Dilla University

Department of Mathematics

Topics In Algebra I

by

Dereje Kifle

Contents

Chapter 1

Introduction to Computer Algebra

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, computer-aided-design (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. Let us say, we want to prove that the determinant of the matrix

$$A_t = \begin{pmatrix} t-1 & 1 & -1 \\ t & t^2+1 & t+1 \\ t & t^2 & t+2 \end{pmatrix} \in \mathbb{C}[t]^{3\times 3}$$

is non-zero as a polynomial without computing it. For example, the determinant may be too complicated (which is of course not the case in the example). However, it may be possible to compute A_{t_0} for a fixed $t_0 \in \mathbb{C}$. For any $t_0 \in \mathbb{C}$, consider the substitution map

$$\begin{aligned} \varphi : \mathbb{C}[t]^{3\times 3} &\to \mathbb{C}^{3\times 3} \\ A_t &\mapsto \varphi(A_t) := A_{t_0} \end{aligned}$$

Since substitution is a ring homomorphism, it is sufficient to find one t_0 such that $\det A_{t_0} \neq 0$, for example,

$$\det A_{t_0} = \det \begin{pmatrix} -1 & 1 & -1 \\ 0 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} = -2\,.$$

It follows that the determinant, as a continuous function, will be non-zero in an open neighbourhood of t_0. In fact, we then know that it is non-zero for all but finitely many values of t, since $0 \neq \det A_t \in \mathbb{C}[t]$ has only finitely many zeros. This means that the determinant is non-zero on an open set in the so called Zariski topology, that is, on the complement of the zero set of a system of polynomial equations.

In the line of this example, our main focus will be on computations in commutative algebra, specifically on all sorts of algorithms concerned with polynomial rings. Here, the fundamental building block is Buchberger's algorithm for computing Gröbner bases, which generalizes Gaussian elimination. Recall that Gaussian elimination transforms multivariate linear systems of equations into row echelon form

$$\begin{matrix} 2x+y=1 \\ 2x+y=1 \end{matrix} \;\mapsto\; \begin{matrix} x+2y & = -1 \\ -3x & = -3 \end{matrix}$$

from which we can read off the solution $(x,y) = (1,-1)$.

Buchberger's algorithm generalizes this idea to higher degree polynomial equations, for example, it transforms

$$\begin{matrix} 2x^2 - xy + 2y^2 - 2 = 0 \\ 2x^2 - 3xy + 3y^2 - 2 = 0 \end{matrix} \;\mapsto\; \begin{matrix} 3y + 8x^3 - 8x = 0 \\ 4x^4 - 5x^2 + 1 = 0 \end{matrix}$$

Example 1.3.1. We can do the above Gröbner basis calculation in SINGULAR, see Section 1.5, by the following code:

```
ring R = 0, (y,x), lp;
ideal I = 2x2-xy+2y2-2, 2x2-3xy+3y2-2;
std(I);
  _[1] = 4x4-5x2+1
  _[2] = 3y+8x3-8x
```

The ring definition specifies the characteristic of the prime field (so 0 corresponds to $\mathbb{Q}$), the variables, and an ordering of the variables (`lp`). To make an analogy to Gaussian elimination, by the ordering you can tell the system, in which order you want to eliminate the variables (for example, if you want a right or left upper triangular matrix as row echelon form). An ideal represents a system of polynomial equations, and `std` refers to the term standard basis, which, in the setup considered here, is synonymous to Gröbner basis. From the resulting system we can read off the four solutions

$$(x, y) = (\pm 1, 0), \left(\pm\frac{1}{2}, \pm 1\right) .$$

The graphs of the above functions will motivate the algebraic concepts by connecting multivariate systems of polynomial equations to the geometry of their set of solutions, the associated algebraic variety. This connection is called *algebraic geometry*, an important branch of mathematics and one of the key applications of commutative algebra. In Chapter 2, we will give a more detailed explanation about the notion of Gröbner bases. The following definitions are also helpful to understand this notion.

What is an Algorithm?

Definition 1.3.2. An *algorithm* is a set of instructions for solving a particular problem in *finitely many, well-defined steps.*

In this definition, 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?

Definition 1.3.3. *Algebraic algorithms* deal with algebraic objects, make us of algebraic methods, and are based on algebraic theorems.

In this definition, 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.

1.4 Some Remarks on Computer Algebra Systems and Their Implementations

Computer algebra algorithms allow us to compute in and with a multitude of mathematical structures. Accordingly, there is a large number of computer algebra systems suiting different needs. There are *general* purpose and *special* purpose computer algebra systems. Some well-known general purpose systems are commercial, whereas many of the special purpose systems are open-source and can be downloaded from the internet for free. General purpose systems aim at providing basic functionality for a variety of different application areas. In addition to tools for symbolic computation, they usually offer tools for numeric computation and for visualization.

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 SINGULAR, MAXIMA, GAP, JULIA and SAGE are open computer algebra systems. However, for most computations we will use the computer algebra system SINGULAR, which is being developed at Technical university of Kaiserslautern (TU KL). SINGULAR can be either downloaded or conveniently access in an online interface, see `https://www.singular.uni-kl.de:8003/`.

The next section introduces SINGULAR.

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 factorial(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
   }
}
factorial(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`.

