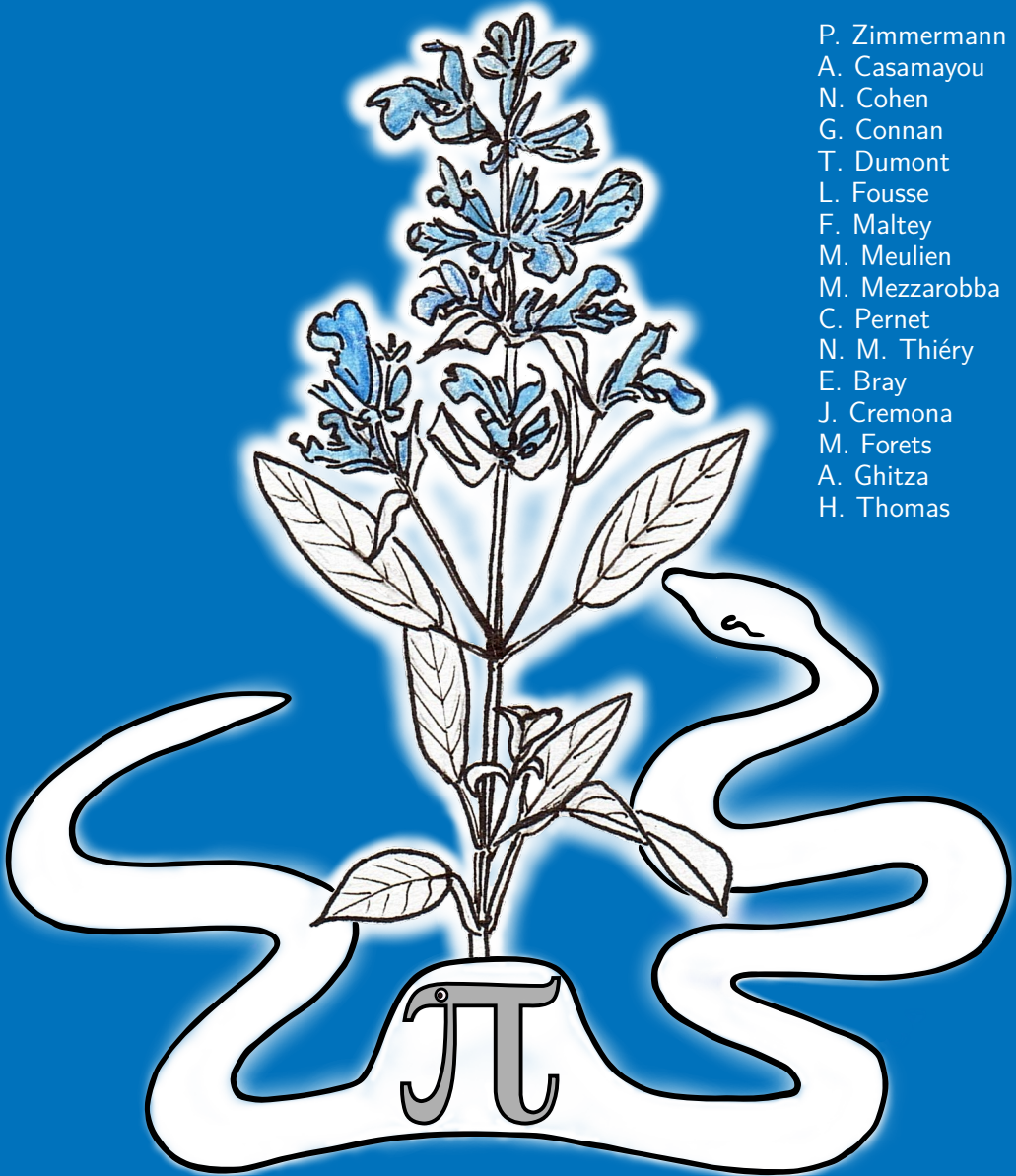
Computational Mathematics with SageMath

P. Zimmermann
A. Casamayou
N. Cohen
G. Connan
T. Dumont
L. Fousse
F. Maltey
M. Meulien
M. Mezzarobba
C. Pernet
N. M. Thiéry
E. Bray
J. Cremona
M. Forets
A. Ghitza
H. Thomas



Computational Mathematics with SageMath

Paul Zimmermann Alexandre Casamayou
Nathann Cohen Guillaume Connan Thierry Dumont
Laurent Fousse François Maltey Matthias Meulien
Marc Mezzarobba Clément Pernet Nicolas M. Thiéry
Erik Bray John Cremona Marcelo Forets
Alexandru Ghitza Hugh Thomas

Parts of this book are inherited from the book *Calcul formel : mode d'emploi. Exemples en Maple* from Philippe Dumas, Claude Gomez, Bruno Salvy and Paul Zimmermann, distributed under cc-by-sa 2.0 fr, in particular Sections 2.1.5, 2.3.5 and 5.3.

Parts of the Sage examples from Chapter 15 are inherited from the tutorials of MuPAD-Combinat [HT04] and Sage-combinat. The enumeration of complete binary trees in §15.1.2 is partly inspired from a classroom problem designed by Florent Hivert.

Exercise 9 on Gauss problem is inspired from a problem designed by François Pantigny, and Exercise 17 on the Magnus effect is extracted from a classroom problem designed by Jean-Guy Stoliaroff.

Graphs from Figure 4.9 and their interpretation reproduce part of paragraph III.4 from the "Que sais-je ?" book *Les nombres premiers* from Gérald Tenenbaum and Michel Mendès France.

Preface

This book was written for those who want to efficiently use a computer algebra system, and Sage in particular. Symbolic computation systems offer plenty of functionality, and finding the right approach or command to solve a given problem is sometimes difficult. A reference manual provides a detailed analytic description of each function of the system; however, this is not very useful since usually we do not know in advance the name of the function we are looking for! This book provides another approach, by giving a global and synthetic point of view, while insisting on the underlying mathematics, the classes of problems we can solve and the corresponding algorithms.

The first part, more specific to Sage, will help getting to grips with this system. This part is written to be understood by undergraduate students, and partly by high school students. The other parts cover more specialised topics encountered in undergraduate and graduate studies. Unlike in a reference manual, the mathematical concepts are clearly explained before illustrating them with Sage. This book is thus in the first place a book about mathematics.

To illustrate this book, Sage was a natural choice, since it is an open-source system, that anybody can use, modify and redistribute at will. In particular the student who learns Sage in high school will be able to continue to use it at undergraduate or graduate levels, in a company, etc. Sage is still a relatively young system, and despite its already extensive capacities, it does contain some bugs. However, thanks to its very active community of developers, Sage evolves very quickly. Every Sage user can report a bug — maybe together with its solution — on `trac.sagemath.org` or via the `sage-support` list.

In writing this book, we have used version 8.2 of Sage. Nevertheless, the examples should still work with later versions. However, some of the explanations may no longer hold, for example the fact that Sage relies on Maxima for numerical integrals.

