Dilla University

Department of Mathematics

Topics In Algebra I Lecture Note

by

Dereje Kifle (PhD)

Contents

Chapter 1

Introduction

1.1 What is Computer Algebra?

Mathematicians in the old period, say before 1850 A.D., solved the majority of mathematical problems by extensive calculations. A typical example of this type of mathematical problem solver is Euler. So it is not astonishing that in the 18th and beginning 19th centuries many mathematicians were real wizards of computation. However, during the 19th century the style of mathematical research changed from quantitative to qualitative aspects. A number of reasons were responsible for this change, among them the importance of providing a sound basis for the vast theory of analysis. But the fact that computations gradually became more and more complicated certainly also played its role. This impediment has been removed by the advent of modern digital computers in general and by the development of program systems in computer algebra in particular. By the aid of computer algebra the capacity for mathematical problem solving has been decisively improved.

Even in our days many mathematicians think that there is a natural division of labor between man and computer: a person applies the appropriate algebraic transformations to the problem at hand and finally arrives at a program which then can be left to a "number crunching" computer. But already in 1844 Lady Augusta Ada Byron, countess Lovelace, recognized that this division of labor is not inherent in mathematical problem solving and may be even detrimental.

A modern digital computer is a "universal" machine capable of carrying out an arbitrary algorithm, i.e. an exactly specified procedure, algebraic algorithms being no exceptions.

Now what exactly is symbolic algebraic computation, or in other words computer algebra? In his introduction to (Buchberger et al. 1983) R. Loos gave the following attempt at a definition:

Computer algebra is that part of computer science which designs, analyzes, implements, and applies algebraic algorithms.

While it is arguable whether computer algebra is part of computer science or mathematics, we certainly agree with the rest of the statement. In fact, in our view computer algebra is a special form of scientific computation, and it comprises a wide range of basic goals, methods, and applications. More formally,

Definition 1.1.1. *Computer Algebra* is a discipline between mathematics and computer science which deals with designing, analyzing, implementing, and applying algebraic algorithms.

In contrast to numerical computation the emphasis is on computing with symbols representing mathematical concepts. Of course that does not mean that computer algebra is devoid of computations with numbers. Decimal or other positional representations of integers, rational numbers and the like appear in any symbolic computation. But integers or real numbers are not the sole objects. In addition to these basic numerical entities computer algebra deals with polynomials, rational functions, trigonometric functions, algebraic numbers, etc. That does not mean that we will not need numerical algorithms any more. Both forms of scientific computation have their merits and they should be combined in a computational environment. For instance, in order to compute an approximate solution to a differential equation it might be reasonable to determine the first n terms of a power series solution by exact methods from computer algebra before handing these terms over to a numerical package for evaluating the power series.

Summarizing, computer algebra has two fundamental goals: Provide algorithms for computations with algebraic structures, like fields, vector spaces, rings, ideals, and modules to the computer. And use the algorithms and their implementations to solve mathematical problems in theory and applications. Here, computations usually refer to exact, that is, symbolic ones. However, in some cases, numerical computations can be helpful in obtaining exact results.

1.2 Application Areas of Computer Algebra

Computer algebra is interdisciplinary in nature, with links to quite a number of areas in mathematics, with applications in mathematics, other branches of science, and engineering:

- Through computer algebra methods, a number of mathematical disciplines become accessible to experiments. This is in particular true for various parts of algebra, number theory, and geometry.

- Modern application areas of mathematics such as cryptography, coding theory, CAD, robotics, algebraic statistics, and algebraic biology heavily rely on computer algebra.

1.3 Why Should we use Computer Algebra?

Of course, there are practical problems, that can be solved by computer algebra, for example, in cryptography, robotics, algebraic statistics, computational biology, and

physics. On the other hand, experiments with the computer allow you to get an insight into theoretical problems and test conjectures. In many settings, you can even obtain theoretical results by handling just a single special case by computer.

As a consequence one can build decision algorithms on computer algebra, e.g. for the factorizability of polynomials, the equivalence of algebraic expressions, the solvability of integration problems in a specific class of expressions, the solvability of certain differential equation problems, the solvability of systems of algebraic equations, the validity of geometric statements, the parametrizability of algebraic curves.

What is an Algorithm?

An *Algorithm* is a set of instructions for solving a particular problem in *finitely many, well-defined steps*. Starting from a given *input*, the instructions describe a computation which eventually will produce an *output* and *terminate*. The transition from one step to the next one is not necessarily *deterministic*: *probabilistic algorithms* incorporate random input, which may lead to random performance and random output. For example,

Algorithm 1.1 `Sample Algorithm`

Input: some input.

Output: some output m.

instruction

What are Algebraic Algorithms?

