An Introductory Course on Sage

Lecture Notes

Summer Term 2020
University of Potsdam

Dr. Saskia Roos

Michael Jung

Last Update: September 30, 2020

Contents

I The Basics

1 Why Sage? 11

2 Getting Started 13

2.1 Installing Sage 13
2.1.1 Linux 13
2.1.2 Windows 14
2.1.3 MacOS 14

2.2 The Jupyter Notebook 15
2.2.1 First Steps 15
2.2.2 Code Cells 15
2.2.3 Markdown Cells 16

2.3 First Calculations 16
2.3.1 Basic Arithmetics 16
2.3.2 Predefined Functions and Values 18
2.3.3 Simple Plots 18

2.4 Useful Features 20

3 Elementary Algebra and Calculus 21

3.1 Variables and Symbolic Expressions 21
3.1.1 Python vs. Sage 21
3.1.2 Symbolic Expressions 23

3.1.3 Simplifications . . . 25
3.2 Elementary Algebra 27
3.2.1 Polynomial and Fractional Expressions . . . 27
3.2.2 Equations . . . 28
3.3 Calculus 32
3.3.1 Symbolic Functions . . . 32
3.3.2 Sums . . . 32
3.3.3 Sequences and Series . . . 34
3.3.4 Derivatives and Integrals . . . 36
3.3.5 Vector and Matrix computation . . . 40

II Programming and Data Structure

4 Algorithmics . . . 49
4.1 Procedures and Functions 49
4.2 Conditionals 51
4.3 Loops 52
4.3.1 Example: A Sequence with an unknown Limit . . . 55
4.4 Output 57

5 Lists and Other Data Structures . . . 59
5.1 Lists 59
5.1.1 Global List Operations . . . 61
5.1.2 List Manipulation . . . 64
5.1.3 Example: The Sieve of Eratosthenes . . . 66
5.1.4 Example: The UlamSsequence . . . 67
5.2 Character Strings 68
5.3 Finite Sets 68
5.3.1 Example: The Inclusion-Exclusion Principle . . . 69
5.4 Dictionaries 71
5.5 More on Data Structures 72
5.5.1 Shared or Duplicated Data Structure . . . 72
5.5.2 Mutable and Immutable Data Structure . . . 72

III Graphics

6 2D Graphics . . . 77
6.1 Drawing Curves 77
6.1.1 Graphical Representation of a Function . . . 77
6.1.2 Parametric Plot . . . 87
6.1.3 Curves in Polar Coordinates . . . 87

6.1.4 Curve defined by an Implicit Equation . . . 89
6.2 Vector Fields 90
6.3 Complex Functions 91
6.4 Density and Contour Plots 92
6.5 Data Plot 95
6.6 More Graphic Primitives 95

7 3D Graphics . . . 107
7.1 Plotting Functions 107
7.1.1 Drawing the Graph of a Function . . . 107
7.1.2 Parametric Plots . . . 112
7.1.3 Implicit Plot . . . 113
7.2 Vector Fields 114
7.3 More Graphic Primitives 114

IV Algebra and Symbolic Computation

8 Computational Domains . . . 119
8.1 Sage is Object-Oriented 119
8.2 Elements and Parents 121
8.3 Domains with a Normal Form 122
8.3.1 Elementary Domains . . . 122
8.3.2 Compount domains . . . 125
8.4 Expressions vs. Computational Domains 128
8.4.1 Symbolic Polynomials vs. Polynomial Rings . . . 128
8.4.2 Applications . . . 133
8.4.3 Example: The Aliquot sequence . . . 135
8.5 Primality Test 136

9 Polynomial Rings . . . 139
9.1 Euclidean Arithmetic 142
9.1.1 Divisibility . . . 142
9.1.2 Ideals and Quotients . . . 145
9.2 Factorization and Roots 147
9.2.1 Factorization . . . 148
9.2.2 Roots . . . 149
9.3 Rational Functions 150
9.4 Formal Power Series 152
9.4.1 Operations on Truncated Power Series . . . 152
9.4.2 Applications involving Power Series . . . 154
9.4.3 Lazy Power Series . . . 156

10 Matrices 159
10.1 Constructions and Elementary Manipulations 159
10.1.1 Matrix Spaces and Groups 159
10.1.2 Matrix and Vector Constructions 162
10.1.3 Basic Manipulations and Arithmetic 163
10.2 Matrix Computations 165
10.2.1 Gaussian Elimination and Normal Forms 165
10.2.2 Linear System Solving, Image and Nullspace Basis 170
10.3 Spectral Decomposition 173
10.3.1 Cyclic Invariant Subspaces and the Frobenius Normal Form 174
10.3.2 Eigenvalues and Eigenvectors 179
10.3.3 Jacobi Normal Form 181

11 Polynomial Systems 185
11.1 Polynomials in Several Variables 185
11.2 Polynomial Systems and Ideals 190
11.2.1 A first Example 190
11.2.2 Ideals and Systems 194
11.2.3 Computing Modulo an Ideal 197
11.2.4 Radical of an Ideal and Solutions 199
11.2.5 Operations on Ideals 201
11.3 Solving Strategies 201
11.3.1 Elimination 202
11.3.2 Zero-Dimensional Systems 208

12 Differential Equations 217
12.1 First Order Ordinary Differential Equations 217
12.1.1 Types of First Order ODEs 219
12.1.2 A Parametric Equation 228
12.1.3 Numerical Solving 229
12.2 Second Order Equations 231
12.2.1 How to solve a PDE: The Heat Equation 232
12.3 The Laplace Transform 233
12.4 Systems of Linear Differential Equations 236
12.4.1 Systems of First Order Differential Equations 236
12.4.2 Systems of Higher Order 237
12.4.3 Numerical Solving 240

V Appendix

A Git It 247

A.1 Getting Started **247**

A.1.1 Installing Git 247

A.2 How to Use Git **248**

A.2.1 Clone and Pull 248

A.2.2 Create your own Repository 248

A.2.3 Stage Area, Commits and Push 249

A.2.4 Branches 249

A.2.5 Workflow Example 250

Bibliography 253

Online **253**

Books **253**

Index 255

I The Basics

1 Why Sage? 11

2 Getting Started 13
2.1 Installing Sage
2.2 The Jupyter Notebook
2.3 First Calculations
2.4 Useful Features

3 Elementary Algebra and Calculus 21
3.1 Variables and Symbolic Expressions
3.2 Elementary Algebra
3.3 Calculus

1. Why Sage?

Even though mathematics is a beautiful kind of art, it sometimes requires tedious tasks: calculating confusing integrals, computing complicated Taylor expansions up to high orders, operating with matrices in high dimensions or repeating the very same calculation over and over again. It would be such a relieve if we had a machine which could do the whole time-consuming work for us. Of course, it is well-known that every personal computer is capable of these tasks. Thus, we only need the right software.

`Sage` ("**S**ystem for **A**lgebra and **G**eometry **E**xperimentation") is an open-source computer algebra system for desktop computers and represents a powerful alternative to `Mathematica` (i. e. `WolframAlpha`), `MATLAB` and `Magma`. Its library contains over 100 additional open-source packages, which are all interfaced via the `Python` programming language. Therewith, Sage is capable of symbolic as well as numerical computations in analysis, algebra, linear algebra, geometry, tensor algebra and more. Although Sage is based on `Python`, it provides a slightly different syntax closer to actual mathematics, which turns it into a perfect tool for mathematicians.

Throughout this course, we will survey `Sage`'s capabilities as well as limitations. In this notes, we discuss `Sage`'s syntax in contrast to Python's and give a glimpse of internal processes. After this course, we will be able to use all basic functionalities of `Sage` and apply them to mathematical problems. For these notes, we use version 9.0 of `Sage`.

2. Getting Started

2.1 Installing Sage

The installation process of SageMath 9.0 depends on the operating system. We give short introductions for Linux, Windows and macOS.

R *It is also possible to run Sage externally via the web-based client CoCalc. After signing up on https://cocalc.com, Sage can be used on a remote CoCalc-server. A free membership allows only small calculations, which should be sufficient for this course. However, as a precaution, we still recommend installing Sage on your local machine.*

2.1.1 Linux

On Debian-based distributions, it is possible to install Sage using the apt manager. However, the standard ppa usually does not provide the recent version of Sage. Rather follow the subsequent instructions to install the most recent version of Sage on your local machine:

1. Visit https://www.sagemath.org/download-linux.html and choose any download server (recommendation: "Freie Universität Berlin, Germany").
2. Extract the archive file into a directory of your choice.
3. Start a new terminal, change to the extraction folder and type the following command into the command line to run Sage:

```
$ ./sage
```

If you have followed the instructions precisely, you should see this message in your terminal:

```
Rewriting paths for your new installation directory
===================================================

This might take a few minutes but only has to be done once.

patching ...  (long list of files)
```

At this point, you should not move this directory if you expect Sage to function properly. In case you want to start Sage from anywhere in your command line, simply use `ln` to create a shortcut:

```
$ sudo ln -s /path/to/SageMath/sage /usr/local/bin/sage
```

Here, `/path/to/SageMath/sage` represents the actual path to your Sage installation. From now on, you can run Sage by typing the command

```
$ sage
```

2.1.2 Windows

Before you install Sage on a Windows system, please make sure that your Windows installation is of 64-bit type. If not, either switch to Linux, use CoCalc or install a 64-bit Windows on your local machine. If Windows is of 64-bit type already, proceed with the following instructions:

1. Visit https://github.com/sagemath/sage-windows/releases and download the installer `SageMath-9.0-Installer-v0.6.0.exe`.
2. Execute the installer and follow the instructions.
3. Open the application "SageMath 9.0" to run the Sage console interface.

2.1.3 MacOS

The following instructions are intended for OS X version 10.4 and higher.

1. Visit https://www.sagemath.org/download-linux.html and choose any download server (recommendation: "Freie Universität Berlin, Germany").
2. Download the `dmg` according to your machine and double click on it.
3. Drag the `sage` folder to your favorite place, e. g. into `/Applications`.
4. Use `finder` to visit the `sage` folder and double click on "sage".
5. Select to run it with `Terminal`:
   a) Choose Applications and select "All Applications" in the "Enable:" drop down.
   b) Change the "Application" drop down to "Utilities".
   c) On the left, scroll and select "Terminal".
6. After clicking "Open", Sage should pop up in a window.

2.2 The Jupyter Notebook

`Jupyter Notebook` is a web-based computational environment for creating documents interacting with `Python` code. It is intended to present code in a nice way and is also capable of LaTeX formatting. `Jupyter Notebook` is part of the open-source project Jupyter [2], and is already included in `Sage`. We discuss the absolute minimum for this course. For a more detailed introduction into `Jupyter Notebook`, consult [1] instead. In the following, when we speak of `Jupyter`, we usually mean `Jupyter Notebook`.

2.2.1 First Steps

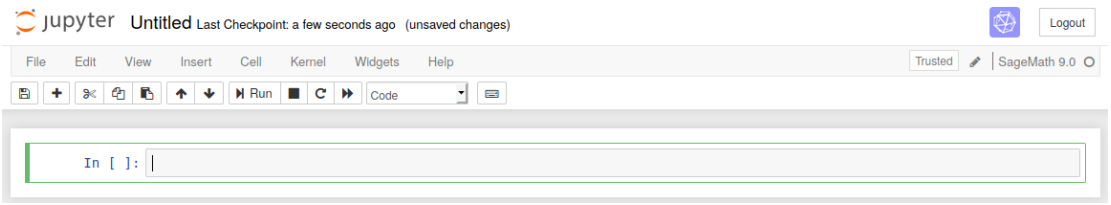
In order to start `Jupyter`, type the following command into the `Sage` command line:

```
sage: !sage -n jupyter
```

A browser window pops up with the `Jupyter` web-interface.

R *We recommend to create a separate folder for your `Jupyter` notebooks. To do so, click on the upper right corner:* New ⟩ Folder. *To rename the new folder, select the checkbox next to it and click on* Rename *in the upper left corner. Click on the folder to change the working directory.*

To create a new `Jupyter` notebook running on the `Sage` kernel, click on New ⟩ SageMath 9.0. Now, your browser should look like this:



`Jupyter` essentially distinguishes between two kind of cells, which we want to discuss in the next two sections.

2.2.2 Code Cells

Code cells are blocks that contain code you want to execute. The code can be executed by pressing ⇧+↵ on your keyboard. Then the corresponding output is displayed below:

```
In [1]: print('Hello World!')
        Hello World!
```

To each executed code cell corresponds a label number which indicates when the cell was executed on the kernel. While working with `Jupyter`, you should check on a regular basis whether your code cells run in the correct order.

2.2.3 Markdown Cells

It is convenient to surround your code with explanations in ***Markdown cells***. Markdown cells contain formattable text and display their output just in-place. A plain-text-formatting is provided via the Markdown language. Markdown supports headings, listings, HTML and more, see [5] for details. Furthermore, LaTeX code can be embedded by enclosing the corresponding expressions with "$...$".

To convert a code cell into a Markdown cell, select the cell you want to convert and press M on your keyboard, or click on Cell > Cell Type > Markdown in the menu bar. You can convert it back to a code cell when you press Y. A Markdown cell carries its content by pressing ⇧+↵.

2.3 First Calculations

Henceforth, all commands and calculations in Sage are presented as follows:

```
sage: 2+3
5
```

Here, the prefix sage indicates that we are using the Sage command line. If you prefer Jupyter instead, just type in the subsequent code. The kernel will do its job.

2.3.1 Basic Arithmetics

Notice that all common operations such as +, -, *, / and parentheses are carried out as usual.

```
sage: (3+4)*2/6
7/3
```

To operate with exponents, we use ^ or **:

```
sage: 5^4
625
sage: 5**4
625
```

R *Notice that Python's syntax differs from Sage's for exponents. In Python, we **must** type ** to use exponents, but since Sage is preferably designed for mathematical purposes, the operation ^ is allowed here, too.*

Apart from usual divisions, Sage offers division operations with pure integers. The so-called ***floor-division*** operator // divides two integers and cuts their decimals.

```
sage: 23/7
23/7
sage: 23//7
3
```

To compute its remainder, we use the modulus operator % instead.

General Arithmetics	
Binary Operations	`a+b, a-b, a*b, a/b`
Exponent	`a^b` or `a**b`
Square Root	`sqrt(a)`
n-th Root	`a^(1/n)`
Integer Arithmetics	
Floor-Division	`a // b`
Remainder	`a % b`
Floor-Division & Remainder	`divmod(a,b)`
Factorial $n!$	`factorial(n)`
Binomial Coefficient $\binom{n}{k}$	`binomial (n,k)`

Table 2.1.: Basic Operations in Sage.

```
sage: 23%7
2
sage: (23//7)*7 + 23%7
23
```

Both operations can be carried simultaneously by using `divmod`.

```
sage: divmod(23,7)
(3, 2)
```

As we can extract from Table 2.1, there are more common integer operations such as factorials and binomial coefficients.

```
sage: factorial(6)
720
sage: binomial(8,3)
56
```

When we carry out the ordinary division with two integers, Sage automatically returns the explicit fraction. To obtain a ***numerical approximation*** instead, we can either write one of the involved numbers with a decimal point or use the `numerical_approx` function.

```
sage: 23./7
3.28571428571429
sage: numerical_approx(23/7)
3.28571428571429
```

Furthermore, we can specify the precision of the numerical approximation by determining the number of digits.

```
sage: numerical_approx(23/7, digits=20)
3.2857142857142857143
```

Predefined Functions	
Exponential, Natural Logarithm	`exp, log`
Logarithm w.r.t. Base b	`log(a,b)`
Trig. Functions	`sin, cos, tan`
Inverse Trig. Functions	`arcsin, arccos, arctan`
Hyp. Trig. Functions	`sinh, cosh, tanh`
Inverse Hyp. Trig. Functions	`arcsinh, arccosh, arctanh`
Absolute Value / Modulus	`abs(a)`
Special Values / Constants	
Imaginary Unit i	`I` or `i`
Plus / Minus Infinity	$\pm$`Infinity` or $\pm$`oo`
π	`pi`
Euler's Constant	`e`
Euler-Mascheroni Constant γ	`euler_gamma`
Golden Ratio $\varphi = (1+\sqrt{5})/2$	`golden_ratio`

Table 2.2.: Predefined Functions and Constants.

2.3.2 Predefined Functions and Values

Several usual mathematical functions and constants are already implemented in Sage. A comprehensive list is provided in Table 2.2. If not stated otherwise, their computation is done with exact results,

```
sage: tan(pi/3)
sqrt(3)
sage: exp(I*pi)
-1
sage: sqrt(2)
sqrt(2)
```

even if the expression sometimes looks unnecessarily complicated.

```
sage: exp(I*pi/4)
(1/2*I + 1/2)*sqrt(2)
```

Simplification of such expressions will be discussed in Section 3.1.3. After all, if desired, we can still get the numerical approximation:

```
sage: numerical_approx(sqrt(2))
1.41421356237310
sage: numerical_approx(exp(I*pi/4))
0.707106781186548 + 0.707106781186548*I
```

2.3.3 Simple Plots

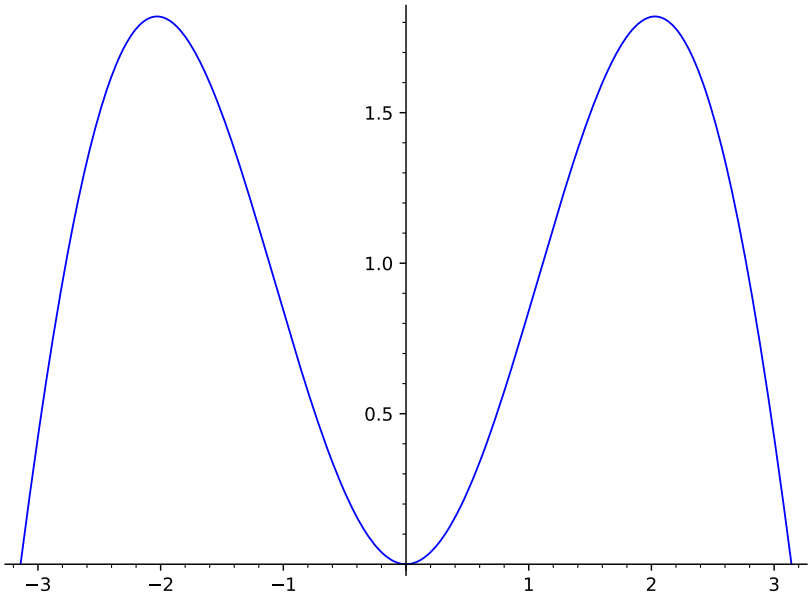
As a short preview, we present the basic plot functionalities of Sage. Using `plot` or `plot3d`, we can draw a graph with respect to one or two variables. The syntax for `plot`

is given as follows:

```
plot(expression, depending variable, min value , max value)
```

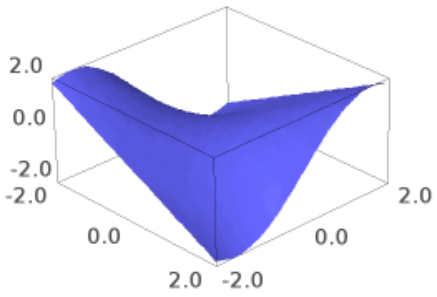
A simple example yields:

```
sage: x = var('x')
sage: plot(x*sin(x), x, -pi, pi)
```



The syntax for `plot3d` is quite similar as can be seen in the following example.

```
sage: x,y = var('x,y')
sage: plot3d(x*sin(y), (x,-2,2), (y,-2,2))
```



These are only the basic plot commands. The graphical capabilities of Sage are much wider. We discuss them in Part III.

2.4 Useful Features

We shortly introduce four useful tools which come in quite handy for further use.

1. It is recommended to *always* leave comments within the code. ***Comments*** are lines between your code which are ignored by the interpreter. They are introduced with a hashtag #:

```
sage: 5+5 # this is a comment
10
```

2. It is possible display all ***outputs in LaTeX***. To achieve this, we type

```
sage: %display latex
```

   Then all following outputs will be printed in LaTeX. This comes in convenient, especially with regards to `Jupyter`: the LaTeX output will be automatically compiled there. The option can be undone with:

```
sage: %display plain
```

3. Due to the open-source nature of `Sage`, we have full access to the documentation and source code of all available methods and functions. To obtain the documantation page containing the function's description, its syntax and some examples of usage, we add a

3. Elementary Algebra and Calculus

3.1 Variables and Symbolic Expressions

3.1.1 Python vs. Sage

In Section 2.3, we discussed explicit calculations within Sage. Since Sage is interfaced via the Python programming language, we can facilitate these calculations and assign explicit values to so-called ***Python variables*** in the following manner:

```
sage: x = 1
sage: x + 1
2
```

As we can see, the Python variable x has been set to 1 and can be used for further calculations. By default, Sage stores the last three results in the Python variables _, __ and ___:

```
sage: 4+5
9
sage: _ + 6
15
sage: __
9
```

Even though Python variables can be named arbitrarily, it is not recommended to redefine predefined constants or functions. This could lead to confusing results:

```
sage: pi = -I
sage: exp(I*pi)
e
```

Fortunately, the original value can be restored via

```
sage: from sage.all import pi
```

To restore *all* predefined variables and functions instead, we use the command

```
sage: restore()
```

When, in addition, all user-defined Python variables should be deleted, we type

```
sage: reset()
```

This command delivers a full reset of Sage.

In contrast to pure programming, in mathematics we mostly deal with indeterminates in order to solve equations or provide general results. In Sage, these indeterminates are realized by so-called ***symbolic variables***, which have to be distinguished from Python variables very carefully. Remember that we already encountered symbolic variables to create plots, see Section 2.3.3. To establish a symbolic variable, we type.

```
sage: x = SR.var('x')
sage: x
x
```

As illustrated above, the command SR.var('x') returns a symbolic variable with name x. From now on, this symbolic variable is stored in the Python variable x and can be utilized for further computations.

```
sage: x-3
x - 3
```

Alternatively, we can use the command var:

```
sage: var('y')
y
sage: y
y
```

This command creates the symbolic variable y and automatically assigns it to its Python pendant y.

R *Even though it is possible to assign symbolic variables to any other Python variable, it is not recommended to do so. In fact, if the symbolic variable x is not linked to the Python variable* x *the code might become quite confusing.*

```
sage: z = SR.var('x')
sage: 2*z - 1
2*x - 1
sage: x = 1
sage: 2*x - 1
1
sage: 2*z - x
2*x - 1
```

It is also possible to assign more than one symbolic variables in a row by typing.

```
sage: a, b, c, x = var('a␣b␣c␣x')
sage: a*x^2 + b*x + c
a*x^2 + b*x + c
```

To create a large number of symbolic variables at once, we can use the syntax

```
x = SR.var('x', n)
```

where n denotes a positive integer. This creates the symbolic variables $x_0, \ldots, x_{n-1}$. Each of them can be accessed via `x[i]`, where i is an integer between 0 and $n-1$:

```
sage: x = SR.var('x', 3)
sage: x[0] + 2*x[1] + 3*x[2]
x0 + 2*x1 + 3*x2
```

3.1.2 Symbolic Expressions

In the previous section, we discussed symbolic variables. In fact, symbolic variables are a special case of ***symbolic expressions***:

```
sage: x = SR.var('x')
sage: type(x)
<class 'sage.symbolic.expression.Expression'>
```

Even our examples from the previous chapter are symbolic expressions, although explicit numbers are involved:

```
sage: type(pi)
<class 'sage.symbolic.expression.Expression'>
sage: type(sqrt(2))
<class 'sage.symbolic.expression.Expression'>
```

Symbolic expressions literally represent *expressions* rather than mathematical objects or values. Hence, it can happen that Sage does not recognize two mathematically equal expressions.

```
sage: bool(arctan(1/abs(x)) == pi/2 - arctan(abs(x)))
False
```

R *The reason for the output above is the way the equality test* `bool(x==y)` *works: At first,* Sage *takes the difference between x and y. Then* Sage *tries to transform this expression to identical zero. If that succeeds, the method returns the value* `True`*. Instead of* `==`*, a bunch of other comparisons are allowed as well*[1]*:*

```
==, !=, <=, >=, <, >
```

When we want to assign a specific value to a symbolic variable in a given symbolic expression, we can ***substitute*** it as follows:

[1] Notice here that `!=` represents the mathematical $\neq$.

```
sage: x = var('x'); expr = cos(x)
sage: expr(x=1)
cos(1)
```

R *The semicolon* ; *allows us to write more separate commands in one single line.*

More complicated expressions can be treated with `subs` or `substitute` respectively. These methods allows us to substitute more variables at once and also replace one symbolic variable by another.

```
sage: a, x = var('a x')
sage: expr = cos(x+a) * (x+1)
sage: expr.subs(a == -x)
x + 1
sage: expr.subs(x == pi, a == pi)
pi + 1
```

We preferably use == instead of = to denote an equality in the *mathematical* sense, since = is reserved for assignments of `Python` variables. A thorough discussion about mathematical equalities is given in Section 3.2.2.

When dealing with mathematical expressions, we need to make sure that our symbolic variables live in the right domain. For that, we can use ***assumptions***. To state an assumption in `Sage` we use the function `assume`. All previous assumptions can be removed with `forget`.

```
sage: x = var('x')
sage: assume(x >= 0)
sage: bool(abs(x) == x)
True
sage: forget(x >= 0)
sage: bool(abs(x) == x)
False
```

Assumptions are stored globally and can be accessed by `assumptions` at any time:

```
sage: x = var('x'); assume(abs(x) < 1)
sage: assumptions()
[abs(x) < 1]
```

It is also possible to assume $n \in \mathbb{Z}$ as demonstrated in the following example:

```
sage: n = var('n'); assume(n, 'integer')
sage: sin(n*pi)
0
```

R *Keep in mind that we can use the help function, i.e. we can type* `assume?` *to obtain more detailed information, see Section 2.4.*

3.1.3 Simplifications

Even though symbolic expressions provide exact results, they sometimes appear unnecessarily complicated:

```
sage: x = var('x')
sage: cos(x)^2 + sin(x)^2
cos(x)^2 + sin(x)^2
```

Although it is well-known that this expression equals 1, Sage does not simplify this expression immediately.

To clean such expressions up, Sage provides several simplification methods. The most elementary method is given by `simplify`.

```
sage: x = var('x')
sage: (x^2 / x).simplify()
x
```

For more complicated expressions, the type of simplification has to be made explicit. Expressions containing factorials can be simplified with `simplify_factorial`:

```
sage: n = var('n')
sage: expr = factorial(n+1) / factorial(n)
sage: expr.simplify_factorial()
n + 1
```

Similarly, `simplify_rational` simplifies expressions containing fractions:

```
sage: x = var('x')
sage: expr = (x^2-1) / (x+1)
sage: expr.simplify_rational()
x - 1
```

To simplify trigonometric expressions, we have to use `simplify_trig` instead.

```
sage: x = var('x')
sage: (cos(x)^2 + sin(x)^2).simplify_trig()
1
```

Furthermore, it is possible to linearize a trigonometric expression using `reduce_trig`, or to anti-linearize it using `expand_trig`:

```
sage: x = var('x')
sage: (cos(x)^3).reduce_trig()
1/4*cos(3*x) + 3/4*cos(x)
sage: (sin(3*x)).expand_trig()
3*cos(x)^2*sin(x) - sin(x)^3
```

To deal with expressions such as square roots, logarithms or exponentials, we use `canonicalize_radical`:

```
sage: x, y = var('x␣y')
sage: (sqrt(abs(x)^2)).canonicalize_radical()
```

Common Simplifications	
Elementary Simplification	`simplify`
Simplify Factorials	`simplify_factorial`
Simplify Rationals	`simplify_rational`
Simplify Trig. Functions	`simplify_trig`
Linearize Trig. Functions	`reduce_trig`
Anti-linearize Trig. Functions	`expand_trig`
Full Simplification	`simplify_full, full_simplify`
Choose Branch and Simplify	`canonicalize_radical`

Table 3.1.: Common Simplifications provided by `Sage`

```
abs(x)
sage: (log(x*y)).canonicalize_radical()
log(x) + log(y)
```

R *The command* `canonicalize_radical` *is special among the simplification methods. In general, roots and logarithms have different* branches *in the complex plane. For instance, the expression* $\sqrt{x^2}$ *is not uniquely defined in* $\mathbb{C}$*. Namely, we can have* x or $-x$ *depending on the branch we have picked. The method* `canonicalize_radical` *chooses one branch by heuristics and simplifies on that.*

All aforementioned simplification are combined in the command `simplify_full`. This command invokes various simplifications in the following order:

`simplify_factorial` → `simplify_factorial`
→ `simplify_trig`
→ `simplify_rational`
→ `expand_sum`

We recommend to use the help function to get to know to the detais of the command `expand_sum`.

In Table 3.1 we have arranged a list of all simplifications discussed so far. These simplifications are already quite powerful, but not omnipotent.

```
sage: x = var('x')
sage: (x^2 + 2*x + 1).simplify_full()
x^2 + 2*x + 1
```

A reasonable simplification of this expression should be $(x+1)^2$. This example shows that "simplification" is not a well-defined term. We resolve this particular situation in the upcoming Section 3.2.1.

R *There is a significant difference between* ***symbolic*** *and* ***numerical*** *computations.*

A computer algebra system is made to deal with mathematical expressions by applying purely symbolic operations leading to exact results. However, we still have to treat

that with care. For example, the expression `a/a` *is* always *simplified to* `1`*. Thus, the program's solution of the equation* $ax = a$ *will return* $x = 1$ *without treating the special case* $a = 0$ *separately.*

The disadvantage of numerical calculations is that numerical methods only approximate to a given precision. Hence, many results are rounded and solutions are in general not exact. For example, a pocket calculator only manipulates integers exactly up to only twelve digits; larger numbers are rounded. Thus, the expression $(1+10^{50})-10^{50}$ *is evaluated "incorrectly" to* 0 *there.*

3.2 Elementary Algebra

3.2.1 Polynomial and Fractional Expressions

In mathematics, we have a vast pool of transformations to rewrite polynomials or fractions in a much simpler form. In the following, we describe the most common transformations on polynomials and fractions supported by `Sage`.

As discussed in Section 3.1.3, the aforementioned simplifications are limited when it comes to an expression like x^2+2x+1. If we want to simplify this expression to $(x+1)^2$, we have to invoke `factor`.

```
sage: x = var('x')
sage: (x^2 + 2*x + 1).factor()
(x + 1)^2
```

To turn it back to its expanded version, we call `expand`.

```
sage: x = var('x')
sage: ((x+1)^2).expand()
x^2 + 2*x + 1
```

Applying the command `factor_list` instead of `factor` returns a list of the factors together with their multiplicity.

```
sage: x, y = var('x␣y')
sage: (x^3 - y^3).factor_list()
[(x^2 + x*y + y^2, 1), (x - y, 1)]
```

In general, if a polynomial expression involves parentheses, we can first use `expand` to expand the polynomial and then `collect` to group terms according to the power of a given variable.

```
sage: x, y = var('x␣y')
sage: p = (x+y) * (x+1)^2
sage: p.expand()
x^3 + x^2*y + 2*x^2 + 2*x*y + x + y
sage: p.expand().collect(x)
x^3 + x^2*(y + 2) + x*(2*y + 1) + y
sage: p.expand().collect(y)
x^3 + 2*x^2 + (x^2 + 2*x + 1)*y + x
```

Transformations for Polynomial Expression p	
Expanding	`p.expand()`
Collect Terms w.r.t. x	`p.collect(x)`
Factorizing	`p.factor()`
List of Factors	`p.factor_list()`
Transformations for Rational Expression r	
Expanding	`r.expand()`
Factorizing	`r.factor()`
Combining Terms with the Denominator	`r.combine()`
Partial Fraction Decomposition w.r.t. x	`r.partial_fraction(x)`

Table 3.2.: Transforming Polynomials and Rationals.

These methods also apply to polynomials in more sophisticated sub-expressions of x such as $\sin(x)$.

```
sage: x, y =var('x␣y')
sage: f = (x + y + sin(x))^2
sage: f.expand().collect(sin(x))
x^2 + 2*x*y + y^2 + 2*(x + y)*sin(x) + sin(x)^2
```

In case of rational functions, the method `combine` groups terms with a common denominator.

```
sage: a, b, x, y = var('a␣b␣x␣y')
sage: r = (a*x+b)/(y^2) + (4*y)/(b-x) - (a*x)/(y^2) - (2*y + 2*a)
   /(b-x)
sage: r.combine()
-2*(a - y)/(b - x) + b/y^2
```

If the denominator of a rational function is a polynomial, we can obtain its partial fraction decomposition with respect to a given variable by invoking `partial_fraction`:

```
sage: x, y = var('x␣y')
sage: r = 1 / ((x^3+1)*(y^2-1))
sage: r.partial_fraction(x)
-1/3*(x - 2)/((x^2 - x + 1)*(y^2 - 1)) + 1/3/((y^2 - 1)*(x + 1))
sage: r.partial_fraction(y)
-1/2/((x^3 + 1)*(y + 1)) + 1/2/((x^3 + 1)*(y - 1))
```

A list of transformations regarding polynomials and fractions is given in Table 3.2.

3.2.2 Equations

Solving equations is often a tedious task, even if we use all the simplifications discussed in Section 3.1.3. For example it is well-known that there is no explicit solution formulas for polynomials of degree 5 or higher. Thus, it is much more convenient to let Sage solve the equations for us. However, there are various subtleties that have to be taken

care of. Since the symbol = is reserved for assignments, we use == for equations.

```
sage: z, phi = var('z phi')
sage: eq1 = z^2 - 2/cos(phi)*z + 5/cos(phi)^2 == 4
sage: eq2 = 3/cos(phi)*z - 3/sin(phi)^2 ==  - 2
```

Here, the equations are stored in the Python variable eq. Thus, an equation is an Python object. The left-hand side can be extracted with lhs, respectively the right-hand side with rhs.

```
sage: eq1.lhs()
z^2 - 2*z/cos(phi) + 5/cos(phi)^2
sage: eq1.rhs()
4
```

Similar to symbolic expressions, we can do basic arithmetic with equations.

```
sage: eq1 + eq2
z^2 + z/cos(phi) + 5/cos(phi)^2 - 3/sin(phi)^2 == 2
sage: eq1 * eq2
3*(z^2 - 2*z/cos(phi) + 5/cos(phi)^2)*(z/cos(phi) - 1/sin(phi)^2)
   == -8
sage: eq1 / eq2
1/3*(z^2 - 2*z/cos(phi) + 5/cos(phi)^2)/(z/cos(phi) - 1/sin(phi)
   ^2) == -2
```

Moreover, we can substitute symbolic expression in equations in the same way as we have already seen in Section 3.1.2.

```
sage: eq1.substitute(z == cos(phi))
cos(phi)^2 + 5/cos(phi)^2 - 2 == 4
```

To solve eq1 for z we use the solve command.

```
sage: solve(eq1,z)
[
z == -(2*sqrt(cos(phi)^2 - 1) - 1)/cos(phi),
z == (2*sqrt(cos(phi)^2 - 1) + 1)/cos(phi)
]
```

By default, Sage solves equations over the complex numbers.

```
sage: y = var('y')
sage: solve(y^4 == y, y)
[
y == 1/2*I*sqrt(3) - 1/2,
y == -1/2*I*sqrt(3) - 1/2,
y == 1,
y == 0
]
```

Yet, solve can not only be used for equations but also for systems of equations, e.g.

linear systems.

```
sage: x, y = var('x␣y')
sage: solve([x + y == 3, 2*x + 2*y == 6], x, y)
[
[x == -r1 + 3, y == r1]
]
```

Since the linear system in the above example is underdetermined Sage parameterizes the set of solutions by introducing a variable `r`. representing a real number. If the solution is parameterized by integers Sage introduces a variable `z`..

```
sage: x, y = var('x␣y')
sage: solve([sin(x) * cos(x) == 1/2, x + y == 0], x, y)
[
[x == 1/4*pi + pi*z32, y == -1/4*pi - pi*z32]
]
```

Last but not least, `solve` can also be applied to inequalities.

```
sage: solve(x^2 + x - 1 > 0, x)
[[x < -1/2*sqrt(5) - 1/2], [x > 1/2*sqrt(5) - 1/2]]
```

In many cases `solve` returns an explicit solution. However, it can happen that `solve` returns "only" numerical approximations of the solution.

```
sage: x, y = var('x␣y')
sage: solve([ x^2 * y == 24, x * y^2 == 18], x, y)
[
[x == 3.174802110817942, y == 2.381101376720901],
[x == (-1.5874010519682 + 2.749459273997207*I), y ==
   (-1.19055078897615 + 2.062094455497903*I)],
[x == (-1.5874010519682 - 2.749459273997207*I), y ==
   (-1.19055078897615 - 2.062094455497903*I)]
]
```

We discuss in Section 11.3 how to obtain exact solutions of polynomial systems.

Although the `solve` command is quite powerful it cannot deal with arbitrary complicated expressions. For example, we consider the following two expressions:

```
sage: a = var('a')
sage: c1 = (a+1)^2 - (a^2 + 2*a + 1)
sage: c2 = cos(a)^2 + sin(a)^2 - 1
```

Since both expressions simplify to 0, the equations $c1 \cdot x = 0$ and $c2 \cdot x = 0$ are both valid for any $x \in \mathbb{C}$.

```
sage: eq1 = c1*x == 0
sage: solve(eq1, x)
[x == x]
sage: eq2 = c2*x == 0
```

```
sage: solve(eq2, x)
[x == 0]
```

However, the `solve` command only returns for `eq1` the right solution. In `eq2` the expression `c2` is not recognized as being equal to 0 although Sage is able to simplify it correctly.

```
sage: c2.simplify_trig()
0
```

If the equation is too complicated, `solve` does not return an explicit solution.

```
sage:  solve(x^(1/x) == (1/x)^x, x)
[
(1/x)^x == x^(1/x)
]
```

But this does not imply that this equation can not be solved with Sage. It just means that we have to proceed differently to solve such an equations. How to avoid these pitfalls and solve more sophisticated equations in Sage is mainly discussed in Chapter 12 where we use `solve` to obtain solutions of differential equations .

If the solution of the equation in consideration is an explicit number a numerical approximation often suffices. In that case we can use `find_root` to obtain a numerical approximation of the root of a symbolic expression or function of one variable in a given interval. With `find_root` we can approximate solutions of equations that can not be solved with `solve`.

```
sage: expr = sin(x) + sin(2*x) + sin(3*x)
sage: solve(expr == 0,  x)
[
sin(3*x) == -sin(2*x) - sin(x)
]
sage: find_root(expr, 0.1, pi)
2.0943951023931957
```

Instead of `find_root` we could also use `root` which returns the roots together with their multiplicity. By default, solutions are returned exactly, but we can also specify the ring in which `roots` searches for solutions. If we declare the ring to be $\mathbb{R}$ (in Sage `RR`) or to be $\mathbb{C}$ (in Sage `CC`) then `roots` returns numerical approximations.

■ **Example 3.1** In the following example, we consider the equation $x^3+2x+1=0$. This equation has one real and two complex roots. The output of `roots` depends on the chosen ring.

```
sage: expr = x^3 - 4*x^2 + 6*x - 4
sage: expr.roots(x)
[(-I + 1, 1), (I + 1, 1), (2, 1)]
sage: expr.roots(x, ring=RR)
[(2.00000000000000, 1)]
sage: expr.roots(x, ring=CC)
```

Solving Scalar Equations	
Symbolic Solution	`solve`
Numerical Solving	`find_root`
Roots with Multiplicities	`roots`
List of Factors	`p.factor_list()`

Table 3.3.: Solving Scalar Equations

```
[(2.00000000000000, 1), (1.00000000000000 - 1.00000000000000*I, 1)
   , (1.00000000000000 + 1.00000000000000*I, 1)]
```

■

3.3 Calculus

In this section we give a short introduction to the basic useful functions for analysis and linear algebra. We will study these and more functions in greater detail in Part IV.

3.3.1 Symbolic Functions

Besides symbolic variables and symbolic expression we can also define ***symbolic functions***.

```
sage: x = var('x')
sage: f(x) = (2*x + 1)^3
```

Symbolic functions are useful to represent mathematical functions and can be modified in the same way as symbolic expressions.

```
sage: f(-3)
-125
sage: f.expand()
x |--> 8*x^3 + 12*x^2 + 6*x + 1
```

R *In Part II we discuss programming constructions called* `Python` ***functions***. *The difference between* `Python` *functions and symbolic functions is similar to the difference between* `Python` *variables and symbolic variables, see Section 3.1.1*

3.3.2 Sums

Expressions like $\sum_{i=a}^{b} f(i)$ are quite common in mathematics. In `Sage` such sums are written as

```
sum(f(i),i, a,b)
```

Recall that all used symbolic variables have to be defined beforehand. For example, the sum of the first n positive integer, i.e. $\sum_{k=1}^{n} k$, is defined as follows:

```
sage: k, n = var('k␣n')
```

```
sage: sum(k, k, 1, n)
1/2*n^2 + 1/2*n
```

As we can see, Sage simplifies this sum without further instructions. Many more simplifications are possible, e.g. sums of binomial coefficients

```
sage: k, n = var('k n')
sage: sum(k * binomial(n, k), k, 0, n)
2^(n - 1)*n
```

or geometric sums

```
sage: a, q, k, n = var('a q k n')
sage: sum(a * q^k, k, 0, n)
(a*q^(n + 1) - a)/(q - 1)
```

We could also try to compute the corresponding infinite sum. However, without further declarations we obtain an error.

```
sage: sum(a  * q^k, k, 0, infinity)
Traceback (most recent call last):
...
ValueError: Sum is divergent.
```

As we know, this infinite sum only converges if $|q| < 1$. Thus, we have to use the `assume` function, as explained in Section 3.1.1.

```
sage: a, q, k, n= var('a q k n')
sage: assume(abs(q) < 1)
sage: sum(a*q^k, k, 0, infinity)
-a/(q - 1)
```

■ **Example 3.2** The Riemann-zeta-function is given by

$$\begin{aligned}\zeta : \mathbb{C} &\to \mathbb{C} \cup \{\infty\},\\ s &\mapsto \sum_{k=1}^{\infty} k^{-s}.\end{aligned}$$

Using the introduced methods we can evaluate ζ at any point.

```
sage: k= var('k')
sage: sum(k^(-2), k, 1, Infinity)
1/6*pi^2
sage: sum(k^(-4), k, 1, Infinity)
1/90*pi^4
sage: sum(k^(-5), k, 1, Infinity)
zeta(5)
sage: numerical_approx(sum(k^(-5), k, 1, Infinity))
1.03692775514337
```

Since the evaluation at $s = 5$ only leads to the correct but "useless" result `zeta(5)` we

ask for a numerical approximation with arbitrarily large precision (depending on the available memory of the used computer). ∎

3.3.3 Sequences and Series

Sage is a good tool to study the behavior of sequences and series. For example, limits of sequences or functions can be easily computed with the `limit` method.

```
sage: x = var('x')
sage: limit((x^(1/3)-2) / ((x+19)^(1/3)-3), x = 8)
9/4
```

As we can see, `limit` has two arguments. The first is the function and the second is the point at which we want to take the limit. Instead of `limit` we can use its alias `lim`. The `limit` functions calculates by default the limit *to the left*. We can add the `dir` option as a third argument to specify the direction of the limit. To calculate limits to the left we choose the option `'minus'`, respectively the option `'plus'` to calculate the limit to the right.

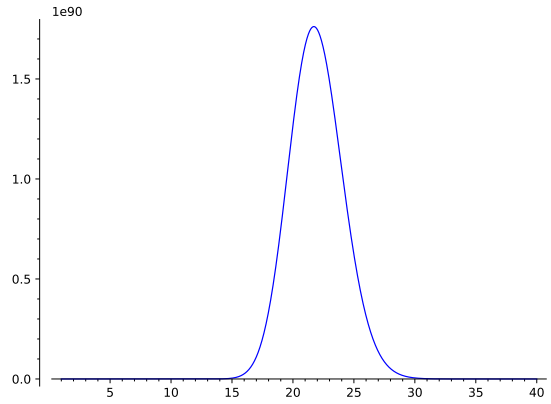
```
sage: x = var('x'); f(x) = (cos(pi/4-x)- tan(x))/(1-sin(pi/4+x))
sage: limit(f(x), x = pi/4)
Infinity
sage: limit(f(x), x = pi/4, dir='minus')
+Infinity
sage: limit(f(x), x = pi/4, dir='plus')
-Infinity
```

This and preceding introduced methods can be used to study the behavior of sequences. We demonstrate this in an example: We want to study the sequence $u_n = \frac{n^{100}}{100^n}$. First, we calculate its first five terms by hand (we learn in Section 4.3 how to shorten this).

```
sage: n = var('n'); u(n) = n^100 / 100^n
sage: u(1.)
0.0100000000000000
sage: u(2.)
1.26765060022823e26
sage: u(3.)
5.15377520732011e41
sage: u(4.)
1.60693804425899e52
sage: u(5.)
7.88860905221012e59
```

So far, it seems like $u(n)$ tends to infinity. To study the behavior of this sequence on a larger scale we plot the graph of the function $n \to u_n$.

```
sage: plot(u(n), n, 1, 40)
```



Now we see that $u(n)$ only increases up to around $n = 22$ and decreases afterwards. According to the graphic, the limit seems to be 0, and indeed, Sage confirms this suggestion.

```
sage: lim(u(n), n = Infinity)
0
```

To discuss the behavior of functions it is often useful to approximate them by a power series . The command `f(x).series(x == x0, n)` returns the power series expansion of the function $f(x)$ of order n at x_0.

```
sage: x = var('x')
sage: ((1+arctan(x))^(1/x)).series(x == 0, 3)
(e) + (-1/2*e)*x + (1/8*e)*x^2 + Order(x^3)
```

Here, Sage displays that there are terms of higher order. To extract the regular part of a power series expansion obtained by the above method, we use the `truncate` method.

```
sage: ((1+arctan(x))^(1/x)).series(x == 0, 3).truncate()
1/8*x^2*e - 1/2*x*e + e
```

Another common way to obtain an asymptotic expansion of a function is the Taylor series . To obtain the Taylor series of a function $f(x)$ around x_0 of order n we type `taylor( f(x), x, x0, n)`. It is also possible to calculate the Taylor series around $x_0 = \infty$.

```
sage: x = var('x')
sage: taylor( (x^3+x)^(1/3)-(x^3-x)^(1/3), x, Infinity, 2)
2/3/x
```

■ **Example 3.3** We use Sage to prove Machin's formula

$$\frac{\pi}{4} = 4\arctan\left(\frac{1}{5}\right) - \arctan\left(\frac{1}{239}\right). \tag{3.1}$$

First we observe that $4\arctan\left(\frac{1}{5}\right)$ and $\frac{\pi}{4} + \arctan\left(\frac{1}{239}\right)$ have the same tangent:

```
sage: tan(4 * arctan(1/5)).simplify_trig()
120/119
```

```
sage: tan(pi/4 + arctan(1/239)).simplify_trig()
120/119
```

This already proves Machin's formula (3.1) as tan is injective on the interval $(0, \pi)$. Having this formula at hand, we can approximate the value of π using the power series approximation of arctan.

```
sage: x = var('x')
sage: f = arctan(x).series(x, 10)
sage: f
1*x + (-1/3)*x^3 + 1/5*x^5 + (-1/7)*x^7 + 1/9*x^9 + Order(x^10)
sage: (16*f.subs(x == 1/5) - 4*f.subs(x == 1/239)).n()
3.14159268240440
sage: pi.n()
3.14159265358979
```

■

3.3.4 Derivatives and Integrals

To compute the derivative of a function in Sage we use `derivative` or its alias `diff`.

```
sage: derivative(sin(x^2), x)
2*x*cos(x^2)
```

It is also possible to differentiate symbolic functions. For example, we can display the chain-rule.

```
sage: f = function('f')(x); g = function('g')(x)
sage: diff(f(g(x)), x)
D[0](f)(g(x))*diff(g(x), x)
```

For higher derivatives we simply add the order as an additional argument.

```
sage: derivative(sin(x^2), x, 2)
-4*x^2*sin(x^2) + 2*cos(x^2)
```

Within Sage we can also compute partial derivatives of functions with more than one variable.

```
sage: x, y = var('x, y'); f(x, y) = x*y + y*sin(x^2) + e^(-y)
sage: derivative(f, x)
(x, y) |--> 2*x*y*cos(x^2) + y
sage: derivative(f, y)
(x, y) |--> x - e^(-y) + sin(x^2)
```

To compute higher or mixed derivatives we simply add every derivative as an additional argument. There, various syntax are possible.

```
sage: derivative(f, x ,x, y)
(x, y) |--> -4*x^2*sin(x^2) + 2*cos(x^2)
sage: derivative(f, x, 2, y)
```

```
(x, y) |--> -4*x^2*sin(x^2) + 2*cos(x^2)
```

■ **Example 3.4** We can use Sage to verify that

$$f(x,y) = \frac{1}{2}\ln(x^2+y^2)$$

is a harmonic function on $\mathbb{R}^2 \setminus \{0\}$.

```
sage: x, y = var('x y'); f(x, y) = ln(x^2+y^2) / 2
sage: delta = diff(f, x, 2) + diff(f, y, 2)
sage: delta.simplify_rational()
(x, y) |--> 0
```

■

■ **Example 3.5** Let $f : \mathbb{R}^2 \to \mathbb{R}$ defined by

$$f(x,y) = \begin{cases} xy\frac{x^2-y^2}{x^2+y^2}, & \text{if } (x,y) \neq (0,0), \\ 0, & \text{if } (x,y) = (0,0). \end{cases}$$

We show that $\partial_x\partial_y f(0,0) \neq \partial_y\partial_x f(0,0)$. Since the denominator of f vanishes at 0 we use the differential quotient.

```
sage: x, y, h = var('x, y, h')
sage: f(x,y) = x * y * (x^2 - y^2) / (x^2 + y^2)
sage: D1f(x, y) = diff(f(x, y), x) #D_y D_x f(0,0)
sage: limit((D1f(0,h)-0) / h, h = 0)
-1
sage: D2f(x, y) = diff(f(x, y), y) #D_x D_y f(0,0)
sage: limit((D2f(h,0)-0) / h, h = 0)
1
```

■

Definite integrals are computed in Sage with `integrate` or its alias `integral`. There are two different syntax available to compute an integral.

```
sage: integrate(sin(x), x, 0, pi/2)
1
sage: sin(x).integrate(x, 0, pi/2)
1
```

If one of the integral limits is infinity, Sage returns the result, if the integral converges and returns an error otherwise.

```
sage: integrate(1 / (1+x^2), x, -infinity,  infinity)
pi
sage: integrate(exp(-x^2), x, 0, infinity)
1/2*sqrt(pi)
```

```
sage: integrate(exp(-x), x, -infinity, infinity)
Traceback (most recent last call):
...
ValueError: Integral is divergent.
```

To compute an indefinite integral we simply drop the integral limits.

```
sage: integrate(1 / (1+x^2), x)
arctan(x)
```

■ **Example 3.6** For $x \in \mathbb{R} \setminus \{0\}$ we define

$$varphi(x) := \int_0^\infty \frac{x\cos(u)}{u^2+x^2}\mathrm{d}u.$$

We use Sage to find a simpler formulation of $\varphi(x)$ without an integral.

```
sage: u, x = var('u x'); f = x*cos(u) / (u^2+x^2)
sage: f.integrate(u, 0, Infinity)
Traceback (most recent last call):
...
ValueError: Computation failed since Maxima requested additional
   constraints;
Is x positive, negative or zero?
```

As we can see, Sage needs additional constraints on x to be able to calculate the integral. Hence, we treat the cases x positive and x negative separately using the `assume` function.

```
sage: assume(x > 0)
sage: f.integrate(u, 0, Infinity)
1/2*pi*e^(-x)
sage: forget(); assume(x < 0)
sage: f.integrate(u, 0, Infinity)
-1/2*pi*e^x
```

Hence, we have derived the following simpler expression for φ.

$$\varphi(x) = \frac{\pi}{2} \cdot \mathrm{sgn}(x) \cdot e^{-|x|}.$$

■

Instead of an explicit symbolic result we can, as usual, ask for a numerical approximation. This can be done with `numerical_approx`, see Section 2.3.1, or with `integral_numerical`, which returns a numerical approximation together with an estimate of the corresponding error.

```
sage: integral_numerical(exp(-x^2),0,infinity)
(0.8862269254527568, 1.714774436012769e-08)
sage: integrate(exp(-x^2),x,0,infinity)
1/2*sqrt(pi)
```

```
sage: (integrate(exp(-x^2),x,0,infinity) - integral_numerical(exp
    (-x^2),0,infinity)[0]).numerical_approx(digits = 20)
1.1829120040267315783e-15
```

■ **Example 3.7** We want to show the BBP formula

$$\sum_{n=0}^{\infty}\left(\frac{4}{8n+1}-\frac{2}{8n+4}-\frac{1}{8n+5}-\frac{1}{8n+6}\right)\left(\frac{1}{16}\right)^n=\pi. \tag{3.2}$$

Our strategy is to represent the partial sums $S_N=\sum_{n=0}^{N}\left(\frac{4}{8n+1}-\frac{2}{8n+4}-\frac{1}{8n+5}-\frac{1}{8n+6}\right)\left(\frac{1}{16}\right)^n$ as an integral. First, we use Sage to verify the identity

$$\int_0^{\frac{1}{\sqrt{2}}} f(t)\cdot\sum_{n=0}^{N} t^{8n}dt=S_N, \tag{3.3}$$

where $f(t)=4\sqrt{(2)}-8t^3-4\sqrt{(2)}t^4-8t^5$. To do so, we define

$$v_n=\left(\frac{4}{8n+1}-\frac{2}{8n+4}-\frac{1}{8n+5}-\frac{1}{8n+6}\right)\left(\frac{1}{16}\right)^n,$$
$$u_n=\int_0^{\frac{1}{\sqrt{2}}} f(t)t^{8n}dt.$$

and check (3.3) component wise.

```
sage: n, t = var('n,␣t')
sage: v(n) = (4/(8*n+1) - 2/(8*n+4) - 1/(8*n+5) - 1/(8*n+6))
    *(1/16)^n
sage: assume(8*n+1 > 0) #ensures that the denominator never
    vanishes.
sage: f(t) = 4*sqrt(2) - 8*t^3 -4*sqrt(2)*t^4 - 8*t^5
sage: u(n) = integrate(f(t)*t^(8*n), t, 0, 1/sqrt(2))
sage: (u(n) - v(n)).canonicalize_radical()
0
```

Using the linearity of the integral we obtain that

$$I_N=\int_0^{\frac{1}{\sqrt{2}}} f(t)\cdot\sum_{n=0}^{N} t^{8n}dt=\sum_{n=0}^{N} u_n=\sum_{n=0}^{N} v_n=S_N$$

Since $\sum_{n=0}^{\infty} t^{8n}$ converges on $\left[0,\frac{1}{\sqrt{(2)}}\right]$ we can interchange the limit and the integral and conclude that

$$\begin{aligned}\lim_{N\to\infty} S_N&=\int_0^{\frac{1}{\sqrt{2}}} f(t)\cdot\lim_{N\to\infty}\sum_{n=0}^{\infty} t^{8n}dt\\&=\int_0^{\frac{1}{\sqrt{2}}} f(t)\frac{1}{1-t^8}dt=:J\end{aligned}$$

Now, the infinite sum is "just" an integral which can be compute with Sage.

Methods in Calculus	
Symbolic Summation	`sum(f(k), k, kmin, kmax`
Limit	`limit(f(x), x=a`
Power Series of Order n at $x = a$	`f.series(x == a ,n`
Taylor of Order n at $x = a$	`taylor(f(x), x, a, n)`
Derivative	`diff(f(x), x)`
n-th Derivative	`diff(f(x), x, n)`
Indefinite Integral	`integrate(f(x), x)`
Definite Integral	`integrate(f(x), x, a, b)`
Numerical Integral	`integral_numerical(f(x), x, a, b)`

Table 3.4.: Useful Functions in Analysis

```
sage: J = integrate(f(t) / (1-t^8), t, 0, 1/sqrt(2))
sage: J.canonicalize_radical()
pi + 2*log(sqrt(2) + 1) + 2*log(sqrt(2) - 1)
```

It remains to tell Sage to combine the log sums.

```
sage: J.simplify_log().canonicalize_radical()
pi
```

Thus, we have shown the BBP formula (3.2). and can use this formula to approximate π.

```
sage: a = sum(v(n), n, 0, 10)
sage: a.numerical_approx(digits = 30)
3.14159265358979312961417056404
sage: pi.numerical_approx(digits = 30)
3.14159265358979323846264338328
sage: (pi.numerical_approx(digits = 30) - a.numerical_approx(
   digits = 30)).numerical_approx()
1.08848472819238e-16
```

These kind of formulas have been used to calculate in 2001 the $4 \cdot 10^{15}$-digit of π. ∎

3.3.5 Vector and Matrix computation

Sage provides various functions on vectors and matrices. Here, we only discuss the basic commands. We discuss more advanced methods on matrices and vectors in Chapter 10.

To construct a vector in Sage we use `vector`.

```
sage: vector([1,2,3,4])
(1, 2, 3, 4)
```

The euclidean scalar product is already implemented as `dot_product` and the corresponding norm is calculated via `norm`.

```
sage: v = vector([1,2,0,0]); w = vector([0,2,3,0])
```

Vector computations	
Vector Construction	`vector`
Standard Scalar Product $v \cdot w$	`v.dot_product(w)`
Euclidean Norm $\|v\|$	`command(v)`
Cross Product $v \times w$	`v.cross_product(w)`

Table 3.5.: Vector Computations

```
sage: v.dot_product(w)
4
sage: norm(v)
sqrt(5)
```

For elements of $\mathbb{R}^3$ Sage can compute the cross product of two vectors.

```
sage: v = vector([1,2,0]); w = vector([0,2,3])
sage: v.cross_product(w)
(6, -3, 2)
```

Next, we describe a few basic operations for matrix. First, we need to construct a matrix in Sage using the `matrix` function.

```
sage: A = matrix([[1, 2], [3, 4]]); A
[1 2]
[3 4]
```

Besides all basic calculations, we can also compute the inverse and the transpose.

```
sage: A^(-1)
[  -2    1]
[ 3/2 -1/2]
sage: A.transpose()
[1 3]
[2 4]
```

To solve a matrix equation $Ax = b$ we use `solve_right`. Similarly the matrix equation $xA = b$ is solved with `solve_left`.

```
sage: b = vector([1, 0])
sage: A.solve_right(b)
(-2, 3/2)
sage: A.solve_left(b)
(-2, 1)
```

In the special case $b = 0$, i.e. if we want to determine the kernel of the linear map A, we use `right_kernel` (for $Ax = 0$), respectively `left_kernel` (for $xA = 0$).

```
sage: A = matrix([[1, 0], [0, 0]])
sage: A.right_kernel()
Free module of degree 2 and rank 1 over Integer Ring
```

```
Echelon basis matrix:
[0 1]
sage: A.left_kernel()
Free module of degree 2 and rank 1 over Integer Ring
Echelon basis matrix:
[0 1]
```

Moreover, Sage can give us more information regarding the properties of a matrix A, like the vector space spanned by the columns, or by the rows, or the row echelon form. For the column and row space Sage provides an echelon basis matrix, where the *rows* are the basis vectors of the column and the row space respectively.

```
sage: A = matrix([[-2, 1, 1], [8, 1, -5]])
sage: A.column_space()
Free module of degree 2 and rank 2 over Integer Ring
Echelon basis matrix:
[1 1]
[0 2]
sage: A.row_space()
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[ 2  4 -2]
[ 0  5 -1]
sage: A.echelon_form()
[ 2  4 -2]
[ 0  5 -1]
```

It is also possible construct block matrices with`block_matrix`. The syntax is the same as for `matrix`, except that the arguments are matrices instead of numbers or symbolic expressions.

```
sage: block_matrix([[3*A, -A], [A, 4*A]])
[ -6   3   3|  2  -1  -1]
[ 24   3 -15| -8  -1   5]
[-----------+-----------]
[ -2   1   1| -8   4   4]
[  8   1  -5| 32   4 -20]
```

■ **Example 3.8** We investigate the matrix

$$A = \begin{pmatrix} 2 & -3 & 2 & -12 & 33 \\ 6 & 1 & 26 & -16 & 69 \\ 10 & -29 & -18 & -53 & 32 \\ 2 & 0 & 8 & -18 & 84 \end{pmatrix}.$$

First we determine a basis of the space of solutions of its kernel.

```
sage: A = matrix(QQ,[[2, -3, 2, -12, 33], [6, 1, 26, -16, 69],
    [10, -29, -18, -53, 32], [2, 0, 8, -18, 84]])
```

```
sage: A.right_kernel() #solving Ax = 0
Vector space of degree 5 and dimension 2 over Rational Field
Basis matrix:
[     1      0  -7/34   5/17   1/17]
[     0      1  -3/34 -10/17  -2/17]
```

Here, the `QQ` in the beginning means that we are calculating in the rational field. As a consequence the results are returned as irreducible fractions. This is a first example of the usage of computations domains which are discussed in Chapter 8.

Next, we determine a basis of the space F generated by the columns of A. This can be done directly using the method `column_space`.

```
sage: A.column_space()
Vector space of degree 4 and dimension 3 over Rational Field
Basis matrix:
[       1        0        0 1139/350]
[       0        1        0    -9/50]
[       0        0        1   -12/35]
```

To obtain a defining equation for F we first calculate the left kernel of A, i.e. the solution space for $xA = 0$.

```
sage: A.left_kernel()
Vector space of degree 4 and dimension 1 over Rational Field
Basis matrix:
[        1  -63/1139 -120/1139 -350/1139]
```

Since F is the orthogonal complement of the left kernel of A it follows that F is the hyperplane of $\mathbb{R}^4$ defined by the equation $1139x - 63y - 120z - 350t = 0$. ∎

For the special case of square matrices Sage can calculate eigenvalues and -vectors, and various normal forms. At this point we only introduce a few basic commands. More advanced methods are discussed in 10.

