CASA Reference Manual (Version 2.3) [1]

Quoc-Nam Tran, Franz Winkler

Research Institute for Symbolic Computation (RISC–Linz),
Johannes Kepler University
A–4040 Linz, Austria
E-mail: {tqnam, winkler}@risc.uni-linz.ac.at

October 22, 1997

[1]Supported by the Austrian Science Foundation (FWF) project HySaX, Proj. No. P11160-TEC

Contents

Chapter 1

Introduction

CASA is a special-purpose system for computational algebra and constructive algebraic geometry. The system has been developed since 1990. With this newest version – version 2.3 – CASA is the ongoing product of the Computer Algebra Group at the Research Institute for Symbolic Computation (RISC-Linz), the University of Linz, Austria, under the direction of the second author. The system is built on the kernel of the widely used computer algebra system Maple@V.4. (An older version of CASA – version 2.3 ALPHA – is still available for Maple V.3 under the subdirectory `libV3`.)

1.1 What is CASA

Constructive methods in algebra and algebraic geometry are gaining more and more importance with the availability of computers and computer algebra softwares. Since its first version (see [1]), CASA has been designed to perform computations and utilize reasoning about algebraic and geometrical objects in the classical affine and projective spaces over algebraically closed fields of characteristic zero. More precisely, the field has to be a computable field in the sense of the underlying computer algebra system Maple, i.e. all the arithmetic operations have to be available in the system. Usually, the field of computation is the rational numbers $\mathbb{Q}$ or a finite algebraic extension thereof.

The system has been developed since 1990. With this newest version – version 2.3 – CASA is the ongoing product of the Computer Algebra Group at the Research Institute for Symbolic Computation (RISC-Linz), the University of Linz, Austria. Several people have contributed to the system in one way or another. The names of the main contributors (see [14, 3], [13], [2], [5, 6, 9, 7] and [10]) are K. Aigner, R. Gebauer, E. Hillgarter, M. Kalkbrener, M. Mňuk, J. R. Sendra, P. Stadelmeyer, Q.-N. Tran (current coordinator), B. Wall, F. Winkler (director).

In the system, an algebraic set – a central notion in algebraic geometry – can be represented in four different ways:

- *Implicit representation:* An algebraic set is the set of common zeros of a system of polynomial equations. To give an algebraic set in implicit form means to give finitely many polynomials.
- *Projected representation:* As a consequence of the primitive element theorem every irreducible d–dimensional algebraic set in n–dimensional space is, after a suitable linear transformation of coordinates, birationally projectable onto an irreducible d–dimensional algebraic set in $(d+1)$–dimensional space, which can be specified by a single polynomial in $d+1$ variables. This can be generalized to unmixed-dimensional algebraic sets. An algebraic set in projected form is given by a polynomial and a tuple of rational functions (specifying the birational mapping).
- *Parametric representation:* Some irreducible algebraic sets can be parametrized by rational functions. An algebraic set in parametric form is given by a tuple of rational functions that parametrizes the

algebraic set.

- *Representation by places:* All algebraic curves can be locally parametrized by a set of power series that are convergent around a point of the curve. An algebraic set is given by places if for each branch passing through a certain point on the algebraic set a tuple of power series that parametrizes the algebraic set around the point is specified.

The system provides a variety of operations on algebraic sets. As the efficiency of these operations is tightly bound to the way algebraic sets are represented, conversion routines have been provided to support various views on one object, to deepen the understanding of its principles, and to speed up algorithms working on algebraic sets. CASA also works with the polynomial ideals corresponding to these geometric objects.

The basic operations available in CASA include:

- ideal theoretic operations $+, *, \cap, /$,
- creating algebraic sets in different representations,
- generating curves of fixed multiplicities at given points,
- intersection, union, and difference of algebraic sets,
- computing tangent cones and tangent spaces,
- computation of the dimension of an algebraic set,
- decomposition into irreducible components,
- transformations of algebraic sets to hypersurfaces,
- computation of the singularities, genus, neighborhood graphs and adjoint curves of an algebraic curve.

Besides these basic operations, the following more advanced operations are available:

- rational parametrization of rational curves over an optimal extension field of coefficients,
- implicitization of parametrically given algebraic sets,
- Puiseux series expansions,
- multivariate resultants and Dixon's resultants,
- Gröbner bases of ideals and modules,
- Gröbner walk,
- hybrid methods for finding solutions of an arbitrary system of equations,
- computation of rational points on conics,
- offset curves,
- plotting both explicitly and implicitly given curves and surfaces,
- computation of syzygy-bases.

The major goal of CASA is to provide a comfortable, easy to use, efficient, flexible and mathematically exact working environment for computational algebra and constructive algebraic geometry where all basic theoretical concepts map easily to available data structures. CASA is built on the kernel of Maple and is fully independent of the operating system; hence, it can be used on every hardware where Maple is running.

There are software systems which partially cover some of the fields of CASA, however, to the authors' knowledge, currently no other system provides all the functionalities of CASA.

1.2 A Quick Tour

1.2.1 Getting Started

CASA is distributed as a single Maple archive of compiled routines and is loaded via the `with` function after having the language Maple loaded.

```
> with(casa):
```

```
           |__|
           |  |         Welcome to CASA 2.3
    |    /\|  |/\
    /=\__|  []  |
   /            \_      Copyright (C) 1990-1997 by Research Institute
   |              \     for Symbolic Computation (RISC-Linz), the
   \    CASA 2.3  |     University of Linz, A-4040 Linz, Austria.
    |             |
   _|      |||    |     For help type '?casa' or '?casa,<topic>'.
__/        |||    |_
```

The code is organized into a number of separate modules. The modules and their major parts are loaded into memory using the `readlib` facility when they get used for the first time. This significantly cuts the amount of memory occupied by the code.

1.2.2 Basic Operations

We can define an algebraic set in implicit, parametric or projected representation.

```
> A:= mkImplAlgSet([x*(x^2+y^2-1),x*z,x^2+y^2+z^2-1], [x,y,z]);
```

$$A := \text{algebraic\_set}(\,[\,x\,(\,x^2+y^2-1\,),x\,z,x^2+y^2+z^2-1\,],[\,x,y,z\,]\,)$$

```
> A2 := mkParaAlgSet([t^2-1,1/(t-1),t+1],[t]);
```

$$A2 := \text{algebraic\_set}\left(\left[t^2-1,\frac{1}{t-1},t+1\right],[\,t\,]\right)$$

```
> A3:=mkProjAlgSet([[u^2-v^3+v^2],[u+v,u-v,1/v]],[u,v]);
```

$$A3 := \text{algebraic\_set}\left(\left[[\,u^2-v^3+v^2\,],\left[u+v,u-v,\frac{1}{v}\right]\right],[\,u,v\,]\right)$$

We may perform conversion between representations, for example:

1. Birational mapping: We may compute a projection of A. The result is a set of algebraic sets, each given as a hypersurface and a birational mapping to 3D-space. The union of these algebraic sets is equal to A, the intersection of the corresponding ideal is equal to the radical of A. In this case we obtain two one-dimensional circles.

```
> Aproj:=convertRep(A,proj);
```

$$\begin{gathered} Aproj := \text{algebraic\_set}(\,[\,[\,x^2+y^2-1\,],[\,x,y,0\,]\,],[\,y,x\,]\,), \\ \text{algebraic\_set}(\,[\,[\,x^2+z^2-1\,],[\,0,-x,z\,]\,],[\,x,z\,]\,) \end{gathered}$$

2. Parametrization:

```
> B:= mkImplAlgSet([x^2-y^3,z+y+x]);
```

$$B := \text{algebraic\_set}(\,[\,x^2-y^3,z+y+x\,],[\,x,y,z\,]\,)$$

```
> Bpara:=convertRep(B,para);
```

$$Bpara := \text{algebraic\_set}\Bigg(\\ \left[-\frac{3\,t+t^3+3\,t^2+1}{3\,t-3\,t^2-1+t^3},\frac{1+t^2+2\,t}{t^2-2\,t+1},2\,\frac{1+t^2+2\,t}{3\,t-3\,t^2-1+t^3}\right],[\,t\,]\Bigg)$$

3. Representation by places (Puiseux expansion)

```
> Bplac:=impl2plac(B):
> shAlgSet(Bplac,10);
The algebraic set is known to have the following properties:
It is given by the following places:
```

$$\Big[\Big[\frac{1}{8}T^3+\frac{3}{32}T^4+\frac{21}{256}T^5+\frac{5}{64}T^6+\frac{1287}{16384}T^7+\frac{21}{256}T^8+\frac{46189}{524288}T^9+ \mathrm{O}(T^{10}), \frac{1}{4}T^2+\frac{1}{8}T^3+\frac{3}{32}T^4+\frac{21}{256}T^5+\frac{5}{64}T^6+\frac{1287}{16384}T^7+\frac{21}{256}T^8 +\frac{46189}{524288}T^9+\mathrm{O}(T^{10}), -\frac{1}{4}T^2-\frac{1}{4}T^3-\frac{3}{16}T^4-\frac{21}{128}T^5-\frac{5}{32}T^6- \frac{1287}{8192}T^7-\frac{21}{128}T^8-\frac{46189}{262144}T^9+\mathrm{O}(T^{10})\Big]\Big]$$

1.2.3 Set and Ideal Operations

The implicit representation of algebraic sets suggests a possibility of dealing with algebraic sets by working on their corresponding defining ideals. The radical of an ideal generates set-theoretically the same algebraic set (possibly without multiple components).

```
> A:= mkImplAlgSet({x*(x^2+y^2-1),x*z,x^2+y^2+z^2-1}, {y,z,x});
```

$$A := \mathrm{algebraic\_set}(\,[\,x\,(\,x^2+y^2-1\,), x\,z, x^2+y^2+z^2-1\,], [\,x, y, z\,]\,)$$

```
> radA := computeRadical(A);
```

$$radA := \mathrm{algebraic\_set}(\,[\,x^2+y^2-1, z\,], [\,x, y, z\,]\,),\\ \mathrm{algebraic\_set}(\,[\,x, y^2+z^2-1\,], [\,x, y, z\,]\,)$$

During the computation of the radical, the set was split into two components. We can compute the union of these two algebraic sets by computing the intersection (i.e. the least common multiple) of their ideals.

```
> G1:=implUnionLCM(radA);
```

$$G1 := \mathrm{algebraic\_set}(\,[\,x^2+y^2+z^2-1, x\,z, z\,y^2+z^3-z\,], [\,x, y, z\,]\,)$$

```
> implEqual(A,G1);
```

$$true$$

This gives us the ideal we started with, hence we know that the ideal by which A was given is a radical ideal.

The set-theoretic difference of sets is computed by determining the quotients of ideals.

```
> H:=mkImplAlgSet([x, y^2+z^2-1],[x, y, z]);
```

$$H := \mathrm{algebraic\_set}(\,[\,x, y^2+z^2-1\,], [\,x, y, z\,]\,)$$

We can compute the difference of the sets A and H and form the Zariski closure.

```
> implDifference(A,H);
```

$$\mathrm{algebraic\_set}(\,[\,x^2+y^2-1, z\,], [\,x, y, z\,]\,)$$

1.2.4 Computing Dimensions and Irreducible Decomposition

The dimension is an important invariant of algebraic sets. The system CASA allows to compute the dimension of sets given in implicit representation. Let us define an algebraic set in three dimensions.

```
> N:=mkImplAlgSet([-y^2*z^2 + 10*y^2*z - y*z^2 - 25*y^2 + 6*y*z + x - 5*y +1,
> y^3*z^2 - 10*y^3*z + 25*y^3 - y^2*z + 5*y^2 - y*z + y]);
```

$$N := \mathrm{algebraic\_set}([-y^2\,z^2 + 10\,z\,y^2 - y\,z^2 - 25\,y^2 + 6\,z\,y + x - 5\,y + 1, \\ z^2\,y^3 - 10\,z\,y^3 + 25\,y^3 - z\,y^2 + 5\,y^2 - z\,y + y], [x, y, z])$$

N turns out to be a curve.

```
> dimension(N);
```

$$1$$

There are two possibilities to compute the irreducible decomposition of N. The first one is based on characteristic sets and works directly on the implicit representation of N.

```
> N;
```

$$\mathrm{algebraic\_set}([-y^2\,z^2 + 10\,z\,y^2 - y\,z^2 - 25\,y^2 + 6\,z\,y + x - 5\,y + 1, \\ z^2\,y^3 - 10\,z\,y^3 + 25\,y^3 - z\,y^2 + 5\,y^2 - z\,y + y], [x, y, z])$$

```
> U:=decompose(N);
```

$$U := \mathrm{algebraic\_set}([x + 1, y], [x, y, z]), \mathrm{algebraic\_set}([\\ -11\,y^3\,x - 76\,y^3 + x^2\,y^3 - 15\,y^2\,x - 15\,y^2 - 3\,y - 3\,y\,x - 1 - x, 2\,x^3\,y^2 \\ + 2\,x\,z - 1064\,y^2 - 306\,y^2\,x - 108 - 8\,x^2\,y^2 - 16\,y\,x^2 - y\,x^3 - 197\,y\,x \\ - 44\,x - 438\,y + x^2\,z + 65\,z, \\ 20\,z\,y + x\,z + 5\,z - 152\,y^2 - 22\,y^2\,x + 2\,x^2\,y^2 - y\,x^2 - 19\,y\,x - 54\,y - 4 \\ , 4\,z^2 + x\,z - 23\,z - 2\,x^2\,y^2 + 22\,y^2\,x + 152\,y^2 + 106\,y + 41\,y\,x + 36 \\ + 16\,x - y\,x^2], [x, y, z])$$

The second decomposition algorithm works on algebraic sets in projected form by factoring the defining polynomial of the hypersurface.

```
> decompose(convertRep(N,impl,proj));
```

$$\mathrm{algebraic\_set}([[x + 1], [-(266\,x\,z^3 - 232\,x\,z^2 - 104\,x\,z^4 - 9\,x\,z + 95\,x \\ + 17\,x\,z^5 - x\,z^6 - 11\,z^2 + 9\,z^3 - 55\,z + 49 - z^4 + z^2\,x^2 - 10\,x^2\,z \\ + 25\,x^2)/(\%1), -(-11\,z^2 + 9\,z^3 - 55\,z + 49 - z^4 - 10\,x\,z^2 - 65\,x\,z \\ + 74\,x + 9\,x\,z^3 - x\,z^4 + z^2\,x^2 - 10\,x^2\,z + 25\,x^2)/(\%1), z]], [x, z]), \\ \mathrm{algebraic\_set}([[z^2\,x^2 - 10\,x^2\,z + 25\,x^2 - 3\,x\,z^3 + 34\,x\,z^2 - 114\,x\,z \\ + 95\,x - z^5 + 13\,z^4 - 59\,z^3 + 126\,z^2 - 174\,z + 91], [-(266\,x\,z^3 \\ - 232\,x\,z^2 - 104\,x\,z^4 - 9\,x\,z + 95\,x + 17\,x\,z^5 - x\,z^6 - 11\,z^2 + 9\,z^3 \\ - 55\,z + 49 - z^4 + z^2\,x^2 - 10\,x^2\,z + 25\,x^2)/(\%1), -(-11\,z^2 + 9\,z^3 \\ - 55\,z + 49 - z^4 - 10\,x\,z^2 - 65\,x\,z + 74\,x + 9\,x\,z^3 - x\,z^4 + z^2\,x^2 \\ - 10\,x^2\,z + 25\,x^2)/(\%1), z]], [x, z]) \\ \%1 := -257\,z^3 + 222\,z^2 + 103\,z^4 - 56\,z - 21 - 17\,z^5 + z^6$$

1.2.5 Algebraic Curves

We define an irreducible projective algebraic curve.

```
> C1:=mkImplAlgSet([-15*y^2*z^3-76*y^3*z^2-z^5-3*y*z^4-15*x*y^2*z^2-11*x*y^3*z-x*z
> ^4-3*x*y*z^3+y^3*x^2],[x,y,z],[basespace=projective]);
```

$$C1 := \mathrm{algebraic\_set}([-15\,y^2\,z^3 - 76\,z^2\,y^3 - z^5 - 3\,z^4\,y - 15\,x\,z^2\,y^2 - 11\,x\,y^3\,z \\ - x\,z^4 - 3\,x\,y\,z^3 + x^2\,y^3], [x, y, z])$$

In the course of studying basic properties of C1 we will look for singular points. The result of the following function call is a list of all singularities of C1 (counted properly). The singular points are collected according to their multiplicities, i.e. first the double points, then the triple points, and so on. Usually, algebraic numbers are introduced in this process.

```
> singularities(C1);
```

$$\begin{aligned}&\mathrm{table}([\\ &2 = \left[\left[\frac{40}{21}\,\mathrm{RootOf}(\,5\,\_Z^2 + 4\,\_Z + 89\,) - \frac{5}{21},\right.\right.\\ &\left.\left.-\frac{2}{21}\,\mathrm{RootOf}(\,5\,\_Z^2 + 4\,\_Z + 89\,) - \frac{5}{21}, 1\right], [\,0, 1, 0\,]\right]\\ &3 = [\,[\,1, 0, 0\,]\,]\\ &])\end{aligned}$$

Let us take a point on C1, (2:alpha:1), say; "alpha" being a root of the irreducible polynomial:

$$94\,y^3 + 45\,y^2 + 9\,y + 3.$$

```
> alias(alpha=RootOf(94*y^3+45*y^2+9*y+3));
```

$$I, \alpha$$

The tangent space (i.e. the set of points on all lines which have the intersection multiplicity > 1) at this point is the following algebraic set (a line):

```
> tangSpace(C1,[2,alpha,1]);
```

$$\begin{aligned}&\mathrm{algebraic\_set}\Big(\\ &\left[-657\,y + 643\,z\,\alpha + 322\,\alpha^2\,z + 48\,z - 48\,x + 14\,\alpha\,x - 322\,\alpha^2\,x\right],\\ &[\,x, y, z\,]\Big)\end{aligned}$$

Since the projective point (0:1:0) is a singular point on C1, the tangent space is the whole projective space.

```
> Ts:=tangSpace(C1,[0,1,0]);
```

$$Ts := \mathrm{algebraic\_set}(\,[\,0\,], [\,x, y, z\,]\,)$$

The tangent cone is a linear subspace generating the tangent space. It is given by the following set.

```
> Tc:=tangSpace(C1,[0,1,0],'cone');
Some of the generating polynomials for the affine tangent space/cone
are homogeneous. Returning only affine form of generators.
%
```

$$Tc := \mathrm{algebraic\_set}(\,[\,-11\,x\,z + x^2 - 76\,z^2\,], [\,x, z\,]\,)$$

The set Tc splits over the algebraic closure of the rational numbers into two lines.

```
> decompose(Tc,absolute);
```

$$\begin{aligned}&\mathrm{algebraic\_set}\left(\left[x + \frac{1}{4}\,\mathrm{RootOf}(\,44\,\_Z + \_Z^2 - 1216\,)\,z\right], [\,x, z\,]\right), \mathrm{al}\backslash\\ &\mathrm{gebraic\_set}\left(\left[x + \left(-11 - \frac{1}{4}\,\mathrm{RootOf}(\,44\,\_Z + \_Z^2 - 1216\,)\right)\,z\right], [\,x, z\,]\right)\end{aligned}$$

It is an interesting question whether coordinates of points on an algebraic set can be expressed as rational functions of some number of parameters. For curves this amounts to representing coordinates of each point in terms of a single parameter. A necessary and sufficient condition is the vanishing of the genus.

```
> genus(C1);
```

$$0$$

Now we are able to compute a birational mapping of C1 to the line. We get a parametric description.

```
> convertRep(C1,para);
```

$$\mathrm{algebraic\_set}\left(\left[-t^3 - 15\,t^2 - 75\,t - 126, -\frac{5 + t}{t^2 + 11\,t + 26}, 1\right], [\,t\,]\right)$$

Often it is desirable to determine (systems of) curves passing through specified points with prescribed multiplicity. The system CASA offers some functions to perform this task, e.g., passGenCurve.