Algebraic algorithms deal with algebraic objects, make us of algebraic methods, and are based on algebraic theorems. Objects are represented exactly and calculations are carried through exactly (no approximation is applied at any step).

Analyzing Algorithms

One way of measuring the efficiency of an algorithm is to give asymptotic bounds on its running time which depend on the size of the input.

Designing Algorithms

When designing algorithms, we will describe them in a somewhat informal way which makes use of the structural conventions of a programming language. We refer to such a description as *pseudocode*, see the above algorithm.

Implementations

There is a large variety of computer algebra systems suiting different needs. Some of the most widely used systems are Mathematica, Maple, Derive, Reduce, SINGULAR, MAXIMA, MAGMA, COCOA, GAP, JULIA, SAGE and MATLAB. Among these computer algebra systems, in this lecture, we work in this lecture with SINGULAR, MAXIMA, GAP, JULIA and SAGE which are open computer algebra systems, that is, they can be downloaded for free from internet.

1.4 Numbers

One of the most important algorithms in mathematics is Euclidean algorithm for finding the greatest common divisor. In a generalized form, it will be presented explicitly or implicitly in many algorithms we will discuss later on.

Definition 1.4.1. For integers a and b, $b \neq 0$, b is called a *divisor* of a, if there exists an integer c such that $a = bc$.

We denote by $b \mid a$ if b is a divisor of a and by $b \nmid a$ if it is not.

Lemma 1.4.2 (**Division with Remainder**). *For $a, b \in \mathbb{Z}$, $b \neq 0$, there are $q, r \in \mathbb{Z}$ with $a = b \cdot q + r$ and $0 \leq r < |b|$.*

Proof. Without loss of generality $b > 0$. The set

$$\{w \in \mathbb{Z} \mid b \cdot w > a\} \neq \emptyset$$

has a smallest element w. Then set $q := w - 1$ and $r := a - qb$. □

Definition 1.4.3. An integral domain R together with a function $d : R \to \mathbb{N} \cup \{\infty\}$ is a *Euclidean domain* if for all $a, b \in R$ with $b \neq 0$, we can divide a by b with remainder, so that there exist $q, r \in R$ such that $a = qb + r$ and $d(r) < d(b)$. We say that $q = a \operatorname{quo} b$ is the *quotient* and $r = a \operatorname{rem} b$ the *remainder*, although q and r need to be unique. Such a d is called a *Euclidean function* on R.

Example 1.4.4.

(i) The function $d : \mathbb{Z} \to \mathbb{N} \cup \{-\infty\}$ defined by $d(a) = |a|$ is an Euclidean function. Here the quotient and the remainder can be made unique with additional requirement that $r \geq 0$.

(ii) The function $d : F[x] \to \mathbb{N} \cup \{-\infty\}$ defined by $d(a) = \deg a$ is an Euclidean function. Here the quotient and the remainder are unique without any further requirement.

Definition 1.4.5. Let R be a ring and $a, b, c \in R$. Then

(1) c is a *greatest common divisor (or gcd)* of a and b if

i) $c|a$ and $c|b$,

ii) if $d|a$ and $d|b$, then $d|c$ for all $d \in R$.

(2) c is called a *least common multiple* of a and b if

i) $a|c$ and $b|c$,

ii) if $a|d$ and $b|d$, then $c|d$ for all $d \in R$.

(3) A *unit* $u \in R$ is any element with a multiplicative inverse $v \in R$, that is, $uv = 1$.

(4) The elements a and b are *associate*, denoted as $a \sim b$, if $a = ub$ for a unit $u \in R$

Remark 1.4.6.

- Neither the gcd nor the lcm are unique, but all gcds of a and b are precisely the associates of one of them and so is for the lcm. For example, 3 and -3 are all gcds of 12 and 15 in $\mathbb{Z}$ because 1 and -1 are the only units in $\mathbb{Z}$.

- For $R = \mathbb{Z}$, we may define $\gcd(a, b)$ as the unique nonnegative greatest common divisor and $\operatorname{lcm}(a, b)$ as the unique nonnegative least common multiple of a and b.

Remark 1.4.7. Let R be an integral domain and $a, b \in R$ such that $\gcd(a, b) = c$ exists. Clearly, all such divisors are obtained by multiplying c with a unit of R. In other words, the gcd's form an equivalence class under being associated. In this lecture, we always assume that in each such equivalence class a *normal form* is d selected. If the class is represented by $a \in R$, we write $N(a)$ for the normal form. Here N is defined as follows:

$$N(a) := \begin{cases} 0 & \text{if } a = 0 \\ 1 & \text{if } a = 1 \\ a/U(a) & \text{otherwise} \end{cases}$$

where $U(a)$ is called the *leading unit* of $a \in R$ such that $a \sim N(a)$, that is, $a = U(a)N(a)$. For $a = 0$, we set $U(a) = 1$.

