+O(t))*D^2  + 2*t;
> Order(L);
```

(continues on next page)

(continued from previous page)

```
2
> Degree(P);
3
> L eq P;
false
> IsWeaklyEqual(L,P);
true
> WeakOrder(P);
2
```

121.11.6 Related Differential Operators

`MonicDifferentialOperator(L): RngDiffOpElt -> RngDiffOpElt`

Given the differential operator L, this function returns the monic differential operator $1/c \cdot L$, where c is the leading coefficient of L.

`Adjoint(L): RngDiffOpElt -> RngDiffOpElt`

Returns the formal adjoint of the differential operator L. The *formal adjoint* of $L = \sum_{i=0}^{n} a_i D^i$ in the differential operator ring $R = F[D]$ over F, is the differential operator $L^* := \sum_{i=0}^{n} (-1)^i D^i * a_i \in R$. It follows from the definition that the orders of L and L^* are the same and that the leading coefficient of L^* is $(-1)^n a_n$.

`Translation(L, e): RngDiffOpElt, RngElt -> RngDiffOpElt, Map`

If R is the parent of the differential operator L and e is a suitable ring element, then the operator in R obtained by replacing $R.1$ by $R.1 + e$ in L is returned. The second argument returned is the translation map on R by e.

`TruncateCoefficients(L): RngDiffOpElt -> RngDiffOpElt`

If L is defined over a differential series ring, then returned is the operator whose coefficients are the truncations of the coefficients of L.

Example: Related Differential Operators (ex-978632)

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> L := z*D^3  + (-4*z + 5)*D + (3*z - 4);
> Order(L);
3
> MonicDifferentialOperator(L);
D^3 + (-4*z + 5)/z*D + (3*z - 4)/z
> Adjoint(L);
-z*D^3 + -3*D^2 + (4*z - 5)*D + 3*z
> trans, mp := Translation(L, 2);
> trans;
z*D^3 + 6*z*D^2 + (8*z + 5)*D + 3*z + 6
```

Example: Truncating Operator Coefficients (ex-32cf07)

```
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> RS<DS> := DifferentialOperatorRing(S);
> L := (5-O(t))*DS^3+(2*t^-1+t^2+O(t^4))*DS - t^-2+t+O(t^3);
> L;
(5 + O(t))*DS^3 + (2*t^-1 + t^2 + O(t^4))*DS + -t^-2 + t + O(t^3)
> TruncateCoefficients(L);
5*DS^3 + (2*t^-1 + t^2)*DS + -t^-2 + t
> L -TruncateCoefficients(L);
O(t)*DS^3 + O(t^4)*DS + O(t^3)
```

121.11.7 Application of Operators

As pointed out in the introduction a differential operator $L = a_n D^n + a_{n-1} D^{n-1} + \cdots + a_1 D + a_0$ in $F[D]$ leads to the differential equation $L(y) = 0$ given by

$$L(y) = a_n \delta_F^n(y) + a_{n-1} \delta_F^{n-1}(y) + \cdots + a_1 \delta_F(y) + a_0 y$$

This notation is formal, but also defines an action of L on any element $y \in F$. The function `Apply` returns the ring element obtained by this action.

`Apply(L, f): RngDiffOpElt, RngElt -> RngElt`

`L(f): RngElt, RngDiffOpElt -> RngElt`

`f @ L: RngElt, RngDiffOpElt -> RngElt`

Given a differential operator L and a ring element f, return the ring element obtained after applying L to f, as an element of the base ring of L. The element f must be coercible into the base ring of L.

Example: Applying an Operator to a Function (ex-1a4536)

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> L := D^2-2/z^2;
> Apply(L, z);
-2/z
> L(z);
-2/z
> Apply(L, z^2);
0
```

121.12 Related Maps

This section is devoted to maps between differential operator rings.

`TranslationMap(R, e): RngDiffOp, RngElt -> Map`

Returns a map on the differential operator ring R that replaces $R.1$ by $R.1 + e$ when applied to a differential operator for some suitable ring element e.

`LiftMap(m, R): Map, RngDiffOp -> Map`

Let $m : F \to M$ be a differential map on differential fields and R a differential operator ring over F. Then this routine lifts the given map to a map on the differential operator rings $R \to S$, where the basefield of S is M.

Example: Maps between Operator Rings (ex-943ab0)

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> transmap := TranslationMap(R, 2 + z);
> Codomain(transmap) eq R;
> transmap(D);
D + z + 2
> transmap(D^2);
D^2 + (2*z + 4)*D + z^2 + 4*z + 5
> P<T> := PolynomialRing(F);
> M<u>, mp := ext<F|T^2+z>;
> liftmap := LiftMap(mp, R);
> Rprime<Dprime> := Codomain(liftmap);
> IsDifferentialOperatorRing(Rprime);
true
```

(continues on next page)

(continued from previous page)

```
> BaseRing(Rprime) eq M;
true
> liftmap(D);
Dprime
> liftmap(R!z);
z
> Derivation(Rprime)(liftmap(z));
1
> Derivation(Rprime)(u);
1/2/z*u
```

121.13 Changing Related Structures

It may happen that certain intrinsics only work for differential operator rings whose derivations are of a specific form, or whose constant fields have to be large enough. Some of the functions available for changing settings of the differential rings or fields can be used to change the desired related structure on the operator ring directly. To alter some of the settings of a differential operator ring, the following functions are available.

ChangeDerivation(R, f): RngDiffOp, RngElt -> RngDiffOp, Map

Returns a differential operator ring isomorphic to R, but whose derivation is given by $f*$ Derivation(R). The ring element f must be non–zero. The isomorphism of R to the new differential ring is also returned. The base ring of the new differential operator ring is isomorphic to the one of R, but it has derivation ChangeDerivation(BaseRing(R)).

ChangeDifferential(R, df): RngDiffOp, DiffFunElt -> RngDiffOp, Map

Returns the differential operator ring with differential df, and whose underlying ring of its basefield coincides with the one of R. The map returned is the bijective map of R into the new operator ring. The base ring of the new differential operator ring is isomorphic to the one of R. However, the returned inclusion map and taking derivatives may not be commutative.

Example: Changing the Differential of an Operator Ring (ex-29b98d)

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> df := Differential(z^3+5);
> RM<DM>, mp := ChangeDifferential(R,df);
> Domain(mp) eq R and Codomain(mp) eq RM;
true
> M<u> := BaseRing(RM);
> IsDifferentialOperatorRing(RM) and IsAlgebraicDifferentialField(M);
true
> mp(RM!z);
u
> mp(D);
3*u^2*DM
> D*z, mp(D*z);
u*D + 1
3*u^3*DM + 1
> DM*u;
u*DM + 1/3/u^2
> Differential(RM);
(3*u^2) d(u)
```

ConstantFieldExtension(R, C): RngDiffOp, Fld -> RngDiffOp, Map

Returns the ring of differential operators with base ring isomorphic to that of the differential

operator ring R, but whose constant field is C. The derivation is extended over the new constant field. The second argument returned is the map from R to the new operator ring.

PurelyRamifiedExtension(R, f): RngDiffOp, RngUPolElt -> RngDiffOp, Map

When R is a differential operator ring over a differential ring F, this function returns an operator ring over the purely ramified extension of F, as induced by the polynomial f. The polynomial f is of the form $X^n - a \cdot (F.1)$ for some constant element a in F and positive integer n.

Example: Purely Ramified Extension of an Operator Ring (ex-43ec30)

```
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> R<D> := DifferentialOperatorRing(S);
> _<T> := PolynomialRing(S);
> Rext<Dext>, mp := PurelyRamifiedExtension(R, T^7-3*t);
> Sext<text> := BaseRing(Rext);
> Domain(mp) eq R and Codomain(mp) eq Rext;
true
> IsDifferentialLaurentSeriesRing(Sext);
true
> BaseRing(Sext) eq S;
true
> RelativePrecision(Sext) eq 7*RelativePrecision(S);
true
> D*t;
t*D + t
> mp(D);
Dext
> mp(R!t) eq Rext!(1/3*text^7);
true
> Dext*text;
text*Dext + 1/7*text
```

Completion(R, p): RngDiffOp, PlcFunElt -> RngDiffOp, Map

```
Precision: RngIntElt                    Default: Infinity()
```

Returns the operator ring $\tilde{R}$, whose base ring is the completion of the base ring of the operator ring R w.r.t. the place p. The second return value is the natural embedding of R into $\tilde{R}$. The precision of the base ring of $\tilde{R}$ can be set by setting `Precision` upon creation. If no precision is set, a default value for the precision is taken.

Localization(R, p): RngDiffOp, PlcFunElt -> RngDiffOp, Map, PlcFunElt

Returns the operator ring whose differential has valuation -1 at p, with derivation $t \cdot d/dt$, where t is the uniformizing element at the place p. The natural map between the operator rings, and the induced image of p are also returned.

Localization(L, p): RngDiffOpElt, PlcFunElt -> RngDiffOpElt, Map, PlcFunElt

Given the differential operator L over an algebraic differential field, returns the localized operator of L at the place p. The embedding map between the parents as well as the induced image of the place are also returned.

Localization(R): RngDiffOp -> RngDiffOp, Map

Given a differential operator ring R over the differential Laurent series ring $C((t))$, returns the operator ring whose derivation is of the form $t \cdot d/dt$, and the natural map between the operator rings.

Localization(L): RngDiffOpElt -> RngDiffOpElt, Map

Given the differential operator L over a differential series ring, returns the localized operator of L and the embedding map between the parents.

Example: Changing Attributes of an Operator Ring over Series (ex-7675d6)

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> R<D> := DifferentialOperatorRing(S);
> D*t;
t*D + t
> Rprime<Dprime>, mp := ChangeDerivation(R,t^2);
> Fprime<tprime> := BaseRing(Rprime);
> mp;
Mapping from: RngDiffOp: R to RngDiffOp: Rprime given by a rule
> Dprime*tprime;
tprime*Dprime + tprime^3
> P<T> := PolynomialRing(Rationals());
> Cext := ext<Rationals()|T^2+1>;
> Rext<Dext>, mp := ConstantFieldExtension(R,Cext);
> Cext eq ConstantRing(BaseRing(Rext));
true
> mp(D);
Dext
```

Example: Completion of an Operator Ring (ex-397606)

This examples illustrates how to use `Completion` on operator rings.

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> pl := Zeros(z)[1];
> Rcompl<Dcompl>, mp := Completion(R,pl);
> IsDifferentialOperatorRing(Rcompl);
true
> S<t> := BaseRing(Rcompl);
> IsDifferentialLaurentSeriesRing(S);
true
> mp(D);
Dcompl
> Dcompl*t;
t*Dcompl + 1
```

Example: Localization of an Operator Ring (ex-885bd3)

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> pl := Zeros(z-1)[1];
> Rloc<Dloc>, mp, place:=  Localization(R,pl);
> Domain(mp) eq R, Codomain(mp) eq Rloc;
true true
> place;
(z - 1)
> Differential(BaseRing(Rloc));
(1/(z - 1)) d(z)
> mp(D);
1/(z - 1)*Dloc
> Dloc*(z-1);
(z - 1)*Dloc + z - 1
> L := D + z;
> Lloc, mp, place := Localization(L,Zeros(z)[1]);
> Lloc;
1/z*$.1 + z
```

121.14 Euclidean Algorithms, GCDs, and LCMs

A ring of differential operators shares many properties with a univariate polynomial ring. Two of them are GCD and LCM algorithms. However, a consequence of the non–commutative multiplication of a differential operator ring is that the GCD and LCM algorithms cannot be used directly. For instance, in the euclidean algorithm multiplication of the quotient can be done on the left or the right. Therefore one needs to specify the direction of the multiplication in the GCD and LCM algorithms for differential operator rings.

121.14.1 Euclidean Right and Left Division

`EuclideanRightDivision(N, D): RngDiffOpElt, RngDiffOpElt -> RngDiffOpElt, RngDiffOpElt`

Given differential operators N and D, return two differential operators Q and R, such that $N = Q \cdot D + R$, with `Degree(R) < Degree(D)`. An error occurs if D is 0.

`EuclideanLeftDivision(D, N): RngDiffOpElt, RngDiffOpElt -> RngDiffOpElt, RngDiffOpElt`

Given differential operators D and N, return two differential operators Q and R, such that $N = D \cdot Q + R$, with `Degree(R) < Degree(D)`. An error occurs if D is 0.