The determinant and the characteristic polynomial of a square matrix are calculated in Sage as follows.

```
sage: A = matrix([[2, 4, 3],[-4, -6, -3],[3 , 3, 1]])
sage: A.determinant()
4
sage: A.characteristic_polynomial()
x^3 + 3*x^2 - 4
```

While `eigenvalues` returns the eigenvalues of a matrix the method `eigenvalues_right` returns triplets consisting of the eigenvalue, the corresponding eigenvector and its algebraic multiplicity.

```
sage: A.eigenvalues()
[1, -2, -2]
sage: A.eigenvectors_right()
[(1, [
```

```
(1, -1, 1)
], 1), (-2, [
(1, -1, 0)
], 2)]
```

In the above example there is only one eigenvector shown for the second eigenvalue although its algebraic multiplicity is two. Thus, we conjecture that the matrix A is not diagonalizable. To prove this small conjecture we compute its Jordan normal form J with Sage. Adding the option `transformation = True` returns the transformation matrix T, i.e. $T^{-1}AT = J$.

```
sage: A.jordan_form()
[ 1| 0  0]
[--+-----]
[ 0|-2  1]
[ 0| 0 -2]
sage: A.jordan_form(transformation=True)
([ 1| 0  0]
[--+-----]
[ 0|-2  1]
[ 0| 0 -2], [ 1  1  1]
[-1 -1  0]
[ 1  0 -1])
```

■ **Example 3.9** As a small comparison, we shortly show, how `eigenvectors_right()` deals with diagonalizable matrices.

```
sage: B = matrix([[1, 0, 0], [0, 2, 0], [0, 0, 2]])
sage: B.eigenvectors_right()
[(1, [
(1, 0, 0)
], 1), (2, [
(0, 1, 0),
(0, 0, 1)
], 2)]
```

Here, `eigenvectors_right()` returns two eigenvectors for the second eigenvalue. ■

Usual Methods on Matrices	
Matrix Constructor	`matrix`
Transpose Matrix A	`A.transpose()`
Solve $Ax = b$	`A.solve_right(b)`
Solve $xA = b$	`A.solve_left(b)`
Right/Left Kernel	`right_kernel`, `left_kernel`
Row Echelon Form	`echelon_form`
Column-Generated Vector Space	`column_space`
Row-Generated Vector Space	`row_space`
Block Matrix	`block_matrix`
Operations for Square Matrices	
Inverse of A	`A^(-1)`
Determinant	`determinant`
Characteristic Polynomial	`characteristic_polynomial`
Minimal Polynomial	`minimal_polynomial`
Eigenvalues	`eigenvalues`
Eigenvectors with Properties	`eigenvectors_right`
Jordan Normal Form	`jordan_form`

Table 3.6.: Matrix Operations

II Programming and Data Structure

4 Algorithmics 49
4.1 Procedures and Functions
4.2 Conditionals
4.3 Loops
4.4 Output

5 Lists and Other Data Structures 59
5.1 Lists
5.2 Character Strings
5.3 Finite Sets
5.4 Dictionaries
5.5 More on Data Structures

4. Algorithmics

In Part I we have only done one-line mathematical calculations. In this chapter we want to demonstrate how we can program functions using sequences of instructions within Sage. The Sage computer algebra system is in fact an extension of the Python computer language [1]. Hence, we can use, with a few exceptions, the Python programming constructs. We explain how to use classical Python programming structures, like loops and lists, in Sage without requiring to know the Python language. These constructions will be familiar to those who are fluent in Python or another programming language.

The paradigm of structured programming consists in designing a computer program as a finite sequence of instructions, which are executed in order. Those instructions can be ***atomic*** or ***composed***:

- ***atomic*** instructions are single commands, e.g. the assignment of a variable or a result output.
- ***composed*** instructions are made up from several instructions which are themselves atomic or composed, e.g. conditionals or loops.

4.1 Procedures and Functions

In Chapter 3 we have used many predefined procedures like `sum`, `integrate`,... . But, as in other computer languages, we can define our own procedures and functions using the `def` command. A *Python* ***function***, respectively ***procedure***, is a sub-program with several arguments, which returns a result. For example, we can define the function $(x,y) \mapsto x^2+y^2$ as a Python function which can be evaluated at any point (x,y).

```
sage: def fct(x,y):
```

[1] The Sage version considered here uses Python 3

```
....:     return x^2 + y^2
sage: fct(3, 5)
34
sage: a = var('a')
sage: fct(a, 2*a)
5*a
```

The above example illustrates the syntax of `def`:

```
def 'name'('arguments'):
    'instructions'
    return 'result'
```

The indentation here in the line after the colon : is essential. Only the indented instructions belong to the body of the Python function `'name'`. Usually, a Python function ends with the `return` command returning the result. In the following sections we describe various program constructions that can be used to design diverse Python functions.

■ **Example 4.1** For two real numbers x, y the three classical Pythagorean means are

- the arithmetic mean $\frac{x+y}{2}$,
- the geometric mean $\sqrt{x \cdot y}$,
- the harmonic mean $2\left(\frac{1}{x}+\frac{1}{y}\right)^{-1}$.

We write a Python procedure returning these three means at once.

```
sage: def means(x,y):
....:     a = (x+y) / 2 #arithmetic mean
....:     g = sqrt(x * y) #geometric mean
....:     h = 2 * (1/x + 1/y)^(-1) #harmonic mean
....:     return a, g, h
sage: means(3, 5)
(4, sqrt(15), 15/4)
sage: means(6, 18)
(12, 6*sqrt(3), 9)
```

■

If a Python function only contains one line of instruction we can also use the ***lambda construction***. This shortens the code but might worsen the readability. Below we redefine the Pythagorean means from the above example using the lambda construction.

```
sage: amean = lambda x, y: (x+y) / 2
sage: gmean = lambda x, y: sqrt(x * y)
sage: hmean = lambda x, y: 2 * (1/x + 1/y)^(-1)
sage: amean(3, 5), gmean(3, 5), hmean(3, 5)
(4, sqrt(15), 15/4)
```

4.2 Conditionals

The ***conditional*** is an important instruction, which enables us to execute some construction depending on the result of a boolean condition. There are two syntax possible:

```
if 'a condition':
    'instructions'
```

```
if 'a condition':
    'instructions'
else:
    'other instructions'
```

For example, we know that a geometric sum $\sum_{k=0}^{\infty} q^k$ converges if and only if $|q| < 1$. In Section 3.3.2 we resolved this problem by using the command `assume(abs(q) < 1)`. Another possibility is to use a conditional. This has the advantage that no error is raised if q does not satisfy the assumption.

```
sage: def geo_sum(q):
....:     if abs(q) < 1:
....:         k  = var('k')
....:         return sum(q^k, k, 0, oo)
....:     else:
....:         return 'This sum diverges.'
sage: geo_sum(1/2)
2
sage: geo_sum(2)
This sum diverges.
```

The `if-else` construction also allows nested tests in the `else` branch. There the code can be shortened using `elif`. Hence, nested conditionals can be formulated in the following two different ways.

```
if 'cond1':
    'inst1'
else:
    if 'cond2':
        'inst2'
    else:
        if 'cond3':
            'inst3'
            else:
                'inst4'
```

```
if 'cond1':
    'inst1'
elif 'cond2':
    'inst2'
elif 'const3':
    'inst3'
else:
    'inst4'
```

■ **Example 4.2** It is well-known that the roots of the equation $ax^2 + bx + c = 0$ are given by

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Below we write a procedure that only returns the real roots of a quadratic equations with real coefficients. We have to distinguish three cases: there are no real roots, if $b^2 < 4ac$, one real root if $b^2 = 4ac$ and two real roots if $b^2 > 4ac$. Since this requires a nested conditional construction we use the `elif` syntax.

```
sage: def realroot(a,b,c):
....:     if b^2 < 4*a*c:
....:         return('There␣are␣no␣real␣roots.')
....:     elif b^2 == 4*a*c:
....:         x=-b/(2*a)
....:         return(x)
....:     else:
....:         x1 = (-b + sqrt(b^2 - 4*a*c))/(2*a)
....:         x2 = (-b - sqrt(b^2 - 4*a*c))/(2*a)
....:         return(x1, x2)
sage: realroot(1, 2, 1)  #Apply to x^2 + 2x + 1 = 0
-1
sage: realroot(1, 0, 1)  #Apply to x^2 + 1 = 0
There are no real roots.
sage: realroot(1, 1, -2) #Apply to x^2 + x - 2 = 0
(1, -2)
```

■

4.3 Loops

A loop is a program construct that performs the same calculation several times. Below we describe the two most common loops: The ***enumerative loops*** and the ***while loops***. Roughly speaking, the main difference is that an enumerative loop performs the same calculation for a fixed number of times, while the `while` loop performs the same calculation as long as a given condition is satisfied.

An ***enumerative loop***, also called ***for loop***, performs the same calculation for a fixed number of times, e.g. for all integer values of an index $k \in [a,b]$. For example, we can use an enumerative loop to calculate the first square numbers.

```
sage: for k in srange(5):
....:   print(k^2)
0
1
4
9
16
```

Here, we obtained the square numbers for $0 \leq k \leq 4$, because

```
sage: srange(5)
[0, 1, 2, 3, 4]
```

The above `for` loop performs the calculation "taking squares" 5-times, where the variable k runs from 0 to 4. Moreover, the code in the second line is indented. The indentation means that the line `print(k ^2)` belongs to the body of the `for` loop. As for `Python` functions, the indentation is essential for the program structure in `Sage`, as can be seen in the following example.

■ **Example 4.3** We consider the following `for` loop.

```
sage: S = 0
sage: for k in srange(4):
....:     S = S + k
....:     S = 2 * S
sage: S
22
```

Here, both lines below the `for` loop are indented. Thus, both instructions are performed whenever the loop is activated, i.e.

$$S = ((((0+0)\cdot 2+1)\cdot 2+2)\cdot 2+3)\cdot 2 = 22.$$

Without the indentation the `for` loop does not perform the instructions.

```
sage: S = 0
sage: for k in srange(4):
....:     S = S + k
sage: S = 2 * S
sage: S
12
```

Here, the instruction `S = 2 * S` is only performed once after the loop, i.e.

$$S = (0+0+1+2+3)*2 = 12.$$

■

At this point, we want to describe the modifications of `srange` in more detail. The command `srange(n)` returns the list of the integers $(0, \ldots, n-1)$. However, we can modify more than just the endpoint. The extended syntax `srange(i, j, k)` with $i, j, k \in \mathbb{R}$ returns the list $(i, i+k, i+2k, \ldots, i+n\cdot k)$, where $i+n\cdot k < j \leq i+(n+1)\cdot k$. We can also use negative values for k to obtain a decreasing list. Moreover, adding the option '`include_endpoint = True`' adds the endpoint j to the list constructed by `srange(i, j ,k)`.

```
sage: srange(1, 5, include_endpoint = True)
[1, 2, 3, 4, 5]
sage: srange(0, 10, 2)
[0, 2, 4, 6, 8]
sage: srange(10, 0, -2)
[10, 8, 6, 4, 2]
sage: srange(1, 5/2, 1/3)
[1, 4/3, 5/3, 2, 7/3]
```

One common application of `for` loops is the calculation of terms of a recurrent sequence, e.g. the term u_{10} of the recurrent sequence $(u_n)_n$ with

$$u_0 = 0,\ u_n = \frac{1}{1+u_{n-1}^2}.$$

```
sage: u = 0
sage: for k in srange(1,10, include_endpoint = True):
....:     u = 1 / (1+u^2)
sage: numerical_approx(u)
0.677538380912204
```

However, there are situations where a calculation should be performed until a specific condition is satisfied but we do not know a priori how many repetitions we need. For these kind of problems we use the ***while loop***. Roughly speaking, the `while` loop performs the instructions in its body as long as a given condition is fulfilled, e.g. for a given number $x > 1$ we can determine the smallest integer n with $x < 2^n$ as follows.

```
sage: x = 10^4; u = 1; n = 0
sage: while u <= x:
....:     n += 1
....:     u = 2 * u
sage: n
14
```

The above `while` loop executes the indented instructions as long as the condition $2^n \leq x$ is fulfilled. The loop terminates as soon as this condition is no longer fulfilled, i.e. when $x < 2^n$. The body of the loop, i.e. the intended instructions, are never carried out if the condition is not satisfied. In particular, a `while` loop is not carried out at all if the condition is already false at the first test.

R *For short loop instructions it is also possible to write everything on a single line after the colon* :.

```
sage: x = 10^4; u = 1; n = 0
sage: while u <= x: n = n +  1; u = 2 * u
sage: n
14
```

The `for`and the `while` loop repeat the same instructions under fixed conditions. To be more flexible, we can use `break` to exit the loop earlier or go directly to the next iteration using `continue`. These commands allow us to check the terminating condition at every place in the loop.

Below, we determine the smallest positive integer x satisfying $\log(x+1) \leq \frac{x}{10}$ using four different loop constructions. In this first attempt a `for` loop tries the first 100 integers and stops as soon as a solution is found.

```
sage: x = 1.0
sage: for x in srange(1,100, include_endpoint= True):
....:     if log(x+1) <= x/10: break
sage: x
37
```

However, if no solution is found, we do not know if there is no solution at all or if there is a solution for larger values of x. Hence, we use a `while` loop in our next construction to

look for the smallest solution. The downside here is that this loop might never terminate if the condition is never fulfilled, i.e. if $\log(x+1) > \frac{x}{10}$ for all $x \in \mathbb{N}$.

```
sage: x = 1.0
sage: while log(x+1) > x / 10:
....:     x += 1
sage: x
37.0000000000000
```

This problem is resolved in the next construction by setting the upper bound $x < 100$. However, we again encounter the same problem as in the first attempt.

```
sage: x = 1.0
sage: while log(x+1) > x/10 and x < 100:
....:     x += 1
sage: x
37.0000000000000
```

Although the next loop is unnecessarily complicated it illustrates the usage of `continue` in combination with `break`.

```
sage: x = 1.0
sage: while True:
....:     if log(x+1) > x/10:
....:         x = x+1
....:         continue
....:     break
sage: x
37.0000000000000
```

4.3.1 Example: A Sequence with an unknown Limit

It is still an open problem whether the sequence

$$a_n = \frac{1}{n^2 \sin(n)},\ n \in \mathbb{N}.$$

converges or not. It is conjectured that this sequence tends to zero as n tends to infinity, but no proof is known yet. We use the methods describe above to study the behavior of this sequence for large values of n. First, we define the sequence in consideration in Sage.

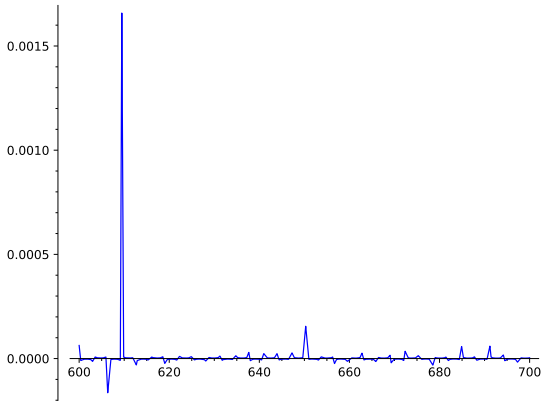
```
sage: n = var('n')
sage: seq(n) =  1 / (n^2 * sin(n))
```

and try to calculate the limit.

```
sage: n = var('n')
sage: limit(seq, n = Infinity)
und
```

The output `und` means "undefined". Since `Sage` cannot compute the limit, we need to try different methods. To obtain more insight in the problematic of this sequence we plot it for $600 \leq n \leq 700$.

```
sage: plot(seq(n), n, 600, 700)
```



We observe that the sequence (a_n) behaves chaotic and does not uniformly approaches 0.

In a next step, we approach this problem from a more analytic point of view. Recall that a sequence (a_n) tends to 0 if and only if for all $\varepsilon > 0$ there is an $N \in \mathbb{N}$ such that $|a_n| < \varepsilon$ for all $n \geq N$. Since we need to deal with finite objects in `Sage` we have to "approximate" this definition in the following way. Instead of all integers $n \geq N$ we only check all integers $N \leq n < N + I_{\max}$ where $I_{\max}$ is a fixed number.

Hence, our goal is to write a program that estimates such an N for a given ε and $I_{\max}$. To do so, we first write a small sub-program `check_epsilon` that returns the value

$$\min\{n \in \mathbb{N} |\ N \leq n < I_{\max} + N \text{ and } |a_n| \geq \varepsilon\}$$

given a sequence (a_n), an $\varepsilon > 0$, a fixed number N and a fixed number $I_{\max}$ of maximal iterations.

```
sage: def epsilon_check(seq, eps, N, I):
....:     for n in srange(N, N+I):
....:         if abs(seq(n)) >= eps:
....:             return n
....:         return Infinity
```

Here we used that the minimum of the empty set is infinity. Now, we use our function `epsilon_check` to write the function `N_estimate` that estimates an N for a given ε and $I_{\max}$ using a `while` loop.

```
sage: def N_estimate(seq, eps, I):
....:     N = 1
....:     while(True):
....:         eps_check = epsilon_check(seq, eps, N, I)
....:         if eps_check == Infinity:
```

```
....:             break
....:         N = eps_check + 1
....:     return(N)
```

In a final step we use the function `N_estimate` to further investigate the behavior of the sequence $(a_n)_n$.

```
sage: N_estimate(seq, 0.1, 1000)
4
sage: N_estimate(seq, 0.001, 1000)
33
```

Although Sage cannot prove that $(a_n)_n$ converges to 0 as n tends to infinity, it can give us evidence that the conjecture might be true.

4.4 Output

This section is devoted to the main output command `print`. By default, the arguments are printed one after the other, separated by spaces.

```
sage: print(2^2, 3^2, 4^2); print(5^2, 6^2)
4 9 16
25 36
```

The comma separates the various print commands and tells `print` to continue the output on the same line. If we want to print text we can either use quotation marks `'...'` or the `str` function. The command `str` can also be used to transform numbers into character strings and character strings can be concatenated with +, see Section 5.2 for more informations to character strings.

```
sage: print(42, 3.14)
42 3.14
sage: print(42 + 3.14)
45.1400000000000
sage: print(str(42) + str(3.14))
423.1400000000000
```

The output can be modified with the `%.d` placeholder and the `%` operator.

```
sage: for k in srange(1,6):
....:     print('%2d! = %3d' % (k, factorial(k)))
 1! =   1
 2! =   2
 3! =   6
 4! =  24
 5! = 120
```

The `%` operator replaces the expression to its right (here `((k, factorial(k)))` into the character strings to its left, in place of the placeholder (% 2d) and (% 3d). Here, `3d` means that the placeholder `% 3d` gets at least three characters. If necessary, those are

replaced with white spaces. Analogously the `%.4f` placeholder allows only an output with maximal 4 digits after the decimal point.

```
sage: print('%.4f', % numerical_approx(pi))
3.1416
```

5. Lists and Other Data Structures

In this chapter we describe various data structures available in Sage. The function `type()` returns the type of an object. To convert an object `obj` to a type `typ` is simply done by `typ(obj)`.

```
sage: type(42) #this is a Sage integer
<class 'sage.rings.integer.Integer'>
sage: type(int(42)) #convert it to a Python integer
<class 'int'>
```

The standard types in Sage are `bool`, `int`, `list`, `tuple` `set`, `str`, `dict`. More specifically, the type `bool` has the two values: `True` and `False`. The `Python` types `int` and `long` are used to represent integers of limited size. In contrast to that, Sage uses its own type `integer` which also allows exact calculation with fractions. The remaining types, i.e. lists, tuples, sets, character strings and dictionaries are described in more detail in the following sections.

5.1 Lists

A list in computer science corresponds to an n-tupel in mathematics. Every object in a list is determined by its value and its position, i.e. $(a,b) \neq (b,a)$, contrary to a set.

In Sage a list is constructed by surrounding its elements with square brackets `[...]`. Accordingly, the empty list is defined as `[]`. To construct large integer list we use the `..` operator

```
sage: L1 = [1, 2, 3];L1
[1, 2, 3]
sage: []
[]
```

```
sage: L2 = [1..5, 9..12]
[1, 2, 3, 4, 5, 9, 10, 11, 12]
\end{Sage}
The \defstyle{length} of a list is the number of elements and can
   be returned with \command{len}.
\begin{CustomSage}
sage: len(L1), len([]), len(L2)
(3, 0, 9)
```

The elements of a list are numbered consecutively starting with 0. The element of index k in a list `L` is accessed via `L[k]`. We can also use negative indices to access end-of-list elements, e.g. `L[-1]` refers to the last element of a list L. Hence, list elements are indexed in two different ways.

$$\texttt{L} = [l_0, l_1, \ldots, l_{n-1}] = [l_{-n}, l_{1-n}, \ldots, l_{-1}].$$

The command `L[p:q]` extracts the sub-list `[L[p], L[p+1], ..., L[q-1]]`, which is empty if $q \leq p$. Moreover, `L[p:]` is equivalent to `L[p:len(L)]`, and `L[:q]` to `L[0:q]`.

```
sage: L = [11, 22, 33, 44, 55, 66]
sage: L[3], L[5]
(44, 66)
sage: L[-1], L[-3]
(66, 44)
sage: L[1:4]
[22, 33, 44]
sage: L[-4:4]
[33, 44]
sage: L[:3], L[2:]
([11, 22, 33], [33, 44, 55, 66])
```

Using these commands each list element `L` can be extracted and modified. The same can be done with sub-lists. This can also change the length of a list, e.g. when we delete a sub-list.

```
sage: L = [0, 1, 2, 3, 4, 5]
sage: L[0] = L[0] + 4; L #change an element
[4, 1, 2, 3, 4, 5]
sage: L[2:5] = [4, 3, 2, 1]; L #change sub-list
[4, 1, 4, 3, 2, 1, 5]
sage: L[1:3] = []; L #delete sublist
[4, 3, 2, 1, 5]
```

To examine lists we can use the operator `in` to check whether an element is contained in a list and the comparison operator == to compare two lists element-wise.

```
sage: L = [1, 2, 4, 8, 16]
sage: L[2: len(L)-1] == L[1-len(L): -2]
False
```

```
sage: [4 in L, 5 in L]
[True, False]
```

5.1.1 Global List Operations

Two lists can be concatenated with +. Another way to concatenate lists is the iterated concatenation which is done by "multiplying" the list with an integer.

```
sage: L1 = [11, 22, 33]; L2 = [44, 33, 22]
sage: L1 + L2
[11, 22, 33, 44, 33, 22]
sage: 3 * L1
[11, 22, 33, 11, 22, 33, 11, 22, 33]
```

In particular, the numbering of list elements is such that the concatenation of the two sub-lists `L[:k]` and `L[k:]` reconstructs the original list `L`.

One way to modify all elements of a list at once is to apply a function to all of its elements using the `map` command. For example, we can apply cos to a list of angles.

```
sage: list(map(cos, [0, pi/4, pi/3, pi/2]))
[1, 1/2*sqrt(2), 1/2, 0]
```

Since `map` returns an iterator and not a list we used `list()` to convert the result of `map` to a list. Instead of predefined functions we can also combine `map` with a `lambda` construction, see Section 4.1.

```
sage: list(map(lambda t: cos(t), [0, pi/4, pi/3, pi/2]))
[1, 1/2*sqrt(2), 1/2, 0]
```

The `filter` command builds an iterator out of those list elements satisfying a given condition, e.g. all integers up to 45 that are prime. As before we use `list()` to convert the result to a list.

```
sage: list(filter(is_prime, [1..45]))
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43]
```

The test condition can also be defined inside the `filter` command, e.g. using a `lambda` construction, see Section 4.1. For example, we can write a procedure returning a list of all divisors of a given number p.

```
sage: def divisor(p):
....:     I = filter(lambda n: p % n == 0, [1,..,p])
....:     return list(I)
sage: divisor(42)
[1, 2, 3, 6, 7, 14, 21, 42]
sage: divisor(123)
[1, 3, 41, 123]
```

The ***comprehension*** form `[.. for .. x .. in ..]` is another way to construct a list with elements satisfying some conditions. For example, combining the comprehension form with an `if` condition yields an equivalent construction to `filter`.

```
sage: list(filter(is_prime, [1..45]))
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43]
sage: [p for p in [1..45] if is_prime(p)]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43]
```

Below, we show how each of the introduced commands can be used to construct a list of all odd integers from 1 to 25.

```
sage: list(map(lambda n: 2*n +1, [0..12]))
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25]
sage: list(filter(lambda n: n % 2 == 1, [1..25]))
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25]
sage: [2*n+1 for n in [0..12]]
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25]
```

These commands can also be combined, e.g. to construct a list of prime numbers congruent to 1 modulo 4.

```
sage: list(filter(is_prime, [4*n+1 for n in [0..20]]))
[5, 13, 17, 29, 37, 41, 53, 61, 73]
```

In the above command Sage first constructs a list of all numbers congruent to 1 modulo 4 between 1 and 81. Then, Sage applies `filter` to this list and deletes all numbers that are not prime using the `is_prime` test.

■ **Example 5.1** We use the above described list operations to compute the first four derivatives of xe^x, once using `map` and once using the comprehension form.

```
sage: def f(n): #calculates n-th derivative of x*e^x
....:     return factor(diff(x*exp(x), n*[x]))
sage: list(map(lambda n: f(n),  [0..4]))
[x*e^x, (x + 1)*e^x, (x + 2)*e^x, (x + 3)*e^x, (x + 4)*e^x]
sage: [f(n) for n in [0..4]]
[x*e^x, (x + 1)*e^x, (x + 2)*e^x, (x + 3)*e^x, (x + 4)*e^x]
```

■

Next, we introduce the `reduce` command which operates by associativity from left to right on the elements of a list. The command `reduce` requires two arguments: a two-parameter function describing the composition of two list elements, and a list.

```
sage: reduce(lambda x, y: 10*x + y, [1, 2, 3, 4])
1234
```

In the above example, `reduce` applies the composition $x \star y = 10x + y$ to all elements of the list `[1, 2, 3, 4]` from left to right, i.e.

$$((1 \star 2) \star 3) \star 4) = (12 \star 3) \star 4 = 123 \star 4 = 1234.$$

So far, `reduce` only makes sense for lists with at least two elements. For one-element lists or the empty list we have to add a third argument describing the image of the

empty list. This element should corresponds to the neutral element of the considered composition as the result might change otherwise.

```
sage: reduce(lambda x, y: 10*x +y, [9, 8, 7, 6], 1)
19876
```

■ **Example 5.2** We can use `reduce` to compute the product of odd integers.

```
sage: L = [2*n+1 for n in [0..9]]
sage: reduce(lambda x, y: x*y, L, 1)
654729075
```

Here we put `1` as a third element which is the neutral element for multiplication. ■

As seen in the above example, `reduce` can be used to compute large sums or product. For these two common composition `Sage` provides the predefined functions `add`[1] for addition and `prod` for multiplication.

```
sage: L = [2*n+1 for n in [0..9]]
sage: prod(L) #product of all list elements
654729075
sage: add(L) #sum of all list elements
100
```

For tests on list elements `Sage` provides the functions `any` and `all`. Their evaluation terminates as soon as the result `True`, respectively `False` is returned. The remaining list elements are not evaluated.

```
sage: fct = lambda x: 4/x == 2
sage: all(fct(x) for x in [2, 1, 0])
False
sage: any(fct(x) for x in [2, 1, 0])
True
```

The list constructions described so far can be combined to construct various "products" of two lists deoending on the explicit combination in used. In the following example, the leftmost `for` operator corresponds to the outermost loop.

```
sage: [[x, y] for x in [1..3] for y in [6..8]]
[[1, 6], [1, 7], [1, 8], [2, 6], [2, 7], [2, 8], [3, 6], [3, 7],
   [3, 8]]
```

In nested comprehension forms, every comprehension form produces its own list.

```
sage: [[[x, y] for x in [1..3]] for y in [6..8]]
[[[1, 6], [2, 6], [3, 6]], [[1, 7], [2, 7], [3, 7]], [[1, 8], [2,
   8], [3, 8]]]
```

If `map` has several lists as arguments it takes one element of each list in turn.

```
sage: list(map(lambda x, y: (x, y), [1..3], [6..8]))
```

[1] This command is different from the `sum` command, which looks for a symbolic expression of a sum.

```
[(1, 6), (2, 7), (3, 8)]
```

Another way to combine two lists is the `zip` command which groups several lists entry-wise.

```
sage: L1 = [0..2]; L2=[5..7]
sage: list(zip(L1, L2))
[[0, 5], [1, 6], [2, 7]]
```

As seen above it is possible that the elements of a list are lists themselves. This can be iterated various times. The number of nested lists is called the ***level*** of a list. The command`flatten` reduces the level of a list. The degree of reduction can be customized with the option `max_level`.

```
sage: L = [[1, 2, [3]], [4, [5, 6]], [7, [8, [9]]]]
sage: flatten(L, max_level = 1)
[1, 2, [3], 4, [5, 6], 7, [8, [9]]]
sage: flatten(L, max_level = 2)
[1, 2, 3, 4, 5, 6, 7, 8, [9]]
sage: flatten(L) #flattens everything
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

5.1.2 List Manipulation

Here, we introduce various methods which modify the given list in-place. This means that the given list is lost and replaced by the modified list. We start by introducing methods to rearrange the elements in the list. The most basic one is the `reverse` method which simply reverts the order of the elements in a list. The sort the elements of a list in increasing order we use the `sort` method. A decreasing order can be achieved by adding the option `reverse = True`,

```
sage: L = [1, 8, 5, 2, 9]
sage: L.reverse(); L
[9, 2, 5, 8, 1]
sage: L.sort(); L
[1, 2, 5, 8, 9]
sage: L.sort(reverse = True); L
[9, 8, 5, 2, 1]
```

The `sort` command also accepts a ***key*** function as an additional argument. This can be used to modify the sorting criteria of `sort`. For example, we can sort a list of integer lists in the increasing order of their second element.

```
sage: def TakeSecond(x):
....:     return x[1]
sage: L = [[2, 2], [3, 4], [4, 1], [1, 3]]
sage: L.sort(key = TakeSecond); L
[[4, 1], [2, 2], [1, 3], [3, 4]]
```

There are various other methods available to modify the elements of a list. In the code

List Operations	
Reversing the Order of Elements	`L.reverse()`
Sorting a List	`sort`
Adding an Element at the End	`L.append(x)`
Adding a List at the End	`L.extend(L1)`
Inserting an Element x at Position i	`L.insert(i,x)`
Counting Elements equal to x	`L.count(x)`
Removing i-th Element	`L.pop(i)`
Index of first Element equal to x	`L.index(x)`
Removing first Element equal to x	`L.remove(x)`
Removing a Sub-List	`del L[p:q]`

Table 5.1.: List Operations

below we shortly introduce the most basic modifications methods.

```
sage: L = [1, 2, 3, 4]; L1 = [5, 6]
sage: L.append(7); L    # equivalent to L[len(L):] = [7]
[1, 2, 3, 4, 7]
sage: L.extend(L1); L   # equivalent to L[len(L):] = L1
[1, 2, 3, 4, 7, 5, 6]
sage: L.insert(1, 7); L # equivalent to L[1:1] = [7]
[1, 7, 2, 3, 4, 7, 5, 6]
sage: del L[2:3]; L     # equivalent to L[2:3] = []
[1, 7, 3, 4, 7, 5, 6]
```

If we want to know how often an element x appears in a list `L` we can use the command `L.count(x)`. Another useful command is `L.pop(i)` which returns the element of index i, respectively the last element if no index is declared, i.e. `L.pop()`. This element is also removed from the list. Regarding a fix element x, `L.index(x)` returns the index of the first element equal to x and `L.remove(x)` removes the first element equal to x. Both of these methods do not deal with further elements equal to x and raise an error if x is not contained in the list.

```
sage: L.count(7), L
(2, [1, 7, 3, 4, 7, 5, 6])
sage: L.pop(3), L
(4, [1, 7, 3, 7, 5, 6])
sage: L.pop(), L
(6, [1, 7, 3, 7, 5])
sage: L.index(7), L
(1, [1, 7, 3, 7, 5])
sage: L.remove(7); L
[1, 3, 7, 5]
```

5.1.3 Example: The Sieve of Eratosthenes

The sieve of Eratosthenes is an ancient algorithm to find all prime numbers up to a given upper limit N. Roughly speaking this algorithms works as follows:

- Start with a list of integers from 2 to N.
- Starting with 2 as the first prime number, remove all multiples of 2 from the list.
- Go to the next number q and remove all multiples of q from the list.
- Repeat the previous step until the end of the list is reached.
- The remaining numbers are all prime numbers smaller than or equal to N.

We want to write a Sage program that replicates this algorithm. First, we write a small sub-program which removes all multiples of a given number p from a list.

```
sage: def rem(p, L):
....:     for k in srange(2, ceil(L[-1]/p), include_endpoint =
   True):
....:         if k*p in L:
....:             L.remove(k*p)
....:     return L
```

Since the highest multiple of a number p in an integer list `[2, ..., n]` is bounded from above by $\left\lceil \frac{n}{p} \right\rceil$ we use this term as an upper bound to avoid unnecessary repetitions.

In the next step we use our `rem` function to define the sieve of Eratosthenes in Sage using a `while` loop. The function `eratos` takes a number n and returns a list with all prim numbers up to n

```
sage: def eratos(n):
....:     L = [2..n]
....:     i = 0
....:     while i < len(L) or i <= sqrt(n):
....:         rem(L[i], L)
....:         i += 1
....:     return L
```

In the above code the length of the list `L` changes with every iteration of the `while` loop. As an additional upper bound we added `sqrt(n)`. The reason is the small elementary fact that a number n is prime if and only if all numbers smaller or equal to $\sqrt{n}$ do not divide n. Now, we can use our function `eratos` to obtain a list of all prime numbers up to a given number n.

```
sage: eratos(10)
[2, 3, 5, 7]
sage: eratos(40)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]
```

5.1.4 Example: The UlamSsequence

The Ulam sequence is a recursively defined sequence introduced by Stanislaw Ulam in 1964. The classical Ulam sequence starts with $u_0 = 1$ and $u_1 = 2$ but any distinct pair of integer numbers is also possible. For $n \geq 2$, u_n is defined to be the smallest integer larger than u_{n-1} that can be represented in exactly one way as the sum of two earlier terms, i.e. there exists exactly one pair (k,l), $k \neq l$, such that $u_k + u_l = u_n$ and $k,l < n$. We want to write a function in Sage that takes two starting values u_0, u_1 and an integer $n \geq 2$ and returns a list containing the first n-terms of the Ulam sequence starting with u_0 and u_1.

One possible first step is to write a sub-programm `next_elem` that takes a list `L = [u0,..., un]` of integers and calculates the next term following the Ulam algorithm. The rough idea is to check for every integer starting with `a = L[-1] + 1` whether it can be represented in a unique way as a sum of two distinct elements of `L`. Therefore, we use a `for` loop to go through the list and test if the difference `a - L[k]` is in `L` and not equal to `L[k]`. If that is the case we add one to our counter `pairs`. After the `for` loop has terminated we check whether `pairs` is equal to two. If that is the case, the value `a` is returned, otherwise we repeat the whole procedure with `a + 1`. In our implementation the correct value for the counter `pairs` is two and not one. The reason is the following: `a = L[k] + L[l] = L[l] + L[k]` is in the mathematical sense one way of writing `a` as a sum but the counter `pairs` will have the value 2 since the `for` loop adds one for the pair (k,l) and one for the pair (l,k). Following these considerations the procedure `next_elem` is written as follows.

```
sage: def next_elem(L):
....:     # set starting integer:
....:     a = L[-1] + 1
....:     while True:
....:         # start counter:
....:         pairs = 0
....:         for k in srange(0,len(L)):
....:             if (a - L[k]) in L and L.index(a-L[k]) != k:
....:                 # raise counter if a is sum:
....:                 pairs += 1
....:         if pairs == 2:
....:             # stop if number of pairs is exactly 2:
....:             break
....:         else:
....:             # otherwise repeat with next integer:
....:             a += 1
....:     return a
```

Now it is an easy task to write a procedure returning the list of the first n elements of the Ulam sequence starting with u_0 and u_1. Using a `for` loop we simply add the first n elements with our procedure `next_elem`.

```
sage: def ulam(u0, u1, n):
....:     L = [u0, u1]
....:     for k in srange(2, n+1):
```

```
....:         L.append(next_elem(L))
....:     return L
```

Now we can determine the Ulam sequence with any pair of starting values (u_0, u_1).

```
sage: ulam(1, 2, 10)  # the classical sequence
[1, 2, 3, 4, 6, 8, 11, 13, 16, 18, 26]
sage: ulam(5, 11, 15) # other starting values
[5, 11, 16, 21, 26, 27, 31, 36, 38, 41, 46, 48, 49, 51, 56, 58]
```

5.2 Character Strings

Character strings are mainly used for text. They are delimited by single or double quotes, '...', or "...".

```
sage: S = 'I␣am␣a␣character␣string.'
```

The comparison of two character strings is performed according to the internal encoding of each character. Similar to lists, the length of a string can be returned with `len` and concatenation is performed with the addition symbol `+` and the multiplication symbol `*`. Accessing sub-strings of a given string `S` uses the same syntax as for lists, i.e. `S[n]`, `S[p:q]` and so on. The result is in that case again a character string. However, the `Python` language forbids the replacement of an initial string by such an assignment because character strings are ***immutable***, see Section 5.5.2.

To convert a given object `obj` into a character string we type `str(obj)`. The command `split` cuts a given string at spaces and returns the single parts in a list. Moreover, we can split a string with at any character by adding it as an additional argument to `split`.

```
sage: S = 'Let␣us␣split!'; S.split()
['Let', 'us', 'split!']
sage: S.split('s')
['Let␣u', '␣', 'plit!']
```

5.3 Finite Sets

In contrast to lists, the set type only keeps track of whether an element is present or absent without considering its position or its order of repetition, e.g. $\{a,b,a\} = \{b,b,a\}$. Finite Sets are constructed in `Sage` by applying the `Set` function to a list of its elements. Similar to mathematics, the output is with curly brackets.

```
sage: E = Set([1, 1, 2, 3, 5]); E
{1, 2, 3, 5}
```

Similar to lists, the operation `in` checks whether a set contains a specific element. Moreover, `Sage` provides all usual set constructions, like the union of sets by `+` or `|`, the intersection by `&`, the set difference by `-` and the symmetric difference by `^^`.

```
sage: E = Set([1, 2, 4, 8]); F = Set([1, 3, 5, 7])
sage: 5 in E, 5 in F
```

```
(False, True)
sage: E & F
{1}
sage: E + F
{1, 2, 3, 4, 5, 7, 8}
sage: E - F
{8, 2, 4}
sage: E ^^ F
{2, 3, 4, 5, 7, 8}
```

■ **Example 5.3** Using the union operator we can define a function `include` that checks whether E is a subset of F.

```
sage: def included (E, F): return E + F == F
```

■

Analogously to lists, `len` returns cardinalityt. Also the operations `map`, `filter` and the comprehension form apply to sets as well, but the type of the result does not change. Accessing an element in a set `E` is done as usual via `E[k]`. In the following example we construct the list of the elements of a set in two different ways using the comprehension form.

```
sage: E = Set([4, 7, 1, 5, 8])
sage: [E[k] for k in [0..len(E)-1]]
[1, 4, 5, 7, 8]
sage: [t for t in E]
[1, 4, 5, 7, 8]
```

5.3.1 Example: The Inclusion-Exclusion Principle

Let $(A_n)_{0\leq n\leq N}$ be a family of finite sets. The inclusion-exclusion principle is a counting technique to calculate the cardinality of $\bigcup_{n=0}^{N} A_n$ generalizing the familiar identity

$$|A\cup B| = |A| + |B| - |A\cap B|.$$

for finite sets A,B. The pattern of the inclusion-exclusion principle can be seen more clearly in the case of three finite sets A,B,C. There we have

$$|A\cup B\cup C| = |A| + |B| + |C| - |A\cap B| - |B\cap C| - |A\cap C| + |A\cap B\cap C|.$$

Generalizing this to a family $(A_n)_{0\leq n\leq N}$ the cardinality of $\bigcup_{n=0}^{N} A_n$ is calculated via

$$\left|\bigcup_{n=0}^{N} A_n\right| = \sum_{\emptyset\neq I\subset\{0,\ldots,N\}} (-1)^{|I|+1} \left|\bigcap_{i\in I} A_i\right|. \tag{5.1}$$

Here, the index I goes over all non-empty subsets of $\{0,\ldots,N\}$, i.e. I goes over all non-empty elements of the power set $P(\{0,\ldots N\})$. Recall, that a power set $P(A)$ of a set A is the set of all subsets of A.

Our goal is to write a Sage program `inex` that takes a set $S = \{A_1, \ldots, A_N\}$ and returns the cardinality of $\left|\bigcup_{n=0}^{N} A_n\right|$ using the inclusion-exclusion principle. Taking a look at (5.1) we see that all summands have the structure $\left|\bigcap_{i\in I} A_i\right|$ for a given index set I. Thus, we write a small sub-program `indexsection` that takes a set of sets S and an index set I and returns the set $\bigcap_{i\in I} A_i$.

```
sage: def indexsection(S, I):
....:     r'''
....:     Return the intersection over a given index set
....:     '''
....:     X = S[I[0]]
....:     for k in srange(1, len(I)):
....:         X = X & S[I[k]]
....:     return X
```

Now we use this programm to write our procedure `inex`. Since, I goes over all non-empty elements of the power set $P(\{0,\ldots N\})$, we use the function `powerset` in Sage. However, for our purpose we want to have a list of all elements of the powerset, i.e.

```
sage: list(powerset([0..2]))
[[], [0], [1], [0, 1], [2], [0, 2], [1, 2], [0, 1, 2]]
```

With this command in our hand the procedure `inex` is written as follows:

```
sage: def inex(S):
....:     # start the sum with 0:
....:     res = 0
....:     # create the indexset:
....:     L = list(powerset([0..len(S)-1]))
....:     for k in srange(1, len(L)):
....:         if len(L[k]) % 2 == 0: #substract if even
....:             res = res - len(indexsection(S, L[k]))
....:         else: #add otherwise
....:             res = res + len(indexsection(S, L[k]))
....:     return res
```

Now, we can use the inclusion-exclusion principle in Sage.

```
sage: A1 = Set([1, 6,45, 2, 6,4,4])
sage: A2 = Set([4,23,6,23,6,1])
sage: A3 = Set([4,341,7,2,4,77,7,2])
sage: A4 = Set([4,23,7,33,76,2])
sage: S = Set([A1, A2, A3, A4])
sage: inex(S) == len(A1 + A2 + A3 + A4)
True
```

5.4 Dictionaries

Like a dictionary in the usual sense, the Python type dictionary associates a value to a given key. The syntax is similar to lists, using assignments from the empty dictionary `dict()` or its alias `{}`.

```
sage: D = {}
sage: D['one'] = 1; D['two'] = 2; D['three'] = 3
sage: D
{'one': 1, 'two': 2, 'three': 3}
sage: D['two'] + D['three']
5
```

In the above example we see how to add an entry (key, value) to a dictionary and how to access the value associated to a given key. Similar to lists, the operator `in` checks whether a given key `x` is contained in the dictionary and the commands `del D[x]` and `D.pop(x)` erase the entry of key `x` in the dictionary.

```
sage: a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
sage: 'fruit' in a_dict
True
sage: 'plant' in a_dict
False
```

Furthermore, we can iterate over the keys of dictionaries or over their values.

```
sage: a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
sage: for key in a_dict:
....:     print(key)
color
fruit
pet
sage: a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
sage: for value in a_dict.values():
....:     print(value)
blue
apple
dog
```

Similarly, we can iterate over both, keys and values, simultaneously.

```
sage: a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
sage: for item in a_dict.items():
....:     print(item)
('color', 'blue')
('fruit', 'apple')
('pet', 'dog')
```

One possible application for dictionaries in mathematics is the representation of a

function on a finite set, e.g.

$$A = \{a_0, a_1, a_2\},\ f : A \to B,\ f(a_i) = b_i.$$

The code below implements the above function and returns the input set A and the output set $\mathrm{Im} f = f(A)$ via the aforementioned methods `keys` and `values`:

```
sage: D = {'a0':'b0', 'a1':'b1', 'a2':'b2'}
sage: A = Set(D.keys()); B = Set(D.values())
sage: A, B
({'a0', 'a1', 'a2'}, {'b1', 'b0', 'b2'})
```

Dictionaries may also be constructed from list or pairs `[key, value]` via

```
dict(['a0', 'b0'], ['a1', 'b1'], ...)
```

■ **Example 5.4** The following test on the number of distinct values can be used to determine whether a function f represented by D is injective.

```
sage: def injective(D):
....:     return len(D) == len(Set(D.values()))
```

■

5.5 More on Data Structures

5.5.1 Shared or Duplicated Data Structure

Assigning a list to a variable only ***shares*** the data structure and does not duplicate the data structure. In the following example the lists `L1` and `L2` remain identical, i.e. they are two aliases of the same object. In particular, modifying `L1` also modifies `L2` and vice versa.

```
sage: L1 = [1, 2, 3]; L1 = L2
sage: L1[1] = 42; L2
sage: [42, 2, 3]
```

In contrast, `map`, `filter` and `flatten` ***duplicate*** the data structures. The same holds for list constructions by `L[p:q]`, comprehension forms and the concatenations `+` and `*`. Checking for shared data structure can be done in `Sage` with the `is` operator.

```
sage: L1 = [1, 2, 3]; L2 = L1; L3 = L1[:]
sage: [L1 is L2, L1 is L3, L1 == L3]
[True, False, True]
```

5.5.2 Mutable and Immutable Data Structure

As seen in Section 5.1 a list can be changed in-place with methods like `reverse` or `sort`. This is only possible because lists are ***mutable*** data structures. In contrast to that sets are ***immutable***. This means that sets cannot be modified in-place after their construction.

```
sage: L = [0..5]; L
[0, 1, 2, 3, 4, 5]
sage: L[0] = 10
[10, 1, 2, 3, 4, 5]
sage: S = Set([0..5]); S
{0, 1, 2, 3, 4, 5}
sage: S[0] = 10
Traceback (most recent call last):
...
TypeError: 'Set_object_enumerated_with_category' object does not
   support item assignment
```

The immutable counterpart to lists are ***sequences*** or ***tuples*** and are denoted by parentheses `(...)` instead of square brackets. A tuple with only one element is defined by adding a comma after the element, to distinguish it from mathematical parentheses.

```
sage: S0 = (); S1 = (1, ); S2 = (1, 2)
sage: S0, S1, S2
((), (1,), (1, 2))
sage: [1 in S1, (1) is S1]
[True, False]
```

The available operations on tuples are essentially the sames as those on lists, e.g. `map`, `filter` and the comprehension form. However, the type of the result thus not change, e.g. the comprehension form transforms a tuple into a list.

```
sage: S1 = (1, 4, 9, 16); [k for k in S1]
[1, 4, 9, 16]
```

III Graphics

6 2D Graphics 77

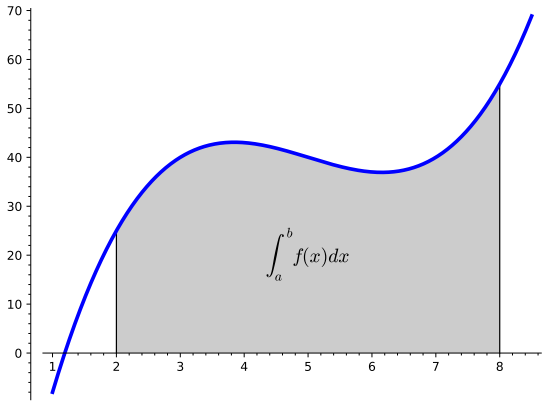
6.1 Drawing Curves
6.2 Vector Fields
6.3 Complex Functions
6.4 Density and Contour Plots
6.5 Data Plot
6.6 More Graphic Primitives

7 3D Graphics 107

7.1 Plotting Functions
7.2 Vector Fields
7.3 More Graphic Primitives

6. 2D Graphics

Drawing the plot of a function or visualizing a series of data makes it often easier to grasp a mathematical phenomena, e.g. the integration of a function, helps us make conjectures, e.g. the integration of a function.



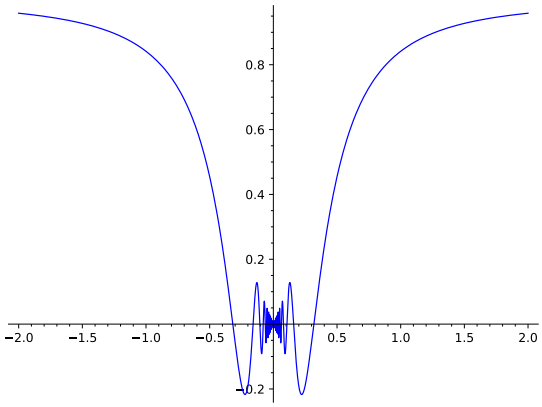
Sage provides many possibilities to draw a plane curve, e.g. as the graph of a function, from a parametric system, using polar coordinates or by an implicit equation. In this chapter we describe these cases and give various examples of data visualization.

6.1 Drawing Curves

6.1.1 Graphical Representation of a Function

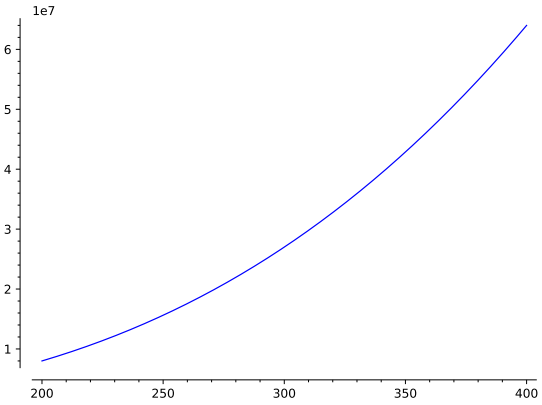
In Section 2.3.3 we used the `plot` command to draw the graph of a symbolic or Python function $f(x)$ on an interval $[a,b]$. There the basic syntax is given by `plot(f(x), x, a, b)`. However, there are various options available that allow us to modify the plot in many different ways. These modifications will be discussed in this section.

```
sage: plot(x *sin(1/x), x, -2, 2)
```



First, we observe that Sage makes it easy to tell whether a graph is on both sides of both axes, as the axes only cross if the origin is actually part of the viewing area. If one of the axes labels are very large, the scientific notation, i.e. the *e*-notation, is used.

```
sage: plot(x^3, x, 200, 400)
```

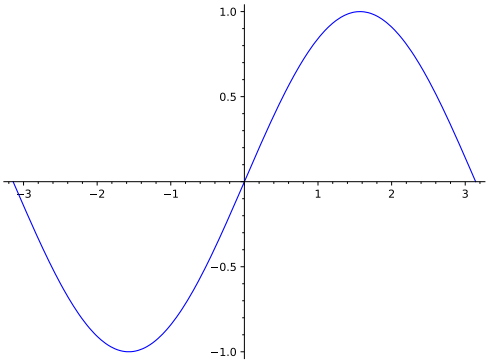


We start with introducing a few basic options which do not need many explanations.

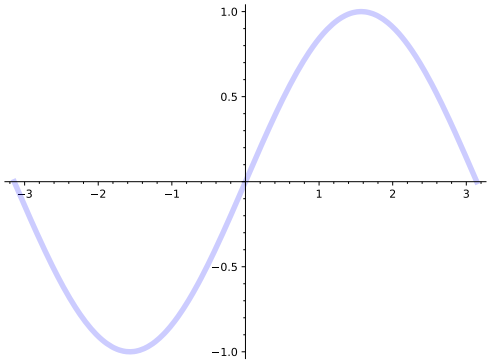
- `plot_points` are the minimal number of computed points. The default value here is 200.
- `xmin` and `xmax` are the interval bounds over which the function is displayed.
- `alpha` is the option for the line transparency. Admissible values are between 0 for invisible and 1 for completely non-transparent. The default value here is 1.
- `thickness` controls the line thickness. The default value here is 1. To thicken the line we insert a larger number.

We consider a basic graph of the sinus function and modify it with the above options.

```
sage:  plot(sin(x), x, -pi, pi) #standard plot
sage:  plot(sin(x), x, -pi, pi,  alpha = 0.2, thickness = 5) #
   modified plot
```



Standard Plot of $\sin(x)$

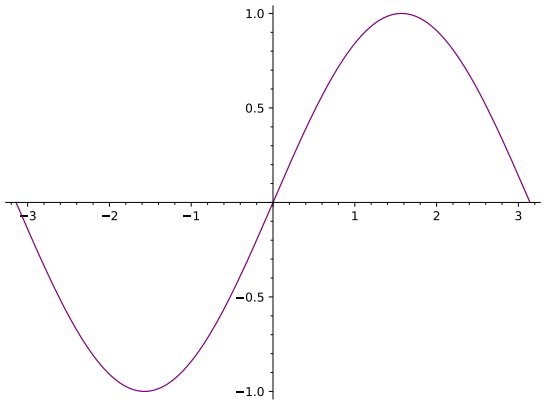


Modified Plot of $\sin(x)$

Moreover, we can change the color of the graph with the `color` option. There we have various possibilities to define the color.

- as a character string, e.g. `'blue'` (default value), `'red'`, ...,
- as a RGB-triple `(r, g, b)`, where the three values are between 0 and 1,
- as an HTML-color, such as `#aaff0b`
- as HSV-color, such as `hue(0.5)`, where the number should be between 0 and 1.

```
sage: plot(sin(x), x, -pi, pi, color = 'purple')
```

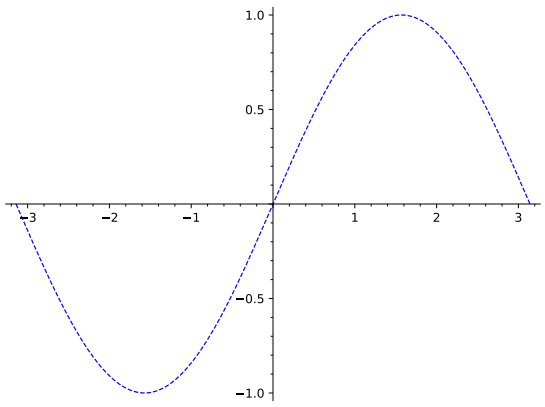


The `linestyle` option allows us to change the line style of the graph to one of the following possibilities.

- `"-"` or `"solid"`,
- `"- -"` or `"dashed"`,

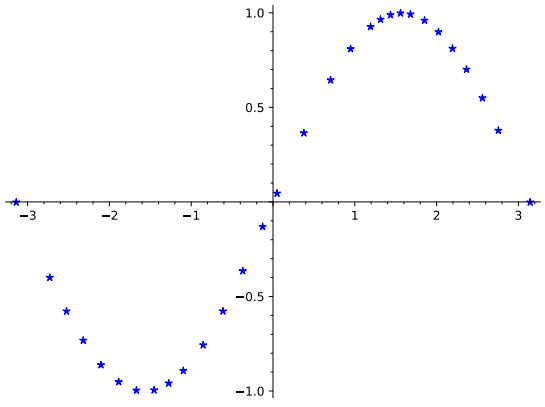
- ":" or "dotted",
- "-." or "dash dot",
- " " or "None".

```
sage: plot(sin(x), x, -pi, pi, linestyle = "--")
```



If we have an empty linestyle and specify a marker, like a circle "o", a point ".", or a square "s", we can see the points that have been actually computed. Also TeX symbol can be used as marker.

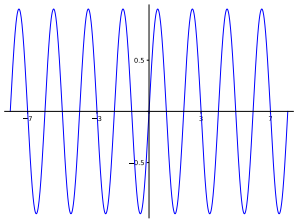
```
sage: plot(sin(x), x, -pi, pi, plot_points = 10, linestyle = "␣",
   marker = r'$\star$')
```



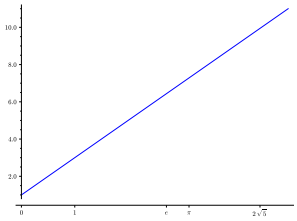
Besides the appearance of the graph we can also change the scaling of the axes from its default value `linear` to a logarithmic scale using the `scale` option. There, the possible values are `'loglog'` if both axes should have logarithmic scale and `'semilogx'` and `'semilogy'` if only the x-, respectively the y-axis should have logarithmic scale. The base of the logarithmic scale is by default 10 but can be changed to any value greater than 1 using the `base` option. We can assign different bases to the separate axes via `base = (basex, basey)`. If the linear scale is chosen, the `base` option is ignored. A more advanced scaling of the axes is done with the tick formatting. The command

`ticks = [x, y]` takes a list `x` and `y` containing all values that should be marked on the x-, respectively y-axis. Alternatively, we can insert a positive real number defining the new linear scaling. By default the labels are displayed the usual Sage format but can be changed to a LaTeXformat via `tick_formatter = "latex"`. For trigonmetric functions it is often useful to label the axis with multiplicites of π. To do so, we use the option `tick_formatter = pi`.

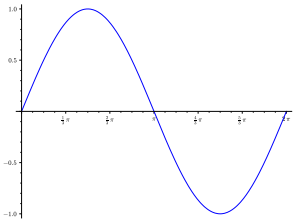
```
sage: h1 = plot(sin(pi*x), (x, -8, 8), ticks
   =[[-7,-3,0,3,7],[-1/2,0,1/2]])
sage: h2 =  plot(2*x+1,(x,0,5),ticks=[[0,1,e,pi,sqrt(20)],2],
   tick_formatter="latex")
sage: h3 = plot(sin(x),(x,0,2*pi),ticks=pi/3,tick_formatter=pi)
```



Plot `h1`



Plot `h2`



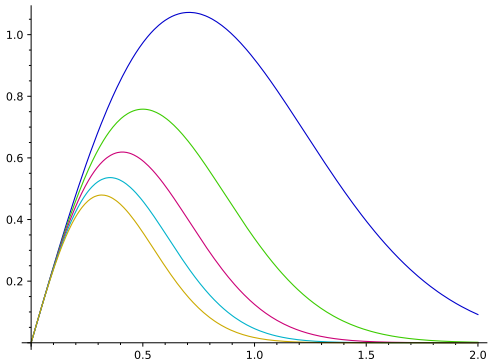
Plot `h3`

Just as any other object discussed so far, also graphical objects can be assigned to a `Python` variable, say `g`. Afterwards we can use the `show` command to display it. There, we can add additional options like bounds for the y-axis, e.g. `g.show(ymin = -1, ymax = 3)`, the aspect ratio, e.g. `g.show(aspect_ratio = 1)` to have equal scales for x and y. Moreover, the options `axes_label`,`title` and `scale` are also valid. To export a graphical object, we use the `save` command, e.g. `g.save('filename.png')`. There we can choose between several formats, e.g. `.pdf`, `.png`, `.ps`, `.eps`, `.svg` and `.sobj`.

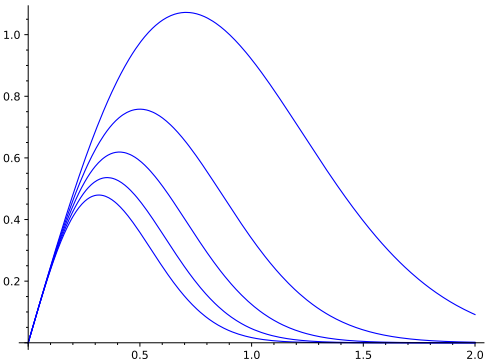
R *To include a graphic in a LaTeXdocument using* `includegraphics` *we recommend to use the* `eps` *format if the file is compiled with* `latex` *and the* `pdf` *format if the file is compiled with* `pdflatex`.

To draw the graphs of various functions into one graphical object we hand over a list of functions as an argument in `plot` or by simply add the different plots.

```
sage: g1 = plot([x*exp(-n*x^2)/(0.4) for n in [1..5]],x,  0, 2)
sage: g2 = plot([]) #empty plot
sage: for n in [1..5]:
....:     g2 += plot(x*exp(-n*x^2)/(0.4), x, 0, 2)
```



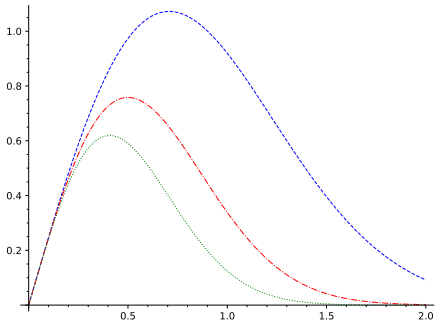
Plot g1



Plot g2

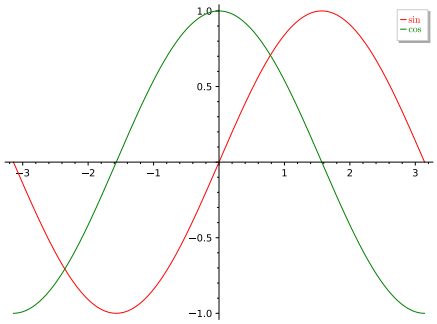
Although we have plotted the same functions in both graphics the coloring is different. The reason is the following. If a list of functions is inserted into `plot`, as done in `g1`, the color changes by default from curve to curve. In the plot `g2`, however, the single curves are first drawn in the default setting, i.e. the curves are blue, and added afterwards to the plot `g2`. To modify the appearance of a plot, where a list of functions has been used as an input, as in `g1`, we simply use lists as arguments for the various options.