Note that

- two elements of R have the same normal form if and only if they are associate, that is, $N(a) = N(b)$ iff $a \sim u \cdot b$ for some unit $u \in R$.

- the normal form of a product is equal to the products of the normal forms, that is, $N(a \cdot b) = N(a) \cdot N(b)$.

Example 1.4.8.

i) If $R = \mathbb{Z}$, $U(a) = \text{sign}(a)$ if $a \neq 0$ and $N(a) = |\, a \,|$ defines a normal form, so that an integer is normalized if and only if it is nonnegative.

ii) If $R = F[x]$ for a field F, then letting $U(a) = \text{lc}(a)$ (with the convention that $U(0) = 1$) and $N(a) = a/\text{U(a)}$ defines a normal form, and a nonzero polynomial is normalized if and only if it is monic.

Theorem 1.4.9 (**Euclidean Algorithm**). *Suppose* $a_1, a_2 \in \mathbb{Z}\backslash\{0\}$. *Successive division with remainder terminates*

$$\begin{aligned} a_1 &= q_1 a_2 + a_3 \\ &\vdots \\ a_j &= q_j a_{j+1} + a_{j+2} \\ &\vdots \\ a_{n-2} &= q_{n-2} a_{n-1} + a_n \\ a_{n-1} &= q_{n-1} a_n + 0 \end{aligned}$$

and

$$\gcd(a_1, a_2) = a_n \,.$$

Reading the equation backwards

$$\begin{aligned} a_n &= a_{n-2} - q_{n-2} a_{n-1} \\ &\vdots \\ a_3 &= a_1 - q_1 a_2 \end{aligned}$$

gives a representation

$$\gcd(a_1, a_2) = x \cdot a_1 + y \cdot a_2$$

with $x, y \in \mathbb{Z}$.

Proof. We have $|a_{i+1}| < |a_i|$ for $i \geq 2$ so after finitely many steps $a_i = 0$. Then a_n divides a_{n-1}, hence also $a_{n-2} = q_{n-2} a_{n-1} + a_n$ and inductively $a_{n-2}, \ldots, a_1$. If t is a divisor of a_1 and a_2, then also of $a_3, \ldots, a_n$. □

Example 1.4.10. We compute the gcd of 36 and 15:

$$\begin{aligned} 36 &= 2 \cdot 15 + 6 \\ 15 &= 2 \cdot 6 + 3 \\ 6 &= 2 \cdot 3 + 0 \end{aligned}$$

hence $\gcd(36, 15) = 3$. Furthermore, we can express $\gcd(36, 15)$ as a $\mathbb{Z}$ linear combination of 36 and 15:

$$3 = 15 - 2 \cdot 6 = 15 - 2 \cdot (36 - 2 \cdot 15) = 5 \cdot 15 + (-2) \cdot 36 \,.$$

Given a normal form, we define $\gcd(a, b)$ to be the unique normalized associate of all greatest common divisors of a and b, and similarly $\operatorname{lcm}(a, b)$ as the normalized associate of all least common multiples of a and b. Thus $\gcd(a, b) > 0$ for $R = \mathbb{Z}$ and $\gcd(a, b)$ is monic for $R = F[x]$ if at least one of a, b is nonzero, and $\gcd(0, 0) = 0$ in both cases.

The Euclidean algorithm in Theorem 2.3 can easily be summarized in pseudocode form as follows:

Algorithm 1.2 `Euclid's Algorithm for integers`

Input: $m, n \in \mathbb{Z}$.

Output: $\gcd(m, n)$.

1: $a := n, b := m$
2: **while** $b \neq 0$ **do**
3: $\quad r := a \operatorname{rem} b$// division with remainder
4: $\quad a := b, b := r$
5: $a := n(a)$// normal form
6: **return** a

Definition 1.4.11. An element $p \in \mathbb{Z}_{>1}$ is called *prime number*, if $p = a \cdot b, a, b \in \mathbb{Z}_{\geq 1}$ implies $a = 1$ or $b = 1$ and we call p a *composite number* if it is not prime.

Theorem 1.4.12 (**The Fundamental Theorem of Arithmetic**). *Every integer* $n \in \mathbb{Z} \setminus \{-1, 0, 1\}$ *has a unique representation may be expressed uniquely in the form*

$$n = \pm \prod_{i=1}^{k} p_i^{\alpha_i}$$

with ***prime factors*** $p_1 < \ldots < p_k$ *and* $\alpha_i \in \mathbb{N}$.

Algorithm 1.4.13. Let $n \in \mathbb{Z}$ be composite. The smallest prime factor p of n satisfies

$$p \leq m := \lfloor \sqrt{n} \rfloor \,.$$

If we know all primes $p \leq m$, then we can test $p|n$ by division with remainder and, hence, factor n.