Example: Euclidean Right and Left Division (ex-872ed1)

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> L1 := D;
> L2 := (D-3)*(D+z);
> EuclideanRightDivision(L1, L2);
0
D
> Q, R := EuclideanRightDivision(L2, L1);
> Q, R;
D + z - 3
-3*z + 1
> L2 eq Q*L1+R;
true
> EuclideanLeftDivision(L2, L1);
0
D
> S, T := EuclideanLeftDivision(L1, L2);
> S, T;
D + z - 3
-3*z
> L2 eq L1*S+T;
true
```

121.14.2 Greatest Common Right and Left Divisors

`GreatestCommonRightDivisor(A, B): RngDiffOpElt, RngDiffOpElt -> RngDiffOpElt`

`GCRD(A, B): RngDiffOpElt, RngDiffOpElt -> RngDiffOpElt`

Given two differential operators $A, B \in R$, return the unique monic differential operator $G \in R$ that generates the left ideal $RA + RB$.

`ExtendedGreatestCommonRightDivisor(A, B): RngDiffOpElt, RngDiffOpElt -> RngDiffOpElt, RngDiffOpElt, RngDiffOpElt`

Given two differential operators $A, B \in R$, this function returns three operators $G, U, V \in R$, that satisfy $U \cdot A + V \cdot B = G$. The differential operator G is the unique monic right GCD of A and B.

`GreatestCommonLeftDivisor(A, B): RngDiffOpElt, RngDiffOpElt -> RngDiffOpElt`

`GCLD(A, B): RngDiffOpElt, RngDiffOpElt -> RngDiffOpElt`

Given two differential operators $A, B \in R$, return the unique monic differential operator $G \in R$ that generates the right ideal $AR + BR$.

`ExtendedGreatestCommonLeftDivisor(A, B): RngDiffOpElt, RngDiffOpElt -> RngDiffOpElt, RngDiffOpElt, RngDiffOpElt`

Given two differential operators $A, B \in R$, this function returns three operators $G, U, V \in R$, that satisfy $A \cdot U + B \cdot V = G$. The differential operator G is the unique monic left GCD of A and B.

Example: Greatest Common Right and Left Divisors (ex-5508d6)

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> L1 := D^3+z*D^2+D-z;
> L2 := D^2+(z-3)*D-3*z+1;
> GreatestCommonRightDivisor(L1, L2);
D + z
> GreatestCommonRightDivisor(L1, L2) eq GCRD(L1, L2);
true
> G, U, V :=ExtendedGreatestCommonRightDivisor(L1, L2);
> G, U, V;
D + z
1/8
-1/8*D + -3/8
> G eq U*L1+V*L2;
true
> GreatestCommonLeftDivisor(L1, L2);
1
> GCLD(L2,L2*L1) eq L2;
true
```

121.14.3 Least Common Left Multiples

`LeastCommonLeftMultiple(L): RngDiffOpElt -> RngDiffOpElt`

Let $L = D - r$ be a monic operator of degree 1 in $R = F[D]$. Return the least common left multiple of L and all its conjugates over the base ring of F, with respect to the coercion of this base ring into F.

`LeastCommonLeftMultiple(A, B): RngDiffOpElt, RngDiffOpElt -> RngDiffOpElt`

`LCLM(A, B): RngDiffOpElt, RngDiffOpElt -> RngDiffOpElt`

Given two differential operators $A, B \in R$, return the unique monic differential operator $L \in R$, that generates the left ideal $RA \cap RB$. The order of the least common multiple of A and B is at most `Order(A)+ Order(B)`.

`ExtendedLeastCommonLeftMultiple(A, B): RngDiffOpElt, RngDiffOpElt -> RngDiffOpElt, RngDiffOpElt, RngDiffOpElt`

Given two differential operators $A, B \in R$, return three operators $L, U, V \in R$, that satisfy $L = U \cdot A = V \cdot B$. The differential operator L is the unique monic left LCM of A and B.

`ExtendedLeastCommonLeftMultiple(S): [RngDiffOpElt] -> RngDiffOpElt, SeqEnum`

Given the non–empty sequence of differential operators S, this function returns the unique monic left LCM L of the entries of S, as well as a sequence Q of length $\#S$, satisfying $L = Q[i] \cdot S[i]$ for $i = 1, 2, \ldots, \#S$.

Example: Least Common Left Multiples (ex-22d5f6)

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
```

(continues on next page)

(continued from previous page)

```
> LCLM(D, D-z);
D^2 + (-z^2 - 1)/z*D
> L1 := D^3+z*D^2+D-z;
> L2 := D^2+(z-3)*D-3*z+1;
> LeastCommonLeftMultiple(L1, L2);
D^4 + (z - 3)*D^3 + (-3*z + 2)*D^2 + (-z - 3)*D + 3*z - 1
> L, U, V := ExtendedLeastCommonLeftMultiple(L1, L2);
> L, U, V;
D^4 + (z - 3)*D^3 + (-3*z + 2)*D^2 + (-z - 3)*D + 3*z - 1
D + -3
D^2 + -1
> L eq U*L1;
true
> L eq V*L2;
true
> L, Q := ExtendedLeastCommonLeftMultiple([D,D+1,z*D+1]);
> L;
D^3 + (z^2 - 6)/(z^2 - 2*z)*D^2 + (2*z - 6)/(z^2 - 2*z)*D
> Q[3]*(z*D+1) eq L;
true
```

Example: Least Common Left Multiple of Conjugates (ex-138207)

```
> F<z> := RationalDifferentialField(Rationals());
> P<T> := PolynomialRing(F);
> M<u> := ext<F|T^2+T+1>;
> RM<DM> := DifferentialOperatorRing(M);
> LeastCommonLeftMultiple(DM-u^2);
DM^2 + DM + 1
> lclm := LeastCommonLeftMultiple(DM-u+1);
DM^2 + 3*DM + 3
> EuclideanRightDivision(lclm, DM-u+1);
DM + u + 2
0
> N<v>, mp := ext<F|T^2-z>;
> RN<DN> := DifferentialOperatorRing(N);
> lclm := LeastCommonLeftMultiple(DN-v);
> lclm;
DN^2 + -1/2/z*DN + -z
> LeastCommonLeftMultiple(DN-v, DN+v) eq lclm;
true
> EuclideanRightDivision(lclm,DN-v);
DN + v - 1/2/z
0
> EuclideanRightDivision(lclm,DN+v);
DN + -v - 1/2/z
0
```

121.15 Related Matrices

The *companion matrix* of a monic linear differential operator

$$D^n + a_{n-1}D^{n-1} + \cdots + a_0 \in F[D]$$

is defined as the $n \times n$ matrix

$$\begin{pmatrix} 0 & 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 1 \\ -a_0 & -a_1 & -a_2 & -a_3 & \dots & -a_{n-1} \end{pmatrix}$$

CompanionMatrix(L): RngDiffOpElt -> AlgMatElt

Returns the companion matrix of the monic differential operator L.

Example: Companion Matrix of an Operator (ex-62793f)

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> L := D^3-z*D^2+2*D+5;
> CompanionMatrix(L);
[0 1 0]
[0 0 1]
[-5 -2 z]
```

121.16 Singular Places and Indicial Polynomials

All functions treated in this section concern differential operator rings defined over a function field of transcendence degree one. They all require that the derivation of such an operator ring $F[D]$ over F is defined with respect to a differential.

Any solution of a differential operator $L \in F[D]$ is also a solution of the operator $c \cdot L$, for any non–zero $c \in F$. For considering solutions we may therefore consider L to be monic of the form

$$L = D^n + a_{n-1}D^{n-1} + \cdots + a_0,$$

with coefficients $a_i \in F$ for $i = 1, 2, \dots n-1$. Each of the coefficients is a rational function in F. They play an important role in the rational solutions of $L(y) = 0$ in F that come to expression in so-called regular and singular places.

121.16.1 Singular Places

Given a place (q) in the set of places of F, a putative rational solution $y \in F$ of $L(y) = 0$ has a q-adic expansion

$$y = y_\alpha \, q^\alpha + y_{\alpha+1} \, q^{\alpha+1} + \dots$$

It has a pole at the place (q) if the valuation α is negative. After substituting this solution in $L(y)$ it becomes clear that there is only a finite number of places which can occur as poles of an arbitrary solution of $Ly = 0$. Such a place is either a pole of one of the coefficients a_i or a zero or a pole of the differential ω of $F[D]$. There exists a classification for the poles of solutions of $L(y) = 0$.

Given a place (q) and local parameter t at (q), the differential operator can be rewritten as a differential operator $\tilde{L} \in \tilde{F}[\tilde{D}]$, with $\tilde{F} \cong F$, the valuation of whose differential at (q) is 0. The place (q) is defined to be a *singular place* of L, if one of the coefficients of $\tilde{L}$ has negative valuation at (q). Places that are not singular are called *regular*.

There are two kinds of singular places of a differential operator; the *regular singular* places and the *irregular singular* places. With the notation as above, a singular place (q) of L is regular singular if the valuation of the coefficient of $(\tilde{D})^i$ in $\tilde{L}$ is at most $i - n$ for every $i \in \{0, 1, \dots, n-1\}$. Otherwise it is irregular singular.

`IsRegularPlace(L, p): RngDiffOpElt, PlcFunElt -> BoolElt`

Returns `true` iff the place p is a regular place of the differential operator L. If p is not a regular place of L `false` is returned. This function only works for operators whose derivation is defined by a differential.

`IsRegularSingularPlace(L, p): RngDiffOpElt, PlcFunElt -> BoolElt`

Returns `true` iff the place p is a regular singular place of the differential operator L. If p is not a regular singular place of L `false` is returned. This function only works for operators whose derivation is defined by a differential.

`IsIrregularSingularPlace(L, p): RngDiffOpElt, PlcFunElt -> BoolElt`

Returns `true` iff the place p is an irregular singular place of the differential operator L. If p is not an irregular singular place of L `false` is returned. This function only works for operators whose derivation is defined by a differential.

`SetsOfSingularPlaces(L): RngDiffOpElt -> SetEnum, SetEnum`

Two sets are returned. The first set contains precisely all regular singular places of the differential operator L. The second set consists of all irregular singular places of L. This function only works for operators whose derivation is defined by a differential.

`IsFuchsianOperator(L): RngDiffOpElt -> BoolElt, SetEnum`

Returns `true` iff the differential operator L is Fuchsian (i.e. if all singular places of L are regular singular). If L is not Fuchsian, `false` is returned. Secondly, the set of all singular places of L, is returned only if L is a Fuchsian differential operator. This function only works for operators whose derivation is defined by a differential.

`IsRegularSingularOperator(L): RngDiffOpElt -> BoolElt, SetEnum`

Returns `true` if and only if the differential operator L is regular singular. The operator may be defined over a differential Laurent series ring, F say. In this case being regular singular means that the operator must be regular singular at $F.1$. Then also there is no second argument returned. In the case that the derivation is defined with respect to a differential, then the values from `IsFuchsianOperator(L)` are returned.