```
sage: g3 =  plot([x*exp(-n*x^2)/(0.4) for n in [1..3]],x,  0, 2,
    color = ['blue', 'red', 'green'], linestyle = ["--", "-.", ":"
    ])
```



If more than one curve is contained in one plot it is recommended to use a label explaining the different curves. Such a label can be added with the option `legend_label`. There TeX code can be used. Moreover the color of each label can be changed with `legend_color` using the same ways as described for the `color` option. If multiple functions are in one plot, their labels are combined in one legend.

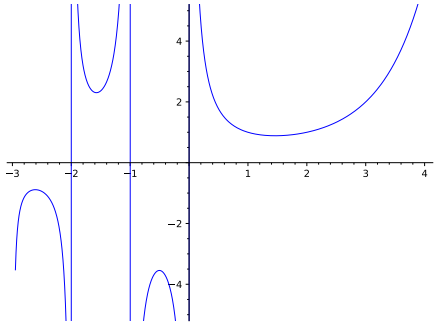
```
sage: p1 = plot(sin(x), x, -pi, pi, color = 'red', legend_label =
    r'$\sin$', legend_color = 'red')
sage: p2 = plot(cos(x), x, -pi, pi, color = 'green', legend_label
    = r'$\cos$', legend_color = 'green')
sage: p = p1 + p2
```



If a list of functions is used in `plot` and `legend_label` is set to `'automatic'` Sage creates a label for each function according to their internal representation. But as soon as `legend_label` is any string different from `'automatic'` it will be repeated for all members. To modify the single labels, we have to use a list with the desired labels, i.e. `legend_label = ['name1', 'name2',... ]` using the same syntax as for the other `plot` options.

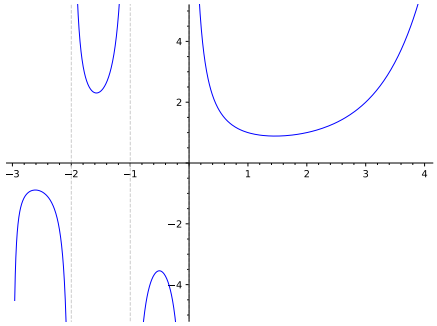
So far, we have only drawn the graph of functions which have been well-defined on the whole interval. What happens if we try to plot a function with poles like the Gamma function Γ.

```
sage: plot(gamma(x), -3, 4).show(ymin = -5, ymax = 5)
```



Here it is very important to add the bounds on the y-axis. Otherwise we would not be able to see the original function because of the high values of the function values near the poles. Since `plot` only connects computed function values vertical lines near the poles appear. These vertical lines can be removed with `detect_poles = True`. Setting `detect_poles = 'show'` adds dashed vertical lines to highlight the position of the poles.

```
sage: plot(gamma(x), -3, 4, detect_poles = 'show').show(ymin = -5,
    ymax = 5)
```

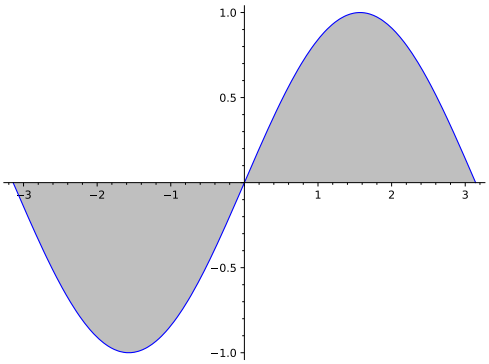


Last but not least, we describe the basic filling options available in `plot`. The option `fill` takes the following values:

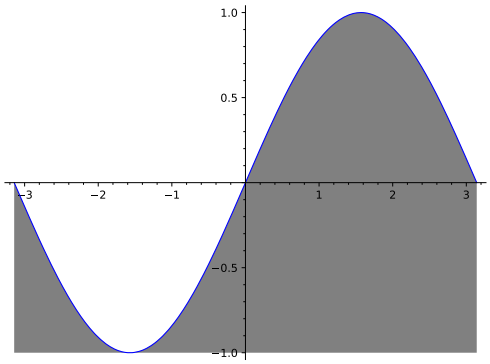
- `'axis'`: fills the space between the graph and the x-axis.
- `'min'`: fills the space under the graph.
- `'max'`: fills the space over the graph.
- some one-valued function: fills the space between the graph of the given function and the plotted graph.

This filling comes with a default color and transparency which can be further modified with `fillcolor` and `fillalpha` respectively. The usage of these options is the same as for `alpha` and `colors` respectively.

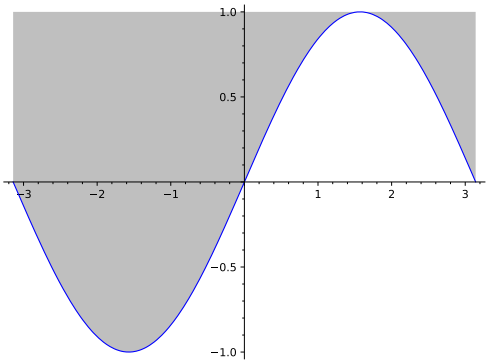
```
sage: p1 = plot(sin(x), -pi, pi, fill='axis')
sage: p2 = plot(sin(x), -pi, pi, fill='min', fillalpha=1)
sage: p3 = plot(sin(x), -pi, pi, fill='max')
sage: p4 = plot(sin(x), -pi, pi, fill=(1-x)/3, fillcolor='blue',
   fillalpha=.2)
```



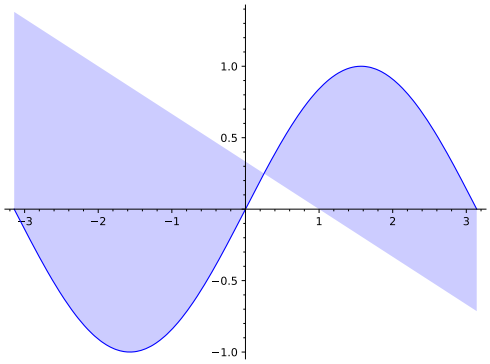
Plot p1



Plot p2



Plot p3



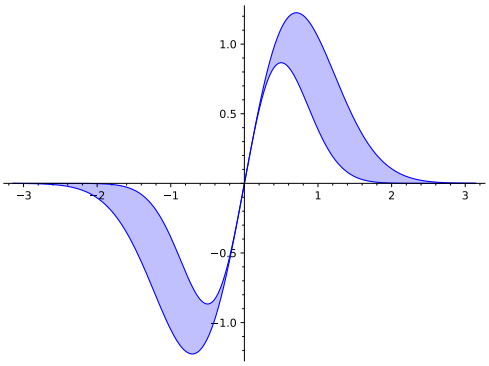
Plot p4

If a list of functions is used in `plot` we can, as usual, either set a global option or hand over a list with the desired filling options for each single function. In addition, the dictionary key-value syntax `{i: [j] }` creates a filling between the i-th the j-th function. Here, the square brackets around `j` are important since the syntax `{i : j }` fills the space between the i-th function and the function $y = j$. In particular, we can fill the space between the i-th function and an arbitrary function $f(x)$ with `{i : f(x) }`.

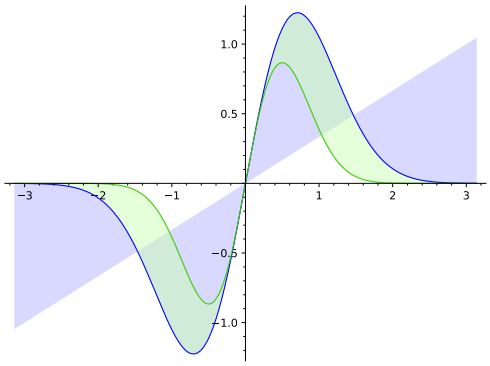
```
sage: (f1, f2) = x*exp(-1*x^2)/.35, x*exp(-2*x^2)/.35
sage: p1 = plot([f1, f2], -pi, pi, fill={1: [0]}, fillcolor='blue'
    , fillalpha=.25, color='blue')
sage: p2 = plot([f1, f2], -pi, pi, fill={0: x/3, 1:[0]}, color=['
    blue'])
sage: p3 = plot([f1, f2], -pi, pi, fill=[0, [0]], fillcolor=['
    orange','red'], fillalpha=1, color={1: 'blue'})
sage: p4 = plot([f1, f2], (x,-pi, pi), fill=[x/3, 0], fillcolor=['
    grey'], color=['red', 'blue'])
```

The Plot	
Plotting a Function x	`plot(f(x), x, xmin, xmax)`
Showing a Plot g	`g.show()`
Saving a Plot g	`g.save(filename.png))`
Plot Options	
Number of Minimal Plot Points	`plot_points =)`
Line Transparency	`alpha)`
Line Thickness	`thickness`
Line Color	`color`
Line Style	`linestyle`
Scaling of the Axes	`scale`
Adding a Legend	`legend_label`
Color of Legend	`legend_color`
Filling	`fill`
Transparency, Color of the Filling	`fillalpha`, `fillcolor`

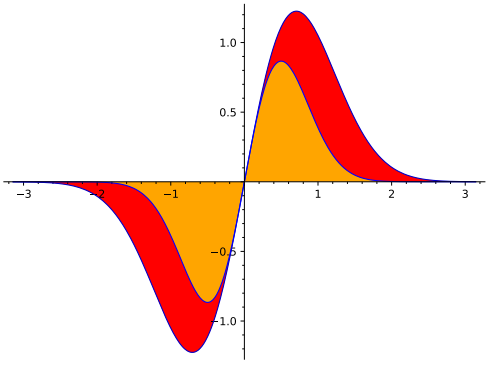
Table 6.1.: Plot Commands



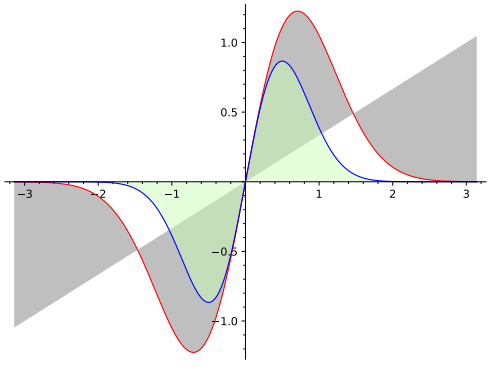
Plot p1



Plot p2



Plot p3

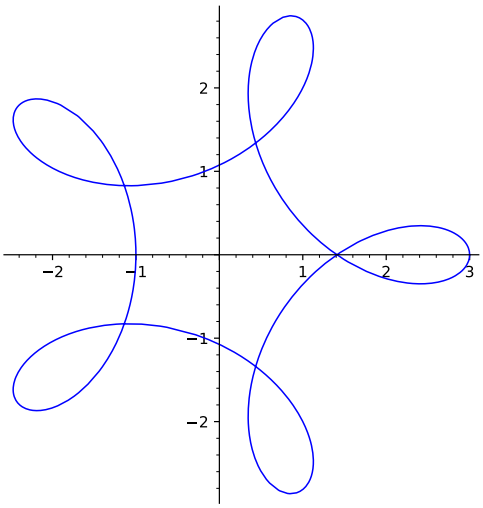


Plot p4

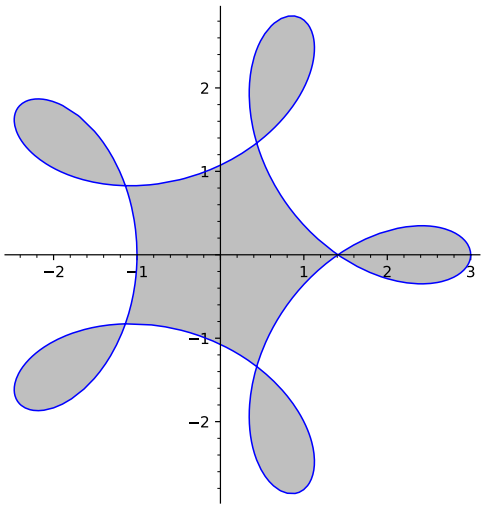
6.1.2 Parametric Plot

Parametric curves $(x = f(t), y = g(t))$ can be drawn with the command`parametric_plot`. The command `parametric_plot` supports the same options as `plot` with one difference. The option `fill` only takes the boolean values `True` or `False`. Thus, there is only fill or no fill. A further difference is that we cannot pass a list of functions.

```
sage: g1 = parametric_plot([cos(x) + 2 * cos(x/4), sin(x) - 2 *
   sin(x/4)], (x, 0, 8*pi))
sage: g2 = parametric_plot([cos(x) + 2 * cos(x/4), sin(x) - 2 *
   sin(x/4)], (x,0, 8*pi), fill = True)
```



A Hyptrochoid (g1)



A filled Hyptrochoid (g2)

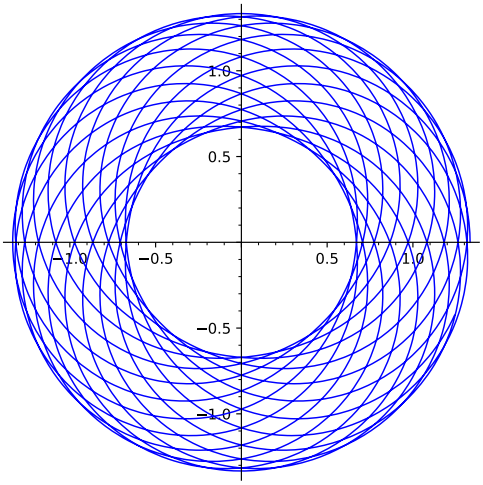
6.1.3 Curves in Polar Coordinates

Besides the usual cartesian coordinates it is also quite common to endow $\mathbb{R}^2$ with polar coordinates (r, ϕ), where the transformation is given by

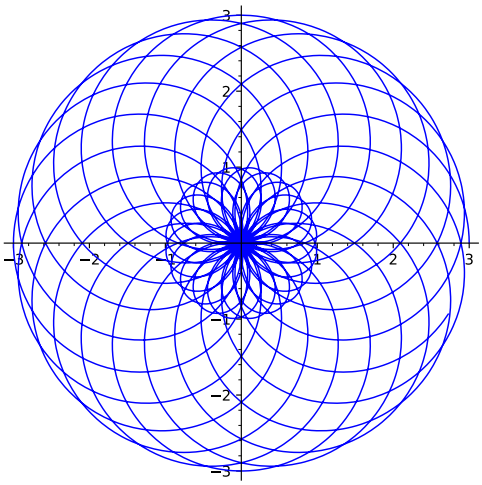
$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} r\sin(\phi) \\ r\cos(\phi) \end{pmatrix}.$$

While `plot` takes a function $f(x)$ with $x \in [a,b]$ and draws the graph $\{(x, f(x)) | x \in [a,b]$ the command `polar_plot` takes a function $\rho(\phi)$ with $\phi \in [a,b]$ and draws the graph $\{(\rho(\phi), \phi) | \phi \in [a,b]\}$ in polar coordinates which translates to $\{(\rho(\phi)\sin(\phi), \rho(\phi)\cos(\phi)) | \phi \in [a,b]\}$ in cartesian coordinates. For example we can plot the rose-curves $\rho(\phi) = (1 + e \cdot \cos(n\phi))$ for $n = \frac{20}{19}$ and $e \in \{\frac{1}{3}, 2\}$

```
sage: t = var('t'); n = 20/19
sage: r1 = polar_plot(1 + 1/3*cos(n*t), (t, 0, n*36*pi),
   plot_points = 5000)
sage: r2 = polar_plot(1 + 2*cos(n*t), (t, 0, n*36*pi), plot_points
    = 5000)
```



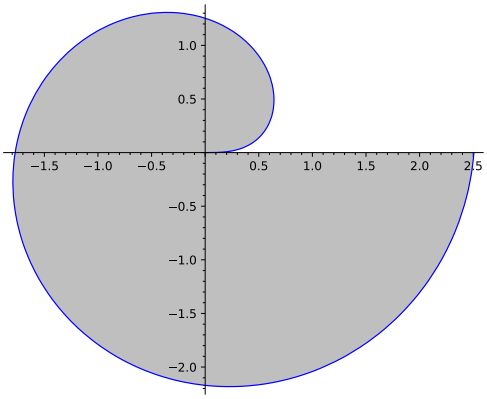
Rose curve with $e = \frac{1}{3}$ (`r1`)



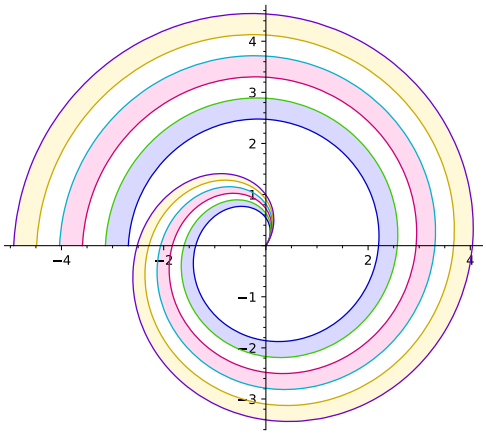
Rose curve with $e = 2$ (`r2`)

Similar to `parametric_plot` all options for `plot` pass over to `polar_plot`, except for the `fill` option. This option only takes the boolean values, `False` and `True`, and symbolic functions in one variable. In addition, we can insert a list of functions in `polar_plot`.

```
sage: p1 = polar_plot(sqrt, 0, 2 * pi, fill=True)
sage: p2 = polar_plot([(1.2+k*0.2)*log(x) for k in range(6)], 1, 3
    * pi, fill={0: [1], 2: [3], 4: [5]})
```



A filled Spiral (`p1`)

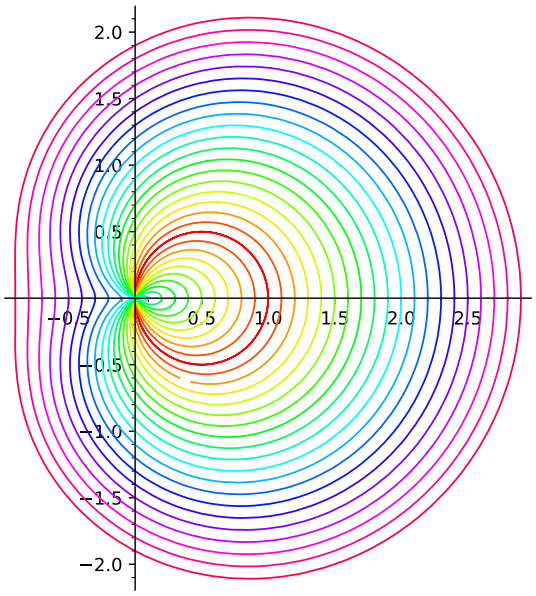


Colorful Spirals (`p2`)

■ **Example 6.1** A conchoid is a curve derived from a fixed point O, another curve α, and a length d. These type of curves were invented by the ancient Greek mathematician Nicomedes. The polar equation of a conchoid is given by $\rho(\theta) = \alpha(\theta) + d$.

One example of conchoids are Pascal conchoids whose polar equation is given by $\rho(\phi) = \cos(\phi) + d$ for $\phi \in [0, 2\pi]$ and $d > 0$. In the plot below we draw a family of Pascal conchoids where the parameter d varies from 0 to 2 in steps of 0.1. To obtain a colorful picture we varied the color using the hue-representation.

```
sage: t = var('t')
sage: L = [d + cos(t) for d in srange(0, 2, 0.1)]
sage: g = polar_plot(L, (t, 0, 2*pi), color = [hue(t/20) for t in
   srange(0,20)])
```

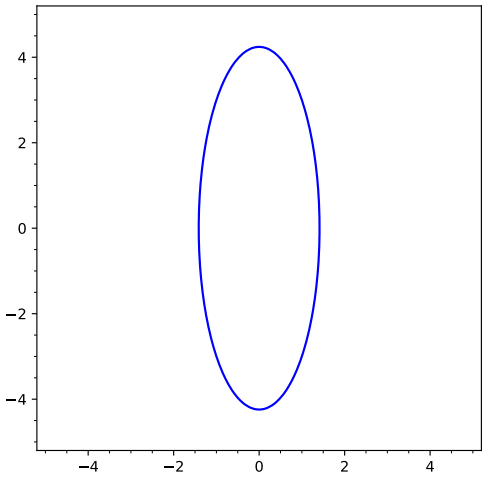


■

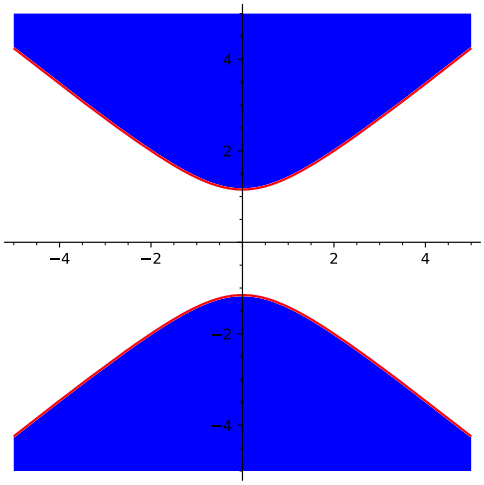
6.1.4 Curve defined by an Implicit Equation

Sometimes, a curve is defined as the solution set of $f(x,y) = 0$ for a function $f : \mathbb{R}^2 \to \mathbb{R}$, i.e. the curve is given by an ***implicit equation***. To plot these kind of course we use the command `implicit_plot (f, (x, xmin, xmax), (y, ymin,ymax))`. Similar with to the previous plot commands, we can use almost all options of `plot` to modify the appearance of the plot. Some differences are the option `fill` which only takes the boolean values `True` and `False` and the option `thickness` which has to be replaced by the option `linewidth`. Furthermore, it is not possible to hand over a list of functions. In the below example we plotted an ellipse given by the implicit equation $\frac{x^2}{4} + \frac{y^2}{25} - 2 = 0$ and a hyperbel given by the implicit equation $x^2 - \frac{3}{2}y^2 + 2 = 0$.

```
sage: x, y = var('x␣y')
sage: elip = implicit_plot(x^2 + y^2/9 - 2, (x, -5, 5), (y, -5, 5)
   )
sage: hyp = implicit_plot(x^2 - 3/2*y^2 + 2, (x, -5, 5), (y, -5,
   5), fill = True, color = 'red')
```



Ellipse `elip`

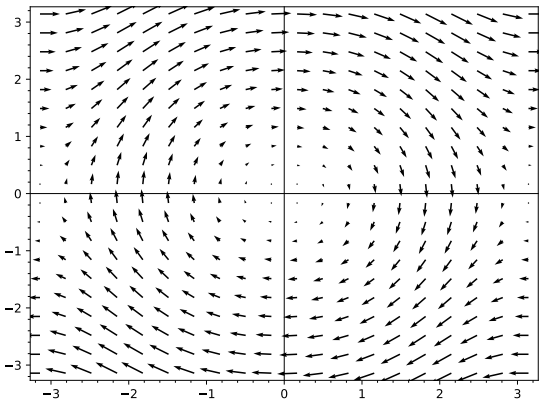


Hyperbel `hyp`

6.2 Vector Fields

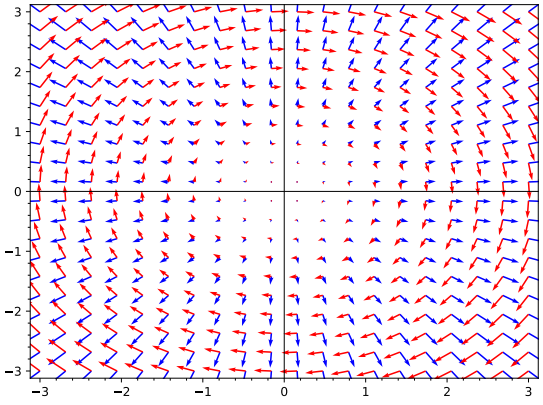
In the previous sections we only considered curves. But these are not the objects that can be drawn with Sage. Another possibility is the visualization of vector fields with `plot_vector_field`. This command takes two function of two variables, e.g. $(f(x,y),g(x,y))$ and plot vector arrows of the function over the specified range.

```
sage: x, y = var('x y')
sage: v = plot_vector_field( (y, (cos(x)-2)*sin(x)), (x, -pi, pi),
    (y, -pi, pi))
```



In `plot_vector_field` the options `color` and `plot_points` are available. Their usage is the same as for `plot`, see Section 6.1.1. As usual, we can add graphics to combine various vector fields in one plot. Below, we create a plot of two orthogonal vector fields.

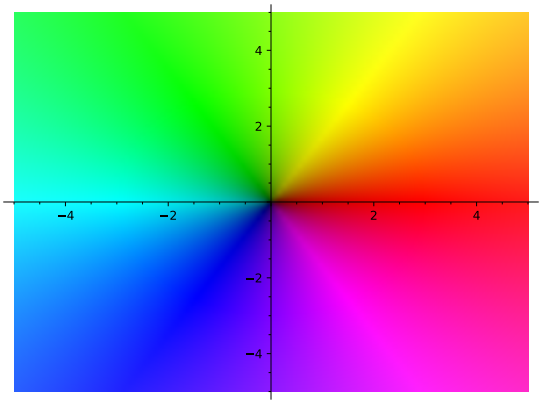
```
sage: a = plot_vector_field((x,y), (x,-3,3), (y,-3,3), color = '
   blue')
sage: a += plot_vector_field((y, -x), (x, -3, 3), (y, -3, 3),
   color = 'red')
```



6.3 Complex Functions

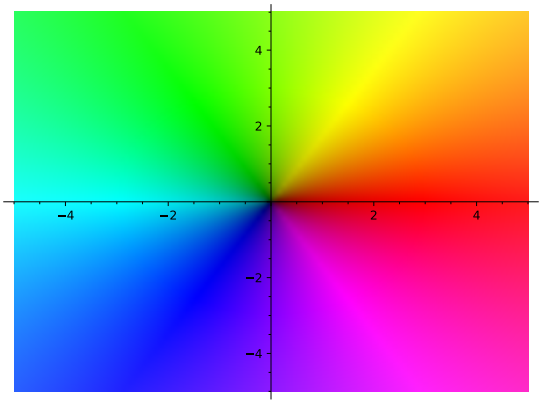
Beside real functions, the study complex functions $f : \mathbb{C} \to \mathbb{C}$ is an interesting topic on its own. In particular the study of holomorphic and meromorphic functions like the Riemann-Zeta function ζ or the Gamma function Γ. Although $\mathbb{C} \equiv \mathbb{R}^2$, Sage can visualize the graph of functions $f : \mathbb{C} \to \mathbb{C}$. There we use the command `complex_plot(f, (xmin, xmax), (ymin, ymax))`, where we interpret $f(z)$ as $f(x,y)$ by identifying $z = x + iy$. Then Sage represents the point $f(x+iy)$ at (x,y) using colors. To obtain an intuition for this color encoding we take a look on the plot of the identity function $f(z) = z$.

```
sage: z = var('z')
sage: p = complex_plot(z, (-5, 5), (-5, 5))
```



The above image shoes that Sage assigns a color to a complex number as follows: Sage takes the complex number in polar coordinates, i.e. $z = |z| \cdot e^{i\arg(z)}$. Then the magnitude $|z|$ is indicated by the brightness, with zero being black and infinity being white, and the argument, $\arg(z)$ is represented by the hue of a color with red being positive real and increasing through orange, yellow, ... as the argument increases. With this color code we can now represent meromorphic functions, like $f(z) = \frac{(z-1+I)(z+2-I/2)^3}{(z+1+3*I)^2*(z-2+2*I)}$.

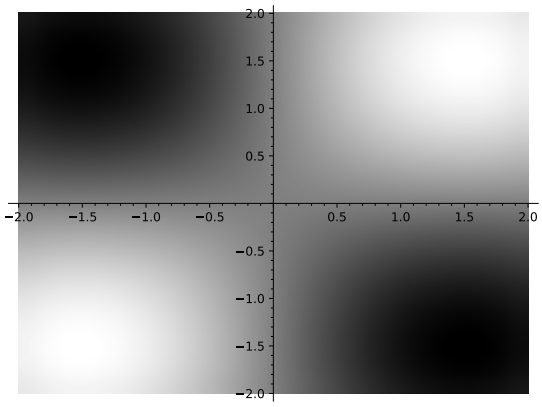
```
sage: g = complex_plot((z-1+I)*(z+2-I/2)^3/((z+1+3*I)^2*(z-2+2*I))
    , (-5, 5), (-5, 5))
```



6.4 Density and Contour Plots

In a similar fashion, as for complex plots, it is possible to plot the graph of a function of two variables, $f : \mathbb{R}^2 \to \mathbb{R}$ in a two dimensional graphic by representing the function value $f(x,y)$ with a color. This can be done with the command `density_plot`. The input is a function of two variables together with their range and the output is, by default, a two dimensional black white graphic, where white denotes high function values and black denotes low function values.

```
sage: x, y = var('x␣y')
sage: d1 = density_plot(sin(x) * sin(y), (x, -2, 2), (y, -2, 2))
```



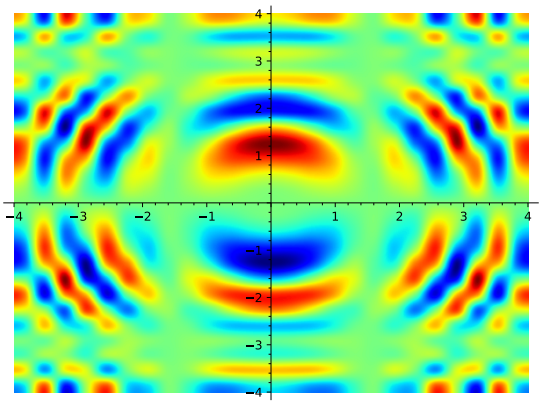
We can change the minimal calculated points, as usual, with `plot_points` (the default value is 250) . To change the color we have to use colormaps. Roughly spoken, a colormap is a map from an interval into a set of colors. Sage provides many different colormaps. A list of the available colomaps within Sage is given by `sorted(colormaps)`.

```
sage: sorted(colormaps)
['Accent', 'Blues', 'BrBG', 'BuGn', 'BuPu', 'CMRmap', 'Dark2', '
   GnBu', 'Greens', 'Greys', 'OrRd', 'Oranges', 'PRGn', 'Paired',
    'Pastel1', 'Pastel2', 'PiYG', 'PuBu', 'PuBuGn', 'PuOr', 'PuRd
   ', 'Purples', 'RdBu', 'RdGy', 'RdPu', 'RdYlBu', 'RdYlGn', '
```

```
    Reds', 'Set1', 'Set2', 'Set3', 'Spectral', 'Wistia', 'YlGn', '
    YlGnBu', 'YlOrBr', 'YlOrRd', 'afmhot', 'autumn', 'binary', '
    bone', 'brg', 'bwr', 'cool', 'coolwarm', 'copper', 'cubehelix'
    , 'flag', 'gist_earth', 'gist_gray', 'gist_heat', 'gist_ncar',
     'gist_rainbow', 'gist_stern', 'gist_yarg', 'gnuplot', '
    gnuplot2', 'gray', 'hot', 'hsv', 'jet', 'nipy_spectral', '
    ocean', 'pink', 'prism', 'rainbow', 'seismic', 'spring', '
    summer', 'tab10', 'tab20',  'tab20b', 'tab20c', 'terrain', '
    winter']
```

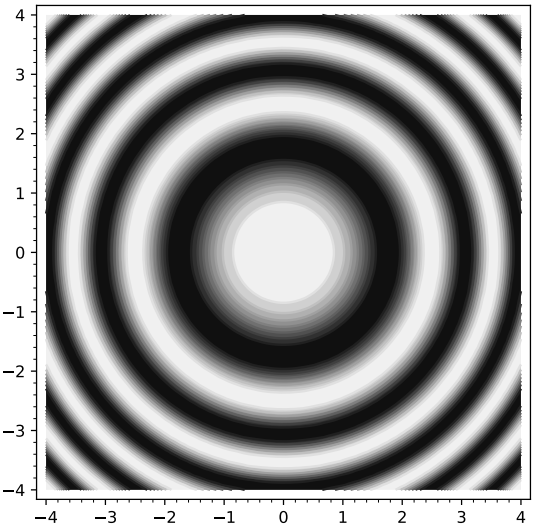
To use such a colormap we use the option `cmp = 'name'`, where instead of `'name'` we insert the name of the used color map as a string. This allows us to create quite colorful density plots.

```
sage: d2 = density_plot(sin(x^2+ y^2) * cos(x) * sin(y), (x, -4,
    4), (y, -4, 4), cmap = 'jet')
```



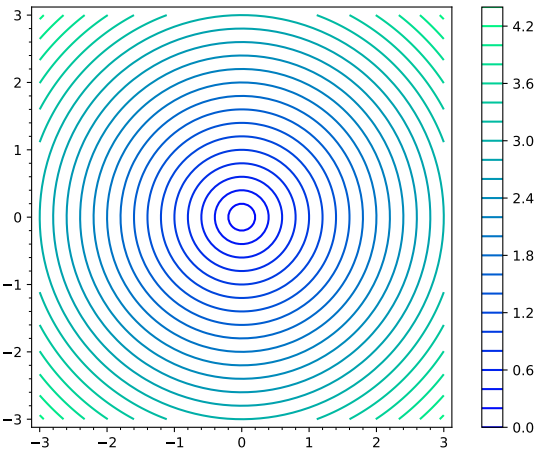
A quite similar but yet different way of plotting the graph of function of two variables in a two dimensional graphic is the command `contour_plot`. Given a function $f(x,y)$ of two variables together with their range, `contour_plot` draws a default choice of contour lines $l_z = \{(x,y) \ : \ f(x,y) = z\}$ and fills the space between them.

```
sage: c1 = contour_plot(cos(x^2 + y^2), (x, -4, 4), (y, -4, 4))
```



As usual, Sage provides various options to modify the appearance of the plot. The usage of the options `plot_points`, `linewidth`, `linestyle` and `labels` are the same as discussed in Section 6.1.1. By default, the option `fill` is set to `True` and fills the area between the contours. Furthermore, we can add a colormap using the option `cmp`. The specific contour lines that are drawn can be customized with the option `contours` in two ways: Either we insert a list of integers, determining the contour levels that are drawn, or a single integer determining only the number of contour lines that are drawn. In the second case the specific contour levels that are drawn are determined automatically. In addition, we can add a colorbar via `colorbar = True`. There are various options available to modify the appearance of the colorbar. We refer to the SageMathe reference manual [3] for more information.

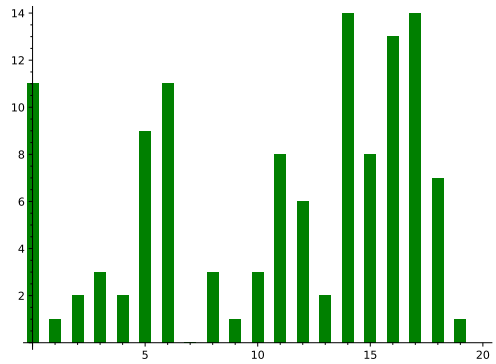
```
sage: c2 = contour_plot(sqrt(x^2 + y^2), (x, -3, 3), (y, -3, 3),
    cmap = 'winter', contours = 20, fill = False, colorbar = True,
     colorbar_spacing = 'proportional')
```



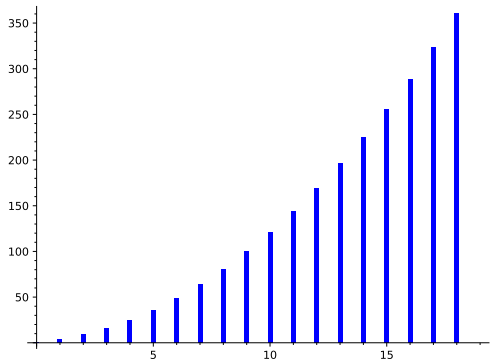
6.5 Data Plot

Besides functions and vector fields it is also useful to visualize data sets using charts. One of the most common ones are bar charts. To draw a bar chart in Sage we insert a list of Python integers, i.e. our data set, into the command `bar_chart`. Then Sage draws vertical bars whose height is given by the list element. It is important to note that we can only use Python integers. This means, we can use integers and fractions but not symbolic expressions symbolizing irrational numbers, e.g. π. The two main options to modify the bar charts are the options `width` and `color` to adjust the width and the color of the drawn bars respectively. The option `width` simply takes a number and the usage of the option `color` is the same as explained for `plot` in Section 6.1.1.

```
sage: b1 = bar_chart([randrange(15) for i in srange(20)], color =
    'green')
sage: b2 = bar_chart([x^2 for x in srange(1,20)], width = 0.2)
```



Bar Chart b1



Bar Chart b2

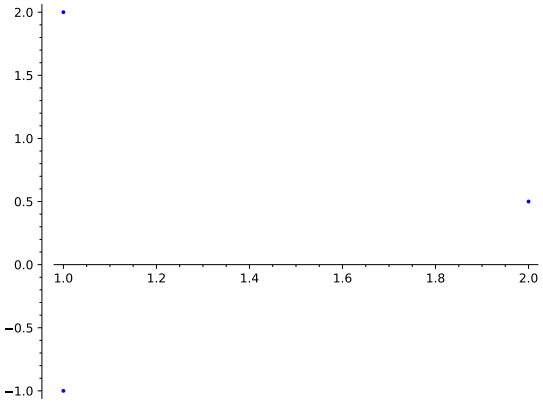
R *In the first bar chart we used the command* `randrange(15)` *to obtain a random integer between* 0 *and* 14. *The syntax of* `randrange` *is the same as for the command* `srange`, *see Section 4.3, i.e.* `randrange(i,j,k)` *returns a random element from the list* `[i, i+k, ..., i+n*k]`, *where* $i+nk < j \leq i+(n+1)k$.

6.6 More Graphic Primitives

Plotting a curve using one of the methods described in the previous sections creates a ***graphics object***, the plot, consisting one ***graphics primitive***, the drawn curve. But there are more kinds of graphic primitives than curves which can be added to a graphics object. In this section we describe the usage of some available graphics object in Sage. A few of these graphics primitves can also be used in 3D-graphics which are discussed in the next chapter. If this is the case, we indicate their three dimensional usage and refer to Section 7.

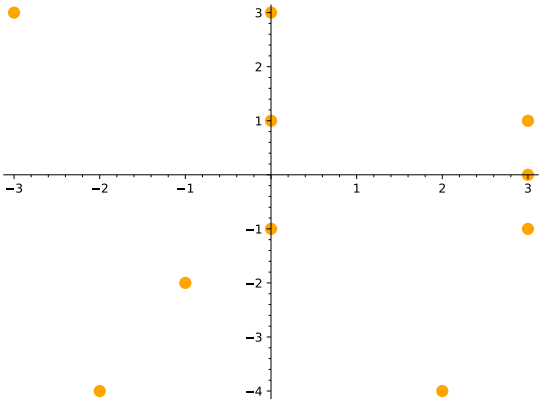
First, we introduce the most simple graphics primitive: a point. The command `point` takes either a single point or a list of points. Points are described using tuples with two or three coordinates.

```
sage: p1 = point([(1, 2), (2, 0.5), (1, -1)])
```



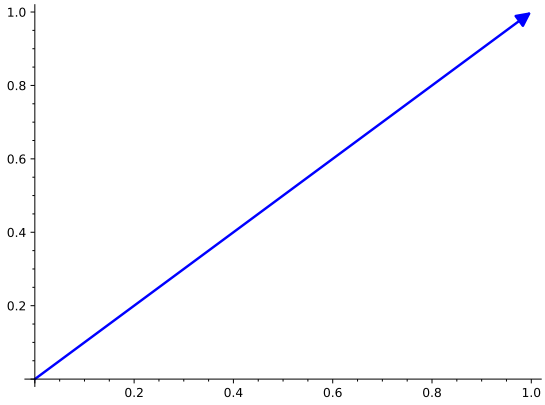
Here, the options `alpha`, `color`, `legend_label` and `marker` which we already know from `plot` are available, see Section 6.1.1. The size of the points is changed with the option `size`.

```
sage: r = [(randrange(-4, 4), randrange(-4, 4)) for i in srange
    (0,10)]
sage: p2 = point(r, color = 'orange', size = 100 )
```



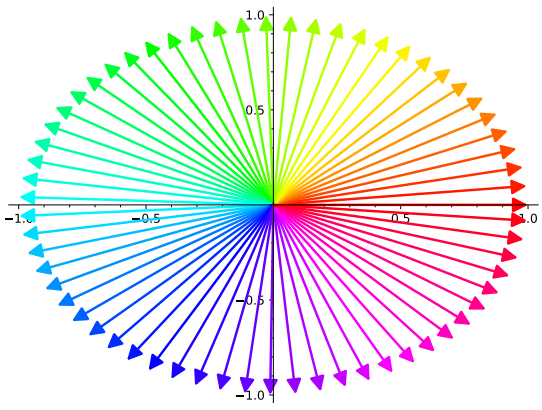
If we want to point to something in our plot we can add an arrow to our plot with `arrow`. Arrows can be used in 2D- or 3D-graphics. The command `arrow` needs two arguments. The first argument denotes the starting point and the second one denotes the ending point of the arrow, i.e. where the arrow is pointing at.

```
sage: a1 = arrow((0,0), (1,1))
```



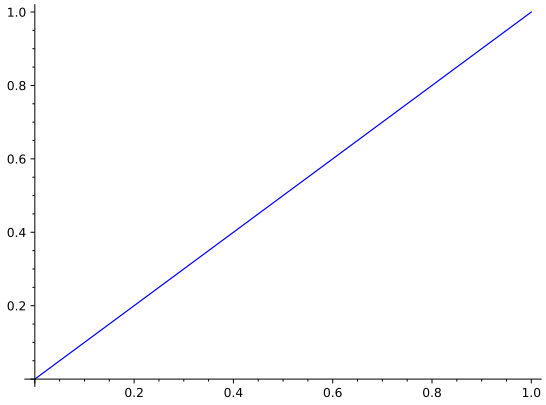
Here, the options `linestyle`, `color` and `legend_label` are available and used as explained in Section 6.1.1. In addition, the width of the arrow shaft is controlled with the option `width` and the size of the arrow head is changed with the option `arrowsize`. If the arrow should have a head at the end and at the start of the arrow we simply add `head = 2`. Combining these options we can create interesting images out of arrows, like this colorful arrow circle.

```
sage: a2 = add([arrow((0,0), (cos(x),sin(x)),color =  hue(x/(2*pi)
    ) ) for x in [0..2*pi,step=0.1]])
```



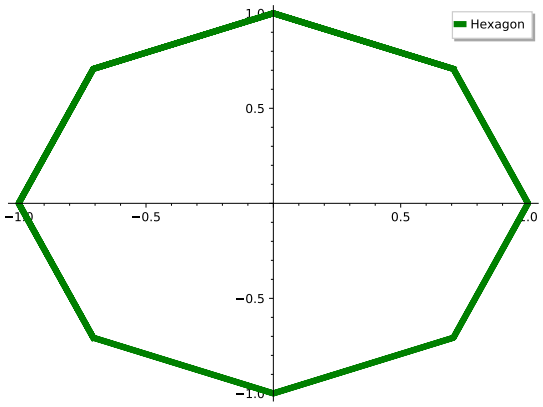
To draw a list of linked points we use `line(p)`, where p is a list of points in two or three dimensions. As usual the coordinates of points are written in a tuple, see the description of the command `point` above.

```
sage: l1 = line([(0,0), (1,1)])
```



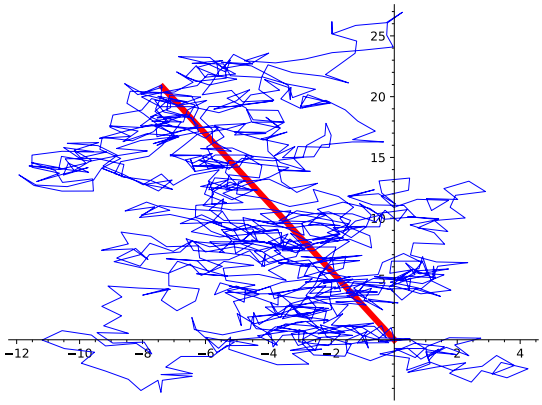
Here, most options from `plot` are available, i.e. `alpha`, `thickness`, `linestyle` `color`, `legend_label`. Their usage is described in Section 6.1.1.

```
sage: l2 = line([(cos(n/8 * 2*pi), sin(n/8*2*pi)) for n in
    [1..21]], color = "green", thickness = 5, legend_label= "␣
    Hexagon")
```



■ **Example 6.2** During a random walk, a particle starts at the origin and moves a fixed distance l every t seconds in a random direction, independently of the preceding moved. We can use `line` to draw its trajectory. Then we add a red line that shows the direct way from the initial to the final position.

```
sage: n = 1000; l = 1; x = 0; y = 0; p = [[0,0]]
sage: for k in srange(n):
....:     theta = (2 * pi * random()).n(digits= 5)
....:     x = x + l*cos(theta)
....:     y = y + l*sin(theta)
....:     p.append([x, y])
sage: g1 = line([p[n], [0,0]], color = 'red', thickness = 5)
sage: g1 += line(p, thickness = 0.4)
```



■

■ **Example 6.3** Given a real sequence $(u_n)_n$, we construct the polygonal line whose successive vertices are the points in the plane $\mathbb{R}^2$ given by

$$\begin{pmatrix} x_N \\ y_N \end{pmatrix} = \sum_{n \le N} \begin{pmatrix} \cos(2\pi u_n) \\ \sin(2\pi u_n) \end{pmatrix}. \tag{6.1}$$

If the sequence $(u_n)_n$ is uniformly distributed modulo 1, the polygonal line should behave like a random walk, and thus not go too far away from the origin. Recall that a sequence $(u_n)_n$ is uniformly distributed modulo 1 if for any interval $[a,b) \subset [0,1)$.

$$\lim_{N\to\infty} \frac{\sharp\{1 \le n \le N \ : \ (u_n - \lfloor u_n \rfloor) \in [a,b)\}}{N} = b - a.$$

We study the following three sequences and plot the corresponding polygonal line.

- $u_n = n\sqrt{2}$ with $N = 100$,
- $u_n = n\log(n)\sqrt{(2)}$ with $N = 500$,
- $u_n = \lfloor n\ln(n) \rfloor \sqrt{2}$ with $N = 500$.

First we define our series.

```
sage: n = var('n')
sage: u1(n) = n * sqrt(2)
sage: u2(n) = n * log(n) * sqrt(2)
sage: u3(n) = floor(n * log(n)) * sqrt(2)
```

Next, we construct the corresponding set of vertices. To do so, we have to calculate the points $p_m = (x_m, y_m)$ using the assignment (6.1). To improve calculation time we use the relation

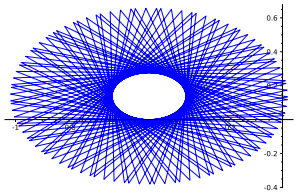
$$\begin{pmatrix} x_N \\ y_N \end{pmatrix} = \begin{pmatrix} x_{N-1} \\ y_{N-1} \end{pmatrix} + \begin{pmatrix} \cos(2\pi u_N) \\ \sin(2\pi u_N) DasB \end{pmatrix}$$

and a numerical approximation of π as the approximated precision is sufficient for our application.

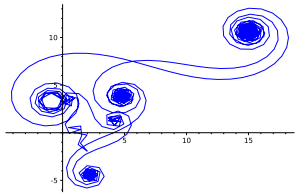
```
sage: def vert(seq, N): #constructing the list of points
....:     L = [(0,0)]
....:     for n in srange(1, N+1):
....:         x = L[-1][0] + cos(2*pi*seq(n))
....:         y = L[-1][1] + sin(2*pi*seq(n))
....:         L.append((x,y))
....:     return L
```

Now, we can plot everything.

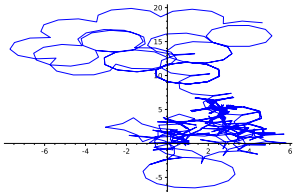
```
sage: p1 = line(vert(u1, 100))
sage: p2 = line(vert(u2, 500))
sage: p3 = line(vert(u3, 500))
```



$u_n = n\sqrt{2}$



$u_n = n\log(n)\sqrt{(2)}$

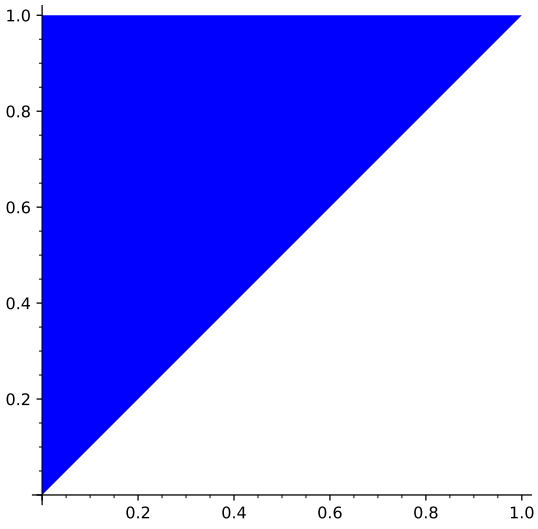


$u_n = \lfloor n\ln(n)\rfloor\sqrt{2}$

■

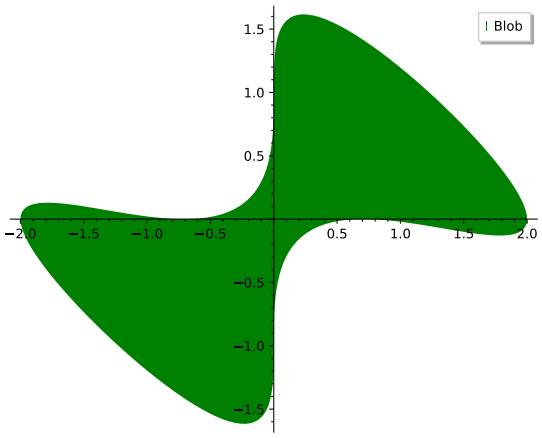
As we have seen above, we can use `line` to draw a polygon. However, the possible modifications are pretty limited. For example, we even can not fill a polygon whose boundary is drawn with `line`. To surround this problem, Sage provides the `polygon` function. This function takes a list of points and connects them with straight lines. Afterwards it connects the last point with the starting point. Hence, the result is always a closed curve. By default `polygon` creates a filled polygon.

```
sage: p1 = polygon([(0,0), (1,1), (0,1)])
```



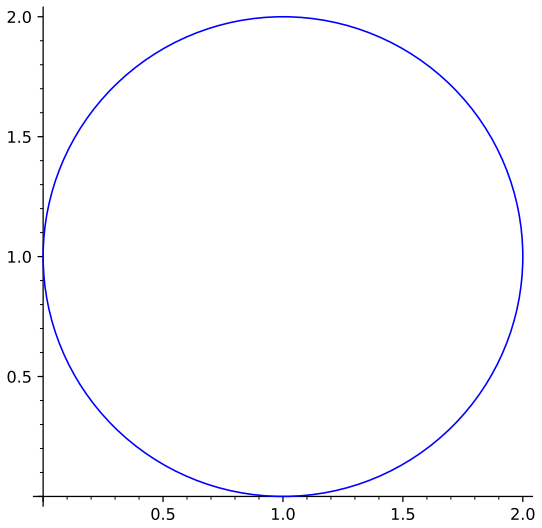
The color and the transparency of the filling can be changed with the options `color` and `alpha` respectively. To change the color of the boundary separately we use the option `edgecolor`. Here, the usage is the same as for `color`. If no filling is desired we can set `fill = False`. Morover, the options `thickness` and `legend_label` are also available. Combining these tools allows us to draw interesting shapes, like the following green blob:

```
sage: L = [[cos(pi*i/100)*(1+cos(pi*i/50)), sin(pi*i/100)*(1+sin(
    pi*i/50))] for i in range(200)]
sage: blob = polygon(L, color= 'green', legend_label = 'Blob')
```



Also a simple circle can be added as a graphics primitve in Sage. To draw a simple circle of radius r with middlepoint at p we use the command `circle(p,r)`.

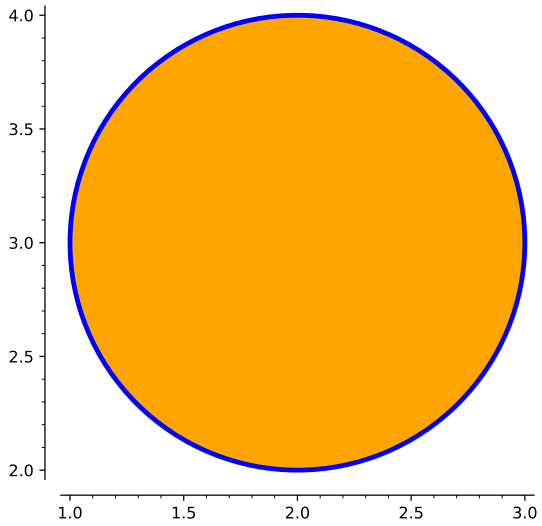
```
sage: c1 = circle((1,1), 1)
```



We can use the options `alpha`, `thickness`, `linestyle` and `legend_label` as explained in Section 6.1.1 to modify the appearance of the circle. Furthermore, setting `fill = True` adds a filling to the circle. To change the color of the plot we use the

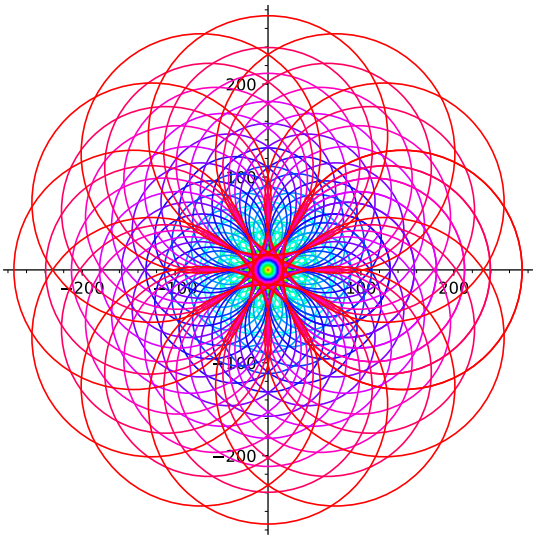
options `edgecolor` and `facecolor` to change to color of the boundary and the filling respectively.

```
sage: c2 = circle((2,3),1, fill = True, edgecolor = 'blue',
   thickness = 3, facecolor = 'orange')
```



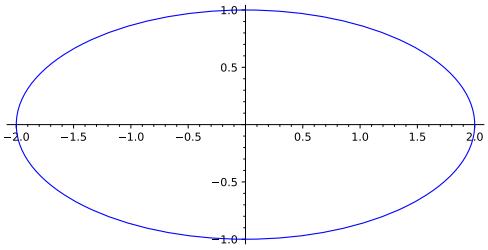
Combining these options it is also possible to create complicated and astonishing plots with many circles of different colors.

```
sage: g = Graphics() #start with empty graphic
sage: step = 6; ocur = 1/5; paths = 16
sage: PI = math.pi    # numerical for speed - fine for graphics
sage: for r in srange(1,paths+1):
....:     for x,y in [((r+ocur)*math.cos(n), (r+ocur)*math.sin(n))
    for n in srange(0, 2*PI+PI/step, PI/step)]:
....:         g += circle((x,y), ocur, color=hue(r/paths))
....:     rnext = (r+1)^2
....:     ocur = (rnext-r)-ocur
```

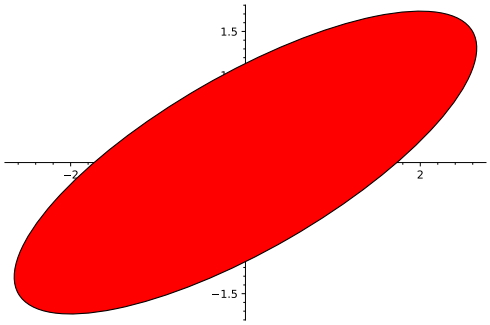


Instead of circles, we can also draw ellipses with Sage. The command `ellipse(p, r1, r2, phi)` returns an ellipse centered at the point `p` in $\mathbb{R}^2$, with radii `r1, r2` and angle `phi`. Here, the same options as for `circle` are available. Indeed, if `r1 = r2` the output is just a circle.

```
sage: e1 = ellipse((0, 0), 2, 1)
sage: e2 = ellipse((0, 0), 3, 1, pi/6, fill = True, edgecolor = '
    black', facecolor = 'red')
```



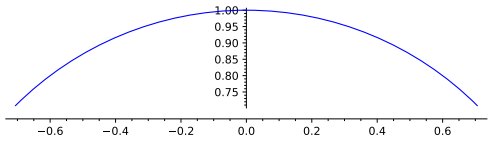
Ellipse `e1`



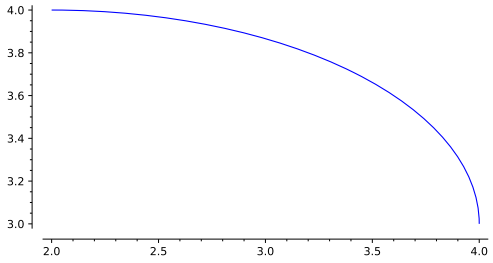
Ellipse `e2`

In some cases, we might not want to draw the whole circle or ellipses, but only a part of it. To do so, we use the command `arc`. Here, the input is the same as for `ellipse` together with a section in which the arc will be drawn. This section is defined by a two tuple containing the start and the end angle. The options `alpha`, `thickness`, `color` and `linestyle` can be used to modify the output as explained in Section 6.1.1.

```
sage: a1 = arc((0, 0), 1, sector = (pi/4, 3 * pi/4))
sage: a2 = arc((2, 3), 2, 1, sector = (0, pi/2))
```



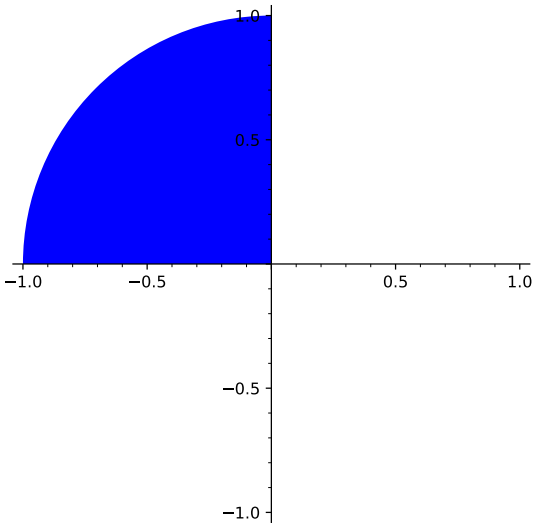
Arc of a Circle a1



Arc of an Ellipse a2

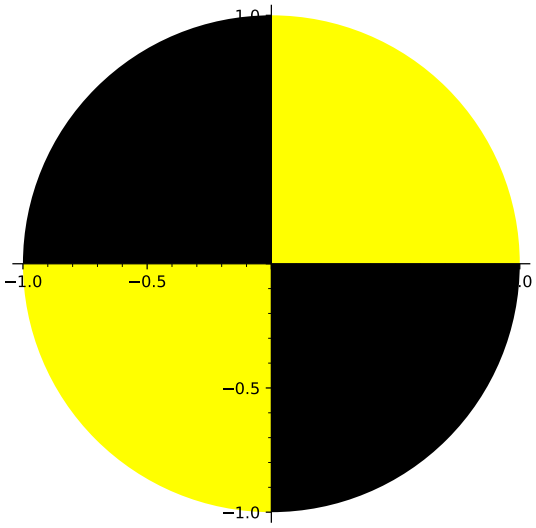
In the special case of circles Sage also provides the `disk` function to draw a filled sector or wedge of a circle. Here, we insert the coordinates of the center, the radius and a two tuple with the beginning and the ending angle.

```
sage: d1 = disk((0,0), 1, (pi/2, pi))
```



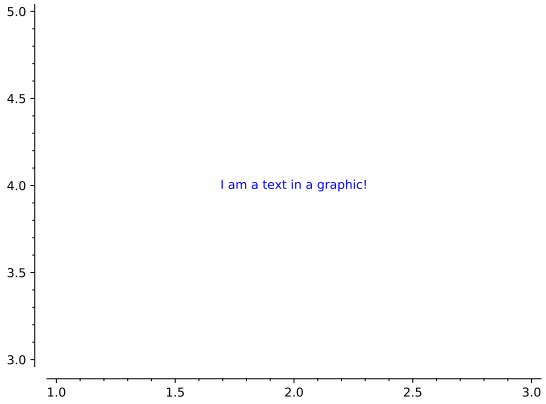
Some of the available options for `disk` are `alpha`, `color` and `legend_label`. Their usage is explained in Section 6.1.1.

```
sage: bl = disk((0.0,0.0), 1, (pi, 3*pi/2), color='yellow')
sage: tr = disk((0.0,0.0), 1, (0, pi/2), color = 'yellow')
sage: tl = disk((0.0,0.0), 1, (pi/2, pi), color = 'black')
sage: br = disk((0.0,0.0), 1, (3*pi/2, 2*pi), color='black')
sage: P  = tl+tr+bl+br
```



Last but not least it is also possible add text to our graphics object with the command `text`. There, we insert a text as a character string and a point at which the text is displayed.

```
sage: t1 = text("I␣am␣a␣text␣in␣a␣graphic!", (2, 4))
```



Below we describe the basic available options to modify the output of `text`:

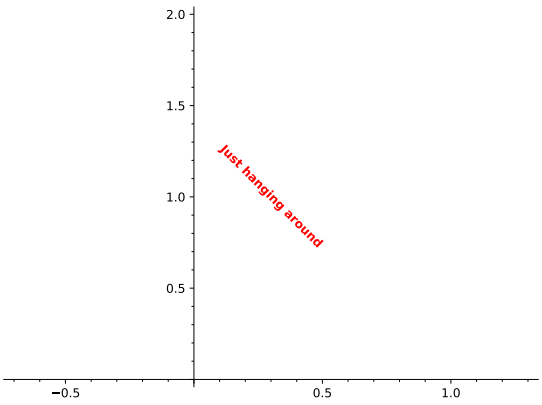
- `fontsize`: Changes the size of the text using an integer specifying the size in points.
- `fontstyle`: The font style can be changed to `'italic'` or `'oblique'`.
- `fontweight`: The thickness of the font is changed using a numeric value between 0 and 1000.
- `alpha`: Controls the transparency of the text
- `rotation`: Rotates the text with respect to a given angle.
- `color`: Changes the color of the text.

2D Graphic Primitives	
Point at (x,y)	`point((x,y))`
Arrow from p to q g	`arrow(p, q)`
Line from p to q	`line([p, q])`
Filled Polygon with Corner $p_1,\ldots,p_n$	`polygon([p1, ... , pn])`
Circle around p with Radius r	`circle(p ,r)`
Ellipse around p with Radii r_1,r_2	`ellipse(p, r1, r2)`
Arc of a Circle or an Ellipse from Angle ϕ_1 to ϕ_2	`arc(p, r1, r2, phi1, phi2)`
Sector of a Circle from Angle ϕ_1 to ϕ_2	`disk(p, r, phi1, phi2)`

Table 6.2.: 2D Graphic Primitives

- `background_color`: Changes the background color.