Note that $\lfloor x \rfloor = \max\{a \in \mathbb{Z} \mid a \leq x, x \in \mathbb{R}\}$, the largest integer less than or equal to x. The function $\lfloor x \rfloor$ is called the *floor* function of x.

Algorithm 1.4.14 (Sieve of Eratosthenes). We can find all prime numbers smaller than n in the following way: Note all numbers from 2 to n. Starting with $p = 2$, delete all $a \cdot p$ for $a > 1$, and continue with the next largest number p which not has been deleted. Note that p is prime, since it is not a multiple of smaller prime. Stop if $p > \sqrt{n}$.

Example 1.4.15. We compute all primes ≤ 21:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
2	3		5		7		9		11		13		15		17		19		21
2	3		5		7				11		13				17		19		

In the first step we delete all multiple of 2, in the second step all multiple of 3. All remaining numbers are prime, since $5 > \sqrt{21}$.

One can even describe the distribution of the primes over all integers:

Theorem 1.4.16 (Prime Number Theorem). *For $x \in \mathbb{R}_{>0}$ let*

$$\pi(x) = |\{p \leq x \mid p \in \mathbb{N} \text{ prime }\}|\,.$$

Then

$$\lim_{x \to \infty} \frac{\pi(x)}{\frac{x}{\ln(x)}} = 1\,.$$

Example 1.4.17. The number of prime ≤ 21 is $\pi(21) = 8$, see Example 1.4.15.

Computer algebra in this spirit has many theoretical applications in number theory and algebraic geometry and practical applications, for example, in coding theory or RSA public key cryptography.

In general, number theory explores the properties of numbers, most importantly the interaction of addition and multiplication. This leads to many problems which are easy to formulate, but highly non-trivial to solve. The most famous one is Fermat's last theorem of 1637: There is no (non-trivial) integer solution of the equation

$$x^n + y^n = z^n$$

for $n \geq 3$. With the help of a computer one can test Fermat's last theorem for very large n (using the theoretical result that you only have to test it for so called irregular primes). Fermat's last theorem was finally proven in 1995 (by A. Wiles) after 350 years of work of many people, which led to many new concepts in mathematics.

Definition 1.4.18. The Fibonacci numbers are the sequence of numbers $\{F_n\}_{n=1}^{\infty}$ defined by the linear recurrence equation $F_n = F_{n-1} + F_{n-2}$ with $F_1 = F_2 = 1$.

How can I tell if a given number is a Fibonacci number? a is a Fibonacci number if and only if $5a^2 + 4$ or $5a^2 - 4$ is a square number.

1.5 Introduction to Programming in Singular

1.5.1 First Step

Once SINGULAR is started, it awaits an input after the prompt >. Every statement has to be terminated by ;.

```
23 + 5;
==> 28
```

All objects have a type, for example, integer variables are defined by the word `int` . An assignment is made using the symbol `=` .

```
int k = 5;
```

Test for equality resp. inequality is done using `==` resp. `!=` (or `<>`), where 0 represents the boolean value `FALSE` , and any other value represents `TRUE`.

```
k == 5;
==> 1
k !=5 ;
==> 0
```

The value of an object is displayed by simply typing its name.

```
k;
==> 5
```

On the other hand, the output is suppressed if an assignment is made.

```
int j = k+1;
==> 6
```

The last displayed (`!`) result can be retrieved via the special symbol `_` .

```
9 + _;// the value from k displayed above
==> 14
```

Text starting with // denotes a comment and is ignored in calculations, as seen in the previous example. Furthermore SINGULAR maintains a history of the previous lines of input, which may be accessed by CTRL-P (previous) and CTRL-N (next) or the arrows on the keyboard.

The whole manual is available online by typing the command help; . Documentation on single topics, for exmaple, on intmat , which defines a matrix of integers, is obtained by

```
help intmat;
```

This will display the text of intmat , in the printed manual.

Next, we define a 3×3 matrix of integers and initialize it with some values, row by row from left to right:

```
intmat m[3][3] = 1,2,3,4,5,6,7,8,9;
m;
```

A single matrix entry may be selected and changed using square brackets [and] .

```
m[1,2]=0;
m;
==> 1,0,3,
==> 4,5,6,
==> 7,8,9
```

To calculate the trace of this matrix, we use a for loop. The curly brackets { and } denote the beginning resp. end of a block. If you define a variable without giving an initial value, as the variable tr in the example below, SINGULAR assigns a default value for the specific type. In this case, the default value for integers is 0. Note that the integer variable j has already been defined above.

```
int tr;
for ( j=1; j <= 3; j++ ) { tr=tr + m[j,j]; }
tr;
==> 15
```