Example: Singular Places of an Operator (ex-3dba78)

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> H := (z^2-z)*D^2+(3*z-6)*D+1;
> IsRegularPlace(H, Zeros(z)[1]);
false
> IsRegularSingularPlace(H, Zeros(z)[1]);
true
> SetsOfSingularPlaces(H);
{ (1/z), (z - 1), (z) }
{}
> IsFuchsianOperator(H);
true { (z), (1/z), (z - 1) }
> IsFuchsianOperator(D^2-1/z^3);
false
```

Example: Regular Singular Operators over Series (ex-5e64d4)

```
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> R<D> := DifferentialOperatorRing(S);
> IsRegularSingularOperator(D^2 -t*D+2);
true
> IsRegularSingularOperator(D^2 +3);
true
> IsRegularSingularOperator(D^2 +3 +O(t));
```

(continues on next page)

(continued from previous page)

```
true
> IsRegularSingularOperator(D^2 +3*t^(-1));
false
```

121.16.2 Indicial Polynomials

For the definition of a indicial polynomial at a place, we refer to Section 4.1 in [vdPS03].

`IndicialPolynomial(L, p): RngDiffOpElt, PlcFunElt -> RngElt`

Returns the monic indicial polynomial of the differential operator L at the place p. This function only works for operators whose derivation is defined by a differential and whose base ring has one generator.

Example: Indicial Polynomial of an Operator (ex-eadf1f)

```
> F<z> := RationalDifferentialField(Rationals());
> _<T> := PolynomialRing(Rationals());
> R<D> := DifferentialOperatorRing(F);
> H := (z^2-z)*D^2+(3*z-6)*D+1;
> IndicialPolynomial(H, Zeros(z)[1]);
T^2 + 5*T
> IndicialPolynomial(H, Zeros(z-1)[1]);
T^2 - 4*T
> IndicialPolynomial(H, Zeros(1/z)[1]);
T^2 - 2*T + 1
> Apply(H, (z-1)^4/z^5);
0
```

121.17 Rational Solutions

`RationalSolutions(L): RngDiffOpElt -> SeqEnum`

Given a differential operator L, a basis of the nullspace of rational solutions of $L(y) = 0$ in F is returned as a sequence of basis elements. This function only works for operators whose derivation is defined by a differential. The algorithm that is used is described in Section 4.1 of [vdPS03].

`HasRationalSolutions(L, g): RngDiffOpElt, RngElt -> BoolElt, RngElt, SeqEnum`

Given a differential operator L with coefficients in F and an element g of F, return `true` if there is an element $y \in F$ satisfying $L(y) = g$. If such a solution exists a particular solution in F and the basis of the nullspace of rational solutions in F are also returned. If there is no solution, only `false` is returned. This function only works for operators whose derivation is defined by a differential.

Example: Rational Solutions of an Operator (ex-a82ef2)

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> H := (z^2-z)*D^2+(3*z-6)*D+1;
> RationalSolutions(H);
[ (z^4 - 4*z^3 + 6*z^2 - 4*z + 1)/z^5 ]
> L := D^2-6/z^2;
> RationalSolutions(L);
[ z^3, 1/z^2 ]
> Apply(L, z^3+1/z^2);
0
> HasRationalSolutions(L, 6/z);
true -z [ z^3, 1/z^2 ]
```

(continues on next page)

(continued from previous page)

```
> L(-z);
6/z
```

121.18 Newton Polygons

The Newton polygon of a differential operator L and its Newton polynomials can be used to factorize L. A classical example of the Newton polygon uses the derivation $z \cdot d/dz$, where z is a generator of the basefield of L. This Newton polygon is known as *the* Newton polygon of L. Its definition, as used in Magma, is given in §3 of [vH97b] and is applicable to operators over a Laurent series ring with generator z, as well as to operators over fields for which a set of places exist. For fields in the latter category it is however not necessary to restrict to the derivation $z \cdot d/dz$ based at $z = 0$ (in other words: at the place (z)). More generally, the Newton polygon of L at the place (p) is the Newton polygon at $t = 0$ after rewriting L as a differential operator $\tilde{L}$ in a local parameter t of (p), such that derivation of $\tilde{L}$ is of the form $t \cdot d/dt$.

`NewtonPolygon(L): RngDiffOpElt -> NwtnPgon, RingDiffOpElt`

Returns the Newton Polygon of the differential operator L over a differential Laurent series ring. This means that for the computation of the Newton polygon L may have had to be rewritten as a differential operator $\tilde{L}$ over a differential Laurent series ring $C((t))$, say, such that $\tilde{L}$ has derivation $t \cdot d/dt$. The second argument returned is the operator $\tilde{L}$.

`NewtonPolygon(L, p): RngDiffOpElt, PlcFunElt -> NwtnPgon, RingDiffOpElt`

Returns the Newton polygon of the differential operator L at the place p. The derivation of L must be defined with respect to a differential and the base ring of L should have one generator. For the computation of the Newton polygon another differential operator $\tilde{L}$, say, may have had to be calculated. The differential of the derivation of $\tilde{L}$ has valuation -1 at the place p. The differential operator $\tilde{L}$ is also returned.

`NewtonPolynomial(F): NwtnPgonFace -> RngUPolElt`

Returns the Newton polynomial of the face F of a Newton polygon. The Newton polygon must have been created with respect to a differential operator. The Newton polynomial depends on a uniformizing element, therefore, its variable is well–defined up to scalar multiplication by a non–zero element. The definition of the Newton polynomial of a face that is used by Magma, is given in Section 3 of [vH97b].

`NewtonPolynomials(L): RngDiffOpElt -> SeqEnum, SeqEnum`

Returns all Newton polynomials of L with respect to the faces of its Newton polygon. The second argument returned is the corresponding slopes.

Example: Newton Polygon of an Operator `(ex-893538)`

```
> K := RationalDifferentialField(Rationals());
> F<z> := ChangeDerivation(K, K.1);
> Differential(F);
(1/z) d(z)
> R<D> := DifferentialOperatorRing(F);
> L := 10*z*D^2+3*D-1;
> npgon, op := NewtonPolygon(L, Zeros(z)[1]);
> npgon;
Newton Polygon of 10*z*$.1^2 + 3*$.1 - 1 over Algebraic function field
defined over Rational Field by
$.2 - 4711 at (z)
> op;
10*z*D^2 + 3*D + -1
> faces:= Faces(npgon);
> faces;
```

(continues on next page)

(continued from previous page)

```
[ <0, 1, 0>, <-1, 1, -1> ]
> _<T> := PolynomialRing(Rationals());
> NewtonPolynomial(faces[1]);
3*T - 1
> NewtonPolynomial(faces[2]);
10*T + 3
```

Example: Newton Polygon at Infinity (ex-692056)

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> L := D^2+z*D-3*z^2;
> npgon, op := NewtonPolygon(L, Zeros(1/z)[1]);
> op;
1/z^2*$.1^2 + (-z^2 + 1)/z^2*$.1 + -3*z^2
> Differential(Parent(op));
(-1/z) d(z)
> Valuation($1,Zeros(1/z)[1]);
-1
> faces:= Faces(npgon);
> faces;
[ <-2, 1, -2> ]
> _<T> := PolynomialRing(Rationals());
> NewtonPolynomial(faces[1]);
T^2 - T - 3
```

Example: Newton Polygon over a Laurent Series Ring (ex-312e21)

This example corresponds to Examples 3.46 and 3.49.2 from [vdPS03].

```
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> R<D> := DifferentialOperatorRing(S);
> L := t*D^2+D-1;
> npgon, op := NewtonPolygon(L);
> L eq op;
true
> Faces(npgon);
[ <0, 1, 0>, <-1, 1, -1> ]
> _<T> := PolynomialRing(Rationals());
> NewtonPolynomials(L);
[
    T - 1,
    T + 1
]
[ 0, 1 ]
> L := D^2+(1/t^2+1/t)*D+(1/t^3-2/t^2);
> npgon, op := NewtonPolygon(L);
> L eq op;
true
> NewtonPolynomials(L);
[
    T + 1,
    T + 1
]
[ 1, 2 ]
```

121.19 Symmetric Powers

`SymmetricPower(L, m): RngDiffOpElt, RngIntElt -> RngDiffOpElt`

Returns the m-th symmetric power of the differential operator L as an element of the parent of L. The symmetric power is monic where possible. If n denotes the order of L, then the degree of the m-th symmetric power of L is at most $\binom{n+m-1}{n-1}$. The algorithm that is used is based on algorithms given in [BMW97].

Example: Symmetric Powers of Operators (ex-6160ae)

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> SymmetricPower(D^2, 3);
D^4
> SymmetricPower(D^3-1, 2);
D^6 + -7*D^3 + -8
```

121.20 Differential Operators of Algebraic Functions

An algebraic function g in a differential field extension M/F satisfies a linear differential equation $L(y) = 0$ with coefficients in $F \subset M$. If the minimal polynomial of g over F is of degree n, then the order of L is at most n.

`DifferentialOperator(f): RngUPolElt -> RngDiffOpElt`

Given the irreducible polynomial $f(X) \in F[X]$, return the monic differential operator over F of minimal degree to which a formal root of f is a solution. The field F must be a differential field. The base ring of the created differential operator is F.

The algorithm used in this function is straightforward. If g is a root of an irreducible polynomial $f(X) \in F[X]$, where F is a differential field, then $f(g) = 0$ induces a unique derivation on g. The field $M = F(g)$ is an algebraic differential field extension of F containing all derivatives of g. If n is the degree of the polynomial f, then M/F is a field extension of degree n. This implies that there must be at least one non–trivial linear relation between $g, \delta_M(g), \ldots, \delta_M^n(g)$. The linear relation between these elements involving the lowest powers of δ_M^i gives exactly the desired monic differential operator after a suitable normalization.

Example: Differential Operator of an Algebraic Function (ex-a16c5a)

```
> F<z> := RationalDifferentialField(Rationals());
> _<X> := PolynomialRing(F);
> f := X^3-z;
> L := DifferentialOperator(f);
> L;
$.1 + -1/3/z
> M<alpha> := ext<F|f>;
> R<D> := DifferentialOperatorRing(M);
> Apply(R!L,alpha);
0
```

121.21 Factorisation of Operators over Differential Laurent Series Rings

When factoring a non-trivial linear differential operator in

$$P[\delta] := k((t))[\delta],$$

with constant field k, differential Laurent Series ring $k((t))$ in t, and derivation δ, one at least wants to compute two operators L and R in P such that $f = L \cdot R$. It is important to distinguish the left and right hand operators (L and R) as such as multiplication generally is non-commutative. When considering differential operators in the operator ring P, it is common to consider δ to be $t \cdot d/dt$. This specific form has the advantage that the degree of $\delta \cdot t$ in t remains one, since $\delta \cdot t = t \cdot \delta + t$. This specific derivation is called the *projective derivation* and we consider this derivation in the rest of this section. In this sense powers of t are eigenvectors under the application of the derivation.

A possible approach for factoring a linear operator would be to compute all non-constant irreducible right hand factors and then use recursion on the appropriate left hand factors. However, this causes a problem as there may be infinitely many factorisations. For instance $\delta^2 - \delta$ has infinitely many factorisations (parametrisations) $(\delta - c/(t+c))(\delta - t/(t+c))$ for any $c \in P^1(k)$. The approach we take to find right hand factors follows [vH97b], and chooses a canonical representative from a class of right hand factors. The obtained representatives can be used in the factorisation of linear differential operators over a rational function field, see [vH97a].

A non-trivial linear differential operator f in P acts on the solution space V of all differential operators in the differential closure of $k((t))$. This action is $\bar{k}$-linear and surjective on V. The kernel $V(f)$ of this map has dimension equal to the order of the operator. The general solution space V can be split into a direct sum of smaller $\bar{k}$-vector spaces V_e. These are minimal such that $f(V_e) \subset V_e$ for every operator in P. Its kernel $V_e(f)$ consists of all solutions of $f(y) = 0$ in V_e. As a consequence the solution space of the operator then is $V(f) = \oplus_e V_e(f)$. For each of the e with non-trivial solution space of f, the idea now is to find an irreducible right hand factor of f in $k((t))[\delta]$ that annihilates $V_e(f)$.

121.21.1 Slope Valuation of an Operator

The Newton Polynomial of an operator f and its slopes contain useful information regarding its factorisation possibilities. It was proved by Malgrange that an operator over a Laurent series ring is reducible if its Newton polygon has at least two slopes. When on the other hand the Newton polygon has one positive slope, and the accompanying Newton polynomial has two relatively prime factors, then the operator is irreducible as well. It can be shown that an irreducible right hand factor has an irreducible Newton polynomial that divides the Newton polynomial of f. When an appropriate irreducible factor of the polynomial of f is taken, One can start building a right hand factor operator with coefficients of a certain precision that has exactly the irreducible factor as its Newton polynomial. Subsequently one can try to lift the coefficients to a better, possibly predescribed precision.

Various measures for the precision are possible, for instance the absolute precision of the coefficients is common. Another valuation metric is or one related to the slope of the Newton polynomial. It is defined as follows. Assume that $P := k((t))[\delta]$ has projective derivation $t \cdot d/dt$, Let s be a rational with numerator n and denominator d such that $\gcd(n, d) = 1$ and $d > 0$ hold. Then the *slope valuation* of a monomial $ct^i\delta^j$, $c \in k$
$\{0\}$, with respect to s is defined as $V_s(ct^i\delta^j) := id - jn$. The *slope valuation* of the operator $L \in P$ with respect to s subsequently is defined as the minimum of the slope valuation of each non-zero monomial in L w.r.t s. Commonly s is a slope of the Newton polynomial of L. If the operator is the zero operator, the slope valuation is defined to be infinite.

`SlopeValuation(L, s): RngDiffOpElt, RngElt -> FldRatElt`

Returns the valuation of L, with respect to the rational slope s, when the derivation of L is projective.