```
sage: t2 = text('Just␣hanging␣around', (0.3 ,1), color = 'red',
   fontweight = 800, rotation = 315)
```



7. 3D Graphics

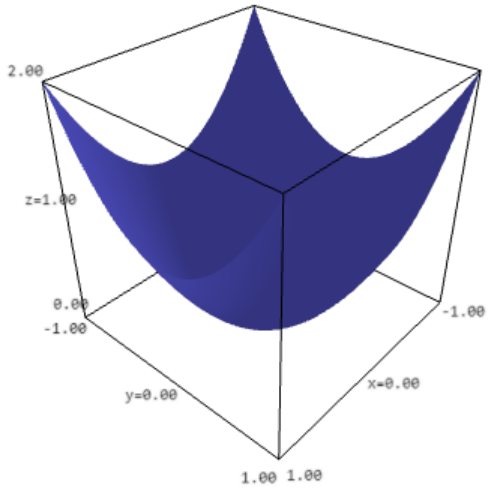
This chapter deals with the creation of 3D-graphics with Sage, like the graph of functions $f : \mathbb{R}^2 \to \mathbb{R}$, parametrized surfaces or various basic three dimensional shapes. Many command described below are three dimensional analogues of already discussed commands in Chapter 6

7.1 Plotting Functions

7.1.1 Drawing the Graph of a Function

We have already encountered the standard command `plot3d` in Section 2.3.3. This command is used to draw the graph of a function $f : \mathbb{R}^2 \to \mathbb{R}$, i.e. the surface $\{(x,y,f(x,y)) \in \mathbb{R}^3 \mid (x,y) \in \mathbb{R}^2$. If a symbolic function is handed over, like in the example below we need to ensure that all needed symbolic variables are defined beforehand.

```
sage: x, y = var('x␣y')
sage: plot3d(x^2 + y^2, (x, -1, 1), (y, -1,1))
```



Instead of a symbolic function, we can also draw the graph of a `Python` function, e.g. a lambda construction. For example, the following command returns the same plot as above.

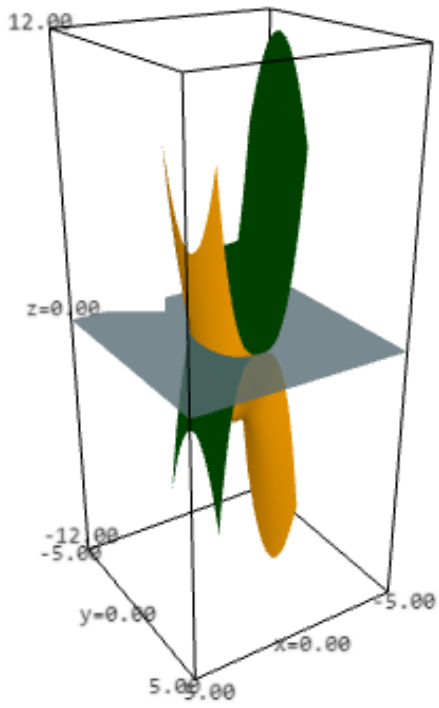
```
sage: plot3d(lambda x, y: x^2 + y^2,(x, -1, 1), (y, -1,1) )
```

Similar to `plot` Sage provides many different option to modify the graph of `plot3d`:

- `mesh`: Shows the mesh grid lines if set to `True`.
- `dots`: Shows the dots of the mesh grid if set to `True`.
- `plot_points`: Initial number of sample points in each direction.
- `opacity`: Takes values between 0 (for invisible) and 1 (for non-transparent).
- `color`: Changes the color of the graph, see Section 6.1.1 for the usage.

Moreover, we can also add three dimensional graphics objects to combine the graphs of various functions in one plot.

```
sage: L = plot3d(lambda x,y: 0, (-5,5), (-5,5), color="lightblue",
    opacity=0.8)
sage: Q = plot3d(lambda x,y: x^3 + y^2 - 4, (-2,2), (-2,2), color=
   'orange')
sage: P = plot3d(lambda x,y: 4 - x^3 - y^2, (-2,2), (-2,2), color=
   'green')
sage: L + P + Q
```

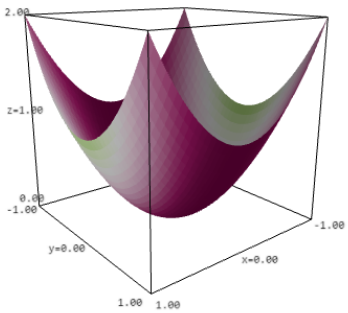


Furthermore, we can use color maps to change the color of the graph. To do so, we define a color function $c(x,y) : \mathbb{R}^2 \to [0,1]$ and combine it with an imported color map from `Matplotbib` which assigns each value in $[0,1]$ a color. The syntax is

```
color = (c, colormap.name)
```

where `name` has to be replaced with the name of the color map that is used, see Section 6.4 for a list of the available color maps. For example, we can color our parabel with the color map `PiYG` and the color function c defined below.

```
sage: def c(x,y): return (sin(x)*cos(y))^2 #color function
sage: plot3d(x^2 + y^2, (x,-1, 1), (y,-1,1), color = (c, colormaps
    .PiYG))
```



In some situation, we are not interested in the graph of a function in cartesian coordinates but in some other coordinate system like polar coordinates or cylindrical coordinates. Such coordinate changes can be added to `plot3d` with the option `transform`. There we insert a 4-tuple `(x_fct, y_fct, z_fct, independent_vars)`, where the first three entries are the transformation functions from the used coordinate system to Cartesian coordinates in terms of `u, v` and `fvar` for which the value `f` in `plot3d` is substituted. To be more concrete, we endow $\mathbb{R}^3$ with the coordinates (u,v,r), where the transformation functions are given by

$$\begin{pmatrix} u \\ v \\ r \end{pmatrix} \mapsto \begin{pmatrix} x(u,v,r) \\ y(u,v,r) \\ z(u,v,r) \end{pmatrix}.$$

Handing over this transformation to `plot3d` together with a function $f(u,v)$ returns the surface $\{(x(u,v,f(u,v)),y(u,v,f(u,v)),z(u,v,f(u,v))\}$.

■ **Example 7.1** The most common coordinate systems in $\mathbb{R}^3$ besides Cartesian coordinates are spherical and cylindrical coordinates, where the transformation function for spherical coordinates is given by

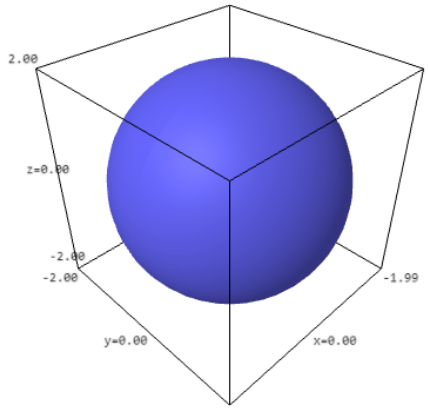
$$\begin{pmatrix} u \\ v \\ r \end{pmatrix} \mapsto \begin{pmatrix} r\cos(u)\sin(v) \\ r\sin(u)\sin(v) \\ r\cos(v) \end{pmatrix}$$

and the transformation function for cylindrical coordinates is given by

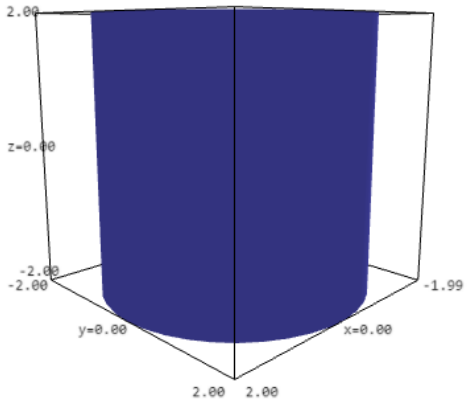
$$\begin{pmatrix} u \\ v \\ r \end{pmatrix} \mapsto \begin{pmatrix} r\cos(u) \\ r\sin(u) \\ v \end{pmatrix}$$

Next, we declare u,v to be our independent variables, i.e. we will consider functions f depending on u,v and the value of r will be replaced by the value $f(u,v)$. Although Sage provides the commands (`spherical_plot3d` and `cylindrical_plot3d`) for these coordinate systems, we use them to illustrate the usage of `transform`.

```
sage: r, u,  v= var('r,␣u,␣v')
sage: TS = (r * cos(u) * sin(v), r * sin(u) * sin(v), r * cos(v),
    [u, v]) #spherical coordinates
sage: plot3d(2, (u, 0, 2*pi), (v, 0, pi), transformation = TS) #
    sphere of radius 2
sage: TC = (r * cos(u), r * sin(u), v, [u, v]) #cylindrical
    coordinates
sage: plot3d(2, (u, 0, 2*pi), (v, -2, 2), transformation = TC) #
    cylinder of radius 2
```



Sphere of Radius Two

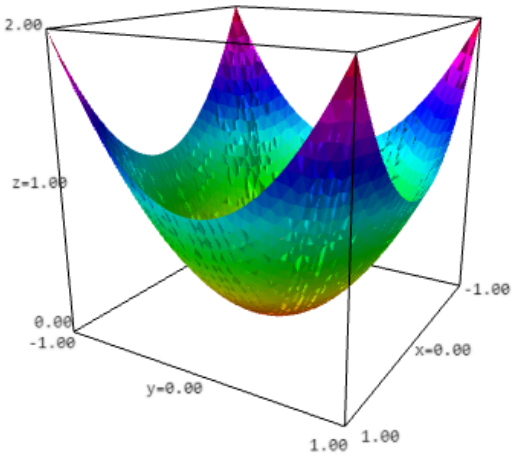


Cylinder of Radius Two

■

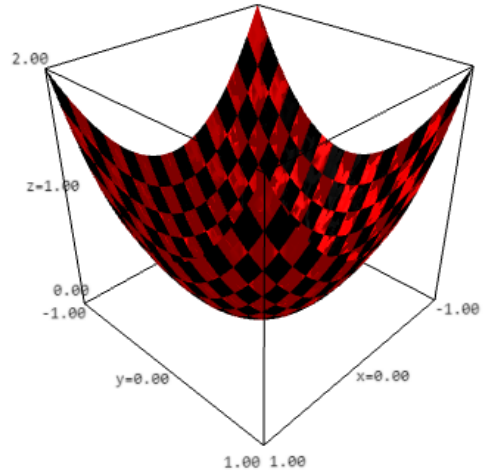
We can unlock more customizing options by setting `adaptive = True`. This slows down the computation time but might improve the result.

```
sage: plot3d(x^2 + y^2, (x,-1, 1), (y,-1,1), adaptive = True)
```



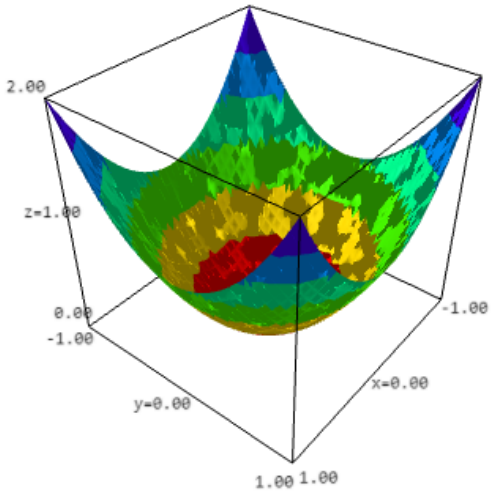
In that setting the default color is a rainbow of 128 colors. This can be changed in various ways. For example, we can insert a list of colors that are then evenly distributed over the plot. In particular, a list of two colors gives us a check board pattern according to the mesh.

```
sage: plot3d(x^2 + y^2, (x,-1, 1), (y,-1,1), adaptive = True,
   color = ['red', 'black'])
```



Another possibility is to use the `rainbow(n, s)` value, where `n` is the number of colors that are used for the rainbow, always starting with red and ending with blue, and `s` is a string denoting the used color modus, e.g. `hex` (default value) or `rgbtuple`.

```
sage: plot3d(x^2 + y^2, (x,-1, 1), (y,-1,1), adaptive = True,
    color = rainbow (7, 'rgbtuple'))
```



There also other options available to change the color or to control the precision. We refer to [3, 3D Graphics].

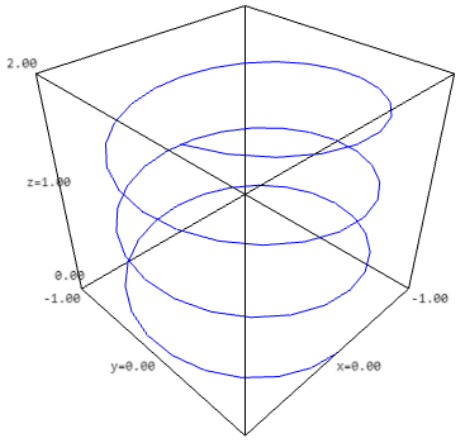
As in the two dimensional case, the `show` function allows us further modifications of the output, e.g. we can change the scaling of the axis with the option `aspect_ratio`, where `aspect_ratio = [1, 1, 1]` means that all axis are shown evenly, i.e. the sphere looks perfectly round in this setting. The `save` command exports images using the formats .png, .bmp, .gif. If we plot a 3D-graphics in the Jupyter Notebook we can interact with 3D-plot and use the button on the bottom right to save the plot.

7.1.2 Parametric Plots

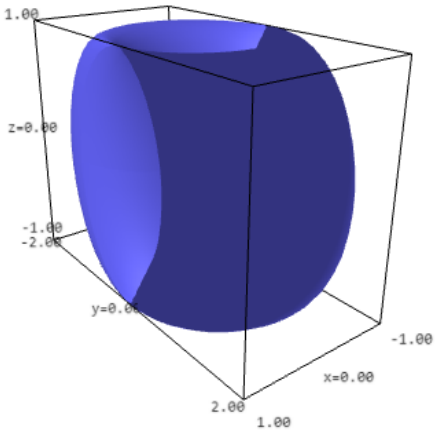
Similar to `parametric_plot`, the command `parametric_plot3d` draws parametrized curves and surfaces in three dimensions. The syntax is similar to those of `parametric_plot`

and almost all options of `plot3d` carry over. The only difference is that there is no `adaptive` available and therefore also all further options belonging to it.

```
sage: p1 =  parametric_plot3d((sin(u), cos(u), u/10), (u,0,20))
sage: p2 = parametric_plot3d((cos(u), sin(u)+cos(v), sin(v)), (u
    ,0,2*pi), (v,-pi,pi))
```



A Spiral Curved, `p1`

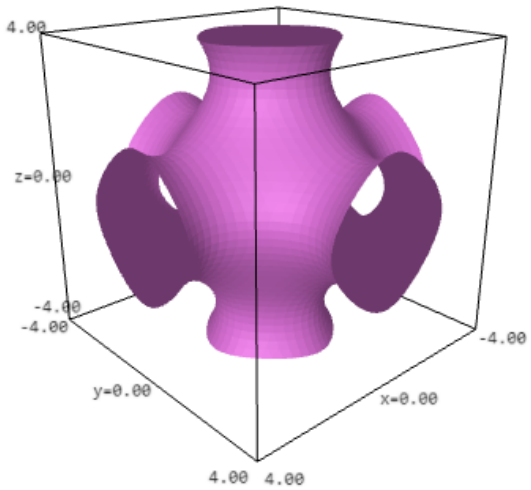


A Parametrized Surface, `p2`

7.1.3 Implicit Plot

The command `implicit_plot3d` is the three dimensional analogue to `implicit_plot`, see Section 6.1.4. The syntax is similar to those of `implicit_plot`. In addition, as for `parametric_plot3d`, all option except for `adaptive` are available.

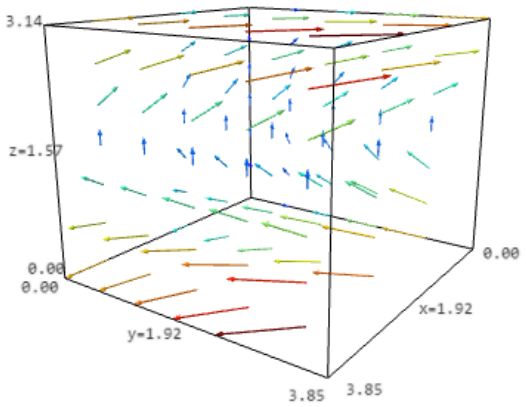
```
sage: implicit_plot3d(-(cos(x) + cos(y) + cos(z)), (x,-4,4), (y
    ,-4,4), (z,-4,4), color='orchid')
```



7.2 Vector Fields

It is also possible to plot three dimensional vector fields using `plot_vector_field3d`. The syntax is similar to the two dimensional analogue `plot_vector_field`, compare with Section 6.2. Within this command only the options `plot_point` and `color` are available.

```
sage: plot_vector_field3d((x*cos(z),-y*cos(z),sin(z)), (x,0,pi), (
   y,0,pi), (z,0,pi),plot_points=[3,5,7])
```



7.3 More Graphic Primitives

As in the two dimensional setting, Sage provides many basic three dimensional shapes. But we first have to import them with the following command before we can use them.

```
sage: from sage.plot.plot3d.shapes import *
```

The graphics primitives `plot`, `arrow`, `line`, `polygon` and `text` we introduced in Section 6.6 also work in three dimensions. However, not all options might be available. We can also call their three dimensional analogue directly by adding `3d` after their name.

We summarize the other available three dimensional graphic primitives in Table 7.1. For all of them, the options `opacity` and `color` are available. The shapes `Cone` and `Cylinder` also have the additional option `close` which can be set to `False` if the basis should not be displayed. Moreover, there is no argument for a center point. Instead, the objects are by default always centered around the origin. To move the objects to another position we can use the `translation` method which takes a translation vector. Combining these different commands we can draw nice three dimensional objects, like the following Christmas tree. Here, we only used cones and the `translation` method.

```
sage: T = sum(Cone(exp(-n/5), 4/3*exp(-n/5), color=(0, .5, 0)).
   translate(0, 0, -3*exp(-n/5)) for n in [1..7])
sage: T += Cone(1/8, 1, color='brown').translate(0, 0, -3)
```

3D Graphic Primitives	
Box with Side Lengths a, bc	`Box([a, b, c])`
Cone with Radius r and Height h	`Cone(r, h)`
Cylinder with Radius r and Height h	`Cylinder(r, h)`
Sphere of Radius r	`Sphere(r)`
Torus with Inner Radius $r1$ and outer Radius $r2$	`Torus(r1, r2)`

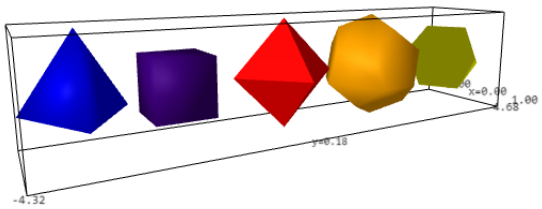
Table 7.1.: 3D Graphic Primitives

3Platonic Solids	
Tetrahedron	`tetrahedron()`
Cube	`cube())`
Octahedron	`octahedron()`
Dodecahedron	`dodecahedron())`
Icosaeder()	`icosaeder()`

Table 7.2.: Platonic Solids

Furthermore, there are all five platonic solids available in Sage. Recall that these are the only five bodies where all sides and angles are equal. Hence, their size is completely determined by the length of their sides. This length is changed with the option `size` whose default value is 1. In addition we can specify a center point.

```
sage: G = tetrahedron((0,-3.5,0), color='blue') + cube((0,-2,0),
    color=(.25,0,.5))
sage: G += octahedron(color='red') + dodecahedron((0,2,0), color='
    orange')
sage: G += icosahedron(center=(0,4,0), color='yellow')
```



Bodies.pdf Bodies.pdf

IV Algebra and Symbolic Computation

8 Computational Domains 119
8.1 Sage is Object-Oriented
8.2 Elements and Parents
8.3 Domains with a Normal Form
8.4 Expressions vs. Computational Domains
8.5 Primality Test

9 Polynomial Rings 139
9.1 Euclidean Arithmetic
9.2 Factorization and Roots
9.3 Rational Functions
9.4 Formal Power Series

10 Matrices 159
10.1 Constructions and Elementary Manipulations
10.2 Matrix Computations
10.3 Spectral Decomposition

11 Polynomial Systems 185
11.1 Polynomials in Several Variables
11.2 Polynomial Systems and Ideals
11.3 Solving Strategies

12 Differential Equations 217
12.1 First Order Ordinary Differential Equations
12.2 Second Order Equations
12.3 The Laplace Transform
12.4 Systems of Linear Differential Equations

8. Computational Domains

In mathematics, we always keep careful track on where our objects live, as the type of our objects determines the available methods. For example, it is a huge difference whether we are dealing with real numbers or with elements of a finite field. The same holds for computer algebra systems like Sage. Sage provides various ways to specify more or less rigorously the computational domain. In the following, we describe the most common computational domains and their usage.

8.1 Sage is Object-Oriented

Sage uses heavily the object-oriented programming paradigm. This paradigm consists in modeling each abstract entity we want to manipulate by a programming language construction called an ***object***. In most cases, as in Python, each object is an element of a ***class***. For example, a fraction is represented by an object, which is an element of the Rational class. In particular, the type is specified by the fraction and not by the Python variable.

```
sage: o = 12/35; type(o)
<class 'sage.rings.rational.Rational'>
sage: type(12/35)
<class 'sage.rings.rational.Rational'>
```

To be more precise, an object stores the required information to represent the corresponding entity. In contrast to that a class defines two main properties:

1. The ***data structure*** of an object, i.e. how the information is organized in the memory. For example, in the class Rational every object is characterized by two integers, the numerator and the denominator.
2. The available ***methods***.

For example, to obtain the factorization of an integer, we use the method `factor`.

```
sage: a = 300
sage: a.factor()
2^2 * 3 * 5^2
```

This syntax can be understood as follows: "Take the value of `a` and apply the method `factor` without further arguments to it". Hence, `a.factor()` is an abbreviation of

```
sage: type(a).factor(a)
2^2 * 3 * 5^2
```

which translates to "Request from the class of `a` the factorization method `type(a).factor` and apply it to `a`". Since the method `factor` is tied to the class of the object it does not have to work in every class the same. For example, if we try to factorize 300 viewed as a real number and not as an integer, the method `factor` does not return the expected factorization into prime factors.

```
sage: type(300.0)
<class 'sage.rings.real_mpfr.RealLiteral'>
sage: (300.0).factor()
300.000000000000
```

Although there are many different classes with their own collection of methods, almost all methods in Sage are ***polymorphic***, i.e. they can be applied to different classes. For example, the method `factor` can be applied to integers and to rational numbers. In both cases, the syntax is `a.factor()`, but the class determines the explicit definition of `factor`. Similarly, any product of two objects `a, b` is written as `a * b`, but the specific definition of multiplication is determined by the class of the factors.

```
sage: 3 * 7 #integers
21
sage: (2/3) * (6/5) #rational numbers
4/5
sage: (1 + I) * (1 - I) #complex numbers
2
sage: (x + 2) * (x + 1) #symbolic expressions
(x + 2)*(x + 1)
```

One advantage of polymorphic methods is that we can write generic `Python` programs which can be applied to all objects whose class admits the involved methods.

```
sage: def square(a):
....:     return a * a
sage: square(2), square(3/2), square(I), square(x+1)
(4, 9/4, -1, (x + 1)^2)
sage: M = matrix([[0, -1], [1, 0]])
sage: square(M)
[-1  0]
[ 0 -1]
```

Sage objects also have some ***introspection*** features. This means that we can always "ask" an object for its class, its methods, etc. and manipulate the obtained information. As already described in Section 5, the `type` function returns the class of an object.

```
sage: type(5)
<class 'sage.rings.integer.Integer'>
sage: type(5/1)
<class 'sage.rings.rational.Rational'>
```

Moreover, as explained in Section **??** we have access to an on-line help of methods via

```
sage: 720.factor?
```

and to the source code via

```
sage: 720.factor??
```

Last but not least, the auto-completion can be used to obtain all available methods that can be applied to an object `o`.

8.2 Elements and Parents

While the notion of objects and classes are well-known concepts from Python, Sage introduces another concept closer to mathematics: the ***parent*** of an object. For example, whether an element a is invertible or not depends not only on the element a itself but also on the mathematical set A it belongs to. For example, the number 5 is invertible in $\mathbb{Q}$ but not in $\mathbb{Z}$ since $\frac{1}{5}$ is not an integer.

```
sage: a = 5; type(a)
<class 'sage.rings.integer.Integer'>
sage: a.is_unit()
False
sage: a = 5/1; type(a)
<class 'sage.rings.rational.Rational'>
sage: a.is_unit()
True
```

In Sage, the set A to which an element a belongs to is called the ***parent*** of a. Similar to classes, we can retrieve the parent of an element `a` in Sage with the command `parent(a)`.

```
sage: parent(5)
Integer Ring
sage: parent(5/1)
Rational Field
```

The usual number domains $\mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{C}$ are already implemented in Sage.

```
sage: ZZ, QQ, RR, CC
(Integer Ring, Rational Field, Real Field with 53 bits of
   precision, Complex Field with 53 bits of precision)
```

Analogous to data structures, we can also convert the parent of objects to another parent, if possible.

```
sage: QQ(5).parent()
Rational Field
sage: ZZ(1/5)
Traceback (most recent call last):
...
Type Error: no conversion of this rational to an integer
```

In addition, parents are Python objects themselves, i.e. there are methods which can be applied to them.

```
sage: cartesian_product([QQ, QQ])
The Cartesian product of (Rational Field, Rational Field)
sage: ZZ.fraction_field() #retrieve QQ as the fraction field of ZZ
Rational Field
sage: ZZ['x'] #construction of a polynomial
Univariate Polynomial Ring in x over Integer Ring
```

8.3 Domains with a Normal Form

In this section, we describe the most commonly used parents in Sage. Each of them is a computational domain with a normal form, where a ***normal form*** is a fixed way of representing an object uniquely, e.g. any element of $\mathbb{Q}$ can be written in many different ways but there is only one way to write is as a reduced fraction with integer numerator and denominator. Hence, fixing a normal form allows, for instance, an easy determination whether two elements are equal or not.

8.3.1 Elementary Domains

Elementary domains are the classical sets of elements with no indeterminates involved, e.g. integers and rational numbers. Below, we describe the construction and properties of these various elementary domains together with their normal form.

Integers

Integers are internally represented in radix two but printed in radix ten. As integers have a unique representation they are already in normal form. As already seen above, Sage integers are objects of the class `sage.rings.integer.Integer` and belong to the parent `Integer Ring`. In particular, they differ from Python integers `int`. The conversion between Sage `Integer` and Python `int` is usually done automatically. Otherwise it can be conversed manually in the usual way.

```
sage: 5.parent()
Integer Ring
sage: type(5), type(int(5))
(<class 'sage.rings.integer.Integer'>, <class 'int'>)
```

Rational Numbers

The normal form property of integers carries over to rational numbers , i.e. elements of `QQ`. Every element in `QQ` is uniquely represented by a reduced fraction with integer numerator and denominator. Any object in `QQ` is immediately put into normal. This conversion is even done in-between every calculation step. For example, in

```
sage: factorial(99) / factorial(100) - 2/100
-1/100
```

Sage first evaluates the factorials and transforms the result to its normal form $\frac{1}{100}$. Then Sage puts the second fraction in normal form, i.e. performs the calculation $\frac{2}{100} = \frac{1}{50}$. Afterwards Sage takes the difference of these two fraction in normal form and reduces the result again to its normal form. This kind of in-between transformations are done in every domain with a normal form.

Floating-Point Numbers

Since irrational numbers can not be expressed in a finite format, their numerical value is approximated by ***floating-point numbers***. Within Sage, floating-point numbers are encoded in radix two. As a consequence, also all rational numbers that can not be exactly represented in binary, like $\frac{1}{10}$, are only approximated. Thus, the numerical value of the input `0.1` differs slightly from the real numerical value of $\frac{1}{10}$.

The parent of a real floating-point number with p-bit significant is `Reals(p)`. The default value is $p = 53$ and is denoted by `RR`.

```
sage: RR, Reals(16)
(Real Field with 53 bits of precision, Real Field with 16 bits of
   precision)
```

In calculations, floating-point numbers are dominant, i.e. as soon as a floating-point number is involved, the complete expression is evaluated as a floating-point number.

```
sage: 3*4/22 - 6*0.7
-3.65454545454545
sage: cos(1), cos(1.0)
(cos(1), 0.540302305868140)
```

Complex Floating-Point Numbers

The floating-point approximations of complex numbers with precision p are elements of `Complexes(p)`, or its alias `ComplexField(p)`, or simply `CC` for complex floating-point numbers with the default precision $p = 53$. The normal form of complex numbers is the representation $z = x + iy$ with x, y being real floating-point numbers.

```
sage: z = CC(1, 2); z
1.00000000000000 + 2.00000000000000*I
```

R *We have already seen the imaginary unit i in computations with symbolic expressions in Section 3.1. However, the parent of the symbolic expressions I is the symbolic ring and not* `CC`.

```
sage: (1 + 2*I).parent()
Symbolic Ring
sage: CC(1 + 2*I).parent()
Complex Field with 53 bits of precision
```

Booleans

Event though logic expressions form a computational domain with the two normal forms `True` and `False`, the class of boolean values has no specific parent in `Sage`. Nevertheless, they are essential in the construction of `Python` functions, for example as conditions for `if-else`-constructions or `while` loops.

Logic expressions are simply evaluated from left to right. This means, the evaluation of the operator `or` terminates as soon as the first `True` is encountered without bothering the remaining elements. The same happens with `and` and `False`. Hence, the following divisibility test of b by a does not produce an error if $a = 0$.

```
sage: a = 0; b = 12
sage: (a == 0 and b == 0) or (a != 0 and b % a == 0)
False
```

The boolean operators are ordered as follows: The operator `not` is dominant over the operator `and` which in turn is dominant over the operator `or`. Thus, the above example can be rewritten as follows:

```
sage: a == 0 and b == 0 or not a == 0 and b % a ==  0
False
```

Furthermore, `Sage` allows multiple equality or inequality tests like in mathematics, e.g. $x \leq y < z \leq t$ corresponds to `x <= y < z <= t`. Usually, a boolean operation is evaluated automatically. Otherwise we can use the command `bool` to force the evaluation.

```
sage: x, y = var('x, y')
sage: bool((x-y)*(x+y) == x^2-y^2)
True
```

Integers Modulo n

The parent ring $\mathbb{Z}/n\mathbb{Z}$ is constructed, in `Sage`, via

```
sage: Z4 = IntegerModRing(4); Z4
Ring of integers modulo 4
```

As done in mathematics, the normal form of an element in $\mathbb{Z}/n\mathbb{Z}$ is its value modulo n. Any computation involving an element from $\mathbb{Z}/n\mathbb{Z}$ is automatically reduced modulo n.

```
sage: m = Z4(7); m
3
sage: 3 * m + 5
2
```

If n is a prime number the quotient $\mathbb{Z}/n\mathbb{Z}$ is not only a ring but a field. Since the available operations for fields differs from those of rings, we can choose to build $\mathbb{Z}/p\mathbb{Z}$, where p is a prime number, as a finite field. The normal form for finite fields is the same as for the ring$\mathbb{Z}/n\mathbb{Z}$. Observe that the resulting parent is now field and not a ring. Hence, `IntegerModRing(3)` is another parent than `GF(3)` althoug both describe the mathematical set $\mathbb{Z}/3\mathbb{Z}$.

```
sage: Z3 = GF(3); Z3 #GF stands for Galois Field
Finite Field of size 3
sage: IntegerModRing(3) == GF(3)
False
```

8.3.2 Compount domains

In Sage we can use already defined computation domains with a normal form to construct new computatio domains with a normal, so-called compount domains. The most important ones are matrices, polynomials, rational functions and truncated power series. The properties of these domains are mostly determined by the computation domain of their coefficient. For example, polynomials with rational coefficients behave differently than polynomials with coefficients in a finite field.

Matrices

A matrix with coefficients in the computation domain D is in normal form if all of its entries are in the normal form corresponding to D.

```
sage: a = matrix(QQ, [[1, 2, 3], [2, 4, 8], [3, 9, 27]])
sage: (a^2 + 1) * a^(-1)  #1 denotes the identity matrix
[  -5 13/2  7/3]
[   7    1 25/3]
[   2 19/2   27]
sage: parent(a)
Full MatrixSpace of 3 by 3 dense matrices over Rational Field
```

Here, the `matrix` function is a shortcut for the following procedure: First, Sage builds the corresponding parent and then uses it to construct the matrix. We replicate this procedure below.

```
sage: M = MatrixSpace(QQ, 3, 3); M #first construct the parent
Full MatrixSpace of 3 by 3 dense matrices over Rational Field
sage: a = M([[1, 2, 3], [2, 4, 8], [3, 9, 27]]) #construct the
    matrix with parent M
```

The usage of matrices is discussed in detail in Chapter 10.

Polynomials and Fraction Fields

Similar to matrices, the parent of a polynomial depends on the parent of the coefficients. For example, the polynomial rings $\mathbb{Z}[x]$ and $\mathbb{C}[x,y,z]$ are build via

```
sage: P = ZZ['x']; P
Univariate Polynomial Ring in x over Integer Ring
```

```
sage: Q = CC['x, y, z']; Q
Multivariate Polynomial Ring in x, y, z over Complex Field with 53
    bits of precision
```

We have already seen in Section 3.1.3 that there is no optimal representation. In Sage, an element of a polynomial ring is in normal form if it is expanded and all the coefficients are in normal form. Thus, a polynomial ring has a normal form if and only if the computation domain of the coefficients has a normal form.

```
sage: p = P(x+1) * P(3*x^2 - 4); p
3*x^3 + 3*x^2 - 4*x - 4
sage: q = Q(x+1) * Q((3 + 2*I)*x)^2; q
(5.00000000000000 + 12.0000000000000*I)*x^3 + (5.00000000000000 +
   12.0000000000000*I)*x^2
```

A fraction field of an integral domain R , i.e. a commutative ring where the products of any two nonzero elements is nonzero, is the smallest field containing R. It is not hard to see that the fraction field of a polynomial ring $P[x]$ is given by

$$F(x) = \left\{ \frac{p(x)}{q(x)} \ : \ p(x), q(x) \in P[x] \right\}.$$

In Sage fraction fields are easily constructed with the method `fraction_field`. Similar to the normal form of `QQ`, the normal form in a fraction field is given by a reduced fraction with numerator and denominator being in normal form.

```
sage: F = P.fraction_field(); F
Fraction Field of Univariate Polynomial Ring in x over Integer
   Ring
sage: p + 1/p
(9*x^6 + 18*x^5 - 15*x^4 - 48*x^3 - 8*x^2 + 32*x + 17)/(3*x^3 + 3*
   x^2 - 4*x - 4)
```

We discuss the available methods for univariate polynomials in Chapter 9. The usage of multivariate polynomials is explained in Chapter 11.

R *Polynomials in a polynomial ring differ from the polynomial expressions in the symbolic ring we have discussed in Section 3.2.1. If the coefficients are in the symbolic ring, there is no well-defined coefficient type. However, the latter can be useful to mix polynomials and other expressions, but the price we pay is that we have to modify the results manually with commands like* `expand`.

```
sage: p = (x + 1)*(x - 1)
sage: parent(p)
Symbolic Ring
sage: p
(x + 1)*(x - 1)
sage: ZZ['x'](p)
x^2 - 1
```

Algebraic Number Fields

An ***algebraic number field*** is a finite field extension of $\mathbb{Q}$, i.e. they are fields containing $\mathbb{Q}$ which have the structure of a finite dimensional vector space over $\mathbb{Q}$. The study of number fields is a central topic in algebraic number theory. One common way to obtain these fields is to "add" algebraic numbers to it. We shortly recall that an algebraic number is a complex number which can be realized as the root of a polynomial with rational coefficients. For example, the roots of the polynomial x^2+1 are $\pm i$. Hence, $\pm i$ are algebraic numbers. The corresponding number field are the so-called ***Gaussian rationals***,

$$\mathbb{Q}[i] = \{a+ib \ : \ a,b \in \mathbb{Q}\}.$$

To construct this number field in Sage we use the construction `NumberField` together with the defining polynomial x^2+1 of i.

```
sage: k.<a> = NumberField(x^2 + 1) #the added number is called a
sage: k
Number Field in a with defining polynomial x^2 + 1
sage: a^2 #behaves as the imaginary unit i
-1
sage: (3/2 + 4/3 * a) * (1/7 - 2/9 * a) #is put into normal form
-1/7*a + 193/378
```

Equivalently, we can construct number fields using the "mathematical syntax".

```
sage: k.<a> = QQ[I]; k
Number Field in I with defining polynomial x^2 + 1 with I = 1*I
sage: a^2
-1
```

Sage also provides a parent for the field of all algebraic numbers, `QQbar`, or only its real subfield, `AA`. These numbers are displayed using numerical approximations. Nevertheless they are stored exactly and can be used for further computations. For example, we can calculate the roots of the polynomial x^2-2 explicitly in `AA`.

```
sage: p = AA[x](x^2 - 2)
sage: roots = p.roots(); roots
[(-1.414213562373095?, 1), (1.414213562373095?, 1)]
sage: a = roots[0][0]^2; a
2.000000000000000?
sage: a.simplify(); a
2
```

At this point, we shortly want to introduce two common methods for algebraic numbers. The method `minpoly` applied to an algebraic number a returns the ***minimal polynomial*** of a, i.e. the polynomial of least degree having a as a root, and the method `deegree` returns the ***degree*** of an algebraic number which is the degree of the minimal polynomial.

```
sage: roots[0][0].minpoly(), roots[0][0].degree()
```

```
(x^2 - 2, 2)
```

8.4 Expressions vs. Computational Domains

During our first calculations with Sage, compare Chapter 3, we only worked with symbolic expressions, whose computational domain is the ***symbolic ring***.

```
sage: parent(sin(x))
Symbolic Ring
```

The properties of this ring are rather fuzzy. For example, all computation rules assume, roughly speaking, that all symbolic variables are in $\mathbb{C}$. Since a symbolic expressions can have many different forms (polynomials, fractions, trigonometric expressions) there is no distinctive normal form, i.e. one mathematical expression can be represented in different ways. Therefore, Sage only performs basic simplifications automatically. Any other transformation has to be done manually using the methods introduced in Section 3.

8.4.1 Symbolic Polynomials vs. Polynomial Rings

Here, we compare the behavior of polynomials in a constructed polynomial ring, as introduced in Section 8.3.2, and polynomials viewed as symbolic expressions in the symbolic ring. In the ring `QQ['x1, x2, x3, x4']` all elements are put automatically into normal form, i.e. in the expanded form.

```
sage: R.<x1, x2, x3, x4> = QQ['x1, x2, x3, x4']
sage: R
sage: x1 * (x2 - x3) #any element is put into normal form
\end{Customsage}
Although a normal form makes it very easy to test whether two
   elements are equal, the expanded form is not always optimal.
   For example, the Vandermonde determinant $\prod_{1 \leq i < j
   \leq n}$ has the following normal form.
\begin{sagecommandline}
sage: prod((a - b) for (a, b) in Subsets([x1, x2, x3, x4], 2))
\end{sagecommandline}
These are $4! = 24$ terms. Doing the same construction in the
   symbolic ring, the Vandermonde determinant keeps his factored
   form which is much more compact and readable.
\begin{sagecommandline}
sage: x1, x2, x3, x4 = SR.var('x1, x2, x3, x4')
sage: prod((a - b) for (a, b) in Subsets([x1, x2, x3, x4], 2))
\end{sagecommandline}
A factored representation allows faster gcd computations and shows
    the roots of a polynomial. Although the factored form of a
   polynomials is also a normal form, it is not recommended to
   put every polynomial automatically into its factored form
   since the factorization of a polynomial is computationally
   expensive operation. Hence, we see that the normal form is not
```

```
    always the optimal form. Thus, when working in the symbolic
   ring, \Sage only does basic simplifications automatically and
   provides further specialized commands that can be used to
   modify the expression further.  Moreover, the factorization in
    the symbolic ring might not be the factorization we are
   looking for. Since factorizing a polynomial means to write it
   as a product of irreducible polynomials, \Sage should know
   whether a polynomial is irreducible or not. But, this depends
   on the domain of the coefficient. As there is no specified
   domain for the coefficients in the symbolic, \Sage only
   proposes one possible factorization.
\begin{sagecommandline}
sage: x = var('x'); p = 54*x^4 + 36*x^3 - 102*x^2 - 72*x - 12
sage: factor(p)
\end{sagecommandline}
The factorization of the polynomial $p$ becomes unique as soon as
   we fix the polynomial ring in which $p$ lives. Depending on
   the polynomial ring, we obtain different factorizations.
\begin{sagecommandline}
sage: ZZ['x'](p).factor() #factor p as a polynomial with integer
   coefficients
sage: QQ['x'](p).factor() #factor p as a polynomial with rational
   coefficients
sage: GF(5)['x'](p).factor()  #factor p as a polynomial with
   integer coefficients modulo 5
\end{sagecommandline}
Since $\sqrt{2}$ is the only irrational root, it might be useful
   to build the number field $\bbQ(\sqrt{2})$ and interpret $p$
   as a polynomial with coefficients in that number field. This
   returns us an explicit factorization of $p$ into linear
   irreducible polynomials.
\begin{sagecommandline}
sage: QQ[sqrt(2)]['x'](p).factor()
\end{sagecommandline}

\subsection{Conclusion}
\begin{sagesilent}
reset()
\end{sagesilent}

We have seen that many computations can be carried out in the
   symbolic ring or in a special computation domain. Both of
   these computation domains  have their own advantages and
   disadvantages.
```

```
On the one hand, symbolic expressions are very flexible and can be
    used to combine different expressions without any effort, e.g
   .\ mixing polynomials and trigonometric expressions. However,
   the simplification of symbolic expressions is a quite tedious
   task which could also lead to misleading results. On the other
    hand, we can construct our own computation domain explicitly.
    This gives us full control over the available operations and,
    if there is a normal form, we can easily recognize whether
   two elements are mathematical equal. But, this setting is also
    quite restrictive, e.g.\ adding a new variable means that we
   have to construct a new computation domain which includes this
    new variable and transform everything to this new computation
    domain.

To summarize, the main advantage of symbolic expressions is their
   easy usage: no explicit declaration of the computational
   domain is needed, easy addition of new variables or functions,
    easy change of the computations domains, use of all possible
   calculus tools... The main advantage of computational domains
   are the more rigorous computations, the normal form and an
   easy access to more advanced constructions, like computations
   in a finite field or in a number field. Thus, there is no
   general recommendation. It depends on the explicit situation
   in considerations which approach is more useful.


\chapter{Finite Fields}

Finite rings and fields are basic objects in number theory and
   throughout computer algebra. Indeed, many algorithm in
   computer algebra involve computations over finite fields. For
   example the rational reconstruction and the Chinese remainder
   Theorem can transform calculations on fractions into more
   efficient calculations over finite fields.

\section{Finite Fields and Rings}
\begin{sagesilent}
reset()
\end{sagesilent}

As seen in Section \ref{subsubsec:part04:compdomain:modn}, the
   parent corresponding to  the ring $\bbZ / n \bbZ$ \index{
   finite ring}\index{integer modulus ring} is built with the
   command \command{IntegerModRing(n)}.  All objects with this
   parent are automatically put in their normal form, i.e.\ they
   are reduced modulo $n$, without displaying it explicitly in
```

```
   the output.
\begin{CustomSage}
sage: Z15 = IntegerModRing(15); Z17 = IntegerModRing(17)
sage: a = Z15(3); b = Z17(3); a, b #the output is the same
(3, 3)
sage: a == b #but they are not the same
False
```

Given an integer modulo n, we can retrieve the corresponding parent with the methods `base_ring` or `parent`. The method `characteristic` applied to the computation domain itself returns the value of n, i.e. the characteristic of the ring $\mathbb{Z}/n\mathbb{Z}$.

```
sage: a.parent(), a.base_ring()
(Ring of integers modulo 15, Ring of integers modulo 15)
sage: Z15.characteristic()
15
```

As $\mathbb{Z}/n\mathbb{Z}$ is a ring, all basic arithmetic operations (addition, subtraction, and multiplication) are available. In calculations, ements of $\mathbb{Z}/n\mathbb{Z}$ are dominant. This means that a calculation is carried out modulo n as soon as one element of $\mathbb{Z}/n\mathbb{Z}$ is involved.

```
sage: a - 14 + a^3
1
```

In general, $\mathbb{Z}/n\mathbb{Z}$ is only a ring and not a field. Hence, not every element has a multiplicative inverse. Nevertheless, we can use the division operator `/` in $\mathbb{Z}/n\mathbb{Z}$. If the division is defined, Sage carries out the calculation as usual. Otherwise an error is raised.

```
sage: 1/(a+1) # 4 mod 15 is invertible since 4 * 4 mod 15 = 16 mod
    15 = 1
4
sage: 1/a #3 mod 15 is not invertible since 3*5 mod 15 = 0
Traceback (most recent call last):
...
ZeroDivisionError: inverse of Mod(3, 15) does not exist
```

To lift an element $a \in \mathbb{Z}/n\mathbb{Z}$ to $\mathbb{Z}$ we can either use the `lift` method or simply convert it to an integer using `ZZ`.

```
sage: z = a.lift(); y = ZZ(a); z, y
(3, 3)
sage: z.parent() , y.parent()
(Integer Ring, Integer Ring)
sage: y == z
```

Next, we recall a few basic definitions for elements in $\mathbb{Z}/n\mathbb{Z}$. The ***additive order*** of an element $a \in \mathbb{Z}/n\mathbb{Z}$ is the smallest integer $k > 0$ such that $ka = 0 \bmod n$. It is not hard to see that $k = \frac{n}{g}$, where $g = \gcd(a,n)$. In Sage, the additive order of an element is returned by the method `additive_order`. The ***multiplicative order*** of an invertible

element $a \in \mathbb{Z}/n\mathbb{Z}$ is the smallest integer k such that $a^k = 1 \bmod n$ and is calculated with the method `multiplicative_order`.

In the special case, where the multiplicative order of an element a equals the number of all invertible elements in $\mathbb{Z}/n\mathbb{Z}$, i.e. the order of multiplicative group $(\mathbb{Z}/n\mathbb{Z})^*$, the group $(\mathbb{Z}/n\mathbb{Z})^*$ is cyclic with generator a.

```
sage: [[x, Z15(x).multiplicative_order()] for x in srange(1, 15)
    if gcd(x, 15) == 1]
[[1, 1], [2, 4], [4, 2], [7, 4], [8, 4], [11, 2], [13, 4], [14,
    2]]
sage: [[x, Z17(x).multiplicative_order()] for x in srange(1, 17)
    if gcd(x, 17) == 1]
[[1, 1], [2, 8], [3, 16], [4, 4], [5, 16], [6, 16], [7, 16], [8,
    8], [9, 8], [10, 16], [11, 16], [12, 16], [13, 4], [14, 16],
    [15, 8], [16, 2]]
```

In the above example, the length of the lists corresponds to the order of the respective multiplicative group. Hence, the multiplicative group $(\mathbb{Z}/15\mathbb{Z})^*$ has 8 elements, but the multiplicative order of all of its element is 4 or less. Thus, $(\mathbb{Z}/15\mathbb{Z})^*$ is not a cyclic group. In the case of the multiplicative group $(\mathbb{Z}/17\mathbb{Z})^*$, we observe that there are 8 elements whose multiplicative order is 16. As the order $(\mathbb{Z}/17\mathbb{Z})^*$ is equal to 16, it is a cyclic group and there are 8 possible generators $\{3,5,6,7,10,11,12,14\}$.

One further important operation on $\mathbb{Z}/n\mathbb{Z}$ is the ***modular exponentiation***, i.e. the calculation $a^k \mod n$. For example, the RSA crypto-system relies on this operation. Sage provides the method `power_mod` to do this calculation. The used algorithm is much more efficient than computing first a^k and then taking the residue. At this point, we shortly mention that the calculation time can be displayed with`%timeit`. Note, that the output of `%timeit` depends on the hardware and used memory. Thus, it can differ from computer to computer.

```
sage: n = 3^100000; a = n-1; k = 100
sage: %timeit (a^k) % n
1 loop, best of 5: 1.37 s per loop
sage: %timeit power_mod(a,k,n)
10 loops, best of 5: 24.1 ms per loop
```

Among finite integer rings, the rings $\mathbb{Z}/p\mathbb{Z}$ with p being a prime number, are special. Since every element in $\mathbb{Z}/p\mathbb{Z}$ except 0 has a multiplicative inverse, $\mathbb{Z}/p\mathbb{Z}$ is not only a finite ring but a ***finite field***, also called ***Galois field***. We have already seen in Section 8.3.1 that these finite fields can be constructed with `GF`.

```
sage: F17 = GF(17) #constructed Z17 as a finite field
sage: F17.parent(), Z17.parent() #parents are different
(<class 'sage.rings.finite_rings.finite_field_prime_modn.
    FiniteField_prime_modn_with_category'>, <class 'sage.rings.
    finite_rings.integer_mod_ring.
    IntegerModRing_generic_with_category'>)
sage: [1/F17(x) for x in srange(1, 17)] #all elements are
```

```
    invertible.
[1, 9, 6, 13, 7, 3, 5, 15, 2, 12, 14, 10, 4, 11, 8, 16]
```

8.4.2 Applications

One application of modular methods is the ***rational reconstruction*** which is based on the following lemma.

Lemma 8.4.1 Let $a,n \in \mathbb{N}$ with $0 < a < n$. Then there exists at most one pair of coprime integers $x,y \in \mathbb{Z}$ such that $x \equiv a y \bmod n$ and $0 < |x|, y \leq \sqrt{\frac{n}{2}}$.

This result allows us, in some cases, to replace the computation of the rational number $\frac{x}{y}$ by the computation of $a \bmod n$ and then recover the rational number with ***rational reconstruction***. The second approach is often more efficient, since rational computations often involve costly gcd calculations due to the normal form in the computation domain `QQ`, see Section 8.3.1. But, such a pair x,y does not exist for all a,n, e.g. $a = 2$ and $n = 5$. To ensure that the rational reconstruction is possible, it is necessary to assume that n is sufficiently large relative to x and y. Typically, a range for x and y is known a priori, i.e. $|x| \leq N$ and $0 < y \leq D$. Then the rational reconstruction is possible for all choices of $n \geq 2ND$ and $0 < a \leq n$. The method `rational_reconstruction` takes two integers a,n and computes the rational reconstruction, i.e. it returns the fraction $\frac{x}{y}$ such that $x \equiv a y \bmod n$.

```
sage: rational_reconstruction(411, 1000)
-13/17
sage: (-13/17)%1000 #test
411
sage: rational_reconstruction(409, 1000) #this is not solvable
Traceback (most recent call last):
...
ArithmeticError: rational reconstruction of 409 (mod 1000) does
    not exist
```

We illustrate the usage of rational reconstruction in the computation of harmonic numbers $H_n = 1 + \frac{1}{2} + \ldots + \frac{1}{n}$. The first naive attempt is to simply use rational numbers and just type the above definition into a `Python` function.

```
sage: def harmonic(n):
....:     return(add([1/x for x in srange(1, n+1)]))
```

Since `Sage` puts each intermediate result into its normal form, i.e. reduces the fraction completely, there are many gcd calculations involved. These can be avoided by using rational reconstructions, i.e. we choose a sufficiently large integer m. Then we calculate $H_n \bmod m := a$ and retrieve the harmonic number H_n via the rational reconstruction of a and m. Since we need to ensure that the rational reconstruction is solvable, we need to find a priori bounds on the numerator and denominator of H_n, as explained above. First, we observe that $H_n = \frac{p_n}{q_n}$, where $q_n = \operatorname{lcm}(1,2,\ldots,n)$. Next, we use the fact that $H_n \leq (\log(n)+1)$ and obtain the upper bound $p_n \leq q_n \cdot (\log(n)+1)$. Thus, we have to choose an $m > 2q_n^2(\log(n)+1)$.

```
sage: def harmonic_mod(n, m): #calculating H_n modulo
....:     return add([1/x % m for x in srange(1, n+1)])
sage: def harmonic2(n):
....:     q = lcm(srange(1, n+1))
....:     pmax = RR(q * (log(n)+1))
....:     m = ceil(2*pmax*q)+1
....:     a = harmonic_mod(n, m)
....:     return rational_reconstruction(a, m)
sage: harmonic(8) == harmonic2(8)
True
```

Another useful application of modular arithmetics involves the Chinese Remainder Theorem.

Theorem 8.4.2 Let $n_1,\ldots,n_k$ be pairwise coprime numbers and $a_1,\ldots,a_k \in \mathbb{Z}$ arbitrary. Then there exists exactly one integer $x < \prod_j n_j$ such that $x \equiv a_j \bmod n_j$ for all $1 \leq j \leq k$.

In addition to this existence statement there is an explicit algorithm how to calculate this number x. We shortly describe it in the case $k = 2$: Let m,n be two coprime integers, and $a,b \in \mathbb{Z}$ be arbitrary. Our goal is to determine an x such that

$$x \equiv a \bmod m, \qquad x \equiv b \bmod n.$$

Since $x \bmod m = a \bmod m$ there is a $\lambda \in \mathbb{Z}$ such that $x = a + \lambda m$. Inserting this into the second equality, we obtain $\lambda_0 = \frac{b-a}{m} \bmod n$. Thus,

$$x = x_0 + \mu nm$$

with $x_0 = a + \lambda_0 m$ and $\mu \in \mathbb{Z}$ chosen such that $0 \leq x < mn$. This algorithm is implemented in Sage and can be called with `crt(a, b ,n, m)`.

```
sage: a = 2; b = 3; m = 5; n = 7
sage: lambda0 = ((b-a)/m) % n #Doing the algorithm manually
sage: a + lambda0*m
17
sage: crt(2, 3, 5, 7)
17
```

It is also possible to use two list of integers as arguments in `crt(a,n)`, where the list `a` contains the remainders and `n` contains the divisors. As an example we compute the harmonic numbers H_n using the Chinese Remainder Theorem. There, we first compute $H_n \bmod m_i$ with consecutive prime numbers $m_1, ldots, m_l$. Then the Chinese Remainder Theorem returns us the value $a = H_n \mod m_1 \cdot m_k$ and we can recover H_n with the rational reconstruction applied to a and $m_1 \cdot m_k$.

```
sage: def harmonic3(n):
....:     q = lcm(range(1, n+1))
....:     pmax = RR(q * (log(n)+1))
....:     B = floor(2 * pmax^2) #calculating upper bound
```

```
....:     a = 0; m = 1; p = 2^63
....:     while m < B: #succesives application of crt
....:         p = next_prime(p)
....:         b = harmonic_mod(n,p)
....:         a = crt(a, b, m, p)
....:         m = m * p
....:     return rational_reconstruction(a, m)
sage: harmonic3(8) == harmonic(8)
True
```

8.4.3 Example: The Aliquot sequence

The Aliquot sequence associated to a positive integer s_0 is the sequence $(s_k)_{k\in\mathbb{N}}$ defined by the recurrence relation

$$s_{k+1} = \sigma(s_k) - s_k,$$

where $\sigma(s_k)$ is the sum of the proper divisors of s_k, i.e. s_{k+1} is the sum of all proper divisors of s_k without s_k itself. The iteration stops as soon as $s_k = 1$ which means that s_{k-1} is prime. For example, the Aliquot sequence for $s_0 = 30$ is

$$30, 42, 54, 66, 78, 90, 144, 259, 45, 33, 15, 9, 4, 3, 1.$$

But not every Aliquot sequence terminates. Sometimes, the sequence $(s_k)_k$ enters a cycle. If the cycle has length one, the corresponding number is called ***perfect***, i.e. $6 = 1+2+3$ or $28 = 1+2+4+7+14$. When the cycle has length two, the two integers of the cycle are called ***amicable***, e.g. 220 and 284, and if the length of the cycle is larger than two, the involved integers are called ***sociable***. Using the `sigma` function we can write a procedure calculating Aliquot sequences as follows:

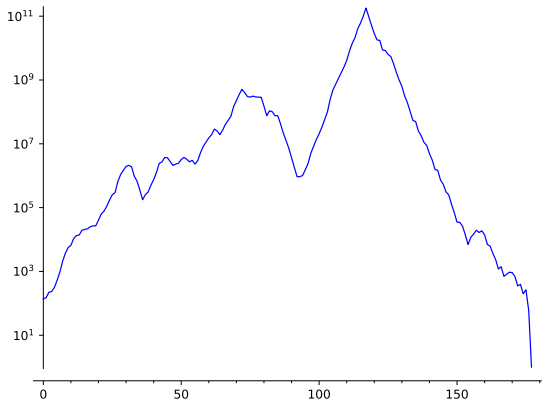
```
sage: def aliquot(n):
....:     L = [n]
....:     while n != 1: #terminates if the last entry equals 1
....:         n = sigma(n) - n
....:         if n in L:  #break if we enter a cycle
....:             L.append(n)
....:             break
....:         L.append(n)
....:     return L
```

In this procedure, the sequence terminates either if $s_k = 1$ or if the sequence has entered a cycle.

```
sage: aliquot(30)
[30, 42, 54, 66, 78, 90, 144, 259, 45, 33, 15, 9, 4, 3, 1]
sage: aliquot(220)
[220, 284, 220]
```

The length of the Aliquot sequence of s_0 can be larger than s_0 itself and can have large peaks, e.g. if we choose $s_0 = 138$.

```
sage: l = aliquot(138)
sage: l[0:5], l[-5:], len(l)
([138, 150, 222, 234, 312], [394, 200, 265, 59, 1], 178)
sage: P = [(x, l[x]) for x in srange(0, len(l))]
sage: p = line(P)
```



It is still an open problem in mathematics whether any Aliquot sequences either terminates or enters a cycle. The ***Catalan-Dickson conjecture*** claims that this should be the case. However, there are numbers whose Aliquot might be infinite yet never repeats. The first five candidate, whose complete Aliquot sequence is still unknown, are 276, 553, 564, 660 and 966. These numbers are also called the ***Lehmer five***.

8.5 Primality Test

To do symbolic calculations, a computer algebra system has to continuously check whether an integer is prime or not. For example, the factorization of a polynomial with integer coefficients starts by factoring it in $\mathbb{F}_p[x]$ for some prime number p, e.g. using the Cantor-Zassenhaus algorithm, see [4, Section 21.3]. In particular, a suitable prime number has to be found.

There are two main classes of primality tests . The most efficient ones are the ***pseudo-primality tests***. If a pseudo-primality test returns `False` then the number is certainly not prime. But if the test result is `True` then no definite conclusion is possible, i.e. the number can be either prime or not. The pseudo-primality test `is_pseudoprime` used in Sage is based on Fermat's little theorem.

Theorem 8.5.1 If p is prime then every integer $0 < a < p$ is an element of the multiplicative group $(\mathbb{Z}/p\mathbb{Z})^*$, i.e. $a^{p-1} \equiv 1 \mod p$.

Hence, if $a^{p-1} \not\equiv 1 \mod p$, then p is certainly not prime. But if this is not the case, then p we can not say for sure whether p is prime or not.

The second class of primality tests consists of ***true primality tests***. The true primality test `is_prime` always returns a correct answer, but it is in general less efficient than the pseudo-primality test `is_pseudoprime`.

```
sage: p = previous_prime(2^600)
```

```
sage: %timeit is_pseudoprime(p)
100 loops, best of 5: 2.7 ms per loop
sage: %timeit is_prime(p)
1 loop, best of 5: 2.33 s per loop
```

Sage provides various commands to look for prime numbers:

- `previous_prime(m)` returns the larges prime number $p < m$,
- `next_prime(m)` returns the smallest prime number $m < p$,
- `prime_range(m)` constructs a list of all prime numbers up to m.

9. Polynomial Rings

In Section **??** we have dealt with polynomials as symbolic expressions in the symbolic ring. Most of the introduced methods are also available when calculating in a polynomial ring, but their implementation often differ, because Sage takes the properties of the considered polynomial ring like $\mathbb{Q}[x]$ or $\mathbb{Z}/4\mathbb{Z}[x]$ into account. One of the main differences between the symbolic ring and the polynomial ring is that polynomials as elements in the polynomial ring are always put into normal form while polynomials in the symbolic ring just remain there appearance.

```
sage: x = var('x'); p = (x^4 - 1)*(2*x + 1)*(x + 2)
sage: parent(p)
Symbolic Ring
sage: print("␣{}␣is␣of␣degree␣{}".format(p, p.degree(x)))
(x^4 - 1)*(2*x + 1)*(x + 2) is of degree 6
sage: x = polygen(QQ, 'x'); p = (2*x+1) * (x+2) * (x^4-1)
sage: parent(p)
Univariate Polynomial Ring in x over Rational Field
sage: print("␣{}␣is␣of␣degree␣{}".format(p, p.degree()))
2*x^6 + 5*x^5 + 2*x^4 - 2*x^2 - 5*x - 2 is of degree 6
```

Here, the line `x = polygen(QQ, 'x')` translates to "assign the `Python` variable `x` to the indeterminate of the polynomial ring in x with rational coefficients". We need to be aware that this assignment is not equivalent to `x = var('x')`, where the `Python` variable `x` is assigned to the *symbolic* variable x.

The available methods and functions on polynomials in a specified polynomial ring are much wider and more efficient than those on polynomials as symbolic expressions. Polynomials in Sage, like many other algebraic objects, in general have coefficients in a commutative ring or in a field. To distinguish them we henceforth denote a commutative

ring by A and a field by K.

Before we can perform calculations in a polynomial ring R, we have to build R itself in Sage. For example, the polynomial ring $\mathbb{Q}[x]$ is build as follows:

```
sage: R = PolynomialRing(QQ, 'x')
sage: x = R.gen()
```

In the first line, `'x'` is simply a character string which represents the name of the indeterminate, or the ***generator***, of the ring. This generator `'x'` is then assigned to the Python variable `x` in the second line. Now, the Python variable `x` represents the polynomial $x \in \mathbb{Q}[x]$. Its parent is therefore the ring `QQ[x]`:

```
sage: x.parent()
Univariate Polynomial Ring in x over Rational Field
```

The command `x = polygen(QQ, 'x')` is equivalent to `x = PolynomialRing(QQ, 'x').gen()` and simultaneously builds the ring $\mathbb{Q}[x]$ and assigns the indeterminate x to the Python variable x. However, the polynomial ring itself is not assigned to a Python variable `x`. Instead of the command `PolynomialRing(QQ, 'x')` we can use the command `QQ[x]` which is closer to the mathematical syntax. This abbreviation is often combined with the construction `R.<x> = ...` which simultaneously assigns the structure $\mathbb{Q}[x]$ to the Python variable `R` and its generator x to the Python variable `x`.