1.6 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 1.6.1. 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 1.6.2 (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 $0 \leq r <\mid b \mid$.*

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 1.6.3. 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 1.6.4.

(i) The function $d : \mathbb{Z} \to \mathbb{N} \cup \{-\infty\}$ defined by $d(a) =\mid a \mid$ 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 1.6.5. 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) $c \mid a$ and $c \mid b$,

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

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

  i) $a \mid c$ and $b \mid c$,

  ii) if $a \mid d$ and $b \mid d$, then $c \mid d$ 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 1.6.6.

- 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 1.6.7. 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 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, $N(a) = N(b)$ if and only if $a = u \cdot b$ 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 1.6.8.

i) If $R = \mathbb{Z}$, $U(a) = \operatorname{sign}(a)$ if $a \neq 0$ and $N(a) =\mid a \mid$ 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 $U(a) = \operatorname{lc}(a)$ (with the convention that $U(0) = 1$) and $N(a) = a/\mathrm{U(a)}$ defines a normal form, and a nonzero polynomial is normalized if and only if it is monic.

Theorem 1.6.9. *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

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

Proof. We have $| a_{i+1} | < | a_i |$ for $i \geq 2$ so after finitely many steps $a_i = 0$. □

Example 1.6.10. 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 $\gcd(36, -15) = -3$.

The Euclidean algorithm in Theorem 1.2 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:     $r := a \operatorname{rem} b$ // division with remainder
4:     $a := b$
5:     $b := r$
6: $a := N(a)$ // normal form
7: return $a$
```

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 $\gcd(0, 0) = 0$.

Example 1.6.11. Given a normal form, we compute the gcd of 36 and -15 as follows:

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

hence $\gcd(36, -15) = N(-3) = -3/U(-3) = -3/-1 = 3 > 0$.

In the following lemma, we see some of the properties of the gcd in $\mathbb{Z}$:

Lemma 1.6.12. *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) $\gcd(c \cdot a, c \cdot b) =\mid c \mid \cdot \gcd(a,b)$ *(distributivity),*

vi) $\mid a \mid = \mid b \mid \rightarrow \gcd(a,c) = \gcd(b,c)$.

Consider the following example:

Example 1.6.13. Recall that the gcd of 36 and 15 is computed as follows:

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

hence $\gcd(36, 1515) = 3$. Furthermore, one can express $\gcd(36,15)$ as a $\mathbb{Z}$ linear combination of 36 and 15:

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

In the following section we discuss how to express a gcd of two inputs as a linear combination of the two inputs.

1.7 Extended Euclidean Algorithm for Integers

This section extends Algorithm 1.2. The advantage of extending this algorithm is that it computes not only the gcd but also a representation of it as a linear combination of the inputs. This important method is called the *Extended Euclidean Algorithm (EEA)* and it works over any Euclidean domain.

Theorem 1.7.1. *Let* $a_1, a_2 \in \mathbb{Z} \setminus \{0\}$ *and* $a_3, \ldots, a_n \in \mathbb{Z}$ *be as in Theorem 1.6.9. Then reading the equation in Theorem 1.6.9 backwards yields the equation*

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

and it gives a representation

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

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

Proof. Since 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 $a_3, \ldots, a_n$. □

Using the method described above, the gcd 3 of 15 and 36 can be represented as a linear combination of 15 and 36, that is, :

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

The method described in Theorem 1.7.1 for representing a gcd of two integers as a linear combination of its inputs is summarized in the following algorithm, see Algorithm 1.3.

Algorithm 1.3 `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 $0 \le i \le l+1$, and $q_i \in R$ for $1 \le i \le l$, 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:     $q_i := r_{i-1} \operatorname{qou} r_i$// division with remainder
 5:     $r_{i+1} := r_{i-1} - q_i r_i$
 6:     $s_{i+1} := s_{i-1} - q_i s_i$
 7:     $t_{i+1} := t_{i-1} - q_i t_i$
 8:     $i := i + 1$
 9: $l := i - 1;$
10: return $l, r_i, s_i, t_i$ for $0 \le i \le l+1$, and $q_i$ for $1 \le i \le l$
```

Note that:

- the algorithm terminates because the $d(r_i)$ are strictly decreasing non-negative integers for $1 \le i \le l$, where d is the Euclidean function on R.
- The elements r_i for $1 \le i \le l+1$ are the remainders and the q_i for $1 \le i \le l$ 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 $1 \le i \le l+1$. 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 1.7.2.

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

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

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

Invariants of the Traditional EEA

The following lemma collects some invariants of the TEEA.

Lemma 1.7.3. *For $0 \le i \le l$, 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* $r_{l+1} = 0$.

Proof. For (i) and (ii) we proceed by induction on i. The case $i = 0$ is clear from step 1 of the algorithm, and we may assume $i \ge 1$. Then

$$Q_i \begin{pmatrix} r_{i-1} \\ r_i \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -q_i \end{pmatrix} \begin{pmatrix} r_{i-1} \\ r_i \end{pmatrix} = \begin{pmatrix} r_i \\ r_{i-1} - q_i r_i \end{pmatrix} = \begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix},$$

and (i) follows from $R_i = Q_i R_{i-1}$ and the induction hypothesis. Similarly, (ii) follows from

$$Q_i \begin{pmatrix} s_{i-1} & t_{i-1} \\ s_i & r_i \end{pmatrix} = \begin{pmatrix} s_i & t_i \\ s_{i+1} & t_{i+1} \end{pmatrix}$$

and the induction hypothesis.