1.2.6 Gröbner Bases and Syzygies

Besides the classical Gröbner bases algorithm, CASA contains an implementation of Gröbner bases for multivariate polynomial modules and is able to solve systems of linear equations with polynomial coefficients. CASA also contains an implementation of the Gröbner Walk algorithm for Gröbner bases conversion.

```
> msolveGB([[x^3-y,-x+y],[x*y-1,x^3-1],[-x+y^2,y-1]],[x*y^2-y^2,-x^5+x^2-x+y^
> 2],[x,y],term,tdeg);
```

$$\begin{aligned}&[[\,y, -x^2, x\,], [[-x^4 + x + x^3\,y^2 - y^2 - x\,y^2 + x\,y + y - 1,\\ &\quad x\,y + x^3\,y - x^3 - y^2 - y^3 + y - x^2 + x\,y^2,\\ &\quad x^3 + x^3\,y + x\,y^2 - x^2\,y + x - 2\,y - x^6]]]\end{aligned}$$

```
> GWalk([t^2+y^2+z^2-1,t^2+z^2-y,t-z],[t,y,z]);
```

$$[t - z,\; 2\,z^2 - y,\; 4\,z^4 - 1 + 2\,z^2]$$

1.2.7 Scientific Visualization

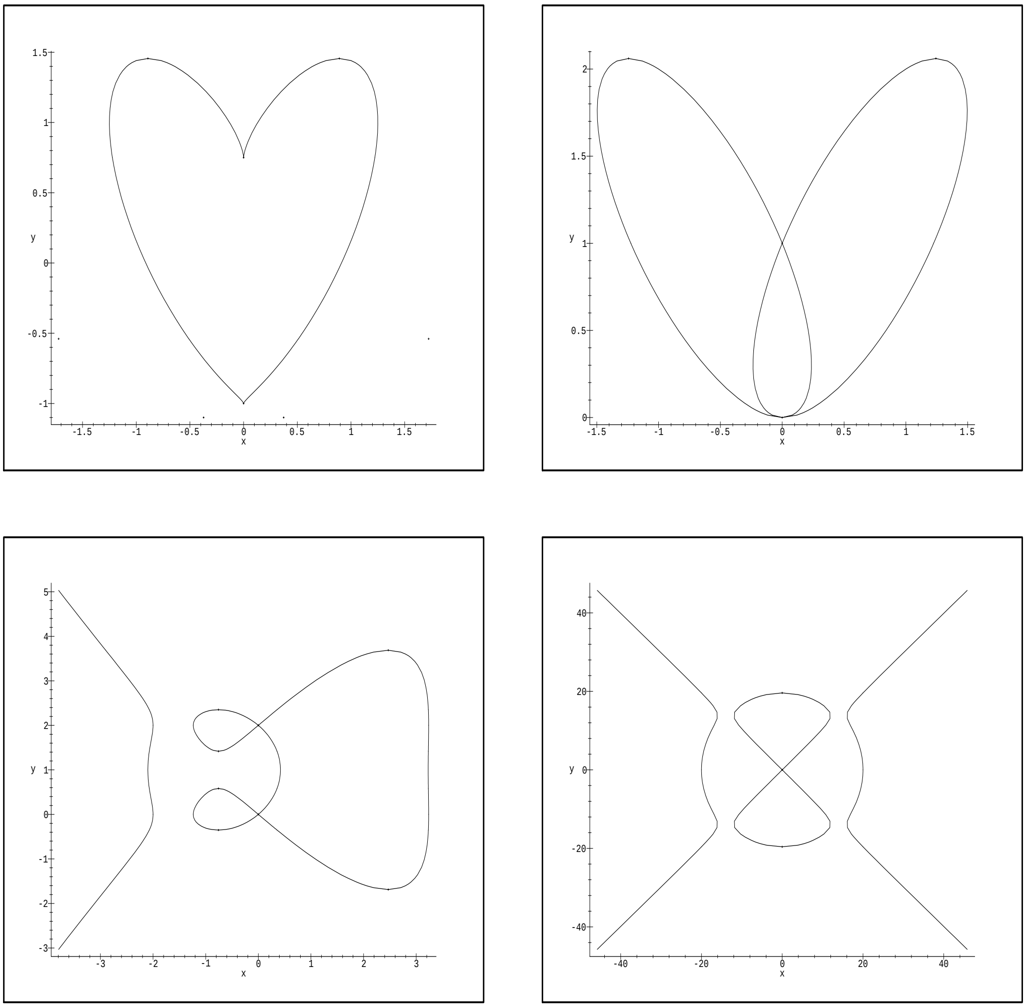
A powerful feature of CASA is the capability to visualize algebraic sets. For the majority of tasks, only singular or other distinguished points of algebraic sets yield some interesting information. These objects have to be treated with care, requiring special analysis. CASA makes a thorough effort to obtain correct information about the local topology in the neighborhood of singularities and other critical points while still keeping the algorithm efficient by using hybrid symbolic-numerical methods. Although Maple provides a function which is claimed to plot sets in implicit representation, its output for more complicated varieties is by far not satisfactory. Figure 1.1 shows the graphs produced by CASA for the following examples:

1. $\frac{93392896}{15625}x^6 + (\frac{94359552}{625}y^2 + \frac{(91521024}{625}y - \frac{249088}{125}x^4 + (\frac{1032192}{25}y^4 - 36864y^3 - \frac{7732224}{25}y^2 - 207360y + \frac{770048}{25}x^2 + (65536y^6 + 49152y^5 - 135168y^4 - 72704y^3 + 101376y^2 + 27648y - 27648$
2. $2x^4 - 3x^2y + y^2 - 2y^3 + y^4$
3. $y^4 - 96a^2y^2 + 100a^2x^2 - x^4$
4. $8(y^2 - x^2) - 8(y^3 + x^3) + 2y^4 + x^5$

Figure 1.2 shows the graphs produced by CASA for the following surface-to-surface intersections taken from literature.

1. $x - 1,\, y - 1$
2. $x^2 + y^2 + z^2 - 5,\, xyz - 1$
3. $z - 2x^4 - y^4,\, z - 3x^2y + y^2 - x^3$
4. $z - 2x^4 - y^4,\, z^2 - 3x^2y - y^3$
5. $z^2 + 3x^4 - y^4,\, z + x^3 - 3x^2y + y^2 - 2y^3$
6. $y^4 - 384y^2 + 400x^2 - x^4 + z^3,\, x^2 + y^2 + z^2 - 30$

Figure 1.1: Plotting



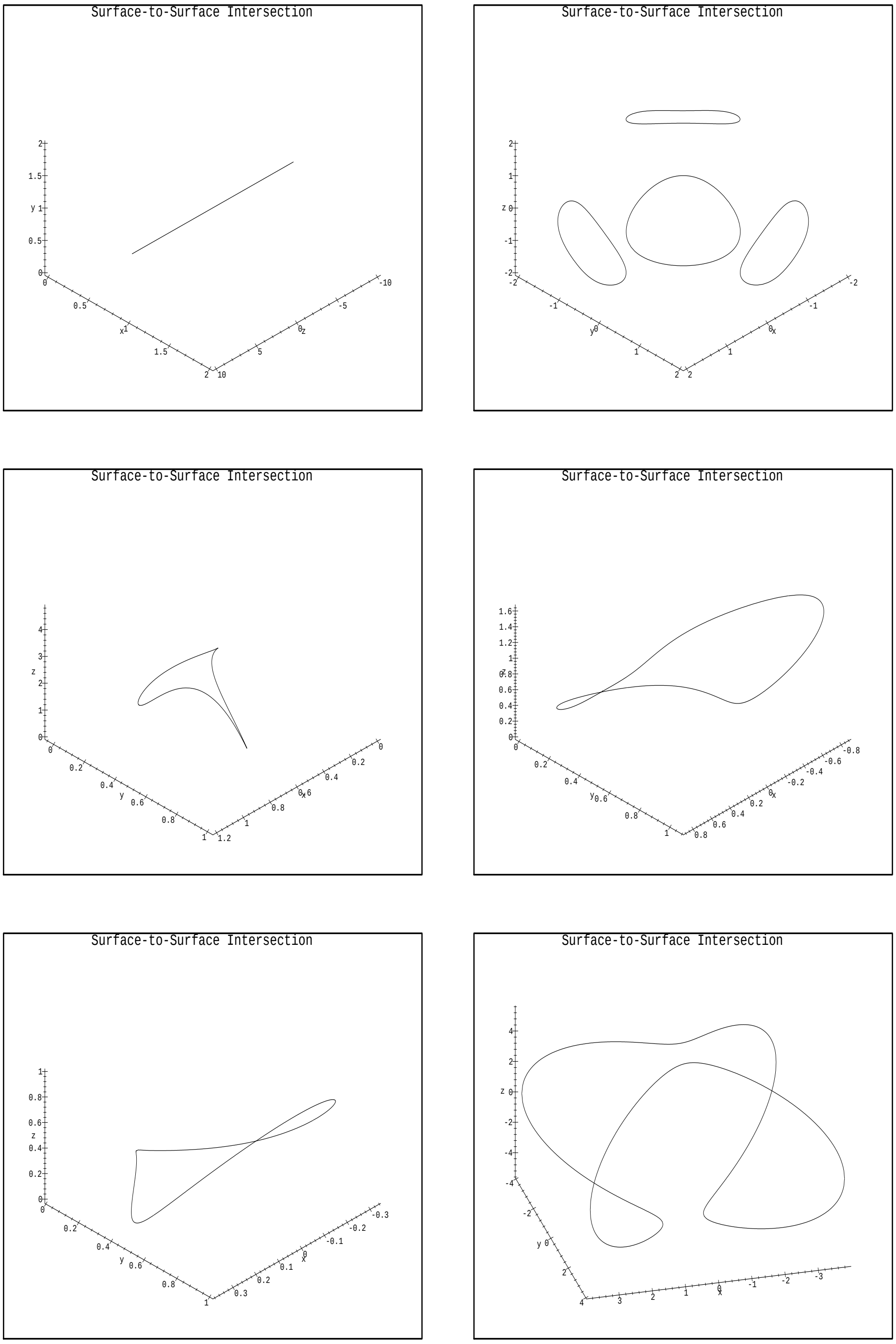
1.3 Distribution and Requirements

The system can be obtained by anonymous ftp at ftp.risc.uni-linz.ac.at. Major releases will be located in the directory /pub/CASA. Bug correction and minor updates will be put in /pub/CASA/update. We have also a home page on World Wide Web at http://www.risc.uni-linz.ac.at/software/casa.

The system is freely distributed under the following conditions: any research activity which uses CASA should cite the authors and the system explicitly. The system can be freely distributed to other users. New users are encouraged to notify the CASA coordinator so that they can be included in a user list where they will be kept up to date about the progress of the system.

Bug reports, questions and suggestions should be sent to the e-mail address alggeo@risc.uni-linz.ac.at.

Figure 1.2: Surface-to-Surface Intersections



Chapter 2

What's New

2.1 Help Pages

The help feature of CASA has been totally redesigned in order to match with features of the new release of the language Maple (Maple V.4). This new version of CASA provides online help pages with hyperlinks. Just click on a highlighted item to pull up its subsequent help page.

2.2 New and Improved Functionalities

2.2.1 Parametrization over an Optimal Field of Extension

The function `impl2para` for converting an algebraic set in implicit representation into parametric representation has been totally re-implemented using the new results in [4]. Currently, it may be applied to one-dimensional algebraic sets (i.e. both plane and space algebraic curves). With the optional argument *optimal* = *true*, the function will compute an optimal parametrization, i.e. a parametrization over the original ground field if possible, or otherwise over an algebraic extension field of degree 2. The user can supply some known simple points to the function by using the optional argument *points=[point1, point2, ...]*. The function will check if those points are simple points on the curve. Moreover, the points which are not on the curve will be automatically removed.

Example:

A rational algebraic curve of odd degree can always be parametrized over the base field of coefficients

```
> a1:=mkImplAlgSet([y^2*z^3-x^5],[x,y,z],[basespace=projective]);
```

$$a1 := \mathrm{algebraic\_set}([y^2\,z^3 - x^5],\, [x,\, y,\, z])$$

```
> impl2para(a1);
```

$$\mathrm{algebraic\_set}([t^2,\, -t^5,\, 1],\, [t])$$

A rational algebraic curve of even degree may be parametrized over a near optimal extension field of coefficients (i.e. extension of degree at most two).

```
> a2:=mkImplAlgSet([y^4-10*x*y^3+35*x^2*y^2-50*x^3*y+24*x^4+x^3],[x, y]);
```

$$a2 := \mathrm{algebraic\_set}([y^4 - 10\,x\,y^3 + 35\,x^2\,y^2 - 50\,y\,x^3 + 24\,x^4 + x^3],\, [x,\, y])$$

```
> impl2para(a2);
```

$$\mathrm{algebraic\_set}([(29 + 88\,t^4 + 40\,\mathrm{RootOf}(\_Z^2+1)\,t^4\,\mathrm{RootOf}(\_Z^2-7) + 72\,\%1 + 192\,t^3$$

$$- 128\,\%4 - 24\,\%3 - 408\,t^2 - 7\,\%2 + 40\,t)/(-2261\,t^4$$
$$+ 607\,\mathrm{RootOf}(\_Z^2+1)\,t^4\,\mathrm{RootOf}(\_Z^2-7) + 4602\,t^3 + 2482\,\%4 + 7277\,t^2 - 1879\,\%3$$
$$- 696 + 168\,\%2 - 2410\,t - 1378\,\%1), t$$
$$(7\,\%2 - 48\,t^3 + 32\,\%4 - 29 + 204\,t^2 + 12\,\%3 - 30\,t - 54\,\%1)/(-2261\,t^4$$
$$+ 607\,\mathrm{RootOf}(\_Z^2+1)\,t^4\,\mathrm{RootOf}(\_Z^2-7) + 4602\,t^3 + 2482\,\%4 + 7277\,t^2 - 1879\,\%3$$
$$- 696 + 168\,\%2 - 2410\,t - 1378\,\%1)],\ [t])$$
$$\%1 := \mathrm{RootOf}(\_Z^2+1)\,\mathrm{RootOf}(\_Z^2-7)\,t$$
$$\%2 := \mathrm{RootOf}(\_Z^2+1)\,\mathrm{RootOf}(\_Z^2-7)$$
$$\%3 := \mathrm{RootOf}(\_Z^2+1)\,t^2\,\mathrm{RootOf}(\_Z^2-7)$$
$$\%4 := \mathrm{RootOf}(\_Z^2+1)\,t^3\,\mathrm{RootOf}(\_Z^2-7)$$

or over the optimal extension field of coefficients if the option 'optimal=true' is specified.

```
> impl2para(a2,optimal=true);
```

$$\mathrm{algebraic\_set}([-625\,\frac{t^4}{2544190\,t^3 + 625 + 6344736\,t^4 + 380975\,t^2 + 25250\,t},$$
$$125\,\frac{t^3\,(38\,t+5)}{2544190\,t^3 + 625 + 6344736\,t^4 + 380975\,t^2 + 25250\,t}],\ [t])$$

A space curve can also be parametrized

```
> a3:=mkImplAlgSet([x^2-y^3,z+y+x]);
```

$$a3 := \mathrm{algebraic\_set}([x^2 - y^3,\ z + y + x],\ [x,\ y,\ z])$$

```
> impl2para(a3,parameter=c);
```

$$\mathrm{algebraic\_set}([\frac{1 + 3\,c + 3\,c^2 + c^3}{-1 + 3\,c - 3\,c^2 + c^3},\ \frac{1 + 2\,c + c^2}{c^2 - 2\,c + 1},\ -2\,\frac{c\,(1 + 2\,c + c^2)}{-1 + 3\,c - 3\,c^2 + c^3}],\ [c])$$

2.2.2 Hybid Symbolic-Numerical Methods

Finding Solutions of an Arbitrary System of Nonlinear Equations

In CAGD or solid modeling, we often need to find some solutions for a system of nonlinear equations where the field of coefficients may not be algebraically closed (for example the field of real numbers). The new function **extNewton** iteratively computes solutions for an arbitrary system of nonlinear equations, i.e. the system can be nonzero-dimensional or zero-dimensional. The system can be underdetermined or overdetermined. In case the number of equations is equal to the number of variables, the method is similar to Newton's method. The method is a hybrid symbolic-numerical method, in that we use symbolic techniques to transform a system of algebraic equations into a regular system whenever the system is not underdetermined or does not have a full-rank Jacobian matrix (see [9]).

Example:

```
> f1:=x1^2+x2^2+x3^2-1:
> f2:=x4*x1+x5*x2+x6*x3:
> f3:=(x4+3*x3)^2+x5^2+(x6-3*x1)^2-1:
> List_of_polys:=normal([f1,f2,f3]);
> Varlist:=[x1,x2,x3,x4,x5,x6];
```

$$List\_of\_polys := [x1^2 + x2^2 + x3^2 - 1,\ x4\,x1 + x5\,x2 + x6\,x3,$$
$$x4^2 + 6\,x4\,x3 + 9\,x3^2 + x5^2 + x6^2 - 6\,x6\,x1 + 9\,x1^2 - 1]$$

$$Varlist := [x1,\ x2,\ x3,\ x4,\ x5,\ x6]$$

```
> Func:=extNewtonFunc(List_of_polys,Varlist):
> so:=extNewton(Func,[rand(20)(),rand(20)(),rand(20)(),rand(20)(),rand(20)(),rand(20)()]);
```

$$so := [.714849223293176846,\ .0254139182380360877,\ .698816657453824081,\ -2.56280010705017309,\ .761979375842043262,\ 2.59388605770756449]$$

```
> sub:={'Varlist[i]=so[i]' $ 'i'=1..nops(so)};
> evalf(subs(sub,List_of_polys));
```

$$sub := \{x4 = -2.56280010705017309,\ x3 = .698816657453824081,\ x2 = .0254139182380360877,\ x1 = .714849223293176846,\ x6 = 2.59388605770756449,\ x5 = .761979375842043262\}$$

$$[-.1\,10^{-9},\ .1\,10^{-8},\ .2\,10^{-8}]$$

Real Root Isolation

The new function **realroot_sb** isolates real roots of a polynomial in strictly separated intervals in an open interval using non-recursive algorithms. The function returns a sorted list of strictly separated isolating intervals for all real roots of the univariate polynomial within the interval. If the interval is omitted, all realroots of the polynomial will be isolated (easier problem). The function now accepts polynomials with floating point coefficiences. The polynomial will be converted to a rational polynomial before isolating the realroots. For isolating real roots within an OPEN interval, the algorithm in this function avoids to use transformations and therefore does not blow up the coefficients of the polynomial. It is many times faster than algorithms using transformations. For example with `randpoly(x, coeffs = rand(-100..100), terms = 10, degree = 100)`, it is 20 times faster (see [6] for more details).

Example:

```
> realroot_sb(2.5*x^8+x^7-4*x^6-20*x^5+4*x^4+20*x^3, 1/1000);
```

$$[[\frac{-493}{512},\ \frac{-985}{1024}],\ [0,\ 0],\ [\frac{569}{512},\ \frac{1139}{1024}],\ [\frac{951}{512},\ \frac{1903}{1024}]]$$

```
> p1:=expand(subs({n=20,a=5},x^n-2*(a*x-1)^2)): # Mignotte polynomials
> realroot_sb(p1,[0,1]);
```

$$[[\frac{53687087}{268435456},\ \frac{107374175}{536870912}],\ [\frac{13421773}{67108864},\ \frac{6710887}{33554432}]]$$

The new function **realroot_a** isolates real roots of a polynomial with algebraic number coefficients in strictly seperated intervals using non-recursive algorithms and norm. The function returns a sorted list of strictly separated isolating intervals for all real roots of the polynomial.

Example:

```
> eq1:=expand(2*x^4-3*x^2*y+y^2*(y-1)^2);
> Ay:=8*y^2-16*y-1:
> alpha:=RootOf(Ay):
> eq2:=subs(y=alpha,eq1):
> # Algebraic sqrfree
> Geq2 := evala(Gcd(eq2,diff(eq2,x))):
> evala(Divide(eq2,Geq2,evaln(eq2))):
> eq2 := primpart(subs(alpha=y,eq2));
> r1:=`realroot_a/realroot`(Ay);
```

$$eq1 := 2\,x^4 - 3\,x^2\,y + y^4 - 2\,y^3 + y^2$$

$$eq2 := 4\,x^2 - 3\,y$$

$$r1 := [[\frac{-1}{16}, \frac{-1}{32}], [2, 3]]$$

```
> realroot_a(Ay,r1[1],eq2);
```

$$[]$$

```
> realroot_a(Ay,r1[2],eq2);
```

$$[[\frac{-3}{2}, -1], [1, \frac{3}{2}]]$$

2.2.3 Multivariate Resultants

Classical Multivariate Resultants

The new function **mvresultant** computes the resultant system of a system of multivariate polynomials in order to set up conditions for their solvability as well as formulas for calculating their solutions. The vanishing of the resultant system provides a solvability creterion for the existance of common solutions of the system (in projective space). We use the algorithm described in [11].

Example:

```
> F1:=[x2^2+x1^2-1,x1*x2-1,x2^2 -x1]:
> mvresultant(F1,[x1,x2]);
```

$$[4]$$

```
> F2:=[v*x2^2+RootOf(x^3+1)*x1^2-1,x1*x2-1,x2^2-x1]:
> mvresultant(F2,[x1,x2]);
```

$$[v^3 + 3\,v\,\mathrm{RootOf}(\_Z^3+1) + \mathrm{RootOf}(\_Z^3+1)^3 - 1]$$

```
> F3:=[x3-x1*x2-x2,x2*x3-x1,x1*x3-x2]:
> mvresultant(F3,[x1]);
```

$$[-x3 + x2 + x2^2\,x3,\; x2 - x2\,x3^2]$$

```
> mvresultant(F3,[x1,x2]);
```

$$[-16\,x3\,(121 - 58\,x3^2 + x3^4)]$$

Dixon's Resultant

The new function **dixon** computes an extended Dixon's resultant of a system of $n{+}1$ polynomials in n variables in order to set up necessary conditions for their solvability. The vanishing of the extended Dixon's resultant provides a necessarily solvability creterion for the existance of common solutions of the system (in projective space). This method is much more efficient than other methods for elimination. We use the algorithm described in [8].