```
sage: R.<x> = QQ['x']
sage: R
Univariate Polynomial Ring in x over Rational Field
sage: x, x.parent()
(x, Univariate Polynomial Ring in x over Rational Field)
```

The base ring, i.e. the ring of the coefficients, can be accessed with `base_ring`.

```
sage: R.base_ring()
Rational Field
```

R *Similar to the assignment of symbolic variables, the name of the Python variable and the name of the generator of the polynomial ring can differ. However, this is not recommended as the code can get quite confusing.*

```
sage: x = polygen(QQ, 'y'); y = polygen(QQ, 'x')
sage: x^2 + 1
y^2 + 1
sage: (y^2 + 1).parent()
Univariate Polynomial Ring in x over Rational Field
```

After the construction of a polynomial ring `R` in Sage all calculations are carried out in `R` as soon as one of its element is involved.

```
sage: R.<x> = QQ['x']
sage: p = x + 2; p
sage: p.parent()
```

Basic Commands for Polynomial Rings $R = A[x]$	
Construction	`R.<x> = A[]`
Accessing the base ring A	`R.base_ring()`
Accessing the indeterminate x	`R.gen()`

Table 9.1.: Polynomial Rings

```
Univariate Polynomial Ring in x over Rational Field
```

Similar to symbolic expressions we can substitute the indeterminate x in a polynomial p with a specific value or another polynomial q. In the second case, Sage calculates the composition $p \circ q$.

```
sage: p = R.random_element(degree = 4); p #a random element in QQ
    ['x'] of degree 4
sage: p.subs(matrix([[1, 2], [3, 4]])) #replace x by a matrix
[305/6 541/3]
[541/2 964/3]
sage: p.subs(x^2) #replace x by the polynomial x^2
x^8 - 2*x^6 - 1/6*x^4 - 73
```

There are also other modifications for polynomials available in Sage. For example, the method `map_coefficients()` takes a function f and applies to all non-zero coefficients of the considered polynomial.

```
sage: p.map_coefficients(lambda a: a^2) #square all coefficients
x^4 + 4*x^3 + 1/36*x^2 + 5329
```

Since any polynomial can be interpreted as a function we can also calculate its derivative using the same methods as for symbolic expressions, see Section 3.3.4.

```
sage: p.derivative(), p.diff()
(4*x^3 - 6*x^2 - 1/3*x, 4*x^3 - 6*x^2 - 1/3*x)
```

Instead of defining a polynomial in the usual way, we can also transform a list `L = [a0, a1, ..., an]` into the polynomial $a_n x^n + \ldots + a_1 x + a_0$ in the polynomial ring `R` using the usual conversion command `R[L]`.

```
sage: p = R([0, 2, 0, 4, 0, 6]); p
6*x^5 + 3*x^3 + 2*x
```

Vice versa, the list of coefficients can be recovered with `list()`. Accordingly the coefficient of x^k is accessed with `p[k]` and `leading_coefficient()` returns the leading coefficient. The method `coefficients()` only returns a list of all nonzero coefficients. In particular, the numbering of the coefficients does not coincide with the numbering as list elements. In addition, a dictionary "degree $\to$ coefficient" of the non-zero coefficients is obtained with the method `dict()`. Last but not least, Sage provides many methods to test for different properties of a polynomial, e.g. `is_monic()`, `is_constant`, ...

```
sage: p.degree()
5
sage: p.list()
[0, 2, 0, 4, 0, 6]
sage: p[1]
2
sage: p.leading_coefficient()
6
sage: p.coefficients()
[2, 4, 6]
sage: p.dict()
{1: 2, 3: 4, 5: 6}
sage: p.is_constant()
False
```

The list of available operations, their meaning, and their efficiency heavily depends on the base ring. For example, we can ask for small roots in `GF(p)['x']` with the method `small_roots` but not in `QQ['x']`. Thus, it might be useful to change the base ring A of the polynomial ring $A[x]$ to the base ring B with the method `change_ring(B)`. This conversion is usually given by a canonical morphism from A to B, e.g. the canonical embedding $\mathbb{Z} \hookrightarrow \mathbb{Q}$ or the projection $\mathbb{Z} \twoheadrightarrow \mathbb{F}_3$. Sometimes a change of the base ring can be used to gain additional algebraic properties. For example, the following polynomial is irreducible over $\mathbb{Q}$ but not over $\mathbb{R}$ and also not over $\mathbb{F}_3$.

```
sage: x = polygen(QQ, 'x')
sage: p = x^2 - 16*x + 3
sage: p.factor()
x^2 - 16*x + 3
sage: p.change_ring(RR).factor()
(x - 15.8102496759067) * (x - 0.189750324093346)
sage: p.change_ring(GF(3)).factor()
x * (x + 2)
```

9.1 Euclidean Arithmetic

Apart from taking sums and products, the most elementary operations on polynomials are the Euclidean division and determing the greatest common divisor (gcd). However, these operations are often hidden by an additional abstract layer which we want to unravel in the following.

9.1.1 Divisibility

The Euclidean division is well-defined if the base ring is a field or, more generally, if the leading coefficient of the divisor is invertible. The used methods have the same name as those we used for symbolic expressions, see Section **??**.

```
sage: R.<t> = IntegerModRing(42)['t']
sage: p = t^20 - 1; d = t^5 + 8*t + 7
```

Accesing Data of a Polynomial p	
Indeterminate x	`p.variables()`
Coefficient of x^k	`p[k]`
Leading Coefficient	`p.leading_coefficient()`
Degree	`p.degree()`
List of Coefficients	`p.list()`
List of non-zero Coefficients	`p.coefficients()`
Dictionary: degree $\mapsto$ coefficients	`p.dict()`
Tests (monic, constant, ...)	`p.is_monic(), p.is_constant(), ...`
Transformations r	
Substitution $x := a$	`p.subs(a)`
Derivative	`p.derivative()`
Transformation of Coefficients	`p.map_coefficients(f)`
Change of Base Ring $A[x] \to B[x]$	`p.change_ring(B)`

Table 9.2.: Transforming Polynomials and Rationals.

```
sage: p.quo_rem(d) #Euclidean divison
(t^15 + 34*t^11 + 35*t^10 + 22*t^7 + 28*t^6 + 7*t^5 + 34*t^3 + 35,
    22*t^4 + 14*t^3 + 14*t + 6)
sage: p // d #exact divison
t^15 + 34*t^11 + 35*t^10 + 22*t^7 + 28*t^6 + 7*t^5 + 34*t^3 + 35
sage: p % d # the remainder
22*t^4 + 14*t^3 + 14*t + 6
```

To perform an exact division we use the operator `//`. Using the usual division operator `/` either raises an error, if the division is not well-defined, or returns an element of the corresponding quotient field. In particular, the parent of the result is different from those of the input.

```
sage: ((t^2 + t) / t).parent()
Traceback (most recent call last):
...
TypeError: self must be an integral domain.
sage: R.<x> = QQ['x']; R
Univariate Polynomial Ring in x over Rational Field
sage: ((x^2 + x) / x).parent()
Fraction Field of Univariate Polynomial Ring in x over Rational
   Field
```

Similar to integers, we can also compute the gcd of two polynomials over a field, but also over some rings, like $\mathbb{Z}$.

```
sage: Z.<x> = ZZ['x']; p = 2 * (x^10 - 1) * (x^8 - 1)
sage: p.gcd(p.diff())
2*x^2 - 2
```

One way to determine the gcd by hand is the Euclidean algorithm . This algorithm determines the gcd of two elements a, b of an Euclidean domain as follows. First, we set $r_0 = b$ and do a euclidean division $a = q_1 r_0 + r_1$. In every further step, we do a further euclidean division until there is no remainder.

$$\begin{aligned} r_0 &= q_2 r_1 + r_2, \\ r_1 &= q_3 r_2 + r_3, \\ &\vdots \\ r_{n-1} &= q_{n+1} r_n + 0. \end{aligned}$$

The last nontrivial remainder is the gcd of a and b, i.e. $\gcd(a,b) = r_n$. It is also possible to calculate the explicit relation $\gcd(p,q) = ap + bq$ with `p.xgcd(q)`. This is the so-called Bézout relation of p and q.

```
sage: R.<x> = QQ['x']; p = x^5 - 1; q = x^3 - 1
sage: p.xgcd(q)
(x - 1, -x, x^3 + 1)
sage: print("The␣gcd␣of␣p␣=␣", p, "␣and␣q␣=␣", q, "␣is␣%s␣=␣(%s)*p
    ␣+␣(%s)*q" % p.xgcd(q))
The gcd of p =  x^5 - 1  and q =  x^3 - 1  is x - 1 = (-x)*p + (x
    ^3 + 1)*q
```

One way to determine the Bézout relation is the extended Euclidean algorithm. We shortly illustrate this algorithm $a = 99$, $b = 78$ viewed as elements of $\mathbb{Z}$. First, we apply the classical Euclidean algorithm:

$$\begin{aligned} 99 &= 1 \cdot 78 + 21, \\ 78 &= 3 \cdot 21 + 15, \\ 21 &= 1 \cdot 15 + 6, \\ 15 &= 2 \cdot 6 + 3, \\ 6 &= 2 \cdot 3 + 0. \end{aligned}$$

Hence, $\gcd(99,78) = 3$. To obtain the Bézout relation, we read these equations backwards:

$$\begin{aligned} 3 &= 15 - 2 \cdot 6 \\ &= 15 - 2 \cdot (21 - 1 \cdot 15) = 3 \cdot 15 - 2 \cdot 21 \\ &= \ldots \qquad\qquad = 3 \cdot 78 - 11 \cdot (99 - 1 \cdot 78) \\ &= 14 \cdot 78 - 11 \cdot 99. \end{aligned}$$

R *Although the method* `xgcd` *is available for polynomials in* `ZZ['x']` *the result is in general no Bézout relation since* $\mathbb{Z}[x]$ *is not a principal ring. Nevertheless,* $ap + bq$ *is still an integer multiple of the gcd.*

```
sage: R.<x> = ZZ['x']
sage: p = -x^2 - x - 1; q = -x^2 - x - 3
sage: B = p.xgcd(q)
```

Divisibility and Euclidean Division	
Divisibility Test $p\|q$	`p.divides(q)`
Euclidean division $p = qd + r$	`q, r = p.quo_rem(d)` or `q = p // d, r = p % d`
Greatest Common Divisor (gcd)	`p.gcd(q), gcd([p1, ... , pn])`
Least Common Multiple	`p.lcm(q), lcm([p1, ..., pn])`
Extended gcd $g = up + vq$	`g, u, v = p.xgcd(q)`

Table 9.3.: Division in Polynomial Rings

```
sage: B[1]*q + B[2]*q
4
sage: p.gcd(q)
1
```

We summarized the introduced methods together with a few further useful methods in Table 9.3.

■ **Example 9.1** Usually, Sage represents polynomial in $\mathbb{Q}[x]$ in the monomial basis $(x^n)_n$. Another common basis is given by the family of Chebyshev polynomials $(T_n)_n$ which are defined by the relation $T_n(\cos(\theta)) = \cos(n\theta)$. The first Chebyshev polynomials are given by

```
sage: R.<x> = QQ['x']; [chebyshev_T(n, x) for n in [0..4]]
```

In the procedure below we use the Euclidean division to determine the coefficients of a polynomial $p \in \mathbb{Q}[x]$ with respect to the basis $(T_n)_n$.

```
sage: def cheb_coeff(p):
....:     L = []
....:     for k in srange(p.degree(), 0, -1, include_endpoint =
    True ):
....:         q, r = p.quo_rem(chebyshev_T(k, x))
....:         L.append(q)
....:         p = r
....:     L.reverse()
....:     return L
sage: p = R.random_element(degree = 6)
sage: p
1/3*x^6 - 5*x^5 - x^4 + 3*x^2 - x
sage: cheb_coeff(p)
[59/48, -33/8, 37/32, -25/16, -1/16, -5/16, 1/96]
```

■

9.1.2 Ideals and Quotients

Ideals of polynomial rings and quotients by these ideals are represented in Sage by objects built from polynomial rings by the methods `ideal` and `quo`. An ideal of a

polynomial ring R is written in Sage either as the product of a tuple of polynomials and the polynomial ring R or constructed with the method R.ideal(p), where p is a collection of polynomials defining the ideal. Sage automatically reduces the collections of defining polynomial to its minimum. Since $K[x]$ are principal rings, each ideal in $K[x]$ is generated by one polynomial.

```
sage: R.<x> = QQ['x']
sage: J1 = (x^2 - 2*x + 1, 2*x^2 + x - 3)*R; J1
Principal ideal (x - 1) of Univariate Polynomial Ring in x over
   Rational Field
sage: (x^2 - 2*x + 1) / (x - 1) #generators are multiple of (x-1)
x - 1
sage: (2*x^2 + x - 3) / (x - 1)
2*x + 3
```

Apart basic arithmetic like sums and product, we can reduce a polynomial modulo an idea. In this case, the reduced polynomial remains an element of QQ['x'].

```
sage: J2 = R.ideal(x^5 + 2); J2
Principal ideal (x^5 + 2) of Univariate Polynomial Ring in x over
   Rational Field
sage: p = ((3*x+5) * J1 * J2).reduce(x^10); p
421/81*x^6 - 502/81*x^5 + 842/81*x - 680/81
sage: p.parent()
Univariate Polynomial Ring in x over Rational Field
sage: ((3*x+5) * J1 * J2).reduce(x^10 - p) #Difference is an
   element of (3*x+5) * J1 * J2
0
```

Another way to obtain the reduction of p modulo an ideal J in QQ['x'] is to first construct the quotient of QQ['x'] by the ideal J.

```
sage: B = R.quo((3*x+5) * J1 * J2); B
Univariate Quotient Polynomial Ring in xbar over Rational Field
   with modulus x^7 + 2/3*x^6 - 5/3*x^5 + 2*x^2 + 4/3*x - 10/3
```

We observe that Sage introduces the variable xbar as indeterminate of the quotient ring B. However, the new introduced variable xbar is not automatically assigned to the Python variable xbar. If we need this indeterminate we have to do this assignment manually.

```
sage: xbar
Traceback (most recent call last):
...
NameError: name 'xbar' is not defined
sage: xbar = B.gen()
sage: xbar, xbar.parent()
(xbar,
 Univariate Quotient Polynomial Ring in xbar over Rational Field
```

Ideals and Quotient Rings $Q = R/J$	
Construction of an Ideal (p,q)	`R.ideal(p, q)` or `(p, q) * R`
Reduction of p modulo an Ideal J	`J.reduce(p)`
Construction of the Quotient Ring R/J	`R.quo(J)`
Access the Cover Ring of a Quotient Q	`Q.cover_ring()`
Lift u from R/J to R	`u.lift()`

Table 9.4.: Ideals and Quotients

```
    with modulus x^7 + 2/3*x^6 - 5/3*x^5 + 2*x^2 + 4/3*x - 10/3)
```

Now we can project the element x^10 to `B`. The parent of the projected element is then `B`. To change the parent back to the polynomial ring, we have to lift it, i.e. use the `lift` method.

```
sage: B(x^10) #x^10 projected down to the quotient ring B
421/81*xbar^6 - 502/81*xbar^5 + 842/81*xbar - 680/81
sage: B(x^10).parent()
Univariate Quotient Polynomial Ring in xbar over Rational Field
   with modulus x^7 + 2/3*x^6 - 5/3*x^5 + 2*x^2 + 4/3*x - 10/3
sage: B(x^10).lift() #lift the element back
421/81*x^6 - 502/81*x^5 + 842/81*x - 680/81
sage: B(x^10).lift().parent()
Univariate Polynomial Ring in x over Rational Field
```

As quotient rings by ideals are again commutative rings, we can use them again as a coefficient domain for a new polynomial ring, e.g. we can construct $(\mathbb{F}_5[t]/\langle t^2+3\rangle)[x]$:

```
sage: R.<t> = GF(5)['t']
sage: p = R.quo(t^2 + 3)['x'].random_element()
sage: p
0
sage: p.parent()
Univariate Polynomial Ring in x over Univariate Quotient
   Polynomial Ring in tbar over Finite Field of size 5 with
   modulus t^2 + 3
```

9.2 Factorization and Roots

A further elementary yet important operation on polynomials is the decomposition of a polynomial into a product of irreducible factors. We have already seen in some examples that the factorization of a polynomial depends on the polynomial ring it belongs to, e.g. see the discussion in Section 8.4.1. Although the factorization of a polynomials is a quite expensive operations there are many advantages like a simpler determination of roots and greatest common divisors.

9.2.1 Factorization

To decompose a polynomial into a product of irreducible polynomials, we have know what an irreducible polynomial is. The answer to this question depends on the base ring. In Sage we can use the method `is_irreducible` to check whether a polynomial is irreducible or not. For example, the polynomial $3x^2 - 6$ is irreducible over $\mathbb{Q}[x]$ but not over $\mathbb{Z}[x]$.

```
sage: R.<x> = QQ['x']; p = 3*x^2 - 6
sage: p.is_irreducible()
True
sage: p.change_ring(ZZ).is_irreducible()
False
sage: p.change_ring(ZZ).factor()
3 * (x^2 - 2)
```

The irreducibility test and the factorization are performed on the base ring. For example, the factorization of a polynomial over the integers contains a constant part, which itself splits into prime factors, and a product of irreducible polynomials.

```
sage: p = 54*x^4 + 36*x^3 - 102*x^2 - 72*x - 12
sage: for A in [ZZ, QQ, ComplexField(16), GF(5), QQ[sqrt(2)]]:
....:     print(str(A) + ":")
....:     print(A['x'](p).factor())
Integer Ring:
2 * 3 * (3*x + 1)^2 * (x^2 - 2)
Rational Field:
(54) * (x + 1/3)^2 * (x^2 - 2)
Complex Field with 16 bits of precision:
(54.00) * (x - 1.414) * (x + 0.3333)^2 * (x + 1.414)
Finite Field of size 5:
(4) * (x + 2)^2 * (x^2 + 3)
Number Field in sqrt2 with defining polynomial x^2 - 2 with sqrt2
   = 1.414213562373095?:
(54) * (x - sqrt2) * (x + sqrt2) * (x + 1/3)^2
```

Since an element of a polynomial ring is always displayed in the normal form $a_k x^k + \ldots + a_1 x + a_0$ the parent of a factorized polynomial f is not the polynomial ring but the class `Factorization`. This class provides similar methods as the corresponding polynomial ring, e.g. the methods `gcm` and `lcm`. To isolate a factor of a factorized polynomial f we use the same syntax as for list elements, i.e. the i-th factor is accessed with `f[i]`.

```
sage: f = QQ['x'](p).factor(); f
(54) * (x + 1/3)^2 * (x^2 - 2)
sage: parent(f)
<class 'sage.structure.factorization.Factorization'>
sage: f[0], f[1]
((x + 1/3, 2), (x^2 - 2, 1))
```

Factorizing a polynomial is a computationally expensive operation. Especially for huge

polynomials one might has to wait quite a long time for the result.

A weaker but way more efficient factorization is the ***square-free decomposition***. There a polynomial p is split into a product $p = \prod_{i=1}^{r} f_i^{m_i}$ of square-free polynomials f_i. Recall, that a polynomial f is ***square-free*** if its factorization only contains factors with multiplicity one, i.e. $f = q_1 \cdots q_k$ with q_i being irreducible and $q_i \neq q_j$ whenever $i \neq j$. This decomposition is calculated with the method `squarefree_decomposition`.

```
sage: QQ['x'](p).squarefree_decomposition()
(54*x^2 - 108) * (x + 1/3)^2
```

Since the square-free part $f_1 \cdots f_r$ of a polynomial $p = \prod_{i=1}^{r} f_i^{m_i}$ has, up to multiplicity, the same roots as p the square-free decomposition of a polynomial is for many applications sufficient.

9.2.2 Roots

There are various ways to determine the roots of a polynomial depending on the explicit situation. Do we want complex or real roots? In which domain do we want to calculate them? Do we need them exact or is a numerical approximation sufficient? With or without multiplicities? The method `roots` returns the roots of the considered polynomials in its base ring as a list of pairs consisting of the root and its multiplicity. In particular, the result of `roots` depends on the base ring.

```
sage: p = (2*x^2 - 5*x + 2)^2 * (x^4 - 7)
sage: for A in [ZZ, QQ, ComplexField(16), GF(5), QQ[7^(1/4)]]:
....:     print(str(A) + ":")
....:     print(A['x'](p).roots())
Integer Ring:
[(2, 2)]
Rational Field:
[(2, 2), (1/2, 2)]
Complex Field with 16 bits of precision:
[(-1.627, 1), (0.5000, 2), (1.627, 1), (2.000, 2), (-1.627*I, 1),
   (1.627*I, 1)]
Finite Field of size 5:
[(3, 2), (2, 2)]
Number Field in a with defining polynomial x^4 - 7 with a =
   1.626576561697786?:
[(a, 1), (-a, 1), (2, 2), (1/2, 2)]
```

Instead of changing the base ring we can also add the desired domain for the roots as an additional argument in `roots`. For example, we can obtain the rational roots of a polynomial in $\mathbb{Z}[x]$ without changing the base ring as follows:

```
sage: p = ZZ['x'](p)
sage: p.roots(QQ), p.parent()
([(2, 2), (1/2, 2)], Univariate Polynomial Ring in x over Integer
   Ring)
```

Since every root of a polynomial with rational coefficients is an algebraic we can use the

domain QQbar of algebraic numbers to compute the exact roots of p.

```
sage: roots = p.roots(QQbar); roots
[(-1.626576561697786?, 1),
 (0.50000000000000000?, 2),
 (1.626576561697786?, 1),
 (2, 2),
 (-1.626576561697786?*I, 1),
 (1.626576561697786?*I, 1)]
```

Despite their appearance, the results are exact and not just approximated values. For example, calculating the fourth power of the first root is *exactly* 7 and not just approximately. In particular, these roots can be reused in further exact calculations.

```
sage: a = roots[0][0]^4; a, parent(a)
(7.000000000000000?, Algebraic Field)
sage: a.simplify(); a
7
```

There are more possibilities to calculate roots numerically using approximation methods, e.g. in the domains RR and CC. There it is often useful to first isolate the root, i.e. determine an interval consisting exactly one root. The specific methods real_roots and complex_roots offer additional options or give slightly different results from the roots method.

9.3 Rational Functions

A ***rational function*** is a quotient of two polynomials of a polynomial ring R. The parent of a rational function is the fraction field of the polynomial ring R. In Sage the fraction field of a polynomial ring R is build with Frac(R).

```
sage: R.<x> = RR['x']
sage: Frac(R)
Fraction Field of Univariate Polynomial Ring in x over Real Field
   with 53 bits of precision
sage: p = 1 + x; q = 1 - x^2; r = p / q
sage: r, r.parent()
((x + 1.00000000000000)/(-x^2 + 1.00000000000000), Fraction Field
   of Univariate Polynomial Ring in x over Real Field with 53
   bits of precision)
```

Since RR is an inexact ring, as its elements are approximations of mathematical objects, the simplification is not done automatically. Here, we have to use the reduce method to put the fraction in reduced form.

```
sage: r.reduce(); r
-1.00000000000000/(x - 1.00000000000000)
```

In exact rings, like ZZ or QQ, rational functions are automatically reduced.

The available operations on rational functions are similar to those on polynomials,

e.g. substitution, derivative and factorization. In addition, we can isolate the numerator and the denominator to work with them separately.

```
sage: r.numerator()
-1.00000000000000
sage: r.denominator()
x - 1.00000000000000
```

We have already seen that we can calculate the partial fraction decomposition of a rational function $r = \frac{p}{q}$ if considered as a symbolic expression, see Section 3.2.1. The corresponding method `partial_fraction_decomposition()` is also available for rational functions in a fraction field. The result contains a polynomial part and a list of rational functions whose denominators are the irreducible factors of the denominator q.

```
sage: R.<x> = QQ['x']; r = x^10 / ((x^2 - 1)^2 * (x^2 + 3))
sage: r.partial_fraction_decomposition()
(x^4 - x^2 + 6, [17/32/(x - 1), 1/16/(x^2 - 2*x + 1), -17/32/(x +
   1), 1/16/(x^2 + 2*x + 1), -243/16/(x^2 + 3)])
```

The above result tells us that the partial fraction decomposition of r is given by

$$r = \frac{x^{10}}{(x^2-1)^2(x^2+3)} = x^4 - x^2 + 6 + \frac{\frac{17}{32}}{x-1} + \frac{\frac{1}{16}}{(x-1)^2} - \frac{\frac{17}{32}}{x+1} + \frac{\frac{1}{16}}{(x+1)^2} - \frac{\frac{243}{16}}{x^2+3}$$

In $\mathbb{C}$ the denominator of the last term is not irreducible. Thus, we can get a finer partial fraction decomposition by changing the base ring to the complex numerical `CC` if we want an approximate decomposition, or to $\mathbb{Q}[i]$ or `QQbar`, if we want an exact decomposition.

```
sage: C = ComplexField(16)
sage: Frac(C['x'])(r).partial_fraction_decomposition()
(x^4 - x^2 + 6.000, [0.5312/(x - 1.000), 0.06250/(x^2 - 2.000*x +
   1.000), 4.384*I/(x - 1.732*I), (-4.384*I)/(x + 1.732*I),
   (-0.5312)/(x + 1.000), 0.06250/(x^2 + 2.000*x + 1.000)])
sage: Frac(QQbar['x'])(r).partial_fraction_decomposition()
(x^4 - x^2 + 6, [4.384253606658720?*I/(x - 1.732050807568878?*I),
   (-4.384253606658720?*I)/(x + 1.732050807568878?*I),
   0.53125000000000000?/(x - 1), 0.062500000000000000?/(x^2 - 2*x
    + 1), (-0.53125000000000000?)/(x + 1),
   0.062500000000000000?/(x^2 + 2*x + 1)])
```

As for integers, the rational reconstruction also exists for polynomials with coefficients in a commutative ring like $A = \mathbb{Z}/n\mathbb{Z}$. Given two polynomials $m, s \in A[x]$ and two integers d_p, d_q the command `s.rational_reconstruct(m, dp, dq)` computes, if possible, two polynomial $p, q \in A[x]$ such that

$$q \cdot s \equiv p \bmod m, \quad \deg p \leq d_p, \quad \deg q \leq d_q.$$

If m is a prime number a nontrivial solution always exists if the degrees of p and q satisfy the relation $d_p + d_q \geq \deg m - 1$.

Rational Functions r	
Fraction Field of R	`Frac(R)`
Numerator	`r.numerator()`
Denominator	`r.denominator()`
Simplification	`r.reduce()`
Partial Fraction Decomposition	`r.partial_fraction_decomposition()`
Rational Reconstruction of $s \mod m$	`s.rational_reconstruct(m)`

Table 9.5.: Rational Functions

```
sage: A = IntegerModRing(5); x = polygen(A)
sage: m = (1 + x) * (x^2 - 3); s = (x + 1) * (x^3 + 4*x - 2)
sage: m, s
(x^3 + x^2 + 2*x + 2, x^4 + x^3 + 4*x^2 + 2*x + 3)
sage: p, q = s.rational_reconstruct(m, 1, 3)
sage: p, q
(4*x + 4, x + 1)
sage: ((q*s) % m ) == (p % m) #Test
True
```

9.4 Formal Power Series

A formal power series of indeterminate x with coefficients in a commutative ring A is a formal sum $\sum_{n=0}^{\infty} a_n x^n$, where $(a_n)_n$ is a sequence of elements in A, without considering convergence. Together with the natural addition and multiplication operations,

$$\sum_{n=0}^{\infty} a_n x^n + \sum_{n=0}^{\infty} b_n x^n = \sum_{n=0}^{\infty} (a_n + b_n) x^n,$$
$$\left(\sum_{n=0}^{\infty} a_n x^n\right) \cdot \left(\sum_{n=0}^{\infty} b_n x^n\right) = \sum_{n=0}^{\infty} \left(\sum_{i+j=n} a_i b_j\right) x^n,$$

the formal power series form the ring $A[[x]]$.

In computer algebra systems, these series are useful to approximate analytic functions for which we have no closed form. However, since the computer only performs the calculations, we have to ensure ourselves that the considered sequences converge.

9.4.1 Operations on Truncated Power Series

The ring $\mathbb{Q}[[x]]$ of formal power series with rational coefficients is constructed by

```
sage: R.<x> = PowerSeriesRing(QQ); R
Power Series Ring in x over Rational Field
```

or equivalently by its alias `R.<x> = QQ[['x']]`, following more the mathematical syntax. As a computer can only deal with finite objects, the elements of `A[['x']]` are

truncated power series, i.e. they are of the form

$$a_0 + a_1 x + \ldots + a_{n-1} x^{n-1} + O(x^n).$$

In particular, they are only approximations of the corresponding infinite sequences, i.e. the ring `A[['x']]` is always an inexact ring, even if its base ring A is exact. Each element of `A[['x']]` has its own truncation order which is automatically carried over through computations.

```
sage: f = 1 + x + O(x^2); g = x + 2*x^2 + O(x^4)
sage: f, g
(1 + x + O(x^2), x + 2*x^2 + O(x^4))
sage: f + g
1 + 2*x + O(x^2)
sage: f * g
x + 3*x^2 + O(x^3)
```

The polynomial ring $A[x]$ can be interpreted as a subset of $A[[x]]$. In fact, polynomials are those formal power series with infinite precision.

```
sage: (1 + x^3).prec()
+ Infinity
```

A default precision is used, when it is necessary to truncate an exact result. This precision can be modified either at the ring creation with the option `default_prec`, or afterwards with the method `set_default_prec`.

```
sage: R.<x> = PowerSeriesRing(Reals(24), default_prec = 4); R
Power Series Ring in x over Real Field with 24 bits of precision
sage: 1/ (1 + numerical_approx(pi)*x)^2
1.00000 - 6.28319*x + 29.6088*x^2 - 124.025*x^3 + O(x^4)
sage: R.set_default_prec(8)
sage: 1/ (1 + numerical_approx(pi)*x)^2
1.00000 - 6.28319*x + 29.6088*x^2 - 124.025*x^3 + 487.046*x^4 -
   1836.12*x^5 + 6729.73*x^6 - 24162.4*x^7 + O(x^8)
```

A consequence of the truncation is that we can not test whether two formal power series are mathematically equivalent. Indeed, two elements in `A[['x']]` are considered as equal as soon as they match up to the smallest of their precision. In particular, the test `O(x^2) == 0` returns `True` although $0 \subsetneqq O(x^2)$.

```
sage: R.<x> = QQ[['x']]
sage: 1 + x + O(x^2) == 1 + x + x^2 + O(x^3)
True
sage: O(x^2) == 0
True
```

All basic arithmetic operations on series works as for polynomials. Further available methods include `f.exp()`, when $f(0) = 0$, derivatives and antiderivative methods.

```
sage: p = x + 3*x^2 - 2 * x^3 + O(x^4)
```

Truncated Power Series	
Construction of $A[[x]]$	`PowerSeriesRing(A, 'x', default_prec = n)`
Coefficient of x^k in f	`f[k]`
Truncation	`x + O(x ^n)`
Precision	`f.prec()`
Useful Operations $\sqrt{f}, \exp(f), \ldots$	`f.sqrt(), f.exp(), ...`

Table 9.6.: Truncated Power Series

```
sage: p.exp()
1 + x + 7/2*x^2 + 7/6*x^3 + O(x^4)
sage: p.derivative()
1 + 6*x - 6*x^2 + O(x^3)
sage: p.integral()
1/2*x^2 + x^3 - 1/2*x^4 + O(x^5)
```

■ **Example 9.2** We can use formal power series to compute an asymptotic expansion of $\frac{1}{x^2}\exp\left(\int_0^x \sqrt{\frac{1}{1+t}}dt\right)$ as $x \to 0$ up to order 5 as follows:

```
sage: R.<x> = QQ[['x']]
sage: (1/(1+x)).sqrt().integral().exp() / x^2 + O(x^6)
x^-2 + x^-1 + 1/4 + 1/24*x - 1/192*x^2 + 11/1920*x^3 - 103/23040*x
    ^4 + 1177/322560*x^5 + O(x^6)
```

■

The above example also demonstrates that for two elements $f, g \in K[[x]]$ (recall that K stands for a field) with $g(0) = 0$ the quotient $\frac{f}{g}$ is a ***formal Laurent series***. In contrast to Laurent series in complex analysis, formal Lauren series only have a finite number of term of negative exponent. The restriction is mandatory for the product of two formal sums to be well-defined, as otherwise the coefficients of a product would be infinite series on their own.

9.4.2 Applications involving Power Series

Padé Approximation

A Padé approximation of type $(k, n-k)$ of a formal power series $f \in K[[x]]$ is a rational function $\frac{p}{q}$ such that

- $\deg p \leq k-1$,
- $\deg q \leq n-k$,
- $q(0) = 1$,
- $\frac{p}{q} = f + O(x^n)$, i.e. $\frac{p}{q} \equiv f \mod x^n$.

A Padé approximation of type $(2,4)$ of the series $f = \sum_{i=0}^{\infty}(i+1)^2 x^i$ with coefficients in $\mathbb{Z}/101\mathbb{Z}$ can be easily calculated with rational reconstruction.

```
sage: A = IntegerModRing(101); R.<x> = A['x']; R
Univariate Polynomial Ring in x over Ring of integers modulo 101
sage: f6 = sum((i+1)^2 * x^i for i in (0..5)); f6
36*x^5 + 25*x^4 + 16*x^3 + 9*x^2 + 4*x + 1
sage: num, den = f6.rational_reconstruct(x^6, 2, 4); num/den
(100*x + 100)/(x^3 + 98*x^2 + 3*x + 100)
```

To test our result, we expand the quotient back to a power series.

```
sage: S.<x> = A[['x']]; S
Power Series Ring in x over Ring of integers modulo 101
sage: S(num) / S(def) == f6 + O(x^6)
True
```

R *Using Taylor series, we can also construct Padé approximations of functions.*

Finding Fix Points

Another application of power series is to solve fixed-point equations like $e^{xf(x)} = f(x)$. This procedure is based on the famous Banach fixed-point theorem:

Theorem 9.4.1 Let M be a closed subspace of a complete metric space (X,d) and let $\Phi : M \to X$ be a contraction, i.e. there exists a $k \in [0,1)$ such that

$$d(\Phi(x),\Phi(y)) \leq k \cdot d(x,y)$$

for all $x,y \in M$. Then for any $x_0 \in M$ the recursively defined sequence $(x_n)_n$ with

$$x_n = \Phi(x_{n-1}) = \Phi^n(x_0)$$

converges to the unique fix point $\tilde{x} \in M$, i.e. $\tilde{x} = \Phi(\tilde{x})$.

Since the solution of $e^{xf(x)} = f(x)$ in $\mathbb{Q}[[x]]$ is the fix point of the transformation $\Phi : f \mapsto e^{xf}$ we want to apply Banach fixed-point theorem. First, we endow $\mathbb{Q}[[x]]$ with the metric

$$\mathrm{d}(\sum_i f_i x^i, \sum_j g_j x^j) = 2^{-k},$$

where $k = \min\{k \in \mathbb{N} \mid f_k \neq g_k\}$. It is not hard to check that $(\mathbb{Q}[[x]],\mathrm{d})$ is a complete metric space and that Φ is a contraction around $1 \in \mathbb{Q}[[x]]$. Hence, by Banach fixed-point theorem the solution $f(x)$ is the limit of the sequence $(\Phi^n(1))_n$ as n tends to infinity.

```
sage: S.<x> = PowerSeriesRing(QQ, default_prec = 6)
sage: f = S(1) #the series 1 is our starting point
sage: for i in srange(6):
....:     f = (x*f).exp() #here the iteration happens
....:     print(f)
1 + x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + 1/120*x^5 + O(x^6)
1 + x + 3/2*x^2 + 5/3*x^3 + 41/24*x^4 + 49/30*x^5 + O(x^6)
1 + x + 3/2*x^2 + 8/3*x^3 + 101/24*x^4 + 63/10*x^5 + O(x^6)
```

```
1 + x + 3/2*x^2 + 8/3*x^3 + 125/24*x^4 + 49/5*x^5 + O(x^6)
1 + x + 3/2*x^2 + 8/3*x^3 + 125/24*x^4 + 54/5*x^5 + O(x^6)
1 + x + 3/2*x^2 + 8/3*x^3 + 125/24*x^4 + 54/5*x^5 + O(x^6)
```

We see that with each iteration process a new stable term appears. To explain this behavior we take a closer look on the equation $e^{xf(x)} = f(x)$. By Banach fixed-point theorem we know that this equation has a solution $f(x) \in \mathbb{Q}[[x]]$. Since f is a power series we can write it as $f(x) = \sum_{n=0}^{\infty} f_n x^n$. In addition, the exponential function can be written as the power series $e^t = \sum_{k=0}^{\infty} \frac{t^k}{k!}$. Inserting these two expansions into the equation $e^{xf(x)} = f(x)$ leads to the relation

$$\sum_{n=0}^{\infty} f_n x^n = \sum_{k=0}^{\infty} \frac{1}{k!} \left(x \sum_{j=0}^{\infty} f_j x^j \right)^k = \sum_{n=0}^{\infty} \left(\frac{1}{k!} \sum_{\substack{j_1,\ldots,j_k \in \mathbb{N} \\ j_1+\ldots+j_k = n-k}} f_{j_1} \cdot f_{j_2} \ldots f_{j_k} \right) x^n. \quad (9.1)$$

In particular, that each coefficient f_n can be calculated from the preceding coefficients $f_0, \ldots, f_{n-1}$. Hence, each iteration of Φ yields a new correct term.

■ **Example 9.3** We use this iteration process to determine the series expansion to order 15 of $\tan(x)$ near zero using the differential equation $\tan' = 1 + \tan^2$. First, we rewrite this as a fix point equation $\tan(x) = \int_0^x 1 + \tan(t)^2 dt$. Since the transformation $f(x) \mapsto \int_0^x 1 + f(t)^2 dt$ is a contraction around $0 = \tan(0)$ we can determine the series expansion of $\tan(x)$ as follows:

```
sage: S.<x> = QQ[[x]]
sage: t = S(0) #Setting the starting value 0
sage: for _ in srange(15): #do the iteration
....:     t = (1 + t^2).integral() + O(x^15) #restrict the maximal
    precision
sage: t
x + 1/3*x^3 + 2/15*x^5 + 17/315*x^7 + 62/2835*x^9 + 1382/155925*x
    ^11 + 21844/6081075*x^13 + O(x^15)
sage: S(tan(x)) == t
True
```

■

9.4.3 Lazy Power Series

There are many power series $\sum_{n=0}^{\infty} a_n x^n$ where the coefficients a_n can be either calculated directly with a function $a(n)$, e.g. $a(n) = \frac{1}{n!}$ for the exponential function $e^x = \sum_{n=0} \infty \frac{x^n}{n!}$ or from the preceding coefficients $a_0, \ldots, a_{n-1}$, as seen in (9.1). Hence, the complete infinite sequence is defined by a finite number of relations. This motivates the definition of so called ***lazy power series***. These are not truncated sequences, but real infinite sequences. The adjective "lazy" means here that the coefficients are computed on demand only. As a counterpart, we can only represent series whose coefficients are computable, e.g. the lazy power series for the exponential is defined by

```
sage: L.<x> = LazyPowerSeriesRing(QQ)
```

```
sage: lazy_exp = x.exponential(); lazy_exp
O(1)
```

A lazy power series is an object which contains in its internal representation all the information needed to compute the series expansion of $\exp(x)$ to any order. The initial output is `O(1)` as we have not asked for any coefficient so far. As soon as we ask for another coefficient, e.g. the coefficient of x^5, the corresponding computations are performed, and the computed coefficients are stored.

```
sage: lazy_exp[5]
1/120
sage: lazy_exp
1 + x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + 1/120*x^5 + O(x^6)
```

As discussed above, the coefficients of the solution of the equation $e^{xf(x)} = f(x)$ are recursively defined, see (9.1). Therefore, we can also solve this equation with lazy power series.

```
sage: f = L(1) #starting value is the LAZY sequence 1 with
   rational coefficients
sage: for i in srange(5):
....:     f = (x*f).exponential()
....:     f.compute_coefficients(5) #forces the commputation
....:     print(f)
1 + x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + 1/120*x^5 + O(x^6)
1 + x + 3/2*x^2 + 5/3*x^3 + 41/24*x^4 + 49/30*x^5 + O(x^6)
1 + x + 3/2*x^2 + 8/3*x^3 + 101/24*x^4 + 63/10*x^5 + O(x^6)
1 + x + 3/2*x^2 + 8/3*x^3 + 125/24*x^4 + 49/5*x^5 + O(x^6)
1 + x + 3/2*x^2 + 8/3*x^3 + 125/24*x^4 + 54/5*x^5 + O(x^6)
```

Although the output is the same, the huge difference is, that we can now ask for higher order coefficients, like `f[7]`.

```
sage: f[7]
28673/630
```

If we would try to do this in the classic power series ring `Sage` would have raised an error, because the truncation order of f is strictly smaller than 7. Nevertheless, f is not the real solution but the iterate $\Phi^5(1)$. The real power of lazy series is the possibility to directly get the limit, by defining f itself as a lazy power series. As this a more advanced application of lazy power series we have to import the specific package.

```
sage: from sage.combinat.species.series import LazyPowerSeries
sage: f = LazyPowerSeries(L, name = 'f')
sage: f.define((x*f).exponential())
sage: f.coefficients(8)
[1, 1, 3/2, 8/3, 125/24, 54/5, 16807/720, 16384/315]
```

Here, the iterative computation did work due to the recursive relation we have discovered in (9.1). In fact, `Sage` deduces this recursive relation and stores it in the `Python` variable

`f`. This allows `Sage` to store the explicit solution `f` as an infinite series, i.e. we can ask for every coefficient of `f`.

10. Matrices

We have already learned some basic commands for computations with vectors and matrices, in Section 3.3.5. In this chapter, we consider matrices and vectors with coefficients in a specific computation domain, such as $\mathbb{Z}$, finite fields or polynomial rings. In particular, we discuss various normal forms of matrices and the study of eigenvalues and eigenspaces with Sage.

10.1 Constructions and Elementary Manipulations

10.1.1 Matrix Spaces and Groups

Similar to polynomials, vectors and matrices are handled as algebraic objects belonging to a space. Vector spaces are built with the constructor `VectorSpace`. The coefficients of a vector field are, by definition, elements of a field. If the coefficients should be elements of a ring, the analogous algebraic object to a vector space is a free module which can be constructed with `FreeModule`.

```
sage: VS = VectorSpace(GF(5), 3); VS
Vector space of dimension 3 over Finite Field of size 5
sage: FM = FreeModule(IntegerModRing(4), 5); FM
Ambient free module of rank 5 over Ring of integers modulo 4
```

For matrices it suffices if the coefficient domain is a commutative ring. As usual in Python, rows are always named before columns.

```
sage: MS1 = MatrixSpace(ZZ, 2, 3); MS1 #2 rows, 3 columns
Full MatrixSpace of 2 by 3 dense matrices over Integer Ring
sage: MS2 = MatrixSpace(QQ, 4, 2); MS2 #4 rows, 2 columns
Full MatrixSpace of 4 by 2 dense matrices over Rational Field
```

A canonical basis of a matrix space $M_{I,J}(A)$ is given by the matrices $(B_{ij})_{\substack{0\le i\le I-1\\ 0\le j\le J-1}}$, where B_{ij} is the matrix whose (i,j)-th entry is equal to 1 and all other entries are equal to zero. This canonical basis can be obtained with the methods `basis` or `gen`. Here, the output is a dictionary "$(i,j)\mapsto B_{ij}$".

```
sage: B = MS1.basis()
sage: B
Finite family {(0, 0): [1 0 0]
[0 0 0], (0, 1): [0 1 0]
[0 0 0], (0, 2): [0 0 1]
[0 0 0], (1, 0): [0 0 0]
[1 0 0], (1, 1): [0 0 0]
[0 1 0], (1, 2): [0 0 0]
[0 0 1]}
sage: B[1,2]
[0 0 0]
[0 0 1]
```

Similar to polynomials, the method `change_ring` allows us to change the coefficient domain of a given matrix space.

```
sage: MS1 = MatrixSpace(ZZ, 2, 3); MS1 #2 rows, 3 columns
Full MatrixSpace of 2 by 3 dense matrices over Rational Field
sage: MS1.change_ring(QQ)
Full MatrixSpace of 2 by 3 dense matrices over Rational Field
```

For many algebraic applications we do not need the whole matrix space, but only particular matrix groups. The most common ones are already implemented Sage, see Table **??**.

We want to elaborate a little bit on the orthogonal group $\mathrm{O}_n(R)$ consisting of all $n\times n$-matrices with coefficients in R preserving a symmetric bilinear form. With only one exception, there is, up to isomorphism, only one symmetric bilinear form. In these cases, the orthogonal group is constructed with `GO(n, R)`. However, if n is even and R is a finite field, there are two inequivalent symmetric bilinear forms. In that case, we have to choose one by adding a third parameter $q\in\{\pm 1\}$ in the constructor, i.e. the orthogonal group with parameter q is built with `GO(n, R, q)`. The same holds for the special orthogonal group $\mathrm{SO}_n(R)$.

```
sage: GO(4, GF(5), 1), SO(6, ZZ)
(General Orthogonal Group of degree 4 and form parameter 1 over
   Finite Field of size 5,
Special Orthogonal Group of degree 6 over Integer Ring)
```

The method `invariant_bilinear_form()` returns the symmtric bilinear form which is fixed by the group, i.e. the matrix m such that $gmg^t = m$ holds for all group elements g. Similar, the method `invariant_form()` returns the invariant form fixed by a symplectic group.

```
sage: GO(4, GF(5), 1).invariant_bilinear_form(), GO(4, GF(5), -1).
```

```
   invariant_bilinear_form()
([0 1 0 0]
[1 0 0 0]
[0 0 2 0]
[0 0 0 2], [0 1 0 0]
[1 0 0 0]
[0 0 4 0]
[0 0 0 2])
sage: Sp(4, QQ).invariant_form()
[ 0  0  0  1]
[ 0  0  1  0]
[ 0 -1  0  0]
[-1  0  0  0]
```

Vice versa, we can also construct the (special) orthogonal group with respect to a given symmetric bilinear form *m* using the option `invariant_form = m`. The same can be done for symplectic groups. Here the bilinear form *m* has to be skew-symmetric, i.e. representing a symplectic form.

```
sage: m = matrix(QQ, [[-1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0],
   [0, 0, 0, 1]])
sage: m
[-1  0  0  0]
[ 0  1  0  0]
[ 0  0  1  0]
[ 0  0  0  1]
sage: GO(4, QQ, invariant_form = m) #orthogonal group wrt the
   minkowski metric m
General Orthogonal Group of degree 4 over Rational Field with
   respect to non positive definite symmetric form
[-1  0  0  0]
[ 0  1  0  0]
[ 0  0  1  0]
[ 0  0  0  1]
```

Furthermore, if we have a list of invertible matrices `[A, B, ...]` the constructor `MatrixGroup([A, B, ...])` returns the matrix group generated by these matrices. All of the methods in Table 10.1 also apply to matrix groups.

```
sage: A = matrix(GF(11), 2, 2, [[1,0], [0, 2]]); B = matrix(GF(11)
   , 2, 2, [[0, 1], [1, 0]]); A, B
(
[1 0]  [0 1]
[0 2], [1 0]
)
sage: MG = MatrixGroup([A, B]); MG
Matrix group over Finite Field of size 11 with 2 generators (
[1 0]  [0 1]
```

Matrix Spaces	
Construction of the Matrix Space $\mathscr{M}_{I,J}(A)$	`M = MatrixSpace(A, I, J)`
Canonical Basis of M	`M.basis()` or `M.gen()`
Number of Generators	`M.ngens()`
Number of Elements	`M.order()`
Base Ring	`M.base_ring()`
Changing the Ring	`M.change_ring()`
Matrix Groups	
General Linear Group $\mathrm{GL}_n(R)$	`GL(n, R)`
Special Linear Group $\mathrm{SL}_n(R)$	`SL(n, R)`
Orthogonal Group $\mathrm{O}_n(R)$	`GO(n, R)`
Special Orthogonal Group $\mathrm{SO}_n(R)$	`SO(n, R)`
Symplectic Group $\mathrm{Sp}_n(R)$	`Sp(n, R)`
Matrix Group generated by $A, B, \ldots$	`MatrixGroup([A, B, ...])`

Table 10.1.: Matrix Spaces

```
[0 2], [1 0]
)
sage: MG.order()
200
```

10.1.2 Matrix and Vector Constructions

Matrices and vectors are generated as elements of their space by providing a list of their coefficients. For matrices, the coefficients are listed in a row major mode, i.e. the constructor takes a list of coefficients and fills the rows from left to right starting with the top row.

```
sage: MS = MatrixSpace(ZZ, 3, 3)
sage: A = MS([1, 2, 3, 4, 5, 6, 7, 8, 9]); A
[1 2 3]
[4 5 6]
[7 8 9]
```

Sage provides constructors for the most common matrices.

```
sage: MS() #zero matrix
[0 0 0]
[0 0 0]
[0 0 0]
sage: MS.identity_matrix() #only works for square matrices
[1 0 0]
[0 1 0]
[0 0 1]
sage: MS.random_element()
```

```
[ 1  4 -1]
[-1  0 -1]
[-3  1  0]
sage: jordan_block(3, 4)
[3 1 0 0]
[0 3 1 0]
[0 0 3 1]
[0 0 0 3]
```

Furthermore, it is possible to block matrices from several submatrices. There we can either build the block matrix as described in Section 3.3.5, or hand over a single list of matrices and use the optional arguments `nrows` and `ncols` to determine the size of the block matrix. Whenever it makes sense, a scalar coefficient k is interpreted as the corresponding multiple of the identity matrix.

```
sage: A = matrix([[1, 2], [3, 4]])
sage: block_matrix([[A, -A], [ 2*A, A^2]])
[ 1  2|-1 -2]
[ 3  4|-3 -4]
[-----+-----]
[ 2  4| 7 10]
[ 6  8|15 22]
sage: block_matrix([1, A, 0, 0, -A, 2], ncols = 3)
[ 1  0| 1  2| 0  0]
[ 0  1| 3  4| 0  0]
[-----+-----+-----]
[ 0  0|-1 -2| 2  0]
[ 0  0|-3 -4| 0  2]
```

In the special case of block diagonal matrices, the list of the diagonal blocks can be directly passsed to the constructor `block_diagonal_matrix`.

```
sage: block_diagonal_matrix(A, A.transpose())
[1 2|0 0]
[3 4|0 0]
[---+---]
[0 0|1 3]
[0 0|2 4]
```

The block structure is only a display feature. Sage treats a block matrix just as any other matrix of the corresponding size. This display mode can be disabled by providing the additional argument `subdivide = False`.

10.1.3 Basic Manipulations and Arithmetic

Coefficients and submatrices of a matrix `A` are accessed through the square bracket operator `A[i, j]` where `i` and `j` are the row and the column index respectively. The entries `i` and `j` can be integers or intervals using the same notation as described for accessing list elements, see Section 5.1. The entry `i:j:k` lists all elements between

`i` and `j-1` by steps of `k`. Since matrices are mutable objects, we can modify their coefficients and submatrices directly. After each change replaces the matrix be the modified metric, i.e. the original matrix is lost if not saved otherwise.

```
sage: A = matrix(3, 3, srange(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: A[:, 1] = vector([1, 1, 1]); A
[0 1 2]
[3 1 5]
[6 1 8]
sage: A[::-1] #steps can be negative
[6 1 8]
[3 1 5]
[0 1 2]
sage: A[:, ::-1] #only change columns
[2 1 0]
[5 1 3]
[8 1 6]
sage: A[::2, -1]
[2]
[8]
```

R *By default, matrix objects are mutable. i.e. we can modify their coefficients after construction. If we want to protect a matrix* `A` *against such modifications we can make it immutable via* `A.set_immutable()`. *Afterwards, it is still possible to create mutable copies using the copy function* `copy(A)`.

Using the above syntax it can be quite a tedious task to extract a special submatrix. In particular, if we are only interested in individual rows or columns. To ease this process Sage provides further the methods `matrix_from_rows`, `matrix_from_columns`to extract specified rows or columns, respectively. These two commands are combined in `matrix_from_rows_and_columns`.

```
sage: B = matrix(ZZ, 4, 4, srange(16)); B
[ 0  1  2  3]
[ 4  5  6  7]
[ 8  9 10 11]
[12 13 14 15]
sage: B.matrix_from_rows([2, 3])
[ 8  9 10 11]
[12 13 14 15]
sage: B.matrix_from_columns([0, 3])
[ 0  3]
[ 4  7]
[ 8 11]
[12 15]
```

```
sage: B.matrix_from_rows_and_columns([0, 2, 3], [1, 2])
[ 1  2]
[ 9 10]
[13 14]
```

In addition, the command `submatrix(i, j, m, n` returns the submatrix of size $m \times n$ whose upper left coefficient is at position (i, j).

```
sage: B.submatrix(0, 1, 2, 3)
[1 2 3]
[5 6 7]
```

The methods `rows` and `columns` return the rows and columns as a list, while the methods `nrows` and `ncols` only returns the numbers of the rows ad columns, respectively.

```
sage: B.rows(), B.nrows()
([(0, 1, 2, 3), (4, 5, 6, 7), (8, 9, 10, 11), (12, 13, 14, 15)],
   4)
sage: B.columns(), B.ncols()
([(0, 4, 8, 12), (1, 5, 9, 13), (2, 6, 10, 14), (3, 7, 11, 15)],
   4)
```

The basic arithmetic operations on matrices are done with the usual operators `+`, `-`, `*`, `^`. In computations, a scalar is interpreted as the corresponding multiple of the identity matrix, if the matrix space permits it. The only exception is the product, as the scalar multiplication of a matrix is always well-defined. The methods `transpose` and `conjugate` return the transpose and the conjugate of a matrix respectively.

10.2 Matrix Computations

In linear algebra, matrices are typically used to represent families of vectors, systems of linear equations, linear transformations, or vector subspaces. Consequently, computing properties such as the rank of a family of vectors, the solution to a linear system, the eigenspaces of a linear transformations or the dimension of a subspace all boil down to operations on the corresponding matrices.

Geometrically, these transformations often correspond to the change of the basis of the considered vector spaces. This translates to the equivalence transformation $B = PAQ^{-1}$, where P, Q represent the base changes. In particular, P, Q have to be invertible. Accordingly, two matrices are called ***equivalent***, if such a transformation exists. The resulting equivalence classes of matrices are characterized by normal forms like the reduced echelon form or the Frobenius form.

10.2.1 Gaussian Elimination and Normal Forms

Gaussian elimination is a well-known algorithm in linear algebra which transforms a matrix A into an upper triangular matrix using equivalence transformations. The resulting upper triangular matrix reveals various fundamental properties of the original matrix A such as the rank and the determinant. We shortly describe the elementary row

operations involved in Gaussian elimination and the corresponding methods in Sage.

- `swap_rows(i1, i2)`: Putting two rows $L_{i_1} \leftrightarrow L_{i_2}$,
- `add_multiple_of_row(i1, i2, s)`: Add $s \cdot L_{i_2}$ to the row L_{i_1}, i.e. L_{i_1} is replaced by $L_{i_1} + s \cdot L_{i_2}$.

The corresponding methods for columns are called by the methods `swap_columns(j1, j2)` and `add_multiple_of_column(j1, j2, s)`.

For an $m \times n$ matrix $A = (a_{ij})$ the Gaussian elimination algorithm proceeds iteratively from the leftmost column to the rightmost column. Assuming that the first $k-1$ first columns have already been processed, generating $p \leq k$ pivots, the k-th column is treated as follows:

1. Find the first invertible coefficient a_{ik} in the column C_k in a row $i > p$. This element is called the ***pivot***.
2. If no pivot can be found, move to the next column.
3. Otherwise, swap the lines $L_i \leftrightarrow L_p$.
4. Eliminate all elements below a_{pk} via $L_i \mapsto L_i - \frac{a_{ik}}{a_{pk}} L_p$.

■ **Example 10.1** We transform the matrix

$$A = \begin{pmatrix} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{pmatrix}$$

using the Gaussian elimination algorithm..

```
sage: A = matrix(QQ, [[2, 1, -1, 8], [-3, -1, 2, -11], [-2, 1, 2,
   -3]]); A
[  2   1  -1   8]
[ -3  -1   2 -11]
[ -2   1   2  -3]
```

Here, the entry $a_{00} = 2$ is our first pivot. Thus, we have to eliminate the elements below it, as explained in step 4.

```
sage: A.add_multiple_of_row(1, 0, 3/2); A #L_1 is replaced by L_1
   + 3/2*L_0
[  2   1  -1   8]
[  0 1/2 1/2   1]
[ -2   1   2  -3]
sage: A.add_multiple_of_row(2, 0, 1); A #L_2 is replaced by L_2 +
   L_0
[  2   1  -1   8]
[  0 1/2 1/2   1]
[  0   2   1   5]
```

Our next pivot is $a_{11} = \frac{1}{2}$. After eliminating the element below this pivot, we have found an equivalent upper triangular matrix.

```
sage: A.add_multiple_of_row(2,1,-4); A
[  2   1  -1   8]
[  0 1/2 1/2   1]
[  0   0  -1   1]
```

■

For a given matrix A the ***echelon form*** is an equivalent upper triangular matrix. However, there are various upper triangular matrices equivalent to A. Any echelon form of A can be further transformed to the reduced echelon form.

Definition 10.2.1 A matrix is in ***reduced echelon form*** if

- all zero rows are at the bottom,
- the leading coefficient of every non-zero row, called a pivot, is 1 and it to the right of the pivot of the row above,
- pivots are the only non-zero elements in their column.

One main advantage of the reduced echelon form compared to the echelon form is that every equivalence class of matrices over a field admits a unique reduced echelon form, i.e. the reduced echelon form is a normal form.

Theorem 10.2.1 For every matrix A over a field there is a unique $m \times n$ matrix R in reduced echelon form and an invertible $m \times m$ matrix such that $UA = R$.

In Sage, the reduced echelon form can be obtained in two different ways. The method `echelonize` replaces the input matrix by its reduced echelon form, while the method `echelon_form` returns an immutable matrix in reduced echelon form, without modifying the input matrix.

```
sage: A.echelon_form()
[ 1  0  0  2]
[ 0  1  0  3]
[ 0  0  1 -1]
```

Apart from matrices over fields, it is also quite common to work with matrices over commutative rings, e.g. matrices with integer coefficients. There, the pivot element might be non-zero but not invertible. However, for the special case of Euclidean rings, we can still define an equivalence transformation eliminating the leading coefficient in a row with that of another row. The generalized Gauss elimination process works as follows: Let

$$A = \begin{pmatrix} a & * \\ b & * \end{pmatrix}, \quad g = \gcd(a,b).$$

Further, let u, v be the corresponding Bézout coefficients, i.e. $g = u \cdot a + v \cdot b$. Setting $s = -\frac{b}{g}$ and $t = \frac{a}{g}$ we obtain the unimodular transformation

$$\begin{pmatrix} u & v \\ s & t \end{pmatrix} \begin{pmatrix} a & * \\ b & * \end{pmatrix} = \begin{pmatrix} g & * \\ 0 & * \end{pmatrix}.$$

To be more precise, $\det \begin{pmatrix} u & v \\ s & t \end{pmatrix} = 1$ and its inverse is given by $\begin{pmatrix} t & -v \\ -s & u \end{pmatrix}$.

This modified Gauss elimination process leads to the Hermite normal form, providing a normal form for matrices over Euclidean rings, generalizing the reduced echelon form.

Definition 10.2.2 A matrix is in ***Hermite normal form*** if

- its zero rows are at the bottom,
- the leading coefficient of each non-zero row, called the pivot, is to the right of the pivot of the preceding row,
- all coefficients above a pivot are reduced modulo the pivot.

Theorem 10.2.2 For any $m \times n$ matrix A over an Euclidean ring, there is a unique $m \times n$ matrix H in Hermite normal form and a unimodular $m \times m$ matrix U such that $UA = H$.

The Hermite normal form for matrices over a field coincides with the reduced echelon form. Due to that correspondence, the method `echelon_form` returns the reduced echelon form, if the matrix is build over a field, and the Hermite normal form, if the matrix is build over an Euclidean ring which is not a field.

```
sage: A = matrix(ZZ, 4, 6, [2, 1, 2, 2, 2, -1, 1, 2, -1, 2, 1, -2,
    2, 1, -2, -1, 2, 2, 2, 1, 1, -1, -1, -1]); A
sage: A.echelon_form() #the Hermite normal form
[ 1  2  0  5  4 -2]
[ 0  3  0  5  0 -6]
[ 0  0  1  3  3  0]
[ 0  0  0  9 12  3]
sage: A.change_ring(QQ).echelon_form() #the reduced echelon form
[    1     0     0     0  16/9  13/9]
[    0     1     0     0 -20/9 -23/9]
[    0     0     1     0    -1    -1]
[    0     0     0     1   4/3   1/3]
```

This Hermite normal form can be simplified further using unimodular right transforming, i.e. the column actions `swap_columns(j1, j2)` and `add_multiple_of_column(j1, j2, s)`. In the end we obtain a diagonal normal form, called ***Smith normal form***. Its diagonal coefficients $(s_1, \ldots, s_n)$ are the ***elementary divisors*** of the matrix. They are totally ordered under the divisibility relation $s_i | s_{i+1}$.

Theorem 10.2.3 For any $m \times n$ matrix A with coefficients over a principal ideal ring, there exist unimodular matrices U and V of dimension $m \times m$ and $n \times n$ respectively, and a unique diagonal $m \times n$ matrix S such that $S = UAV$ such that the coefficients $s_i = S_{i,i}$ for $1 \leq i \leq m$ are the elementary divisors of A, satisfying $s_i | s_{i+1}$ for all $i < \min\{m, n\}$.

The method `elementary_divisors` returns the list of elementary divisors and the Smith normal form is computed, together with the transformation matrices U, V, with the method `smith_form`.