When in December 2009 I asked Alexandre Casamayou, Guillaume Connan, Thierry Dumont, Laurent Fousse, François Maltey, Matthias Meulien, Marc Mezzarobba, Clément Pernet and Nicolas Thiéry to write the first version (in French) of this book, all agreed with enthusiasm — including Nathann Cohen who joined us later on. Given the success of the French version, it was clear that an English version would be welcome. In March 2017, I decided to start working on the English version; I want to thank once again those of the "dream team" who helped me translating the text into English, updating the examples to the new version of Sage, and moreover improving the content of the book

(Guillaume Connan, Thierry Dumont, Clément Pernet, Nicolas Thiéry), as well as the new authors of the English version (Erik Bray, John Cremona, Marcelo Forets, Alexandru Ghitza, Hugh Thomas).

Several people had proof-read the French version: Gaëtan Bisson, Françoise Jung, Hugh Thomas, Anne Vaugon, Sébastien Desreux, Pierrick Gaudry, Maxime Huet, Jean Thiéry, Muriel Shan Sei Fan, Timothy Walsh, Daniel Duparc, and especially Kévin Rowanet and Kamel Naroun. The following people helped us to improve the English version by proof-reading one or several chapters, or simply reporting a typo: Fredrik Johansson, Pierre-Jean Spaenlehauer, Jacob Appelbaum, Nick Higham, Helmut Büch, Shashank Singh, Annegret Wagler, Bruno Grenet, Daniel S. Roche, Jeroen Demeyer, Evans Doe Ocansey, Minh Van Nguyen, Simon Willerton, and last but not least Adil Hasan and Dimitris Papachristoudis for their wonderful feedback. On the technical and typographic side, we thank Emmanuel Thomé, Sylvain Chevillard, Gaëtan Bisson, Jérémie Detrey and Denis Roegel.

When writing this book, we have learned a lot about Sage, and we have of course encountered some bugs — some of which have already been fixed. We hope this book will be also useful to others, high school students, undergraduate or graduate students, engineers, researchers or simply mathematical hobbyists. Despite several proof-readings, this book is surely not perfect, and we expect the reader to tell us about any error, typo or make any suggestion, by referring to the page `sagebook.gforge.inria.fr`.

Nancy, France
May 2018

Paul Zimmermann

Contents

Part I

Getting to Grips with Sage

1 First Steps

This introductory chapter presents the way the Sage mathematical system thinks. The next chapters of this first part develop the basic notions: how to make symbolic or numerical computations in analysis, how to work with vectors or matrices, write programs, deal with data lists, produce graphics, etc. The following parts of this book treat in more detail some branches of mathematics where computers are very helpful.

1.1 The Sage Program

1.1.1 A Tool for Mathematics

Sage is a piece of software implementing mathematical algorithms in a variety of contexts. To start with, it can be used as a scientific pocket calculator, and can manipulate all sorts of numbers, from integers and rational numbers to numerical approximations of real and complex numbers with arbitrary precision, and also including elements of finite fields.

However, mathematical computations go far beyond numbers: Sage is a *computer algebra system*; it can for example help junior high school students learn how to solve linear equations, or develop, factor, or simplify expressions; or carry out such operations in arbitrary rings of polynomials or rational function fields. In analysis, Sage can manipulate expressions involving square roots, exponentials, logarithms or trigonometric functions: integration, computation of limits, simplification of sums, series expansion, solution of certain differential equations, and more. In linear algebra it computes with vectors, matrices, and subspaces. It can also help illustrate and solve problems in probability, statistics, and combinatorics.

To summarise, Sage strives to provide a consistent and uniform access to features in a wide area of mathematics — ranging from group theory to numerical analysis — and beyond — visualisation in two and three dimensions, animation, networking, databases, ... Using a single unified piece of software frees the (budding) mathematician from having to transfer data between several tools and learn the syntax of several programming languages.

Access to Sage

To use Sage, all that is needed is a web browser. As a starter, the service http://sagecell.sagemath.org/ allows for testing commands. To go further, one can use one of the online services. For example, CoCalc (http://cocalc.com, formerly known as SageMathCloud) gives access to a lot of computational software and collaborative tools, together with course management features. Developed and hosted by SageMathInc, an independent company founded by William Stein, its access is free for casual use, and most of its code is free. Other similar services are hosted by universities and institutions. Ask around to find out what is available near you.

For regular usage, it is recommended to use Sage on one's own machine, installing it if this has not yet been done by the system administrator. Sage is available for most operating systems: Linux, Windows, MacOS; see the `Download` section on http://sagemath.org.

How to start Sage depends on the environment; therefore we do not go into details here. On CoCalc one needs to create an account, a project, and finally a Jupyter worksheet. On a desktop, the system may provide a startup icon. Under Linux or MacOS, one typically would launch the command `sage --notebook jupyter` in a terminal.

Resources

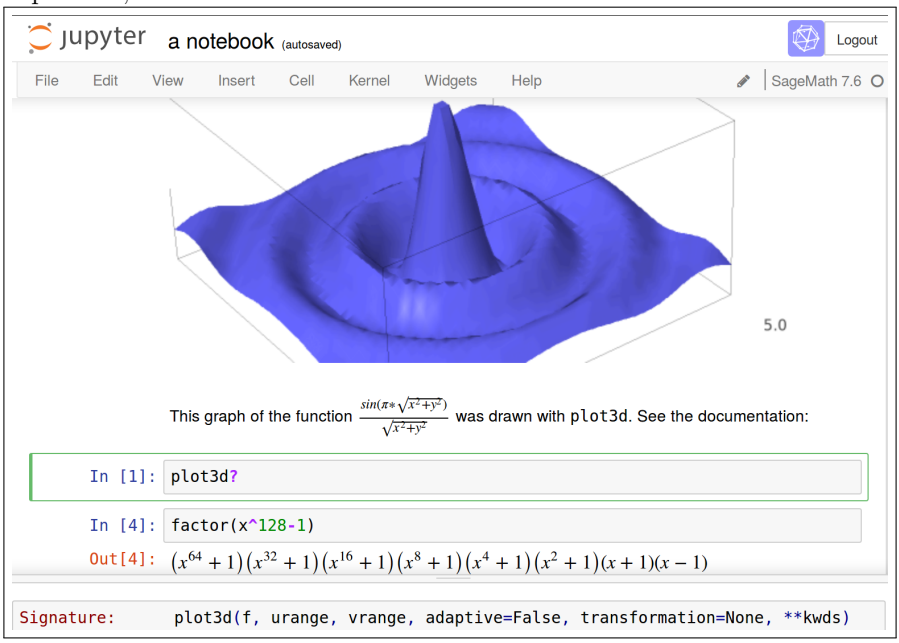
The official Sage website offers many resources:

http://www.sagemath.org/	official site
http://doc.sagemath.org/	documentation
http://wiki.sagemath.org/quickref	command lists

To get help on using Sage, the Question and Answer site http://ask.sagemath.org/ is very active. For technical questions (installation, troubleshooting, ...), the best medium is the mailing list sage-support@googlegroups.com.