Example:

```
> F1:=[x2^2+x1^2-1,x1*x2-1,x2^2 -x1]:
> dixon(F1,[x1,x2]);
```

$$-4$$

```
> F2:=[x1*y2-(y1^2-y2^2),x2*y1-(y1^2-y2^2),x3*y2-1,t*y1*y2-1]:
> dixon(F2,[t,y2,y1]);
```

$$x3\ x2\ (-x1^2 + x3\ x2^2\ x1 + x2^2)$$

2.2.4 Gröbner Walk

The new function **GWalk**(F, $X1$) computes the reduced, minimal Groebner basis of the polynomials F in the indeterminates $X1$ with respect to the lexicographical ordering. The algorithm first computes a Groebner basis with respect to *tdeg* ordering by using the function `grobner[gbasis]` and converts it to a lexicographical Groebner basis by the use of the Groebner Walk algorithm.

If the procedure is called with 3 arguments then the list of indeterminates $X2 := [x_1, x_2, ..., x_n]$ induces the ordering $x_1 > x_2 > \ldots > x_n$ and is used for the computation of the *tdeg* Groebner basis. Otherwise, $X1$ is used for this computation.

Example:

```
> GWalk([t^2+y^2+z^2-1,t^2+z^2-y,t-z],[t,y,z]);
```

$$[t - z,\ 2\,z^2 - y,\ 4\,z^4 - 1 + 2\,z^2]$$

```
> GWalk([1+x^2+z,x*z+y^2*z],[z,x,y],[x,y,z]);
```

$$[1 + x^2 + z,\ x^2\,y^2 + x^3 + y^2 + x]$$

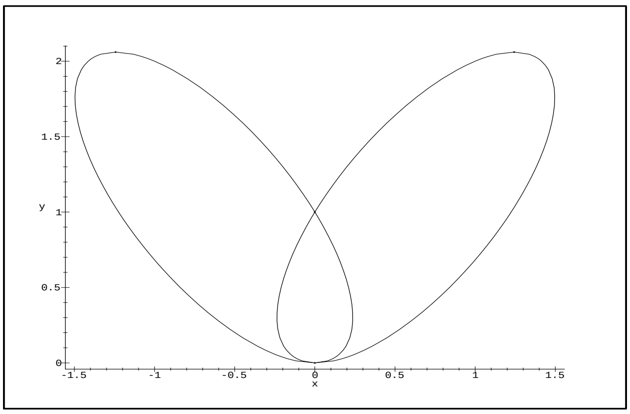
2.2.5 Scientific Visualization

Plotting plane algebraic curves

The new function `pacPlot` plots plane algebraic curves in implicit form using hybrid symbolic-numerical method. The range for the variables will be automatically choosen base on the critical points of the curve. Options are given as equations of the form *option=value*. All options that are supported by Maple's plotting functions (`plot[options]`) are accepted.

Example:

```
> pacPlot(2*x^4-3*x^2*y+y^2*(y-1)^2);
```

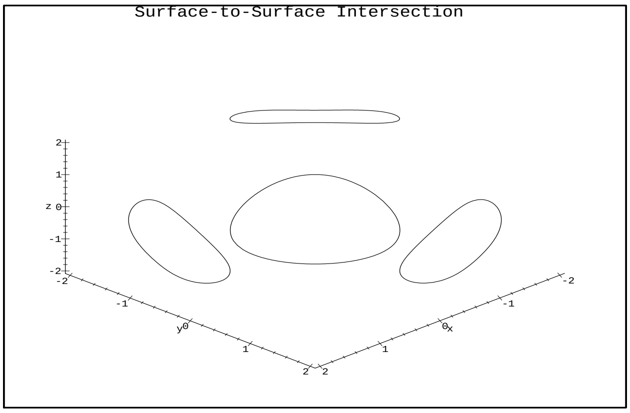


Plotting the intersection of two surfaces

The new function **ssiPlot** plots the intersection of two three-dimensional and implicitly defined surfaces using hybrid symbolic-numerical method. An appropriate range for each variable will be computed automatically. The user do not need to specify the ranges. Options are given as equations of the form *option=value*. The following options are supported: (1) All options that are supported by Maple's plotting functions (`plot[options]`). (2) *epsilon=-n*. Specifies precision for real root isolation procedures used in the function. (3) *maxiterate=n*. Specifies the maximum number of extended Newton iterations used in the function.

Example:

```
> ssiPlot(x^2+y^2+z^2 -5, x*y*z-1,[x,y,z],color=green);
```



2.2.6 Puiseux Series Expansion

Puiseux series expansion can now be used with both exact and floating point computation. See the descriptions for the functions **shAlgSet**, **impl2plac** and **proj2plac** for more details.

2.2.7 Rational Points on Conics

The function **ratpoint** decides whether there is a rational point on a conic, say g, and if so, computes one. The coefficients of g may be in $\mathbb{Q}$ or in $\mathbb{Q}(t)$ (and hence the coordinates of a rational point have to be in $\mathbb{Q}$ or in $\mathbb{Q}(t)$ respectively). Additional information is provided by the function such as the decision whether g defines a parabola or not, the irreducibility of the conic, the time needed for the calculation, and g evaluated at the computed point (i.e. 0).

2.2.8 Offset Curves

RPH Parametrization

The function **RPHcurve** checks if there exists an RPH (Rational Phytagorean hodograph) parametrization of an algebraic curve by analyzing the reducibility and rationality of the reparametrizing curve. If the reparametrizing curve is irreducible and rational then a quadratic reparametrizing function is assigned to the parameter.

Offset Curve in Implicit Representation

The function `implOffset` computes the offset curve in implicit representation at distance d to a given plane algebraic curve (in implicit or parametric representation).

Offset Curve in Parametric Representation

The function `paraOffset` computes the offset curve in parametric representation at distance d to a given plane algebraic curve (in implicit or parametric representation).

2.3 Bugs Fixed

Bugs of the following functions in the previous versions have been fixed. We thank the users who pointed out the bugs.

- `singularities`.
- `neighbGraph`
- `genus`
- `dimension`
- `decompose`
- `proj2plac`

Chapter 3

Data Types

The main data type in CASA is algebraic set. It is an unevaluated function call with head `algebraic_set` and three parameters. The first one specifies the representation, the second one contains a list of variables occuring in the representation, and the third argument is a table of attributes.

Function: type/algset

Action: check for algebraic set

Calling sequence:

`type(`*expr*`, algset)`
`type(`*expr*`, algset(impl))`
`type(`*expr*`, algset(para))`
`type(`*expr*`, algset(proj))`
`type(`*expr*`, algset(plac))`
`type(`*expr*`, algset(fin))`

Arguments:

expr – any expression

Synopsis:

- The call `type(`*expr*`,algset)` yields `true` if *expr* is an algebraic set in any representation.
- The call `type(`*expr*`,algset(`*repr*`))` checks whether *expr* is an algebraic set in specified representation.

Example:

```
> a1:=mkParaAlgSet([t^2,t^3],[t]);
                                         2   3
                     a1 := algebraic_set([t , t ], [t])

> op(a1);
                             2   3
                           [t , t ], [t], attr

> op(op(3,a1));

table([

    rep = para

    oth = []
```

```
 ])

> type(a1,algset(para));

                              true

> type(a1,algset(proj));

                              false
```

See also: `mkImplAlgSet`, `mkParaAlgSet`, `mkProjAlgSet`, `mkPlacAlgSet`

Function: type/casaCoeffType

Action: check for proper type of coefficients CASA system is working on

Calling sequence:

`type(`*expr*`,casaCoeffType)`

Arguments:

expr – any expression

Synopsis:

- This procedure checks whether *expr* is of type over which polynomials and varieties in CASA may be defined.

Example:

```
> a1:=RootOf(x^2+1);
                                  2
                   a1 := RootOf(_Z  + 1)

> type(a1,casaCoeffType);

                          true
```

See also: `casa`

Function: type/planecurve

Action: check for a plane algebraic curve or a system of plane algebraic curves

Calling sequence:

`type(`*expr*`,planecurve)`
`type(`*expr*`,planecurve(system))`

Arguments:

expr – any expression

Synopsis:

- The call `type(`*expr*`,planecurve)` checks whether *expr* defines a plane algebraic curve, i.e. *expr* has to be an algebraic set in implicit representation represented by a sigle homogeneous polynomial in three variables.
- The call `type(`*expr*`,planecurve(system))` checks whether *expr* is a system of plane curves, i.e. if it contains additional free variables other than those from `varlist(`*expr*`)`.

Example:

```
> a1:=mkImplAlgSet([x^2-y^2],[x,y]);

                                   2    2
                  a1 := algebraic_set([x  - y ], [x, y])

> a2:=mkImplAlgSet([x^2-c*y^2],[x,y]);

                                  2      2
                 a2 := algebraic_set([x  - c y ], [x, y])

> type(a1,planecurve);

                                  true

> type(a2,planecurve);

                                 false

> type(a2,planecurve(system));

                                  true
```

See also: `mkImplAlgSet`

Function: type/projPoint

Action: check for a valid point in projective space

Calling sequence:

`type(`*point*`,projPoint)`

Arguments:

point – list of coordinates

Synopsis:

- This function checks whether *point* is a valid point in a projective space, i.e. whether at least one coordinate is different from zero.

Example:

```
> alias(alpha=RootOf(x^3-2)); >

                                I, alpha

> type([1,0,0,1],projPoint);

                                  true

> type([0,0,0,alpha^3-2],projPoint);

                                 false
```

Function: type/projalgpoly

Action: check for homogeneity of a polynomial or a system of polynomials

Calling sequence:

`type(`*expr*`,projalgpoly)`
`type(`*expr*`,projalgpoly(system))`
`type(`*expr*`,projalgpoly(`*vars*`))`
`type(`*expr*`,projalgpoly(`*vars*`,system))`

Arguments:

expr – any expression
vars – list of variables

Synopsis:

- The call `type(`*expr*`,projalgpoly)` checks whether *expr* is a homogeneous polynomial with coefficients of type `casaCoeffType` in variables specified by a global variable `casa/homvars`. The call `type(`*expr*`,projalgpoly(system))` checks for a system of such polynomials.
- The calls `type(`*expr*`,projalgpoly(`*vars*`))` and type(*expr*,projalgpoly(*vars*,system)) specify additionally a variable list – *vars*.

Example:

```
> type(x^2-a*y^2,projalgpoly([x,y],system));

                              true
```

Chapter 4

Function Description

4.1 Loading the System

Function: casa

Action: initialize CASA package

Calling sequence:

`with(casa)`

Arguments:

none

Synopsis:

- Functions from the CASA package are loaded using the command `with(casa,`*function*`)` or `with(casa)`. It is not possible to use the long notation `casa[`*function*`]` since the package needs an initialization procedure to be run before performing any computation.

- The user may change the setting introduced by CASA in a procedure `‘casa/user-init‘`. If such a procedure is defined prior to calling `with(casa)` it is invoked at the very end of the initialization stage possibly overriding previous definitions. BE CAREFUL NOT TO REDEFINE ANY INTERNAL SYMBOLS OF CASA.

Example:

```
> restart:with(casa):
           |__|
           |  |          Welcome to CASA 2.3
     |   /\|  |/\
    /=\__|  []  |
   /            \_       Copyright (C) 1990-1997 by Research Institute
   |              \      for Symbolic Computation (RISC-Linz), the
   \    CASA 2.3  |      University of Linz, A-4040 Linz, Austria.
    |             |
   _|     |||     |      For help type ‘?casa’ or ‘?casa,<topic>’.
__/       |||     |_
```

See also: `with`

4.2 Creator and Selector Functions for Algebraic Sets

Function: addattr

Action: updates the attribute list of an algebraic set
Calling sequence:
`addattr(`*attr*, *a*`)`
Arguments:
attr – attribute of alg. set
a – algebraic set
Synopsis:

- The function allows to connect useful properties of algebraic sets. The attribute *attr* has to be an equation where the left hand side gives the name of the attribute and the right hand side specifies the value.
- Some CASA functions store intermediate results during computations as attributes in order to prevent multiple evaluation.

Example:

```
> mkImplAlgSet([x*y*z^2-4,x^2+y^2+z^2-1,x-y-1]);
                       2       2    2    2
 algebraic_set([x y z  - 4, x  + y  + z  - 1, x - y - 1], [x, y, z])

> addattr(dimension = 0,'');

                                   [dimension = 0]
```

See also: `Attributes`, `hasattr`, `hasattrib`, `repattr`, `mkImplAlgSet`, `mkParaAlgSet`, `mkProjAlgSet`, `mkPlacAlgSet`, `shAlgSet`

Function: Attributes

Action: list attributes
Calling sequence:
`Attributes(`*a*`)`
Arguments:
a – algebraic set
Synopsis:

- The function gives back the attribute list of an algebraic set. This list stores useful properties of an algebraic set. This list can be extended by using the function `addattr`.
- Some CASA functions store intermediate results during computations as attributes in order to prevent multiple evaluation.
- Following attributes are known to CASA:
  - *basespace* - either ‘affine’ or ‘projective’; describe the global space in which the set is lying,
  - *dimension* - dimension of the algebraic set,
  - *genus* - genus of a curve,
  - *groebnerbasis* - Groebner basis of the defining ideal,
  - *neighborhood* - ’true’ or ’false’ depending on whether the curve has neighborhood points or not (applicable only for curves),

- Important remark: If any of these attributes appear in the attribute list of an algebraic set, they will be used whenever appropriate instead of recomputing the values over and over again. Hence, if you store an incorrect value to the attribute list, it will be believed to be correct and procedures where it is requested will retrieve it. So, be careful.

Example:

```
> a1 := mkImplAlgSet([x*y*z^2-4,x^2+y^2+z^2-1,x-y-1]);

                                  2       2     2     2
a1 := algebraic_set([x y z  - 4, x  + y  + z  - 1, x - y - 1], [x, y, z])

> addattr(dimension = 0,a1);

                               [dimension = 0]

> Attributes(a1);

                               [dimension = 0]
```

See also: `addattr`, `mkImplAlgSet`, `mkParaAlgSet`, `mkProjAlgSet`, `mkPlacAlgSet`, `shAlgSet`, `represent`, `varlist`

Function: hasattr

Action: querying attributes
Calling sequence:
`hasattr`(*attr*, *a*)
Arguments:
attr – attribute
a – algebraic set
Synopsis:

- The function allows to question for properties which might be stored in the attribute list of an algebraic set. The result is a list of length 1 or 2, whose first element is either `true` or `false`. In the first case the property is stored in the attribute list of the algebraic set and the second element of the result is the value of the property. If the first element is `false` the length of the result is 1.
- If `hasattr` returns `[false]` this does not mean that the algebraic set does not have the questioned property. It only means that it is not stored in the attribute list.
- Some CASA functions store intermediate results during computations as attributes in order to prevent multiple evaluation.

Example:

```
> a1 := mkParaAlgSet([t^2-1,1/(t-1),t+1],[t]);
                                    2        1
            a1 := algebraic_set([t  - 1, -----, t + 1], [t])
                                         t - 1

> addattr(genus=0,a1);

                                 [genus = 0]
```

```
> hasattr(genus,a1);

                              [true, 0]

> hasattr(dimension,a1);

                               [false]
```

See also: `addattr`, `Attributes`, `mkImplAlgSet`, `mkParaAlgSet`, `mkProjAlgSet`, `mkPlacAlgSet`, `shAlgSet`

Function: hasattrib

Action: test for an attribute

Calling sequence:

`hasattrib(`*attr*, *a*`)`
`hasattrib(`*attr*, *a*, *name*`)`

Arguments:

attr – attribute name
a – algebraic set

Optional Arguments:

name – unevaluated name

Synopsis:

- This function is much like the `hasattr` but it returns a boolean value telling whether the given attribute is present or not.
- The optional parameter *name* must be an unevaluated name. If the algebraic set has the specified attribute and this is an attribute with a value, this value is assigned to *name*.

Example:

```
> a1 := mkImplAlgSet(x^2-y^2,[x,y]);
                                          2    2
                  a1 := algebraic_set([x  - y ], [x, y])

> genus(a1);

                                 -1

> hasattrib('genus',a1,evaln(tmp));

                                true

> tmp;

                                 -1
```

See also: `hasattr`

Function: mkImplAlgSet

Action: create an algebraic set in implicit representation

Calling sequence:

`mkImplAlgSet(`*repr*`)`
`mkImplAlgSet(`*repr*, *vars*`)`
`mkImplAlgSet(`*repr*, *vars*, *attr*`)`

Arguments:

repr – list of polynomials in *vars* with coefficients of type `casaCoeffType`.

Optional Arguments:

vars – list of variables
attr – list of initial attributes

Synopsis:

- The function works as an input interface to the datatype algebraic set. It takes as argument a list of polynomials whose common zeros define the algebraic set.
- The function creates an algebraic set in implicit representation. The generating ideal is specified by the first parameter. This ideal is considered to be in the ring with elements to be polynomials with coefficients of type `casaCoeffType` with variables *vars* (default `[x,y,z]`).
- The second, optional argument is a list of indeterminates. It is given to distinguish from parameters. If this argument is not given, all indeterminates in the polynomials in *repr* are considered not to be parameters.
- The optional third argument may be used to specify some initial attributes.
- This function returns a Maple object representing an algebraic set. It is an unevaluated function call with head `algebraic_set` and three parameters. The first one specifies the representation, the second one contains a list of variables occuring in the representation, and the third argument is a table of attributes.

Example:

```
> a1 := mkImplAlgSet([x*y*z^2-4,x^2+y^2+z^2-1,x-y-1]);
                                      2       2    2    2
a1 := algebraic_set([x y z  - 4, x  + y  + z  - 1, x - y - 1], [x, y, z])

> op(a1);

                  2       2    2    2
           [x y z  - 4, x  + y  + z  - 1, x - y - 1], [x, y, z], attr
```

See also: `casaCoeffType`, `mkParaAlgSet`, `mkProjAlgSet`, `mkPlacAlgSet`, `shAlgSet`

Function: mkParaAlgSet

Action: create an algebraic set in parametric representation

Calling sequence:

`mkParaAlgSet(`*repr*, *v*`)`
`mkParaAlgSet(`*repr*, *v*, *attr*`)`

Arguments:

repr – list of rational functions
v – variable

Optional Arguments:
attr – initial attributes

Synopsis:

- The function works as an input interface to the datatype algebraic set.
- The function creates an algebraic set in parametric representation. It takes as argument a list of rational functions which parametrize the algebraic set, whose common zeros define the algebraic set.
- The second, optional argument is a list of indeterminates. It is given to distinguish from parameters. If this argument is not given, all indeterminates in the rational functions in *repr* are considered not to be parameters.
- The optional third argument may be used to specify initial attributes of the algebraic set.

Example:

```
> a1 := mkParaAlgSet([t^2-1,1/(t-1),t+1],[t]);
                                   2        1
               a1 := algebraic_set([t  - 1, -----, t + 1], [t])
                                            t - 1
```

See also: `mkImplAlgSet`, `mkProjAlgSet`, `mkPlacAlgSet`, `shAlgSet`

Function: mkPlacAlgSet

Action: create an algebraic set represented by places

Calling sequence:
`mkPlacAlgSet`(*repr*, *v*)
`mkPlacAlgSet`(*repr*, *v*, *attr*)

Arguments:
repr – place
v – variable

Optional Arguments:
attr – initial attributes

Synopsis:

- The function works as an input interface to the datatype algebraic set. It takes as argument a list of power series or polynomials. The polynomials are also interpreted as power series and automatically converted to power series. They describe the places of the algebraic set around the point which they are centered.
- The second (optional) argument is a list of variables. It is given to distinguish from parameters. If this argument is not given, all variables in the power series in *repr* are considered not to be parameters.
- The output is an algebraic set which consists of a list of lists of procedures which compute the coefficients of the represented power series up to an arbitrary order. The first coefficient can be seen by using the function `shAlgSet`.

Example:

```
> s:=powseries[evalpow](powseries[powpoly](T,T)/powseries[powpoly]
  (1-T,T)):
> mkPlacAlgSet([[T,T^8+T^10,s],[s,T^3,1]],[T]):
  shAlgSet(",10);
The algebraic set is known to have the following properties:
None are known.
It is given by the following places:
```

```
       8       10         2    3    4    5    6    7    8    9       10
[[T, T   + O(T   ), T + T   + T   + T   + T   + T   + T   + T   + T   + O(T   )]

                2    3    4    5    6    7    8    9      10    3
    , [T + T   + T   + T   + T   + T   + T   + T   + T   + O(T   ), T , 1]]
```

See also: `powseries`, `mkImplAlgSet`, `mkParaAlgSet`, `mkProjAlgSet`, `shAlgSet`

Function: mkProjAlgSet

Action: create an algebraic set in projective representation

Calling sequence:

`mkProjAlgSet([`P, R`],`v`)`
`mkProjAlgSet([`P, R`],`$v, attr$`)`

Arguments:

P – list of polynomials
R – list of rational functions
v – list of indeterminates (not including parameters) in P

Optional Arguments:

$attr$ – initial attributes

Synopsis:

- The function works as an input interface to the datatype algebraic set. It takes as argument a list of two items. The first is an algegraic set, usually a hyperplane in a lower dimensional space, given in implicit form by a list of polynomials P. The second item R is a list of rational functions which map the algebraic set given by P into a higher dimensional space.
- The second, optional argument is a list of variables. It is given to distinguish from parameters. If this argument is not given, all indeterminates in the polynomials in P are considered not to be parameters.