```
sage: A = matrix(ZZ, 4, 5, [-1, -1, -1, -2, -2, -2, 1, 1, -1, 2,
   2, 2, 2, 2, -1, 2, 2, 2, 2, 2]); A
sage: S, U, V = A.smith_form(); S
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 3 0 0]
[0 0 0 6 0]
sage: S == U*A*V
True
sage: A.elementary_divisors()
[1, 1, 3, 6]
```

■ **Example 10.2** One application of the Smith normal form is the following similarity result:

> **Lemma 10.2.4** Two square matrices A, B over the same field are similar if and only if the Smith normal form of the characteristic matrices $x1 - A, x1 - B$ coincides.

Using this characterization, we write a procedure which determines whether two matrices are similar or not.

```
sage: def charsmith(A): #calculating the smith normal form
....:     R.<x> = A.base_ring()['x'] #of the characteristic matrix
....:     n = A.nrows()
....:     M = MatrixSpace(R, n, n)
....:     cA = x*M.identity_matrix() - M(A)
....:     S, U, V = cA.smith_form()
....:     return S
sage: def similar(A, B): #checking similarity
....:     return charsmith(A) == charsmith(B)
```

Now, this procedure can determine whether two matrices over a field are equivalent.

```
sage: A = matrix(QQ, [[1, 2], [0, 1]])
sage: charsmith(A)
[            1             0]
[            0 x^2 - 2*x + 1]
sage: B = matrix(QQ, [[3, -4], [1, -1]])
sage: charsmith(B)
[            1             0]
[            0 x^2 - 2*x + 1]
sage: C = matrix(QQ, [[1, 0], [1, 2]])
sage: charsmith(C)
[            1             0]
[            0 x^2 - 3*x + 2]
sage: similar(A, B), similar(B,C)
(True, False)
```

■

Gaussian elimination reveals many matrix properties, such as the rank and the determinant. These values can also be accessed directly with the methods `det` and `rank`. More generally, the notion of rank profile is of interest when considering the matrix as a sequence of vectors.

Definition 10.2.3 The ***column rank profile*** of an $m \times n$ matrix A of rank r is the lexicographically minimal sequence of the r indices of the linearly independent columns in A.

The column rank profile can be read directly off the reduced echelon form as the sequence of column indices of the pivots. It is obtained directly with the method `pivots`.

```
sage: B = matrix(GF(7), 5, 4, [4, 5, 1, 5, 4, 1, 1, 1, 0, 6, 0, 6,
    2, 5, 1, 6, 4, 4, 0, 2]); B
[4 5 1 5]
[4 1 1 1]
[0 6 0 6]
[2 5 1 6]
[4 4 0 2]
sage: B.echelon_form()
[1 0 0 3]
[0 1 0 1]
[0 0 1 2]
[0 0 0 0]
[0 0 0 0]
sage: B.pivots()
(0, 1, 2)
```

The ***row rank profile*** is defined similarly when considering the matrix as a sequence of m row vectors and can be obtained with `pivot_rows`. It is not hard to see that the row rank profile is equivalent to the column rank profile of the transposed matrix.

```
sage: B.transpose().echelon_form()
[1 0 5 0 3]
[0 1 2 0 6]
[0 0 0 1 5]
[0 0 0 0 0]
sage: B.pivot_rows()
(0, 1, 3)
sage: B.transpose().pivots() == B.pivot_rows()
True
```

10.2.2 Linear System Solving, Image and Nullspace Basis

A linear system of equations can be represented by a matrix A and a right-hand or left-hand side vector or matrix b in equations of the form $Ax = b$ or $x^t A = b^t$, respectively. In Section 3.3.5, we have already seen that `A.solve_right(b)` and `A.solve_left(b)` solve these systems. Alternatively, we can also use `A\b` and `b/A` respectively. When

Gaussian Elimination and Applications	
Row Transvection	`add_multiple_of_row(i1, i2, s)`
Column Transvection	`add_mutiple_of_column(j1, j2, s)`
Row, Column swapping	`swap_rows(i1, i2)`, `swap_columns(j1,j2)`
Reduced Row Echelon Form, Immutable	`echelon_form`
Reduced Row Echelon Form, Mutable	`echelonize`
Elementary Divisors	`elementary_divisors`
Smith Normal Form	`smith_form`
Determinant, Rank	`det`, `rank`
Column, Row Rank Profile	`pivots`, `pivot_row`

Table 10.2.: Gaussian Elimination

the system is given by a matrix over a ring, the resolution is performed over the corresponding fraction field, e.g. a linear system over $\mathbb{Z}$ is solved over $\mathbb{Q}$. A linear system can have no solution, a unique solution or infinitely many solutions forming a vector space. However, in the case of infinitely many solutions, Sage only returns one of the solutions, zeroing out the coefficients corresponding to linearly dependent columns in the system.

```
sage: R.<x> = GF(5)['x']; R
Univariate Polynomial Ring in x over Finite Field of size 5
sage: M = MatrixSpace(R, 2, 2); M
Full MatrixSpace of 2 by 2 dense matrices over Univariate
   Polynomial Ring in x over Finite Field of size 5
sage: A = M([[2*x + 4, x^2 + 4*x + 3], [0, 3]] ); A
[      2*x + 4 x^2 + 4*x + 3]
[            0             3]
sage: b = matrix(R, [[4*x + 1], [4*x + 2]]); b
[4*x + 1]
[4*x + 2]
sage: X = A.solve_right(b); X #solution is in the fraction field
[(x^3 + 2*x^2 + 2*x + 2)/(x + 2)]
[                        3*x + 4]
sage: A = matrix(QQ, 2, 2, [1, 1, 2, 2]); b = matrix(QQ, 2, 1, [3,
    6]); A, b #has infinitely many solutions
(
[1 1]  [3]
[2 2], [6]
)
sage: A.solve_right(b) #only one solution of this infinite system
   is shown
[3]
[0]
sage: C = matrix(ZZ, [[1, 0], [0, 0]]); v = matrix(ZZ, [[0] ,[1]])
   ; C, v #has no solution
```

```
(
[1 0]  [0]
[0 0], [1]
)
sage: C.solve_right(v) #thus, we obtain an error message.
Traceback (most recent call last):
...
ValueError: matrix equation has no solutions
```

How can we determine the complete solution space? Recall, that any $m \times n$ matrix A can be interpreted as a linear map $A : K^n \to K^m$. Thus, A defines two subspaces: the image and the kernel. The ***image*** is the set of all vectors in K^m that are a linear combination of the columns of A. This subspace, together with a basis in reduced echelon form, is returned by `image`. The ***kernel*** is the subspace of K^n formed by all vectors x satisfying $Ax = 0$. This is set is computed via `right_kernel` returning the vector space with a basis in reduced echelon form. At this point, we shortly want to recall the well-known dimension formula

$$\dim(K^n) = \dim(\mathrm{Im}(A)) + \dim(\ker(A)).$$

The left kernel is defined similarly as the set of vectors $x \in K^m$ such that $x^t A = 0$, which is equivalent to the right kernel of the transposed matrix A. It is calculated with `left_kernel` or its alias `kernel`. Again, the basis is given in reduced echelon form.

```
sage: a = matrix(QQ, 3, 5, [2, 2, -1, 2, -1, 2, 1, 1, 2, -1/2, 2,
   -2, -1, 2, -1/2])
sage: a
[   2    2   -1    2   -1]
[   2    1    1    2 -1/2]
[   2   -2   -1    2 -1/2]
sage: a.image()
Vector space of degree 5 and dimension 3 over Rational Field
Basis matrix:
[    1     0     0     1 -9/32]
[    0     1     0     0  -1/8]
[    0     0     1     0  3/16]
sage: a.right_kernel()
Vector space of degree 5 and dimension 2 over Rational Field
Basis matrix:
[   1    0    0   -1    0]
[   0    1 -3/2  9/4    8]
```

Now, the right kernel can be used to describe the set of all possible solutions of an undetermined system $Ax = b$. If $\bar{x}$ is a solution of $Ax = b$ then $\bar{x} + \ker(A)$ is the set of all solutions. For example, the solution set of the equation

$$\begin{pmatrix} 1 & 1 \\ 2 & 2 \end{pmatrix} x = \begin{pmatrix} 3 \\ 6 \end{pmatrix}$$

Linear Systems	
Solving $x^tA = b$	`b/A` or `A.solve_left(b)`
Solving $Ax = b$	`A \b` or `A.solve_right(b)`
Image Space	`image`
Left Kernel, $x^tA = 0$	`kernel` or `left_kernel`
Right Kernel, $Ax = 0$	`right_kernel`

Table 10.3.: Linear Systems

is determined as follows.

```
sage: A = matrix(QQ, 2, 2, [1, 1, 2, 2]); b = matrix(QQ, 2, 1, [3,
    6]);
sage: xbar = A.solve_right(b)
sage: xbar
[3]
[0]
sage: A.right_kernel()
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[ 1 -1]
```

Thus, the solution space is

$$\begin{pmatrix}3\\0\end{pmatrix} + \mathbb{Q} \cdot \begin{pmatrix}1\\-1\end{pmatrix}.$$

```
sage: k = matrix(QQ, [[5], [-5]]);
sage: A * k
[0]
[0]
sage: A * (xbar + k) == b
True
```

10.3 Spectral Decomposition

A square matrix can be seen as the representative of an endomorphism Φ in a given basis. Any change of basis corresponds to a similarity transformation $B = U^{-1}AU$, where U is the basis change. Thus, B represents the endomorphism Φ in another basis. Two square matrices A and B are ***similar***, if they represent the same endomorphism Φ, i.e. there is an invertible matrix U such that $B = U^{-1}AU$. Thus, the properties of the endomorphism Φ are encoded in the similarity invariants of the matrix A, like the determinant and the rank.

Another similarity invariant of a square matrix A is its characteristic polynomial $\chi_A(A) = \det(x1 - A)$ and its roots, the eigenvalues. It is immediate that for each eigenvalue λ the matrix $\lambda 1 - A$ has a nontrivial kernel E_λ, called the eigenspace of λ. In

conclusion, eigenvalues can be viewed either from the algebraic point of view, as roots of the characteristic polynomial, or from the geometric point of view, considering the action of the linear operator A. Although, we obtain the same eigenvalues, there multiplicity might differ. The ***algebraic multiplicity*** of an eigenvalue is its multiplicity as the root of the characteristic polynomial, and the ***geometric multiplicity*** is the dimension of the corresponding eigenspace. The geometric multiplicity of eigenvalues is less than or equal to the algebraic multiplicity. In particular, a square matrix A is diagonalizable if and only if the geometric and the algebraic multiplicity of all of its eigenvalues coincide.

10.3.1 Cyclic Invariant Subspaces and the Frobenius Normal Form

Let A be an $n \times n$ matrix over a field K. The ***minimal polynomial*** of the matrix A is defined as the least degree monic polynomial φ_A satisfying $\varphi_A(A) = 0$. In particular, the characteristic polynomial $\chi(A)$ is a multiple of φ_A. These polynomials are calculated in Sage with the methods `characteristic_polyomial` and `minimal_poynomial`, respectively. Alternatively, we can also use their alias `charpoly` and `minpoly`

```
sage: A = matrix(ZZ, 3, 3, [0, 1, 0, 1, 0, 0, 0, 0, 1]); A
[0 1 0]
[1 0 0]
[0 0 1]
sage: A.characteristic_polynomial()
x^3 - x^2 - x + 1
sage: A.minimal_polynomial()
x^2 - 1
```

The ***Krylov sequence*** of a vector $u \in K^n$ is the family $(u, Au, A^2u, \ldots, A^nu)$ of $n+1$ vectors in K^n. Hence, there exists a minimal index $d \leq n$ such that $(u, Au, \ldots, A^du)$ is a linear dependent family. Thus, $A^du = \sum_{i=0}^{d-1} \alpha_i A^i u$ for some coefficients $\alpha_i \in K$. The associated monic polynomial

$$\varphi_{A,u}(x) = x^d \sum_{i=0}^{d-1}$$

is called the ***minimal polynomial*** of u with respect to the matrix A. By construction $\varphi_{A,u}(A)u = 0$. Since the minimal polynomial φ_A of A satisfies by definition $\varphi_A(A) = 0$, it follows that $\varphi_A(A)u = 0$. Hence, φ_A has to be a multiple of $\varphi_{A,u}$. Furthermore, there always exists a vector u whose minimal polynomial coincides with the minimal polynomial of A.

Proposition 10.3.1 Let A be an $n \times n$ matrix over a field K. Then there exists a vector $u \in K^n$ such that its minimal polynomial coincides with the minimal polynomial of A, i.e. $\varphi_{A,u} = \varphi_A$.

The command `A.maxspin(u)` returns the Krylov iterates $(u^t, u^tA, \ldots, u^tA^d)$. To obtain the Krylov iterates in our convention, we have to apply the method `maxspin` to the transpose of the matrix.

```
sage: A = matrix(GF(7), 5, 5, [0, 0, 3, 0, 0, 1, 0, 6, 0, 0, 0, 1,
```

```
    5, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 1, 5]); At = A.transpose();
   A
[0 0 3 0 0]
[1 0 6 0 0]
[0 1 5 0 0]
[0 0 0 0 5]
[0 0 0 1 5]
sage: e1 = identity_matrix(GF(7), 5)[0]; e4 = identity_matrix(GF
   (7), 5)[3] #first and fourth basis vector
sage: At.maxspin(e1) #here d = 3
[(1, 0, 0, 0, 0), (0, 1, 0, 0, 0), (0, 0, 1, 0, 0)]
sage: A.maxspin(e4) #here d = 2
[(0, 0, 0, 1, 0), (0, 0, 0, 0, 1)]
sage: At.maxspin(e1 + e4) #here d = 5
[(1, 0, 0, 1, 0),
 (0, 1, 0, 0, 1),
 (0, 0, 1, 5, 5),
 (3, 6, 5, 4, 2),
 (1, 5, 3, 3, 0)]
```

In particular, $A^3 \cdot e_1$ and $A^2 \cdot e4$ are not linear independent of the other Krylov iterates anymore.

```
sage: A^3 * e1
(3, 6, 5, 0, 0)
sage: A^2 * e4
(0, 0, 0, 5, 5)
```

In the above example, we can read off the corresponding minimal polynomials immediately.

$$\varphi_{A,e_1} = x^3 - \left(5x^2 + 6x + 3\right),$$
$$\varphi_{A,e_4} = x^2 - (5x + 5).$$

The ***companion matrix*** associated to a monic polynomial $P = x^k - \sum_{i=0}^{k-1} \alpha_i x^i$ is the matrix

$$C_P = \begin{pmatrix} 0 & & & \alpha_0 \\ 1 & & & \alpha_1 \\ & \ddots & & \vdots \\ & & 1 & \alpha_{k-1} \end{pmatrix}.$$

By construction P is the minimal polynomial of C_P.

Proposition 10.3.2 Let $K_u = (u, Au, \ldots, A^{d-1}u)$ be the matrix formed by the first $d = \deg \varphi_{A,u}$ Krylov iterates of the vector u. Then $AK_u = K_u C_{\varphi_{A,u}}$.

If $d = n$, K_u is an invertible matrix. Hence $K_u^{-1} A K_u = C_{\varphi_{A,u}}$. Since this is a similarity transformation it preserves the characteristic and the minimal polynomial. But now, we

can read off the minimal polynomial directly from the last column of the companion matrix $C_{\varphi_{A,u}}$.

```
sage: A = matrix(GF(97), 4, 4, [86, 1, 6, 68, 34, 24, 8, 35, 15,
   36, 68, 42, 27, 1, 78, 26]); A
[86  1  6 68]
[34 24  8 35]
[15 36 68 42]
[27  1 78 26]
sage: e1 = identity_matrix(GF(97), 4)[0]; e1
(1, 0, 0, 0)
sage: U = matrix(A.transpose().maxspin(e1)).transpose(); U, U.rank
   ()
(
[ 1 86 44 24]
[ 0 34 52 82]
[ 0 15 12 19]
[ 0 27 57 69], 4
)
sage: F = U^(-1)*A*U; F
[ 0  0  0 83]
[ 1  0  0 77]
[ 0  1  0 20]
[ 0  0  1 10]
sage: K.<x> = GF(97)['x']
sage: P = x^4 - add(F[i, 3] * x^i for i in srange(4)); P #the
   minimal polynomial
x^4 + 87*x^3 + 77*x^2 + 20*x + 14
sage: A.characteristic_polynomial() == P
True
```

If $d \leq n$ the iterates $u, \ldots, A^{d-1}u$ form a basis of an A-invariant linear subspace I i.e. $AI \subset I$. Such a subspace I is called a ***cyclic invariant subspace***. The dimension of a cyclic invariant subspace is equal to the degree of the minimal polynomial $\varphi_{A,u}$. Thus, it is bounded by the degree of the minimal polynomial of A.

If the dimension of I_1 is maximal, i.e. equals the dimension is equal to the degree of the minimal polynomial φ_A, the space is generated by the Krylov iterates of the vector u_1 from Proposition 10.3.1. Considering the complementary A-invariant subspace V, we can repeat the same computation modulo I_1 to find a second cyclic invariant subspace I_2 whose basis is given by the Krylov iterates of a vector u_2. Continuing this procedure, we can iteratively build the block matrix $K = \mathrm{Diag}(K_{u_1}, K_{u_2}, \ldots, K_{u_k})$. The matrix K is invertible and

$$K^{-1}AK = \begin{pmatrix} C_{\varphi_1} & & \\ & \ddots & \\ & & C_{\varphi_k} \end{pmatrix}$$

Let φ_i be the minimal polynomial of u_i. By construction, each u_i is annihilated by each

minimal polynomial φ_j with $j \leq i$. Thus, we have the division order $\varphi_i | \varphi_{i-1}$ for any $2 \leq i \leq k$. One can show that for every matrix A, there is a unique sequence of such polynomials $\varphi_1, \ldots, \varphi_k$. Therefore, the block diagonal matrix $\mathrm{Diag}(C_{\varphi_1}, \ldots, C_{\varphi_k})$ is a normal form, called the ***Frobenius normal form***.

Theorem 10.3.3 For every matrix A over a field there is a unique matrix $F = \mathrm{Diag}(C_{\varphi_1}, \ldots, C_{\varphi_k})$ with $\varphi_i | \varphi_{i-1}$ for all $i \leq k$, that is similar to A.

So far, Sage can only compute the Frobenius normal form over $\mathbb{Q}$ of matrices with coefficients in $\mathbb{Z}$. There is a slight abuse of notation involved, because formally the Frobenius normal form is defined for matrices over a field, here $\mathbb{Q}$, but the method `frobenius` is only available for matrices with integer coefficients

```
sage: A = matrix(ZZ, [[6, 0, -2, 4, 0, 0, 0, -2], [14, -1, 0, 6,
   0, -1, -1, 1], [2, 2, 0, 1, 0, 0, 1, 0], [-12, 0, 5, -8, 0, 0,
    0, 4], [0, 4, 0, 0, 0, 0, 4, 0], [0, 0, 0, 0, 1, 0, 0, 0],
   [-14, 2, 0, -6, 0, 2, 2, -1], [-4, 0, 2, -4, 0, 0, 0, 4]]); A
[  6   0  -2   4   0   0   0  -2]
[ 14  -1   0   6   0  -1  -1   1]
[  2   2   0   1   0   0   1   0]
[-12   0   5  -8   0   0   0   4]
[  0   4   0   0   0   0   4   0]
[  0   0   0   0   1   0   0   0]
[-14   2   0  -6   0   2   2  -1]
[ -4   0   2  -4   0   0   0   4]
sage: A.frobenius()
[0 0 0 4 0 0 0 0]
[1 0 0 4 0 0 0 0]
[0 1 0 1 0 0 0 0]
[0 0 1 0 0 0 0 0]
[0 0 0 0 0 0 4 0]
[0 0 0 0 1 0 0 0]
[0 0 0 0 0 1 1 0]
[0 0 0 0 0 0 0 2]
```

Now we can read off the corresponding polynomial $\varphi_1, \varphi_2, \varphi_3$. But we can also obtain the polynomials directly by passing 1 as an argument.

```
sage: A.frobenius(1)
[x^4 - x^2 - 4*x - 4, x^3 - x^2 - 4, x - 2]
```

Moreover, inserting 2 as an argument returns the transformation matrix K. Observe that K is here the *transposed inverse* of the matrix containing the Krylov iterates we discussed above. The reason is that Sage prefers rows over columns. hence, inverting and transposing K gives us the matrix consisting of the Krylov iterates and we can read off the used vectors u_1, u_2, u_3.

```
sage: F, K = A.frobenius(2); invK = K.transpose()^(-1); At = A.
   transpose()
```

```
sage: u1 = invK[0]; At.maxspin(u1)
[(1, 0, 0, 0, 0, 0, 0, 0),
 (6, 14, 2, -12, 0, 0, -14, -4),
 (-8, 8, 14, 18, 0, 0, -8, 12),
 (-28, 8, 10, 70, 0, 0, -8, 36)]
sage: u2 = invK[4]; At.maxspin(u2)
[(17/128, 17/64, 17/448, -51/224, 1, 0, -17/64, -17/224),
 (-17/448, 187/448, 17/56, 51/448, 0, 1, -187/448, 17/112),
 (-153/224, -39/56, 51/112, 187/112, 0, 0, 95/56, 51/56)]
sage: u3 = invK[7]; At.maxspin(u3)
[(-3/16, -135/64, -45/32, -21/64, 0, 0, 135/64, 3/8)]
```

■ **Example 10.3** As an application, we write procedure that determines whether two input matrices A and B are similar and, in the case of similarity, returns the transformation matrix U.

```
sage: def Similar(A, B):
....:     F1, U1 = A.frobenius(2)
....:     F2, U2 = B.frobenius(2)
....:     if F1 == F2:
....:         return True, U2^(-1)*U1
....:     else:
....:         return False, None
```

Below we test this procedure at an explicit example.

```
sage: B = matrix(ZZ, [[0, 1, 4, 0, 4], [4, -2, 0, -4, -2], [0, 0,
    0, 2, 1], [-4, 2, 2, 0, -1], [-4, -2, 1, 2, 0]]);B
[ 0  1  4  0  4]
[ 4 -2  0 -4 -2]
[ 0  0  0  2  1]
[-4  2  2  0 -1]
[-4 -2  1  2  0]
sage: U = matrix(ZZ, [[3, 3, -9, -14, 40], [-1, -2, 4, 2, 1], [2,
    4, -7, -1, -13], [-1, 0, 1, 4, -15], [-4, -13, 26, 8, 30]]); U
[  3   3  -9 -14  40]
[ -1  -2   4   2   1]
[  2   4  -7  -1 -13]
[ -1   0   1   4 -15]
[ -4 -13  26   8  30]
sage: A = (U^-1* B * U).change_ring(ZZ); A
[  -887  -1349   3191   2979  -5240]
[  3320   5028 -11917 -11197  19893]
[  3622   5453 -12996 -12386  22478]
[ -3975  -5963  14259  13705 -25178]
[  -759  -1137   2723   2627  -4852]
sage: ok, V = Similar(A, B); ok
True
```

```
sage: V^{-1}*B*V == A
True
sage: ok, V = Similar(2*A, B); ok
False
```

■

10.3.2 Eigenvalues and Eigenvectors

Let A be a $n \times n$ matrix over a field K with Frobenius normal form $F = \mathrm{Diag}(C_{\varphi_1}, \ldots, C_{\varphi_k})$ and let $\varphi_1 = \psi_1^{m_1} \cdots \psi_s^{m_s}$ be the factorization into irreducible polynomials. Since $\varphi_i | \varphi_1$ for all $1 \leq i \leq k$ every further minimal polynomial can be written as a product $\varphi_i = \psi_1^{m_{i,1}} \cdots \psi_s^{m_{i,s}}$ with $0 \leq m_{i,j} \leq m_j$. Thus, each companion block C_{φ_i} in the Frobenius normal form can be replaced by the diagonal block $\mathrm{Diag}(C_{\psi_1^{m_{i,1}}}, \ldots, C_{\psi_s^{m_{i,s}}})$.

$$\begin{pmatrix} \begin{bmatrix} C_{\psi_1^{m_1}} & & \\ & \ddots & \\ & & C_{\psi_s^{m_s}} \end{bmatrix} & & & \\ & \begin{bmatrix} C_{\psi_1^{m_{2,1}}} & & \\ & \ddots & \end{bmatrix} & & \\ & & \ddots & \\ & & & \begin{bmatrix} C_{\psi_1^{m_{k,1}}} & & \\ & \ddots & \end{bmatrix} \end{pmatrix}$$

If an irreducible factor ψ_j has degree and multiplicity equal to 1, its companion block C_{ψ_j} is simply a 1×1 matrix containing the simple root λ_j of ψ_j. In particular, it follows that λ_j is an eigenvalue. If all minimal polynomial φ_i split and are square-free the matrix A is diagonalizable.

In Sage, the eigenvalues of a matrix can be obtained with the method `eigenvalues`.

```
sage: A = matrix(GF(7), 4, [5, 5, 4, 3, 0, 3, 3, 4, 0, 1, 5, 4, 6,
    0, 6, 3]); A
[5 5 4 3]
[0 3 3 4]
[0 1 5 4]
[6 0 6 3]
sage: A.eigenvalues()
[4, 1, 2, 2]
```

There are various methods available in Sage that return more information of the corresponding eigenspaces. First, the methods `eigenvectore_right` and `eigenvectors_left` return a list consisting of 3-tuples containing the eigenvalue, the right, respectively left, eigenvectors and the algebraic multiplicity. In addition Sage provides the methods `eigenspaces_right` and `eigenspaces_left` which return the right, respectively left eigenspaces together with a basis of eigenvectors. Last but not least, the methods `eigenmatrix_right` and `eigenmatrix_left` return a tuple containing the diagonal-

ized matrix together with the transformation matrix of right, respectively left, eigenvectors. Keep in mind that the definition of eigenvalues described in linear algebra correspond to the *right* eigenvectors.

```
sage: A.eigenvectors_right()
[(4, [
  (1, 5, 5, 1)
  ], 1), (1, [
  (0, 1, 1, 4)
  ], 1), (2, [
  (1, 3, 0, 1),
  (0, 0, 1, 1)
  ], 2)]
sage: A.eigenspaces_right()
[
(4, Vector space of degree 4 and dimension 1 over Finite Field of
   size 7
User basis matrix:
[1 5 5 1]),
(1, Vector space of degree 4 and dimension 1 over Finite Field of
   size 7
User basis matrix:
[0 1 1 4]),
(2, Vector space of degree 4 and dimension 2 over Finite Field of
   size 7
User basis matrix:
[1 3 0 1]
[0 0 1 1])
]
sage: A.eigenmatrix_right()
(
[4 0 0 0]  [1 0 1 0]
[0 1 0 0]  [5 1 3 0]
[0 0 2 0]  [5 1 0 1]
[0 0 0 2], [1 4 1 1]
)
sage: T = A.eigenmatrix_right()[1]; T^(-1)*A*T
```

If the considered matrix A has eigenvalues outside the fraction field of the base ring, we can still obtain all eigenspaces with option `format = 'all'`. Another option is to request a single eigenspace for each irreducible factor of the characteristic polynomial using the option `format = 'galois'`. There, Sage presents the eigenspaces in finite field extensions of the base ring.

```
sage: MS = MatrixSpace(QQ, 2, 2)
sage: A = MS([1, -4, 1, -1])
sage: A.charpoly() #i*sqrt(3) is not a rational number
x^2 + 3
```

```
sage: A.eigenspaces_right(format = 'all')
[
(-1.732050807568878?*I, Vector space of degree 2 and dimension 1
   over Algebraic Field
User basis matrix:
[                                              1 0.250000000000000000?
    + 0.4330127018922193?*I]),
(1.732050807568878?*I, Vector space of degree 2 and dimension 1
   over Algebraic Field
User basis matrix:
[                                              1 0.250000000000000000?
    - 0.4330127018922193?*I])
]
sage: A.eigenspaces_right(format = 'galois')
[
(a0, Vector space of degree 2 and dimension 1 over Number Field in
    a0 with defining polynomial x^2 + 3
User basis matrix:
[            1 -1/4*a0 + 1/4])
]
```

10.3.3 Jacobi Normal Form

It is well-known that not every $n \times n$ matrix A is diagonalizable. To be more precise, a matrix is diagonalizable if and only if the minimal polynomial splits over the base field such that all factors have multiplicity 1. If the minimal polynomial does not split at all, the "best" normal form we can achieve is the Frobenius normal form, see Theorem 10.3.3. In the intermediate case, i.e. when the minimal polynomial splits but has factors with multiplicity strictly larger than 1, we can still transform A to an upper triangular matrix with the eigenvalues on its diagonal. The most reduced form is the ***Jordan normal form***. This is an upper triangular matrix build out of Jordan blocks $J_{\lambda,k}$. To be more precise, a Jordan block $J_{\lambda,k}$ of order k associated to an eigenvalue λ is the $k \times k$ matrix

$$J_{\lambda,k} = \begin{pmatrix} \lambda & 1 & & \\ & \ddots & \ddots & \\ & & \lambda & 1 \\ & & & \lambda \end{pmatrix}$$

The Jordan block $J_{\lambda,k}$ is built with the command `jordan_block(lambda, k)`. The characteristic polynomial of a Jordan block $J_{\lambda,k}$ equals its minimal polynomial and is given by $(x-\lambda)^k$.

```
sage: jordan_block(2, 3)
[2 1 0]
[0 2 1]
[0 0 2]
```

```
sage: jordan_block(2, 3).charpoly().factor()
(x - 2)^3
```

Every matrix A whose minimal polynomial splits admits a Jordan normal form, i.e. it is similar to a matrix of the form

$$J_A = \begin{pmatrix} J_{\lambda_1,m_1} & & \\ & \ddots & \\ & & J_{\lambda_2,m_2} \end{pmatrix},$$

where $\lambda_1 \leq \ldots \leq \lambda_s$ are the eigenvalues of A. In Sage, the Jordan normal form is calculated with `jordan_form`. Adding the argument `transformation = True` returns the corresponding transformation matrix U.

```
sage: A = matrix(ZZ, 4, [3,-1, 0, -1, 0, 2, 0, -1, 1, -1, 2, 0, 1,
    -1, -1, 3]); A
[ 3 -1  0 -1]
[ 0  2  0 -1]
[ 1 -1  2  0]
[ 1 -1 -1  3]
sage: A.jordan_form()
[3|0|0 0]
[-+-+---]
[0|3|0 0]
[-+-+---]
[0|0|2 1]
[0|0|0 2]
sage: J, U = A.jordan_form(transformation = True)
sage: U^(-1)*A*U == J
True
```

■ **Example 10.4** The following program first test whether the minimal polynomial of the given matrix A splits. If this is the case, the program returns the Jordan normal form. Otherwise it returns a string saying that A does not have a Jordan normal form.

```
sage: def split_minpoly(A): #checks if minimal polynomial splits
....:     p = A.minpoly().factor() #factorize minimal polynomial
....:     L = list(p) #list of factors with multiplicity
....:     B = True #start Boolean
....:     for i in srange(len(L)):
....:         if L[i][0].degree() != 1:
....:             B = False
....:             break
....:     return B
sage: def jordan_check(A):
....:     if split_minpoly(A) == True:
....:         return A.jordan_form()
....:     else:
....:         return 'This matrix has no Jordan normal form.'
```

Spectral Decomposition	
Minimal Polynomial	`minimal_polynomial` or `minpoly`
Characteristic Polynomial	`characteristic_polynomial` or `charpoly`
Krylov Iterates of u on the Left-hand Side	`maxspin(u)`
Frobenius Normal Form	`frobenius`
Eigenvalues	`eigenvalues`
Left, Right Eigenvectors	`eigenvectors_left` or `eigenvectors_right`
Left, Right Eigenspaces	`eigenspaces_left` or `eigenspaces_right`
Diagonalization	`eigenmatrix_left`, `eigenmatrix_right`
Jordan Normal Form	`jordan_form`

Table 10.4.: Spectral Decomposition

In the following we test this program at two matrices: one of them admitting a normal form and the other not.

```
sage: A = matrix(QQ, [[2, 4, 3], [-4, -6, -3], [3, 3, 1]]); A
sage: A.minpoly().factor() #has Jordan normal form
(x - 1) * (x + 2)^2
sage: jordan_check(A)
[ 1| 0  0]
[--+-----]
[ 0|-2  1]
[ 0| 0 -2]
sage: B = matrix(QQ, [[2, 2, 2], [-2, 1, 2], [1, 0, 1]]); B
[ 2  2  2]
[-2  1  2]
[ 1  0  1]
sage: B.minpoly().factor() #has no Jordan normal form
x^3 - 4*x^2 + 7*x - 8
sage: jordan_check(B)
'This␣matrix␣has␣no␣Jordan␣normal␣form.'
```

■

11. Polynomial Systems

A polynomial system is a system of polynomial equations in several variables. Compared to univariate polynomials, i.e. polynomials with one indeterminate, polynomials with several variables, so-called ***multivariate*** polynomials yield nice mathematical properties, but also new difficulties. One of them is that the ring $K[x_1,\ldots,x_n]$ for a field K is not principal anymore. In this chapter, we describe various methods how to study and solve polynomial systems over fields.

11.1 Polynomials in Several Variables

Similar to other algebraic structures in Sage, we first have to construct our computation domain, the polynomial ring $A[x_1,\ldots,x_n]$. This is done analogously to univariate polynomial rings, see Chapter 9.

```
sage: R = PolynomialRing(QQ, 'x, y, z') #constructing the ring
sage: x, y, z = R.gens() #assign indeterminates
sage: R.<x, y, z> = QQ['x, y, z']; R #short version
Multivariate Polynomial Ring in x, y, z over Rational Field
```

Moreover, we can construct multivariate polynomial rings with a large family of indeterminates with the same name and integer indices.

```
sage: R = PolynomialRing(QQ, 'x', 10)
sage: x = R.gens(); x
(x0, x1, x2, x3, x4, x5, x6, x7, x8, x9)
```

After assigning the n-tuple to a Python variable, here `x`, the single indeterminates x_i are accessed via `x[i]`.

```
sage: sum(x[i] for i in srange(5))
```

```
x0 + x1 + x2 + x3 + x4
```

In the above construction it is not possible to use the short version `R.<x>= PolynomialRing(QQ, 'x', 10)`.

```
sage: R.<x>= PolynomialRing(QQ, 'x', 10)
Traceback (most recent call last):
...
TypeError: variable names specified twice inconsistently: ('x0', '
   x1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7', 'x8', 'x9') and ('x'
   ,)
```

Although, we can construct polynomial rings with arbitrarily many indeterminates, it might happen that we do not know a priori how many variables we need in the end. In that case the usage of `PolynomialRing` can be quite tedious. Each time, we need to introduce a new variable, we have to construct a new polynomial ring with more variables and convert everything we have done so far to that new computation domain. To circumvent this problem, Sage provide the constructor`InfinitePolynomialRing`. This command creates a polynomial ring with an infinite number of variables. Each generator of `InfinitePolynomialRing` corresponds to a family of variables indexed by integers. In particular, we can construct multivariate polynomial rings with one or several infinite families of indeterminates.

```
sage: R.<x, y> = InfinitePolynomialRing(ZZ); R
Infinite polynomial ring in x, y over Integer Ring
sage: p = mul(x[k] - y[k] for k in srange(2)); p
x_1*x_0 - x_1*y_0 - x_0*y_1 + y_1*y_0
sage p + x[20]
x_20 + x_1*x_0 - x_1*y_0 - x_0*y_1 + y_1*y_0
```

Although infinite polynomial rings are quite useful and flexible, they are way less efficient than the usual polynomial ring constructed with `PolynomialRing`. Therefore, we recommend to change back to a classical polynomial ring as soon as the number of variables is fixed. The method `polynomial` changes the parent of an element of an infinite polynomial ring to the corresponding polynomial ring with all variables which have been produced so far. The obtained ring is in general not the optimal choice.

```
sage: p.polynomial().parent()
Multivariate Polynomial Ring in x_20, x_19, x_18, x_17, x_16, x_15
   , x_14, x_13, x_12, x_11, x_10, x_9, x_8, x_7, x_6, x_5, x_4,
   x_3, x_2, x_1, x_0, y_20, y_19, y_18, y_17, y_16, y_15, y_14,
   y_13, y_12, y_11, y_10, y_9, y_8, y_7, y_6, y_5, y_4, y_3, y_2
   , y_1, y_0 over Integer Ring
```

Another possibility to define multivariate polynomials is to define a univariate polynomial ring with coefficients in another polynomial ring. However, the polynomial ring $A[x][y]$ behaves differently than the polynomial ring $A[x,y]$. In $A[x][y]$ the variable y is the main variable while x behaves more like a parameter. Starting with $A[x,y]$ we can use the method `polynomial` to isolate one or more variables. In the example below,

`p.polynomial(t)` returns a polynomial in `t` with coefficients in `QQ['x, y, z']`.

```
sage: R.<x, y, z, t> = QQ['x, y, z, t']; p = (x + y + z*t)^2
sage: p.polynomial(t)
z^2*t^2 + (2*x*z + 2*y*z)*t + x^2 + 2*x*y + y^2
sage: p.polynomial(t).parent()
Univariate Polynomial Ring in t over Multivariate Polynomial Ring
   in x, y, z over Rational Field
```

The other conversion is done by a simple "parent change". We observe that the same polynomial is represented in different ways depending on the parent.

```
sage: x = polygen(QQ); y = polygen(QQ[x], 'y')
sage: p = x^3 + x*y + y + y^2; p, p.parent()
(y^2 + (x + 1)*y + x^3,
 Univariate Polynomial Ring in y over Univariate Polynomial Ring
    in x over Rational Field)
sage: q = QQ['x, y'](p); q, q.parent()
(x^3 + x*y + y^2 + y, Multivariate Polynomial Ring in x, y over
   Rational Field)
sage: s = QQ['y']['x'](q); s, s.parent()
(x^3 + y*x + y^2 + y,
 Univariate Polynomial Ring in x over Univariate Polynomial Ring
    in y over Rational Field)
```

Univariate polynomial rings $A[x]$ have the well-defined canonical normal form $p = a_k x^k + \ldots + a_1 x + a_0$. It is not possible to generalize this normal form to multivariate polynomials since we have to choose a way to order the different monomials, e.g. we have to choose between $x^2y + xy^2$ and $xy^2 + yx^2$. By default, Sage first ranks monomials according to their total degree and, in a second step, by the lexicographic order of the degrees of the indeterminates. Thus, viewing the same polynomial $p(x,y)$ once in `Q['x, y']` and once in `QQ['y, x']` leads to equal polynomials with different parents and different representation.

```
sage: R.<x, y> = QQ['x, y']; p1 = x^3 + x^2*y + x*y^2 + y^3 + x^2
   + x*y + y^2 + x + y + 1; p1
x^3 + x^2*y + x*y^2 + y^3 + x^2 + x*y + y^2 + x + y + 1
sage: R.<y, x> = QQ['y, x']; p2= x^3 + x^2*y + x*y^2 + y^3 + x^2 +
    x*y + y^2 + x + y + 1; p2
y^3 + y^2*x + y*x^2 + x^3 + y^2 + y*x + x^2 + y + x + 1
sage: p1.parent(), p2.parent()
(Multivariate Polynomial Ring in x, y over Rational Field,
 Multivariate Polynomial Ring in y, x over Rational Field)
sage: p1 == p2, p1 is p2
(True, False)
```

It is also possible to customize the indices of the indeterminates. For example, we can define the ring $\mathbb{Q}[x_2, x_3, \ldots, x_{37}]$ whose indeterminates are indexed by the prime numbers less than 40 together with the corresponding Python variables `x2, x3, ...,`

x37 to access them.

```
sage: ['x%d' % n for n in primes(40)] #constructing list of
   indetermiantes
['x2', 'x3', 'x5', 'x7', 'x11', 'x13', 'x17', 'x19', 'x23', 'x29',
    'x31', 'x37']
sage: R = PolynomialRing(QQ, ['x%d' % n for n in primes(40)]); R
Multivariate Polynomial Ring in x2, x3, x5, x7, x11, x13, x17, x19
   , x23, x29, x31, x37 over Rational Field
sage: R.inject_variables()
Defining x2, x3, x5, x7, x11, x13, x17, x19, x23, x29, x31, x37
sage: (x2 + x5*x11).parent()
Multivariate Polynomial Ring in x2, x3, x5, x7, x11, x13, x17, x19
   , x23, x29, x31, x37 over Rational Field
```

Henceforth, $R = A[x_1, \ldots, x_n]$ denotes a multivariate polynomial ring with coefficients in A. A ***monomial*** is an expression of the form $x_1^{\alpha_1} x_2^{\alpha_2} \cdots x_n^{\alpha_n} = x^{\alpha}$. The n-tupel $\alpha = (\alpha_1, \ldots, \alpha_n)$ is called the ***exponent*** of the monomial. A ***term*** is the product of a monomial and its ***coefficient***.

As there is no unique way to order the terms of a multivariate poynomial there is no well-defined notion of a leading coefficient in the mathematical sense. Nevertheless, once an order has been fixed at the ring construction, Sage defines leftmost written term as the leading term, consisting of the leading coefficient an the leading monomial. They are returned by the methods `lt`, `lc` and `lm`.

```
sage: R.<x, y, z> = QQ['x,␣y,␣z']
sage: p = 7*x^2*y^2 + 3*y*x^2 + x^3 + 6
sage: p.lc(), p.lm()
(7, x^2*y^2)
sage: p.lc()*p.lm() == p.lt()
True
```

The usual arithmetic operations `+` `-` and `*`, as well as most of the operations for univariate polynomials are also available for multivariate polynomials.

```
sage: p.coefficients() #return non-zero coefficients
[7, 1, 3, 6]
sage: p.dict() #returns non-zero coefficients as a dict
{(2, 2, 0): 7, (3, 0, 0): 1, (2, 1, 0): 3, (0, 0, 0): 6}
sage: p.derivative(x) #derivative wrt to x
14*x*y^2 + 3*x^2 + 6*x*y
```

For multivariate polynomials, the square-bracket operator `[]` accepts as parameter either a monomial or its exponent and returns the corresponding coefficient. Here, the order of the indeterminates is essential.

```
sage: p[x^2*y] == p[(2, 1, 0)] == p[2, 1, 0] == 3
True
```

Likewise, if we want to evaluate a polynomial we need to give values to all indetermi-

nates. Otherwise we use `subs` to substitute arbitrary expressions.

```
sage: p(0, 3, -1)
6
sage: p.subs(x = 1, z = x^2 + 1)
7*y^2 + 3*y + 7
```

In contrast to univariate polynomials a multivariate polynomial has various notions of degree, e.g. we can ask for the total degree or the degree in one of its indeterminates. This is done with the methods `degree` and `degrees`.

```
sage: p.degree() #total degree
4
sage: p.degree(x) #degree in a chosen variable (here x)
3
sage: p.degrees() #partial degrees (here: (in x, in y, in z))
(3, 2, 0)
```

All methods discussed so far can be applied to multivariate polynomials with coefficients in any commutative ring A. However, the more advanced methods and function available in `Sage` are in general limited to multivariate polynomials over a field. Therefore, we only consider multivariate polynomial rings $R = K[x_1, \ldots, x_n]$ over a field K in the remainder.

As for univariate polynomial rings, we have to discuss the division operation in more details. Here, we encounter our first problem. Since multivariate polynomial rings are not principal, they are also not Euclidean. Thus, the Euclidean division is not well-defined. Nevertheless, `Sage` provides the method `quo_rem` and its associated operation `//` and `%` for multivariate polynomials. They still satisfy the relation `(p // q)*q + (p % q) == q` and coincide with the Euclidean division whenever p and q only depend on one common variable. Although it is not a Euclidean division in the mathematical sense, it is still useful when the division is exact or when the divisor is a monomial. The different cases can be treated by reducing a polynomial modulo an ideal using the `mod` method. This method takes the monomial order of the multivariate polynomial ring R into account.

```
sage: R.<x, y> = QQ['x, y']; p = x^2 + y^2; q = x + y
sage: p//q, p%q
(-x + y, 2*x^2)
sage: (p//q)*q + p%q == p
True
sage: p.mod(q) #this is NOT the same as p%q
2*y^2
```

The methods `divides`, `gcd`, `lcm` and `factor` for univariate polynomials also apply to multivariate polynomials.

```
sage: R.<x, y> = QQ[exp(2*I*pi/5)]['x, y']
sage: (x^10 + y^5).gcd(x^4 - y^2)
x^2 + y
```

Construction of Polynomial Rings	
Construction of the Ring $A[x,y]$	`PolynomialRing(A, 'x, y')` or `A['x, y']`
Construction of the Ring $A[x_0,\ldots,x_{n-1}]$	`PolynomialRing(A, 'x', n)`
ring $A[x_0,x_1,\ldots,y_0,y_1,\ldots]$	`InfinitePolynomialRing(A, ['x', 'y'])`
n-tuple of Generators	`R.gens()`
1st, 2nd, ... Generator	`R.0, R.1, ...`
Indeterminates of $R = A[x,y][z][\ldots]$	`R.variable_names_recursive()`
Conversion $A[x_1,x_2,y] \to A[x_1,x_2][y]$	`p.polynomial(y)`
Access to Coefficients	
Non-Zero Coefficients	`p.coefficients()`
Coefficient of the Monomial x^2y	`p[x^2*y]` or `p[2,1]`
Degree (total, in x, partial)	`p.degree()` `p.degree(x)`, `p.degrees()`
Leading Monomial / Coefficient / Term	`p.lm()`, `p.lc()`, `p.lt()`
Basic Operations	
Partial Derivative $\mathrm{d}/\mathrm{d}x$	`p.derivative(x)`
Evaluation $p(x,y)_{\|x=a,y=b}$	`p.subs(x = a, y = b` or `p(a, b)`
Factorization	`p.factor()`
gcd, lcm	`p.gcd(q), p.lcm(q)`

Table 11.1.: Multivariate Polynomials

```
sage: (x^10 + y^5).factor()
(x^2 + y) * (x^2 + (a^3)*y) * (x^2 + (a^2)*y) * (x^2 + (a)*y) * (x
   ^2 + (-a^3 - a^2 - a - 1)*y)
```

11.2 Polynomial Systems and Ideals

Solving a polynomial system is not an easy task. Even if the solution set is finite it is not always the optimal choice to simply list the solution points. In many occasions it is handier to find a suitable parametrization of the solution set or to decompose it into several subsets. The main tool of characterizing and determining the solution set of a polynomial system are ideals J of the corresponding multivariate polynomial ring R and the corresponding quotient rings R/J. This section is devoted to solving strategies for polynomial systems and characterizations of the solution set.

11.2.1 A first Example

Here, we shortly present different ways to find and understand the solutions of a polynomial system along the explicit example

$$\begin{cases} x^2yz &= 18, \\ xy^3z &= 24, \\ xyz^4 &= 6. \end{cases} \tag{11.1}$$

If we use symbolic variables and the `solve` method as discussed in Section 3.2.2, Sage returns one exact solution and 16 numerical complex solutions.

```
sage: x, y, z = var('x␣y␣z')
sage: L = solve([x^2*y*z == 18, x*y^3*z == 24, x*y*z^4 == 6], x, y
   , z)
sage: len(L), L[:5]
(17, [[x == 3, y == 2, z == 1], [x == (1.337215067329613 -
   2.685489874065195*I), y == (-1.700434271459228 +
   1.052864325754712*I), z == (0.9324722294043555 -
   0.3612416661871523*I)], [x == (1.337215067329613 +
   2.685489874065194*I), y == (-1.700434271459228 -
   1.052864325754712*I), z == (0.9324722294043555 +
   0.3612416661871523*I)], [x == (-2.550651407188846 -
   1.579296488632072*I), y == (-0.5473259801441661 +
   1.923651286345638*I), z == (-0.9829730996839015 -
   0.1837495178165701*I)], [x == (-2.550651407188845 +
   1.57929648863207*I), y == (-0.5473259801441662 -
   1.923651286345638*I), z == (-0.9829730996839015 +
   0.1837495178165701*I)]])
```

In particular, we only obtain a simple list of numerical approximations without revealing any kind of structure.

However, translating this system to multivariate polynomial rings, Sage provides various methods which can be used to solve this system exactly and study the structure of the solution set, but no direct methods. Hence, we have to combine our mathematical knowledge with the available methods in Sage. The interpret the equation in an algebraic way, i.e. we consider the ideal generated by the equations of (11.1) in the multivariate polynomial ring $\mathbb{Q}[x,y,z]$. To generate this ideal in Sage we use the method `ideal` where the arguments are the polynomials that should be equal to 0.

```
sage: R.<x, y, z> = QQ['x␣y␣z']
sage: J = R.ideal(x^2*y*z - 18, x*y^3*z - 24, x*y*z^4 - 6)
```

The dimension of the solution space of (11.1) is equal to dimension of the ideal J, which can be computed with `dimension`.

```
sage: J.dimension()
0
```

Since J is a zero dimensional ideal, the system S only has a finite number of solution. In that case, we can apply method `variety` to the ideal to determine the solutions. By default, `variety` computes the solutions in the base field of the polynomial ring.

```
sage: J.variety()
[{z: 1, y: 2, x: 3}]
```

Thus, $(3,2,1)$ is the only rational solutions of S. To find all explicit complex solutions switch to the algebraic closure of $\mathbb{Q}$, i.e. the field of algebraic numbers $\overline{\mathbb{Q}}$.

```
sage: V = J.variety(QQbar)
sage: len(V)
17
```

As algebraic numbers are stored exactly in Sage, V contains the 17 *exact* solutions of S. Each solution point is given by a dictionary whose keys are the generators of `QQbar['x, y, z']` and its value is the corresponding coordinate of the solution point. Except for the rational solution, the first coordinate of the other solutions are algebraic numbers of degree 16. To access the different elements of V we assign the generators of `QQbar['x, y, z']` to the Python variables `xx, yy, zz`. Observe that we should not use the Python variable `x, y, z` as they are already used for the generators of `QQ['x, y, z']`.

```
sage: (xx, yy, zz) =  QQbar['x, y, z'].gens()
sage: [pt[xx].degree() for pt in V]
[1, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16,
   16]
```

Now, we have an exact representation of the complex solutions of (11.1), but we still do not know much about the structure of the solution set. This needs a more thorough study of the solution set V.

A first observation shows that the modulus $(|x|,|y|,|z|) = (3,2,1)$ for any solutions $(x,y,z) \in V$.

```
sage: Set(tuple(abs(pt[i]) for i in (xx, yy, zz)) for pt in V)
{(3, 2, 1)}
```

This is a strong hint that there should exist a substitution

$$(x,y,z) \mapsto (\omega^a x, \omega^b y, \omega^c z)$$

with $\omega^k = 1$ for some $k > 0$, which leaves the system (11.1) invariant. Such a substitution can only exist if (a,b,c) is a root modulo k of the following homogenous linear system represented by the matrix

```
sage: M = matrix([[p.degree(v) for v in (x, y, z)] for p in J.gens
   ()])
sage: M
[2 1 1]
[1 3 1]
[1 1 4]
```

The entries of the matrix M are the degrees of the generators in (11.1). Next, we calculate the determinant of M.

```
sage: M.det()
17
```

Since the $\det(M) = 17$, the matrix M projected down to $\mathbb{F}_{17}$ has vanishing determinant, i.e. $M \bmod 17$ has a kernel. This kernel element is exactly the tuple (a,b,c) we are looking for.

```
sage: M.change_ring(GF(17)).det()
0
sage: M.change_ring(GF(17)).right_kernel()
Vector space of degree 3 and dimension 1 over Finite Field of size
    17
Basis matrix:
[1 9 6]
```

In conclusion we have shown that for any $\omega \in \mathbb{C}$ with $\omega^{17} = 1$ the substitution $(x,y,z) \mapsto (\omega x, \omega^9 y, \omega^6 z)$ leaves the system (11.1) invariant. Since the equation $\omega^{17} = 1$ has exactly 17 solution and we already know that $(3,2,1)$ is a solution the solution of (11.1) is

$$\{(3\omega, 2\omega^9, \omega^6) \mid \omega^{17} = 1\}. \tag{11.2}$$

We confirm this solution by comparing the z-coordinates of the set above and of the variety V.

```
sage: w = QQbar.zeta(17); w #17th primitive root of 1
sage: Set(pt[zz] for pt in V) == Set(w^i for i in srange(17))
True
```

Another way to arrive at this parametrization is the study of the minimal polynomials of the coordinates of the points of V. Indeed, apart from the rational solution $(3,2,1)$, the common minimal polynomial of the z-coordinates is the cyclomatic polynomial Φ_{17}.

```
sage: Set(pt[zz].minpoly() for pt in V[1:])
{x^16 + x^15 + x^14 + x^13 + x^12 + x^11 + x^10 + x^9 + x^8 + x^7
   + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1}
```

Similar, the minimal polynomials of the first and the second coordinate are given by $3^{16} \cdot \Phi_{17}\left(\frac{x}{3}\right)$ and $2^{16}\Phi_{17}\left(\frac{x}{2}\right)$.

Using the parametrization of the solution set (11.2) and the exponential notation of complex numbers we obtain a short and explicit form of the solutions.

```
sage: def polar_form(z):
....:     rho = z.abs(); rho.simplify()
....:     theta = 2 * pi * z.rational_argument()
....:     return(SR(rho) * exp(I * theta))
sage: P = [tuple(polar_form(pt[i]) for i in [xx, yy, zz]) for pt
   in V]
sage: P[:2]
[(3, 2, 1), (3*e^(-14/17*I*pi), 2*e^(10/17*I*pi), e^(-16/17*I*pi))
   ]
```

Another possible approach is to simplify the ideal generated by the (11.1) itself. The fundamental tools offered by Sage are the triangular decomposition and Gröbner bases. We discuss triangular decomposition later, but a thorough introduction to Gröbner bases would be beyond the scope of this lecture. We refer to [7, Chapter 9.3] for a small description of Gröbner bases. For the moment, we simply show the effects of these simplifications on our example (11.1).

```
sage: J.triangular_decomposition()
[Ideal (z^17 - 1, y - 2*z^10, x - 3*z^3) of Multivariate
   Polynomial Ring in x, y, z over Rational Field]
sage: J.transformed_basis()
[z^17 - 1, -2*z^10 + y, -3/4*y^2 + x]
```

In both cases, we obtain the equivalent system

$$\begin{cases} z^{17} & = 1, \\ y & = 2z^{10}, \\ x & = 3z^3, \end{cases}$$

i.e. $V = \{(3\omega^3, 2\omega^{10}, \omega) \mid \omega^{17} = 1\}$ which is equivalent to the set (11.2).

What does solving mean?

In general, polynomial systems have infinite many solutions. As enumerating solutions is not always possible the best we can do is to describe the set of solutions as "exact" as possible, i.e. to compute a representation of the solution set which reveals useful properties. For example, in the case of linear systems, solutions are often represented as a vector space. But even if the solution set is finite, a parametrization of the solution set is often more useful than a plain list of the points.

In the end, we often do not want to compute the solutions itself, but to compute with the solutions to deduce that piece of information we are really interested.

11.2.2 Ideals and Systems

The main tool to solve a polynomial system are ideals of a multivariate polynomial ring. To be more concrete, if s polynomials $p_1, \ldots, p_s \in K[x]$ vanish at a common point $x = (x_1, \ldots, x_n)$ with coordinates in K or its algebraic closure, any element of the ideal J generated by these polynomials also vanishes at x. Therefore, it is natural to associate a polynomial system $p_1(x) = p_2(x) = \ldots = p_s(x) = 0$ to the ideal $J = \langle p_1, \ldots, p_s \rangle \subset K[x]$. If two polynomial systems generate the same ideal, they have the same solution set.

```
sage: R.<x, y> = QQ['x, y']
sage: J1 = R.ideal(x^2 - y); J2 = R.ideal(x^2 - y, x + y^2)
```

R *The* `ideal` *method is also available for* `InfinitePolynomialRing`*. But, the resulting objects do not have the same properties as in (finite) polynomial rings. The reason is that ideals in* $K[(x_n)_{n\in\mathbb{N}}]$ *are in general not finitely generated. Thus, most of the methods discussed in the remainder of this section do not apply to* `InfinitePolynomialRing`*.*

The method `dimension` returns the dimension of an ideal which is, roughly spoken, equivalent to the dimension of its solution space, i.e the number of parameters we need to describe the solution space. The most important case for us is the case of 0-dimensional ideals, i.e. those polynomial systems with a finite solution space.

```
sage: J1.dimension(), J2.dimension()
(1, 0)
```

The generators of an ideal J are returned by the method `gens()`. We can also access the generators directly with `J.i`, where `i` is the index of the generator in the list `J.gens()` of the generators of `J`.

```
sage: J2.gens()
[x^2 - y, y^2 + x]
sage: J2.1
y^2 + x
```

If L is a field containing K, the ***algebraic subvariety*** of L^n associated to the ideal $J = \langle p_1, \ldots, p_s \rangle$ in $K[x_1, \ldots, x_n]$ is the set

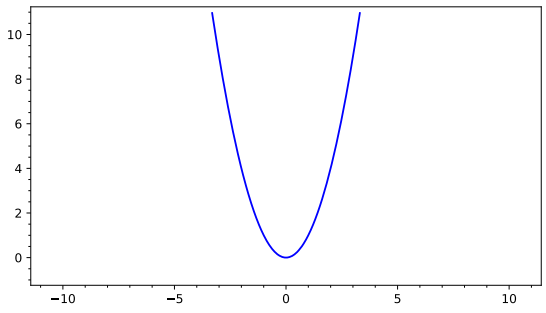
$$V_L(J) = \{x \in L^n | \forall p \in J, p(x) = 0\} = \{x \in L^n | p_1(x) = \ldots = p_s(x) = 0\}.$$

In particular, $V_L(J)$ are the solutions of the system $\{p_1(x) = 0, \ldots p_s(x) = 0\}$ with coordinates in L. Different ideals can have the same variety, e.g. $x = 0$ and $x^2 = 0$ both have the unique solution $x = 0 \in \mathbb{C}$ but $\langle x^2 \rangle \subsetneq \langle x \rangle$. Hence, the variety associated to a polynomial systems contains the solution without multiplicities while the ideal itself rather captures the notion of solutions with multiplicities. So far, the method `variety` can only be applied to zero dimensional ideals. If $J \subset K[x]$ is a zero dimensional ideal, the method `variety` returns the associated variety in K. If the variety should be calculated over another field L containing K, we can add this field as an additional argument. For the most of our applications we use $K = \mathbb{Q}$ and $L = \overline{\mathbb{Q}}$.

```
sage: V1 = J2.variety(); V2 = J2.variety(QQbar)
sage: V1
[{y: 0, x: 0}, {y: 1, x: -1}]
sage: V2
[{y: 0, x: 0}, {y: 1, x: -1}, {y: -0.50000000000000000? -
   0.866025403784439?*I, x: 0.50000000000000000? -
   0.866025403784439?*I}, {y: -0.50000000000000000? +
   0.866025403784439?*I, x: 0.50000000000000000? +
   0.866025403784439?*I}]
```

Although, Sage only calculates the variety of zero dimensional ideals, it is still possible to draw the variety of any ideal in 2 variables using the method `plot`.

```
sage: P = J1.plot()
```

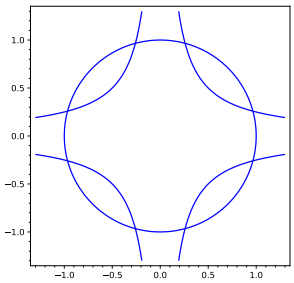


■ **Example 11.1** We consider the intersection between the unit circle $\{(x,y) \in \mathbb{R}^2 \mid x^2 + y^2 = 1\}$ and the union of two equilateral hyperbolas $\{(x,y) \in \mathbb{R}^2 \mid ax^2y^2 = 1\}$ for $a \in \{2,4,16\}$. This leads us to the three polynomial systems.

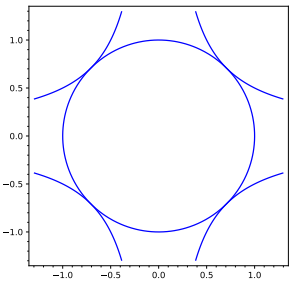
$$S_1 = \begin{cases} x^2+y^2 & = 1 \\ 16x^2y^2 & = 1 \end{cases}, \qquad S_2 = \begin{cases} x^2+y^2 & = 1 \\ 4x^2y^2 & = 1 \end{cases}, \qquad S3 = \begin{cases} x^2+y^2 & = 1 \\ 2x^2y^2 & = 1 \end{cases}.$$

To obtain a first idea of the solution spaces we plot the corresponding curves.

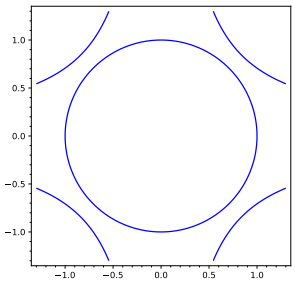
```
sage: x, y = var('x,y')
sage: s1 = implicit_plot(x^2 + y^2 - 1, (x, -1, 1), (y, -1, 1)) +
   implicit_plot(16*x^2*y^2 - 1, (x,-1.3,1.3), (y, -1.3, 1.3))
sage: s2 =  implicit_plot(x^2 + y^2 - 1, (x, -1, 1), (y, -1, 1)) +
    implicit_plot(4*x^2*y^2 - 1, (x,-1.3,1.3), (y, -1.3, 1.3))
sage: s3 =  implicit_plot(x^2 + y^2 - 1, (x, -1, 1), (y, -1, 1)) +
    implicit_plot(2*x^2*y^2 - 1, (x,-1.3,1.3), (y, -1.3, 1.3))
```



System S_1



System S_2



System S_3

We observe that S_1 has eight distinct solutions in $\mathbb{R}$, while S_2 has only four solutions in $\mathbb{R}$ but each of them has multiplicity two. The system S_3 has no real solutions, thus all of its eight solutions are in $\mathbb{C} \setminus \mathbb{R}$. To obtain the concrete coordinates of the solutions we built the corresponding ideals and calculate the associated varieties over the field of algebraic numbers to capture all solutions.