User interfaces: notebooks or command line

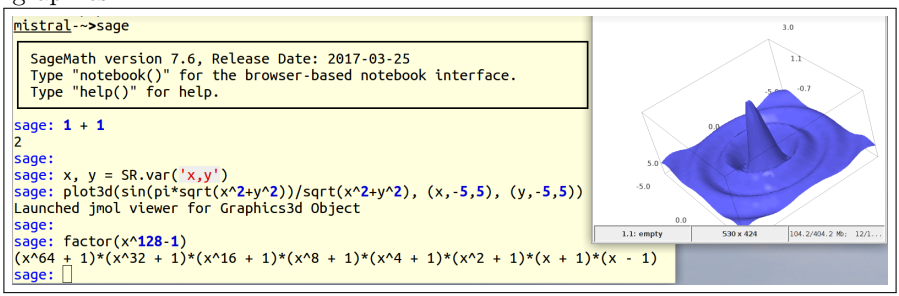
However Sage is accessed, one can use it via a web application enabling the edition and sharing of *notebooks* which mix code, interactive computations, equations, visualisations and text:



The *Help* menu gives access to the documentation. We recommend starting with the *User Interface Tour*, returning often to the *Keyboard Shortcuts*, and progressively exploring the *Thematic Tutorials*.

Sage uses Jupyter as web application. Formerly known as IPython, Jupyter allows the use of a great deal of mathematical software (GAP, PARI/GP, or Singular, ...) and beyond (from Python to C++!), and is supported by a large community. Notebooks are respectively in the `.sws` and `.ipynb` format. CoCalc offers another format `.sagews` which is less portable, but explores advanced interaction features.

As an alternative, one can use Sage in a terminal. Its calculator-like *command line interface* gives full access to all of its capabilities, including graphics:



Sage and Python

Like most software for mathematical computations, Sage is used by issuing commands written in a programming language. For this purpose, Sage uses the general purpose programming language Python, with just a tiny layer of syntactic sugar to support some common mathematical notations in interactive use. For complicated or just repetitive calculations, one can write programs instead of simple one-line commands. When mature and of general interest, such programs can be submitted for inclusion in Sage.

Aims and history of Sage

In 2005, William Stein, an American academic, initiated the Sage project, with the goal of producing free software for mathematical computation, developed by users for users. This meets a longstanding need of mathematicians, and soon an international community of hundreds of developers crystallised around Sage, most of them teachers and researchers. At first Sage had some focus on number theory, the area of interest of its founder. As contributions flowed in, its capabilities progressively extended to many areas of mathematics. This, together with the numerical capabilities brought in by the Scientific Python ecosystem, has made Sage the general purpose mathematics software that it is today.

Not only can Sage be used and downloaded for free, but it is free software: the authors impose no restriction on its usage, redistribution, study or modification, as long as the modifications are free themselves. In the same spirit, the material in this book can be freely read, shared, and reused (with proper credit, of course). This license is in harmony with the spirit of free development and dissemination of knowledge in academia.

Sage, a software in an ecosystem

The development of Sage was relatively quick thanks to its strategy of reusing existing free software, including many specialised mathematical libraries or systems like GAP, PARI/GP, Maxima, Singular, to cite just a few.

Sage itself is written in Python, a programming language used by millions and known for the ease with which it can be learned. Python is particularly well established in the sciences. Within the same computing environment, it is possible to combine the capabilities of Sage with scientific libraries for numerical computations, data analysis, statistics, visualisation, machine learning, biology, astrophysics, and technical libraries for networking, databases, web, ... See for example: https://en.wikipedia.org/wiki/List_of_Python_software.

SAGE, Sage, or SageMath?

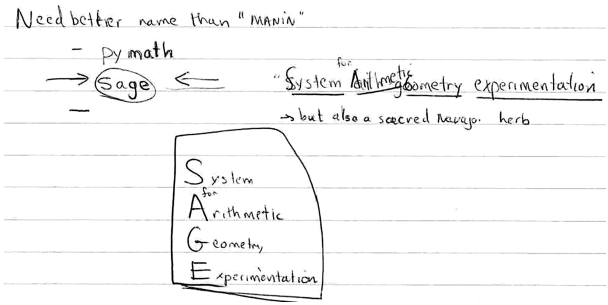


FIGURE 1.1 – The first occurrence of the name Sage, on a handwritten note of W. Stein.

At first, Sage was both an acronym and a reference to the "sage" medicinal plant. When the system later expanded to cover much of mathematics, the acronym part was dropped. As Sage came to be known in larger circles, and to avoid confusion with, for example, the business management software of the same name, the official name was changed to SageMath. When the context is unambiguous, for example in this book, it is traditional to just use Sage.

1.2 Sage as a Calculator

1.2.1 First Computations

In the rest of the book, we present computations in the following form, which mimics a command line Sage session:

```
sage: 1+1
2
```

The `sage:` text in the beginning of the first line is the command prompt of the system. The prompt (which does not appear in the notebook interface) means that Sage awaits a user command. The rest of the line is the command to execute, which is validated with the ⟨Enter⟩ key. The lines below are the system's answer, which in general are the results of the computation. Some commands use several lines (see Chapter 3). The additional command lines can then be recognised by .... at the beginning of the line. A multi-line command should follow the position of linebreaks and the indentation (spaces to align the line with respect to the previous one), without copying the initial .....

In the notebook, one directly enters the commands in a computation cell, and validates by clicking on *evaluate* or using the ⟨Shift⟩+⟨Enter⟩ key combination. The combination ⟨Alt⟩+⟨Enter⟩ not only executes the command of the current cell, but also creates a new cell just below. One can also create a new cell by clicking in the small space just above a given cell, or below the last cell.

Sage interprets simple formulas like a scientific calculator. The operations +, ×, etc. have their usual precedence, and parentheses their common usage:

```
sage: ( 1 + 2 * (3 + 5) ) * 2
34
```

The * character above stands for multiplication, which should not be omitted, even in expressions like $2x$. The power operation is written ^ or **:

```
sage: 2^3
8
sage: 2**3
8
```

and the division is denoted by /:

```
sage: 20/6
10/3
```

Please note the exact computation: the result of the above division, after simplification, is the rational number 10/3 and not an approximation like 3.33333. There is no limit[1] to the size of integers or rational numbers:

```
sage: 2^10
1024
sage: 2^100
1267650600228229401496703205376
sage: 2^1000
1071508607186267320948425049060001810561404811705533607443750\
3883703510511249361224931983788156958581275946729175531468251\
8714528569231404359845775746985748039345677748242309854210746\
0506237114187795418215304647498358194126739876755916554394607\
7062914571196477686542167660429831652624386837205668069376
```

To obtain a numerical approximation, one simply writes one of the numbers with a decimal point (one could replace 20.0 by 20. or 20.000):

```
sage: 20.0 / 14
1.42857142857143
```

Besides, the numerical_approx function gives a numerical approximation of an expression:

```
sage: numerical_approx(20/14)
1.42857142857143
sage: numerical_approx(2^1000)
1.07150860718627e301
```

Numerical approximations can be computed to arbitrarily large precisions. For example, let us increase the precision to 60 digits to exhibit the periodicity of the digit expansion of a rational number:

[1] Except that due to the available memory of the computer used.