Example:

```
> mkProjAlgSet([[u^2-v^3+v^2],[u+v,u-v,1/v]],[u,v]);
                                  2    3    2
    algebraic_set([[u  - v  + v ], [u + v, u - v, 1/v]], [u, v])
```

See also: `mkImplAlgSet`, `mkParaAlgSet`, `mkPlacAlgSet`, `shAlgSet`

Function: repattr

Action: representation attribute of an algebraic set

Calling sequence:

`repattr(`a`)`

Arguments:

a – algebraic set

Synopsis:

- The function gives back one of the attributes `impl`, `para`, `proj` or `plac` depending if the algebraic set is in implicit, parametric or projected representation or is represented by places.

Example:

```
> a1:=mkImplAlgSet([x*y*z^2-4,x^2+y^2+z^2-1,x-y-1]);
                                         2        2    2    2
a1 := algebraic_set([x y z  - 4, x  + y  + z  - 1, x - y - 1],

    [x, y, z])

> repattr(");

                                 impl
```

See also: `Attributes`, `mkImplAlgSet`, `mkParaAlgSet`, `mkProjAlgSet`, `mkPlacAlgSet`, `shAlgSet`

Function: represent

Action: representation of an algebraic set

Calling sequence:

`represent(`a`)`

Arguments:

a – algebraic set

Synopsis:

- The function returns the algbraic objects which represent an algebraic set, i.e.

implicit representation:	the list of polynomials,
parametric representation:	the list of rational functions,
projected representation:	[list of polynomials, list of rational functions],
representation by places:	list of list of power series.

Example:

```
> mkProjAlgSet([[u^2-v^3+v^2],[u+v,u-v,1/v]],[u,v]);
                              2    3    2
     algebraic_set([[u  - v  + v ], [u + v, u - v, 1/v]], [u, v])

> represent(");

                           2    3    2
                        [[u  - v  + v ], [u + v, u - v, 1/v]]
```

See also: `mkImplAlgSet`, `mkParaAlgSet`, `mkProjAlgSet`, `mkPlacAlgSet`, `shAlgSet`, `varlist`, `Attributes`

Function: shAlgSet

Action: information about an algebraic set

Calling sequence:

`shAlgSet(`a`)`
`shAlgSet(`a,n`)`
`shAlgSet(`a,n,m`)`
`shAlgSet(`$a,n,name$`)`

`shAlgSet(`$a, n, m, name$`)`

Arguments:

a – algebraic set

Optional Arguments:

n – positive integer

m – positive integer indicates the precision when one wants to use the floating point approach.

$name$ – unassigned name

Synopsis:

- The function works as an output interface to the datatype algebraic set of CASA. It takes as argument an algebraic set and displays information known about this algebraic set.
- The second, the third and the fourth argument are only useful if the algebraic set is represented by places. In this case the second one specifies the number of terms to be computed in the power series representation. If the user wants to use the floating point approach, the third argument must be a positive integer; in this case, it specifies the precision of the computation. If the third or fourth argument is an unevaluated name, it will be assigned the polynomial parts of the truncated power series.

Example:

```
> a1 := mkImplAlgSet([y^5-4*y^4+4*y^3+2*x^2*y^2-x*y^2+2*x^2*y+2*x*y+x^4+
  x^3],[x,y]);

a1 := algebraic_set(

        5      4      3      2  2      2      2              4    3
    [y  - 4 y  + 4 y  + 2 x  y  - x y  + 2 x  y + 2 x y + x  + x ],

    [x, y])

> shAlgSet(a1,10);

The algebraic set is known to have the following properties:
None are known.
It is given in implicit form in the xy-coordinate system by the following
equation(s)

        5      4      3      2  2      2      2              4    3
    [y  - 4 y  + 4 y  + 2 x  y  - x y  + 2 x  y + 2 x y + x  + x ]

> shAlgSet(impl2plac(a1),5,'tmp');

The algebraic set is known to have the following properties:
It is given by the following places:

              2        4      5
[[T,  - 1/2 T  + 1/8 T  + O(T )],

           2           2   27  3        4      5
    [ - 2 T , T + 1/4 T  - -- T  - 7/8 T  + O(T )]]
                           32

> shAlgSet(impl2plac(a1),5,10,'tmp');

The algebraic set is known to have the following properties:
It is given by the following places:
```

```
                          2                4      5            2
[[1. T,  - .5000000000 T  + .1250000000 T  + O(T )], [ - 2. T ,

                          2                3                4      5
    1. T + .2500000000 T  - .8437500000 T  - .8750000000 T  + O(T )]]
```

See also: `mkParaAlgSet`, `mkProjAlgSet`, `mkPlacAlgSet`, `shAlgSet`

Function: varlist

Action: variable list
Calling sequence:
`varlist(`a`)`
Arguments:
a – algebraic set
Synopsis:

- The function returns the variable list which is used to distinguish between indeterminates and parameters.

Example:

```
> mkProjAlgSet([[a*u^2-v^3+v^2],[u+v,u-b,1/v]],[u,v]);

                         2    3    2
    algebraic_set([[a u  - v  + v ], [u + v, u - b, 1/v]], [u, v])

> varlist(");

                                [u, v]
```

See also: `mkImplAlgSet`, `mkParaAlgSet`, `mkProjAlgSet`, `mkPlacAlgSet`, `shAlgSet`, `represent`, `Attributes`

4.3 Conversion Between Different Representations

Function: convertRep

Action: conversion between different representations of algebraic sets
Calling sequence:
`convertRep(`a, to`)`
`convertRep(`$a, from, to$`)`
`convertRep(`$a, from, to, arg_2, \ldots, arg_n$`)`
Arguments:
a – algebraic set
to – representation to convert to
Optional Arguments:
$from$ – representation to convert from
arg_i – further arguments to `casa/convertRep/`*from*2*to*
Synopsis:

- The function performes conversions between different representations.
- The optional parameter *from* has to be specified only if further arguments need to be given to *from2to* (see below).
- The call convertRep(a, *from*, *to*, arg_2, ..., arg_n) invokes the function *from2to* with parameters a, arg_2, ..., arg_n. An error is reported if such function is not defined.

Example:

```
> a1:=mkImplAlgSet([-15*y^2*z^3-76*y^3*z^2-z^5-3*y*z^4-15*x*y^2*z^2-
  11*x*y^3*z-x*z^4-3*x*y*z^3+y^3*x^2],[x,y,z],[basespace=projective]);
                                3  2       3  2    5        4         2  2
a1 := algebraic_set([-15 z  y  - 76 y  z  - z  - 3 y z  - 15 x y  z

             3        4          3    3  2
     - 11 x y  z - x z  - 3 x y z  + y  x ], [x, y, z])

> convertRep(a1,impl,para);

                               2
                            3 t  - 3 t + 1      t (t - 1)
       algebraic_set([- --------------, - -------------, 1], [t])
                                 3                2
                                t          -1 + 4 t  + t
```

See also: impl2para, impl2proj, impl2plac, para2proj, para2plac, para2impl, proj2impl, proj2plac, proj2para.

Function: convertSpace

Action: conversion between affine and projective space

Calling sequence:

convertSpace(a, *space*, *var*)

Arguments:

a – algebraic set in implicite representation
space – affine or projective
var – variable

Synopsis:

- The function performs conversion of algebraic sets in implicit representation between the affine and projective spaces.
- The second parameter *space* specifies the target space.
- If the target space is affine the algebraic set is assumed to be in a projective space. The generating polynomials are then dehomogenized with respect to the third argument which has to be a name contained in varlist(a).
- If the target space is projective the algebraic set is assumed to be in an affine space. The generating polynomials are then homogenized with respect to the third argument. If some polynomials are already homogeneous an error is reported.

Example:

```
> a:=mkImplAlgSet([x^2-2*y^3+4*x*y^2],[x,y]);
                                        2      3      2
```

```
          a := algebraic_set([x  - 2 y  + 4 y  x], [x, y])

> convertSpace(a,projective,z);

                               2      3      2
          algebraic_set([z x  - 2 y  + 4 y  x], [x, y, z])

> Attributes(");

                      [basespace = projective]

> convertSpace("",affine,z);

                               2      3      2
            algebraic_set([x  - 2 y  + 4 y  x], [x, y])

> Attributes(");

                        [basespace = affine]
```

See also: `varlist`

4.3.1 Parametrization

Function: **impl2para**

Action: conversion from implicit to parametric representation

Calling sequence:

`impl2para(`a`)`

`impl2para(`$a, optpar$`)`

Arguments:

a – algebraic set

Optional Arguments:

optpar – sequence of optional parameters.

Synopsis:

- The function converts an algebraic set in implicit representation into parametric representation. Currently, it may be applied only to one-dimensional algebraic sets (i.e. both plane and space algebraic curves).
- This action may also be achieved by calling the function `convertRep`.
- As a first step the algebraic set is projected onto a union of hypersurfaces (CASA function `impl2proj`). If this results in a single irreducible plane curve of genus 0, it is parametrized by the CASA function `proj2para`. The result of the parametrization is mapped back into space by means of the birational correspondence.
- The sequence of optional arguments will possibly be passed to procedures called within `impl2para`. It provides additional information to these procedures. All optional arguments should have the form *keyword* = *value*.
- The optional argument *parameter* = *name* specifies that the *name* will be used as the parameter in the parametric representation.
- The optional argument *optimal* = *true* specifies that the user wants to have an optimal parametrization, i.e. a parametrization over the original ground field if possible, or otherwise over an algebraic extension field of degree 2.

- The optional argument *check* = *true* specifies that the user wants to check the result (i.e. substituting back into the implicit equation).
- The optional argument *points* = [*point1, point2, ...*] specifies that the user wants to supply some known simple points to the function. The function will check if those points are simple points on the curve. Moreover, the points which are not on the curve will be automatically removed. However, the user should know that supplying simple points may not be a good idea for non-expert users since it may lead to a non-optimal parametrization.

Example:

```
A rational algebraic curve of odd degree can always be parametrized
over the base field of coefficients.
> a1:=mkImplAlgSet([y^2*z^3-x^5],[x,y,z],[basespace=projective]);
                                 2  3    5
             a1 := algebraic_set([y  z  - x ], [x, y, z])

> impl2para(a1);

                                 2    5
                   algebraic_set([t , -t , 1], [t])

A rational algebraic curve of even degree may be parametrized over a
near optimal extension field of coefficients (i.e. extension of degree
at most two).
> a2:=mkImplAlgSet([y^4-10*x*y^3+35*x^2*y^2-50*x^3*y+24*x^4+x^3],[x, y]);
                      4      3         2  2         3      4    3
a2 := algebraic_set([y  - 10 y  x + 35 y  x  - 50 y x  + 24 x  + x ],

    [x, y])

> impl2para(a2);

                                                   2         4   /        2
algebraic_set([- 1/2 (37 + 36 RootOf(_Z  + 1)) t    /  (1260 t
                                                                 /

                      2                3       4
     - 730 RootOf(_Z  + 1) t + 1825 t  + 444 t

           4          2                                2
     + 432 t  RootOf(_Z  + 1) - 74 - 10 t - 72 RootOf(_Z  + 1)

          3          2                     2       2
     - 25 t  RootOf(_Z  + 1) - 1295 RootOf(_Z  + 1) t ), - 1/2

                      2        3   /        2               2
    (-73 + RootOf(_Z  + 1)) t    /  (1260 t  - 730 RootOf(_Z  + 1) t
                                /

           3        4        4          2
     + 1825 t  + 444 t  + 432 t  RootOf(_Z  + 1) - 74 - 10 t

or over the optimal extension field of coefficients if the option
'optimal=true' is specified.
> impl2para(a2,optimal=true);
```

```
                            1                      -2 + t
algebraic_set([- ----------------------, ----------------------], [t])
                             2    3                   2    3
                 t (-t + 2 t  + t  - 2)  t (-t + 2 t  + t  - 2)

A space curve can also be parametrized
> a3:=mkImplAlgSet([x^2-y^3,z+y+x]);
                             2    3
            algebraic_set([x  - y , z + y + x], [x, y, z])

> impl2para(",parameter=c);

algebraic_set(

                 2    3              2                      2
     1 + 3 c + 3 c  + c    1 + 2 c + c        c (1 + 2 c + c )
    [--------------------, ------------, -2 --------------------],
                  2    3   2                              2    3
     -1 + 3 c - 3 c  + c   c  - 2 c + 1     -1 + 3 c - 3 c  + c

    [c])
```

See also: `convertRep`, `mkImplAlgSet`, `impl2proj`, `para2impl`, `para2proj`, `proj2para`, `impl2plac`, `para2plac`, `proj2impl`, `proj2plac`

Function: proj2para

Action: conversion from projective to parametric representation

Calling sequence:

`proj2para(`a`)`

`proj2para(`$a, vars$`)`

Arguments:

a – algebraic set in projected representation

Optional Arguments:

$vars$ – variable

Synopsis:

- The function converts the projective representation of an algebraic set into the parametric one.
- This action may be achieved also by calling the function `convertRep`.
- If the algebraic set in projected form consists of a planar curve and a birational mapping the planar curve is parametrized and the result is mapped back by the birational correspondence. First the singularity structure of the curve is determined. For planar curves of genus zero a parametrization is computed by forcing a pencil of curves through the determined singularities and sufficiently many simple points of the curve.
- The optional argument specifies a variable to be used as parameter in the parametric representation.

Example:

```
> a1 := mkProjAlgSet([[u^2-v^3+v^2],[u+v,u-v,1/v]],[u,v]);
                             2    3    2
    a1 := algebraic_set([[u  - v  + v ], [u + v, u - v, 1/v]], [u, v])
```

```
> proj2para(a1);
                           2        3         2        3        2
                        - t  - 1 + t  + t    t  + 1 + t  + t    t
    algebraic_set([-----------------, - ---------------, ------], [t])
                                3                  3          2
                               t                  t          t  + 1
```

See also: `convertRep`, `mkProjAlgSet`, `impl2para`, `impl2proj`, `para2impl`, `para2proj`, `impl2plac`, `para2plac`, `proj2impl`, `proj2plac`

4.3.2 Implicitization

Function: **para2impl**

Action: conversion from parametric to implicit representation

Calling sequence:

`para2impl(`a`)`

`para2impl(`$a, vars$`)`

Arguments:

a – algebraic set in parametric representation

Optional Arguments:

$vars$ – list of variables

Synopsis:

- The function converts the parametric representation of an algebraic set into the implicit one.
- This action may be achieved also by calling the function `convertRep`.
- The computation of an implicit representations is done by a Gröbner basis calculation with respect to the purely lexicographic term ordering. With the help of the elimination property of Gröbner bases the implicit equations can be found.
- The optional argument specifies a list of variables to be used in the implicit representation.

Example:

```
> a1 := mkParaAlgSet([t^2-1,1/(t-1),t+1],[t]);
                                      2        1
              a1 := algebraic_set([t  - 1, -----, t + 1], [t])
                                             t - 1
> para2impl(a1);
                                                  2
          algebraic_set([- x y + z, - 2 y - 1 + x y ], [x, y, z])
```

See also: `convertRep`, `mkParaAlgSet`, `impl2para`, `impl2proj`, `para2proj`, `proj2para`, `impl2plac`, `para2plac`, `proj2impl`, `proj2plac`

Function: **proj2impl**

Action: conversion from projective to implicit representation

Calling sequence:

`proj2impl(`a`)`
`proj2impl(`$a, vars$`)`

Arguments:

a – algebraic set in projective representation

Optional Arguments:

$vars$ – list of variables

Synopsis:

- The function converts the projective representation of an algebraic set to the implicit one.
- This action may be achieved also by calling the function `convertRep`.
- The conversion from projected to implicit form uses a similar strategy to the strategy used in the implicitization algorithms `para2impl`. Again a Groebner basis with respect to the lexicographical term ordering is computed.
- The optional argument specifies variables to be used in implicit representation.

Example:

```
> a1 := mkProjAlgSet([[u^2-v^3+v^2],[u+v,u-v,1/v]],[u,v]);
                                    2    3    2
    a1 := algebraic_set([[u  - v  + v ], [u + v, u - v, 1/v]], [u, v])

> proj2impl(a1);
  algebraic_set(

                                     2  2                       2    3  2
   [- 4 - 4 y z - y + x - 2 z  y , - 1 + 2 z + 2 y z  + z  y ], [x, y, z])
```

See also: `convertRep`, `mkProjAlgSet`, `impl2para`, `impl2proj`, `para2impl`, `para2proj`, `proj2para`, `impl2plac`, `para2plac`, `proj2plac`

4.3.3 Projection

Function: impl2proj

Action: conversion from implicit to projective representation

Calling sequence:

`impl2proj(`a`)`
`impl2proj(`$a, vars$`)`

Arguments:

a – algebraic set in implicit representation

Optional Arguments:

$vars$ – list of variables

Synopsis:

- The function converts an algebraic set in implicit to projective representation.
- This action may be achieved also by calling the function `convertRep`.
- First the ideal generated by the given basis and whose zeros are the given algebraic set is split at least into an intersection of unmixed-dimensional ideals. For each of the ideals a set of independent variables is computed along the splitting (that is why it is sometimes necessary to split unmixed-dimensional ideals, compare Gianni, Trager, Zacharias, JSC 1988, p 149-167). For each of the unmixed-dimensional ideals the radical is computed and then the radical is put in general position yielding the desired result.
- The optional argument $vars$ is a list of variables which are to be used for representing the set in projective form.

Example:

```
> a1 := mkImplAlgSet([x*y*z^2-4,x^2+y^2+z^2-1,x-y-1]);
                                2       2    2    2
a1 := algebraic_set([x y z  - 4, x  + y  + z  - 1, x - y - 1],

    [x, y, z])

> impl2proj(a1);
                                        2        3        4        5
algebraic_set([[268 + 192 x - 100 x  - 192 x  + 293 x  - 180 x

           6       7    8    11      479   657  2   275  3   15   4
     + 62 x  - 12 x  + x ], [--- x + --- - --- x  + --- x  + --- x
                              289     289   578      578      289

       67   5   21   6           7  190   11      657  2   275  3
     - --- x  + --- x  - 1/289 x , --- + --- x - --- x  + --- x
       578      578                 289   289     578      578

       15   4   67   5   21   6           7    300     99    657  2
     + --- x  - --- x  + --- x  - 1/289 x , - --- x + --- + --- x
       289      578      578                  289     289   578

       275  3   15   4   67   5   21   6          7
     - --- x  - --- x  + --- x  - --- x  + 1/289 x ]], [x])
       578      289      578      578

> a2 := mkImplAlgSet([z^2 + x^2 + y^2 - 1, x*z, -z +z^3 + z*y^2],
  [x,y,z]);
a2 :=

                    2    2    2                 3      2
    algebraic_set([x  + y  + z  - 1, x z, -z + z  + z y ], [x, y, z])

> impl2proj(a2);

                  2    2
algebraic_set([[y  + x  - 1], [x, y, 0]], [y, x]),

                      2    2
    algebraic_set([[x  + z  - 1], [0, -x, z]], [x, z])
```

See also: `convertRep`, `mkImplAlgSet`, `impl2para`, `para2impl`, `para2proj`, `proj2para`, `impl2plac`, `para2plac`, `proj2impl`, `proj2plac`

Function: para2proj

Action: conversion from parametric to projective representation

Calling sequence:

`para2proj(`a`)`

`para2proj(`$a, vars$`)`

Arguments:

a – algebraic set in parametric representation

Optional Arguments:

vars – list of varibles

Synopsis:

- The function converts the parametric representation of an algebraic set to the projective one.
- This action may be achieved also by calling the function `convertRep`.
- The transformation from parametric to projected form is based on the fact that the parametric representation is a special case of the projected representation. An algebraic set in parametric form can also be seen as being in projected form where the hypersurface is a hyperplane - It is the hyperplane defined by $x = 0$ where x is a variable not occuring in the rational functions that constitute the parametrization (and the birational mapping).
- This conversion algorithms can also be implemented as a composition of two other conversion algorithms: `para2proj(`a`)` = `impl2proj(para2impl(`a`))`.
- The optional argument specifies a list of variables to be used in projective representation.

Example:

```
> a1 := mkParaAlgSet([t^2-1,1/(t-1),t+1],[t]);
                                     2        1
            a1 := algebraic_set([t   - 1, -----, t + 1], [t])
                                          t - 1

> a2:=para2proj(a1,[u,v]);

                                      2        1
       a2 := algebraic_set([[v], [u   - 1, -----, u + 1]], [u, v])
                                           u - 1

> a3:=impl2proj(para2impl(a1));

                                  2                1
     a3 := algebraic_set([[x - z   + 2 z], [x, ------, z]], [x, z])
                                               -2 + z

> implEqual(proj2impl(a2),proj2impl(a3));

                                   true
```

See also: `convertRep`, `mkParaAlgSet`, `impl2para`, `impl2proj`, `para2impl`, `proj2para`, `impl2plac`, `para2plac`, `proj2impl`, `proj2plac`

4.3.4 Puiseux Series Expansion or Conversion to Representation by Places

Function: **impl2plac**

Action: conversion from implicit representation to a representation by places

Calling sequence:

`impl2plac(`a`)`

`impl2plac(`$a, point, v$`)`

Arguments:

a – algebraic set

Optional Arguments:

point – affine point
v – variable name

Synopsis:

- The function converts implicit representation to a representation by places of a given algebraic set. The places are series around *point* in a variable *v*.
- This action may be achieved also by calling the function `convertRep`.
- As a first step the algebraic set is projected onto a union of hypersurfaces (CASA function `impl2proj`). If this union consists of a single set and this hypersurfaces is a planar curve, a power series expansion of the branches of the curve is computed by the CASA function `proj2plac`. The result of the series expansion is mapped back into space by means of the birational correspondence.
- Optional arguments are a list of field elements which represents the point around which the expansion should be done, (default is [0,0,0]), and a variable which is used for representation of the places.