Example: Slope Valuation of an Operator (ex-402dde)

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> P<D>:=DifferentialOperatorRing(S);
> L:=t^(-2)*D^3+t^7;
> SlopeValuation(L,0);
-2
> SlopeValuation(L,1/2);
-7
> SlopeValuation(L,5);
```

(continues on next page)

(continued from previous page)

```
-17
> L:=(0+O(t^6))*D;
> SlopeValuation(L,0);
Infinity
> Valuation(0+O(t^6));
6
> SlopeValuation(P!0,3);
Infinity
```

121.21.2 Coprime Index 1 and LCLM Factorisation

Coprime index 1 factorisation and LCLM factorisation are two factorisation methods that use similar factorisation machinery, but may result in different factorisations, see [vH97b]. Given an operator $f \in k((t))[\delta]$, both compute a right hand factor with respect to each distinct irreducible factor of the Newton polynomial of f. A local lifting procedure with respect to the slope valuation metric is performed to obtain the coefficients of the right hand factors up to a certain accuracy. In addition, no intermediate differential field extensions of $k((t))$ are used.

The coprime index 1 algorithm does not factor an operator f that has Newton polynomial of the form p^n, $n \geq 2$, with slope $s > 0$. The sum of the degrees of the obtained right hand factors of f may be less than the degree of f itself. Their least common left divisor M divides f on the right (i.e. $f = N * M$) with a kernel of dimension less or equal to $\deg(f)$. The LCLM algorithm, on the other hand, produces a set of right hand factors of a monic operator f whose LCLM is exactly f up to some precision.

Factorisation(L): RngDiffOpElt -> SeqEnum, SeqEnum

Factorization(L): RngDiffOpElt -> SeqEnum, SeqEnum

```
Precision: RngIntElt                    Default: -1
Algorithm: MonStgElt                    Default: "Default"
```

Returns a sequence M of operator sequences $[A, B]$ such that $L = A \cdot B$ holds, and B does not have a non-trivial coprime index 1 or LCLM factorisation. As the algorithm sometimes cannot conclude if a right hand factor is irreducible, a second sequence entry $N[i]$ states `True` if the right hand factor $M[i][2]$ is undisputedly irreducible. The optional argument for the precision is the accuracy up to which the lifting procedure would be performed. The default accuracy is the relative precision of the basering of L. The algorithms used can be either "LCLM" or "CoprimeIndexOne". The algorithms used are based on various algorithms in [vH97b].

Example: Operator with Infinitely Many Factorisations (ex-502230)

The operator $t^2 \cdot d/dt^2 - t \cdot d/dt$ with coefficients in Q has infinitely many factorisations, since it can be written as $(t \cdot d/dt - c/(t+c)) \cdot (t \cdot d/dt - t/(t+c))$ for any rational number c. The factorisation code chooses a canonical right hand factor.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> R<D>:=DifferentialOperatorRing(S);
> L := D^2 -D;
> factsL,blsL:=Factorisation(L:Algorithm:="LCLM");
Ring precision as default precision taken.
Performing coprime index 1 LCLM factorisation.
The number of slopes of the Newton polynomial: 1
> (#factsL eq #blsL) and (#factsL eq 1);
true
> blsL;
[ false ]
> factsL[1];
[
```

(continues on next page)

(continued from previous page)

```
    1,
    D^2 + -1*D
]
> factsL,blsL:=Factorization(L:Algorithm:="CoprimeIndexOne");
Ring precision as default precision taken.
Performing coprime index 1 factorisation.
> (#factsL eq #blsL) and (#factsL eq 1);
true
> blsL;
[ true ]
> factsL[1];
[
    D,
    D + -1
]
```

Example: Comparing LCLM and Coprime Index 1 Factorisations (ex-6f1759)

This example corresponds to Example 3.46 in [vdPS03].

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> R<D>:=DifferentialOperatorRing(S);
> L:=D*(D+1/t);
> L;
D^2 + t^-1*D + -t^-1
> factsL,blsL:=Factorisation(L:Precision:=4);
Performing coprime index 1 LCLM factorisation.
> blsL;
[ true, true ]
> #factsL eq 2;
true
> factsL[1];
[
    D + t^-1 + 1 - t + 3*t^2 + O(t^3),
    D + -1 + t - 3*t^2 + 13*t^3 + O(t^4)
]
> factsL[2];
[
    D,
    D + t^-1
]
> factsL:=Factorisation(L:Algorithm:="CoprimeIndexOne");
Ring precision as default precision taken.
Performing coprime index 1 factorisation.
> #factsL eq 2;
true
> blsL;
[ true, true ]
> [v[1]*v[2]:v in factsL];
[
    D^2 + (t^-1 + O(t^19))*D + -t^-1 + O(t^19),
    D^2 + t^-1*D + -t^-1
]
```

Example: Factorising a Product of First-Order Operators (ex-6434dc)

This example corresponds to Example 3.49 in [vdPS03].

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> R<D>:=DifferentialOperatorRing(S);
> L:=(D+1/t)*(D+1/t^2);
> L;
D^2 + (t^-2 + t^-1)*D + t^-3 - 2*t^-2
> factsL, blsL:=Factorisation(L:Algorithm:="CoprimeIndexOne",Precision:=6);
Performing coprime index 1 factorisation.
> blsL;
[ true, true ]
> #factsL;
2
> factsL[1];
[
    D + t^-2 + 2 + t - 3*t^2 - 8*t^3 + O(t^4),
    D + t^-1 - 2 - t + 3*t^2 + 8*t^3 - 9*t^4 + O(t^5)
]
> factsL[2];
[
    D + t^-1,
    D + t^-2
]
> [v[1]*v[2] :v in factsL];
[
    D^2 + (t^-2 + t^-1 + O(t^4))*D + t^-3 - 2*t^-2 + O(t^3),
    D^2 + (t^-2 + t^-1)*D + t^-3 - 2*t^-2
]
```

Example: Operator the Factorisation Routine Cannot Split (ex-c4b3ef)

This example shows that not all operators may be factored by the factorisation routine. Notice that the Newton polynomial of the operator is a square of a polynomial.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> R<D>:=DifferentialOperatorRing(S);
> L:=(D+2/t)^2;
> np:=NewtonPolygon(L);
> faces:=Faces(np);
> #faces eq 1;
true
> NewtonPolynomial(faces[1]);
$.1^2 + 4*$.1 + 4
> factsL_lclm,blsL_lclm:=Factorisation(L);
Ring precision as default precision taken.
Performing coprime index 1 LCLM factorisation.
> factsL_lclm;
[
    [
        1,
        D^2 + 4*t^-1*D + 4*t^-2 - 2*t^-1
    ]
]
> blsL_lclm;
[ false ]
> factsL_c1,blsL_c1:=Factorisation(L:Algorithm:="CoprimeIndexOne");
Ring precision as default precision taken.
Performing coprime index 1 factorisation.
> factsL_c1 eq factsL_lclm;
```

(continues on next page)

(continued from previous page)

```
true
> blsL_c1 eq blsL_lclm;
true
```

Example: Unexpected Factorisation of an Operator (ex-8a9dd9)

This example shows that one may not retrieve the factorisation as expected.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> R<D>:=DifferentialOperatorRing(S);
> L := (D+1/(t+1))*D;
> factsL_c1,blsL_c1:=Factorisation(L:Algorithm:="CoprimeIndexOne");
Ring precision as default precision taken.
Performing coprime index 1 factorisation.
> (#factsL_c1 eq 1) and (#blsL_c1 eq 1);
true
> factsL_c1[1][2];
D + O(t^20)
> factsL_lclm, blsL_lclm:=Factorisation(L:Algorithm:="LCLM");
Ring precision as default precision taken.
Performing coprime index 1 LCLM factorisation.
The number of slopes of the Newton polynomial: 1
> (#factsL_lclm eq 1) and (#blsL_lclm eq 1);
true
> factsL_c1[1][2], blsL_lclm[1];
D + O(t^20)
false
> M:=(D+1/(t-1))*D;
> factsM:=Factorisation(M:Algorithm:="CoprimeIndexOne");
Ring precision as default precision taken.
Performing coprime index 1 factorisation.
> # factsM eq 1;
> factsM[1][2]+O(t^4);
D + -1 - 1/2*t - 5/12*t^2 - 3/8*t^3 + O(t^4)
```

Example: Precision of Factors for a Rational Slope (ex-fee1e8)

One may wish to adjust the default precision to retrieve enough terms in the coefficients in the factors. This example shows the effect on a rational slope 1/5 on the number of terms in the series coefficients of the factors obtained.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals():Precision:=15);
> R<D>:=DifferentialOperatorRing(S);
> L:=(D^5+t^(-1))*D;
> np:=NewtonPolygon(L);
> Slopes(np);
[ 0 , 1/5 ]
> factsL,blsL:=Factorisation(L:Algorithm:="LCLM");
Ring precision as default precision taken.
Performing coprime index 1 LCLM factorisation.
> blsL;
[ true, true ]
> [v[2]:v in factsL];
[
    D,
    D^5 + (-1 - t - 63*t^2 + O(t^3))*D^4 + (1 + 3*t + 253*t^2 +
    O(t^3))*D^3 + (-1 - 7*t - 825*t^2 + O(t^3))*D^2 + (1 + 15*t + 2545*t^2
```

(continues on next page)

(continued from previous page)

```
    + O(t^3))*D + t^-1 - 1 - 31*t - 7713*t^2 + O(t^3)
]
> [v[1]*v[2]:v in factsL];
[
    D^6 + t^-1*D,
    D^6 + O(t^3)*D^5 + O(t^3)*D^4 + O(t^3)*D^3 + O(t^3)*D^2 + (t^-1 +
    O(t^3))*D + O(t^2)
]
> factsL:=Factorisation(L:Algorithm:="LCLM",Precision:=75);
Performing coprime index 1 LCLM factorisation.
> [v[1]*v[2]:v in factsL];
[
    D^6 + t^-1*D,
    D^6 + O(t^15)*D^5 + O(t^15)*D^4 + O(t^15)*D^3 + O(t^15)*D^2 + (t^-1 +
    O(t^15))*D + O(t^14)
]
```

121.21.3 Right Hand Factors of Operators

While resorting to field extensions can result in more complex and time consuming computations, they can be used for obtaining irreducible right hand factors over the original base field.

An effective algorithm to obtain a monic irreducible right hand factor of degree one $\tilde{L}$ of $L \in k((t))[\delta]$ in some field extension $\tilde{k}((\tilde{t}))[\tilde{\delta}]$ is presented in [vH97b, §5.1]. Such a factor $\tilde{L} = \tilde{\delta} - r(\tilde{t})$ of L is called a *Ricatti factor* of L. The operator $\mathrm{LCLM}(\tilde{\delta} - \sigma_1(r), \tilde{\delta} - \sigma_2(r), \ldots, \tilde{\delta} - \sigma_m(r))$ where the σ_i are the Galois group elements of $\tilde{k}((\tilde{t}))/k((t))$, then is a monic and irreducible operator invariant under the Galois action. Hence, it naturally reduces to a monic irreducible right hand factor of L.

Other right hand factors of L, that may be defined over a finite field extensions of $k((t))$ are the so-called *semi-regular* parts of L. Such an operator $R_e(L)$ is the monic right hand semi-regular factor of the translation $S_e(L)$ of L by e. Its degree is equal to the dimension of a non-trivial $\bar{k}$-linear vector space $V_e(L)$, and acts as zero on it. In other words $S_{-e}(R_e)$ is a monic right hand factor of L, possibly defined over a field extension of $k((t))$. It can be shown that L is the least common left multiple of all such right hand factors.

The routine `RightHandFactors` returns right hand factors of a given operator possibly by using temporary field extensions. Each of these are canonical representatives of all right hand factors belonging the slopes of the Newton polynomial of the operator. Currently one of the internal routines may fail when performing calculations for some specific right hand factor. In this case we cannot conclude it to be irreducible.

`RightHandFactors(L): RngDiffOpElt -> SeqEnum, [BoolElt]`

```
Precision: RngIntElt                    Default: -1
```

The canonical list of monic right hand factors of L, one per slope of the Newton polynomial of L. The i^{th} entry in the second sequence returned is true if the i^{th} right hand factor is undisputedly irreducible. The precision attribute relates to the absolute precision a coefficient in a right hand factor should minimally have.

Example: Right Hand Factors Matching an LCLM Factorisation `(ex-3c08eb)`

This example is Example 3.49 from [vdPS03]. The same right hand factors as in Example *Factorising a Product of First-Order Operators* are obtained.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> RS<DS> := DifferentialOperatorRing(S);
> L:=DS^2+(1/t^2+1/t)*DS +(1/t^3-2/t^2);
> rhf, bl := RightHandFactors(L);
```