The other parts are left as an exercises. □

Recall that if we are given a normal form, the gcd of two integers is defined as the unique normalized associate of all greatest common divisors of a and b. In the polynomial case, it turns out that it is not only useful to have a normal form for the gcd, but to modify the traditional Euclidean algorithm so that all the remainders r_i are normalized.

The computation of the traditional Euclidean algorithm produce remainders whose coefficients have huge numerators and denominators even for inputs of moderate size, and that the coefficients of the monic associates of the remainders are such smaller. The following variant of the traditional Eulidean algorithm, Algorithm 1.4, works with these monic associates:

Algorithm 1.4 `Extended Euclidean Algorithm (EEA)`

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

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: $\rho_0 := U(f), r_0 := N(f), s_0 := 1/\rho_0, t_0 := 0,$
2: $\rho_1 := U(g), r_1 := N(g), s_1 := 0, t_1 := 1/\rho_1$
3: while $r_i \neq 0$ do
4:     $q_i := r_{i-1} \operatorname{qou} r_i$// division with remainder
5:     $\rho_{i+1} := U(r_{i-1} - q_i r_i)$
6:     $r_{i+1} := (r_{i-1} - q_i r_i)/\rho_{i+1}$
7:     $s_{i+1} := (s_{i-1} - q_i s_i)/\rho_{i+1}$
8:     $t_{i+1} := (t_{i-1} - q_i t_i)/\rho_{i+1}$
9:     $i := i + 1$
10: $l := i - 1;$
11: return $l, r_i, s_i, t_i$ for $0 \leq i \leq l+1$, and $q_i$ for $1 \leq i \leq l$
```

The elements r_i for $0 \leq i \leq l+1$ are the *remainders*, the q_i for $1 \leq i \leq l$ are the *quotients*, and the elements $r - i$, s_i, and t_i form the i-th row in the EEA, for $0 \leq i \leq l+1$. The elements s_l and t_l, satisfying $s_l f + t_l g = \gcd(f, g)$, are the *Bezout coefficients* of f and g.

Example 1.7.4. With monic remainders, we compute the gcd of f and g using the EEA where $f = 18x^3 - 42x^2 + 30x - 6, g = -12x^2 + 10x - 2 \in \mathbb{Q}[x]$.

i	q_i	ρ_i	r_i	s_i	t_i
0		18	$x^3 - \frac{7}{3}x^2 + \frac{5}{3}x - \frac{1}{3}$	$\frac{1}{18}$	0
1	$x - \frac{3}{2}$	-12	$x^2 - \frac{5}{6}x + \frac{1}{6}$	0	$-\frac{1}{12}$
2	$x - \frac{1}{2}$	$\frac{1}{4}$	$x - \frac{1}{3}$	$\frac{2}{9}$	$\frac{1}{3}x - \frac{1}{2}$
3		1	0	$-\frac{2}{9}x + \frac{1}{9}$	$-\frac{1}{3}x^2 + \frac{2}{3}x - \frac{1}{3}$

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

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

One may apply division algorithm to find a list of prime numbers.

Definition 1.7.5. 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 1.7.6 (**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 1.7.7. 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 $p \mid n$ 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 1.7.8 (**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 1.7.9. 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 1.7.10 (**Prime Number Theorem**). *For $x \in \mathbb{R}_{>0}$ let*

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

Then

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

Example 1.7.11. The number of prime ≤ 21 is $\pi(21) = 8$, see Example 1.7.9.

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.

1.8 Groups

The concept of groups has important applications in almost every field of mathematics, including algebraic geometry and commutative algebra. It allows us to describe symmetries in mathematical objects and problems, reducing a more complicated problem to a simpler one. From the practical point of view this can speed up computations.

Recall first that an *action* of a group G on a set S is a function

$$\alpha : G \times S \to S$$

defined by $\alpha(g, x) = gx$ such that for all $x \in S$ and $g_1, g_2 \in G$:

$$ex = x, \quad \text{and } \alpha(g_1 g_2, x) = g_1(g_2 x)\,.$$

When such an action is given, we say that G *acts on the set* S.

Remark and Definition 1.8.1. Let G be a group and H a subgroup.

i) An action of the group H on the set G is given by $(h, x) \mapsto hx$, where hx is the product in G. The action of $h \in H$ on G is called a *(left) translation.* If K is another subgroup of G and S is the set of all left cosets of K in G, then H acts on S by translation: $(h, xK) \mapsto hxK$.