Example:

```
> a1 := mkImplAlgSet([x^2-y^3+y^2,z]);
                                  2    3    2
          a1 := algebraic_set([x  - y  + y , z], [x, y, z])

>  shAlgSet(impl2plac(a1),5);
The algebraic set is known to have the following properties:
It is given by the following places:
                                    2        3    4      5
                [[- I T, T + 1/2 T  + 5/8 T  + T  + O(T ), 0],

                                     2        3    4      5
                    [I T, T + 1/2 T  + 5/8 T  + T  + O(T ), 0]]
```

See also: `convertRep`, `mkImplAlgSet`, `impl2para`, `impl2proj`, `para2impl`, `para2proj`, `proj2para`, `para2plac`, `proj2impl`, `proj2plac`

Function: para2plac

Action: conversion from parametric representation to the representation by places

Calling sequence:

`para2plac(`*a*`)`
`para2impl(`*a*`,`*point*`)`
`para2impl(`*a*`,`*point*`,`*vars*`)`

Arguments:

a – algebraic

Optional Arguments:

point – point
vars – list of variables

Synopsis:

- The function converts the parametric representation of an algebraic set to the representation by places.
- This action may be achieved also by calling the function `convertRep`.
- Numerator and denomiator of the rational functions are viewed as power series and are divided out. Parameter substitutions are performed in order to compute the series around a given center.
- The optional argument *point* specifies a point around which series expansion is to be done (default value is the point corresponding to the parameter value 0). The optional argument *vars* specifies variables to be used in the representation by places.

Example:

```
> a1 := mkParaAlgSet([t^2-1,1/(t-1),t+1],[t]);
                                 2       1
             a1 := algebraic_set([t  - 1, -----, t + 1], [t])
                                        t - 1

> shAlgSet(para2plac(a1,[-1,-1,1]),5);
The algebraic set is known to have the following properties:
None are known.
It is given by the following places
              2      5             2    3    4      5              5
      [[- 1 + T  + O(T ), - 1 - T - T  - T  - T  + O(T ), 1 + T + O(T )]]
```

See also: `convertRep`, `mkParaAlgSet`, `impl2para`, `impl2proj`, `para2impl`, `para2proj`, `proj2para`, `impl2plac`, `proj2impl`, `proj2plac`

Function: proj2plac

Action: convertion from projective representation to the representation by places

Calling sequence:

`proj2plac(`*a*`)`
`proj2plac(`*a*`,`*point*`)`
`proj2plac(`*a*`,`*point*`,`*var*`)`

Arguments:

a – algebraic set in projective representation

Optional Arguments:

point – point
var – variable

Synopsis:

- The function converts the projective representation of an algebraic set to the representation by places.
- This action may be achieved also by calling the function `convertRep`.
- If the algebraic set in projected form consists of a planar curve and a birational mapping, the branches of the planar curve around the specified point (see optional arguments) are expanded into power series and the result is mapped back by the birational correspondence. The series expansion is computed by a rational Newton polygon algorithm.
- The optional argument *point* specifies a point around which series expansion is to be done (default value is [0,0]). The optional argument *var* specifies a variable to be used in the representation by places.
- IMPORTANT NOTE: Make sure that the point is in the algebraic set by using CASA function `pointInAlgSet`. The algorithm will not check it for you.

Example:

```
> a1 := mkProjAlgSet([[u-v^3+v^2],[u+v,u-v,1/(v+1)]],[u,v]);
                              3    2                    1
 a1 := algebraic_set([[u - v  + v ], [u + v, u - v, -----]], [u, v])
                                                    v + 1

> shAlgSet(proj2plac(a1,[0,0]),5);
The algebraic set is known to have the following properties:
It is given by the following places:
```

```
            2        3    4      5
[[T - 1/2 T  + 5/8 T  + T  + O(T ),

                 2        3    4      5
     - T - 3/2 T  - 5/8 T  - T  + O(T ),

                  2        3      5
    1 - T + 1/2 T  - 5/8 T  + O(T )]]
```

See also: convertRep, mkProjAlgSet, impl2para, impl2proj, para2impl, para2proj, proj2para, impl2plac, para2plac, proj2impl

4.4 Boolean Operations on Algebraic Sets in Implicit Form

Function: implEmpty

Action: testing whether an algebraic set is empty
Calling sequence:
implEmpty(a)
Arguments:
a – algebraic set in implicit representation
Synopsis:

- The function tests if an algebraic set in implicit form is empty.
- It is tested if 1 is in the Groebner basis for the ideal of a.

Example:

```
> a1 := mkImplAlgSet([x^3+x^2*y-x,y^2+z^2-1,x,z],[x,y,z]);
                          3    2         2    2
  a1 := algebraic_set([x  + x  y - x, y  + z  - 1, x, z], [x, y, z])

> implEmpty(a1);

                              false

> a2 := mkImplAlgSet([x^2-1,x^2-3],[x,y,z]);

                                2       2
           a2 := algebraic_set([x  - 1, x  - 3], [x, y, z])

> implEmpty(a2);

                               true
```

See also: implUnion, implUnionLCM, implIntersect, implSubSet, implEqual, implIdealQuo

Function: implEqual

Action: testing equality of two algebraic sets in implicit representation
Calling sequence:
`implEqual(`a, b`)`
Arguments:
a, b – algebraic sets in implicit representation
Synopsis:

- The function `implEqual` tests if two algebraic sets in implicit form are equal.
- There is `implEqual(`a, b`)` if and only if (`implSubSet(`a, b`)` and `implSubSet(`b, a`)`).

Example:

```
> a1 := mkImplAlgSet([x^3+x^2*y-x,y^2+z^2-1,x,z],[x,y,z]);
                              3    2          2    2
  a1 := algebraic_set([x  + x  y - x, y  + z  - 1, x, z], [x, y, z])

> a2 := tsolve(a1);

                                        2
          a2 := algebraic_set([z, y  - 1, x], [x, y, z])

> implEqual(a1,a2);

                                    true
```

See also: `implUnion`, `implUnionLCM`, `implIntersect`, `implSubSet`, `implEmpty`, `implIdealQuo`

Function: implIdealQuo

Action: compute the Zariski closure of the difference of two algebraic sets
Calling sequence:
`implIdealQuo(`a, b`)`
Arguments:
a, b – algebraic sets in implicit representation
Synopsis:

- The function computes the algebraic set of the ideal quotient of the ideals of the given algebraic sets.
- The computation is reduced to the computation of the intersection of ideals. Let $\{q_1, \ldots, q_n\}$ be a basis for $I(b)$, $I(b)$ denotes the ideal of b. It holds that

  $$I(a)/I(b) = \bigcup I(a)/ < q_i >,\ i = 1, \ldots, n,$$

  where $< q_i >$ is the principle ideal generated by q_i. Each $I(a)/ < qi >$ can be computed again by ideal intersection. Let $\{p_1, \ldots, p_m\}$ be a basis for $I(a) \cap < qi >$ then $\{p_1/q_i, \ldots, p_m/q_i\}$ is a basis for $I(a)/ < q_i >$.

Example:

```
> a1 := mkImplAlgSet([x^3+x^2*y-x,z],[x,y,z]);
                                      3    2
          a1 := algebraic_set([x  + x  y - x, z], [x, y, z])

> a2 := mkImplAlgSet([x,y^2+z^2-1],[x,y,z]);
```

```
                                         2    2
              a2 := algebraic_set([x, y  + z  - 1], [x, y, z])

> implIdealQuo(a1,a2);

                                  3    2
               algebraic_set([x  + x  y - x, z], [x, y, z])

> implIdealQuo(a1,a1);

                      algebraic_set([1], [x, y, z])
```

See also: `implUnion`, `implUnionLCM`, `implIntersect`, `implEqual`, `implEmpty`, `implSubSet`

Function: implIntersect

Action: intersection of algebraic sets

Calling sequence:

`implIntersect(`$a1, a2, \ldots, an$`)`

Arguments:

$a1$,$a2$,...,an – algebraic sets in implicit representation

Synopsis:

- The function computes the intersection of algebraic sets in implicit form by computing the union of the corresponding ideals. These ideals are given by a finite basis. A basis of the union is obtained by putting all basis elements into one new basis.

Example:

```
> a1 := mkImplAlgSet([x^3+x^2*y-x,z],[x,y,z]);
                                      3    2
               a1 := algebraic_set([x  + x  y - x, z], [x, y, z])

> a2 := mkImplAlgSet([x,y^2+z^2-1],[x,y,z]);
                                       2    2
                a2 := algebraic_set([y  + z  - 1, x], [x, y, z])

> implIntersect(a1,a2);
                              3    2          2    2
          algebraic_set([x  + x  y - x, y  + z  - 1, x, z], [x, y, z])

# To simplify the (zero-dimensional) result we can apply tsolve.
> tsolve(");
                                                 2
                    algebraic_set([x, z, y  - 1], [x, y, z])
```

See also: `implUnion`, `implUnionLCM`, `implSubSet`, `implEqual`, `implEmpty`, `implIdealQuo`

Function: implSubset

Action: testing for subsets
Calling sequence:
`implSubSet(`a, b`)`
Arguments:
a, b – algebraic sets in implicit representation
Synopsis:

- The function tests if an algebraic set is a subset of another algebraic set in implicit form.
- An algebraic set a is a subset of b if and only if rad(I(b)) is a subset of rad(I(a)), where I(x) denotes the ideal of the algebraic set x and rad(I) denotes the radical ideal of I.

Example:

```
> a1 := mkImplAlgSet([x^3+x^2*y-x,z],[x,y,z]);
                                    3    2
            a1 := algebraic_set([x  + x  y - x, z], [x, y, z])

> a2 := mkImplAlgSet([x,y^2+z^2-1],[x,y,z]);
                                     2    2
             a2 := algebraic_set([y  + z  - 1, x], [x, y, z])

> a3 := implIntersect(a1,a2);
                          3    2          2    2
   a3 := algebraic_set([x  + x  y - x, y  + z  - 1, x, z], [x, y, z])

> implSubSet(a3,a1);
                                        true

> implSubSet(a2,a3);
                                       false
```

See also: `implUnion`, `implUnionLCM`, `implIntersect`, `implEqual`, `implEmpty`, `implIdealQuo`

Function: implUnionLCM

Action: union of algebraic sets
Calling sequence:
`implUnionLCM(`$a1, a2, \ldots, an$`)`
Arguments:
$a1, a2, ..., an$ – algebraic sets in implicit representation
Synopsis:

- The function computes the union of algebraic sets in implicit form by computing the intersection of the corresponding ideals.
- For the case of two algebraic sets $a1$ and $a2$: Let the ideal of $a1$ be I_1, a subset of $K[x_1, ..., x_n]$ and let the ideal of $a2$ be I_2. The intersection is computed by the ideal contraction

$$I_3 := < tI_1 \cup (t-1)I_2 > \cap K[x_1, ..., x_n],$$

where t is a new variable and $< B >$ denotes the ideal generated by the set B. A basis for the ideal I_3 can be obtained by a Gröbner basis calculation. Furthermore, $I_3 = I_1 \cap I_2$.

- If the algebraic sets have common components then the multiplicity of these components is not increased in the result. (compare `implUnion`).

Example:

```
> a1 := mkImplAlgSet([x^3+x^2*y-x,z],[x,y,z]);
                                        3    2
                a1 := algebraic_set([x  + x  y - x, z], [x, y, z])

> a2 := mkImplAlgSet([x,y^2+z^2-1],[x,y,z]);
                                         2    2
                 a2 := algebraic_set([y  + z  - 1, x], [x, y, z])

> implUnionLCM(a1,a2);
                              3    2                  2    3
          algebraic_set([x  + x  y - x, z x, z y  + z  - z], [x, y, z])
```

See also: `implUnion`, `implSubSet`, `implEqual`, `implEmpty`, `implIntersect`, `implIdealQuo`

Function: implUnion

Action: union of algebraic sets

Calling sequence:

`implUnion(`*a1*, *a2*, . . . , *an*`)`

Arguments:

a1,*a2*,...,*an* – algebraic sets in implicit representation

Synopsis:

- The function computes the union of algebraic sets in implicit form by computing the product of the corresponding ideals. These ideals are given by a finite basis. A basis of the product is obtained by forming all products of basis elements.
- For the case of two algebraic sets *a1* and *a2*: Let the ideal of a1 have the basis {*p1*,...,*pn*} and let the ideal of *a2* have the basis {*q1*,...,*qm*} then the product has the basis {*p1***q1*, *p2***q1*, ..., *pn***q1*, ..., *p1***qm*, *p2***qm*, ..., *pn***qm*}.
- If the algebraic sets have common components then the result contains these components with higher multiplicity (compare `implUnionLCM`).

Example:

```
> a1 := mkImplAlgSet([x^3+x^2*y-x,z],[x,y,z]);
                                        3    2
                a1 := algebraic_set([x  + x  y - x, z], [x, y, z])

> a2 := mkImplAlgSet([x,y^2+z^2-1],[x,y,z]);
                                         2    2
                 a2 := algebraic_set([y  + z  - 1, x], [x, y, z])

> implUnion(a1,a2);
                         4      3    2
```

```
algebraic_set([x  + y x  - x ,

      2     2          3  2      2  2      2    2  3    3  2    3
    -y  x - z  x + x + y  x  + y x  z  - y x  + y  x  + x  z  - x ,

                 3      2
    x z, -z + z   + z y ], [x, y, z])
```

See also: `implUnionLCM`, `implSubSet`, `implEqual`, `implEmpty`, `implIntersect`, `implIdealQuo`

4.5 Further Operations on Algebraic Sets

Function: computeRadical

Action: computation of a radical

Calling sequence:

`computeRadical(`a`)`

Arguments:

a – algebraic set in implicit representation

Synopsis:

- The function splits the ideal of an algebraic set in implicit representation into the intersection of radical ideals. The ideal of an algebraic set is the set of ALL polynomials vanishing on the algebraic set.
- First the ideal generated by the given basis and whose zeros are the given algebraic set is split at least into an intersection of unmixed-dimensional ideals. For each of the ideals a set of independent variables is computed along the splitting (that is why it is sometimes necessary to split unmixed-dimensional ideals, compare Gianni, Trager, Zacharias, JSC 1988, p 149-167). For each of the unmixed-dimensional ideals the radical is computed by reducing the problem to the zero-dimensional case.

Example:

```
> a1:=mkImplAlgSet([z*x^2+z*y^2-z,x^4+2*x^2*y^2-2*x^2+y^4-2*y^2+1],
  [x,y,z]);
a1 := algebraic_set(

        2      2       4      2  2      2    4      2
    [z x  + z y  - z, x  + 2 y  x  - 2 x  + y  - 2 y  + 1], [x, y, z])

> computeRadical(a1);

                        2    2
algebraic_set([-1 + y  + x , z], [x, y, z]),

                            2    2
    algebraic_set([-1 + y  + x ], [x, y, z])
```

See also: `mkImplAlgSet`, `impl2proj`

Function: decompose

Action: decomposition of an algebraic set into irreducible components

Calling sequence:

`decompose(`*a*`)`

`decompose(`*a*`,absolute)`

Arguments:

a – algebraic set in implicit or projective representation.

Optional Arguments:

`absolute` – string absolute.

Synopsis:

- The function computes the decomposition of an algebraic set into its irreducible components.
- On default the decomposition is done over the rational numbers. The optional argument `absolute` only works for algebraic sets in projected form and performs the decomposition over the algebraic closure.

Example:

```
The algebraic set a1 is the intersection of two cylinders, symmetric
with respect to the z and the y axis, respectively, and with the same
radius. It will be decomposed into two circles.
> a1 := mkImplAlgSet([x^2+y^2-4,x^2+z^2-4],[x,y,z]);
                                   2    2       2    2
      a1 := algebraic_set([x  + y  - 4, x  + z  - 4], [x, y, z])

> decompose(a1);

                2    2
algebraic_set([x  + y  - 4, -z + y], [x, y, z]),

                    2    2
    algebraic_set([x  + y  - 4, z + y], [x, y, z])

> a2 := mkProjAlgSet([[u^2-2*v^2],[u,v,u-v]],[u,v]);

                                    2      2
      a2 := algebraic_set([[u  - 2 v ], [u, v, u - v]], [u, v])

> decompose(a2,absolute);

                                        2
algebraic_set([[v - 1/2 RootOf(_Z  - 2) u], [u, v, u - v]], [u, v]),

                                            2
    algebraic_set([[v + 1/2 RootOf(_Z  - 2) u], [u, v, u - v]], [u, v])
```

See also: `mkImplAlgSet`, `mkParaAlgSet`, `mkProjAlgSet`, `mkPlacAlgSet`

Function: dimension

Action: dimension of an algebraic set

Calling sequence:

`dimension(`*a*`)`

Arguments:

a – algebraic set

Synopsis:

- The function computes the dimension of an algebraic set.
- For algebraic sets in projected representation. It returns the dimension of the implicit algebraic set in the lower dimensional subspace.

Example:

```
> a1 := mkImplAlgSet([x^2-3*x*y^2-7*y^3, x^2-z^2-1]);
                              2      2     3   2    2
  a1 := algebraic_set([x  - 3 x y  - 7 y , x  - z  - 1], [x, y, z])

> dimension(a1);

                                   1
```

See also: `independentVariables`, `mkImplAlgSet`, `mkParaAlgSet`, `mkProjAlgSet`, `mkPlacAlgSet`

Function: independentVariables

Action: computing a set of independent variables
Calling sequence:
`independentVariables(`a`)`
Arguments:
a – algebraic set in implicit representation
Synopsis:

- The function computes a maximal set of independent variables for an algebraic set a in implicit form. The dimension of the algebraic set a is the number of elements of `independentVariables(`a`)`.

Example:

```
> a1 := mkImplAlgSet([x^2-3*x*y^2-7*y^3, x^2-z^2-1]);
                              2      2     3   2    2
  a1 := algebraic_set([x  - 3 x y  - 7 y , x  - z  - 1], [x, y, z])

> independentVariables(a1);

                                  {z}
```

See also: `dimension`, `mkImplAlgSet`, `mkParaAlgSet`, `mkProjAlgSet`, `mkPlacAlgSet`

Function: singLocus

Action: singularities of an algebraic set
Calling sequence:
`singLocus(`a`)`
Arguments:
a – algebraic set
Synopsis:

- The function computes the algebraic set of all singularities of a given algebraic set.
- This can only be done for algebraic sets not given by places.
- For algebraic sets in implicit form the partial derivatives with respect to the variables in `varlist(`a`)` are computed.
- Algebraic sets in parametric or in projected form are first converted to implicit form. The function returns an algebraic set in implicit form, the variables of this algebraic set can be specified as an optional argument.

Example:

```
> a1 := mkImplAlgSet([x^2+x^3-y^2],[x,y]);
                                     2    3    2
             a1 := algebraic_set([x   + x  - y ], [x, y])

> singLocus(a1);

                                2          2    3    2
       algebraic_set([2 x + 3 x , -2 y, x   + x  - y ], [x, y])

> tsolve(");

                    algebraic_set([y, -2 x], [x, y])
```

See also: `singularities`, `tsolve`

Function: tangSpace

Action: tangent space and tangent cone

Calling sequence:

`tangSpace(`a`,` *point*`)`
`tangSpace(`a`,` *point*`, cone)`

Arguments:

a – algebraic set in implicit representation
point – point in a

Optional Arguments:

`cone` – string `cone`

Synopsis:

- The call `tangSpace(`a`,` *point*`)` computes an algebraic set defining the tangent space of variety a at point *point*. The tangent space is defined to be a set of points on all lines intersecting a at *point* with multiplicity bigger than 1.
- If the third optional parameter is the string `cone`, then the tangent cone (set of lines generating the tangent space) is returned.

Example:

```
> a1:=mkImplAlgSet([-15*y^2*z^3-76*y^3*z^2-z^5-3*y*z^4-15*x*y^2*z^2-11*
  x*y^3*z-x*z^4-3*x*y*z^3+y^3*x^2],[x,y,z],[basespace=projective]);
                            2  3       3  2    5       4         2  2
a1 := algebraic_set([-15 y  z  - 76 y  z  - z  - 3 y z  - 15 x y  z

             3        4          3    2  3
```

```
    - 11 x y  z - x z  - 3 x y z  + x  y ], [x, y, z])

> tangSpace(a1,[0,1,0]);

                   algebraic_set([0], [x, y, z])

> tangSpace(a1,[0,1,0],'cone');
Some of the generating polynomials for the affine tangent space/cone
are homogeneous. Returning only affine form of generators.

                                2       2
            algebraic_set([x  - 76 z  - 11 x z], [x, z])
```

Function: tsolve

Action: triangularized representation

Calling sequence:

`tsolve(`*a*`)`

Arguments:

a – algebraic set in implicit representation

Synopsis:

- The function splits an algebraic set in implicit form into the union of several algebraic sets which have a triangulized representation.
- The order of the variables in the variable list of the algebraic set influences the result. In the representation of every output algebraic set there are polynomials in just the last variable and bivariate polynomials in the last two variables.