(continues on next page)

(continued from previous page)

```
Performing coprime index 1 LCLM factorisation.
> #rhf eq 2;
true
> bl;
[ true, true ]
> (Parent(rhf[1]) eq RS) and (Parent(rhf[2]) eq RS);
true
> lhf,rem := EuclideanRightDivision(L, rhf[1]);
> rem;
O(t^20)
> lhf*rhf[1];
DS^2 + (t^-2 + t^-1 + O(t^22))*DS + t^-3 - 2*t^-2 + O(t^20)
> EuclideanRightDivision(L, rhf[2]);
DS + t^-1
0
```

Example: Operator That Is Its Own Right Hand Factor (ex-014248)

This example corresponds to Example 3.52 in [vdPS03]. The answer in the book is erroneous.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> RS<DS> := DifferentialOperatorRing(S);
> L:=DS^2-3/2*DS+(2*t-1)/(4*t);
> rhf, bl := RightHandFactors(L);
Performing coprime index 1 LCLM factorisation.
> bl;
[ true ]
> #rhf eq 1;
true
> rhf[1] eq L;
true
```

Example: Right Hand Factors Where Factorisation Fails (ex-69cc8c)

The operator in this example is the same as in Example *Operator the Factorisation Routine Cannot Split* where the routine `Factorisation` was used. The operator did not factor there, but does when using `RightHandFactors`.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> RS<DS> := DifferentialOperatorRing(S);
> L:=(DS+2/t)^2;
> rhf, bl := RightHandFactors(L);
Performing coprime index 1 LCLM factorisation.
Calculating semi-regular parts.
Performing coprime index 1 LCLM factorisation.
Performing coprime index 1 LCLM factorisation.
Computing a first order Ricatti factor.
Performing LCLM calculation on the Ricatti factor.
> rhf;
[
    DS + 2*t^-1
]
> bl;
[ true ]
```

Example: Right Hand Factors for a Square Newton Polynomial (ex-6d4ec1)

This example corresponds to Example 3.53 in [vdPS03].

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> RS<DS> := DifferentialOperatorRing(S);
> L:=DS^2 +(4+2*t-t^2-3*t^3)/(t^2)*DS+ (4+4*t-5*t^2-8*t^3-3*t^4+2*t^6)/(t^4);
> np:=NewtonPolygon(L);
> faces:=Faces(np);
> #faces eq 1;
true
> _<T> := PolynomialRing(Rationals());
> NewtonPolynomial(faces[1]);
T^2 + 4*T + 4
> factsL, blsL := Factorisation(L);
Ring precision as default precision taken.
Performing coprime index 1 LCLM factorisation.
> blsL;
[ false ]
> (#factsL eq 1) and (factsL[1][2] eq L);
true
> rhf, bl := RightHandFactors(L);
Performing coprime index 1 LCLM factorisation.
Calculating semi-regular parts.
Performing coprime index 1 LCLM factorisation.
Performing coprime index 1 LCLM factorisation.
Performing coprime index 1 LCLM factorisation.
Computing a first order Ricatti factor.
Performing LCLM calculation on the Ricatti factor.
> Degree(rhf[1]);
1
> bl;
[ true ]
> Parent (rhf[1]) eq RS;
true
> L - EuclideanRightDivision(L, rhf[1])*rhf[1];
O(t^22)*DS + O(t^20)
```

Example: Operators Sharing a Newton Polynomial (ex-987714)

A collection of operators having the same Newton polynomial $(T^2+1)(T-1)(T+1)$.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> RS<DS> := DifferentialOperatorRing(S);
> L := DS^4-1/t^4;
> faces:=Faces(NewtonPolygon(L));
> faces;
[ <-1, 1, -4> ]
> _<T> := PolynomialRing(Rationals());
> NewtonPolynomial(faces[1]);
T^4 - 1
> rhf, bl := RightHandFactors(L);
Performing coprime index 1 LCLM factorisation.
> bl;
[ true, true, true ]
> [Degree(v) : v in rhf];
[ 1, 1, 2 ]
>  L - EuclideanRightDivision(L, rhf[1])*rhf[1];
O(t^23)*DS^3 + O(t^22)*DS^2 + O(t^21)*DS + O(t^20)
>  L - EuclideanRightDivision(L, rhf[2])*rhf[2];
O(t^23)*DS^3 + O(t^22)*DS^2 + O(t^21)*DS + O(t^20)
```

(continues on next page)

(continued from previous page)

```
>  L - EuclideanRightDivision(L, rhf[3])*rhf[3];
O(t^23)*DS^3 + O(t^22)*DS^2 + O(t^21)*DS + O(t^20)
> M := DS^4-1;
> faces:=Faces(NewtonPolygon(M));
> faces;
[ <0, 1, 0> ]
> NewtonPolynomial(faces[1]);
T^4 - 1
> rhf, bl := RightHandFactors(M);
Performing coprime index 1 LCLM factorisation.
Calculating semi-regular parts.
Performing coprime index 1 LCLM factorisation.
Performing LCLM calculation on a semi-regular part.
Calculating semi-regular parts.
Performing coprime index 1 LCLM factorisation.
Computing a first order Ricatti factor.
Performing LCLM calculation on the Ricatti factor.
> rhf;
[
    DS^2 + 1,
    DS + -1
]
> bl;
[ true, true ]
```

Example: Factorising an Operator of Order 9 (ex-826452)

This is the main example of [vH97b].

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> RS<DS> := DifferentialOperatorRing(S);
> L:=DS^9 + 2*t^-1*DS^8 + 3*t^-2*DS^7 + 2*t^-3*DS^6 + (t^-4 + 2*t^-2)*DS^5 +
> (-3*t^-5 + 5*t^-4)*DS^3 + 3*t^-5*DS^2 + (2*t^-6 + 2*t^-5)*DS + 7*t^-5;
> facts := Factorisation(L);
Ring precision as default precision taken.
Performing coprime index 1 LCLM factorisation.
> [Degree(v[2]): v in facts];
[ 1 2 2 4 ]
> isweaklyzero := [];
> vals :=[];
> for i in [1..4] do
>   _,rem := EuclideanRightDivision(L, facts[i][2]);
>   isweaklyzero[i] := IsWeaklyZero(rem);
>   vals[i] := [Valuation(v) : v in Eltseq(rem)];
> end for;
> isweaklyzero;
[ true, true, true, true ]
> [Minimum(v) : v in vals];
[ 14 , 4 , 4 , 11]
> rhf, bl := RightHandFactors(L:Precision:=30);
Performing coprime index 1 LCLM factorisation.
Calculating semi-regular parts.
Performing coprime index 1 LCLM factorisation.
Performing coprime index 1 LCLM factorisation.
Performing coprime index 1 LCLM factorisation.
Performing LCLM calculation on a semi-regular part.
```

(continues on next page)

(continued from previous page)