ii) An action of the group H on the set G is given by $(h, x) \mapsto hxh^{-1}$ with the product in G. The action of $h \in H$ on G is called *conjugation* by h and the element hxh^{-1} is said to be a *conjugate* of x. If K is any subgroup of G and $h \in H$, then hKh^{-1} is a subgroup of G isomorphic to K. Hence H acts on the set S of all subgroups of G by conjugation: $(h, K) \mapsto hKh^{-1}$. The group hKh^{-1} is said to be *conjugate* to K.

iii) An action of the symmetric group S_n on the set $I_n = \{1, 2, \ldots, n\}$ is given by $(\sigma, x) \mapsto \sigma(x)$.

Theorem and Definition 1.8.2. Let G be a group that acts on a set S.

a) The relation on S defined by

$$x \sim y \Leftrightarrow gx = y \text{ for some } g \in G$$

is an equivalence relation. The equivalence classes of this equivalence relation are called the *orbits* of G on S. The orbit of $x \in S$ is denoted $\overline{x}$.

b) For each $x \in S$,

$$G_x = \{g \in G \mid gx = x\}$$

is a subgroup of G and is denoted as $\mathrm{Stab}(x)$. The subgroup $\mathrm{Stab}(x)$ is called variously the *subgroup fixing* x, the *isotropy group* of x or the *stablizer* of x.

Proof. (a) $\sim$ satisfies reflexivity since $ex = x$, that is, $x \sim x$. The relation is symmetric since

$$x \sim x' \Rightarrow gx = x' \text{ for some } g \in G \Rightarrow x = g^{-1}x' \text{ and } g^{-1} \in G \Rightarrow x' \sim x\,.$$

Suppose $x \sim x'$ and $x' \sim x''$. Then by definition, $gx = x', g'x' = x''$ for some $g, g' \in G$. Since $g'g \in G$ and

$$(g'g)x = g'(gx) = g'x' = x''$$

we have $x \sim x''$ and, hence, $\sim$ is transitive. Thus $\sim$ is an equivalence relation.

b) The $\mathrm{Stab}(x)$ is non-empty since $ex = x \Rightarrow e \in \mathrm{Stab}(x)$. Let $g, g' \in \mathrm{Stab}(x)$. Then

$$gg'^{-1}x = g(g'x) = gx = x \Rightarrow gg'^{-1} \in \mathrm{Stab}(x)\,.$$

□

Example and Definition 1.8.3. Let G be a group.

a) If a group G acts on itself by conjugation, then the orbit of

$$\{gxg^{-1} \mid g \in G\}$$

of $x \in G$ is called the *conjugacy class* of x.

b) If a subgroup H acts on G by conjugation the isotropy group

$$H_x = \{h \in H \mid hxh^{-1} = x\} = \{h \in H \mid hx = xh\}$$

is called the *centralizer of* x in H and is denoted $C_H(x)$.

c) If $H = G$, $C_G(x)$ is simply called the *centlizer of* x.

d) If H acts by conjugation on the set S of all subgroups of G, then the subgroup of H fixing $K \in S$, namely $\{h \in H \mid hKh^{-1} = K\}$, is called the *normalizer of* K in H and denoted $N_H(K)$.

Consider, for example, the symmetry group G of the octahedron, which contains all rotations, reflections, and rotoreflections, which map the octahedron to itself. Numbering the vertices, we can identify any symmetry with an element of the symmetric group S_6 . For example, the rotation by 90 degrees along the axis through 1 and 6 is given by $(2,3,4,5) \in S_6$ using cycle notation. Let us say, we want to compute the stabilizers of the vertices of the octahedron, that is, the subgroups

$$\mathrm{Stab}(j) = \{\sigma \in G \mid \sigma(j) = j\}\,.$$

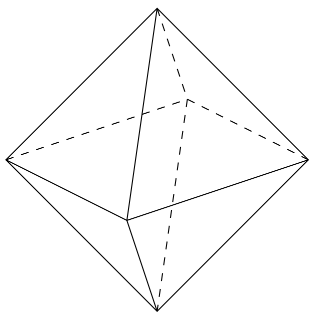

Figure 1.1: Octahedron

By the action of G, it is sufficient to determine Stab(1), since all stabilizers can be identified by conjugation of groups

$$\sigma(j) = \sigma^{-1}\mathrm{Stab}(1)\sigma,$$

where $\sigma \in G$ with $1 = \sigma(j)$. So instead of 6 computations, by taking the symmetry of the problem into account, we only have to do one. To compute the group order of Stab(1), we use:

Theorem 1.8.4. *Let G be a group acting on a set X by*

$$G \times X \to X$$

(that is, $ex = x$ and $(g \circ h)x = g(hx)$ for all $x \in X$ and $g, h \in G$). Fix $x \in X$ and write

$$\mathrm{Orb}(x) = \{gx \mid g \in G\}$$

for the orbit of x. Then

$$\mid G \mid = \mid \mathrm{Orb}(x) \mid \cdot \mid \mathrm{Stab}(x) \mid .$$

Hence

$$\mid \mathrm{Stab}(1) \mid = \frac{\mid G \mid}{6},$$

and, if p is a point which does not lie on any symmetry plane or axis,

$$\mid G \mid = \mid \mathrm{Stab}(p) \mid \cdot \mid Gp \mid = 1 \cdot (6 \cdot 8) = 48,$$

see Figure 1.1, so

$$\mid \mathrm{Stab}(1) \mid = \frac{\mid G \mid}{6} = 8,$$

It is easy to guess elements of G and Stab(1), but how do we know, whether they really generate the respective groups, equivalently, that they generate groups of the correct order? We can leave this tedious calculation to the open source computer algebra system Gap [?], the leading software for group theory:

Example 1.8.5. We use Gap to prove that

$$\mathrm{Stab}(1) = \langle (2,3,4,5),(2,4) \rangle$$