Example:

```
> a1:=mkImplAlgSet([1-x+x*y^2+x*z^2,1-y+y*x^2+y*z^2,1-z+z*x^2+z*y^2]);

a1 := algebraic_set(

                 2       2            2        2              2      2
    [1 - x + x y  + x z , 1 - y + x  y + y z , 1 - z + z x  + z y ],

    [x, y, z])

> tsolve(a1,[x,y,z]);

                                2       3    4      5      7      2
algebraic_set([-1 - 2 z + 2 z  + 4 z  - z  - 5 z  + 2 z , -3 z

          4      2  2      4      6        4  2      6      2
     + 3 y z  + 3 y  z  + 7 z  - 2 z  y - 4 z  y  - 4 z  + z y

          2        3       5           3            2      2
     - 6 y z  - 2 y z  + 2 y z  + y - z + z , 1 - x + x y  + x z ],

    [x, y, z]), algebraic_set([

      7      6      5      3    2                5      5        4
```

```
4 z  - 4 z  - 4 z  - 3 z  + z  - z - 1, 16 y z  - 2 z  + 16 y z

      4        3      3      2                 6
 - 6 z  + 4 y z  + 9 z  + 2 z  + 3 z + 2 + 8 z ,

           2      2
1 - x + x y  + x z ], [x, y, z])
```

4.6 Plane Algebraic Curves

4.6.1 Analysis of Singularities

Function: singularities

Action: singularities of a specified plane curve
Calling sequence:
`singularities(`a`)`
Arguments:
a – algebraic set in implicit representation specifying an irreducible plane curve
Synopsis:

- The function yields a list of singularities of a plane curve a. Singularities are represented by a list where each element is a list with two elements: The first element is a list representing a projective point and the second one an integer specifying the multiplicity.
- This procedure does not compute neighboring singularities.
- Additional information on the trace of computation is provided by the `userinfo` routine.

Example:

```
> a1:=mkImplAlgSet([y^2*z^3-x^5],[x,y,z],[basespace=projective]);
                                        2  3    5
             a1 := algebraic_set([y  z  - x ], [x, y, z])

> singularities(a1);

table([

    2 = [[0, 0, 1]]

    3 = [[0, 1, 0]]

    ])
```

See also: `neighbGraph`

Function: neighbGraph

Action: neighborhood graph of plane algebraic curve
Calling sequence:
`neighbGraph(`a`)`

Arguments:

a – algebraic set in implicit representation specifying a plane curve

Synopsis:

- This function returns the neighborhood tree of a plane curve. It is represented by a list of form:

<neighborhood tree>	::=	[] | [<nnodes>]
<nnodes>	::=	NULL | [<nnode>,<nnodes>]
<nnode>	::=	[*mult*,*point*,*trans*,<neighborhood tree>]

- The above structure has to be interpreted in the following way: On the top level, after applying the transformation *trans* in *nnode*, the top level points become the projective singularities of the curve. Each point is given its multiplicity. This linear coordinate transformation assures that $(1:0:0)$ and $(0:1:0)$ are not points on the homogenized function f of a. On the lower levels, we do the following: Move the singularity to the origin (i.e. $(0:0:1)$, apply the transformation *trans* in *nnode* such that no tangent at $(0:0:1)$ is an irregular line (i.e. $x=0$ or $y=0$), no other fundamental point (i.e. $(1:0:0)$ and $(0:1:0)$) is on the curve. We get a transformed curve, say F. Now the points we have are the singularities (except $(0:1:0)$) of the quadratic transformation of F. Then the neighboring tree of all neighboring points is recursively computed.
- For the theory behind neighboring points we refer to [12].
- Additional information on the trace of computation is provided by the `userinfo` routine.

Example:

```
> a1:=mkImplAlgSet([y^2-x^5],[x,y]);

                                       2     5
                    a1 := algebraic_set([y  - x ], [x, y])

>  neighbGraph(a1);

        [[2, 0, 0, 1, {y = y - x, x = x + y}, [[2, 1, 1, 0, {}, []]]],

            [3, -1, 1, 0, {y = y - x, x = x + y},

                [[2, 2, 1, 0, {x = x + 2 y, y = y - 2 x}, []]]]]
```

See also: `adjointCurve`

Function: `genus`

Action: genus of an algebraic set defining a plane curve

Calling sequence:

`genus(`*a*`)`
`genus(`*a*, *ntree*`)`

Arguments:

a – plane algebraic curve

Optional Arguments:

ntree – an unevaluated name

Synopsis:

- This procedure computes the genus of an algebraic curve in implicit representation. An entry `genus = <genus>` is added to the list of attributes.

- If the second argument *ntree* is used, it will be assigned the neighboring tree of the algebraic curve.

Example:

```
> a1:=mkImplAlgSet([y^3+y^2-x^2],[x,y]);
                                          3    2    2
                 a1 := algebraic_set([y  + y  - x ], [x, y])

> genus(a1,evaln(tmp));

                                         0

> tmp;

                  [[2, 0, 0, 1, {x = x + y, y = y - x}, []]]

> Attributes(a1);

                                   [genus = 0]
```

See also: `singularities`, `neighbGraph`.

Function: adjointCurve

Action: adjoint curve

Calling sequence:

`adjointCurve`(a, deg)
`adjointCurve`(a, deg, c)
`adjointCurve`(a, deg, c, ng)

Arguments:

a – algebraic set defining a plane curve
deg – degree of the adjoint curve

Optional Arguments:

c – unevaluated name
ng – unevaluated name

Synopsis:

- The function computes a linear system L of plane curves passing through all singularities of the given plane curve a with multiplicities $m_P(b) \geq m_P(a) - 1, \forall b \in L$.
- If the (optional) third argument is not specified a global table named c is holding coefficients of the adjoint curve. Otherwise the given name is used instead.
- The (optional) fourth argument is assigned the neighborhood graph of a.

Example:

```
> a1:=mkImplAlgSet([y^2*z^3-x^5],[x,y,z],[basespace=projective]);
                                        3  2    5
                a1 := algebraic_set([z  y  - x ], [x, y, z])

> adjointCurve(a1,3,evaln(w),evaln(neigh));

                           2
w[0, 1, 2] (y - x) z  + w[1, 1, 1] (x + y) (y - x) z
```

```
                          2                       3
    + w[2, 0, 1] (x + y)  z + w[3, 0, 0] (x + y)
```

See also: `neighbGraph`

4.6.2 Reparametrization

Function: **reparametrize**

Action: reparametrization of an algebraic set

Calling sequence:

`reparametrize(`*var1*=*exp1*, *var2*=*exp2*, ..., *a*`)`

Arguments:

vari = *expi* – equation which specifies the substitution

a – algebraic set in parametric representation

Synopsis:

- The function performs substitutions for the parameter value of algebraic sets in parametric representation. The variable *vari* is replaced by *expi*.

Example:

```
> a1 := mkParaAlgSet([t/(t-2),t^3+2,t],[t]);

                                    t     3
              a1 := algebraic_set([-----, t  + 2, t], [t])
                                  t - 2

> reparametrize(t=t-2,a1);

                     t - 2   3      2
        algebraic_set([-----, t  - 6 t  + 12 t - 6, t - 2], [t])
                     t - 4
```

See also: `mkImplAlgSet`, `mkParaAlgSet`, `mkProjAlgSet`, `mkPlacAlgSet`

Function: **properParametrization**

Action: proper parametrization

Calling sequence:

`properParametrization(`*a*`)`

Arguments:

a – algebraic set in parametric representation

Synopsis:

- The function computes a proper parametrization for an algebraic set in parametric representation. A proper parametrization fulfils the condition that the relationship between points on the curve and parameter values is one-to-one (with the exception of finitely many points on the curve).

Example:

```
> a1:=mkParaAlgSet([(8*s^6-12*s^5+32*s^3+24*s^2+12*s)/(s^6-3*s^5+
  3*s^4+3*s^2+3*s+1),(24*s^5+54*s^4-54*s^3-54*s^2+30*s)/(s^6-3*s^
```

```
  5+3*s^4+3*s^2+3*s+1)],[s]);
                             6       5       3       2
                          8 s  - 12 s  + 32 s  + 24 s  + 12 s
a1 := algebraic_set([----------------------------------,
                            6      5      4      2
                           s  - 3 s  + 3 s  + 3 s  + 3 s + 1

        5       4       3       2
    24 s  + 54 s  - 54 s  - 54 s  + 30 s
    -----------------------------------], [s])
      6      5      4      2
     s  - 3 s  + 3 s  + 3 s  + 3 s + 1

> properParametrization(a1);
                          2         3
   -63 + 678 s - 2421 s  + 2870 s
[8 ------------------------------,
                          2         3
   -35 + 399 s - 1497 s  + 1853 s

                              2         3
       -87 + 836 s - 2675 s  + 2850 s
    12 ------------------------------]
                              2         3
       -35 + 399 s - 1497 s  + 1853 s
```

See also: `mkImplAlgSet`, `mkParaAlgSet`, `mkProjAlgSet`, `mkPlacAlgSet`, `reparametize`

4.6.3 Creation of General Curves

Function: passGenCurve

Action: generating a general curve and passing it through specified points

Calling sequence:

`passGenCurve(`*deg*, *points*`)`
`passGenCurve(`*deg*, *points*, *vars*`)`
`passGenCurve(`*deg*, *points*, *vars*, *c*`)`
`passGenCurve(`*deg*, *points*, *vars*, *c*, *ext*`)`

Arguments:

deg – degree of the curve,
points – list of points.

Optional Arguments:

vars – list of variables,
c – name of a global table to holding coefficients,
ext – list of extensions.

Synopsis:

- The function call `passGenCurve(`*deg*, *points*`)` returns a curve (or a system of curves) in the projective space of degree *deg* passing through specified *points*. The resulting curve will contain *vars* as variables (default is `casa/homvars`).
- If the optional fourth parameter is given, it is used as a name of a table holding coefficients of the resulting curve (system).

- When some of points have algebraic coordinates, the fifth argument has to list extensions, where coordinates of points are lying. Precisely, for each point there has to be a list describing the extension of the coordinate field over the rational numbers. If all coordinates of a point are rational, the empty list has to be supplied. Moreover, all elements has to fullfil requirements listed in the corresponding section of the description of `neighbGraph`.
- A point is given either as a list $[x0, x1, x2]$ (denoting a simple point) or as a nested list $[[x0, x1, x2], m]$ (denoting a singular point with multiplicity m). Note that $m = 1$ is possible.

Example:

```
> c1:=passGenCurve(5,[[[RootOf(_z^2+1),1,0],2],[[0,0,1],3],[[-1,-1,1],2]],
  [x,y,z],'u',[[RootOf(_z^2+1)],[],[]]);
                         5                4
c1 := algebraic_set([x  u[1, 4, 0] + x  u[0, 5, 0] y

        4
     + x  (4 u[1, 4, 0] + 4 u[0, 5, 0] - u[1, 3, 1] - u[0, 4, 1]) z

          3              2    3                   3
     + 2 x  u[1, 4, 0] y  + x  u[1, 3, 1] z y + x  (4 u[1, 4, 0]

     + 8 u[0, 5, 0] - 2 u[1, 3, 1] + u[1, 2, 2] - 4 u[0, 4, 1]

                       2      2             3
     + 2 u[0, 3, 2]) z  + 2 x  u[0, 5, 0] y

        2                                                2    2
     + x  (4 u[1, 4, 0] + 4 u[0, 5, 0] - u[1, 3, 1]) z y  + x  (

    -4 u[0, 5, 0] + 2 u[1, 3, 1] - 2 u[1, 2, 2] + 4 u[0, 4, 1]

                       2                 4                  3
     - 3 u[0, 3, 2]) z  y + x u[1, 4, 0] y  + x u[1, 3, 1] z y

                     2  2                5                 4
     + x u[1, 2, 2] z  y  + u[0, 5, 0] y  + u[0, 4, 1] z y

                    2  3
     + u[0, 3, 2] z  y ], [x,y,z])

> setRandomParameters(c1);

                    5        4       4         3  2       3
algebraic_set([7 x  + 8 y x  + 56 x  z + 14 x  y  + 10 x  z y

           3  2       3  2       2    2      2  2          4
     + 78 x  z  + 16 y  x  + 50 x  z y  - 5 x  z  y + 7 x y

               3        2  2      5        4      2  3
     + 10 x z y  - 8 x z  y  + 8 y  - 6 z y  - 5 z  y ], [x, y, z])

> singularities(");

table([
```

```
3 = [[0, 0, 1]]

                                                2
2 = [[-1, -1, 1], [RootOf(_Z  + 1), 1, 0]]

])
```

See also: `setRandomParameters`

Function: setRandomParameters

Action: choose undetermined parameters at random

Calling sequence:

`setRandomParameters(`*a*`)`
`setRandomParameters(`*a*, *range*`)`
`setRandomParameters(`*a*, *range*, *name*`)`

Arguments:

a – system of algebraic sets

Optional Arguments:

range – random coefficients range
name – unevaluated name

Synopsis:

- The function returns a single algebraic set from the input system of algebraic sets *a* containing parameters. The resulting set is obtained by setting undetermined parameters to random values from *range* (default -10..10).
- If an unevaluated name is specified as the third parameter it is assigned a set with particular values selected for each parameter.
- The standard Maple function `rand` is used to select random values.

Example:

```
> c1:=passGenCurve(5,[[[0,1,0],2],[[0,0,1],3],[[1,1,-1],2]],[x,y,z],'u');

c1 := algebraic_set([(-u[3, 1, 1] + 2 u[0, 3, 2] - u[1, 3, 1]

                                                                 4
     + 2 u[1, 2, 2] - u[2, 2, 1] + 2 u[2, 1, 2] + 2 u[3, 0, 2]) z x

                   3  2                  2  3
     + u[2, 3, 0] y  x  + u[3, 2, 0] y  x  + (-3 u[2, 3, 0]

     - 2 u[3, 2, 0] + u[3, 1, 1] - 3 u[0, 3, 2] + 3 u[1, 3, 1]

                                                    4
     - 2 u[1, 2, 2] + 2 u[2, 2, 1] - u[2, 1, 2]) y x  + (-u[3, 1, 1]

     + 4 u[0, 3, 2] - 3 u[1, 3, 1] + 3 u[1, 2, 2] - 2 u[2, 2, 1]

                                                              5
     + 2 u[2, 1, 2] + u[3, 0, 2] + 2 u[2, 3, 0] + u[3, 2, 0]) x

                       3    3                2    3
```

```
     + u[3, 1, 1] y z x  + y  u[0, 3, 2] z  + y  u[1, 3, 1] z x

          2             2      2                2                  2   2
     + y  u[1, 2, 2] z  x + y  u[2, 2, 1] z x  + y u[2, 1, 2] z  x

                      2   3
     + u[3, 0, 2] z  x ], [x, y, z])

> setRandomParameters(c1,'s');
                     4          2   3       3   2         4          5
algebraic_set([-39 x  z - 4 x  y  + 7 x  y  + 78 x  y - 90 x

         3           3   2         3           2   2       2   2
     + 8 x  y z - 6 y  z  + 7 x y  z - 8 x y  z  + 6 x  y  z

         2    2       3   2
     - 5 x  y z  + 10 x  z ], [x, y, z])

> s;
{u[2, 3, 0] = -4, u[3, 2, 0] = 7, u[3, 1, 1] = 8, u[3, 0, 2] = 10,

    u[0, 3, 2] = -6, u[1, 2, 2] = -8, u[2, 1, 2] = -5, u[1, 3, 1] = 7,

    u[2, 2, 1] = 6}
```

See also: `passGenCurve`

4.6.4 Rational Points

Function: **ratpoint**

Action: compute a rational point on a conic

Calling sequence:

`ratpoint(`g`)`

Arguments:

g – bivariate polynomial of total degree 2

Synopsis:

- The function decides whether there is a rational point on the conic defined by g, and if so, computes one. The coefficients of g may be in $\mathbb{Q}$ or in $\mathbb{Q}(t)$ (and hence the coordinates of a rational point have to be in $\mathbb{Q}$ or in $\mathbb{Q}(t)$ respectively). Additional information is provided by **ratpoint** such as the decision whether g defines a parabola or not, the irreducibility of the conic, the time needed for the calculation, and g evaluated at the computed point (i.e. 0).

Example:

```
> c1 := x^2-4*x*y-3*y^2+4*x+8*y-5:
  ratpoint(c1);

Irreducible conic             :   true
Parabola                      :   false
Existence of a rational point :   true
Its x-coordinates             :
                                1
```

```
Its y-coordinates             :

                                 0

Conic equation evaluated there :   0
Time needed for calculation    :   .50e-1

> c2 := t*x^2-2*x*y+(3/t)*y-2:
  ratpoint(c2,[x,y]);

Irreducible conic             :   true
Parabola                      :   false
Existence of a rational point  :   true
Its x-coordinates             :

                               -3 + 8 t
                           1/4 --------
                                  t

Its y-coordinates             :

                            - 1/8 + t

Conic equation evaluated there :   0
Time needed for calculation    :   .80e-1
```

See also: `impl2para`, `convertRep`

4.6.5 Offset Curves

Function: **RPHcurve**

Action: checks if there exists an RPH parametrization of the given curve.

Calling sequence:

`RPHcurve(`C`,` *phi*`)`

Arguments:

C – rational curve in parametric representation,

phi – unevaluated name for the reparametrizing function.

Synopsis:

- The function checks if there exists an RPH (Rational Phytagorean hodograph) parametrization of the algebraic curve C by analyzing the reducibility and rationality of the reparametrizing curve.
- If the reparametrizing curve is irreducible and rational then a quadratic reparametrizing function is assigned to the parameter *phi*.

Example:

```
> a:=mkParaAlgSet([t^7,t^6],[t]);
                                          7   6
                   a := algebraic_set([t , t ], [t])

> RPHcurve(a,'phi');

                                true
```

```
> phi;

                                    t
                          - 12/7 ------
                                   2
                                  t  - 1
```

See also: `mkImplAlgSet`, `mkParaAlgSet`, `paraOffset`, `implOffset`.

Function: implOffset

Action: compute the offset curve in implicit representation to a given curve

Calling sequence:

`implOffset(`C`)`
`implOffset(`C, d`)`

Arguments:

C – algebraic curve in implicit or parametric representation,
d – distance, a variable or a rational number.

Synopsis:

- The function computes the offset curve in implicit representation at distance d to a given plane algebraic curve (in implicit or parametric representation).
- If the parameter d is not provided it is assumed that the distance is unspecified.

Example:

```
> a1:=mkImplAlgSet([x^2-y],[x,y]);
                                            2
               a1 := algebraic_set([x  - y], [x, y])

> implOffset(a1,2);

                                225  2          3       2      2  2
algebraic_set([34 y - 289/4 + --- y  - 17/2 y  + 43 x  - 6 x  y
                               16

                 2   191  4    6    4    4  2        4        2  3
     + 15/8 y x  - --- x  + x  + y  + x  y  - 5/2 x  y - 2 x  y ],
                   16

     [x, y])

> a2:=mkParaAlgSet([2*t/(t^2+1),(t^2-1)/(t^2+1)],[t]);

                                             2
                                    t     t  - 1
             a2 := algebraic_set([2 ------, ------], [t])
                                    2      2
                                   t  + 1  t  + 1

> implOffset(a2);

                 2        2           2
```

```
algebraic_set([x  - 1 + y  - 2 d - d ], [x, y]),

                   2         2           2
    algebraic_set([x  - 1 + y  + 2 d - d ], [x, y])
```

See also: `mkImplAlgSet`, `mkParaAlgSet`, `paraOffset`, `RPHcurve`.

Function: paraOffset

Action: compute the offset curve in parametric representation to a given curve.

Calling sequence:

`paraOffset(`C`)`
`paraOffset(`C`, `d`)`

Arguments:

C – algebraic curve in implicit or parametric representation,
d – distance, a variable or a rational number.

Synopsis:

- The function computes the offset curve in parametric representation at distance d to a given plane algebraic curve (in implicit or parametric representation).
- If the parameter d is not provided it is assumed that the distance is a parameter.

Example:

```
> a1:=mkImplAlgSet([x^2-y],[x,y]);
                                           2
                 a1 := algebraic_set([x  - y], [x, y])

> paraOffset(a1,2);

algebraic_set(

              2              2              6    4    2           3
            (t  + 1 - 8 t) (t  - 1)        t  - t  - t  + 1 + 64 t
    [- 1/4 -----------------------, 1/16 ------------------------],
                      2                         2   2
                 t (t  + 1)                    t  (t  + 1)

    [t])

> a2:=mkParaAlgSet([2*t/(t^2+1),(t^2-1)/(t^2+1)],[t]);

                                                  2
                                          t      t  - 1
              a2 := algebraic_set([2 ------, ------], [t])
                                        2       2
                                       t  + 1  t  + 1

> paraOffset(a2);

                                    2
                  t (-1 + d)      (t  - 1) (-1 + d)
algebraic_set([-2 ----------, - -----------------], [t]),
```

```
                              2                 2
                             t  + 1
```

See also: `mkImplAlgSet`, `mkParaAlgSet`, `implOffset`, `RPHcurve`.