```
Computation of the LCLM failed.
> bl;
[ true, true, true, false ]
> [Degree(v): v in rhf];
[ 1 2 2 4 ]
> isweaklyzero := [];
> vals := [];
> for i in [1..4] do
>   [Degree(v): v in Eltseq(rhf[i])];
>   _,rem := EuclideanRightDivision(L, rhf[i]);
>   isweaklyzero[i] := IsWeaklyZero(rem);
>   vals[i] := [Valuation(v) : v in Eltseq(rem)];
> end for;
[ 35, 0 ]
[ 35, 35, 0 ]
[ 35, 35, 0 ]
[ 31, 32, 33, 34, 0 ]
> isweaklyzero;
[ true, true, true, true ]
> [ Minimum(v) : v in vals];
[ 30, 30, 30, 27 ]
```

121.22 Bibliography

- [BMW97] Manuel Bronstein, Thom Mulders, and Jacques-Arthur Weil. On symmetric powers of differential operators. In *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation (Kihei, HI)*, 156–163 (electronic). New York, 1997. ACM.
- [vdPS03] Marius van der Put and Michael F. Singer. *Galois theory of linear differential equations*. Volume 328 of Grundlehren der Mathematischen Wissenschaften [Fundamental Principles of Mathematical Sciences]. Springer-Verlag, Berlin, 2003. ISBN 3-540-44228-6.
- [vH97a] Mark van Hoeij. Factorization of differential operators with rational functions coefficients. *J. Symbolic Comput.*, 24(5):537–561, 1997.
- [vH97b] Mark van Hoeij. Formal solutions and factorization of differential operators with power series coefficients. *J. Symbolic Comput.*, 24(1):1–30, 1997.

References

[AL94] William Adams and Philippe Loustaunau. *An introduction to Gröbner bases*. Volume 3 of Graduate studies in mathematics. American Mathematical Society, Providence, R.I., 1994.

[AM94] A. Adem and R.J. Milgram. *Cohomology of Finite Groups*. Grundlehren der Mathematischen Wissenschaften. Springer, Berlin-New York-Heidelberg, 1994.

[BS92] David Bayer and Michael Stillman. Computation of Hilbert Functions. *J. Symbolic Comp.*, 14(1):31–50, 1992.

[BW93] Thomas Becker and Volker Weispfenning. *Gröbner Bases*. Graduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 1993.

[BMW97] Manuel Bronstein, Thom Mulders, and Jacques-Arthur Weil. On symmetric powers of differential operators. In *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation (Kihei, HI)*, 156–163 (electronic). New York, 1997. ACM.

[Buc65] Bruno Buchburger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*. PhD thesis, University of Innsbruck, Austria, 1965.

[CKM97] Stephane Collart, Michael Kalkbrener, and Daniel Mall. Converting Bases with the Gröbner Walk. *J. Symbolic Comp.*, 24(3):465–469, 1997.

[CLO96] David Cox, John Little, and Donal O'Shea. *Ideals, Varieties and Algorithms*. Undergraduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 2nd edition, 1996.

[CLO98] David Cox, John Little, and Donal O'Shea. *Using Algebraic Geometry*. Graduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 1998.

[DE02] Wolfram Decker and David Eisenbud. Sheaf algorithms using the Exterior algebra. In Eisenbud and others, editors, *Computations in Algebraic Geometry with Macaulay2*, volume 8 of Springer Algorithms and Computation in Mathematics Series, 215–247. Springer-Verlag, 2002.

[DL06] Wolfram Decker and Christoph Lossen. *Computing in Algebraic Geometry*. Volume 16 of Algorithms and Computation in Mathematics. Springer, New York–Berlin–Heidelberg, 2006.

[Der99] Harm Derksen. Computation of Invariants for Reductive Groups. *Adv. Math.*, 141:366–384, 1999.

[EFS03] Eisenbud, Floystad, and Schreyer. Sheaf Cohomology and Free Resolutions over Exterior Algebras. *Trans. Am. Maths. Soc.*, 355:4397–4426, 2003.

[ES94] Eisenbud and Sturmfels. Finding sparse systems of parameters. *Journal of Pure and Applied Algebra*, 94:143–157, 1994.

[EHV92] D. Eisenbud, C. Huneke, and W. Vasconcelas. Direct Methods for Primary Decomposition. *Inv. math.*, 110:207–235, 1992.

[Eis95] David Eisenbud. *Commutative Algebra with a View Toward Algebraic Geometry*. Volume 150 of Graduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 1995.

[Fau99] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases (F_4). *Journal of Pure and Applied Algebra*, 139 (1-3):61–88, 1999.

[FGLM93] Jean-Charles Faugère, Patrizia Gianni, Daniel Lazard, and Teo Mora. Efficient computations of zero-dimensional Gröbner bases by change of ordering. *J. Symbolic Comp.*, 16:329–344, 1993.

[GP02] G.-M. Greuel and G. Pfister. *A Singular Introduction to Commutative Algebra*. Springer-Verlag, Berlin–Heidelberg–New York, 2002. ISBN 3-540-42897-6.

[Har77] Robin Hartshorne. *Algebraic Geometry, GTM 52*. Springer, ASpringer, 1977.

[Kem96] Gregor Kemper. Calculating Invariant Rings of Finite Groups over Arbitrary Fields. *J. Symbolic Comp.*, 21(3):351–366, 1996.

[Kem99] Gregor Kemper. An Algorithm to Calculate Optimal Homogeneous Systems of Parameters. *J. Symbolic Comp.*, 27(2):171–184, 1999.

[KS97] Gregor Kemper and Allan Steel. Some Algorithms in Invariant Theory of Finite Groups. In P. Dräxler, G.O. Michler, and C.M. Ringel, editors, *Computational Methods for Representations of Groups and Algebras, Euroconference in Essen, April 1-5 1997*, number 173 in Progress in Mathematics. Basel, 1997. Birkhäuser.

[Kin07] Simon King. Minimal generating sets of non-modular invariant rings of finite groups. http://arxiv.org/abs/math/0703035, 2007.

[Laz92] Daniel Lazard. Solving Zero-dimensional Algebraic Systems. *J. Symbolic Comp.*, 13(2):117–131, 1992.

[MQS99] Jörg Müller-Quade and Rainer Steinwandt. Basic algorithms for rational function fields. *J. Symbolic Comp.*, 27(2):143–170, 1999. doi:http://dx.doi.org/10.1006/jsco.1998.0246.

[Mol88] H.M. Möller. On the construction of Gröbner bases using syzygies. *J. Symbolic Comp.*, 6:345–359, 1988.

[MQB99] Jörg Müller-Quade and Thomas Beth. Calculating Generators for Invariant Fields of Linear Algebraic Groups. In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes (Honolulu, HI, 1999)*, number 1719 in LNCS, 392–403. Berlin, 1999. Springer.

[SS98] Roberto La Scala and Michael Stillman. Strategies for Computing Minimal Free Resolutions. *J. Symbolic Comp.*, 26(4):409–431, 1998.

[ST02] Frank-Olaf Schreyer and Fabio Tonoli. Needles in a Haystack: Special Varieties via Small Fields. In Eisenbud and others, editors, *Computations in Algebraic Geometry with Macaulay2*, volume 8 of Springer Algorithms and Computation in Mathematics Series, 251–277. Springer-Verlag, 2002.

[Ste04] Allan Steel. Gröbner Basis Timings Page. http://magma.maths.usyd.edu.au/users/allan/gb/, 2004.

[Ste05] Allan Steel. Conquering Inseparability: Primary Decomposition and Multivariate Factorization over Algebraic Function Fields of Positive Characteristic. *J. Symbolic Comp.*, 40(3):1053–1075, 2005.

[Tra96] Carlo Traverso. Hilbert Functions and the Buchberger Algorithm. *J. Symbolic Comp.*, 22(4):355–376, 1996.

[vdPS03] Marius van der Put and Michael F. Singer. *Galois theory of linear differential equations*. Volume 328 of Grundlehren der Mathematischen Wissenschaften [Fundamental Principles of Mathematical Sciences]. Springer-Verlag, Berlin, 2003. ISBN 3-540-44228-6.

[vH97a] Mark van Hoeij. Factorization of differential operators with rational functions coefficients. *J. Symbolic Comput.*, 24(5):537–561, 1997.

[vH97b] Mark van Hoeij. Formal solutions and factorization of differential operators with power series coefficients. *J. Symbolic Comput.*, 24(1):1–30, 1997.

Index

Symbols

`&meet S` (*operation*), 37, 76
`- f` (*operation*), 97
`- s` (*operation*), 171, 185

A

`Adjoint(L)` (*function*), 188
`AffineAlgebra< R, X | L >` (*constructor*), 80
`AffineAlgebraMapKernel(phi)` (*function*), 84
`Algebra(R)` (*function*), 145
`Algebraic Extension of a Differential Field` (*example*), 178
`Algebraic Relations of an Invariant Ring` (*example*), 145
`Ambient(M)` (*function*), 95
`AmbientMatrix(f)` (*function*), 100
`Annihilator(M)` (*function*), 107
`Apply(L, f)` (*function*), 189
`Applying an Operator to a Function` (*example*), 189
`Arithmetic of Differential Operators` (*example*), 186
`AssignNames(~R, S)` (*procedure*), 166, 182
`Attributes of Invariant Rings` (*example*), 152

B

`BaseField(R)` (*function*), 167
`BaseRing(M)` (*function*), 95
`BaseRing(R)` (*function*), 167, 183
`Basis(I)` (*function*), 9, 73
`Basis(M)` (*function*), 103
`BasisElement(I, i)` (*function*), 9, 73
`BasisElement(M, i)` (*function*), 103
`BasisMatrix(M)` (*function*), 103
`Betti Numbers of a Localized Module` (*example*), 117
`BettiNumber(M, i, j)` (*function*), 113
`BettiNumbers(M)` (*function*), 113
`BettiTable(M)` (*function*), 113
`BinaryForms(N, p)` (*function*), 154
`BinaryForms(n, p)` (*function*), 154
`BooleanPolynomialRing(B, Q)` (*function*), 17
`BooleanPolynomialRing(n)` (*function*), 17
`BooleanPolynomialRing(n, order)` (*function*), 17
`Bounding the Regularity of a Module` (*example*), 119

C

`Category(L)` (*function*), 185
`Category(R)` (*function*), 167, 183
`Category(s)` (*function*), 171
`ChangeDerivation(R, f)` (*function*), 174, 190
`ChangeDifferential(F, df)` (*function*), 174
`ChangeDifferential(R, df)` (*function*), 190
`ChangeOrder(I, order)` (*function*), 30, 78
`ChangeOrder(I, Q)` (*function*), 30, 78
`ChangeOrder(I, T)` (*function*), 30
`ChangePrecision(F, p)` (*function*), 170
`ChangeRing(I, L)` (*function*), 78
`ChangeRing(I, S)` (*function*), 29
`ChangeRing(M, S)` (*function*), 108
`Changing Attributes of an Operator Ring over Series` (*example*), 191
`Changing the Derivation of a Differential Ring` (*example*), 174
`Changing the Differential of a Differential Ring` (*example*), 175
`Changing the Differential of an Operator Ring` (*example*), 190
`Changing the Precision of a Series Ring` (*example*), 170
`Codomain(f)` (*function*), 100
`Coefficient(L, i)` (*function*), 187
`CoefficientField(R)` (*function*), 131
`CoefficientRing(M)` (*function*), 95
`CoefficientRing(Q)` (*function*), 82
`CoefficientRing(R)` (*function*), 131, 183
`Coefficients and Terms of an Operator` (*example*), 187
`Coefficients of Algebraic Differential Elements` (*example*), 173
`Coefficients of Differential Series` (*example*), 173
`Coefficients(f)` (*function*), 98
`Coefficients(L)` (*function*), 186
`CoefficientsAndMonomials(f)` (*function*), 98
`Coercion into an Operator Ring` (*example*), 183
`Cohomology of an Enriques Surface` (*example*), 125
`CohomologyDimension(M,r,n)` (*function*), 125
`Cokernel(f)` (*function*), 101
`ColonIdeal(I, f)` (*function*), 36
`ColonIdeal(I, J)` (*function*), 36
`ColonIdeal(M, N)` (*function*), 107
`ColonIdealEquivalent(I, f)` (*function*), 36
`ColonModule(M, J)` (*function*), 107

Column(f) (*function*), 98
ColumnWeights(M) (*function*), 96
Companion Matrix of an Operator (*example*), 196
CompanionMatrix(L) (*function*), 196
Comparing LCLM and Coprime Index 1 Factorisations (*example*), 204
Completion of an Elliptic Function Field (*example*), 176
Completion of an Operator Ring (*example*), 192
Completion(F, p) (*function*), 176
Completion(R, p) (*function*), 191
Constant Field Extension of a Differential Field (*example*), 175
Constant Field Extension of a Laurent Series Ring (*example*), 175
ConstantField(R) (*function*), 167
ConstantFieldExtension(F, C) (*function*), 175
ConstantFieldExtension(R, C) (*function*), 190
ConstantRing(R) (*function*), 167, 183
Constructing a G-Module (*example*), 136
CoordinateMatrix(I) (*function*), 16, 74
Coordinates with Respect to an Ideal Basis (*example*), 25
Coordinates(I, f) (*function*), 16, 73
Coordinates(M, f) (*function*), 97
Creating a Differential Laurent Series Ring (*example*), 165
Creating a Differential Operator Ring (*example*), 182
Creating a General Differential Ring (*example*), 165
Creating a Rational Differential Field (*example*), 165
Creating Differential Field Elements (*example*), 166

D

Degree(f) (*function*), 6, 98, 101
Degree(L) (*function*), 187
Degree(M) (*function*), 95
Degree-10 Representation of S5 (*example*), 142
Degree-d Groebner Bases (*example*), 28
Denominator(a) (*function*), 88
Depth of an Invariant Ring (*example*), 148
Depth(R) (*function*), 148
Derivation and Differential of a Field (*example*), 168