```
gap> Stab1:=Group((2,3,4,5),(2,4));;
gap> Size(Stab1);
8
gap> Elements(Stab1);
[(),(3,5),(2,3)(4,5),(2,3,4,5),(2,4),(2,4)(3,5),
(2,5,4,3),(2,5)(3,4)]
```

In the same way, we can find generators of the whole symmetry group:

```
gap> G:=Group((2,3,4,5),(1,3)(5,6));;
gap> Size(G);
48
```

Hence

$$G = \langle (2,3,4,5),(1,3)(5,6) \rangle .$$

Gap implements algorithms for computing with subgroups of symmetric groups. As we have seen, given a set of generators, it can determine the group order, and the set of all elements. It can also determine, whether two such groups are isomorphic, and specify an isomorphism.

1.9 Symbolic Integration

The problem of finding a formula for the antiderivative, or indefinite integral, of a given function $f(x)$ is called *symbolic integration*. That is, to find a differentiable function $F(x)$ such that

$$F(x) = \int f(x)\,\mathrm{d}x.$$

The term symbolic is used to distinguish this problem from that of *numerical integration*, where the value of F at a particular input or set of inputs, rather than a general formula for F, is sought. For example,

$$\int \frac{2}{3}x\,\mathrm{d}x = \frac{x^2}{3} + C \text{ and } \int_0^1 \frac{2}{3}x\,\mathrm{d}x = \frac{1}{3}$$

are symbolic results for indefinite and definite integrals, respectively, where C is a constant of integration. A numerical result for the definite integral is:

$$\int_0^1 \frac{2}{3}x\,\mathrm{d}x \approx 0.33\,.$$

Both symbolic and numerical integration problems were held to be of practical and theoretical importance long before the time of digital computers, but they are now generally considered the domain of computer science, as computers are most often used currently to tackle individual instances.

Though one can construct an algorithm for finding the derivative of an expression in a straightforward process, the reverse question, finding the integral, is much more difficult. The general purpose computer algebra systems like the commercial systems AXIOM, REDUCE, MAPLE, and MATHEMATICA are usually less powerful in the specific areas, but provide a larger set of algorithms to manipulate symbolic expressions.

A procedure called the *Risch algorithm* exists which is an algorithm for symbolic integration of functions which are compositions of rational functions, exponentials, logarithm, radicals, and trigonometric functions.

Risch algorithm:

- was, in its original form, not suitable for a direct implementation, and its complete implementation took a long time.
- was first implemented in the computer algebra system REDUCE in the case of purely transcendental functions; the case of purely algebraic functions was solved and implemented in REDUCE by James H. Davenport; the general case was solved and implemented in the computer algebra system AXIOM by Manuel Bronstein.
- is based on the theorem of Liouville, which uses the following definitions: A *differential field* is field K with a differentiation map

$$\begin{aligned} D : K &\to K, \\ f &\mapsto D(f) = f' \end{aligned}$$

which satisfies the usual rules:

$$D(f+g) = D(f) + D(g) \text{ and } D(fg) = fD(g) + D(f)g\,.$$

- applies only to indefinite integrals and most of the integrals of interest to physicists, theoretical chemists and engineers, are definite integrals often related to Laplace transforms, Fourier transforms and Mellin transforms.

Lacking of a general algorithm, the developers of computer algebra systems, have implemented heuristics based on pattern-matching and the exploitation of special functions, in particular the incomplete gamma function. Although this approach is heuristic rather than algorithmic, it is nonetheless an effective method for solving many definite integrals encountered by practical engineering applications. Note that the only almost complete implementation of Risch's algorithm is that of AXIOM.

1.10 Linear Algebra

In addition to the Euclidean algorithm, the second key algorithm generalized by Buchberger's algorithm is Gaussian reduction. We can reformulate Gaussian reduction on a homogeneous linear system of equations over a field K in the following way:

Algorithm 1.10.1 (Gauss)**.** Consider non-zero homogeneous linear polynomials [1]

$$f_1, \ldots, f_n \in K[x_1, \ldots, x_m]\,.$$

Choose an ordering of the variables (without loss of generality $x_1 > x_2 > \ldots > x_m$). Define $L(f_i)$ as the largest monomial of f_i and by $LC(f_i)$ its coefficient. As long as there are f_i and f_j with $L(f_i) = L(f_j)$ replace f_j by the S-pair

$$\operatorname{spoly}(f_i, f_j) = LC(f_i) \cdot f_j - LC(f_j) \cdot f_i$$

If $f_j = 0$, then delete f_j.
Sort the set of f_j by the size of $L(f_j)$.

This algorithm terminates with a row echelon form. If we reduce the resulting polynomials by those with smaller lead monomials, and divide all polynomials by their lead coefficients, we obtain the (unique) reduced row echelon form. When discussing Gröbner bases, we will see how the special case of linear equations generalizes to the higher degree setup.

[1] a homogeneous polynomial is a polynomial whose nonzero terms all have the same degree. For example, $x^d + 2x^{d-3}y^3 + 9x^{d-4}y^4$ is a homogeneous polynomial of degree d in two variables.

Example 1.10.2. We solve the following system of equations:

$$\begin{aligned} f_1 &= x_1 + x_2 + x_5 = 0 \\ f_2 &= x_1 + x_2 + 2x_3 + 2x_4 + x_5 = 0 \\ f_3 &= x_1 + x_2 + x_3 + x_4 + x_5 = 0\,. \end{aligned}$$

Gaussian elimination yields

$$\begin{aligned} f_1 &= x_1 + x_2 + x_5 = 0 \\ \mathrm{spoly}(\mathrm{f}_1, \mathrm{f}_2) &= 2x_3 + 2x_4 = 0 \\ \mathrm{spoly}(\mathrm{f}_1, \mathrm{f}_3) &= x_3 + x_4 = 0 \end{aligned}$$

and since the S-pair of the last two vanishes

$$\begin{aligned} x_1 + x_2 + x_5 &= 0 \\ 2x_3 + 2x_4 &= 0\,. \end{aligned}$$

Utilizing the Gröbner basis engine of Singular, we can do the same computation as follows:

```
ring R = 0, (x(1..5)), lp;
ideal I = x(1) + x(2) + x(5),
          x(1) + x(2) + 2*x(3) + 2*x(4) + x(5),
          x(1) + x(2) + x(3) + x(4) + x(5);
option(redSB);
ideal G = std(I);
G;
_[1] = x(3) + x(4)
_[2] = x(1) + x(2) + x(5)
```

Remark 1.10.3. Solving an inhomogeneous linear system of equations

$$f_i = \sum_j a_{ij} x_j - c_i = 0$$

can be reduced to the homogeneous case by homogenizing the system to

$$f_i = \sum_j a_{ij} x_j - c_i y = 0$$

with a homogenizing variable y.

1.11 Algebraic Geometry

If we pass from linear systems to polynomial equations of higher degree things become much more interesting. Algebraic geometry studies the set of solutions of such systems. It will be the main motivation of the algorithms we encounter in commutative algebra.

Definition 1.11.1. The *affine space* of dimension n over the field K is defined as

$$\mathbb{A}^n(K) = \{(a_1, \ldots, a_n) \in K^n \mid a_1, \ldots, a_n \in K\} = K^n.$$

Definition 1.11.2.

- An *affine algebraic set* is the common zero set

  $$V(f_1, \ldots, f_r) = \{p \in K^n \mid f_i(p) = 0 \text{ for all } 1 \le i \le r\}$$

  of polynomials $f_1, \ldots, f_r \in K[x_1, \ldots, x_n]$.

- An affine algebraic set X is called *irreducible*, if it cannot be written as $X = X_1 \cup X_2$ with affine algebraic sets $X_i \subsetneq X$. Then we also call X an *affine algebraic variety*.

Example 1.11.3. Affine algebraic varieties commonly known also outside algebraic geometry are $V(1) = \emptyset, V(0) = K^n$, the set of solutions of a linear system of equations

$$A \cdot x - b = 0$$

or the graph

$$\Gamma(g) = V(x_2 \cdot b(x_1) - a(x_1)) \subseteq K^2$$

of a rational function

$$g = \frac{a}{b} \in K(x_1)\,.$$

For example, the graph of $g(x_1) = \frac{x_1^3-1}{x_1}$ is

$$V(x_2x_1 - x_1^3 + 1) \subseteq K^2\,.$$

Example and Definition 1.11.4. If $f \in K[x_1, \ldots, x_n]$ is non-zero and non-constant, then $V(f) \subseteq K^n$ is called a *hypesurface.* Thus a subset of K^n is algebraic if and only if it can be written as the intersection of finitely many hypersurfaces. Hypersurfaces in K^2 are called *plane curves.* Note that $V(f)$ is irreducible if and only if f is irreducible.

In the next chapter we will discuss the algorithmic foundation for computing with algebraic sets.

Chapter 2

Ideals, Varieties and Standard Bases

2.1 Ideals and Varieties

Let K be a field. Let us start this section with following easy but important observation about algebraic set $V(f_1, \ldots, f_s)$ with $f_i \in R = K[x_1, \ldots, x_n]$:

If $f_1(p) = 0, \ldots, f_s(p) = 0$ for $p \in K^n$, then also any R-linear combination of the f_i vanishes on p, that is,

$$\left(\sum_{i=1}^{s} r_i \cdot f_i \right)(p) = \sum_{i=1}^{s} r_i(p) f_i(p) = 0$$

for all $r_i \in R$. Hence, $V(f_1, \ldots, f_s)$ depends only on the ideal

$$\langle f_1, \ldots, f_s \rangle = \left\{ \sum_{i=1}^{s} r_i \cdot f_i \,\middle|\, r_i \in R \right\} \subseteq R,$$

generated by $f_1, \ldots, f_s$.

Recall

Definition 2.1.1. Let R be a commutative ring with $1_R = 1$. An *ideal* is a non-empty subset $I \subseteq R$ with

$$a + b \in I\,,\ ra \in I$$

for all $a, b \in I$ and $r \in R$. If $S \subseteq R$, then

$$\langle S \rangle = \left\{ \sum_{\text{finite}} r_i \cdot f_i \,\middle|\, r_i \in R, f_i \in S \right\}$$

is the *ideal generated* by S.

Recall, that the definition of an ideal is motivated in algebra by the following: For a subgroup $I \subseteq R$ the additive group R/I becomes a ring with multiplication induced by that of R if and only if I is an ideal (prove this as an easy exercise).

By the above observation it is natural to consider, instead of the vanishing locus of a set of equations, the vanishing locus of an ideal:

Definition 2.1.2. If $I \subset K[x_1, \ldots, x_n]$ then