4.6.6 Others

Function: imult

Action:
Calling sequence:
`imult(`*a1*, *a2*, *point*`)`
Arguments:
a1,*a2* – algebraic sets in implicit representation specifying two plane curves,
point – a point given as a list with two entries.
Synopsis:

- The funtion computes the intersection-multiplicity at *point* of the two plane curves.

Example:

```
> a1:=mkImplAlgSet([x^4-y^2],[x,y]);
                                        4    2
                   a1 := algebraic_set([x  - y ], [x, y])

> a2:=mkImplAlgSet([expand(x^4-y^3)],[x,y]);

                                        4    3
                   a2 := algebraic_set([x  - y ], [x, y])

> imult(a1,a2,[0,0]);

                                     8
```

4.7 Gröbner Bases

4.7.1 Gröbner Bases for Ideals

Function: Groebnerbasis

Action: Gröbner basis for an implicitly given algebraic set
Calling sequence:
`Groebnerbasis(`*a*`)`
`Groebnerbasis(`*a*, *order*`)`
Arguments:
a – algebraic set in implicit representation
Optional Arguments:
order – string `tdeg` or `plex`
Synopsis:

- The function computes a Gröbner basis for an implicitly given algebraic set. The ordering of the variables is according to the list of variables, the greatest variables comes first.
- The term ordering is the total degree term ordering unless it is specified differently by an optional parameter.

- This function is nothing but an interface between the datatype `algebraic_set` and Maple's package for Groebner bases computation.

Example:

```
> a1:=mkImplAlgSet([x*y*z^2-4,x^2+y^2+z^2-1,x-y-1]);

                              2        2    2    2
a1 := algebraic_set([x y z  - 4, x  + y  + z  - 1, x - y - 1], [x, y, z])

> Groebnerbasis(a1);

                                         4     2    2
                   [x - y - 1, 8 + z , 2 y  + z  + 2 y]
```

4.7.2 Gröbner Bases for $K[X]^r$-modules

Function: **mgrobner**

Action: package for computing Gröbner basis of polynomial modules
Calling sequence:
`mgrobner[`*function*`](`*args*`)`
function(*args*)
Arguments:
none
Synopsis:

- mgrobner is a subpackage of CASA for doing Groebner basis and syzygy computation not only for ideals of polynomials but for modules over polynomial rings. The package consists of five procedures: **`mgbasis`** and **`mgbasisx`** to compute the Groebner basis of a module, **`mnormalf`** to get the normal form of a polynomial tuple modulo a module, **`msolveGB`** and **`msolveSP`** to solve a system of linear equations with polynomial coefficients (syzygy computation).
- To access functions in this package within CASA use either long form `mgrobner[`*function*`](`*args*`)` or after **`with(mgrobner)`** short notations *function*(*args*) are possible.
- Available functions are:

mgbasis	mgbasisx
mnormalf	msolveGB
msolveSP	

See also: `grobner`

Function: **mgrobner[mgbasis]**

Action: normed reduced Gröbner basis of a module
Calling sequence:
`mgrobner[mgbasis]`(F, X)
`mgrobner[mgbasis]`$(F, X,$ *torder1*$)$
`mgrobner[mgbasis]`$(F, X,$ *torder1*, *torder2*$)$
Arguments:
F – list of polynomial tuples
X – list of indeterminates
Optional Arguments:

torder1 – power-product-tuple ordering: **term** (term first) or **index** (default is index first)
torder2 – power-product ordering: **plex** (pure lexicographic) or **tdeg** (default is total degree)

Synopsis:

- The function computes the normed reduced Gröbner basis of F with respect to the indeterminates X and the given orderings.
- The polynomial tuples in F are represented as lists of polynomials.
- If X has the form $[x_1, x_2, ..., x_n]$, then using the pure lexicographic ordering this is interpreted as $x_1 > x_2 > \ldots > x_n$. Within the total degree ordering, ties are broken by inverse lexicographic order.
- The command `with(mgrobner,mgbasis)` allows the use of the abbreviated form of this command.

Example:

```
> F1 := [[x*y-1,x+2],[y^2+x+1,y-1]]:
> mgrobner[mgbasis](F1,[x,y]);

            2                                       2
        [[x  + x + y, - x - 2 y], [x y - 1, x + 2], [y  + x + 1, y - 1],

                  2            2
            [0, x  + x y + 2 y  + 3 x + y + 1]]

> mgrobner[mgbasis](F1,[y,x],term,plex);

                   3        2         2                  2
            [[1/4 x  + 1/4 x  + 1/4, y  + 1/2 y + 1/4 x  + 5/4 x],

                       3        2                     2
                [- 1/2 x  - 1/2 x  - 1/2, y x + 1/2 x  + 1/2 x + 1],

                      2
                [y + x  + x, - 2 y - x]]
```

See also: `grobner`

Function: mgbasisx

Action: normed reduced Gröbner basis along with the transformationmatrix

Calling sequence:
`mgrobner[mgbasisx]`(F, G, GX, X)
`mgrobner[mgbasisx]`$(F, G, GX, X,$ *torder1*$)$
`mgrobner[mgbasisx]`$(F, G, GX, X,$ *torder1*, *torder2*$)$

Arguments:
F – list of polynomial tuples
G – the normed reduced Gröbner basis (output)
GX – the transformation matrix (i.e. GX.F=G) (output)
X – list of indeterminates

Optional Arguments:
torder1 – power-product-tuple ordering: **term** (term first) or **index** (default is index first)
torder2 – power-product ordering: **plex** (pure lexicographic) or **tdeg** (default is total degree)

Synopsis:

- The function computes the normed reduced Gröbner basis of F with respect to the indeterminates X and the given orderings. The result is also returned in the variable G. In GX the transformation matrix is stored. The algorithm used in **mgbasisx** is basically the same as used in **mgbasis**, but the reduction steps are stored to get the transformation matrix GX.
- The polynomial tuples in F are represented as lists of polynomials.
- If X has the form $[x_1, x_2, \ldots, x_n]$, then using the pure lexicographic ordering this is interpreted as $x_1 > x_2 > \ldots > x_n$. Within the total degree ordering, ties are broken by inverse lexicographic order.
- The command `with(mgrobner,mgbasisx)` allows the use of the abbreviated form of this command.

Example:

```
> F1 := [[x*y-1,x+2],[y^2+x+1,y-1]]:
> mgbasisx(F1,'G1','GX',[x,y]):
> G1;

          2                                         2
      [[x  + x + y, - x - 2 y], [x y - 1, x + 2], [y  + x + 1, y - 1],

                2            2
          [0, x  + x y + 2 y  + 3 x + y + 1]]

> GX;

                                          2
             [[- y, x], [1, 0], [0, 1], [y  + x + 1, - x y + 1]]

> mgbasisx(F1,'G1','GX',[y,x],term,plex):
> G1;

                  3         2         2                 2
          [[1/4 x  + 1/4 x  + 1/4, y  + 1/2 y + 1/4 x  + 5/4 x],

                    3         2                  2
              [- 1/2 x  - 1/2 x  - 1/2, y x + 1/2 x  + 1/2 x + 1],

                     2
              [y + x  + x, - 2 y - x]]

> GX;

                2                                                 2
        [[1/2 y  + 1/4 + 1/2 x - 1/4 y x, - 1/2 y x + 1/2 + 1/4 x ],

                                        2
            [1/2 y x + 1/2, - 1/2 x ], [- y, x]]
```

See also: mgbasis

Function: mnormalf

Action: normal form of a tuple of polynomials modulo a submodule

Calling sequence:

`mnormalf`(f, F, X)
`mnormalf`($f, F, X, torder1$)
`mnormalf`($f, F, X, torder1, torder2$)

Arguments:

f – a tuple of polynomials
F – list of polynomial tuples (a Gröbner basis)
X – list of indeterminates

Optional Arguments:

torder1 – power-product-tuple ordering: `term` (term first) or `index` (default is index first)
torder2 – power-product ordering: `plex` (pure lexicographic) or `tdeg` (default is total degree)

Synopsis:

- The function computes the normal form of the tuple of polynomials f modulo the module generated by F with respect to the indeterminates X and the given orderings.
- Note: F must be a Gröbner basis . Therefore, usually first a Gröbner basis for a set of polynomial tuples is computed (with `mgbasis`) and afterwards `mnormalf` is called.
- If X has the form $[x_1, x_2, \ldots, x_n]$, then using the pure lexicographic ordering this is interpreted as $x_1 > x_2 > \ldots > x_n$. Within the total degree ordering, ties are broken by inverse lexicographic order.

Example:

```
> F:=[[x-1,x*y-1],[x*y-1,y],[y,x-1]]:
> G1:=mgbasis(F,[x,y]):
> mnormalf([x^3*y^2+x^3+1,y^3+x+1],G1,[x,y]);

                         2                  2
                  [0, x  + 6/5 x y + 2/5 y  - 8/5 x - 2/5 y - 3/5]

> G2:=mgbasis(F,[x,y],term):
> mnormalf([x^3*y^2+x^3+1,y^3+x+1],G2,[x,y],term);

                                                 2
                         [- 3 x + 3/2 y + 1/2, y  + 3/2 y]
```

See also: `mgbasis`

Function: mgrobner[msolveGB]

Action: basis for the module of syzygies

Calling sequence:

`mgrobner[msolveGB]`($F, rhsf, X$)
`mgrobner[msolveGB]`($F, rhsf, X, torder1$)
`mgrobner[msolveGB]`($F, rhsf, X, torder1, torder2$)

Arguments:

F – list of polynomial tuples
rhsf – right-hand-side of the linear system
X – list of indeterminates

Optional Arguments:

torder1 – power-product-tuple ordering: `term` (term first) or `index` (default is index first)
torder2 – power-product ordering: `plex` (pure lexicographic) or `tdeg` (default is total degree)

Synopsis:

- The function computes a basis for the module of syzygies using the extended Gröbner-basis-algorithm. The procedure returns a list with two elements: the first element is a particular solution, the second one is a basis for the module of syzygies (represented as a list).
- The polynomial tuples in F are represented as lists of polynomials.
- F and *rhsf* have to be interpreted as: e.g., `F:=[[f11,f21],[f12,f22],[f13,f23]];` `rhsf:=[f1,f2,f3];` yields the linear system

$$\begin{pmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \end{pmatrix}$$

- If X has the form $[x_1, x_2, ..., x_n]$, then using the pure lexicographic ordering this is interpreted as $x_1 > x_2 > \ldots > x_n$. Within the total degree ordering, ties are broken by inverse lexicographic order.
- The syzygy basis is computed by determining the Gröbner basis first and then reducing S-polynomials of pairs of basis elements to zero. This is an alternative strategy to that used in the function **msolveSP**.
- The command **with(mgrobner,mgbasis)** allows the use of the abbreviated form of this command.

Example:

```
> F1 := [[x^3-y,-x+y],[x*y-1,x^3-1],[-x+y^2,y-1]]:
> S1 := msolveGB(F1,[0,0],[x,y]):
> nops(S1[2]);

                                   4

> mgbasis(S1[2],[x,y]);

                    3  2    4      2          2
                [[x   y  - x  - x y  + x y - y  + x + y - 1,

                     3      3      2    3    2          2
                    x  y - x  + x y  - y  - x  + x y - y  + y,

                       6    3      3    2         2
                    - x  + x  y + x  - x  y + x y  + x - 2 y]]

> msolveGB(F1,[x*y^2-y^2,-x^5+x^2-x+y^2],[x,y],term,tdeg);

                 2           3  2    4      2          2
        [[- y, x , - x], [[x   y  - x  - x y  + x y - y  + x + y - 1,

                                 3      3      2    3    2          2
                                x  y - x  + x y  - y  - x  + x y - y  + y,

                                   6    3      3    2         2
                                - x  + x  y + x  - x  y + x y  + x - 2 y]] ]
```

See also: `mgrobner`, `msolveSP`

Function: mgrobner[msolveSP]

Action: basis for the module of syzygies

Calling sequence:

`mgrobner[msolveSP]`($F, rhsf, X$),
`mgrobner[msolveSP]`($F, rhsf, X, torder1$),
`mgrobner[msolveSP]`($F, rhsf, X, torder1, torder2$).

Arguments:

F – list of polynomial tuples
rhsf – right-hand-side of the linear system
X – list of indeterminates

Optional Arguments:

torder1 – power-product-tuple ordering: `term` (term first) or `index` (default is index first)
torder2 – power-product ordering: `plex` (pure lexicographic) or `tdeg` (default is total degree)

Synopsis:

- The function computes a basis for the module of syzygies using a different strategy than that in `msolveGB`. The procedure returns a list with two elements: the first element is a particular solution, the second one is a basis for the module of syzygies (represented as a list).
- The polynomial tuples in F are represented as lists of polynomials.
- F and *rhsf* have to be interpreted as: e.g., `F:=[[f11,f21],[f12,f22],[f13,f23]];` `rhsf:=[f1,f2,f3];` yields the linear system

$$\begin{pmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix}$$

- If X has the form [x1,x2,...,xn], then using the pure lexicographic ordering this is interpreted as $x_1 > x_2 > \ldots > x_n$. Within the total degree ordering, ties are broken by inverse lexicographic order.
- The syzygy basis is computed in the course of building the Gröbner basis. This is a slightly different strategy compared to the function `msolveGB`
- The command `with(mgrobner,mgbasis)` allows the use of the abbreviated form of this command.

Example:

```
> F1 := [[x^3-y,-x+y],[x*y-1,x^3-1],[-x+y^2,y-1]]:
  S1 := mgrobner[msolveSP](F1,[0,0],[x,y]):
  nops(S1[2]);
                                3

> mgrobner[mgbasis](S1[2],[x,y]);

   3  2    4       2          2
[[x  y  - x  - x y  + x y - y  + x + y - 1,

     3      3      2    3    2          2
    x  y - x  + x y  - y  - x  + x y - y  + y,

     6    3      3    2        2
    -x  + x  y + x  - x  y + x y  + x - 2 y]]

> mgrobner[msolveSP](F1,[x*y^2-y^2,-x^5+x^2-x+y^2],[x,y],term,tdeg);

      2        4    3  2                     2       2
[[y, -x , x], [[x  - x  y  - x y - y + 1 + x y  - x + y ,

     3      3    2         3    2          2
```

```
-x  y + x  + x  - x y + y  + y  - y - x y ,

 6   3     2   2     3
x  - x  - x y  + x  y - x  y - x + 2 y]]]
```

See also: `msolveGB`

4.7.3 Gröbner Walk

Function: **GWalk**

Action: compute the reduced Groebner basis with respect to the lexicographic order of an ideal

Calling sequence:

`GWalk(`*F*`, `*X1*`)`,
`GWalk(`*F*`, `*X1*`, `*X2*`)`.

Arguments:

F – list of polynomials,
X1 – list of indeterminates,
X2 – list of indeterminates.

Synopsis:

- The function computes the reduced, minimal Groebner basis of the polynomial system F in the indeterminates *X1* with respect to the lexicographical ordering. The algorithm first computes a Groebner basis with respect to `tdeg` ordering by using the function `grobner[gbasis]` and converts it to a lexicographical Groebner basis by the use of the Groebner Walk algorithm
- The list of indeterminates $X1 := [x_1, x_2, ..., x_n]$ induces the ordering $x_1 > x_2 > ... > x_n$.
- If the procedure is called with 3 arguments then the list of indeterminates $X2 := [x_1, x_2, ..., x_n]$ induces the ordering $x_1 > x_2 > ... > x_n$ and is used for the computation of the `tdeg` Groebner basis. Otherwise, *X1* is used for this computation.

Example:

```
> GWalk([t^2+y^2+z^2-1,t^2+z^2-y,t-z],[t,y,z]);

                          2         4         2
               [t - z, 2 z  - y, 4 z  - 1 + 2 z ]

> GWalk([ 1+x^2+z,x*z+y^2*z],[z,x,y],[x,y,z]);

                      2       2  2    3    2
               [1 + x  + z, x  y  + x  + y  + x]
```

See also: `grobner`

4.8 Hybrid Symbolic-Numerical Methods

4.8.1 Finding Solutions of an Arbitrary System of Nonlinear Equations

Function: **extNewton**

Action: find solutions for an arbitrary system of nonlinear equations.

Calling sequence:

`extNewtonFunc(`*funcs*`, `*vars*`)`

`extNewton(`*Func*`,` *vars*`,` *init*`)`

Arguments:

funcs – a list of equations,
Func – the iterative function constructed by `extNewtonFunc`,
vars – a list of variables,
init – an initial point.

Optional Arguments:

loops – maximum iterations used by the function,
epsilon – 10^*epsilon* is the tolerance used during the computation,
digits – the number of digits used during the computation.

Synopsis:

- In CAGD or solid modeling, we often need to find some solutions for a system of nonlinear equations where the field of coefficients may not be algebraically closed (for example the field of real numbers). The function iteratively computes solutions for an arbitrary system of nonlinear equations, i.e. the system can be nonzero-dimensional or zero-dimensional.
- The system can be underdetermined or overdetermined. In case the number of equations is equal to the number of variables, the method is similar to Newton's method.
- The method is a hybrid symbolic-numerical method, in that we use symbolic techniques to transform a system of algebraic equations into a regular system whenever the system is not underdetermined or does not have a full-rank Jacobian matrix (see [9]).

Example:

```
> f1:=x1^2+x2^2+x3^2-1:
  f2:=x4*x1+x5*x2+x6*x3:
  f3:=(x4+3*x3)^2+x5^2+(x6-3*x1)^2-1:
  List_of_polys:=normal([f1,f2,f3]);
  Varlist:=[x1,x2,x3,x4,x5,x6];
                           2     2     2
List_of_polys := [x1  + x2  + x3  - 1, x4 x1 + x5 x2 + x6 x3,

      2                  2     2     2                  2
    x4  + 6 x4 x3 + 9 x3  + x5  + x6  - 6 x6 x1 + 9 x1  - 1]


                 Varlist := [x1, x2, x3, x4, x5, x6]

> Func:=extNewtonFunc(List_of_polys,Varlist):
  so:=extNewton(Func,[rand(20)(),rand(20)(),rand(20)(),rand(20)(),
  rand(20)(),rand(20)()]);

so := [.714849223293176846, .0254139182380360877, .698816657453824081,

    -2.56280010705017309, .761979375842043262, 2.59388605770756449]

> sub:={'Varlist[i]=so[i]' $ 'i'=1..nops(so)};
  evalf(subs(sub,List_of_polys));
  sub := {x4 = -2.56280010705017309, x3 = .698816657453824081,

    x2 = .0254139182380360877, x1 = .714849223293176846,

    x6 = 2.59388605770756449, x5 = .761979375842043262}


                                 -9        -8        -8
```

```
[-.1 10   , .1 10   , .2 10   ]
```

See also: `casa`

4.8.2 Real Root Isolation

Function: **realroot_sb**

Action: isolating real roots of a polynomial in strictly separated intervals using non-recursive algorithms.

Calling sequence:

`realroot_sb(`*poly*`)`,
`realroot_sb(`*poly*, *widthgoal*`)`,
`realroot_sb(`*poly*, *interval*`)`,
`realroot_sb(`*poly*, *interval*, *widthgoal*`)`.

Arguments:

poly – a univariate polynomial with numeric coefficients.

Optional Arguments:

widthgoal – maximal size of each isolating interval,
interval – an interval *[l,r]* where the user wants to isolate all of the real roots of the polynomial.

Synopsis:

- The function returns a sorted list of strictly separated isolating intervals for all real roots of the univariate polynomial *poly* within *interval*. If *interval* is omitted, all real roots of the polynomial will be isolated (easier problem).
- The width of the interval is less than or equal to the optional parameter *widthgoal* – a positive number. If *widthgoal* is omitted, the most convenient width is used for each interval returned.
- The function now accepts polynomials with floating point coefficiences. The polynomial will be converted to a rational polynomial before isolating the real roots.
- For isolating real roots within an OPEN interval, the algorithm in this function avoids to use transformations and therefore does not blow up the coefficients of the polynomial. It is many times faster than algorithms using transformations.

Example:

```
> realroot_sb(2.5*x^8+x^7-4*x^6-20*x^5+4*x^4+20*x^3, 1/1000);
             -493  -985            569  1139    951  1903
          [[----, ----], [0, 0], [---, ----], [---, ----]]
             512   1024            512  1024    512  1024

> p1:=expand(subs({n=20,a=5},x^n-2*(a*x-1)^2)): # Mignotte polynomials
> realroot_sb(p1,[0,1]);

              53687087   107374175    13421773  6710887
           [[---------, ---------], [--------, --------]]
              268435456  536870912    67108864  33554432
```

See also: `realroot_a`, `realroot`

Function: **realroot_a**

Action: isolating real roots of a polynomial with algebraic number coefficients in strictly seperated intervals using non-recursive algorithms and norm.

Calling sequence:

`realroot_a(`*minimal*`, `*interval*`,`*poly*`),`
`realroot_a(`*minimal*`, `*interval*`, `*poly*`, `*widthgoal*`).`

Arguments:

minimal – the defining polynomial of an algebraic number α,
interval – an isolating interval for α,
poly – a square-free univariate polynomial with algebraic number coefficients (α).