Basic arithmetic operations	
"four operations"	`a+b`, `a-b`, `a*b`, `a/b`
power	`a^b` or `a**b`
square root	`sqrt(a)`
n-th root	`a^(1/n)`

Integer operations	
integer division	`a // b`
remainder	`a % b`
quotient and remainder	`divmod(a,b)`
factorial $n!$	`factorial(n)`
binomial coefficient $\binom{n}{k}$	`binomial(n,k)`

Usual functions on real numbers, complex numbers, ...	
integer part	`floor(a)`
absolute value, modulus	`abs(a)`
elementary functions	`sin`, `cos`, ... (see Table 2.2)

TABLE 1.1 – Some usual operations.

```
sage: numerical_approx(20/14, digits=60)
1.42857142857142857142857142857142857142857142857142857142857
```

Differences between exact and numerical computations are discussed in the sidebar on page 10.

The operators `//` and `%` yield the quotient and remainder of the division of two integers:

```
sage: 20 // 6
3
sage: 20 % 6
2
```

Several other functions apply to integers. Among those specific to integers are the factorial and the binomial coefficients (see Table 1.1):

```
sage: factorial(100)
933262154439441526816992388562667004907159682643816\21\
468592963895217599993229915608941463976156518286253\69\
792082722375825118521091686400000000000000000000000
```

Here is a way to decompose an integer into prime factors. We will return to this problem in Chapter 5, then once more in Chapter 6.

```
sage: factor(2^(2^5)+1)
641 * 6700417
```

Fermat had conjectured that all integers $2^{2^n}+1$ are prime. The above example is the smallest counter-example.

Computer algebra and numerical methods

A computer algebra system is a program made to manipulate, simplify and compute mathematical formulas by applying only exact (i.e., symbolic) transformations. The term *symbolic* is opposed here to *numerical*; it means that computations are made using algebraic formulas, manipulating symbols only. This is why *symbolic computation* is sometimes used in place of *computer algebra*. In French, one says *calcul formel* or sometimes *calcul symbolique*.

In general, pocket calculators manipulate integers exactly up to twelve digits; larger numbers are rounded, which induces errors. Thus a pocket calculator wrongly evaluates to 0 the following expression, whereas the correct result is 1:

$$(1 + 10^{50}) - 10^{50}.$$

Such errors are difficult to detect if they arise during an intermediate computation, without being anticipated by a theoretical analysis. On the contrary, computer algebra systems do not have these limitations, and perform all integer computations exactly: they answer 1 to the previous computation.

Numerical methods approximate to a given precision (using the trapezoidal rule, Simpson's rule, Gaussian quadrature, etc.) the definite integral $\int_0^\pi \cos t \, \mathrm{d}t$ to obtain a numerical result near zero (with error 10^{-10} for example). However, they cannot tell the user if the result is exactly 0, or on the contrary is near zero but definitively not zero.

A computer algebra system rewrites using symbolic mathematical transformations the integral $\int_0^\pi \cos t \, \mathrm{d}t$ into the expression $\sin \pi - \sin 0$, which is then evaluated into $0 - 0 = 0$. This method proves whence $\int_0^\pi \cos t \, \mathrm{d}t = 0$.

However, algebraic transformations have limits too. Most expressions handled by symbolic computation systems are rational functions, and the expression a/a is automatically simplified into 1. This automatic simplification is not compatible with solving equations; indeed, the solution to the equation $ax = a$ is $x = a/a$, which is simplified into $x = 1$ without distinguishing the special case $a = 0$, for which any scalar x is solution (see also §2.1.5).

1.2.2 Elementary Functions and Usual Constants

The usual functions and constants are available (see Tables 1.1 and 1.2), as well as for complex numbers. Here also, computations are exact:

```
sage: sin(pi)
0
sage: tan(pi/3)
sqrt(3)
sage: arctan(1)
1/4*pi
sage: exp(2 * I * pi)
1
```

even if symbolic expressions are returned instead of numerical expressions:

Some specials values	
boolean values "true" and "false"	`True`, `False`
imaginary unit i	`I` *or* `i`
infinity ∞	`Infinity` *or* `oo`

Common mathematical constants	
Archimedes' constant π	`pi`
logarithm basis $e = \exp(1)$	`e`
Euler-Mascheroni constant γ	`euler_gamma`
golden ratio $\varphi = (1+\sqrt{5})/2$	`golden_ratio`
Catalan's constant	`catalan`

TABLE 1.2 – Predefined constants.

```
sage: arccos(sin(pi/3))
arccos(1/2*sqrt(3))
sage: sqrt(2)
sqrt(2)
sage: exp(I*pi/7)
e^(1/7*I*pi)
```

One does not always get the expected results. Indeed, only few simplifications are done automatically. If needed, it is possible to explicitly call a simplification function:

```
sage: simplify(arccos(sin(pi/3)))
1/6*pi
```

We will see in §2.1 how to tune the simplification of expressions. Of course, it is also possible to compute numerical approximations of the results, with an accuracy as large as desired:

```
sage: numerical_approx(6*arccos(sin(pi/3)), digits=60)
3.14159265358979323846264338327950288419716939937510582097494
sage: numerical_approx(sqrt(2), digits=60)
1.41421356237309504880168872420969807856967187537694807317668
```

1.2.3 On-Line Help and Automatic Completion

The reference manual of each function, constant or command is accessed via the question mark `?` after its name:

```
sage: sin?
```

The documentation page contains the function description, its syntax and some examples of usage.

The tabulation key ⟨Tab⟩ after the beginning of a word yields all command names starting with these letters: thus `arc` followed by ⟨Tab⟩ prints the name of all inverse trigonometric and hyperbolic functions:

```
sage: arc<tab>
Possible completions are:
arc  arccos  arccosh  arccot  arccoth  arccsc  arccsch
arcsec  arcsech  arcsin  arcsinh  arctan  arctan2  arctanh
```

1.2.4 Python Variables

To save the result of a computation, one *assigns* it to a *variable*:

```
sage: y = 1 + 2
```

to reuse it later on:

```
sage: y
3
sage: (2 + y) * y
15
```

Note that the result of a computation is not automatically printed when it is assigned to a variable. Therefore, we will do the following to also print it,

```
sage: y = 1 + 2; y
3
```

the ';' character separating several instructions on the same line. Since the computation of the result is done before the assignment, one can reuse the same variable:

```
sage: y = 3 * y + 1; y
10
sage: y = 3 * y + 1; y
31
sage: y = 3 * y + 1; y
94
```

Additionally, Sage saves the last three results in the special variables `_`, `__` and `___`:

```
sage: 1 + 1
2
sage: _ + 1
3
sage: __
2
```

The variables we have used above are Python variables; we will discuss them further in §3.1.3. Let us just mention that it is not recommended to redefine predefined constants and functions from Sage. While it does not influence the internal behaviour of Sage, it could yield surprising results:

```
sage: pi = -I/2
sage: exp(2*I*pi)
```

```
e
```

To restore the original value, one can type for example:

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

or alternatively

```
sage: restore()
```

which restores to their default value *all* predefined variables and functions. The `reset()` function performs an even more complete reset, in particular it clears all user-defined variables.