Derivation(R) (*function*), 168, 183
Derivative(s) (*function*), 174
Derivatives and Differentials of Elements (*example*), 174
DerksenIdeal(F) (*function*), 160
DerksenIdeal(R) (*function*), 154
Differential Field Extension by an Operator (*example*), 177
Differential Ideal and Quotient Ring (*example*), 181
Differential Operator of an Algebraic Function (*example*), 201
Differential Ring Extension by an Operator (*example*), 177
Differential(F) (*function*), 168
Differential(R) (*function*), 183
Differential(s) (*function*), 174
DifferentialFieldExtension(L) (*function*), 177
DifferentialIdeal(L) (*function*), 180
DifferentialLaurentSeriesRing(C) (*function*), 165
DifferentialOperator(f) (*function*), 201
DifferentialOperatorRing(F) (*function*), 182
DifferentialRing(P, f, C) (*function*), 165
DifferentialRingExtension(L) (*function*), 177
Dimension(I) (*function*), 50, 78
Dimension(Q) (*function*), 84
DirectSum(M, N) (*function*), 105
DirectSum(S) (*function*), 105
DivisorIdeal(I) (*function*), 82
Domain(f) (*function*), 100

E

EasyBasis(I) (*function*), 15
EasyIdeal(I) (*function*), 15
Element Operations with Ideals (*example*), 40
Element Operations with Local Ideals (*example*), 78
ElementarySymmetricPolynomial(P, k) (*function*), 65, 162
Elimination Ideals from Neural Network Theory (*example*), 45
Elimination over a Quadratic Order (*example*), 44
EliminationIdeal(I, k: parameters) (*function*), 43
EliminationIdeal(I, S) (*function*), 44
Elliptic Curve Function Field (*example*), 85
Eltseq(f) (*function*), 97
Eltseq(L) (*function*), 186
Eltseq(s) (*function*), 172
Embedded Modules over a Polynomial Ring (*example*), 98
EModule(R, k) (*function*), 94
EModule(R, k, order) (*function*), 94
EModule(R, W) (*function*), 94
EModule(R, W, order) (*function*), 94
Equidimensional Decomposition (*example*), 58
EquidimensionalDecomposition(I) (*function*), 58
EquidimensionalPart(I) (*function*), 58
Euclidean Right and Left Division (*example*), 193
EuclideanLeftDivision(D, N) (*function*), 193
EuclideanRightDivision(N, D) (*function*), 193
ExactConstantField(F) (*function*), 167

Exponential and Logarithmic Extensions (*example*), 178
ExponentialFieldExtension(F, f) (*function*), 178
Exponents(s) (*function*), 173
Expressing One in Terms of Generators (*example*), 47
Ext(i, M, N) (*function*), 121
ext< F | f > (*constructor*), 177
ExtendedGreatestCommonLeftDivisor(A, B) (*function*), 194
ExtendedGreatestCommonRightDivisor(A, B) (*function*), 193
ExtendedLeastCommonLeftMultiple(A, B) (*function*), 194
ExtendedLeastCommonLeftMultiple(S) (*function*), 194
Extension and Contraction of Ideals (*example*), 89
Extension(I, U) (*function*), 49
Extension(phi, I) (*function*), 65

F

f * M (*operation*), 105
f * r (*operation*), 97
f + g (*operation*), 97
f @ L (*operation*), 189
f - g (*operation*), 97
f ^ a (*operation*), 132
f ^ G (*operation*), 131, 132
f ^ g (*operation*), 131
f div s (*operation*), 97
f eq g (*operation*), 98
f in I (*operation*), 40, 77
f in M (*operation*), 98
f in R (*operation*), 131
f lt g (*operation*), 98
f notin I (*operation*), 40, 77
f(v) (*literal*), 100
Factorisation(L) (*function*), 203
Factorising a Product of First-Order Operators (*example*), 204
Factorising an Operator of Order 9 (*example*), 210
Factorization over an Affine Algebra (*example*), 86
Factorization(L) (*function*), 203
Field of Fractions (*example*), 88
FieldOfFractions(Q) (*function*), 88
FieldOfFractions(R) (*function*), 166
Finding Primes via Groebner Bases (*example*), 23
FineEquidimensionalDecomposition(I) (*function*), 58
FittingIdeal(M, i) (*function*), 107
FittingIdeals(M) (*function*), 107
Free Resolution of Six Points (*example*), 112
FreeResolution(M) (*function*), 110
FreeResolution(R) (*function*), 147
FunctionField(F) (*function*), 159
Fundamental Invariants of a Permutation Group (*example*), 140
FundamentalInvariants(F) (*function*), 159
FundamentalInvariants(R) (*function*), 139, 154
f[i] (*indexed*), 97, 100

G

GCLD(A, B) (*function*), 193
GCRD(A, B) (*function*), 193
Generators(R) (*function*), 167
Generic(I) (*function*), 37, 76
Generic(M) (*function*), 95
GetSparseMonomialMinRank() (*function*), 4
GModule(G, I, J) (*function*), 136
GModule(G, P, d) (*function*), 136
GModule(G, Q) (*function*), 136
GradedModule(I) (*function*), 103
GradedModule(R, k) (*function*), 95
GradedModule(R, W) (*function*), 95
Grading(M) (*function*), 96
Grading(P) (*function*), 6
Greatest Common Right and Left Divisors (*example*), 194
GreatestCommonLeftDivisor(A, B) (*function*), 193
GreatestCommonRightDivisor(A, B) (*function*), 193
Groebner Bases over a Quadratic Order (*example*), 24
Groebner Bases over a Valuation Ring (*example*), 26
Groebner Bases over the Integers (*example*), 22
Groebner Basis of the Cyclic-6 Ideal (*example*), 18
Groebner Basis over a Smaller Field (*example*), 29
Groebner(I: parameters) (*procedure*), 10
Groebner(M) (*procedure*), 103
GroebnerBasis(I: parameters) (*function*), 13
GroebnerBasis(S, d : parameters) (*function*), 28
GroebnerBasis(S, d: parameters) (*function*), 13
GroebnerBasis(S: parameters) (*function*), 13
GroebnerBasisUnreduced(S: parameters) (*function*), 13
Group(F) (*function*), 159
Group(R) (*function*), 131
GroupIdeal(F) (*function*), 159
GroupIdeal(R) (*function*), 154

H

HasFiniteDimension(Q) (*function*), 84
HasGrevlexOrder(I) (*function*), 39
HasGroebnerBasis(I) (*function*), 15

HasProjectiveDerivation(F) (*function*), 169
HasProjectiveDerivation(R) (*function*), 184
HasRationalSolutions(L, g) (*function*), 198
HasZeroDerivation(F) (*function*), 169
HasZeroDerivation(R) (*function*), 184
Hilbert Series from Betti Numbers (*example*), 117
Hilbert Series of a Matrix Square Ideal (*example*), 63
Hilbert Series of a Quotient Module (*example*), 109
Hilbert-Driven Groebner Basis for Invariants (*example*), 31
HilbertDenominator(I) (*function*), 62
HilbertDenominator(M) (*function*), 108
HilbertGroebnerBasis(S, H) (*function*), 31
HilbertGroebnerBasis(S, N) (*function*), 31
HilbertIdeal(R) (*function*), 154
HilbertNumerator(I) (*function*), 62
HilbertNumerator(M) (*function*), 108
HilbertPolynomial(I) (*function*), 63, 108
HilbertSeries(I) (*function*), 62
HilbertSeries(I, p) (*function*), 62
HilbertSeries(M) (*function*), 108
HilbertSeries(M, p) (*function*), 108
HilbertSeries(R) (*function*), 147
HilbertSeriesApproximation(R, n) (*function*), 147
Hom Module and Explicit Homomorphisms (*example*), 121
Hom(C, N) (*function*), 121
Hom(M, N) (*function*), 121
Homogeneous Module Test (*example*), 68
Homogeneous Module Test for Invariant Rings (*example*), 150
HomogeneousComponent(f, d) (*function*), 6
HomogeneousComponents(f) (*function*), 7
HomogeneousModuleTest(P, S, F) (*function*), 66, 149
HomogeneousModuleTest(P, S, L) (*function*), 67, 149
HomogeneousModuleTestBasis(P, S, L) (*function*), 67
HomogeneousWeightsSearch(S) (*function*), 6
Homogenization(I) (*function*), 48
Homogenization(I, b) (*function*), 48
Homogenization(I, b, order) (*function*), 48
Homogenization(I, order) (*function*), 48
HomologicalDimension(M) (*function*), 114
HomologicalDimension(R) (*function*), 148
Homomorphism(M, N, A) (*function*), 100

I

I * J (*operation*), 36, 76, 82
I * M (*operation*), 105
I + J (*operation*), 36, 76, 82
I / J (*operation*), 36
I ^ k (*operation*), 36, 76
I ^ n (*operation*), 82
I eq J (*operation*), 38, 76, 82
I meet J (*operation*), 37, 76, 82
I ne J (*operation*), 38, 76
I notsubset J (*operation*), 38, 76
I subset J (*operation*), 38, 76, 82
Ideal Arithmetic (*example*), 39
Ideal Arithmetic in a Local Ring (*example*), 77
Ideal(B) (*function*), 8, 73
Ideal(f) (*function*), 8, 73
ideal<P | L> (*constructor*), 8
ideal<R | L> (*constructor*), 72
IdealQuotient(I, f) (*function*), 36
IdealQuotient(I, J) (*function*), 36
Ideals of Affine Algebras (*example*), 83
IdealWithFixedBasis(B) (*function*), 8, 73
Identity(R) (*function*), 166
Image(f) (*function*), 101
Implicitization of a Monomial Map (*example*), 65
Implicitization(phi) (*function*), 65
Inclusion Homomorphism of Submodules (*example*), 101
Indicial Polynomial of an Operator (*example*), 198
IndicialPolynomial(L, p) (*function*), 198
Invariant Field of a Cyclic Group (*example*), 160
Invariant Field of a Non-Reductive Group (*example*), 161
Invariant Ring of SL2 over the Rationals (*example*), 155
InvariantField(G) (*function*), 159
InvariantField(G, K) (*function*), 159
InvariantField(I, A) (*function*), 159
InvariantRing(G) (*function*), 131
InvariantRing(G, K) (*function*), 131
InvariantRing(I, A) (*function*), 153
Invariants of a Non-Reductive Algebraic Group (*example*), 158
Invariants of a Triple Tensor Product of SL2 (*example*), 156
Invariants of Binary Forms (*example*), 157
InvariantsOfDegree(G, d) (*function*), 133
InvariantsOfDegree(G, d, k) (*function*), 134
InvariantsOfDegree(G, K, d) (*function*), 133
InvariantsOfDegree(G, K, d, k) (*function*), 134
InvariantsOfDegree(G, P, d) (*function*), 133
InvariantsOfDegree(G, P, d, k) (*function*), 134
InvariantsOfDegree(R, d) (*function*), 133, 154
InvariantsOfDegree(R, d, k) (*function*), 134
IrreducibleSecondaryInvariants(R) (*function*), 139
IsAlgebraicDifferentialField(R) (*function*), 169

IsAmbient(M) (*function*), 95
IsBijective(f) (*function*), 101
IsCohenMacaulay(R) (*function*), 147
IsDifferentialField(R) (*function*), 169
IsDifferentialIdeal(R, I) (*function*), 181
IsDifferentialLaurentSeriesRing(R) (*function*), 169
IsDifferentialOperatorRing(R) (*function*), 184
IsDifferentialSeriesRing(R) (*function*), 169
IsDomain(R) (*function*), 169
IsEmbedded(M) (*function*), 95
IsField(R) (*function*), 169
IsFree(M) (*function*), 106
IsFuchsianOperator(L) (*function*), 197
IsGraded(f) (*function*), 101
IsGraded(M) (*function*), 96
IsGroebner(S) (*function*), 16
IsHomogeneous(f) (*function*), 6, 98, 101
IsHomogeneous(I) (*function*), 38
IsHomogeneous(M) (*function*), 96
IsIdentical(R, F) (*function*), 169, 184
IsInImage(f, p) (*function*), 64
IsInjective(f) (*function*), 101
IsInRadical(f, I) (*function*), 40
IsInvariant(f, G) (*function*), 132
IsInvariant(f, g) (*function*), 132
IsIrregularSingularPlace(L, p) (*function*), 197
IsMaximal(I) (*function*), 38
IsMonic(L) (*function*), 186
IsNilpotent(f) (*function*), 84
IsOne(L) (*function*), 186
IsOne(s) (*function*), 172
IsOrderTerm(s) (*function*), 172
IsPrimary(I) (*function*), 38, 82
IsPrime(I) (*function*), 38, 82
IsPrincipal(I) (*function*), 38
IsProper(I) (*function*), 38, 76, 82
IsRadical(I) (*function*), 39, 82
IsReduced(M) (*function*), 95
IsRegularPlace(L, p) (*function*), 196
IsRegularSingularOperator(L) (*function*), 197
IsRegularSingularPlace(L, p) (*function*), 197
IsRoot(M) (*function*), 95
IsSurjective(f) (*function*), 64, 101
IsSymmetric(f) (*function*), 65, 162
IsSymmetric(f, S) (*function*), 65, 162
IsUnit(f) (*function*), 84
IsWeaklyEqual(L, P) (*function*), 186
IsWeaklyEqual(s, t) (*function*), 172
IsWeaklyMonic(L) (*function*), 186
IsWeaklyZero(L) (*function*), 186
IsWeaklyZero(s) (*function*), 172
IsZero(f) (*function*), 98, 101
IsZero(I) (*function*), 38, 76, 82
IsZero(L) (*function*), 186
IsZero(M) (*function*), 106
IsZero(s) (*function*), 172
IsZeroDimensional(I) (*function*), 39, 76

J

JacobianIdeal(f) (*function*), 40

K

Kernel(f) (*function*), 101
Koszul Complex (*example*), 114

L

L(f) (*function*), 189
Laurent Series Ring by Completion (*example*), 176
LCLM(A, B) (*function*), 194
LeadingCoefficient(f) (*function*), 98
LeadingCoefficient(L) (*function*), 187
LeadingMonomial(f) (*function*), 98
LeadingMonomialIdeal(I) (*function*), 37, 76
LeadingTerm(f) (*function*), 98
LeadingTerm(L) (*function*), 187
LeadingWeightedDegree(f) (*function*), 6
Least Common Left Multiple of Conjugates (*example*), 195
Least Common Left Multiples (*example*), 194
LeastCommonLeftMultiple(A, B) (*function*), 194
LeastCommonLeftMultiple(L) (*function*), 194
LiftMap(m, R) (*function*), 189
Local Polynomial Rings with Different Orders (*example*), 72
Localization of an Operator Ring (*example*), 192