Variables of type string can also be defined and used without having an active ring. Strings are delimited by " (double quotes). They may be used to comment the output of a computation or to give it a nice format. If a string contains valid SINGULAR commands, it can be executed using the function execute. The result is the same as if the commands would have been written on the command line. This feature is especially useful to define new rings inside procedures.

```
"example for strings:";
==> example for strings:
string s="The element of m ";
s = s + "at position [2,3] is:";  // concatenation of strings by +
s , m[2,3] , ".";
==> The element of m at position [2,3] is: 6 .
s="m[2,1]=0; m;";
execute(s);
==> 1,0,3,
==> 0,5,6,
==> 7,8,9
```

This example shows that expressions can be separated by , (comma) giving a list of expressions. SINGULAR evaluates each expression in this list and prints all results separated by spaces.

1.5.2 Rings and standard bases

In order to compute with objects such as ideals, matrices, modules, and polynomial vectors, a ring has to be defined first.

```
ring r = 0,(x,y,z),dp;
```

The definition of a ring consists of three parts: the first part determines the *ground field*, the second part determines the names of the *ring variables*, and the third part determines the *monomial ordering* to be used. Thus, the above example declares a polynomial ring called `r` with a ground field of characteristic 0 (i.e., the rational numbers) and ring variables called `x`, `y`, and `z`. The `dp` at the end determines that the degree reverse lexicographical ordering will be used.

Other ring declarations:

```
ring r1 = 32003,(x,y,z),dp;
```

characteristic 32003, variables x, y, and z and ordering dp.

```
ring r2 = 32003,(a,b,c,d),lp;
```

characteristic 32003, variable names a, b, c, d and lexicographical ordering.

```
ring r3 = 7,(x(1..10)),ds;
```

characteristic 7, variable names $x(1), ..., x(10)$, negative degree reverse lexicographical ordering (ds).

```
ring r4 = (0,a),(mu,nu),lp;
```

transcendental extension of Q by a, variable names mu and nu, lexicographical ordering.

```
ring r5 = real,(a,b),lp;
```

floating point numbers (single machine precision), variable names a and b.

```
ring r6 = (real,50),(a,b),lp;
```

floating point numbers with precision extended to 50 digits, variable names a and b.

```
ring r7 = (complex,50,i),(a,b),lp;
```

complex floating point numbers with precision extended to 50 digits and imaginary unit i, variable names a and b.

```
ring r8 = integer,(a,b),lp;
```

the ring of integers (see Coefficient rings), variable names a and b.

```
ring r9 = (integer, 60),(a,b),lp;
```

the ring of integers modulo 60 (see Coefficient rings), variable names a and b.

```
ring r10=(integer, 2, 10),(a,b),lp;
```

the ring of integers modulo 2^{10} (see Coefficient rings), variable names a and b.

Typing the name of a ring prints its definition. The example below shows that the default ring in SINGULAR is $Z/32003[x, y, z]$ with degree reverse lexicographical ordering:

```
ring r11;
r11;
==> //   characteristic : 32003
==> //   number of vars : 3
==> //        block   1 : ordering dp
==> //                  : names    x y z
==> //        block   2 : ordering C
```

Defining a ring makes this ring the current active basering, so each ring definition above switches to a new basering. The concept of rings in SINGULAR is discussed in detail in Rings and orderings.

The basering is now `r11`. Since we want to calculate in the ring `r`, which we defined first, we need to switch back to it. This can be done using the function `setring`:

```
setring r;
```

Once a ring is active, we can define polynomials. A monomial, say x^3, may be entered in two ways: either using the power operator `^`, writing x^3 or in short-hand notation without operator, writing `x3`. Note that the short-hand notation is forbidden if a name of the ring variable(s) consists of more than one character(see Miscellaneous oddities for details). Note, that SINGULAR always expands brackets and automatically sorts the terms with respect to the monomial ordering of the basering.

```
poly f =  x3+y3+(x-y)*x2y2+z2;
f;
==> x3y2-x2y3+x3+y3+z2
```

The command `size` retrieves in general the number of entries in an object. In particular, for polynomials, `size` returns the number of monomials.

```
size(f);
==> 5
```

A natural question is to ask if a point, for example, $(x, y, z) = (1, 2, 0)$, lies on the variety defined by the polynomials f and g. For this we define an ideal generated by both polynomials, substitute the coordinates of the point for the ring variables, and check if the result is zero:

```
poly g =  f^2 *(2x-y);
ideal I = f,g;
ideal J = subst(I,var(1),1);
J = subst(J,var(2),2);
J = subst(J,var(3),0);
J;
==> J[1]=5
==> J[2]=0
```

Since the result is not zero, the point $(1, 2, 0)$ does not lie on the variety $V(f, g)$.