1.2.5 Symbolic Variables

We have played so far with constant expressions like $\sin(\sqrt{2})$, but Sage is especially useful in dealing with expressions containing variables like $x + y + z$ or $\sin(x) + \cos(x)$. The "mathematician's" *symbolic variables* x, y, z appearing in those expressions differ in general from the "programmer's" variables encountered in the preceding section. On this point, Sage differs notably from other computer algebra systems like Maple or Maxima.

The symbolic variables should be explicitly declared before being used[2] (`SR` abbreviates *Symbolic Ring*):

```
sage: z = SR.var('z')
sage: 2*z + 3
2*z + 3
```

In this example, the command `SR.var('z')` *builds* and returns a symbolic variable whose name is z. This symbolic variable is a perfect Sage object: it is handled exactly like more complex expressions like $\sin(x)+1$. Then, this symbolic variable is assigned to the "programmer's" variable `z`, which enables one to use it like any other expression, to build more complex expressions.

We could have assigned z to another variable than `z`:

```
sage: y = SR.var('z')
sage: 2*y + 3
2*z + 3
```

Hence, assigning the symbolic variable z to the Python variable `z` is just a convention, which is however recommended to avoid confusion.

Conversely, the Python variable `z` does not interact with the symbolic variable z:

```
sage: c = 2 * y + 3
sage: z = 1
sage: 2*y + 3
2*z + 3
sage: c
```

[2] Except the symbolic variable x, which is predefined in Sage.

```
2*z + 3
```

How can we give a value to a symbolic variable appearing in an expression? One uses the *substitution* operation, as in:

```
sage: x = SR.var('x')
sage: expr = sin(x); expr
sin(x)
sage: expr(x=1)
sin(1)
```

The substitution in symbolic expressions is discussed in detail in the next chapter.

Exercise 1. Explain step by step what happens during the following instructions:

```
sage: u = SR.var('u')
sage: u = u+1
sage: u = u+1
sage: u
u + 2
```

As it would become tedious to create a large number of symbolic variables, there exists a shortcut `x = SR.var('x', n)` where n is a positive integer (notice that indexing starts at 0):

```
sage: x = SR.var('x', 100)
sage: (x[0] + x[1])*x[99]
(x0 + x1)*x99
```

The command `var('x')` is a convenient alternative for `x = SR.var('x')`[3]:

```
sage: var('a, b, c, x, y')
(a, b, c, x, y)
sage: a * x + b * y + c
a*x + b*y + c
```

If the explicit declaration of symbolic variables is too cumbersome, it is also possible to emulate the behaviour of systems like Maxima or Maple. However, this functionality is only available in the notebook interface (but not the Jupyter worksheet). Thus in the notebook, after:

```
sage: automatic_names(True)
```

every use of an unassigned variable yields the creation of a symbolic variable of the same name and its assignment:

```
sage: 2 * bla + 3
2*bla + 3
sage: bla
bla
```

[3]In this book, we will often write `x = var('x')` instead of the better but cumbersome form `x = SR.var('x')`, to avoid the output produced by `var('x')`.

1.2.6 First Graphics

The `plot` command makes it easy to draw the curve of a real function on a given interval. The `plot3d` command is its counterpart for three-dimensional graphics, or for the graph of a real function of two variables. Here are examples of those two commands:

```
sage: plot(sin(2*x), x, -pi, pi)
sage: plot3d(sin(pi*sqrt(x^2 + y^2))/sqrt(x^2+y^2),
....:        (x,-5,5), (y,-5,5))
```

The graphical capacities of Sage are much wider. We will explore them in more detail in Chapter 4.

2 Analysis and Algebra

This chapter uses simple examples to describe the useful basic functions in analysis and algebra. Students will be able to replace *pen and paper* by *keyboard and screen* while keeping the same intellectual challenge of understanding mathematics.

This presentation of the main calculus commands with Sage should be accessible to young students; some parts marked with an asterisk are reserved for higher-level students. More details are available in the other chapters.

2.1 Symbolic Expressions and Simplification

2.1.1 Symbolic Expressions

Sage allows a wide range of analytic computations on *symbolic expressions* formed with numbers, symbolic variables, the four basic operations, and usual functions like `sqrt`, `exp`, `log`, `sin`, `cos`, etc. A symbolic expression can be seen as a tree like in Figure 2.1. It is important to understand that a symbolic expression is a *formula* and not a value or a mathematical function. Thus, Sage does not recognise the two following expressions as equal[1]:

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

Thanks to the commands presented in this chapter, the user can transform expressions into the desired form.

[1]The equality test `==` is not only a syntactic comparison: for example, the expressions `arctan(sqrt(2))` and `pi/2-arctan(1/sqrt(2))` are considered equal. In fact, when one compares two expressions with `bool(x==y)`, Sage tries to prove that their difference is zero, and returns `True` if that succeeds.

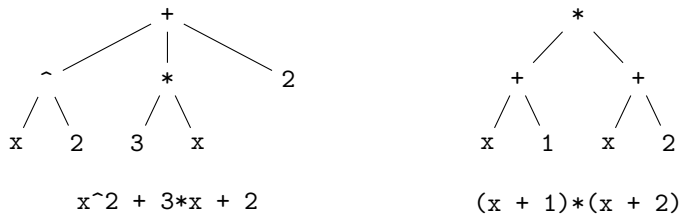


FIGURE 2.1 – Two symbolic expressions representing the same mathematical object.

The most common operation consists of *evaluating* an expression by giving a value to some of its parameters. The `subs` method — which can be made implicit — performs this transformation:

```
sage: a, x = var('a, x'); y = cos(x+a) * (x+1); y
(x + 1)*cos(a + x)
sage: y.subs(a=-x); y.subs(x=pi/2, a=pi/3); y.subs(x=0.5, a=2.3)
x + 1
-1/4*sqrt(3)*(pi + 2)
-1.41333351100299
sage: y(a=-x); y(x=pi/2, a=pi/3); y(x=0.5, a=2.3)
x + 1
-1/4*sqrt(3)*(pi + 2)
-1.41333351100299
```

Compared to the usual mathematical notation $x \mapsto f(x)$, the variable which is substituted must be explicitly given. The substitution of several parameters is done in parallel, while successive substitutions are performed in sequence, as shown by the two examples below:

```
sage: x, y, z = var('x, y, z') ; q = x*y + y*z + z*x
sage: bool(q(x=y, y=z, z=x) == q), bool(q(z=y)(y=x) == 3*x^2)
(True, True)
```