```
sage: R.<x,y> = QQ['x,␣y']
sage: J1 = R.ideal(x^2 + y^2 - 1, 16*x^2*y^2 -1);  V1 = J1.variety
   (QQbar)
sage: J2 = R.ideal(x^2 + y^2 - 1, 4*x^2*y^2 -1); V2 = J2.variety(
   QQbar)
sage: J3 = R.ideal(x^2 + y^2 - 1, 2*x^2*y^2 -1); V3 = J3.variety(
   QQbar)
sage: len(V1), len(V2), len(V3)
(8, 4, 8)
```

As expected, the length of the variety V_2 associated to S_2 is half the length of the other two varieties V_1 and V_3 associated to the systems S_1 and S_3 respectively. Taking a look onto the varieties itself we see that V_1 and V_2 only contain real solutions while all elements of V_3 are complex.

```
sage: V1 #eight real solutions
[{y: -0.9659258262890683?, x: -0.2588190451025208?}, {y:
   -0.9659258262890683?, x: 0.2588190451025208?}, {y:
   -0.2588190451025208?, x: -0.9659258262890683?}, {y:
   -0.2588190451025208?, x: 0.9659258262890683?}, {y:
   0.2588190451025208?, x: -0.9659258262890683?}, {y:
   0.2588190451025208?, x: 0.9659258262890683?}, {y:
   0.9659258262890683?, x: -0.2588190451025208?}, {y:
   0.9659258262890683?, x: 0.2588190451025208?}]
sage: V2 #four real solutions
[{y: -0.7071067811865475?, x: -0.7071067811865475?}, {y:
   -0.7071067811865475?, x: 0.7071067811865475?}, {y:
   0.7071067811865475?, x: -0.7071067811865475?}, {y:
   0.7071067811865475?, x: 0.7071067811865475?}]
sage: V3 #eight complex solutions
[{y: -0.7768869870150186? - 0.3217971264527913?*I, x:
   -0.7768869870150186? + 0.3217971264527913?*I}, {y:
   -0.7768869870150186? - 0.3217971264527913?*I, x:
   0.7768869870150186? - 0.3217971264527913?*I}, {y:
   -0.7768869870150186? + 0.3217971264527913?*I, x:
   -0.7768869870150186? - 0.3217971264527913?*I}, {y:
   -0.7768869870150186? + 0.3217971264527913?*I, x:
   0.7768869870150186? + 0.3217971264527913?*I}, {y:
   0.7768869870150186? - 0.3217971264527913?*I, x:
   -0.7768869870150186? - 0.3217971264527913?*I}, {y:
   0.7768869870150186? - 0.3217971264527913?*I, x:
   0.7768869870150186? + 0.3217971264527913?*I}, {y:
   0.7768869870150186? + 0.3217971264527913?*I, x:
   -0.7768869870150186? + 0.3217971264527913?*I}, {y:
   0.7768869870150186? + 0.3217971264527913?*I, x:
   0.7768869870150186? - 0.3217971264527913?*I}]
```

■

11.2.3 Computing Modulo an Ideal

Similar to univariate polynomial rings, we can construct quotients $K[x]/J$ of multivariate polynomial rings $K[x]$ by ideal J in Sage. We describe the usage of quotient rings using the ideal $J_1 = \langle x^2+y^2-1, 16x^2y^2-1\rangle \subset \mathbb{Q}[x,y] =: K[\mathbf{x}]$ defined in Example 11.1. To build the quotient $Q = K[x]/J_1$ we apply the method `quo` to the polynomial ring with the ideal as an argument. Similar to quotients of univariate polynomial rings, Sage introduces the new variables `xbar`, `ybar` representing the equivalence classes of x and y in the quotient Q. But these new indeterminates `xbar`, `ybar` are not automatically assigned to the corresponding Python variables. This assignment has to be done separately, e.g. with `xbar, ybar = Q.gens()`

```
sage: R.<x, y > = QQ['x,␣y']; J1 = R.ideal(x^2 + y^2 - 1, 16*x^2*y
   ^2 -1)
```

```
sage: Q = R.quo(J1); Q
Quotient of Multivariate Polynomial Ring in x, y over Rational
   Field by the ideal (x^2 - y)
sage: Q(y)
ybar
sage: ybar #Python variable ybar is not defined
Traceback (most recent call last):
...
NameError: name 'ybar' is not defined
sage: xbar, ybar = Q.gens() #assign the generators to the
   corresponding Python variables.
sage: xbar, ybar
(xbar, ybar)
```

Now, we can project elements of $K[x]$ to the quotient ring Q by simply converting them via `Q( )` as usual. Afterwards we can lift it back to an element of $K[x]$ with the `lift` method. if we apply `lift` tp an element $p \in J$, Sage rewrites p as a linear combination of the generators of J. The output is a list of the corresponding coefficients.

```
sage: p = y^4
sage: pbar = Q(p); bar
ybar^2 - 1/16
sage: pbar.lift()
y^2 - 1/16
sage: p = J1.random_element(degree = 4); p #random element of the
   ideal
97/6*x^2*y^2 + 1/6*y^4 + 28/3*x^3 + 1/2*x^2*y + 28/3*x*y^2 + 1/2*y
   ^3 - 1/6*y^2 - 28/3*x - 1/2*y - 1
sage: u = p.lift(J1); u
[1/6*y^2 + 28/3*x + 1/2*y, 1]
sage: (u[0]*J1.0 + u[1]*J1.1) == p #test
True
```

Projecting and lifting an element $p \in K[x]$ can also be done in one step using the equivalent commands `p.mod(J)` and `J.reduce(p)`. Sage further provides the command `p.reduce(J)` whose result might differ from `J.reduce(p)`. Moreover, if $J = \langle p_1, \ldots, p_n \rangle$ the commands `p.reduce(J)` and `p.reduce([p1, ..., pn])` might lead to different results. Nevertheless, their difference is an element of J.

```
sage: p.mod(J1)
y^2 - 1/16
sage: p.reduce([x^2 + y^2 - 1, 16*x^2*y^2- 1]) #NOT equivalent to
   p.mod(J1)
y^4
sage:(p.reduce([x^2 + y^2 - 1, 16*x^2*y^2- 1]) - p.reduce(J1)).mod
   (J1) #difference is an element of J1
0
```

The following theoretical issue leads to these different results: In Sage, the computation domain corresponding $K[x]/J$ has a normal form. For quotient rings of univariate polynomial rings over fields there is a canonical normal form. Since univariate polynomial rings over a field are principal every ideal J is generated by exactly one polynomial q_J. Thus, the normal form in $K[x]/J$ is given by the remainder of the Euclidean division with q_J. Unfortunately, we can not generalize this procedure to multivariate polynomial rings as they are not principal. Roughly spoken, to define a normal form for quotients of a multivariate polynomial ring $K[x]$ by an ideal J uses a particular generating system of J called ***Gröbner basis***. Sage automatically computes a Gröbner basis whenever it is required. But such a computation is quite expensive. In particular, if the numbers of variables is large the computation in quotient rings might be difficult. Moreover, one ideal can have different Gröbner basis, which is the reason for the different results in the code snippet above.

■ **Example 11.2** Having an ideal J in a multivariate polynomial ring $K[x]$ we can decompose any poynomial p into a linear combination of generators of J plus a remainder as follows.

```
sage: def decomp(R, J, p):
....:     r = p.mod(J)
....:     G = J.gens()
....:     L = (p-r).lift(J)
....:     q = str(r)
....:     for i in srange(len(G)):
....:         q = '(' + str(L[i]) + ')' +  '␣*␣' + '(' + str(G[i])
    + ')' + '␣+␣' + q
....:     return q
```

To avoid the forced transformation into the normal form we transform the linear combination into a string. Converting this string into an element of the multivariate polynomial returns our original input polynomial p.

```
sage: R.<x, y > = QQ['x,␣y']; J1 = R.ideal(x^2 + y^2 - 1, 16*x^2*y
    ^2 -1)
sage: p = R.random_element(degree = 6); p
-x^4*y^2 - 1/2*x^2*y^4 + 3*x^2*y^3 - x^3 + 2*x^2*y
s = decomp(R, J1, p); s
'(1/32*y^2␣+␣3/16*y␣-␣1/16)␣*␣(16*x^2*y^2␣-␣1)␣+␣(-x^2*y^2␣-␣x␣+␣
    2*y)␣*␣(x^2␣+␣y^2␣-␣1)␣+␣x*y^2␣-␣2*y^3␣+␣1/32*y^2␣-␣x␣+␣35/16*
    y␣-␣1/16'
sage: R(s) == p
True
```

■

11.2.4 Radical of an Ideal and Solutions

As seen in Section 11.2.1 it takes some time to solve a polynomial special. Therefore, it would be useful if we could test beforehand whether at least one solution exist or not.

One criterion for the existence of solutions follows from ***Hilbert's theorem of zeros***, also known as Nullstellensatz. In the following, let $\overline{K}$ be an algebraic closure of K.

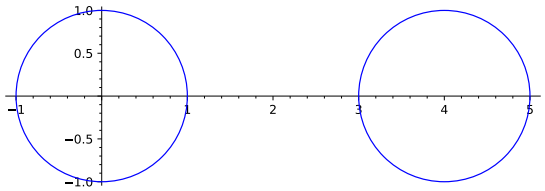
Theorem 11.2.1 Let $p_1,\ldots,p_s \in K[x_1,\ldots,x_n]$ and let $Z \subset \overline{K}^n$ be the set of common zeros. A polynomial $p \in K[x_1,\ldots,x_n]$ vanishes identically on Z if and only if there exists an integer k such that $p^k \in \langle p_1,\ldots,p_s\rangle$.

Since, the constant polynomial $1 \in K[x]$ vanishes identically on Z if any only if Z is empty, we obtain the following criterion immediately.

Corollary 11.2.2 A polynomial system $p_1(x) = \ldots = p_s(x) = 0$ has a solution in $\overline{K}$ if and only if 1 is not contained in the ideal $\langle p_1,\ldots,p_s\rangle$.

This criterion allows us to test quite fast whether a polynomial system has a solution, i.e. we can verify that the circles of radius 1 centered at $(0,0)$ and $(4,0)$ have at least one complex intersection.

```
sage: R.<x, y> = QQ['x, y']
sage: 1 in ideal(x^2 + y^2 - 1, (x-4)^2 + y^2 - 1)
False
```



But if we add the equation $x = y$, the new resulting system does not have any solution in $\overline{QQbar}$ since 1 is now an element of the associated ideal.

```
sage: 1 in ideal(x^2 + y^2 - 1, (x-4)^2 + y^2 - 1, x -y)
True
```

In terms of ideals, Hilbert's theorem on zero states that the set of polynomials vanishing on the variety $V_{\overline{K}}(J)$ associated to the ideal J is the same as the ***radical*** $\sqrt{J}$ of J, where

$$\sqrt{J} = \{p \in K[x] \mid \exists k \in \mathbb{N}, p^k \in J\}.$$

To put it in a nutshell, Hilbert's theorem on zero shows the identity $V_{\overline{K}}(J) = \sqrt{J}$. Intuitively, switching to the radical of an ideal "forgets the multiplicites" of the solutions. Moreover, an ideal J is called ***radical*** if $J = \sqrt{J}$. In particular, an ideal J is radical if all of its solution have multiplicity equal to one. In Sage, the radical of an ideal is computed with the method `radica`. Below we calculate the radicals of the ideal $J_1 = \langle x^2+y^2-1, 16x^2y^2-1\rangle$ and $J_2 = \langle x^2+y^2-1, 4x^2y^2-1\rangle$ defined in Example 11.1. It follows immediately that all zeros of J_1 have multiplicity one while at least one zero of J_2 must have higher multiplicity.

```
sage: J1 = R.ideal(x^2 + y^2 - 1, 16*x^2*y^2 - 1)
```

```
sage: J1 == J1.radical()
True
sage: J2 = R.ideal(x^2 + y^2 - 1, 4*x^2*y^2 - 1)
sage: J2.radical()
Ideal (2*y^2 - 1, 2*x^2 - 1) of Multivariate Polynomial Ring in x,
    y over Rational Field
sage: (2*y^2 - 1) in J2 #J2 ris aeal subset of its radical
False
```

11.2.5 Operations on Ideals

Instead of calculating with elements of ideals we can also perform computations with the ideals themselves. For example, we can take the sum of two ideal I and J resulting in the ideal

$$I+J = \{p+q \mid p \in I, q \in J\rangle = \langle I \cup J\rangle.$$

Geometrically, this corresponds to the intersection of the varieties, i.e. $V(I+J) = V(I) \cap V(J)$. For example, let $C = \langle x^2+y^2-1\rangle$ be the ideal whose variety is the circle $x^2+y^2=1$ and let $H = \langle 16x^2y^2-1\rangle$ be the ideal whose variety is the double hyperbole $16x^2y^2=1$. Then, the ideal $C+H$ is the same as the ideal $J_1 = \langle x^2+y^2-1, 16x^2y^2-1\rangle$ from Example 11.1. In particular, the associated variety of $C+H$ is the intersections of the varieties of C and H.

```
sage: R.<x, y> = QQ['x, y']
sage: C = R.ideal(x^2 + y^2 - 1); H = ideal(16*x^2*y^2 - 1)
sage: (C + H) == R.ideal(x^2 + y^2 - 1, 16*x^2*y^2 - 1)
True
```

Analogously the intersection, the product and the quotient of two ideals I, J can be find. Furthermore, we find for each of these operations a simple relation for the associated varieties.

$$\begin{array}{rclcrcl} I \cap J & = & \{p \mid p \in I \text{ and } p \in J\}, & & V(I \cap J) & = & V(I) \cup V(J), \\ I \cdot J & = & \{pq \mid p \in I, q \in J\}, & & V(I \cdot J) & = & V(I) \cup V(J), \\ I : J & = & \{p \mid pJ \subset I\}, & & V(I : J) & = & \overline{V(I) \setminus V(J)} \end{array}$$

These operations are computed in Sage with the commands `I.intersection(J)`, `I * J` and `I.quotient(J)` respectively.

11.3 Solving Strategies

After discussing the usage of ideals within Sage we are now in a position to use the discussed methods as a tool to characterize the solution set of a polynomial system. Here we discuss two main strategies in detail. First we describe elimination ideals which help us to characterize the solution of polynomial systems with infinitely many solutions. Then we take a closer look onto polynomial systems with finitely many solutions. There we show describe the triangular decomposition of an ideal and how this decomposition can be used to describe the structure of the solution set.

Ideals	
Constructing an Ideal $\langle p_1, p_2\rangle \subset R$	`R.ideal(p1, p2)`
Sum, Product, Power	`I + J`, `I * J`, `I^k`
Intersection $I \cap J$	`I.intersection(J)`
Quotient $I : J$	`I.quotient(J)`
Radical $\sqrt{J}$	`J.radical()`
Dimension of J	`J.dimension()`
Associated Variety $V_L(J)$	`J.variety(L)`
Quotient Ring R/J	`R.quo(J)`
Lift $R/J \to R$	`p.lift()`
Reduction modulo J	`p.mod(J)` or `J.reduce(p)`

Table 11.2.: Ideals

11.3.1 Elimination

Eliminating a variable in a system of equations means that we want to find equations satisfied by any solutions, but which do not contain the eliminated variable. This procedure makes it often easier to analyze the solutions itself. For example, we can easily eliminate the variable x from the linear system.

$$\begin{cases} 2x+y-2z &= 0, \\ 2x+2y+z &= 1. \end{cases} \tag{11.3}$$

Substracting the first equation from the second one yields the equation $y+3z=1$. Therefore, any solution (x,y,z) of (11.3) has to be of the form $(x,1-3z,z)$. Further, we observe that each partial solution $(1-3z,z)$ lifts to a (unique) solution $(\frac{5z-1}{2},1-3z,z)$ of (11.3).

In the context of polynomial systems and ideals, the "consequences" of a system $p_1(x) = \ldots = p_s(x) = 0$ are elements of the ideal $\langle p_1,\ldots,p_s\rangle$. If J is an ideal in $K[x_1,\ldots,x_n]$ the k-th ***elimination ideal*** of J is defined as the set $J_k = J \cap K[x_{k+1},\ldots,x_n]$. In particular, J_k is an ideal in $K[x_{k+1},\ldots,x_n]$ and not of $K[x_1,\ldots,x_n]$. The corresponding method `elimination_ideal` in Sage takes as input the list of variables to eliminate. But we have to pay attention, because instead of J_k the method `elimination_ideal` returns the ideal $\langle J_k\rangle$ in $K[x_1,\ldots,x_n]$ generated by J_k. Applying this method to the linear system (11.3) we obtain the following elimination ideals.

```
sage: R.<x, y, z> = QQ['x, y, z']
sage: J = R.ideal(2*x + y - 2*z, 2*x + 2*y + z - 1); J
Ideal (2*x + y - 2*z, 2*x + 2*y + z - 1) of Multivariate
   Polynomial Ring in x, y, z over Rational Field
sage: J.elimination_ideal(x)
Ideal (y + 3*z - 1) of Multivariate Polynomial Ring in x, y, z
   over Rational Field
sage: J.elimination_ideal([x, y])
Ideal (0) of Multivariate Polynomial Ring in x, y, z over Rational
    Field
```

```
sage: J.elimination_ideal(y)
Ideal (2*x - 5*z + 1) of Multivariate Polynomial Ring in x, y, z
    over Rational Field
```

Now we, as the users, have to interpret these elimination ideals in the context of our problem. The elimination ideal of x is given by $J \cap \mathbb{Q}[y,z] = \langle y+3z-1 \rangle$, i.e. we obtain the relation $y = 1-3z$. If we further eliminate y, we obtain the ideal (0). This ideal corresponds to the system associated to the trivial equation $0 = 0$, i.e. $J \cap \mathbb{Q}[z] = \mathbb{Q}[z]$, i.e. $\mathbb{Q}[z] \subset J$. Thus, we conclude that the solution set can be represented as the graph of a function of $f : \mathbb{R} \to \mathbb{R}^2$ in z. We already have the relation $y = 1-3z$. To derive the corresponding equation for x we compute the elimination ideal of y variable. The result $J \cap \mathbb{Q}[x,z] = \langle 2x-5z+1 \rangle$ gives us the desired relation $x = \frac{5z-1}{2}$. Putting everything together the solution of the linear system (11.3) is given by

$$\left\{ \left(\frac{5z-1}{2}, 1-3z, z \right) \mid z \in \mathbb{Q} \right\}.$$

Although, there are way more efficient methods to solve linear system, compare Section 10.2.2, this example illustrates the mechanics end effects of eliminating variables. Hence, we switch to the slightly more complicated system

$$S_1 = \begin{cases} x^2+y^2 &= 1, \\ 16x^2y^2 &= 1 \end{cases}$$

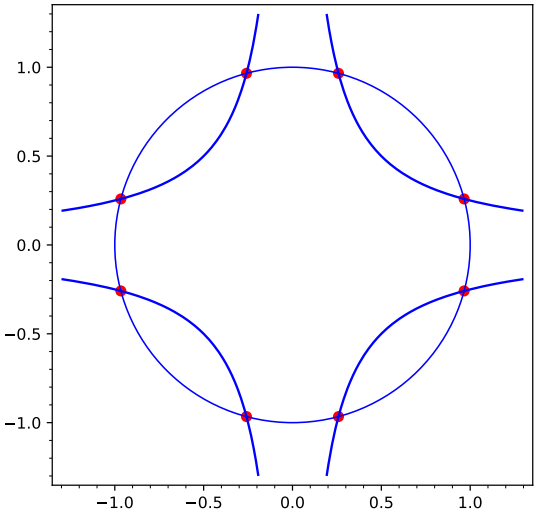
which we already know from Example 11.1. Here, eliminating y yields an ideal in $\mathbb{Q}[x]$ which is a principal ring. Therefore, the elimination ideal is generated by a single polynomial g. Its roots are the abscissas, i.e. the x-coordinates, of the eight solutions of S_1.

```
sage: R.<x,y> = QQ['x, y']
sage: J1 = R.ideal(x^2 + y^2 - 1, 16*x^2*y^2 - 1)
sage: G = J1.elimination_ideal(y)
sage: g = G.0; g #generator of G
16*x^4 - 16*x^2 + 1
sage: SR(g).solve(SR(x))
[x == -1/2*sqrt(sqrt(3) + 2), x == 1/2*sqrt(sqrt(3) + 2), x ==
    -1/2*sqrt(-sqrt(3) + 2), x == 1/2*sqrt(-sqrt(3) + 2)]
```

In the last step, we calculated the roots of g in the symbolic ring to obtain explicit expressions of x in terms of radicals. Now we can easily calculate the corresponding y-coordinates using the defining equation $y = \pm\sqrt{\frac{1}{16x^2}}$.

```
sage: X = [-1/2*sqrt(sqrt(3) + 2), 1/2*sqrt(sqrt(3) + 2), -1/2*
    sqrt(-sqrt(3) + 2), 1/2*sqrt(-sqrt(3) + 2)]
sage: L = []
sage: for i in srange(4): #list of solution points
....:     L.extend([(X[i], sqrt(1/(16*X[i]^2))), (X[i], -sqrt
    (1/(16*X[i]^2)))])
```

```
sage: x, y = var('x, y')
sage: P = circle((0,0), 1) + implicit_plot(16*x^2*y^2 - 1, (x,
    -1.3, 1.3), (y, -1.3, 1.3))
sage: P += point(L, size = 50, color = 'red')
```



In the above example, it appears that eliminating y in a system could be geometrically interpreted as the projection π of the solutions variety onto a hyperplane $y = \text{const.}$. We elaborate on this point of view by study the elimination ideals of y of the single ideals $C = \langle x^2 + y^2 - 1 \rangle$ and $H = \langle 16x^2y^2 - 1 \rangle$ separately.

```
sage: R.<x,y> = QQ['x, y']
sage: C = R.ideal(x^2 + y^2 - 1); H = R.ideal(16*x^2*y^2 - 1)
sage: C.elimination_ideal(y)
Ideal (0) of Multivariate Polynomial Ring in x, y over Rational
    Field
sage: H.elimination_ideal(y)
Ideal (0) of Multivariate Polynomial Ring in x, y over Rational
    Field
```

In both cases, the elimination of y yields the complete ideal $\mathbb{Q}[x]$.Since the coordinates solutions of an ideal in $\mathbb{Q}[x,y]$ have to elements of the algebraic closure $\overline{\mathbb{Q}}$, the elimination of y in C corresponds to the projection on the first coordinate of the solutions in $\overline{\mathbb{Q}}$. As for the circle, the equation $x^2 + y^2 = 1$ has a complex solution y^* for any fixed value of x. Thus, eliminating y in the ideal C returns us the complete space $\mathbb{C}$ explaining the output of `C.elimination_ideal(y)`. In the case of the hyperbola, the equation $16x^2y^2 = 1$ only has for ever $x \neq 0$ a complex solutions. In particular, there is no solution if $x = 0$. Nevertheless, we obtain the whole space since

$$V_{\mathbb{C}}(H \cap \mathbb{Q}[x]) = \mathbb{C} \neq \pi(V_{\mathbb{C}}(H)) = \mathbb{C} \setminus \{0\}.$$

To be more concrete, elimination does not exactly corresponds to the projection over an algebraically closed field but more to the closure of the projection.

■ **Example 11.3** We use elimination to determine the solution set of

$$\begin{cases} xyz + xz + 3y & = -3, \\ x^2yz^2 + x^2z^2 - y & = 1. \end{cases}$$

First, we take a look on the dimension of the associated ideal.

```
sage: R.< x, y, z> = QQ['x,y,z']
sage: J = R.ideal(x*y*z + x*z + 3*y + 3, x^2*y*z^2 + x^2*z^2 - y -
    1)
sage: J.dimension()
2
```

Thus, the solution set is "two dimensional", meaning, we need two parametrization variables to describe it. To obtain more information of the solution space, we calculate the elimination ideal of every variable separately.

```
sage: J.elimination_ideal(x) # obtain here constraint: y = -1
Ideal (y + 1) of Multivariate Polynomial Ring in x, y, z over
   Rational Field
sage: J.elimination_ideal(y) #without y, everything is included
Ideal (0) of Multivariate Polynomial Ring in x, y, z over Rational
    Field
sage: J.elimination_ideal(z) # obtain here constraint: y = -1
Ideal (y + 1) of Multivariate Polynomial Ring in x, y, z over
   Rational Field
```

Eliminating the variables x and z lead both to the constraint $y = -1$. Moreover, the elimination of y shows that $J \subset \mathbb{Q}[x,z]$. Hence, the solution set over $\mathbb{Q}$ is given by

$$\{(x,-1,z) | (x,z) \in \mathbb{Q}^2\}.$$

■

Applications in Plane Geometry

One application of elimination ideals is to derive a implicit equations for a parametrized subset $X \subset \mathbb{C}^k$, i.e. $X = \{(f_1(t),\ldots,f_k(t))\}$, where $f_1,\ldots f_k \in \mathbb{Q}[t_1,\ldots,t_n]$. We first construct the ideal $\langle x_1 - f_1(t),\ldots,x_k - f_k(t)\rangle$ in the multivariate polynomial ring $\mathbb{Q}[x,t]$ and then eliminate the variables $(t_1,\ldots,t_n)$. The generators of this elimination ideals are the implicit equations for the parametrized surface X. As an example this technique to the following parametrization of the circle

$$x = \frac{1-t^2}{1+t^2}, \quad y = \frac{2t}{1+t^2},$$

given by the stereographic projection. We translate these polynomial relations into an ideal in $\mathbb{Q}[x,y,t]$ and then eliminate the parametrization variable t.

```
sage: R.<x, y, t> = QQ['x,y,t']
sage: Param = R.ideal((1 - t^2) - (1 + t^2) * x, 2*t - (1 + t^2) *
    y)
```

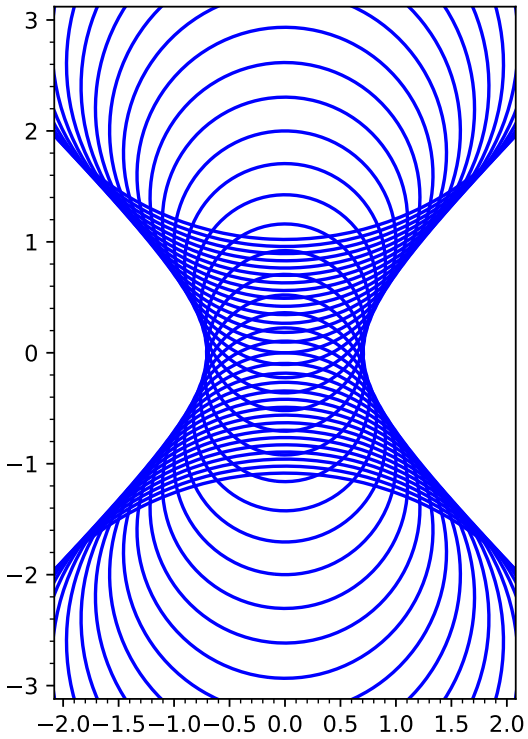
```
sage: Param.elimination_ideal(t)
Ideal (x^2 + y^2 - 1) of Multivariate Polynomial Ring in x, y, t
    over Rational Field
```

As expected, we obtain the well-known implicit equation for the circle.

A further application of elimination ideals is the determination of the implicit equation of an ***envelope*** of the family of circle $(C_t)_t$ given by the equation

$$C_t = \{(x,y) \in \mathbb{R}^2 | x^2 + (y-t)^2 = \frac{t^2+1}{2}\}.$$

```
sage: R.<x, y, t> = QQ['x, y, t']
sage: eq = x^2 + (y-t)^2 - 1/2 *(t^2 + 1)
sage: P = add((eq(t = k/5)*QQ[x,y]).plot() for k in srange(-20,
    20))
```



To be more precise, if f is a differentiable function such that

$$C_t = \{(x,y) \in \mathbb{R}^2 | f(x,y,t) = 0\}$$

for all t, the ***envelope*** of the family $(C_t)_t$ si given by the set

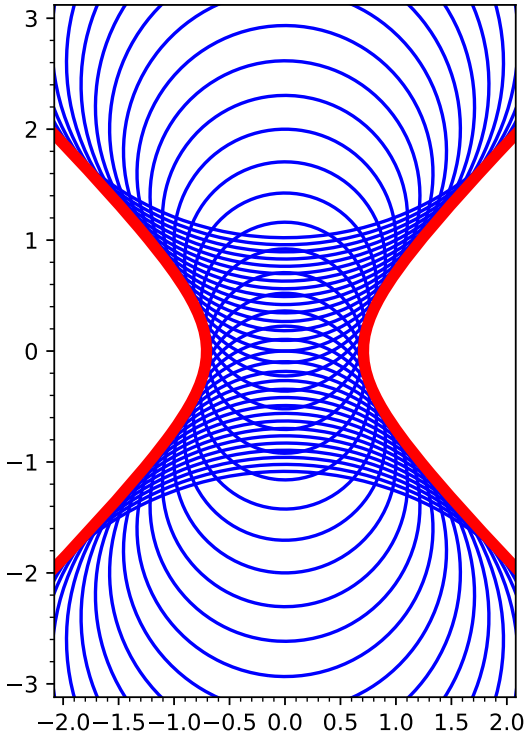
$$\{(x,y) \in \mathbb{R}^2 | \exists t \text{ s.th. } f(x,y,t) = 0 \text{ and } \partial_t f(x,y,t) = 0\}.$$

This is equivalent to eliminate the variable t from the ideal $\langle f, \partial_t f \rangle \subset \mathbb{Q}[t]$.

```
sage: env = R.ideal(eq, eq.derivative(t))
sage: env.elimination_ideal(t)
Ideal (2*x^2 - 2*y^2 - 1) of Multivariate Polynomial Ring in x, y,
    t over Rational Field
```

Thus, $2x^2 - 2y^2 = 1$ is the implicit equation describing the envelope of $(C_t)_t$.

```
sage: x,y = var('x, y')
sage: E = implicit_plot(2*x^2 - 2*y^2 - 1, (x, -3, 3), (y, -3, 3),
    color = 'red', linewidth = 5)
sage: P += E
```



In the following example we use elimination ideals to prove the well-known Euclidean geometric statement that a triangle with two equal angles has to be isosceles, i.e. having to sides of equal length. We consider the triangle with vertices $A = (0,0)$, $B = (1,0)$ and $C = (x,y)$. Next, we assume that the angles $\alpha = \sphericalangle BAC$ and $\beta = \sphericalangle CBA$ are equal. We want to show that this setup implies that the sides AC and BC have the same length. A small observation shows that the triangle ABC has to satisfy the relation $\tan(\alpha) = \frac{y}{x}$ and $\tan(\beta) = \frac{y}{1-x}$. Thus, introducing the variable $t = \tan(\alpha)$ the constraint $\alpha = \beta$ is equivalent to the solution set of the polynomial system

$$\begin{cases} y &= tx, \\ y &= t(1-x). \end{cases}$$

In this setting, the statement that a triangle with two equal angle has to be isosceles is equivalent to show that the solutions set of the above systems imply the equation

$x^2+y^2=(1-x)^2+y^2$, which is simply the equality of the lengths of the sides AC and BC. In the language of ideals, this is the case if and only if

$$x^2+y^2-(1-x)^2-y^2 \in \langle y-tx, y-t(1-x)\rangle$$

Hence, it only remains to build this ideal in Sage and check whether the above statement is true or not.

```
sage: R.<x, y, t> = QQ['x, y, t']
sage: J = R.ideal(y- t*x, y - t*(1-x)); p = x^2 + y^2 - (1-x)^2 -
   y^2
sage: p in J
False
```

What went wrong? The problem here is that we have not excluded the case of flat triangles, i.e. the case $t=0$ which is equivalent to $\alpha=\beta=0$. In that case, the equality of the base angles does not imply that the triangle is isosceles. Thus, we have to exclude the case $t=0$. Here, the trick is to introduce an auxiliary variable u and add the constraint $tu=1$. Then $t=0$ is not a possible solution anymore. Now, Sage returns us the desired result.

```
sage: R.<x, y, t, u> = QQ['x, y, t, u']
sage: J = R.ideal(y- t*x, y - t*(1-x), t*u -1); p = x^2 + y^2 -
   (1-x)^2 - y^2
sage: p in J
tTue
```

11.3.2 Zero-Dimensional Systems

Apart the computation of elimination ideals, Sage does not provide many other relevant tools to solve general polynomial systems. However, if we know that the ideal associated to our polynomial system is zero-dimensional, i.e. that the solution set is finite, Sage provides more operations and possibilities to study the solution set.

To be more precise, an ideal $K \subset K[x]$ is ***zero-dimensional*** if the quotient $K[x]/J$ has the structure of a finite dimensional K-vector space. For algebraically closed fields K this is equivalent to $V(J)$ being finite. For example, the ideal J_1 associated to the system S_1 discussed in Example 11.1 has dimension 0 while the ideal $\langle (x^2+y^2)(x^2+y^2+1)\rangle$ in $\mathbb{Q}[x,y]$ has dimension 1.

```
sage: R.<x,y> = QQ['x, y']
sage: J1 = R.ideal(x^2 + y^2 - 1, 16*x^2*y^2 - 1)
sage: J1.dimension()
0
sage: J = R.ideal((x^2 + y^2) * (x^2 + y^2 + 1))
sage: J.dimension()
1
```

If a polynomial system only has finitely many solutions we can calculate them exactly or approximately. In Section 11.2.2 we introduced the command `J.variety(L)` which

computes the variety $V_L(J)$ of a zero-dimensional ideal J over the field L. In place of L we can insert any field containing K. The most important case for our applications is $K = \mathbb{Q}$ and $L = \overline{\mathbb{Q}}$. In this setting, it is always possible to compute exactly the complete complex variety $V_{\mathbb{C}}(J) = V_{\overline{\mathbb{Q}}}(J)$ of a zero-dimensional ideal $J \subset \mathbb{Q}[x]$.

```
sage: J1.variety(QQbar)
[{y: -0.9659258262890683?, x: -0.2588190451025208?},
 {y: -0.9659258262890683?, x: 0.2588190451025208?},
 {y: -0.2588190451025208?, x: -0.9659258262890683?},
 {y: -0.2588190451025208?, x: 0.9659258262890683?},
 {y: 0.2588190451025208?, x: -0.9659258262890683?},
 {y: 0.2588190451025208?, x: 0.9659258262890683?},
 {y: 0.9659258262890683?, x: -0.2588190451025208?},
 {y: 0.9659258262890683?, x: 0.2588190451025208?}]
```

Internally, `J.variety(L)` goes through a ***triangular decomposition***. This decomposition gives us another description of the variety $V_L(J)$ which might be better for further computations or easier to grasp, see for example the polynomial system discussed in Section 11.2.1. A polynomial system is called ***triangular*** if it is of the following form:

$$\begin{cases} p_1(x_1) & := & x_1^{d_1} + a_{1,d_1-1}x_1^{d_1-1} + \cdots + a_{1,0} & = 0 \\ p_2(x_1,x_2) & := & x_2^{d_2} + a_{2,d_2-1}(x_1)x_2^{d_2-1} + \cdots + a_{2,0}(x_1) & = 0 \\ & \vdots & & \\ p_n(x_1,\ldots,x_n) & := & x_n^{d_n} + a_{n,d_n-1}(x_1,\ldots,x_{n-1})x_n^{d_n-1} + \cdots & = 0 \end{cases}$$

In a triangular polynomial system each polynomial p_i is a monic univariate polynomial in x_i with coefficients in $K[x_1,\ldots,x_{i-1}]$. To solve a triangular polynomial system we first calculate the roots x_1^* of p_1. Then we insert these roots into p_2. Since $p_2(x_1^*)$ is a monic univariate polynomial in x_2 we can calculate its roots x_2^*. Next, we insert x_1^*, x_2^* into p_3 and repeat this procedure, until we have calculated the roots x_n^* of the polynomial $p_n(x_1^*,\ldots,x_{n-1}^*)$. Sage provides the method `triangular_decomposition` for zero-dimensional ideals.

```
sage: R.<x,y,z> = QQ['x, y, z'] #example from the beginning
sage: J = R.ideal(x^2*y*z - 18, x*y^3*z - 24, x*y*z^4 - 6)
sage: V = J.variety(QQbar) #solutions in the field of algebraic
   numbers
sage: V[:2], len(V)
([{z: 1, y: 2, x: 3},
 {z: -0.9829730996839017? - 0.1837495178165704?*I, y:
    -0.5473259801441657? + 1.923651286345639?*I, x:
    -2.550651407188843? - 1.579296488632068?*I},
 {z: -0.9829730996839017? + 0.1837495178165704?*I, y:
    -0.5473259801441657? - 1.923651286345639?*I, x:
    -2.550651407188843? + 1.579296488632068?*I}], 17)
sage: J.triangular_decomposition()
[Ideal (z^17 - 1, y - 2*z^10, x - 3*z^3) of Multivariate
   Polynomial Ring in x, y, z over Rational Field]
```

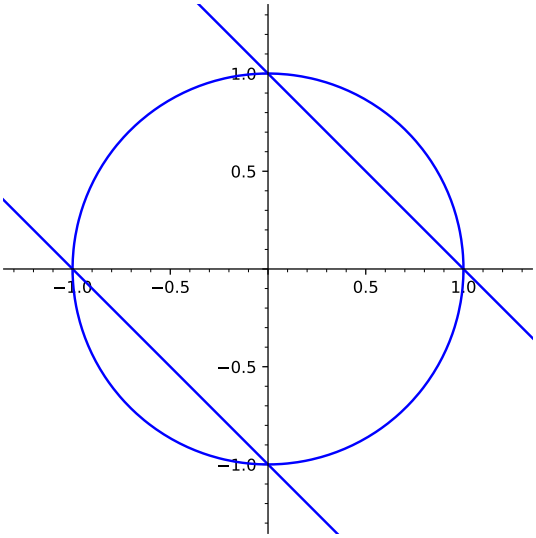
Both methods `J.variety(QQbar)` and `J.triangular_decomposition()` lead to same set of exact solutions. But while `V` is only a plain list of points the triangular decomposition shows us that the solution set can be parametrized as

$$V = \{(3z^3, 2z^10, z) | z^17 = 1\}.$$

This representation reveals more properties of the solution set than the list stored in `V`.

However, not every zero-dimensional ideal has a triangular decomposition. As an example we consider the ideal $J = C + D$ in $\mathbb{Q}[x,y]$ which corresponds to the intersection of the circle $C = \langle x^2 + y^2 - 1 \rangle$ and the union of two lines $D = \langle (x+y-1)(x+y+1) \rangle$.

```
sage: R.<x, y> = QQ['x, y']
sage: C = R.ideal(x^2 + y^2 - 1)
sage: D = R.ideal((x + y - 1) * (x + y + 1))
sage: J = C + D
sage: P = C.plot()+ D.plot()
```



Here $V(J)$ contains two points with $x = 0$ but only one point with $x = -1$, and similarly only one point $y = -1$ but two points with $y = 0$. Thus, J can not be described by a triangular system.

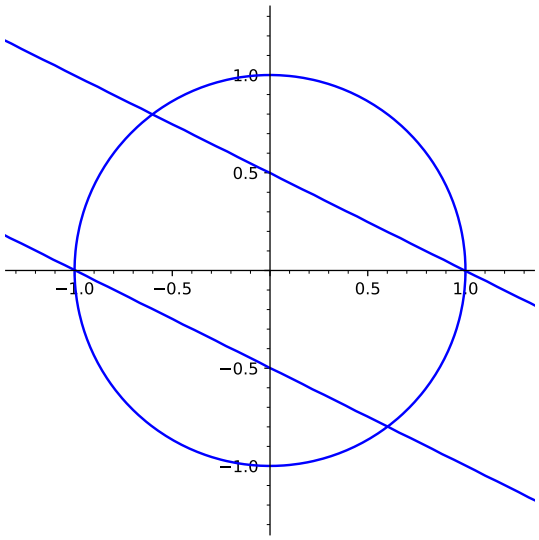
Nevertheless, every zero-dimensional ideal can be written as a finite intersection of ideals equivalent to a triangular system. If this is the case, the method `triangular_decomposition` returns the list of the involved triangular systems.

```
sage: J.triangular_decomposition()
[Ideal (y, x^2 - 1) of Multivariate Polynomial Ring in x, y over
   Rational Field,
 Ideal (y^2 - 1, x) of Multivariate Polynomial Ring in x, y over
    Rational Field]
```

The output shows that J is the intersection of the two triangular ideal $T_1 = \langle y, x^2 - 1 \rangle$ and $T_2 = \langle y^2 - 1, x \rangle$. Hence, we obtain the solutions $(1,0)$ and $(-1,0)$ from the first triangular ideal T_1 and the solutions $(0,1), (0,1)$ from the second triangular ideal T_2.

Instead of using the triangular decomposition, we can also obtain the solutions with elimination ideals introduced in Section 11.3.1. Starting with a zero dimensional system it is always possible to find a univariate polynomial whose roots are exactly the first coordinates of the solutions by computing some elimination ideals. Then substituting these roots decreases the number of variables and then we can determine a univariate polynomial whose roots are the second coordinates and so on. Hence, iterating this process, we also find the solutions after finitely many steps. However this substitution might cause problems. To illustrate this problematic we slightly modify the above example by tilting the two lines.

```
sage: D = R.ideal((x + 2*y - 1) * (x + 2*y + 1)); J = C + D
sage: P = C.plot()+ D.plot()
```




```
sage: J.variety()
[{y: 0, x: 1}, {y: 0, x: -1}, {y: 4/5, x: -3/5}, {y: -4/5, x:
   3/5}]
sage: J.triangular_decomposition()
[Ideal (y, x^2 - 1) of Multivariate Polynomial Ring in x, y over
   Rational Field,
 Ideal (25*y^2 - 16, 4*x + 3*y) of Multivariate Polynomial Ring in
     x, y over Rational Field]
```

Again, the triangular decomposition returns us ideals of which we can easily read off the solutions. Now, we try to solve this system using elimination ideals. First we eliminate x.

```
sage: Jy = J.elimination_ideal(x); Jy.gens()
[25*y^3 - 16*y]
```

Next, we substitute the roots of this polynomial into the polynomial defining the ideal J, e.g. x^2+y^2-1.

```
sage: ys = QQ['y'](Jy.0).roots(); ys
[(4/5, 1), (0, 1), (-4/5, 1)]
sage: QQ['x'](J.1(y = ys[0][0])).roots()
[(-3/5, 1), (-13/5, 1)]
```

Here, we obtain the same solution as with the triangular decomposition but also the additional solution $\left(-\frac{13}{5}, \frac{4}{5}\right)$ which is wrong. It follows that in the elimination process, it might not be sufficient to insert the obtained roots only in one generator of the ideal. Instead, we have to test every generator to eliminate the wrong solutions.

This problematic might get worse, if we want to solve equations numerically. Although calculating with algebraic numbers is computationally expensive, switching to numerical calculations means that we can not check for "true equality anymore" as all objects are only approximated. For example, using the complex double field `CDF`, i.e. complex floating point numbers with 64-bits precision, we obtain the following solutions of the above system J.

```
sage: ys = CDF['y'](Jy.0).roots(); ys
[(-0.8000000000000002, 1), (0.0, 1), (0.8, 1)]
sage: [CDF['x'](p(y = ys[0][0])).roots() for p in J.gens()]
[[(-0.5999999999999998, 1), (0.5999999999999999, 1)],
 [(0.6000000000000001, 1), (2.6, 1)]]
```

Here, the substitution of $y \simeq 0.8$ yields three values of x near 0.6. But, how can we check, whether these approximated values also approximate an exact solution or not, without knowing the exact solutions beforehand? This phenomena gets trickier as the number of variables and equations grows. When dealing with numerical approximations it is recommended to work with triangular systems. One reason is that in each substitution step only one monic polynomial equation has to be considered. Hence, solving these equations numerically does not change the number of solutions. Nevertheless, the values still might be not near the exact solutions. This phenomena can be observed when we try to compute the variety of $J = \langle x^7 - (100x-1)^2, y - x^7 + 1\rangle$ over the real floating point numbers with 50-bits precision. The command `J.variety(RealField(50))` first computes an exact triangular decomposition of J and then finds the real solutions numerically .

```
sage: R.<x, y> = QQ['x,␣y']; J = R.ideal([x^7 - (100*x-1)^2, y - x
    ^7 + 1])
sage: J.variety(RealField(50))
[{y: -1.0000000249450, x: -3563.5557925117},
 {y: -0.99999997505503, x: 3563.5757038914},
 {y: 396340.89016654, x: 160.71370521084}]
```

However, if we perform the calculations exactly until the end, we see that the numerical approximations are far off the exact solutions.

```
sage: J.variety(AA)
[{y: -0.999999999999990?, x: 0.00999999900000035?},
 {y: -0.999999999999990?, x: 0.01000000100000035?},
 {y: 396340.8901665450?, x: 6.305568998641385?}]
```

As mentioned in the beginning of this section, if $J \subset K[x]$ is a zero-dimensional ideal, the quotient $K[x]/J$ has the structure of a K-vector space. The vector space dimension $\dim_K K[x]/J$ does not only provide an upper bound for the cardinality of the variety $V(J)$ but can be interpreted as the number of solutions "with multiplicities". This dimension is calculated in Sage with the method `vector_space_dimension`. Moreover, Sage provides the method `normal_basis` which returns a list of monomials whose projection to $K[x]/J$ constitute a basis. As an example, we consider the polynomial system

$$S_2 = \begin{cases} x^2+y^2 & = 1, \\ 4x^2y^2 & = 1 \end{cases}$$

from Example 11.1. As already discussed, this system has four solutions, each of them with multiplicity two. Hence, the length of the variety $V_{\mathbb{C}}(J_2)$ is four while the vector space dimension of J_2 is eight.

```
sage: R.<x,y> = QQ['x, y']
sage: J2 = R.ideal(x^2 + y^2 - 1, 4*x^2*y^2 - 1)
sage: len(J2.variety(QQbar)), J2.vector_space_dimension()
(4, 8)
sage: J2.normal_basis()
[x*y^3, y^3, x*y^2, y^2, x*y, y, x, 1]
```

Two explicit examples

We close this chapter by illustrating the explicit solution strategy for two polynomial systems. First, we consider the polynomial system

$$\begin{cases} x^2-1 & = 0, \\ y^2-1 & = 0, \\ (x-1)(y-1) & = 0. \end{cases}$$

To solve this system we first check the dimension of the associated ideal.

```
sage: R.<x,y> = QQ['x, y']
sage: J = R.ideal(x^2 - 1, y^2 - 1, (x-1)*(y-1))
sage: J.dimension()
0
```

Hence, there are only finitely many solutions. Thus, we calculate its variety to determine the solutions and then its vector space dimension to detect any multiplicites.

```
sage: J.variety()
[{y: 1, x: 1}, {y: 1, x: -1}, {y: -1, x: 1}]
sage: len(J.variety()), J.vector_space_dimension()
(3, 3)
```

Since $V(J)$ contains three explicit solutions and the vector space dimension of J is also equal to three it follows immediately that the solution set is $\{(1,1),(-1,1),(1,-1)\}$. In particular, the multiplicity of each solution is equal to one.

Next, we solve this following more complex system.

$$\begin{cases} vy - 6u - 2xy + 5v + 2x & = 0, \\ ux - 3u - 3xy + 3v + 2x & = 0, \\ u - x^2 & = 0, \\ v - y^2 & = 0 \end{cases}$$

Again, we first calculate the dimension of the associated ideal to determine the further strategy.

```
sage: R.<u, v, x, y> = QQ['u, v, x, y']
sage: J = R.ideal(v*y - 6*u - 2*x*y + 5*v + 2*x, u - x^2, v - y^2,
    u*x - 3*u - 3*x*y + 3*v + 2*x)
sage: J.dimension()
0
```

Since J is zero-dimensional we calculate the variety over the field of algebraic numbers and compare its length with the vector space dimension of J.

```
sage: J.variety(QQbar)
[{y: 0, x: 0, v: 0, u: 0},
  {y: 2, x: 2, v: 4, u: 4},
  {y: 1, x: 1, v: 1, u: 1},
  {y: 1, x: -1, v: 1, u: 1},
  {y: -2, x: -1, v: 4, u: 1},
  {y: 0.5071118397992805?, x: -0.4105844881290678?, v:
     0.2571624180646111?, u: 0.1685796218922086?},
  {y: -8.75355591989964? - 1.501107075408794?*I, x:
     4.205292244064534? - 6.385042887008311?*I, v:
     74.37141879096769? + 26.28004945269577?*I, u:
     -23.08428981094611? - 53.70194266151094?*I},
  {y: -8.75355591989964? + 1.501107075408794?*I, x:
     4.205292244064534? + 6.385042887008311?*I, v:
     74.37141879096769? - 26.28004945269577?*I, u:
     -23.08428981094611? + 53.70194266151094?*I}]
sage: len(J.variety(QQbar)), J.vector_space_dimension()
(8, 9)
```

We observe that there are 5 rational and 3 irrational solutions Moreover, since the vector space dimension of J is one larger than the length of the variety $V(J)$ there has to be one solution with multiplicity two. Although we already have a complete list of the exact solutions, we do not know much about the structure. In particular, we want to answer the following questions: What is the origin of the irrational solutions and which solution has higher multiplicity? A good starting point is the triangular decomposition of J.

```
sage: J.triangular_decomposition()
[Ideal (y - 1, x^2 - 1, v - 1, u - 1) of Multivariate Polynomial
   Ring in u, v, x, y over Rational Field,
```

```
Ideal (y^7 + 17*y^6 + 66*y^5 - 108*y^4 - 280*y^3 + 160*y^2, 3840*
    x - 19*y^6 - 345*y^5 - 1608*y^4 + 660*y^3 + 6256*y^2, v - y
    ^2, 3840*u + y^6 + 3*y^5 - 72*y^4 - 732*y^3 - 2128*y^2) of
    Multivariate Polynomial Ring in u, v, x, y over Rational
    Field]
```

Sage tells us that $J$is the intersection of the triangular ideals

$$\begin{aligned}
T_1 &= \langle y-1, x^2-1, v-1, u-1\rangle,\\
T_2 &= \langle y^7+17*y^6+66*y^5-108*y^4-280*y^3+160*y^2,\\
&\qquad 3840*x-19*y^6-345*y^5-1608*y^4+660*y^3+6256*y^2, v-y^2,\\
&\qquad 3840*u+y^6+3*y^5-72*y^4-732*y^3-2128*y^2\rangle.
\end{aligned}$$

T_1 yields the two solutions $(1,1,1,1)$ and $(-1,1,1,1)$. Hence, the complex solutions are contributed by T_2. Since there are also 3 other rational solutions missing we first factorize the first generator, i.e. the univariate polynomial $y^7+17y^6+66y^5-108y^4-280y^3+160y^2$.

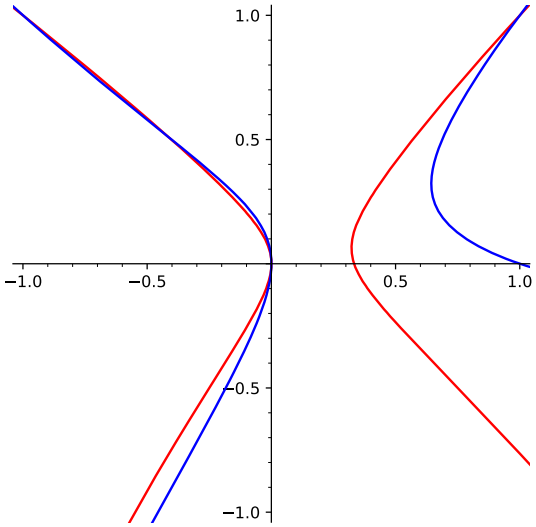
```
sage: p = y^7 + 17*y^6 + 66*y^5 - 108*y^4 - 280*y^3 + 160*y^2
sage: QQ['y'](p).factor()
(y - 2) * (y + 2) * y^2 * (y^3 + 17*y^2 + 70*y - 40)
```

This factorization shows us various things. First, that the remaining three rational solutions arise by inserting the roots $p = y_3 = -2, y_4 = 2$ and $y_5 = 0$ into the equations

$$\begin{aligned}
x &= \frac{1}{3840}\left(+19*y^6+345*y^5+1608*y^4-660*y^3-6256*y^2\right)\ u \quad = x^2,\\
v &= y^2.
\end{aligned}$$

Next, we observe that the y-coordinates of the irrational solutions are the roots of the polynomial $y^3+17*y^2+70*y-40$. Solving these polynomial over the symbolic ring leads to three exact expressions of the solutions. However, they are not really enlightening. Thus, we stop at this point with the study of the irrational solutions. Last but not least the factor y^2 in the factorization of p shows us that the solution corresponding to $y = 0$ has to be the solution with multiplicity two. This can be made visible by taking a look on the plot corresponding to the J. To draw the corresponding plot in two dimensions we use the substitutions $u = x^2, v = y^2$ and draw the implicit plot of the remaining two equations. The plot shows us that $(0,0,0,0)$ is indeed the solution of higher multiplicity.

```
sage: p = v*y - 6*u - 2*x*y + 5*v + 2*x
sage: p = p.subs(u= x^2, v = y^2)
sage: q = u*x - 3*u - 3*x*y + 3*v + 2*x
sage: q = q.subs(u = x^2, v = y^2)
sage: P = implicit_plot(SR(p), (x, -2.1, 2.1), (y, -2.1, 2.1),
    color = 'red')
sage: Q = implicit_plot(SR(q), (x, -2.1,2.1), (y, -2.1, 2.1))
sage: S = P + Q
```


1.0
0.5
−1.0
−0.5
0.5
1.0
−0.5
−1.0

12. Differential Equations

In this chapter, we explain how to solve differential equations symbolically and numerically with Sage. We need to keep in mind that, in the mathematical sense, a solutions of a differential equation is a differentiable function defined over a certain interval. But Sage views differential equations just as symbolic expressions which can be manipulated without defining a definition domain. Thus, we need to figure out the definition domain ourselves. Henceforth we use the following notation and declaration.

An ***ordinary differential equation (ODE)*** is an equation involving an unknown function of a single variable, as well as one or more derivatives of the unknown function For example, in the equation

$$y'(x) + x \cdot y(x) = \sin(x)$$

the unknown function y is called the ***dependent variable*** and the variable x (relative to which y varies) is called the ***independent variable.***

A ***partial differential equation (PDE)*** involves several independent variables as well as the partial derivatives of the dependent variable with respect to these independent variables.

12.1 First Order Ordinary Differential Equations

Before we can solve a first order ODE $F(x, y(x), y'(x)) = 0$ in Sage, we have to define the independent variable x and the function y depending on it.

```
sage: x = var('x')
sage: y = function('y')(x); y
y(x)
```

These definitions are needed to use the basic differential solving command in Sage:

`desolve(equation, variable, ics = ..., ivar = ..., show_method = ..., contrib_ode = ... )`,
where

- `equation` is the differential equation. As usual, an equality is designated by ==. For instance, the equation $y' = 2y + x$ is written as `diff(y, x) == 2*y + x`;
- `variable` is the dependent variable, e.g. y in $y' = 2y + x$;
- `ics` is optional and stands for ***initial conditions***. For a first order equation this is a known value $y(x_0) = y_0$, written as `[x0, y0]` and for a second order equation this would be either two known values $y(x_0) = y_0, y(x_1) = y_1$, written as `[x0, y0, x1, y1]`, or a known value together with a known derivative at one point $y(x_0) = y_0, y'(x_0) = y'_0$, written as `[x0, y0, y0]`;
- `ivar` is optional and stands for ***independent variable***, e.g. x in $y' = 2y + x$. The independent variable must be specified if the differential equation involves parameters as in $y' = ay + bx$;
- `show_method` is an optional boolean whose set to `False` by default. If set `True` Sage returns a pair `[solution, method]`, where the method is a string describing the used method. Some of the available methods are `linear`, `separable`, `exact`, `homogeneous`, `bernoulli`, `laplace`, `clairaut`, `ricatti`. `contrib_ode` is an optional boolean set to `False` by default. Setting it `True`, we can also solve Clairaut, Lagrange, Ricatti, and some other equations. As the solving of these kind of differential equations might have a large computation time, this options is turned off by default.

If no initial condition is specified, Sage introduces a new variable `_C` which represents an arbitrary constant $C \in \mathbb{R}$.

```
sage: S = desolve(diff(y, x) == 2*y + x, y); S
-1/4*((2*x + 1)*e^(-2*x) - 4*_C)*e^(2*x)
```

Hence, the function

$$y(x) = -\frac{1}{4}\left((2x+1)e^{-2x} - 4C\right),$$

where $C \in \mathbb{R}$ is arbitrary, solves the differential equation $y' = 2y + x$. The constant C is fixed if we add an initial condition, e.g. $y(0) = 0$.

```
sage: desolve(diff(y, x) == 2*y + x, y, ics = [0, 0])
-1/2*x + 1/4*e^(2*x) - 1/4
```

Thus, the solution of the initial value problem $y' = 2y + x, y(0) = 0$, is given by

$$y(x) = -\frac{1}{2}x + \frac{1}{4}e^{2x} - \frac{1}{4}. \tag{12.1}$$

At this point, we want to remark, that the introduced variable `_C` is only a displayed variable of Sage which has not been assigned to any Python variable yet.

```
sage: _C
Traceback (most recent call last):
...
NameError: name '_C' is not defined.
```

To assign `_C` to a Python variable, we first apply the method `variables` to the solution to obtain a list of all used variables. Afterwards, we can assign them to `Python` variables.

```
sage: S.variables()
(_C, x)
sage: _C = S.variables()[0] #assigne _C to the corresponding
   Python variable
```

Now, we can substitute the constant `_C` by some explicit value of our choice, using the method `substitute` as usual, see e.g. Section 3.1.2. For example, we can insert $C = \frac{1}{4}$ which gives us the same solution as above.

```
sage: S.substitute(_C = 1/4)
-1/4*((2*x + 1)*e^(-2*x) - 1)*e^(2*x)
```

Although the solution is equivalent to (12.1) its representation differs. The reason is that `Sage` solves differential equations over the symbolic ring which is a domain without a designated normal form, see Section 8.4 for more details. We can obtain the same expression by transforming it manually with the methods discussed in Section 3.1.3 and in Section **??**. For example, to obtain the same representation as above, we can apply the `expand` method.

```
sage: S.substitute(_C == 1/4).expand()
-1/2*x + 1/4*e^(2*x) - 1/4
```

12.1.1 Types of First Order ODEs

First order ODEs are classified into different types. These types determines the solution method used internally in `Sage` to solve them. Below we shortly describe the various types and discuss along explicit examples how to solve them with `Sage`.

Linear Equations

A first order ODE is called ***linear*** if the equation is linear in the dependent variable y and its derivatives. Hence, a linear first order ODE is of the form

$$y' + P(x)y = Q(x)$$

for continuous functions P and Q on a given interval. These differential equations are quite easy to solve. Using `Sage`, the solution is almost always given explicitly. For example, we can solve the linear equation $y' + 3y = e^x$.

```
sage: x = var('x'); y = function('y')(x)
sage: desolve(diff(y, x) + 3*y == exp(x), y, show_method = True )
[1/4*(4*_C + e^(4*x))*e^(-3*x), 'linear']
```

Nevertheless, since we are working in the symbolic ring, the solution function is not always displayed in its simplest form. Thus, we might have to do further transformations

manually before we obtain a convenient form of the solution function. This is, for example, the case when we want to solve

$$y' + 2y = x^2 - 2x + 3.$$

```
sage: S = desolve(diff(y, x) + 2*y == x^2 - 2*x + 3, y)
sage: S
1/4*((2*x^2 - 2*x + 1)*e^(2*x) - 2*(2*x - 1)*e^(2*x) + 4*_C + 6*e
   ^(2*x))*e^(-2*x)
```

This solution takes a much shorter and more readable form after applying the `expand` method.

```
sage: S.expand()
1/2*x^2 + _C*e^(-2*x) - 3/2*x + 9/4
```

It is also possible to obtain the expanded solution directly with `desolve(...).expand()`.

```
sage: desolve(diff(y, x) + 2*y == x^2 - 2*x + 3, y).expand()
1/2*x^2 + _C*e^(-2*x) - 3/2*x + 9/4
```

Bernoulli Equations

Bernoulli equations are first order ODEs of the form

$$y' + P(x)y = Q(x)y^{\alpha},$$

where P and Q are continuous functions over a given interval and $\alpha \notin \{0,1\}$, e.g. $y' - y = xy^4$.

```
sage: x = var('x'); y = function('y')(x)
sage: desolve(diff(y, x) - y == x*y^4, y)
e^x/(-1/3*(3*x - 1)*e^(3*x) + _C)^(1/3)
```

For the excluded values $\alpha \in \{0,1\}$ the resulting differential equation is a linear equation which we have discussed above.

Separable Equations

A first order ODE is ***separable*** if we can separate the dependent and the independent variable, i.e. if the differential equation is of the form

$$y'Q(y) = P(x)$$

for continuous functions P and Q defined over a given interval. One of the simplest separable equation is given by $y'y = x$. We try to solve this differential equation with Sage as usual.