$$V(I) = \{p \in K^n \mid f(p) = 0 \text{ for all } f \in I\}$$

is called the *vanishing locus* of I.

This is indeed an affine variety, because any ideal $I \subset K[x_1, \ldots, x_n]$ is finitely generated, as we will prove in Theorem ?.

Definition 2.1.3. Let $S \subseteq K^n$ be a subset. Consider the polynomial ring $R = K[x_1, \ldots, x_n]$. Then

$$I(S) = \{f \in R \mid f(p) = 0 \text{ for all } p \in S\}$$

is an ideal, the *vanishing ideal* of S.

Example and Definition 2.1.4. Consider the elliptical arc

$$S = \{(x_1, x_2) \in \mathbb{R}^2 \mid x_1^2 + 2x_2^2 = 1 \text{ and } x_1, x_2 \geq 0\}$$

Then the vanishing ideal of S is

$$I(S) = \langle x_1^2 + 2x_2^2 - 1 \rangle \,.$$

However, the vanishing locus $V(I(S))$ of $I(S)$ is the complete ellipse, the **smallest algebraic set** containing S. The set $V(I(S))$ is called the *closure* of S in the so called *Zariski topology* and is denoted by

$$\overline{S} = V(I(S))\,.$$

The Zariski topology on K^n has as closed sets the $V(I)$ for ideals $I \subseteq K[x_1, \ldots, x_n]$.

By I and V [1] inclusion reversing maps

$$\begin{aligned} I : \{ \text{ affine algebraic sets} X \subseteq K^n\} &\to \{ \text{ ideals in } K[x_1, \ldots, x_n]\} \\ V : \{ \text{ ideals in } K[x_1, \ldots, x_n]\} &\to \{ \text{ affine algebraic sets } X \subseteq K^n\} \end{aligned}$$

between the set of algebraic subsets of K^n and the set of ideals of $K[x_1, \ldots, x_n]$ are given, it remains to show that any ideal $I \subseteq K[x_1, \ldots, x_n]$ is finitely generated, that is, there are finitely many $f_1, \ldots, f_s \in K[x_1, \ldots, x_n]$ with $I = \langle f_1, \ldots, f_s \rangle$. We begin with a characterization of these ideals:

Theorem and Definition 2.1.5. Let R be a commutative ring with 1. The following conditions are equivalent:

[1]the map V is not injective since $V(x) = V(x^2) \subseteq K$.

a) Every ideal $I \subseteq R$ is *finitely generated.*

b) Every ascending chain
$$I_1 \subseteq I_2 \subseteq \ldots \subseteq I_n \subseteq \ldots$$
of ideals terminates, that is, there is an m, such that
$$I_m = I_{m+1} = I_{m+2} = \ldots .$$

c) Every non-empty set of ideals has a maximal element with respect to inclusion. If R satisfies one of these conditions, then R [2] is called *Noetherian.*

Proof. a) $\Rightarrow$ b): Let $I_1 \subseteq I_2 \subseteq \ldots$ be a claim of ideals. Then
$$I = \sum_{j=1}^{\infty} I_j$$
is also an ideal of I. If $a, b \in I$, then there are $j_1, j_2 \in \mathbb{N}$ with $a \in I_{j_1}$, $b \in I_{j_2}$, and
$$a + b \in I_{\max\{j_1, j_2\}} \subseteq I .$$
By (a) the ideal I is finitely generated, hence there are $f_1, \ldots, f_s \in I$ with $I = \langle f_1, \ldots, f_s \rangle$. For every f_k there is a j_k with $f_k \in I_{j_k}$. For
$$m := \max\{j_k \mid k = 1, \ldots, s\}$$
we have $f_1, \ldots, f_s \in I_m$, so
$$I = \langle f_1, \ldots, f_s \rangle \subseteq I_m \subseteq I_{m+1} \subseteq \ldots \subseteq I$$
and hence
$$I_m = I_{m+1} = \ldots .$$

b) $\Rightarrow$ c): Assume that c) does not hold. Then there is a set M of ideals such that for every $I \in M$ there is an $I' \in M$ with $I \subsetneqq I'$ strictly contained. Hence, by induction, we obtain a sequence
$$I_1 \subsetneqq I_2 \subsetneqq \ldots$$
of ideals in M, which does not terminate, that is, b) is not satisfied.

[2]Rings satisfying one of these conditions of the theorem called Noetherian after Emmy Noether (1882-1935), who has formulated the general structure theory for this class of rings and used this to give a simpler and more general proof of the theorems of Kronecker and Lasker.

c) $\Rightarrow$ a): Let I be an arbitrary ideal. The set

$$M = \{I' \subseteq I \mid I' \text{ finitely generated }\}$$