Another question is to decide whether some function vanishes on a variety, or in algebraic terms, if a polynomial is contained in a given ideal. For this we calculate a standard basis using the command `groebner` and afterwards reduce the polynomial with respect to this standard basis.

```
ideal sI = groebner(f);
reduce(g,sI);
==> 0
```

As the result is 0 the polynomial g belongs to the ideal defined by f.

The function `groebner`, like many other functions in SINGULAR, prints a protocol during calculations, if desired. The command `option(prot);` enables protocolling whereas `option(noprot);` turns it off. `option`, explains the meaning of the different symbols printed during calculations.

The command `kbase` calculates a basis of the polynomial ring modulo an ideal, if the quotient ring is finite dimensional. As an example we calculate the Milnor number of a hypersurface singularity in the global and local case. This is the vector space dimension of the polynomial ring modulo the Jacobian ideal in the global case resp. of the power series ring modulo the Jacobian ideal in the local case. See Critical points, for a detailed explanation.

The Jacobian ideal is obtained with the command `jacob`.

```
ideal J = jacob(f);
==> // ** redefining J **
J;
==> J[1]=3x2y2-2xy3+3x2
==> J[2]=2x3y-3x2y2+3y2
==> J[3]=2z
```

SINGULAR prints the line `// **` redefining `J **`. This indicates that we had previously defined a variable with name `J` of type ideal (see above).

To obtain a representing set of the quotient vector space we first calculate a standard basis, and then apply the function `kbase` to this standard basis.

```
J = groebner(J);
ideal K = kbase(J);
K;
==> K[1]=y4
==> K[2]=xy3
==> K[3]=y3
==> K[4]=xy2
==> K[5]=y2
==> K[6]=x2y
==> K[7]=xy
==> K[8]=y
==> K[9]=x3
==> K[10]=x2
==> K[11]=x
==> K[12]=1
```

Then

```
size(K);
==> 12
```

gives the desired vector space dimension $K[x,y,z]/\text{jacob}(f)$. As in SINGULAR the functions may take the input directly from earlier calculations, the whole sequence of commands may be written in one single statement.

```
size(kbase(groebner(jacob(f))));
==> 12
```

When we are not interested in a basis of the quotient vector space, but only in the resulting dimension we may even use the command `vdim` and write:

```
vdim(groebner(jacob(f)));
==> 12
```

1.5.3 Procedures and Libraries

SINGULAR offers a comfortable programming language, with a syntax close to `C`. So it is possible to define procedures which bind a sequence of several commands in a new one. Procedures are defined using the keyword `proc` followed by a name and an optional parameter list with specified types. Finally, a procedure may return a value using the command `return`.

We may e.g. define the following procedure called `Milnor`: (Here the parameter list is (`poly h`) meaning that `Milnor` must be called with one argument which can be assigned to the type poly and is referred to by the name `h`.)

Note: if you have entered the first line of the procedure and pressed `RETURN`, SINGULAR prints the prompt `.` (dot) instead of the usual prompt `>`. This shows that the input is incomplete and SINGULAR expects more lines. After typing the closing curly bracket, SINGULAR prints the usual prompt indicating that the input is now complete.

Then we can call the procedure:

```
Milnor(f);
==> 12
```

Note that the result may depend on the basering as we will see in the next chapter.

The distribution of SINGULAR contains several libraries, each of which is a collection of useful procedures based on the `kernel` commands, which extend the functionality of SINGULAR. The command `listvar(package);` list all currently loaded libraries. The command `LIB "all.lib";` loads all libraries.

One of these libraries is `sing.lib` which already contains a procedure called `milnor` to calculate the Milnor number not only for hypersurfaces but more generally for complete intersection singularities.

Libraries are loaded using the command `LIB`. Some additional information during the process of loading is displayed on the screen, which we omit here.

```
LIB "sing.lib";
```

As all input in SINGULAR is case sensitive, there is no conflict with the previously defined procedure Milnor, but the result is the same.

```
milnor(f);
==> 12
```

The procedures in a library have a `help` part which is displayed by typing

```
help milnor;
```

as well as some examples, which are executed by

```
example milnor;
```

Likewise, the library itself has a help part, to show a list of all the functions available for the user which are contained in the library.

```
help sing.lib;
```

The output of the help commands is omitted here.

The user may add their own commands to the commands already available in SINGULAR by writing SINGULAR procedures. There are basically two kinds of procedures:

- procedures written in the SINGULAR programming language (which are usually collected in SINGULAR libraries).
- procedures written in C/C++ (collected in dynamic modules).

At this point, we restrict ourselves to describing the first kind of (library) procedures, which are sufficient for most applications. The syntax and general structure of a library (procedure) is described in Procedures, and Libraries.