Optional Arguments:

widthgoal – maximal size of each isolating interval.

Synopsis:

- The function returns a sorted list of strictly separated isolating intervals for all real roots of the polynomial *poly*.
- The width of the interval is less than or equal to the optional parameter *widthgoal* – a positive number. If *widthgoal* is omitted, the most convenient width is used for each interval returned.
- The polynomial with algebraic number coefficients has to be square-free.

Example:

```
> eq1:=expand(2*x^4-3*x^2*y+y^2*(y-1)^2);
  Ay:=8*y^2-16*y-1:
  alpha:=RootOf(Ay):
  eq2:=subs(y=alpha,eq1):
  # Algebraic sqrfree
  Geq2 := evala(Gcd(eq2,diff(eq2,x))):
  evala(Divide(eq2,Geq2,evaln(eq2))):
  eq2 := primpart(subs(alpha=y,eq2));
  r1:=`realroot_a/realroot`(Ay);
                         4      2      4      3    2
              eq1 := 2 x  - 3 x  y + y  - 2 y  + y


                                    2
                         eq2 := 4 x  - 3 y


                               -1  -1
                      r1 := [[--, --], [2, 3]]
                               16  32

> realroot_a(Ay,r1[1],eq2);

                                  []

> realroot_a(Ay,r1[2],eq2);

                       [[-3/2, -1], [1, 3/2]]
```

See also: `realroot_sb`, `realroot`

4.8.3 Multivariate Resultants

Function: mvresultant

Action: compute the resultant system of a system of multivariate polynomials.

Calling sequence:
`mvresultant(`*repr*, *vars*`)`.

Arguments:
repr – list of polynomials in *vars* with (optional) parameters and coefficients of type `casaCoeffType`,
vars – list of variables.

Synopsis:

- The function computes the resultant system of a system of multivariate polynomials in order to set up conditions for their solvability as well as formulas for calculating their solutions.
- The vanishing of the resultant system provides a solvability creterion for the existance of common solutions of the system (in projective space).
- We use the algorithm described in [11].

Example:

```
> F1:=[x2^2+x1^2-1,x1*x2-1,x2^2 -x1]:
  mvresultant(F1,[x1,x2]);
                                   [4]

> F2:=[v*x2^2+RootOf(x^3+1)*x1^2-1,x1*x2-1,x2^2 -x1]:
  mvresultant(F2,[x1,x2]);

                3              3                 3     3
              [v  + 3 v RootOf(_Z  + 1) + RootOf(_Z  + 1)  - 1]

> F3:=[x3-x1*x2-x2,x2*x3-x1,x1*x3-x2]:
  mvresultant(F3,[x1]);

                                   2               2
                       [-x3 + x2 + x2  x3, x2 - x2 x3 ]

> mvresultant(F3,[x1,x2]);

                                          2     4
                        [-16 x3 (121 - 58 x3  + x3 )]
```

See also: `dixon, resultant`

Function: dixon

Action: compute an extended Dixon's resultant of a system of multivariate polynomials.

Calling sequence:
`dixon(`*repr*, *vars*`)`.

Arguments:
repr – list of polynomials in *vars* with (optional) parameters and coefficients of type rational,
vars – list of variables.

Synopsis:

- The function computes an extended Dixon's resultant of a system of $n+1$ polynomials in n variables in order to set up necessary conditions for their solvability.
- The vanishing of the extended Dixon's resultant provides a necessarily solvability creterion for the existance of common solutions of the system (in projective space).
- This method is much more efficient than other methods for elimination.

- We use the algorithm described in [8].

Example:

```
> F1:=[x2^2+x1^2-1,x1*x2-1,x2^2 -x1]:
  dixon(F1,[x1,x2]);
                                   -4
> F2:=[x1*y2-(y1^2-y2^2),x2*y1-(y1^2-y2^2),x3*y2-1,t*y1*y2-1]:
  dixon(F2,[t,y2,y1]);
                               2        2        2
                  x3 x2 (-x1  + x3 x2  x1 + x2 )
```

See also: `mvresultant`, `resultant`

4.9 Scientific Visualization

Function: **pacPlot**

Action: plotting plane algebraic curves using hybrid symbolic-numerical method

Calling sequence:

`pacPlot(`a`)`

`pacPlot(`a`,` *options*`)`

Arguments:

a – a bivariate polynomial.

Optional Arguments:

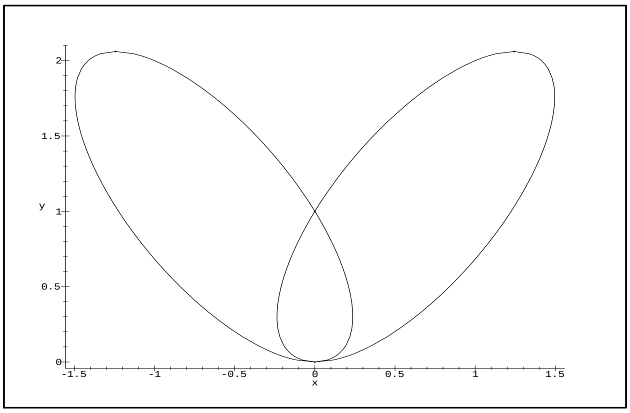
options – options to the function.

Synopsis:

- The function plots plane curves in implicit form. The range for the variables will be automatically choosen based on the critical points of the curve.
- Options are given as equations of the form *option* = *value*. All options that are supported by Maple's plotting functions (`plot[options]`) are accepted.

Example:

```
> pacPlot(2*x^4-3*x^2*y+y^2*(y-1)^2);
```



See also: `ssiPlot`, `plotAlgSet`

Function: ssiPlot

Action: plotting the intersection of two surfaces (i.e. space curve) using hybrid symbolic-numerical method.

Calling sequence:

`ssiPlot(`*expr1*, *expr2*, *vars*`)`
`ssiPlot(`*expr1*, *expr2*, *vars*, *options*`)`

Arguments:

expr1,*expr2* – two polynomials in three variables
vars – list of variables.

Optional Arguments:

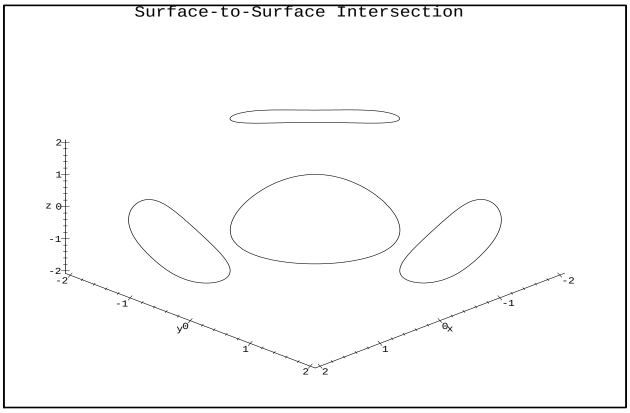
options – options to the function.

Synopsis:

- The function plots the intersection of two three-dimensional and implicitly defined surfaces. An appropriate range for each variable will be automatically computed. The user does not need to specify the ranges.
- Options are given as equations of the form *option=value*. The following options are supported: (1) All options that are supported by Maple's plotting functions (`plot[options]`). (2) *epsilon=-n*. Specifies the precision for real root isolation procedures used in the function. (3) *maxiterate=n*. Specifies the maximum number of iterations used in the function.

Example:

```
> ssiPlot(x^2+y^2+z^2 -5, x*y*z -1,[x,y,z],color=green);
```



See also: `pacPlot`, `plotAlgSet`

Function: plotAlgSet

Action: plotting algebraic set

Calling sequence:

`plotAlgSet(`*a*`)`
`plotAlgSet(`*a*, *options*`)`

Arguments:

a – algebraic set

Optional Arguments:

options – options to the function

Synopsis:

- The function plots the following algebraic sets: planar curves represented implicitly, in parametric form, or by places, space curves represented implicitly, in parametric form, in projected form, or by places, surfaces represented implicitly, or in parametric form.
  OPTIONS: Options are given as equations of the form *option* = *value*. The following options are supported:

	All options that are supported by Maple's plotting-functions (see `?plot[options]`, `?plot[device]`, `?plot[setup]`, etc. for the details).
x=-5..5	Specifies the range -5..5 for the variable x. For every variable occurring in the algebraic set the user must specify a range.
`order`=O	Option used for plotting places. Each formal powerseries will be developed into a power series up to order O (default is the global variable `Order`).
`style=point`	Always possible. Curves in implicit or projected representation are plotted in a few seconds.

Example:

```
> a1:=mkImplAlgSet([2*x^4-3*x^2*y+y^2*(y-1)^2],[x,y]):
> plotAlgSet(a1,x=-2..2,y=-1..3,numpoints=200);
> a2:=convertRep(a1,impl,para,'t'):
> plotAlgSet(a2,t=-10..10,scaling=constrained,axes=boxed);
> a3:=convertRep(a1,impl,plac,[0,0],'T'):
> plotAlgSet(a3,T=-1..1,order=6,title=`Powerseries expansion`);
```

See also: `ssiPlot`, `pacPlot`, `interface`

4.10 Auxiliary Functions

Function: casa/assign

Action: perform assignments

Calling sequence:

`` `casa/assign`( ``*a*, *b*`)`
`` `casa/assign`( ``*a* = *b*`)`
`` `casa/assign`( ``*l*`)`

Arguments:

a – a name
b – any expression
l – a list or set of equations

Synopsis:

- This function is equivalent with the function `assign` but it returns the value which was assigned.

Example:

```
> `casa/assign`(evaln(a),1);
```

```
                                    1

> a;
                                    1
```

See also: `assign`

Function: delete

Action: delete an element from a list or a set

Calling sequence:

`delete(`*e*`, `*l*`)`
`delete(`*e*`, `*l*`, all)`

Arguments:

e – expression
l – list or set

Optional Arguments:

`all` – string `all`

Synopsis:

- Given a list *l*, the function removes some or all occurences of the expression *e*.
- If `all` is specified as the third argument, all occurences of *e* are removed from *l*. Otherwise only the first occurence is deleted.
- The result is a sublist of *l* with some or all occurences of *e* removed.

Example:

```
> delete(x,[x,y,z,x]);

                              [y, z, x]

> delete(x,[x,y,z,x],all);

                               [y, z]
```

Function: homogenize

Action: homogenization of a polynomial

Calling sequence:

`homogenize(`*poly*`, `*vars*`)`
`homogenize(`*poly*`, `*vars*`, `*v*`)`
`homogenize(`*poly*`, `*vars*`, `*v*`, `*deg*`)`

Arguments:

poly – polynomial in *vars*
vars – list of variables

Optional Arguments:

v – new variable
deg – homogeneous degree

Synopsis:

- The call homogenize(*poly*, *vars*) expects a bivariate polynomial in *vars*. It returns a list $[F, V]$ where F is homogeneous polynomial in V.
- If the optional third parameter is specified, it is interpreted as the name of the homogenizing variable. The optional fourth parameter specifies the degree of the resulting homogeneous polynomial (`degree(`*poly*`,` *vars*`)` is taken instead).

Example:

```
> homogenize(x^2+3*x^2*y+y,[x,y],z);

                          2         2         2
                        [x  z + 3 x  y + y z , [x, y, z]]
```

See also: `degree`, `type/polynom`

Appendix A

Writing Documentation

A.1 Introduction

The CASA system contains programs supplying it with a capability of creating both printed manual and web page from one source written in LaTeX format. The creation of both types of documentation is done in two steps. First, all source files are run through a preprocessor which creates a suitable structure for the particular type of document. Second, the dvi-source for the printed manual or the web page are created. For creating CASA's web page, we just need to run `hyperlatex casa` from the directory of CASA's homepage.

A.2 Source File Style Description

Generally, the CASA manual consists of a *root file* which contains the `\documentclass` macro and describes the environment in which the whole manual is typeset, and a number of files each describing one function.

A.2.1 Root File

The name of the root file may be chosen arbitrarily (default is *manual.tex*). Due to technical reasons each function description has to be included in the root file via `\input` macro, each on a separate line. Moreover, only those `\input` macros are significant where the `\input` keyword if followed by a blank (e.g. `\input␣{proc}`). This is to distinguish between `\input` macros used as "control sequences" and macros used to include other files into the manual. Commenting out such a line causes the particular function not to be included in the printed manual *and* also not to be included in the process of creation of the web page. Hence the content of the web page will mirror the printed manual. Here is a small example:

```
\documentclass[]{manual}
\usepackage{makeidx,latexsym,a4,amssymb,graphicx,mapleenv,fancybox,
hyperlatex,CASAman}
\usepackage[german,american]{babel}
...
\title{CASA Reference Manual}
...
\begin{document}
...
\section{CASA package}
\input{casa}
```

```
\section{Creator and Selector Functions for Algebraic Sets}
\input{addattr}
\input{attributes}
\end{document}
```

A.2.2 Function Description

Now, we give the format of source files to describe each function characterized by following items:

- function-name
- description of the action it performs
- calling sequence
- mandatory parameters
- optional parameters
- synopsis
- example
- references to other functions.

The way how there items are formatted is defined in the LaTeX-style file *CASAman.sty*.

The main part is the macro `\procheader` having six parameters (the first six items from above). Example and references to other functions are handled separately. The overall structure of `procheader` is as follows:

`\procheader`{*function-name*}
{*description of the action it performs*}
{*calling sequence*}
{*mandatory parameters*}
{*optional parameters*}
{*synopsis*}.

Function Name

The function name parameter may (should) contain an index entry with syntax required by program *makeindex* (not relevant for Maple help). It has the form:

```
<function name> [<index entry>]
```

Example:

```
mysum\index{summation!mysum@{\tt mysum}}.
```

Action

The action is described by ordinary text having no special format.

Calling Sequence

If there is just one way to invoke the function the calling sequence has the form:

```
<function name>([<parameter list>])
```

Example:

```
sum(\var{f},\var{range}).
```

The macro `\var` simply emphasizes the parameter text (in Maple help files the text is enclosed in quotes).

In case of multiple possibilities for invocation there is an environment `cs` which cares for formatting. It has the same syntax as the `itemize`-environment:

```
\begin{cs}
  \item <calling sequence 1>
  \item <calling sequence 2>
  ...
\end{cs}
```

Example:

```
\begin{cs}
  \item mysum(\var{f})
  \item mysum(\var{f},\var{range})
\end{cs}
```

Parameters

If there is just one parameter the description has the form:

```
<variable> -- description
```

e.g.:

```
\var{f} -- function
```

The variable and its description has to be separated by two dashes (`--`).

In case of more variables these should be enclosed in the `pars`-environment:

```
\begin{pars}
  \item <variable 1> -- description
  \item <variable 2> -- description
  ...
\end{pars}
```

Example:

```
\begin{pars}
  \item \var{f} -- function
  \item \var{range} -- range of values
\end{pars}
```

Remark:

- The string `--` is allowed to appear only for the purpose of separating parameters and their description. Its occurrentce within the verbal description of parameters will cause a loop during the parse process.
- The whole description of one parameter has to appear on a single line. Hence after a newline character the first word must be `\item`.

Optional Parameters

Optional parameters are specified in the same way as mandatory parameters.

Synopsis

The synopsis serves the purpose of detailed description of the function. It is divided into items. Each of them starts with `\item`. Each item may contain following types of text:

- text which is included in both printed version and web page;
- text to be included only in the printed manual (will be formatted by LaTeX);
- text to be included only in the web page.

The reason for this distinction is to enable both high quality typesetting of formulas in the printed manual and low quality typesetting of formulas in the web page.

The text to be included only in the printed manual has to be enclosed in the `latexonly`-environment. In the preprocessing step, anything between lines containing `\begin{latexonly}` and `\end{latexonly}` is used as input into LaTeX.

The text to be included only in the web page has to be enclosed in the `mapleonly`-environment. All lines between `\begin{mapleonly}` and `\end{mapleonly}` are put "verbatim" in the resulting Maple help file (without additional formatting).

All three types of text may appear arbitrarily often and in any order in each item of the synopsis. Hence the sixth parameter of `\procheader` has the form:

```
\item {[<plain text>][<latexonly text>][<mapleonly text>]}
\item {[<plain text>][<latexonly text>][<mapleonly text>]}
...
```

(braces denote repetition one or more times). **Remark:** Do not leave blank lines between items.

Example:

```
\item The function \fun{mysum} computes a symbolic solution of
  the summation  problem
  \begin{latexonly}
    \begin{displaymath}
      \sum_{i \in {\em range}}f(i).
    \end{displaymath}
  \end{latexonly}
  \begin{mapleonly}
                        ---
                        \
                        /    f(i)
                        ---
                      i \in range
  \end{mapleonly}
  ...
```

Example

Examples are to be put after the `\procheader` macro. It is enclosed in the `example`-environment. The content of this environment is formatted as Maple code as described in *maple.sty*.

Example:

```
\begin{example}
> mysum(i^2,1..n);

                                3               2
                    1/3 (n + 1)  - 1/2 (n + 1)  + 1/6 n + 1/6

\end{example}
```

References to Other Functions

References to other functions are given as arguments of the macro `\seealso` which usually appears after examples.

Example:

```
\seealso{sum, int}
```

Important Remarks

- Since many environments in the root file are changed to other structures, all enclosing `\begin` and `\end` macros have to be on separate lines and no other text (including braces) should be neither before nor behind them on the same line.
- Optional parameters and synopsis may be empty.

Example of Function Description

The following is an example for the description of a function. It is supposed to be saved in a file with `tex`-extension and included via `\input`-macro in the root file.

```
\procheader{mysum\index{summation!mysum@{\tt mysum}}}
{closed form of a summation problem}{
\begin{cs}
  \item mysum(\var{f})
  \item mysum(\var{f},\var{range})
\end{cs}
}{
\begin{pars}
  \item \var{f} -- function
  \item \var{range} -- range of values
\end{pars}
}{}{
\item The function \var{mysum} computes a symbolic solution of
  the summation  problem
  \begin{latexonly}
    \begin{displaymath}
      \sum_{i \in {\em range}}f(i).
    \end{displaymath}
  \end{latexonly}
  \begin{mapleonly}
                        ---
                        \
                        /    f(i)
```

```
                            ---
                          i \in range
  \end{mapleonly}
}
\begin{example}
> mysum(i^2,1..n);

                                 3                2
                   1/3 (n + 1)  - 1/2 (n + 1)  + 1/6 n + 1/6

>
\end{example}
\seealso{sum, int}
```

A.2.3 Help Pages

For creating help pages in CASA, the worksheets in `help/casa/mws` are needed and the commands in `help.maple` are invoked from a Maple worksheet.

Bibliography

[1] R. Gebauer, M. Kalkbrener, B. Wall, and F. Winkler. CASA: A Computer Algebra Package for Constructive Algebraic Geometry. In S. M. Watt, editor, *ISSAC '91*, pages 403–410, Bonn, Germany, July 1991.

[2] M. Mňuk. *Algebraic and Geometric Approach to Parameterization of Rational Curves.* PhD thesis, Research Institute for Symbolic Computation, Linz, Austria, 1995.

[3] M. Mňuk and F. Winkler. CASA - a system for computer aided constructive algebraic geometry. In *Proceedings of the International Symposium DISCO'96*, pages 297–307, 1996.

[4] J. R. Sendra and F. Winkler. Parametrization of algebraic curves over optimal field extensions. *J. Symb. Comput.*, 23:191–207, 1997.

[5] Q.-N. Tran. A hybrid symbolic-numerical method for tracing surface-to-surface intersections. In *The proceedings of ISSAC'95*, pages 51–58, 1995.

[6] Q.-N. Tran. On the symbolic-numerical methods for finding the roots of an arbitrary system of non-linear algebraic equations. In *Proceedings of the First Asian Technology Conference in Mathematics, ATCM-95*, Singapore, 1995.

[7] Q.-N. Tran. *A Hybrid Symbolic-Numerical Approach in Computer Aided Geometric Design (CAGD) and Visualization.* PhD thesis, Research Institute for Symbolic Computation (RISC–Linz), University of Linz, Austria, 1996.

[8] Q.-N. Tran. Extended Dixon's resultant and its applications. In D. Wang, editor, *Automated Deduction in Geometry.* 1997. To appear.

[9] Q.-N. Tran. Extending Newton's method for finding the roots of an arbitrary system of equations and its applications. *International Journal of Modeling and Simulation*, 17(4), 1997. To appear.

[10] Q.-N. Tran and F. Winkler. An overview of CASA – a system for computational algebra and constructive algebraic geometry. In T. R. T. V. Effelterre and F. Winkler, editors, *The Symbolic and Algebraic Computation (SAC) Newsletter*, volume 2. Stichting CAN, Computer Algebra Nederland, Universiteit van Amsterdam, the Netherlands, 1997.

[11] B. L. van der Waerden. *Moderne Algebra*, volume 2. Springer Verlag, Berlin, second edition, 1940.

[12] R. J. Walker. *Algebraic Curves.* Springer–Verlag, second edition, 1978.

[13] B. Wall. *Symbolic Computation with Algebraic Sets.* PhD thesis, RISC-Linz, Universität Linz, 1993.

[14] F. Winkler. Algebraic computation in geometry. In G. Jacobs, N. E. Oussous, and S. Steinberg, editors, *IMACS'93 Conference, Lille, special issue J. Math. in Comp. Simulation*, 1993.

Index