To replace an expression more complex than a single variable, the `substitute` method is available:

```
sage: y, z = var('y, z'); f = x^3 + y^2 + z
sage: f.substitute(x^3 == y^2, z==1)
2*y^2 + 1
```

2.1.2 Transforming Expressions

The simplest non-constant expressions are polynomials and rational functions of one or more variables. The functions allowing to rewrite expressions in several forms or to put them in normal form are summarised in Table 2.1. For example, the `expand` method is useful to expand polynomials:

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

Symbolic Functions

Sage allows also to define *symbolic functions* to manipulate expressions:

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

A symbolic function is just like an expression that we can call like a command and where the order of variables is fixed. To convert a symbolic expression into a symbolic function, we use either `f(x) = ...`, or the `function` method:

```
sage: y = var('y'); u = sin(x) + x*cos(y)
sage: v = u.function(x, y); v
(x, y) |--> x*cos(y) + sin(x)
sage: w(x, y) = u; w
(x, y) |--> x*cos(y) + sin(x)
```

Symbolic functions are useful to represent mathematical functions. They differ from Python functions or *procedures*, which are programming constructions described in Chapter 3. The difference between symbolic functions and Python functions is similar to the difference between symbolic variables and Python variables, described in §1.2.5.

A symbolic function can be used like an expression, which is not the case for Python functions; for example, the `expand` method does not exist for the latter.

```
sage: p = (x+y)*(x+1)^2
sage: p2 = p.expand(); p2
x^3 + x^2*y + 2*x^2 + 2*x*y + x + y
```

whereas the `collect` method groups terms together according to the powers of a given variable:

```
sage: p2.collect(x)
x^3 + x^2*(y + 2) + x*(2*y + 1) + y
```

Those methods do not only apply to polynomials in symbolic variables, but also to polynomials in more complex sub-expressions like $\sin x$:

```
sage: ((x+y+sin(x))^2).expand().collect(sin(x))
x^2 + 2*x*y + y^2 + 2*(x + y)*sin(x) + sin(x)^2
```

For rational functions, the `combine` method enables us to group together terms with common denominator; the `partial_fraction` method performs the partial fraction decomposition over $\mathbb{Q}$. (To specify a different decomposition field, we refer the reader to §7.4.)

The more useful representations are the expanded form for a polynomial, and the reduced form P/Q with P and Q expanded in the case of a fraction. When

Polynomial $p = zx^2 + x^2 - (x^2 + y^2)(ax - 2by) + zy^2 + y^2$	
`p.expand()`	$-ax^3 + 2\,bx^2y - axy^2 + 2\,by^3 + x^2z + y^2z + x^2 + y^2$
`p.expand().collect(x)`	$-ax^3 - axy^2 + 2\,by^3 + (2\,by + z + 1)x^2 + y^2z + y^2$
`p.collect(x).collect(y)`	$2\,bx^2y + 2\,by^3 - (ax - z - 1)x^2 - (ax - z - 1)y^2$
`p.factor()`	$-(ax - 2\,by - z - 1)\left(x^2 + y^2\right)$
`p.factor_list()`	$\left[(ax - 2\,by - z - 1, 1)\,, \left(x^2 + y^2, 1\right), (-1, 1)\right]$

Fraction $r = \frac{x^3+x^2y+3\,x^2+3\,xy+2\,x+2\,y}{x^3+2\,x^2+xy+2\,y}$	
`r.simplify_rational()`	$\frac{x^2+(x+1)y+x}{x^2+y}$
`r.factor()`	$\frac{(x+y)(x+1)}{x^2+y}$
`r.factor().expand()`	$\frac{x^2}{x^2+y} + \frac{xy}{x^2+y} + \frac{x}{x^2+y} + \frac{y}{x^2+y}$

Fraction $r = \frac{(x-1)x}{x^2-7} + \frac{y^2}{x^2-7} + \frac{b}{a} + \frac{c}{a} + \frac{1}{x+1}$	
`r.combine()`	$\frac{(x-1)x+y^2}{x^2-7} + \frac{b+c}{a} + \frac{1}{x+1}$

Fraction $r = \frac{1}{(x^3+1)y^2}$	
`r.partial_fraction(x)`	$\frac{-(x-2)}{3\,(x^2-x+1)y^2} + \frac{1}{3\,(x+1)y^2}$

TABLE 2.1 – Polynomials and fractions.

two polynomials or fractions are written in this form, it suffices to compare their coefficients to decide if they are equal: we say they are in *normal form*.

2.1.3 Usual Mathematical Functions

Most mathematical functions are known to Sage, in particular the trigonometric functions, the logarithm and the exponential: they are summarised in Table 2.2.

Knowing how to transform such functions is crucial. To simplify an expression or a symbolic function, the `simplify` method is available:

```
sage: (x^x/x).simplify()
x^(x - 1)
```

However, for more subtle simplifications, the desired kind of simplification should be explicit:

```
sage: f = (e^x-1) / (1+e^(x/2)); f.canonicalize_radical()
e^(1/2*x) - 1
```

For example, to simplify trigonometric expressions, the `simplify_trig` method should be used:

```
sage: f = cos(x)^6 + sin(x)^6 + 3 * sin(x)^2 * cos(x)^2
sage: f.simplify_trig()
1
```

Usual mathematical functions	
Exponential and logarithm	`exp, log`
Logarithm in base a	`log(x, a)`
Trigonometric functions	`sin, cos, tan`
Inverse trigonometric functions	`arcsin, arccos, arctan`
Hyperbolic functions	`sinh, cosh, tanh`
Inverse hyperbolic functions	`arcsinh, arccosh, arctanh`
Integer part, etc.	`floor, ceil, trunc, round`
Square and n-th root	`sqrt, nth_root`
Rewriting trigonometric expressions	
Simplification	`simplify_trig`
Linearisation	`reduce_trig`
Anti-linearisation	`expand_trig`

TABLE 2.2 – Usual functions and simplification.

To linearise (resp. anti-linearise) a trigonometric expression, we use `reduce_trig` (resp. `expand_trig`):

```
sage: f = cos(x)^6; f.reduce_trig()
1/32*cos(6*x) + 3/16*cos(4*x) + 15/32*cos(2*x) + 5/16
sage: f = sin(5 * x); f.expand_trig()
5*cos(x)^4*sin(x) - 10*cos(x)^2*sin(x)^3 + sin(x)^5
```

Expressions containing factorials can also be simplified:

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

The `simplify_rational` method tries to simplify a fraction; whereas to simplify square roots, logarithms or exponentials, the `canonicalize_radical` method is recommended:

```
sage: f = sqrt(abs(x)^2); f.canonicalize_radical()
abs(x)
sage: f = log(x*y); f.canonicalize_radical()
log(x) + log(y)
```

The `simplify_full` command applies the methods `simplify_factorial`, `simplify_rectform`, `simplify_trig`, `simplify_rational` and `expand_sum` (in that order).