is non-empty, for example, $\langle 0\rangle \in M$. Let J be a maximal element of M. So there are $f_1, \ldots, f_s \in J$ with $J = \langle f_1, \ldots, f_s\rangle$. We show that $I = J$. If this is not true, then there is an $f \in I \setminus J$ with

$$J \subsetneqq \langle f_1, \ldots, f_s, f\rangle \subseteq I\,.$$

This contradicts the maximality of J. □

Example 2.1.6.

1. The ring of integers $\mathbb{Z}$ is Noetherian, since all of $\mathbb{Z}$ are of the form

$$\langle n\rangle = n\mathbb{Z} = \{nk \mid k \in \mathbb{Z}\}$$

(an exercise) and, hence, are finitely generated (by a single element).

2. A field K only has the ideals $\langle 0\rangle$ and $K = \langle 1\rangle$, in particular, K is Noetherian.

3. If R is Noetherian and $I \subseteq R$ an ideal, then the quotient ring R/I is Noetherian.

Proof. Let $\pi : R \to R/I$ be the canonical epimorphism. If $J \subseteq R/I$ an ideal then by assumption $\pi^{-1}(J) = \langle f_1, \ldots, f_s\rangle$, and $J = \langle \pi(f_1), \ldots, \pi(f_s)\rangle$. □

Hilbert has shown in 1890, that the polynomial ring $K[x_1, \ldots, x_n]$ over a field K is Noetherian:

Theorem 2.1.7 (Hilbert's Basis Theorem(HBT)**).** *If R is a Noetherian ring, then also $R[x]$ is Noetherian.*

Using that a field K and the ring of integers $\mathbb{Z}$ are Noetherian, by induction on the number n of variables

$$R[x_1, \ldots, x_n] = R[x_1, \ldots, x_{n-1}][x_n]$$

we obtain:

Corollary 2.1.8. *Let K be a field. Then the polynomial rings $K[x_1, \ldots, x_n]$ and $\mathbb{Z}[x_1, \ldots, x_n]$ in n variables are Noetherian.*

The fact that $K[x_1, \ldots, x_n]$ is Noetherian, is the basis of all algorithms, we will discuss.
For the proof of HBT we consider the lead coefficients in R of polynomials in $R[x]$. If

$$f = a_k x^k + \ldots + a_1 x + a_0 \in R[x]$$

with $a_k \neq 0$, then the *degree* of f is $\deg f = k$, its *leading coefficients* is $\mathrm{LC}(f) = a_k$, its *lead term* $\mathrm{LT}(f) = a_k x^k$, and its *lead monomial* $\mathrm{L}(f) = x^k$. We now prove the HBT:

Proof. Assume $R[x]$ is not Noetherian. Then there is an ideal $I \subseteq R[x]$ which is not finitely generated. Let $f_1 \in I$ with $\deg f_1$ minimal, $f_2 \in I \setminus \langle f_1 \rangle$ with $\deg f_2$ minimal, and inductively

$$f_k \in I \setminus \langle f_1, \ldots, f_{k-1} \rangle$$

with $\deg f_k$ minimal. Then

$$\deg f_1 \leq \deg f_2 \leq \ldots \leq \deg f_k \leq$$

and we obtain an ascending chain of ideals in R.

$$\langle LC(f_1) \rangle \subseteq \langle LC(f_1), LC(f_2) \rangle \subseteq \ldots \subseteq \langle LC(f_1), LC(f_2), \ldots, LC(f_k) \rangle \subseteq \ldots .$$

We show that the inclusions are strict (and hence R is not Noetherian): Assume

$$\langle LC(f_1), LC(f_2), \ldots, LC(f_k) \rangle = \langle LC(f_1), LC(f_2), \ldots, LC(f_{k+1}) \rangle .$$

Then we can write

$$LC(f_{k+1}) = \sum_{j=1}^{k} b_j LC(f_j)$$

with $b_j \in R$. Hence

$$g := \sum_{j=1}^{k} b_j x^{\deg f_{k+1} - \deg f_j} f_j \in \langle f_1, \ldots, f_k \rangle$$

has the same lead term as f_{k+1}, so

$$\deg (g - f_{k+1}) < \deg f_{k+1},$$

a contradiction, since f_{k+1} was chosen to have minimal degree. □

Theorem 2.1.9 (**Weak Nullstellensatz** (WN)). *Let K be an algebraically closed field and $I \subseteq K[x_1, \ldots, x_n]$ an ideal. Then*

$$V(I) = \emptyset \Longleftrightarrow I = K[x_1, \ldots, x_n] .$$

Remark 2.1.10. The condition that K is algebraically closed, is necessary. For example, $V(x^2 + y^2 + 1) \subseteq \mathbb{R}^2$ is empty (it is not empty over $\mathbb{C}$).

From WN theorem, we obtain:

Theorem 2.1.11 (**Strong Nullstellensatz** (SN)). *Let K be an algebraically closed field and $I \subseteq K[x_1, \ldots, x_n]$ an ideal. Then*

$$I(V(I)) = \sqrt{I}$$

where

$$\sqrt{I} = \{f \in K[x_1, \ldots, x_n] \mid \exists a \in \mathbb{N} \text{ with } f^a \in I\} .$$

denotes the radical of I.

Bibliography