Procedures

Syntax:

```
[static] proc proc_name [(<parameter_list>)]
[<help_string>]
{
<procedure_body>
}
```

```
[example
{
<sequence_of_commands>
}]
```

Purpose:

- Defines a new function, the `proc proc_name`.
- The help string, the parameter list, and the example section are optional. They are, however, mandatory for the procedures listed in the header of a library. The help string is ignored and no example section is allowed if the procedure is defined interactively, i.e., if it is not loaded from a file by the LIB or load command (see LIB and see load).

Example of an interactive procedure definition and its execution:

```
proc factorialn(int n)
{
   if(n==0)
   {
      return(1); // 0! = 1
   }
   else
   {
         int k = 1;
         for(int i=0;i<=n;i++)
         {
            k = k*i;
         }
         return(k); // the value of k is returned
   }
}
factorialn(5);
==> 120
```

The probably most efficient way of writing a new library is to use one of the official SINGULAR libraries, say `ring.lib` as a sample. On a Unix-like operating system, type `LIB "ring.lib";` to get information on where the libraries are stored on your disk.

SINGULAR provides several commands and tools, which may be useful when writing a procedure, for instance, to have a look at intermediate results (see Debugging tools).

If such a libarary should be contributed to SINGULAR some formal requirements are needed:

the library header must explain the purpose of the library and (for non-trivial algorithm) a pointer to the algorithm (text book, article, etc.) all global procedures must have a help string and an example which shows its usage. it is strongly recommend also to provide test scripts which test the functionality: one should test the essential functionality of the library/command in a relatively short time (say, in no more than 30s), other tests should check the functionality of the library/command in detail so that, if possible, all relevant cases/results are tested. Nevertheless, such a test should not run longer than, say, 10 minutes.

Libraries

- A library is a collection of SINGULAR procedures in a file.
- To load a library into a SINGULAR session, use the `LIB` or load command. Having loaded a library, its procedures can be used like any built-in SINGULAR function, and information on the library is obtained by entering help `libname.lib;`
- See SINGULAR libraries, for all libraries currently distributed with SINGULAR.
- When writing your own library, it is important to comply with the guidelines described in this section. Otherwise, due to potential parser errors, it may not be possible to load the library.
- Each library consists of a header and a body. The first line of a library must start with a double slash `//`.
- The library header consists of a version string, a category string, an info string, and `LIB` commands. The strings are mandatory. `LIB` commands are meant to load the additional libraries used by the library under consideration.
- The library body collects the procedures (declared static or not).
- No line of a library should consist of more than 60 characters.

For a more detailed description, see www.singular.uni-kl.de.

Chapter 2

The Euclidean Algorithm

Integers and polynomials with coefficients in a field behave similarly in many aspects: Often but not always the algorithms for both types of objects are quite similar, and sometimes one can find a common abstraction of both domains. In this chapter, the Euclidean domain covers the structural similarities between gcd computations for integers and polynomials. Typically, in such a situation the polynomial version is slightly simpler. Note, however, that we have already seen that how to compute the gcd for integers. Nevertheless, we will see some of its properties.

2.1 The Extended Euclidean Algorithm

In this section, we discuss the extension of the Euclidean algorithm Algorithm 2.3 from Chapter 1. To begin with, we first start by some of the properties of the gcd in $\mathbb{Z}$:

Lemma 2.1.1. *The gcd in $\mathbb{Z}$ has the following properties, for all $a, b, c \in \mathbb{Z}$.*

i) $\gcd(a, b) = |a| \iff a \mid b$.

ii) $\gcd(a, a) = \gcd(a, 0) = |a|$ *and* $\gcd(a, 1) = 1$,

iii) $\gcd(a, b) = \gcd(b, a)$ *(commutativity)*,

iv) $\gcd(a, \gcd(b, c)) = \gcd(\gcd(a, b), c)$ *(associativity)*,

v) $\gcd(c \cdot a, c \cdot b) = |c| \cdot \gcd(a, b)$ *(distributivity)*,

vi) $|a| = |b| \to \gcd(a, c) = \gcd(b, c)$.

As aforementioned the following algorithm which is the extension of the traditional Euclidean algorithm, Algorithm 2.3, computes not only the gcd but also a representation of it as a linear combination of the inputs. It generalizes the representation

$$3 = 15 - 2 \cdot 6 = 15 - 2 \cdot (36 - 2 \cdot 15) = 5 \cdot 15 + (-2) \cdot 36$$

as described in Theorem 1.4.9 from the bottom up. This important method is called the *Extended Euclidean Algorithm (EEA)* and works in any Euclidean domain.

Algorithm 2.3 `Traditional Extended Euclidean Algorithm (TEEA)`

Input: $f, g \in R$, where R is a Euclidean domain.