All that is needed to determine the variation of a function (derivatives, asymptotes, extrema, localisation of zeroes and graph drawing) can be easily obtained using a computer algebra system. The main Sage operations applying to functions are presented in §2.3.

2.1.4 Assumptions

During a computation, the symbolic variables appearing in expressions are in general considered as taking potentially any value in the complex plane. This

might be a problem when a parameter represents a quantity in a restricted domain (for example, a positive real number).

A typical case is the simplification of the expression $\sqrt{x^2}$. The proper way consists of using the `assume` function, which enables us to define the properties of a variable, which can in turn be reverted by the `forget` instruction:

```
sage: assume(x > 0); bool(sqrt(x^2) == x)
True
sage: forget(x > 0); bool(sqrt(x^2) == x)
False
sage: n = var('n'); assume(n, 'integer'); sin(n*pi)
0
```

2.1.5 Some Pitfalls

The Simplification Problem

The examples of §2.1.5 demonstrate how important normal forms are, and in particular the *test of zero*.

Some families of expressions, like polynomials, have a decision procedure for the equality to zero. As a consequence, for those families, a program is able to decide whether a given expression is zero or not. In most cases, this test is done via the reduction to the normal form: the expression is zero if and only if its normal form is 0.

Unfortunately, not all classes of expressions have a normal form, moreover for some classes it is possible to show that no general method is able to decide in a finite amount of time whether an expression is zero. An example of such a class is made of the rational numbers, the constants π, $\log 2$ and a variable, together with additions, subtractions, multiplications, exponentials and the sine function. The repeated use of `numerical_approx`, while increasing the precision, succeeds in most cases to conjecture if a given expression is zero or not; however, it has been proven impossible to write a computer program taking as input an expression of this class, and returning true if this expression is zero, and false otherwise.

The simplification problem is much harder in those classes. Without any normal form, computer algebra systems can only provide some rewriting functions that the user must play with to obtain some result. Some hope is however possible, if we can identify sub-classes of expressions which have a normal form, and if we know which methods should be applied to compute those normal forms. The Sage approach to handle those issues is presented in more details in Chapter 5.

Let c be a slightly complex expression:

```
sage: a = var('a')
sage: c = (a+1)^2 - (a^2+2*a+1)
```

where we want to solve the equation $cx = 0$ in the variable x:

```
sage: eq =  c * x == 0
```

One might be tempted to divide out this equation by c before solving it:

```
sage: eq2 = eq / c; eq2
x == 0
sage: solve(eq2, x)
[x == 0]
```

Fortunately, Sage avoids this mistake:

```
sage: solve(eq, x)
[x == x]
```

Sage was able to correctly solve the equation because the coefficient c is a polynomial expression. It is thus easy to check whether c is zero, by expanding it:

```
sage: expand(c)
0
```

and use the fact that two mathematically identical polynomials share the same expanded form, or said otherwise, that the expanded form is a normal form for polynomials.

However, on a slightly more complex example, Sage does not avoid the pitfall:

```
sage: c = cos(a)^2 + sin(a)^2 - 1
sage: eq = c*x == 0
sage: solve(eq, x)
[x == 0]
```

even if Sage is able to correctly simplify and test to zero this expression:

```
sage: c.simplify_trig()
0
sage: c.is_zero()
True
```

2.2 Equations

We now deal with equations and how to solve them; the main functions are summarised in Table 2.3.

2.2.1 Explicit Solving

Let us consider the following equation, with unknown z and parameter φ:

$$z^2 - \frac{2}{\cos\varphi} z + \frac{5}{\cos^2\varphi} - 4 = 0, \qquad \text{with } \varphi \in \left]-\frac{\pi}{2}, \frac{\pi}{2}\right[.$$

It is written in Sage:

Scalar equations	
Symbolic solution	`solve`
Roots (with multiplicities)	`roots`
Numerical solving	`find_root`
Vector and functional equations	
Solving linear equations	`solve_right, solve_left`
Solving differential equations	`desolve`
Solving recurrences	`rsolve`

TABLE 2.3 – Solving equations.

```
sage: z, phi = var('z, phi')
sage: eq =  z**2 - 2/cos(phi)*z + 5/cos(phi)**2 - 4 == 0; eq
z^2 - 2*z/cos(phi) + 5/cos(phi)^2 - 4 == 0
```

We can extract the left-hand (resp. right-hand) side with the `lhs` (resp. `rhs`) method:

```
sage: eq.lhs()
```
$z^2 - \frac{2\,z}{\cos(\varphi)} + \frac{5}{\cos(\varphi)^2} - 4$
```
sage: eq.rhs()
```
0

then solve it for z with `solve`:

```
sage: solve(eq, z)
```
$\left[z = -\frac{2\sqrt{\cos(\varphi)^2-1}-1}{\cos(\varphi)}, z = \frac{2\sqrt{\cos(\varphi)^2-1}+1}{\cos(\varphi)}\right]$

Let us now solve the equation $y^7 = y$.

```
sage: y = var('y'); solve(y^7==y, y)
[y == 1/2*I*sqrt(3) + 1/2, y == 1/2*I*sqrt(3) - 1/2, y == -1,
 y == -1/2*I*sqrt(3) - 1/2, y == -1/2*I*sqrt(3) + 1/2, y == 1, y == 0]
```

The roots of the equation can be returned as an object of type *dictionary* (cf. §3.3.9):

```
sage: solve(x^2-1, x, solution_dict=True)
[{x: -1}, {x: 1}]
```

The `solve` command can also solve systems of equations:

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

This linear system being underdetermined, the variable allowing to parametrise the set of solutions is a real number named `r1`, `r2`, etc. If this parameter is known to be an integer, it is named `z1`, `z2`, etc. (below, `z...` stands for `z36`, `z60`, or similar, according to the Sage version):

```
sage: solve([cos(x)*sin(x) == 1/2, x+y == 0], x, y)
[[x == 1/4*pi + pi*z..., y == -1/4*pi - pi*z...]]
```

The `solve` command can also solve inequalities:

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

Sometimes, `solve` returns the solutions of a system as floating-point numbers. For example, let us solve in $\mathbb{C}^3$ the following system:

$$\begin{cases} x^2yz &= 18, \\ xy^3z &= 24, \\ xyz^4 &= 6. \end{cases}$$

```
sage: x, y, z = var('x, y, z')
sage: solve([x^2 * y * z == 18, x * y^3 * z == 24,\
....:        x * y * z^4 == 6], x, y, z)
[[x == 3, y == 2, z == 1],
 [x == (1.337215067329613 - 2.685489874065195*I),
  y == (-1.700434271459228 + 1.052864325754712*I),
  z == (0.9324722294043555 - 0.3612416661871523*I)], ...]
```

Sage returns here 17 tuples, among which 16 are approximate complex solutions. To obtain a fully symbolic solution, we refer to Chapter 9.