Localization(I) (*function*), 72
Localization(L) (*function*), 191
Localization(L, p) (*function*), 191
Localization(M) (*function*), 95
Localization(R) (*function*), 72, 191
Localization(R, p) (*function*), 191
LocalPolynomialAlgebra(K, n, order) (*function*), 71
LocalPolynomialRing(K, n) (*function*), 71
LocalPolynomialRing(K, n, order) (*function*), 71
LocalPolynomialRing(K, n, T) (*function*), 71
LogarithmicFieldExtension(F, f) (*function*), 178

M

M ! 0 (*operation*), 96
M ! Q (*operation*), 96
M ! v (*operation*), 96
M * f (*operation*), 105
M * I (*operation*), 105
M + N (*operation*), 105
M / N (*operation*), 105
M eq N (*operation*), 106
M meet N (*operation*), 105
M subset N (*operation*), 106
Maps between Operator Rings (*example*), 189

MarkGroebner(I) (*procedure*), 15
Matrix Group Action on Polynomials (*example*), 132
Matrix(f) (*function*), 100
MatrixAlgebra(Q) (*function*), 84
MaximumBettiDegree(M, i) (*function*), 113
Milnor and Tjurina Numbers (*example*), 43
MilnorNumber(f) (*function*), 42
Minimal Algebra Generators (*example*), 149
Minimal Free Resolution of a Module (*example*), 110
Minimal Polynomial of an Algebraic Element (*example*), 173
Minimal Polynomial of an Algebraic Number (*example*), 85
MinimalAlgebraGenerators(L) (*function*), 66, 149
MinimalBasis(M) (*function*), 106
MinimalBasis(S) (*function*), 106
MinimalDecomposition(S) (*function*), 52
MinimalFreeResolution(R) (*function*), 148
MinimalPolynomial(f) (*function*), 85
MinimalPolynomial(s) (*function*), 173
MinimalSyzygyModule(M) (*function*), 107
MinimizeGenerators(L) (*function*), 160
MinRankSystem(K, n, k, r) (*function*), 17
Module of an Invariant Ring (*example*), 144
Module(R) (*function*), 144
Molien Series of a Matrix Group (*example*), 137
MolienSeries(G) (*function*), 137
MolienSeriesApproximation(G, n) (*function*), 137
MonicDifferentialOperator(L) (*function*), 188
MonomialBasis(Q) (*function*), 84
MonomialOrder(P) (*function*), 4
MonomialOrder(R) (*function*), 71
MonomialOrderWeightVectors(P) (*function*), 4
MonomialOrderWeightVectors(R) (*function*), 71
Monomials(f) (*function*), 98
MonomialsOfDegree(P, d) (*function*), 7
MonomialsOfWeightedDegree(P, d) (*function*), 7
Morphism(M, N) (*function*), 102
Multiple Field Extension (*example*), 86

N

Name(R, i) (*function*), 166, 182
Newton Polygon at Infinity (*example*), 200
Newton Polygon of an Operator (*example*), 199
Newton Polygon over a Laurent Series Ring (*example*), 200
NewtonPolygon(L) (*function*), 199
NewtonPolygon(L, p) (*function*), 199
NewtonPolynomial(F) (*function*), 199
NewtonPolynomials(L) (*function*), 199
Ngens(R) (*function*), 169
Noether Normalisation (*example*), 59
NoetherNormalisation(I) (*function*), 59
NoetherNormalization(I) (*function*), 59
NormalForm(f, I) (*function*), 16, 77
NormalForm(f, S) (*function*), 16, 97
Normalisation of Plane Curves (*example*), 60
Normalisation(I) (*function*), 59
Normalization(I) (*function*), 59
Normalize(f) (*function*), 97
Number Field Extensions as Affine Algebras (*example*), 81
Number Field of Degree 8 (*example*), 87
NumberField(A) (*function*), 87
Numerator(a) (*function*), 88

O

O(s) (*function*), 172
One(R) (*function*), 166, 183
Operations in a Graded Polynomial Ring (*example*), 7
Operator That Is Its Own Right Hand Factor (*example*), 208
Operator the Factorisation Routine Cannot Split (*example*), 205
Operator with Infinitely Many Factorisations (*example*), 203
Operators Sharing a Newton Polynomial (*example*), 209
Order and Degree of Operators (*example*), 187
Order(L) (*function*), 187
OriginalRing(Q) (*function*), 82

P

P / J (*operation*), 80
Parent(L) (*function*), 185
Parent(R) (*function*), 167, 183
Parent(s) (*function*), 171
PolyMapKernel(f) (*function*), 64
Polynomial Rings with Different Orders (*example*), 4
PolynomialAlgebra(R, n) (*function*), 4
PolynomialAlgebra(R, n, order) (*function*), 4
PolynomialAlgebra(R, n, T) (*function*), 4
PolynomialAlgebra(R, Q) (*function*), 6
PolynomialRing(R) (*function*), 131
PolynomialRing(R, n) (*function*), 4
PolynomialRing(R, n, order) (*function*), 4
PolynomialRing(R, n, T) (*function*), 4
PolynomialRing(R, Q) (*function*), 6
Precision of Factors for a Rational Slope (*example*), 206
Predicates for Differential Ring Elements (*example*), 172
Predicates for Differential Rings (*example*), 169
Predicates for Operator Rings (*example*), 184
Predicates for Operator Rings over Series (*example*), 184
Predicates on the Derivation (*example*), 169
PreimageIdeal(I) (*function*), 82

PreimageRing(Q) (*function*), 82
Presentation(M) (*function*), 96
PresentationMatrix(f) (*function*), 100
Primary Decomposition (*example*), 52
Primary Invariants of an A5 Representation (*example*), 138
PrimaryAlgebra(R) (*function*), 145
PrimaryDecomposition(I) (*function*), 51, 82
PrimaryIdeal(R) (*function*), 145
PrimaryInvariants(R) (*function*), 138
ProbableRadicalDecomposition(I) (*function*), 51
Purely Ramified Extension of a Laurent Series Ring (*example*), 179
Purely Ramified Extension of an Algebraic Differential Field (*example*), 178
Purely Ramified Extension of an Operator Ring (*example*), 191
Purely Ramified Extension with an Inexact Derivation (*example*), 179
PurelyRamifiedExtension(f) (*function*), 178
PurelyRamifiedExtension(R,f) (*function*), 191

Q

Q . i (*operation*), 81
QuadeIdeal(L) (*function*), 160
quo< P | a_1, ..., a_r > (*constructor*), 80
quo< P | J > (*constructor*), 80
quo<M | L> (*constructor*), 102
QuotientDimension(I) (*function*), 36, 76
QuotientModule(I) (*function*), 103
QuotientRing(R, I) (*function*), 180

R

R ! s (*operation*), 166, 182
r * f (*operation*), 97
R . i (*operation*), 166, 182
R eq F (*operation*), 169, 184
R`HilbertSeries (*literal*), 152
R`PrimaryInvariants (*literal*), 152
R`SecondaryInvariants (*literal*), 152
Radical of a Zero-Dimensional Ideal (*example*), 46
Radical of an Ideal (*example*), 50
Radical(I) (*function*), 50
RadicalDecomposition(I) (*function*), 51, 83
Random Space Curve of Genus 11 (*example*), 119
Rank of a Module and Its Localization (*example*), 107
Rank(M) (*function*), 106
Rank(Q) (*function*), 82
Rational Solutions of an Operator (*example*), 198
RationalDifferentialField(C) (*function*), 165
RationalSolutions(L) (*function*), 198
Reduce(S) (*function*), 16
ReduceGroebnerBasis(S) (*function*), 16
ReesIdeal(P, I) (*function*), 37
ReesIdeal(P, J, I) (*function*), 37
ReesIdeal(R, I) (*function*), 37
Regular Singular Operators over Series (*example*), 197
Regularity(M) (*function*), 113
RegularSequence(I) (*function*), 37
Related Differential Operators (*example*), 188
Related Structures of a Differential Field (*example*), 167
Related Structures of a Laurent Series Ring (*example*), 168
Related Structures of an Operator Ring (*example*), 184
RelationIdeal(Q) (*function*), 47
RelationIdeal(Q, T) (*function*), 47
RelationIdeal(R) (*function*), 145
RelationMatrix(M) (*function*), 96
RelationModule(M) (*function*), 96
Relations(M) (*function*), 96
Relations(R) (*function*), 145
Relative Precision of a Derivation (*example*), 170
Relative Precision of a Series Ring (*example*), 170
Relative Precision of an Operator Ring (*example*), 185
RelativePrecision(F) (*function*), 170
RelativePrecisionOfDerivation(F) (*function*), 170
RelativePrecisionOfDerivation(R) (*function*), 185
Representation(F) (*function*), 159
Representation(R) (*function*), 154
RepresentationMatrix(f) (*function*), 84
Resolution over an Exterior Algebra (*example*), 115
Reynolds Operator and Invariants of a Degree (*example*), 134
ReynoldsOperator(f, G) (*function*), 133
Right Hand Factors for a Square Newton Polynomial (*example*), 208
Right Hand Factors Matching an LCLM Factorisation (*example*), 207
Right Hand Factors Where Factorisation Fails (*example*), 208
RightHandFactors(L) (*function*), 207
RingOfFractions(Q) (*function*), 88
RingOfFractions(R) (*function*), 165
RModule(R, k) (*function*), 95
RModule(R, W) (*function*), 95
Runge-Kutta 2 System (*example*), 19

S

s * t (*operation*), 171, 186
s + t (*operation*), 171, 185
s / t (*operation*), 172
s - t (*operation*), 171, 186

s ^ n (*operation*), 171, 186
s div t (*operation*), 171
s eq t (*operation*), 172, 186
SAT(B) (*function*), 32
Saturation(I) (*function*), 37
Saturation(I, f) (*function*), 36
Saturation(I, J) (*function*), 37
Secondary Invariants of a Cyclic Group (*example*), 139
SecondaryInvariants(R) (*function*), 138
SecondaryInvariants(R, H) (*function*), 138
SeparatingElement(F) (*function*), 166
SetAllInvariantsOfDegree(R, d, Q) (*procedure*), 135
SetFaugereModular(f) (*procedure*), 14
SetGBGlobalModular(f) (*procedure*), 14
SetsOfSingularPlaces(L) (*function*), 197
SetSparseMonomialMinRank(R) (*procedure*), 4
Setting All Invariants of a Degree (*example*), 135
SetVerbose("Buchberger", v) (*procedure*), 18
SetVerbose("Decomposition", v) (*procedure*), 52
SetVerbose("Faugere", v) (*procedure*), 18
SetVerbose("FGLM", v) (*procedure*), 18
SetVerbose("Groebner", v) (*procedure*), 18
SetVerbose("GroebnerWalk", v) (*procedure*), 18
SetVerbose("HilbertGroebner", v) (*procedure*), 31
SetVerbose("Invariants", v) (*procedure*), 133
SetVerbose("Resolution", v) (*procedure*), 110
Singular Places of an Operator (*example*), 197
Slope Valuation of an Operator (*example*), 202
SlopeValuation(L,s) (*function*), 202
SmallBasis(I) (*function*), 15
Solving a Boolean Polynomial System (*example*), 20
Solving a Boolean System with SAT (*example*), 33
SPolynomial(f, g) (*function*), 16, 97
Standard Basis of an Ideal (*example*), 74
Standard Basis Versus Groebner Basis (*example*), 75
StandardBasis(I) (*function*), 73
StandardBasis(S) (*function*), 73
Steenrod Operations (*example*), 148
SteenrodOperation(f, i) (*function*), 148
sub<M | L> (*constructor*), 102
Submodule(I) (*function*), 102
Submodules and Quotients of a Reduced Module (*example*), 104
Submodules and Quotients of an Embedded Module (*example*), 103
Symmetric Polynomial in Elementary Terms (*example*), 65, 162
Symmetric Powers of Operators (*example*), 201
SymmetricPower(L, m) (*function*), 201
Syzygy Matrix (*example*), 64
SyzygyMatrix(Q) (*function*), 64
SyzygyModule(M) (*function*), 107
SyzygyModule(Q) (*function*), 107

T

Tensor Products and Tor Modules (*example*), 123
TensorProduct(C, N) (*function*), 123
TensorProduct(M, N) (*function*), 123
Terms(f) (*function*), 98
Terms(L) (*function*), 187
TjurinaNumber(f) (*function*), 43
Tor(i, M, N) (*function*), 123
Transitive Groups of Degree 7 (*example*), 140
Translation(L, e) (*function*), 188
TranslationMap(R, e) (*function*), 189
Triangular Decomposition of Cyclic-5 (*example*), 56
TriangularDecomposition(I) (*function*), 56
Truncate(s) (*function*), 172
TruncateCoefficients(L) (*function*), 188
Truncating Operator Coefficients (*example*), 188
Twist(M, d) (*function*), 106
Type(L) (*function*), 185
Type(R) (*function*), 167, 183
Type(s) (*function*), 171

U

UnderlyingField(R) (*function*), 167
UnderlyingRing(R) (*function*), 167
Unexpected Factorisation of an Operator (*example*), 206
UnitVector(M, i) (*function*), 96
Univariate Generators by Change of Order (*example*), 30
UnivariateEliminationIdealGenerator(I, i) (*function*), 45
UnivariateEliminationIdealGenerators(I) (*function*), 45

V

v * f (*operation*), 100
VariableExtension(I, k, b) (*function*), 48
VariableExtension(I, k, b, order) (*function*), 48
VariableWeights(P) (*function*), 6
Variety over a Finite Field (*example*), 42
Variety(I) (*function*), 41
Variety(I, L) (*function*), 41
VarietySequence(I) (*function*), 42
VarietySequence(I, L) (*function*), 42
VarietySize(I) (*function*), 42
Vector(f) (*function*), 97
VectorSpace(Q) (*function*), 84

W

WeakDegree(L) (*function*), 187

`WeakOrder(L)` (*function*), 187
`WeightedDegree(f)` (*function*), 6, 98
`Wronskian Matrix and Determinant` (*example*), 182
`WronskianDeterminant(L)` (*function*), 182
`WronskianMatrix(L)` (*function*), 181

Z

`Zero(M)` (*function*), 96
`Zero(R)` (*function*), 166, 183