```
sage: x = var('x'); y = function('y')(x)
sage: desolve(y*diff(y, x) == x, y, show_method = True)
[1/2*y(x)^2 == 1/2*x^2 + _C, 'separable']
```

In contrast to the examples before, the solution function y is not given explicitly anymore, but as the solution of the equation

$$\frac{1}{2}y(x)^2 = \frac{1}{2}x^2 + C.$$

In this case it is not hard to see that the solution is given by

$$y(x) = \sqrt{x^2 + 2C}$$

as long as long as $x^2 + 2C > 0$. Since the function `y(x)` is treated as a symbolic expression and not as a function with a specified domain, we need to take care of the right domains ourselves.

R *Sometimes, it happens that `Sage` solves separable equations as exact, see Section 12.1.1, e.g.* $y' = e^{x+y}$

```
sage: desolve(diff(y, x) == exp(x + y), y, show_method = True)
[-(e^(x + y(x)) + 1)*e^(-y(x)) == _C, 'exact']
```

The example above showed that the solution function y might not be given explicitly but implicitly as the solution of an equation. In that case, we have to modify the solution further by ourselves to obtain an explicit expression for the solution function y. To illustrate this procedure we try to solve the following differential equation with `Sage`:

$$y' \log(y) = y \sin(x). \tag{12.2}$$

This is a separable equation since we can divide by y, if we assume $y \neq 0$. First, we take a look on the result of the `desolve` command.

```
sage: desolve(diff(y, x)*log(y) == y*sin(x), y, show_method = True
    )
[1/2*log(y(x))^2 == _C - cos(x), 'separable']
```

Again, we do not obtain y directly but only the relation.

$$\frac{1}{2}\log(y)^2 - \cos(x) = C.$$

As we have to work further with this equation we store it in a `Python` variable.

```
sage: ed = desolve(diff(y, x)*log(y) == y*sin(x), y)
```

Since we are interested in the solution y of this equation we use the `solve` command, i.e. we treat this equation like a symbolic equation where y is "just" avariable.

```
sage: solve(ed, y)
[
y(x) == e^(-sqrt(2*_C - 2*cos(x))),
y(x) == e^(sqrt(2*_C - 2*cos(x)))
]
```

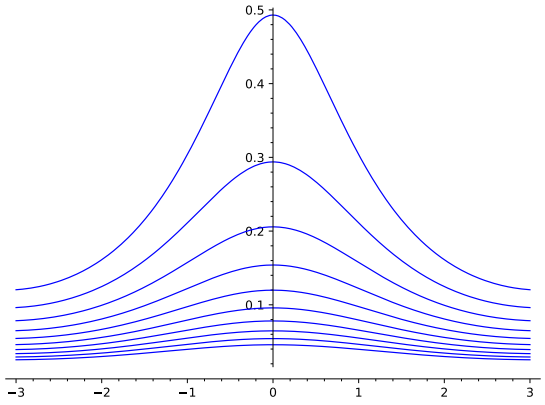
Now we have two possible solutions. Moreover, we have to choose C such that $\sqrt{2C-2\cos(x)}$ is nonnegative, i.e. $C \geq 1$.

To draw the graph of various solution of the differential equation (12.2), we have to evaluate the solution function y for different values of C. First, we need to assign the introduced variable `_C` to a `Python` variable. Moreover, to avoid unnecessary computations, we assign the two solutions to separable `Python` variables.

```
sage: C = ed.variables()[0]
sage: S1 = solve(ed, y)[0].rhs(); S2 = solve(ed, y)[1].rhs()
```

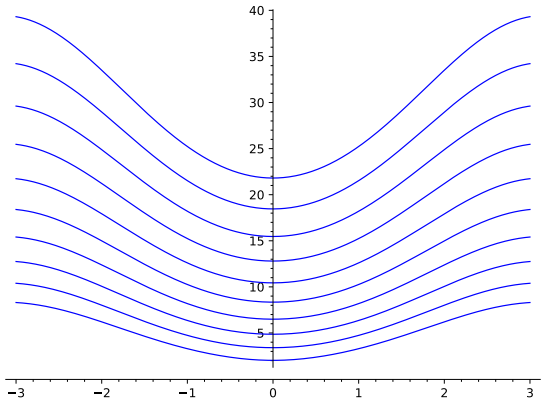
Using a `for` loop we plot a family of solution curves for the first solution S_1.

```
sage: P = Graphics()
sage: for k in range(1, 20, 2):
....:     P += plot(S1.substitute(C == 1 + k/ 4), x, -3, 3)
```

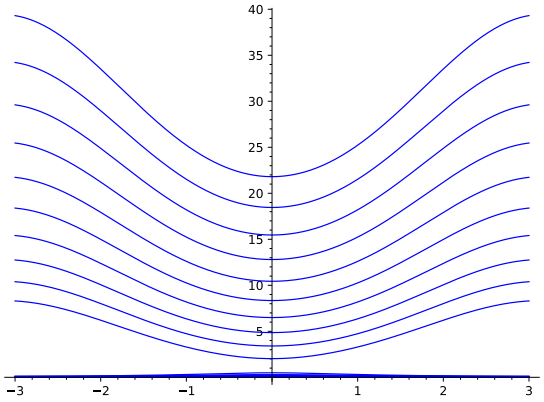


The same can be done for the second solution S_2.

```
sage: Q = Graphics()
sage: for k in range(1, 20, 2):
....:     Q += plot(S2.substitute(C == 1 + k/ 4), x, -3, 3)
```



Due to the different scale it is not recommended to display these two different set of solutions in one single plot.



Both Families of Solutions for (12.2)

Since the solving of differential equation heavily depends on the particular case, we want to discuss another example in detail. We want to solve the separable equation

$$\frac{y'y}{\sqrt{1+y^2}} = \sin(x). \tag{12.3}$$

Again, we first take a look on the result of `desolve`

```
sage: x = var('x'); y = function('y')(x)
sage: ed = desolve(diff(y, x) * y / sqrt(1 + y^2) == sin(x), y);
    ed
sqrt(y(x)^2 + 1) == _C - cos(x)
```

We immediately observe that we have to assume that $C - \cos(x) > 0$.

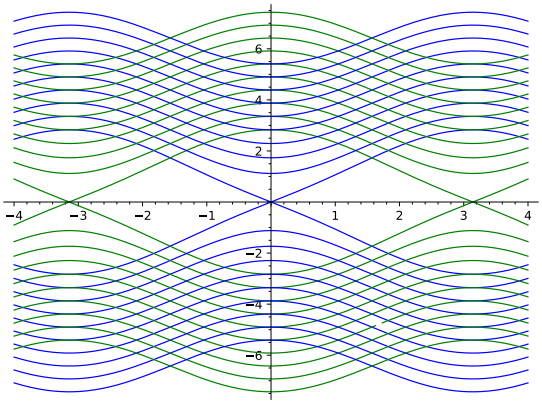
```
sage: _C = ed.variables()[0]
sage: assume(_C - cos(x) > 0)
sage: sol = solve(ed, y); sol
[
y(x) == -sqrt(_C^2 - 2*_C*cos(x) + cos(x)^2 - 1),
y(x) == sqrt(_C^2 - 2*_C*cos(x) + cos(x)^2 - 1)
]
```

Thus, the solutions of (12.3) is given by

$$y(x) = \pm\sqrt{C^2 - 2C\cos(x) + \cos(x)^2 - 1}$$

for all $C > 2$. Moreover, we see that these solutions are also well-defined for $C < -2$. Next, we plot a family of solutions curves for different values of C.

```
sage: P = Graphics()
sage: for j in [0, 1]:
....:     for k in srange(0, 20, 2):
....:         P += plot(sol[j].substitute(_C == 2 + 0.25*k).rhs(),
    x, -4, 4)
....:         P += plot(sol[j].substitute(_C == -2 - 0.25*k).rhs(),
     x, -4, 4, color = 'green')
```



Family of Solutions for (12.3)

Homogeneous Equations

A first order ODE is called a ***homogeneous equation*** if it is given as the quotient of two homogeneous functions of the samge degree, i.e.

$$y' = \frac{P(x,y)}{Q(x,y)}.$$

Definition 12.1.1 A function $f(x_1,\ldots,x_n)$ is called ***homogeneous*** of degree k if for all $\lambda \in \mathbb{R}$

$$f(\lambda x_1,\ldots,\lambda x_n) = \lambda^k f(x_1,\ldots,x_n).$$

An example of a homogeneous equation is the ODE

$$x^2y' = x^2 + xy + y^2, \tag{12.4}$$

because$Q(x) = x^2$ and $P(x,y) = x^2 + xy + y^2$ are both homogeneous functions of degree 2. As usual, we first use the `desolve` command.

```
sage: x = var('x'); y = function('y')(x)
sage: desolve(x^2*diff(y, x) == x^2 + x*y + y^2, y, show_method =
    True)
[_C*x == e^(arctan(y(x)/x)), 'homogeneous']
```

Again, the solution is not given explicitly. Hence, we use the `solve` command.

```
sage: ed = desolve(x^2*diff(y, x) == x^2 + x*y + y^2, y)
sage: solve(ed, y)
[

]
```

Sage is not able to solve this equation with respect to y. However, by the theorem of Picard-Lindelöf we know that there has to exist a solution. Thus, we have to modify the equation

$$Cx = e^{\arctan\left(\frac{y}{x}\right)}.$$

further. We immediately see that this equation is only well-defined over $\mathbb{R}$ if and only if Cx is positive. Thus, we add this condition using the `assume` command and apply the logarithm on both sides.

```
sage: C = ed.variables()[0]
sage: assume(C*x > 0)
sage: eq = log(C*x) == arctan(y/x)
sage: solve(eq, y)
[
y(x) == x*tan(log(_C*x))
]
```

Hence, the solution of (12.4) is given $byy(x) = x\tan\big(\log(Cx)\big)$ whenever $Cx > 0$.

We might need various steps of transforming the output of the `desolve` command to arrive at an expressions for the solution function if we are trying to solve more complicated differential equations like,

$$y' = \frac{y + \sqrt{y^2 + x^2}}{x}. \tag{12.5}$$

```
sage: x = var('x'); y = function('y')(x)
sage: ed = desolve(diff(y, x) == (y + sqrt(y^2 + x^2))/x, y)
sage: ed
x == (x*y(x)/sqrt(x^4) + sqrt(y(x)^2/x^2 + 1))^(x/sqrt(x^2))*_C
```

Again, we only obtain the solution implicitly. But here, the derived equation

$$x = C\left(\frac{xy}{\sqrt{x^4}} + \sqrt{\frac{y^2}{x^2} + 1}\right)^{\frac{x}{\sqrt{x^2}}}$$

is too complicated for the `solve` command.

```
sage: solve(ed, y)
[]
```

Thus, we first have to simplify the equation. The assumption $x > 0$ already implies the simplifications

$$\frac{x}{\sqrt{x^4}} = \frac{1}{x^3}, \quad \frac{x}{\sqrt{(x^2)}} = \frac{1}{x}.$$

Hence, we concentrate us on the case $x > 0$.

```
sage: assume(x > 0)
sage: ed.simplify()
sage: sol = solve(ed, y)[0]; sol
y(x) == (x^2 - sqrt(x^2 + y(x)^2)*_C)/_C
```

However, we still only obtain an implicit equation

$$y = \frac{x^2 - \sqrt{x^2 + y^2}C}{C}.$$

But, we can simplify this equation further by considering the squares of both sides. Hence, we assign `_C` to a `Python` variable and isolate the equation as the output of `solve` is a one-element list and not the equation itself.

```
sage: _C = ed.variables()[0]
sage: sol = sol^2; sol
y(x)^2 == (x^2 - sqrt(x^2 + y(x)^2)*_C)^2/_C^2
sage: solve(sol, y)
Traceback (most recent call last):
...
TypeError: Computation failed since Maxima requested additional
   constraints; using the 'assume' command before evaluation *may
   * help (example of legal syntax is 'assume(_C>0)', see `assume
   ?` for more details)
Is _C positive or negative?
```

The error message tells us, that we should assume `_C`. Hence, we do so and try to solve again.

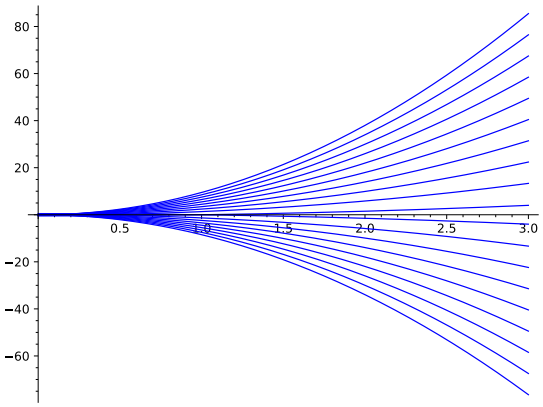
```
sage: assume(_C > 0)
sage: Sp = solve(sol, y)[0]; Sp
y(x) == 1/2*(_C^2 - x^2)/_C
```

Thus, we have finally shown that for $x > 0$ solutions of (12.5) is given by

$$y(x) = \frac{C^2 - x^2}{2C}$$

for any $C \in \mathbb{R}$. Moreover, it is not hard to show that this equation also solves (12.5) for $x < 0$. Again, we plot the solutions for different values of C to visualize them.

```
sage: P = Graphics()
sage: for k in srange(-19, 19, 2):
....:     P += plot(Sp.rhs().substitute(_C == 1/k), x, 0, 3)
```



Exact Equations

A first order ODE is called ***exact*** if there exists a differentiable function $f(x,y)$ such that the differential equation in consideration can be written as

$$\frac{\partial f}{\partial x}+\frac{\partial f}{\partial y}=0.$$

For example the differential equation

$$y'=\frac{\cos(y)-2x}{y+x\sin(y)}$$

is an exact equation, if we choose $f(x,y)=x^2-x\cos(y)+\frac{y^2}{2}$ since

$$\begin{aligned}\frac{\partial f}{\partial x}+\frac{\partial f}{\partial y}&=(2x-\cos(y))+\left(x\sin(y)y'+yy'\right)\\&=(2x-\cos(y))+y'\left(x\sin(y)+y\right).\end{aligned}$$

These kind of equations are solved with the usual `desolve` command.

```
sage: x = var('x'); y = function('y')(x)
sage: desolve(diff(y, x) == (cos(y)- 2*x)/(y + x*sin(y)), y,
   show_method = True)
[x^2 - x*cos(y(x)) + 1/2*y(x)^2 == _C, 'exact']
```

As the solution is only given implicitly, we apply the `solve` command to obtain an explicit solution.

```
sage: ed = desolve(diff(y, x) == (cos(y)- 2*x)/(y + x*sin(y)), y)
sage: solve(ed, y)
[
y(x) == -sqrt(-2*x^2 + 2*x*cos(y(x)) + 2*_C),
y(x) == sqrt(-2*x^2 + 2*x*cos(y(x)) + 2*_C)
]
```

Ricatti Equation

Ricatti equations have the structure of a quadratic equation, i.e. they are of the form

$$y'=P(x)y^2+Q(x)y+R(x),$$

where P, Q and R are continuous functions over a given interval. This kind of first order differential equations are too complicated for the standard algorithms of `desolve`. Therefore, we need to set `contrib_ode = True` to force Sage to use more complex methods. For example, the solution of the Ricatti equation

$$y'=xy^2+\frac{1}{x}y-\frac{1}{x^2}$$

is obtained as follows.

```
sage: x = var('x'); y = function('y')(x)
sage: desolve(diff(y, x) == x*y^2 + y/x - 1/x^2, y, contrib_ode =
    True, show_method = True)
[[y(x) == -1/2*((_C*(bessel_Y(4, 2*sqrt(-x)) - bessel_Y(2, 2*sqrt
    (-x))) + bessel_J(4, 2*sqrt(-x)) - bessel_J(2, 2*sqrt(-x)))*x
    + 3*(_C*bessel_Y(3, 2*sqrt(-x)) + bessel_J(3, 2*sqrt(-x)))*
    sqrt(-x))/((_C*bessel_Y(3, 2*sqrt(-x)) + bessel_J(3, 2*sqrt(-x
    )))*sqrt(-x)*x^2)],
 'riccati']
```

Here, the solution is given explicitly. We observe that the solution is a linear combination of Bessel functions of the first kind, `bessel_J` and of the second kind, `bessel_Y`.

Lagrange and Clairault Equations

A ***Lagrange equation*** is a first order ODE of the form

$$y = xP(y') + Q(y'),$$

where P and Q are C^1-functions over a given interval. The special case $P = \mathrm{Id}$, i.e.

$$y = xy' + Q(y')$$

is called a ***Clairault equation***. Similar to Ricatti equations, we have to set `contrib_ode = True` to solve this equation with Sage. As an example, we solve the Clairault equation

$$y = xy' - (y')^2.$$

```
sage: x = var('x'); y = function('y')(x)
sage: desolve(y == x*diff(y, x) - diff(y, x)^2, y, contrib_ode =
    True, show_method = True)
[[y(x) == -_C^2 + _C*x, y(x) == 1/4*x^2], 'clairault']
```

12.1.2 A Parametric Equation

In the previous sections we have only considered ODEs with x and $y(x)$ being the only involved variables. However, it is also possible to solve ODEs with parameters. For example we can study the general solutions of the ODE

$$y' = ay - by^2, \tag{12.6}$$

with a, b being positive real numbers. As usual, we start with the `desolve` command, but this time we have to use the option `ivar` to identify x as the only independent variable.

```
sage: x, a, b = var('x, a, b'); y = function('y')(x)
sage: sol = desolve(diff(y, x) == a*y - b*y^2, y, ivar = x); sol
-(log(b*y(x) - y) - log(y(x)))/a == _C + x
```

Since we do not obtain y explicitly we try to isolate it with `solve`.

Differential Equation	
Variable Declaration	`x = var('x')`
Function Declaration	`y = function('y')(x)`
Solving an Equation	`desolve(equation, y, <options>)`
First Order ics $y(x_0) = y_0$	`ics = [x0, y0]`
Second Order ics $y(x_0) = y_0 y(x_1) = y_1$	`ics = [x0, y0, x1, y1]`
or $y(x_0) = y0, y'(x_0) = y'_0$	`ics = [x0, y0, y0']`
Independent Variable	`ivar = x`
Display Resolution Method	`show_method = True`
Call for Special Methods	`contrib_ode = True`

Table 12.1.: Differential Equations

```
sage: solve(sol, y)
[
log(y(x)) == -C*a + a*x + log(b*y(x) - a)
]
```

Unfortunately, we still do not have an explicit equation for y. Hence, we group together the terms on the left-hand side and use `simplify_log` to combine the log-terms. Then we try again the `solve` command to obtain an explicit expression for y.

```
sage: Sol = solve(sol, y)[0]
sage: Sol = Sol.lhs() - Sol.rhs(); Sol
-_C*a - a*x - log(b*y(x) - a) + log(y(x))
sage: Sol = Sol.simplify_log(); Sol
-_C*a - a*x + log(y(x)/(b*y(x) - a))
sage: solve(Sol, y)[0].simplify()
y(x) == a*e^(_C*a + a*x)/(b*e^(_C*a + a*x) - 1)
```

Thus, we have shown that for every $a, b \geq 0$ the solution of (12.6) is given by

$$y(x) = \frac{ae^{Ca+ax}}{be^{Ca+ax} - 1}$$

for any $C \in \mathbb{R}$.

12.1.3 Numerical Solving

Instead of explicitly solving a differential equation with `desolve` we can also solve them numerically with `desolve_rk4`, where `rk4` indicates that a Runge-Kutta method of 4th order is used. The syntax for `desolve_rk4` is essentially the same as for `desolve`. But since we are solving numerically, the option `ics` is not optional but essential. Moreover, we have to specify the interval over which the differential equation is solved using the option `end_points`, where `end_points = [a,b]` means that `desolve_rk4` solves the equation between `min(ics[0], a)` and `max(ics[0], b)`. In addition, we can determine the length of the steps used in the Runge-Kutta method with the option `steps`, whose default value is 0.1.

Instead of an expression for the solution function y, `desolve_rk4` only returns a list of function values, i.e. `[(x0, y(x0), ..., (xn, y(xn))]`. Applying the `line` command to this list returns plot of the approximated solution curve. Although we do not have an explicit solution function given, the plot of the approximated solution can still be useful to read off various properties. Moreover, numerical calculations are in general much fast. Nevertheless, as numerical methods only approximate the exact solution, the results of `desolve_rk4` and `desolve` might differ slightly.

To illustrate this difference we draw a family of integral curves of the ODE

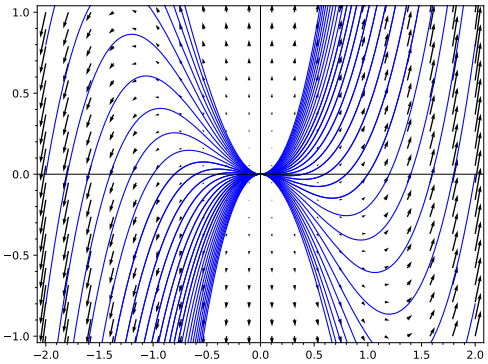
$$xy' = 3y + x^3 \tag{12.7}$$

once exactly and once numerically. First, we draw the plot for the exact solutions obtained with `desolve`. To optimize computation time, we first solve (12.7) and substitute the different initial values afterwards. For a better visualization we also added the corresponding vector field.

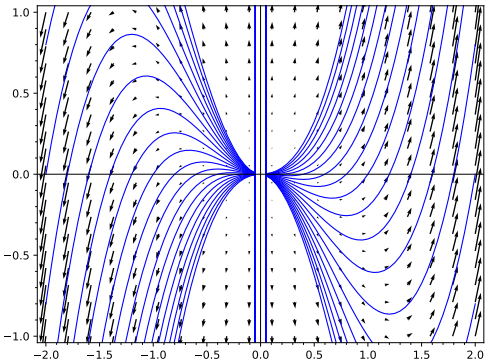
```
sage: x = var('x'); y = function('y')(x)
sage: sol = desolve(x*diff(y, x) == 2*y + x^3, y) #exact solution
sage: C = sol.variables()[0]
sage: S = Graphics()
sage: for i in srange(-2, 2, 0.2):
....:     S += plot(sol.substitute(C == i - 1), (x, -2, 2))
....:     S += plot(sol.substitute(C == i + 1), (x, -2, 2))
sage: y = var('y')
sage: S += plot_vector_field((x, 2*y + x^3), (x, -2, 2), (y, -1,
    1))
```

Before, we take a look at the plot, we draw the equivalent plot for the numerical solutions. Since we can not obtain an exact solution in which we can substitute the initial value, we have to solve (12.7) and draw the corresponding approximated solution for each initial value separately.

```
sage: x = var('x'); y = function('y')(x)
sage: DE = x*diff(y, x) == 2*y + x^3
sage: N = Graphics()
sage: for i in srange(-2, 2, 0.2): #drawing the numerical
    solutions
....:     N += line(desolve_rk4(DE, y, ics = [1, i], ivar = x, step
     = 0.05, end_points = [0,2]))
....:     N += line(desolve_rk4(DE, y, ics = [-1, i], ivar = x,
    step = 0.05, end_points = [-2 ,0]))
sage: y = var('y')
sage: N += plot_vector_field((x, 2*y + x^3), (x, -2, 2), (y, -1,
    1))
```



Symbolic Solutions



Numerical Solutions

12.2 Second Order Equations

The `desolve` command is also able to solve some second order ODEs, e.g.

$$y'' + 3y = x^2 - 7x + 31.$$

```
sage: x = var('x'); y = function('y')(x)
sage: DE = diff(y, x, 2) + 3*y == x^2 - 7*x + 31
sage: sol = desolve(DE, y).expand()
sage: sol
1/3*x^2 + _K2*cos(sqrt(3)*x) + _K1*sin(sqrt(3)*x) - 7/3*x + 91/9
```

Since a second order ODE needs two initial conditions to be uniquely determined, Sage introduces two variables K_1, K_2 representing real numbers. To assign them to Python variable we apply the method `variables` to the solution.

```
sage: K1, K2 = sol.variables()[0], sol.variables()[1]
```

For a second order ODE there are two types of initial conditions we can add: Either the values of y and its derivative y' at a specific point x_0, e.g. $y(0) = 1, y'(0) = 2$,

```
sage: desolve(DE, y, ics = [0, 1, 2]).expand()
1/3*x^2 + 13/9*sqrt(3)*sin(sqrt(3)*x) - 7/3*x - 82/9*cos(sqrt(3)*x
    ) + 91/9
```

or the value of y at two distinct points, e.g. $y(0) = 1, y(-1) = 0$

```
sage: desolve(DE, y, ics = [0, 1, -1, 0]).expand()
1/3*x^2 - 7/3*x - 82/9*cos(sqrt(3))*sin(sqrt(3)*x)/sin(sqrt(3)) +
    115/9*sin(sqrt(3)*x)/sin(sqrt(3)) - 82/9*cos(sqrt(3)*x) + 91/9
```

It is also possible to solve some non-linear second order ODEs like

$$y'' + y(y')^3 = 0.$$

```
sage: desolve(diff(y,x,2)+y*(diff(y,x,1))^3==0,y).expand()
1/6*y(x)^3 + _K1*y(x) == _K2 + x
```

12.2.1 How to solve a PDE: The Heat Equation

Many well-known PDEs are of second order involving the Laplacian. In this section we describe how we can use Sage to solve the one-dimensional heat equation

$$\frac{\partial^2 u}{\partial x^2}(x,t) = \frac{\partial u}{\partial t}(x,t) \tag{12.8}$$

with the initial conditions

$$u(0,t) = 0, \quad u(1,t) = 0, \quad \forall t > 0. \tag{12.9}$$

The common ansatz for the heat equation is the method of separation of variables, i.e. we are looking for a nonzero solution of the form

$$u(x,t) = f(x)g(t).$$

```
sage: x, t = var('x, t'); f = function('f')(x); g = function('g')(
    t)
sage: u = f*g
sage: eq(x,t) = diff(u,x,2) == diff(u, t)
sage: eq(x, t)
g(t)*diff(f(x), x, x) == f(x)*diff(g(t), t)
```

This ansatz leads us to the equation

$$g(t)f''(x) = f(x)g'(t).$$

Assuming $u(x,t) = f(x)g(t)$ to be nonzero we can divide by u on both sides.

```
sage: eqn = eq/u
sage: eqn(x,t)
diff(f(x), x, x)/f(x) == diff(g(t), t)/g(t)
```

This leads us to an equation where each side only depends on one variable,

$$\frac{f''(x)}{f(x)} = \frac{g'(t)}{g(t)}.$$

It follows that each side has to be constant. Hence, we can separate these equations by introducing a constant k and then solve them separately.

```
sage: k = var('k')
sage: eq1(x,t) = eqn(x,t).lhs() == k; eq2(x,t) = eqn(x,t).rhs() ==
    k
sage: g(t) = desolve(eq2(x,t), g, ivar = t); g(t)
_C*e^(k*t)
```

Thus, $g(t) = Ce^{kt}$ for any $C, k \in \mathbb{R}$. Now we want to find an explicit expression for f.

```
sage: f(x) = desolve(eq1(x,t), f, ivar = x)
Traceback (most recent call last):
...
TypeError: Computation failed since Maxima requested additional
   constraints; using the 'assume' command before evaluation *may
   * help (example of legal syntax is 'assume(k>0)', see `assume
   ?` for more details)
Is k positive, negative or zero?
```

Simply applying the `desolve` command raises an error. The corresponding error message tells us that Sage needs to know whether k is positive, negative or 0. Hence, we use `assume` to first deal the case $k > 0$.

```
sage: assume(k > 0)
sage: desolve(eq1, f, ivar = x)
_K1*e^(sqrt(k)*x) + _K2*e^(-sqrt(k)*x)
```

Thus, $f(x) = K_1 e^{\sqrt{k}x} + K_2 e^{-\sqrt{k}x}$ if k is positive. However, inserting the initial conditions

$$0 = u(0,t) = f(0)g(t) = (K_1 + K_2)Ce^{kt},$$
$$0 = u(1,t) = f(1)g(t) = (K_1 e^{\sqrt{k}} + K_2 e^{-\sqrt{k}})Ce^{kt},$$

implies $K_1 = K_2 = 0$, i.e. $f \equiv 0$. Thus, $k > 0$ is not the correct assumption. Since $k = 0$ would only lead to a trivial solution, we try $k < 0$.

```
sage: desolve(diff(f, x, 2) == -k*f, f, ivar = x)
_K2*cos(sqrt(k)*x) + _K1*sin(sqrt(k)*x)
```

Applying our initial conditions $u(0,t) = u(1,t) = 0$ to this solution shows that $K_2 = 0$ and $k = -\pi^2 n^2$ for $n \in \mathbb{N}$. Putting everything together, we have shown that the solution to the one dimensional heat equation with the initial conditions (12.9) is given by

$$u(x,t) = \sum_{n=1}^{\infty} a_n \sin(\pi n x) e^{-\pi^2 n^2 t},$$

where the a_n are specified by adding an initial condition

$$u(x,0) = f(x) = \sum_{n=1}^{\infty} a_n \sin(\pi n x).$$

12.3 The Laplace Transform

Instead of the `desolve` command it is also possible to solve differential equations using the Laplace transform in Sage. The Laplace transform allows us to rewrite a differential equation with initial conditions into an algebraic equation. After solving this algebraic equation the inverse Laplace transform returns us the solution of the differential equation.

The ***Laplace transform*** of a continuous function $f : [0,\infty) \to \mathbb{R}$ is given by

$$\mathscr{L}(f(x)) = F(s) = \int_0^\infty e^{-sx} f(x)dx,$$

where s can be any complex number. It is immediate that the Laplace transform is linear, i.e. $\mathscr{L}(af(x) + bg(x)) = a\mathscr{L}(f(x)) + b\mathscr{L}(g(x))$ for $a,b \in \mathbb{R}_+$ and functions $f,g : [0,\infty) \to \mathbb{R}$. The inverse Laplace transform of a complex function $F : \mathbb{C} \to \mathbb{R}$ is given by

$$\mathscr{L}^{-1}(F(s)) = \frac{1}{2\pi i}\int_{\gamma-i\infty}^{\gamma+i\infty} e^{st}F(s)ds,$$

where $\gamma \in \mathbb{R}$ is the largest real part of a singularity of F.

■ **Example 12.1**

$$\begin{aligned}
\mathscr{L}(e^{-ax}) &= \frac{1}{s+a},\\
\mathscr{L}(x^n) &= \frac{n!}{s^{n+1}},\\
\mathscr{L}(\sin(ax)) &= \frac{a}{s^2+a^2},\\
\mathscr{L}(\cos(ax)) &= \frac{s}{s^2+a^2}.
\end{aligned}$$

■

In Sage the Laplace transform of a function is calculated with the method `laplace` and the inverse Laplace transform is returned by the method `inverse_laplace`.

```
sage: x, s = var('x, s'); f(x) = sin(x)
sage: f.laplace(x, s)
x |--> 1/(s^2 + 1)
sage: (1 / (s^2 + 1)).inverse_laplace(s, x)
sin(x)
```

Using partial integration, we see that if f is continuously differentiable, then

$$\mathscr{L}(f\prime) = s\mathscr{L}(f) - f(0). \tag{12.10}$$

Similarly, if f is C^2, then

$$\mathscr{L}(f'') = s^2\mathscr{L}(f) - sf(0)f'(0). \tag{12.11}$$

These identities allow us to reformulate a differential equation as an algebraic equation. We show this procedure along the example

$$y'' - 3y' - 4y = \sin(x)$$

with initial conditions $y(0) = 1$, $y'(0) = -1$.

First we use the Laplace transform on both sides,

$$\mathscr{L}(y'' - 3y' - 4y) = \mathscr{L}(\sin(x)).$$

Laplace Transform	
Laplace Transform of $f(x) \mapsto \mathscr{L}(f)(s) = X(s)$	`f.laplace(s))`
Inverse Laplace Transform $X(s) \mapsto f(x)$	`X(s).inverse_laplace(s,x)`
Solving an ODE with the Laplace Transform	`desolve_laplace(equation, function)`

Table 12.2.: Laplace Transform

Using the linearity of the Laplace transform and the identities (12.10), (12.11), we obtain the equation

$$(s^2-3s-4)\mathscr{L}(y)-sy(0)-y'(0)+3y(0)=\mathscr{L}(\sin(x)).$$

We already have seen above that the Laplace transform of $\sin(x)$ is given by $\frac{1}{s^2+1}$, i.e.

$$(s^2-3s-4)\mathscr{L}(y)-sy(0)-y'(0)+3y(0)=\frac{1}{s^2+1}.$$

Inserting the given initial conditions, we conclude that the Laplace transform of y is given by

$$\mathscr{L}(y)=\frac{1}{(s^2-3s-4)(s^2+1)}+\frac{s-4}{s^2-3s-4}.$$

Thus, we the solution y is given by the inverse Laplace transformation of the above expression which we can obtain with the method `inverse_lapace`.

```
sage: X(s) = 1/((s^2 - 3*s - 4)*(s^2 + 1)) + (s-4)/(s^2 - 3*s - 4)
sage: X(s).inverse_laplace(s, x)
3/34*cos(x) + 1/85*e^(4*x) + 9/10*e^(-x) - 5/34*sin(x)
```

Hence, the solution of $y''-3y'-4y=\sin(x)$ is given by $y(x)=\frac{3}{34}\cos(x)+\frac{1}{85}e^{4x}+\frac{9}{10}e^{-x}-\frac{5}{34}\sin(x)$.

This whole solving strategy is carried out at once by calling the command `desolve_laplace`.

```
sage: x = var('x'); y = function('y')(x)
sage: DE = diff(y, x, x) - 3*diff(y, x) - 4*y - sin(x) == 0
sage: desolve_laplace(DE, y) #general solution
1/85*(17*y(0) + 17*D[0](y)(0) + 1)*e^(4*x) + 1/10*(8*y(0) - 2*D
    [0](y)(0) - 1)*e^(-x) + 3/34*cos(x) - 5/34*sin(x)
sage: desolve_laplace(DE, y, ics = [0, 1, -1])
3/34*cos(x) + 1/85*e^(4*x) + 9/10*e^(-x) - 5/34*sin(x)
```

As we can see in the above code snippet, it is possible to obtain a general solution by declaring no initial conditions. In that case, Sage shows, where the values of $y(0)$ and $y'(0)$ appear explicitly. Although this general expressions is not as readable as those with declared initial conditions, it shows how the initial conditions modify the solution function.

12.4 Systems of Linear Differential Equations

It is also possible to solve systems of linear differential equations in Sage. The used commands here are `desolve_system` for the symbolic solution and `desolve_system_rk4` for the numerical solution. The syntax remains essentially the same. There are only a few exceptions: First, the system of differential equations as well as the dependent variables are handed over as a list, e.g. `[y1, y2, y3]`. The corresponding syntax for the initial conditions is `ics = [x0, y1(x0), y2(x0), y3(x0)]` when dealing with a system of first order linear ODEs. In fact, `desolve_system` and `desolve_system_rk4` only accept systems of first order ODEs. But we will see that this is not a restriction, as any system of linear differential equations can be reduced to a first order system.

12.4.1 Systems of First Order Differential Equations

A system of linear first order differential equations can be written as

$$y'(x) = A \cdot y(x)$$

with $y : I \to \mathbb{R}^n$, where $I \subset \mathbb{R}$ is an interval and A is an $n \times n$ matrix. Accordingly the initial condition is given by a vector $c \in \mathbb{R}^n$.

If we want to solve such an initial value problem

$$\begin{cases} y'(x) &= A \cdot y(x), \\ y(0) &= c, \end{cases} \tag{12.12}$$

with

$$A = \begin{pmatrix} 2 & -2 & 0 \\ -2 & 0 & 2 \\ 0 & 2 & 2 \end{pmatrix}, \quad y(x) = \begin{pmatrix} y_1(x) \\ y_2(x) \\ y_3(x) \end{pmatrix}, \quad c = \begin{pmatrix} 2 \\ 1 \\ -2 \end{pmatrix},$$

in Sage, we need to hand over the system of differential equations as well as the dependent variables as a list.

```
sage: x = var('x')
sage: y1 = function('y1')(x); y2 = function('y2')(x); y3 =
   function('y3')(x)
sage: y = vector([y1, y2, y3]) #function vector
sage: A = matrix([[2, -2, 0], [-2, 0, 2], [0, 2, 2]])
sage: system = [diff(y[i], x) - (A * y)[i] for i in srange(3)];
   system
[-2*y1(x) + 2*y2(x) + diff(y1(x), x),
 2*y1(x) - 2*y3(x) + diff(y2(x), x),
 -2*y2(x) - 2*y3(x) + diff(y3(x), x)]
sage: desolve_system(system, [y1, y2, y3], ics = [0, 2, 1, -2])
[y1(x) == e^(4*x) + e^(-2*x),
 y2(x) == -e^(4*x) + 2*e^(-2*x),
 y3(x) == -e^(4*x) - e^(-2*x)]
```

Systems of Differential Equations	
Solving a System	`desolve_system([eq1, ...], [y1,...]`
System Initial Conditions $y_1(x_0) = c_1, \ldots$	`ics = [x0, c1, ...]`

Table 12.3.: Systems of Differential Equations

Hence,

$$y(x) = \begin{pmatrix} e^{4x} - e^{-2x} \\ -e^{4x} + 2e^{-2x} \\ -e^{4x} - e^{-2x} \end{pmatrix}$$

is the solution of the initial value problem (12.12). At this point, we want to remark that the factors in the exponents correspond to real eigenvalues of the matrix A.

```
sage: A.eigenvalues()
[4, 2, -2]
```

What happens, if the matrix A has complex eigenvalues? For example, in the initial value problem

$$\begin{cases} y'(x) &= A \cdot y(x) \\ y(0) &= c \end{cases}$$

with

$$A = \begin{pmatrix} 3 & -4 \\ 1 & 3 \end{pmatrix}, \quad c = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$$

the matrix A has the complex eigenvalues $3 \pm 2i$, as can be verified with Sage.

```
sage: A = matrix([[3, -4], [1, 3]])
sage: A.change_ring(QQbar).eigenvalues()
[3 + 2*I, 3 - 2*I]
```

This structure of the eigenvalues has an impact on the structure of the solution.

```
sage: x = var('x'); y1 = function('y1')(x); y2 = function('y2')(x)
sage: y = vector([y1, y2])
sage: system = [diff(y[i], x) - (A*y)[i] for i in srange(2)]
sage: desolve_system(system, [y1, y2], ics = [0,2,0])
[y1(x) == 2*cos(2*x)*e^(3*x), y2(x) == e^(3*x)*sin(2*x)]
```

We see that the complex part of the eigenvalues appears as factors in the trigonometric functions $\cos(2x), \sin(2x)$, while the real part of the eigenvalues again appears as a factor in the exponent.

12.4.2 Systems of Higher Order

Since `desolve_system` and `desolve_system_rk4` only accept first order systems, we have to reduce higher order systems of linear differential equations to first order systems.

We explain this reduction only for second order systems in detail since it contains all major ideas and generalizes easily to higher order systems.

If we deal with a system of linear second order equations, we introduce for any second derivative y_i'' that is involved a new dependent variable u and extend the system by the linear first order equation $u = y'$. Thus, we can replace y_i'' with u', transforming it to a first order differential equation. To illustrate this procedure we apply it to the second order system

$$\begin{cases} y_1''(x) - y_1(x) + 6y_2(x) - y'(x) - 3y_2'(y) & = 0, \\ y_2''(x) + 2y_1(x) - 6y_2(x) - y_1'(x) + y_2'(x) & = 0. \end{cases} \tag{12.13}$$

To reduce this to a first order system we define

$$u = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_1' \\ y_2' \end{pmatrix}.$$

Thus, (12.13) is equivalent to the first order system

$$\begin{cases} u_1' & = u_3, \\ u_2' & = u_4, \\ u_3' & = 2u_1 - 6u_2 + u_3 + 3u_4, \\ u_4' & = -2u_1 + 6u_2 + u_3 - u_4. \end{cases}$$

This, in turn, can be rewritten as $u'(x) = A \cdot u(x)$ with

$$A = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 2 & -2 & 1 & 3 \\ -2 & 6 & 1 & -1 \end{pmatrix}.$$

Now, we can use the same procedure as explained in Section 12.4.1 to solve this system of linear first order equations.

```
sage: x = var('x')
sage: u1 = function('u1')(x); u2 = function('u2')(x); u3 =
   function('u3')(x); u4 = function('u4')(x)
sage: u = vector([u1, u2, u3, u4])
sage: A = matrix([[0, 0, 1, 0], [0, 0, 0, 1], [2, -6, 1, 3], [-2,
   6, 1, -1]])
sage: system = [diff(u[i], x) - (A*u)[i] for i in srange(4)]
sage: sol = desolve_system(system, [u1, u2, u3, u4])
```

Since we want to determine $y(x) = \begin{pmatrix} y_1(x) \\ y_2(x) \end{pmatrix}$, we are only interested in the first coordinates $u_1 = y_1$ and $u_2 = y_2$ of the solution.

```
sage: sol[0] #y1
u1(x) == 1/12*(2*u1(0) - 6*u2(0) + 5*u3(0) + 3*u4(0))*e^(2*x) +
   1/24*(2*u1(0) - 6*u2(0) - u3(0) + 3*u4(0))*e^(-4*x) + 3/4*u1
   (0) + 3/4*u2(0) - 3/8*u3(0) - 3/8*u4(0)
sage: sol[1] #y2
u2(x) == -1/12*(2*u1(0) - 6*u2(0) - u3(0) - 3*u4(0))*e^(2*x) -
   1/24*(2*u1(0) - 6*u2(0) - u3(0) + 3*u4(0))*e^(-4*x) + 1/4*u1
   (0) + 1/4*u2(0) - 1/8*u3(0) - 1/8*u4(0)
```

This solution can be summarized in a more readable way as

$$\begin{cases} y_1(x) &= k_1 e^{2x} + k_2 e^{-4x} + 3k_3, \\ y_2(x) &= k_4 e^{2x} - k_2 e^{-4x} + k_3, \end{cases}$$

with

$$\begin{aligned} k_1 &= \frac{1}{12}(2u_1(0) - 6u_2(0) + 5u_3(0) + 3u_4(0)), \\ k_2 &= \frac{1}{24}(2u_1(0) - 6u_2(0) - u_3(0) + 3u_4(0)), \\ k_3 &= \frac{1}{8}(2u_1(0) + 2u_2(0) - u_3(0) - u_4(0)), \\ k_4 &= -\frac{1}{12}(2u_1(0) - 6u_2(0) - u_3(0) - 3u_4(0)). \end{aligned}$$

Iterating this process we can solve linear differential equations of arbitrary order. For example, the initial value problem

$$\begin{cases} y'''(x) - 2y''(x) + 4y'(x) + 7y(x) &= 0, \\ y(0) &= -1, \\ y'(0) &= 4, \\ y''(0) &= -11, \end{cases} \tag{12.14}$$

can be reduced to a first order system by introducing the vector

$$u = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} = \begin{pmatrix} y \\ y' \\ y'' \end{pmatrix}.$$

Thus, the initial value problem (12.14) reduces to the following first order system

$$\begin{cases} u_1' &= u_2 \\ u_2' &= u_3 \\ u_3' &= -7u_1 - 4u_2 + 2u_3, \\ u(0) &= (-1, 4, -11)^t, \end{cases}$$

which is equivalent to $u' = A \cdot u$ with

$$A = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -7 & -4 & 2 \end{pmatrix}.$$

Now we can solve this system with `desolve_system`.

```
sage: x = var('x'); u1 = function('u1')(x); u2 = function('u2')(x)
    ; u3 = function('u3')(x)
sage: u = vector([u1, u2, u3]); A = matrix([[0, 1, 0], [0, 0, 1],
    [-7, -4, 2]])
sage: system = [diff(u[i], x) - (A*u)[i] for i in srange(3)]
sage: sol = desolve_system(system, [u1, u2, u3], ics = [0, -1, 4,
    -11])
sage: sol[0]
u1(x) == -1/209*(29*sqrt(19)*sin(1/2*sqrt(19)*x) - 361*cos(1/2*
    sqrt(19)*x))*e^(3/2*x) - 30/11*e^(-x)
```

Hence, the solution of (12.14) is given by

$$y(x) = -\frac{19\sqrt{19}\sin\left(\frac{1}{2}\sqrt{19}x\right)}{209} - 361\cos\left(\frac{1}{2}\sqrt{19}x\right)e^{\frac{3}{2}x} - \frac{30}{11}e^{-x}.$$

12.4.3 Numerical Solving

To solve first order systems numerically we use `desolve_system_rk4`. Again, the output consists of a list of points `[(t0, u1(t0), ..., uk(t0)), ..., (tn, u1(tn), ..., uk(tn))]` representing the function values of the approximated solution curves $u_1(t),\ldots,u_k(t)$. The approximated curves u_i can be visualized by applying the `line` command to an accordingly extracted sublist `[(t0, ui(t0)), ..., tn,ui(tn))]`. We show the usage of `desolve_system_rk4` in the example of the Lotka-Voltera-predator-prey-model. This system of first order differential equations models the variation of a set of prey and predators,

$$\begin{cases} u'(t) &= au(t) - bu(t)v(t), \\ v'(t) &= cv(t) + dbu(t)v(t), \end{cases}$$

where u is the number of preys (for example rabbits) and v is the number of predators (for example foxes). The parameters a,b,c,d describe the different evolution of popoluation:

- a is the natural growth of the rabbits without foxes to eat them,
- b is he decrease of rabbits, when foxes kill them,
- c is the decrease of foxes without any rabbits to eat,
- d indicates how many rabbits are needed for a new fox to appear.

Of course we can try to solve this system in full generality with `desolve_system`.

```
sage: a, b, c, d, t = var('a, b, c, d, t'); u = function('u')(t);
    v = function('v')(t)
system = [diff(u, t) - a*u + b*u*v, diff(v, t) + c*v - d*b*u*v]
sage: desolve_system(system, [u, v], ivar = t)
[u(t) == ilt((b*laplace(u(t)*v(t), t, g1655) - u(0))/(a - g1655),
    g1655, t),
 v(t) == ilt((b*d*laplace(u(t)*v(t), t, g1655) + v(0))/(c + g1655)
     , g1655, t)]
```

The symbolic solution does not give us many information. The output `ilt` stand for *inverse Laplace transform* and appears if Sage can not find an explicit inverse Laplace transform on its own. Thus, it is convenient to ask for a numerical approximation. There, we first have to fix values for the parameters:

$$a = 1, \quad b = 0.1, \quad c = 1.5, \quad d = 075$$

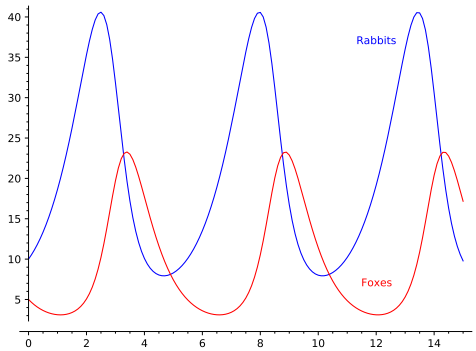
and the initial values

$$u(0) = 10, \quad v(0) = 5.$$

```
sage: u, v, t = var('u, v, t')
sage: a, b, c, d = 1., 0.1, 1.5, 0.75
sage: S = desolve_system_rk4([a*u - b*u*v, -c*v + d*b*u*v], [u, v
    ], ics = [0, 10, 5], ivar = t, end_points = [0, 15])
```

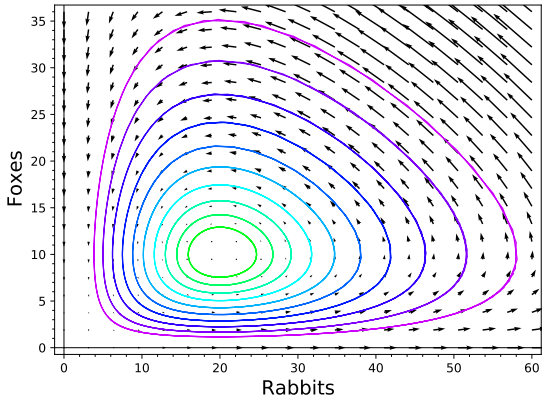
`S` is a list of points `[(t0, u(t0), v(t0)), ((t1), u(t1), v(t1)), ..., (tn, u(tn), v(tn))]`. Therefore, we first have to extract the pairs $(t_i, u(t_i))_i$ and $(t_i, v(t_i))$ to draw the corresponding curves u and v.

```
sage: Qr = [[i, j] for i,j,k in S] #the rabbits
sage: Qf = [[i, k] for i,j,k, in S] #the foxes
sage: P = Graphics()
sage: P += line(Qr, color = 'blue')
sage: P += line(Qf, color = 'red')
sage: P.axes_labels(['time', 'population'])
None
sage: P += text('Rabbits', (12, 37), fontsize = 10, color = 'blue'
    )
sage: P += text('Foxes', (12 , 7), fontsize = 10, color = 'red')
```



We could also draw the curves $(u(t), v(t))$ which are the integral curves of the vector field defined by (**??**) for different starting values $(u(0), v(0))$.

```
sage: n = 10; R = srange(6, 18, 12/n); F = srange(3, 9, 6/n) #set
   of starting values
sage: V = plot_vector_field([a*u - b*u*v, -c*v + d*b*u*v], (u, 0,
   60), (v, 0, 36)) #start with vectorfield
sage: for j in srange(n): #solve for all starting pairs
....:     S = desolve_system_rk4([a*u - b*u*v, -c*v + d*b*u*v], [u,
    v], ics = [0, R[j], F[j]], ivar = t, end_points = [0, 15])
....:     Q = [[j, k] for i, j, k in S] #extracting the pairs (u(t)
   , v(t))
....:     V += line(Q, color = hue(0.8 - j/(2*n))) #adding line
   with small color change
sage: V.axes_labels(['Rabbits', 'Foxes'])
None
```

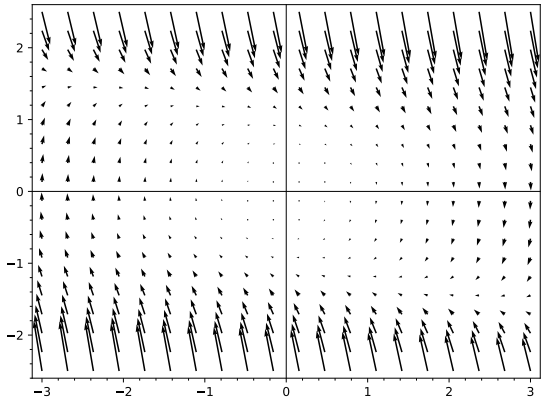


We finish this section by discussing a further example where we are interested in the integral curves $(x(t), y(t))$ of the vector field

$$\begin{cases} x'(t) &= y(t), \\ y'(t) &= 0.5y(t) - x(t) - y(t)^3. \end{cases}$$

As above, we would like to have the vector field and the integral curves in one picture. However, the vector field values are so small at the origin that we can not recognize the direction anymore.

```
sage: x, y, t = var('x, y, t')
sage: DE = [y, 0.5*y- x - y^3] #the system
sage: V = plot_vector_field(DE, (x, -3, 3), (y, -2.5, 2.5))
```

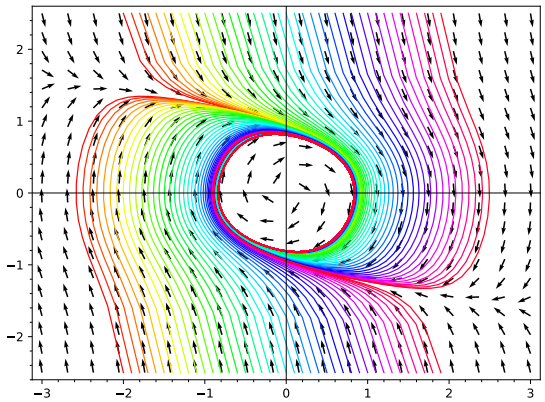


Therefore, we first write a small procedure normalizing the vectors, i.e. we sacrifice the information of the length of the vector field arrows to emphasize their direction.

```
sage: def p(x,y):
....:     v = vector(DE)
....:     return v/v.norm()
sage: V = plot_vector_field(p(x, y), (x, -3, 3), (y, -2.5, 2.5))
```

Next, we add a family of integral curves to this plot. Remember, that the solution of the system gives of list of points $(t_i, x(t_i), y(t_i))_i$ where we have to extract the pairs $(x(t_i), y(t_i))$ before we can draw the curves with `line`.

```
sage: for j in srange(-2, 2, 0.1): #solve for all starting pairs
....:     S1 = desolve_system_rk4(DE, [x, y], ics = [0, j, 2.5],
    ivar = t, end_points = [0, 40])
....:     Q1 = [[j, k] for i, j, k in S1] #extracting the pairs (x(
    t), y(t))
....:     V += line(Q1, color = hue((j+2)/4)) #adding line with
    small color change
....:     S2 = desolve_system_rk4(DE, [x, y], ics = [0, j, -2.5],
    ivar = t, end_points = [0, 40])
....:     Q2 = [[j, k] for i, j, k in S2] #extracting the pairs (x(
    t), y(t))
....:     V += line(Q2, color = hue((j+2)/4))
```



V Appendix

A Git It **247**
A.1 Getting Started
A.2 How to Use Git

Bibliography **253**
Online
Books

Index **255**

A. Git It

A.1 Getting Started

`Git` is a version-control system that simplifies working on a collaborative level. It has the great advantage that all changes which are made during the development process never get lost. In particular, one can always switch back to previous versions, compare them with others and easily undo mistakes. Due to its powerful algorithms, it is possible to work simultaneously, because `git` is able to merge changes done by different collaborators.

A.1.1 Installing Git

The installation on Debian-based systems is easy. Run the command

```
$ sudo apt install git
```

and you are done. However, the installation of `git` on macOS or Windows requieres more several steps:

1. Visit `https://git-scm.com/download/win` and download the setup file.
2. Execute it and follow the instructions.
3. Start `git` with the application `git bash`.

R *Alternatively, you can use graphical user interfaces for `git` such as "git fork" (freely downloadable from `https://git-fork.com/`). But there are many other free clients available on the market.*

In the following, we explain the usage of `git` within the command line. If you use a graphical `git` client, you can find the corresponding buttons in the interface: the commands still remain the same here.

A.2 How to Use Git

To use `git` collaboratively, you need to be registered on a remote server. The most common platforms are `GitHub` and `GitLab`. Students and employees of the University of Potsdam have free access to the University `GitLab` server.

A.2.1 Clone and Pull

We have already organized a repository for this course. There, you can find recent exercise sheets, lecture notes and material regarding the lecture. If you want to ***clone*** this repository to your local machine, first switch to a directory of your choice:

```
$ cd directory/of/your/choice
```

Afterwards, clone our repository with the following command:

```
$ git clone https://gitup.uni-potsdam.de/micjung/sagecourse.git
```

The content is now available in the directory `directory/of/your/choice`.

To keep up the pace with our weekly updates, you have to ***pull*** the repository from the remote server on a regular basis. To do so, switch to the directory above on your local machine. First, you have to make sure that you are in the correct branch:

```
$ git checkout master
```

Now, you pull the recent version of our repository by typing

```
$ git pull
```

A.2.2 Create your own Repository

If you want to share a git repository with others, you have to use a remote server. The University of Potsdam offers a GitLab server for git projects. To create a remote git project, you need to sign in first:

1. Visit `https://gitup.uni-potsdam.de/` and choose “University of Potsdam” in the list.
2. Log in with your personal central university account and accept the terms of use.
3. You are now free to create your own projects or add this one to your list.

If you have followed these instructions, you can see the button “New Project” in the upper right corner. Click on it to create your own awesome project and fill in all necessary information. To clone this project to your local machine, type

```
$ git clone https://gitup.uni-potsdam.de/youraccount/yourproject.git
```

Furthermore, it is possible to create a git repository *locally*. This can be achieved as follows:

```
$ git init
```

This command creates a new local git repository in the local directory. To link this repository with a preexisting remote repository, you can type

```
$ git remote add origin https://gitup.uni-potsdam.de/youraccount/
    yourproject.git
```

A.2.3 Stage Area, Commits and Push

Now that you have your own git repository, you probably want to work on it. Just make changes in the directory's git repository in your file browser as usual. If you want to save your changes, you have to communicate it to git. Go back to the bash and type in

```
$ git add --all
```

Now, *all* changes are moved to the so-called ***stage area***.[1] Roughly speaking, your changes are now saved in an intermediate state. To make these changes official, you have to ***commit*** your changes:

```
$ git commit -m 'a short message about your changes'
```

When you want to update your remote git project with these changes, type

```
$ git push
```

Your remote repository should be up-to-date now.

A.2.4 Branches

Imagine you work together with a fellow student on a common git project, and both of you push their changes at the same time. Evidently, this causes conflicts. For this reason, git provides ***branches***. The main branch, where all changes converge, is the `master` branch. To switch to `master`, type

```
$ git checkout master
```

as we did before for our pull request. If you want to create a new branch from `master`, simply write

```
$ git checkout -b your_new_branch
```

Now, all commits are done in this branch without affecting the `master` branch. To share this branch and all its changes on the remote server, type

```
$ git push -u origin your_new_branch
```

[1] Notice that you can also move single files to the stage area. Just type the file name instead of `--all`.

When you have finished your work, you should ***merge*** it into the branch `master` again. Make sure to inform your coworkers about this.[2] Fortunately, git provides powerful algorithms to do the merging automatically. Change to the `master` branch again and use´

```
$ git merge your_new_branch
```

If the algorithm fails, the corresponding conflicts will be reported. If that happens, fix them manually and finally commit the changes. Usually, your new branch is not needed anymore. You can delete it with

```
$ git branch -D your_new_branch
```

A.2.5 Workflow Example

A typical workflow with git can look as follows:

1. Open the git-repository (using the terminal or a git-client).
2. Download the current master branch from the remote server:

   ```
   $ git pull
   ```

3. Create a new branch for your changes:

   ```
   $ git git checkout -b mychanges master
   ```

4. Work on the project.
5. Stage the changed files:

   ```
   $ git add --all
   ```

6. Commit the changes.

   ```
   $ git commit -m 'nothing done'
   ```

7. Change back to the `master` branch:

   ```
   $ git checkout master
   ```

8. Pull the newest code from the remote server:

   ```
   $ git pull
   ```

[2] For this reason, the GitLab platform provides so-called ***merge requests***. This is a convenient way to double-ckeck changes with your coworkers.

9. Merge the new branch with the master branch:

```
$ git merge mychanges
```

10. Delete the branch `mychanges`:

```
$ git branch -D mychanges
```

11. Inform your coworkers about the changes and push the newest version to the remote server:

```
$ git push
```

Bibliography

Online

[1] Dataquest. *Jupyter Notebook for Beginners: A Tutorial*. URL: https://www.dataquest.io/blog/jupyter-notebook-tutorial/. Accessed: 2020-04-12 (cited on page 15).

[2] Project Jupyter. *Homepage*. URL: https://jupyter.org/. Accessed: 2020-04-11 (cited on page 15).

[3] Project SageMath. *Sage Reference Manual*. URL: https://doc.sagemath.org/html/en/reference/. Accessed: 2020-05-12 (cited on pages 94, 112).

[5] The Daring Fireball. *Markdown: Syntax*. URL: https://daringfireball.net/projects/markdown/syntax/. Accessed: 2020-04-12 (cited on page 16).

[6] Vel. *The Legrand Orange Book*. URL: https://www.latextemplates.com/template/the-legrand-orange-book. Accessed: 2020-04-11 (cited on page 2).

Books

[4] Victor Shoup. *A computational introduction to number theory and algebra*. Second. Cambridge University Press, Cambridge, 2009, pages xviii+580. ISBN: 978-0-521-51644-0 (cited on page 136).

[7] Paul Zimmermann et al. *Computational Mathematics with SageMath*. Freely downloadable from: http://sagebook.gforge.inria.fr/english.html. Dec. 2018. Accessed: 2020-04-11 (cited on pages 2, 193).

Index

A

algebraic number field 127
assumptions 24

B

Bézout relation 144
binomial cofficient 17
boolean 124

C

character Strings 68
characteristic polynomial 173
characteristic polynomials 174
code cell 15
color maps 109
colormaps 92
compount domains 125
comprehension form 61
computational domain 121
conditional 51
contour plot 93

D

data structures 59
density plot 92
derivative 36
dictionary 71
differential equations 217
duplicated data structure 72

E

eigenspace 173, 179
eigenvalues 173, 179
eigenvectors 179
elementary domain 122
elimination ideal 202
enumerative loop 52
equation 28
Euclidean algorithm 144

F

factorial 17
factorization of a polynomial . . 148
finite field 132
floating-point numbers 123
    real floating-point numbers . 123
for loop . . . . . *see* enumerative loop
formal power series 152
    truncated power series 153
fraction 27
fraction field 126, 150
free module 159

G

Gaussian elimination 165
git 247

graphics primitive 95

H

homogeneous function 224

I

ideal
    variety 195
ideals 145
immutable data structure 72
integer modulo ring 124
integers 122
integral
    definite integral 37
    indefinite integral 38
irreducible polynomial 148

J

Jupyter 15
Jupyter Notebook . . . . . *see* Jupyter

K

Krylov sequence 174

L

lambda construction 50
laplace transform 234
lazy power series 156
limit 34
linear system 170
list 59

M

Markdown cell 16
matrices 125
matrix group 160
matrix normal form
    Frobenius normal form 177
    Hermite normal form 168
    Jacobi normal form 181
    Smith normal form 168
    reduced echelon form 167
minimal polynomial 174
multivariate polynomial ring . . . 125
mutable data structure 72

N

normal form 122
numerical approximation 17

P

Padé approximation 154
parent 121
pivot 166
plot 18, 77
plot3d 18, 107
polynomial 27, 125
polynomial ring 125, 140
polynomial system 185
primality tests 136
    pseudo-primality tests 136
    true primality tests 136
print 57
Project Jupyter . . . . . *see* Jupyter
Python function 49
Python variable 21

Q

quotient ring 146, 197

R

radical of an ideal 200
rank profile 170
Rational Field 123
rational function 27, 150
rational reconstruction 133
roots 31
roots of a polynomial 149

S

sequence 34
series 34
    power series 35
    Taylor series 35
Sets 68
shared data structure 72
simplification 25
square-free decomposition 149
substitute 23
sum 32
symbolic expression 23, 128
symbolic function 32
symbolic ring 128
symbolic variable 22

T

triangular decomposition 209
trigonometric functions 18
tuples 73

V

variable . . . . . *see* Python variable, symbolic variable

vector 40
vector spaces 159

W
while loop 54

Z
zero-dimensional ideals 208