Output: $l \in \mathbb{N}, r_i, s_i, t_i \in R$ for $0 \leq i \leq l+1$, and $q_i \in R$ for $1 \leq i \leq l$, as computed below.

1: $r_0 := f, s_0 := 1, t_0 := 0,$
2: $r_1 := g, s_1 := 0, t_1 := 1$
3: **while** $r_i \neq 0$ **do**
4: $\quad q_i := r_{i-1} \operatorname{qou} r_i$// division with remainder
5: $\quad r_{i+1} := r_{i-1} - q_i r_i$
6: $\quad s_{i+1} := s_{i-1} - q_i s_i$
7: $\quad t_{i+1} := t_{i-1} - q_i t_i$
8: $\quad i := i + 1$
9: $l := i - 1;$
10: **return** l, r_i, s_i, t_i for $0 \leq i \leq l+1$, and q_i for $1 \leq i \leq l$

Note that:

- the algorithm terminates because the $d(r_i)$ are strictly decreasing non-negative integers for $1 \leq i \leq l$, where d is the Euclidean function on R.
- The elements r_i for $1 \leq i \leq l+1$ are the remainders and the q_i for $1 \leq i \leq l$ are the quotients in the EEA.
- In EEA, the elements r_i, s_i, and t_i form the i-th row in the TEEA, for $1 \leq i \leq l+1$. The central property is that $s_i f + t_i g = r_i$ for all i; in particular, $s_l f + t_l g = r_l$ is a gcd of f and g.

Example 2.1.2.

a) Consider the ring $R = \mathbb{Z}$, and $f = 126$ and $g = 35$. The following table illustrates the computation.

i	q_i	r_i	s_i	t_i
0		126	1	0
1	3	35	0	1
2	1	21	1	-3
3	1	14	-1	4
4	2	7	2	-7
5		0	-5	18

From this we can read off row 4 that $\gcd(126, 35) = 7 = 2 \cdot 126 + (-7) \cdot 35$.

b) Consider the ring $R = \mathbb{Q}[x]$, and the polynomials $f = 18x^3 - 42x^2 + 30x - 6, g = -12x^2 + 10x - 2 \in R$. Compute a gcd of f and g using the TEEA. The TEEA applied to f and g goes as follows: Row $i + 1$ is obtained from the two preceding ones by first computing the quotient $q_i = r_{i-1} \operatorname{quo} r_i$ and then for each of the three remaining columns by subtracting the quotient times the entry in row i of that column from the entry in row $i - 1$.

i	q_i	r_i	s_i	t_i
0		$18x^3 - 42x^2 + 30x - 6$	1	0
1	$-\frac{3}{2}x + \frac{9}{4}$	$-12x^2 + 10x - 2$	0	1
2	$-\frac{8}{3}x + \frac{4}{3}$	$\frac{9}{2}x - \frac{3}{2}$	1	$\frac{3}{2}x - \frac{9}{4}$
3		0	$\frac{8}{3}x - \frac{4}{3}$	$4x^2 - 8x + 4$

From this table, we have $l = 2$, and from row 2, we find that a gcd of f and g is

$$\frac{9}{2}x - \frac{3}{2} = 1 \cdot f + \left(\frac{3}{2}x - \frac{9}{4}\right) g\,.$$

From a global view of the algorithm, it is convenient to consider the matrices

$$R_0 = \begin{pmatrix} s_0 & t_0 \\ s_1 & t_1 \end{pmatrix}, Q_i = \begin{pmatrix} 0 & 1 \\ 1 & -q_i \end{pmatrix} \text{ for } 1 \leq i \leq l.$$

in $R^{2\times 2}$, and $R_i = Q_i \cdots Q_1 R_0$ for $0 \leq i \leq l$.

Invariants of the Traditional EEA

The following lemma collects some invariants of the TEEA.

Lemma 2.1.3. *For $0 \leq i \leq l$, we have*

i) $R_i \begin{pmatrix} f \\ g \end{pmatrix} = \begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix}$,

ii) $R_i = \begin{pmatrix} s_i & t_i \\ s_{i+1} & t_{i+1} \end{pmatrix}$,

iii) $\gcd(f, g) \sim \gcd(r_i, r_{i+1}) \sim r_l$,

iv) $s_i f + t_i g = r_i$ *(this also holds for $i = l + 1$,*

v) $s_i t_{i+1} - t_i s_{i+1} = (-1)^i$,

vi) $\gcd(r_i, t_i) \sim \gcd(f, t_i)$,

vii) $f = (-1)^i(t_{i+1}r_i - t_i r_{i+1})$, $g = (-1)^{i+1}(s_{i+1}r_i - s_i r_{i+1}$ *with the convention that* $r_{l+1} = 0$.