To solve equations numerically, the `find_root` function takes as input a function of one variable or a symbolic equality, and the bounds of the interval in which to search. Sage does not find any symbolic solution to this equation:

```
sage: expr = sin(x) + sin(2 * x) + sin(3 * x)
sage: solve(expr, x)
[sin(3*x) == -sin(2*x) - sin(x)]
```

Two choices are then possible: either a numerical solution,

```
sage: find_root(expr, 0.1, pi)
2.0943951023931957
```

or first rewrite the expression:

```
sage: f = expr.simplify_trig(); f
2*(2*cos(x)^2 + cos(x))*sin(x)
sage: solve(f, x)
[x == 0, x == 2/3*pi, x == 1/2*pi]
```

Last but not least, the `roots` function gives the roots of an equation with their multiplicity. The ring in which solutions are looked for can be given; with `RR` $\approx \mathbb{R}$ or `CC` $\approx \mathbb{C}$, we obtain floating-point roots. The solving method is specific to the given equation, contrary to `find_roots` which uses a generic method.

Let us consider the degree-3 equation $x^3 + 2\,x + 1 = 0$. This equation has a negative discriminant, thus it has a real root and two complex roots, which are given by the `roots` method:

```
sage: (x^3+2*x+1).roots(x)
```

$$\left[\left(-\frac{1}{2}\left(I\sqrt{3}+1\right)\left(\frac{1}{18}\sqrt{3}\sqrt{59}-\frac{1}{2}\right)^{\left(\frac{1}{3}\right)}+\frac{-\left(I\sqrt{3}-1\right)}{3\left(\frac{1}{18}\sqrt{3}\sqrt{59}-\frac{1}{2}\right)^{\left(\frac{1}{3}\right)}},1\right),\right.$$
$$\left(-\frac{1}{2}\left(-I\sqrt{3}+1\right)\left(\frac{1}{18}\sqrt{3}\sqrt{59}-\frac{1}{2}\right)^{\left(\frac{1}{3}\right)}+\frac{-\left(-I\sqrt{3}-1\right)}{3\left(\frac{1}{18}\sqrt{3}\sqrt{59}-\frac{1}{2}\right)^{\left(\frac{1}{3}\right)}},1\right),$$
$$\left.\left(\left(\frac{1}{18}\sqrt{3}\sqrt{59}-\frac{1}{2}\right)^{\left(\frac{1}{3}\right)}+\frac{-2}{3\left(\frac{1}{18}\sqrt{3}\sqrt{59}-\frac{1}{2}\right)^{\left(\frac{1}{3}\right)}},1\right)\right]$$

```
sage: (x^3+2*x+1).roots(x, ring=RR)
```
$[(-0.453397651516404, 1)]$

```
sage: (x^3+2*x+1).roots(x, ring=CC)
```
$[(-0.453397651516404, 1), (0.226698825758202 - 1.46771150871022 * I, 1),$
$(0.226698825758202 + 1.46771150871022 * I, 1)]$

2.2.2 Equations with no Explicit Solution

In most cases, as soon as the equation or system becomes too complex, no explicit solution can be found:

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

However, this is not necessarily a limitation! Indeed, a specificity of computer algebra is the ability to manipulate objects defined by equations, and in particular to compute their properties, without solving them explicitly. Even better: in some cases, the equation defining a mathematical object is the best algorithmic representation for it.

For example, a function given by a linear differential equation and initial conditions is perfectly defined. The set of solutions of linear differential equations is closed under sum and product (among other operations), and thus forms an important class where equality to zero can be decided. However, if we explicitly solve such an equation, the obtained solution might be part of a much larger class where very few questions are decidable.

```
sage: y = function('y')(x)
sage: desolve(diff(y,x,x) + x*diff(y,x) + y == 0, y, [0,0,1])
-1/2*I*sqrt(2)*sqrt(pi)*erf(1/2*I*sqrt(2)*x)*e^(-1/2*x^2)
```

We will go back to this in more detail in Chapter 14 and in §15.1.2.

2.3 Analysis

This section is a quick introduction of useful functions in real analysis. For more advanced usage or more details, we refer to the following chapters, in particular about numerical integration (Chapter 14), non-linear equations (Chapter 12), and differential equations (Chapter 10).

2.3.1 Sums

The `sum` function computes symbolic sums. Let us obtain for example the sum of the n first positive integers:

```
sage: k, n = var('k, n')
sage: sum(k, k, 1, n).factor()
```

$\frac{1}{2}\,(n+1)n$

The `sum` function allows simplifications of a binomial expansion:

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

$(x+y)^n$

Here are more examples, among them the sum of the cardinalities of all parts of a set of n elements:

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

$\left(2^n, 2^{n-1}n, 0\right)$

Finally, some examples of geometric sums:

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

$\frac{aq^{n+1}-a}{q-1}$

To compute the corresponding power series, we should tell Sage that the modulus[2] of q is less than 1:

```
sage: assume(abs(q) < 1)
sage: sum(a*q^k, k, 0, infinity)
```

$-\frac{a}{q-1}$

```
sage: forget(); assume(q > 1); sum(a*q^k, k, 0, infinity)
Traceback (most recent call last):
...
ValueError: Sum is divergent.
```

[2]Remember that by default, symbolic variables represent complex values.

Exercise 2 (Computing a sum by recurrence)**.** Compute, without using the `sum` command, the sum of p-powers of integers from 0 to n, for $p = 1, ..., 4$:

$$S_n(p) = \sum_{k=0}^{n} k^p.$$

The following recurrence can be useful to compute this sum:

$$S_n(p) = \frac{1}{p+1}\left((n+1)^{p+1} - \sum_{j=0}^{p-1}\binom{p+1}{j} S_n(j)\right).$$

This recurrence is easily obtained when computing by two different methods the telescopic sum $\sum_{0\leq k\leq n} (k+1)^{p+1} - k^{p+1}$.

2.3.2 Limits

To determine a limit, we use the `limit` command or its alias `lim`. Let us compute the following limits:

a) $\lim_{x\to 8} \dfrac{\sqrt[3]{x} - 2}{\sqrt[3]{x+19} - 3}$;

b) $\lim_{x\to \frac{\pi}{4}} \dfrac{\cos\left(\frac{\pi}{4} - x\right) - \tan x}{1 - \sin\left(\frac{\pi}{4} + x\right)}$.

```
sage: limit((x**(1/3) - 2) / ((x + 19)**(1/3) - 3), x = 8)
9/4
sage: f(x) = (cos(pi/4-x)-tan(x))/(1-sin(pi/4 + x))
sage: limit(f(x), x = pi/4)
Infinity
```

The last output says that one of the limits to the left or to the right is infinite. To know more about this, we study the limits to the left (`minus`) and to the right (`plus`), with the `dir` option:

```
sage: limit(f(x), x = pi/4, dir='minus')
+Infinity
sage: limit(f(x), x = pi/4, dir='plus')
-Infinity
```

2.3.3 Sequences

The above functions enable us to study sequences of numbers. We illustrate this by comparing the growth of an exponential sequence and a geometric sequence.

Example. (*A sequence study*) Let us consider the sequence $u_n = \frac{n^{100}}{100^n}$. Compute the first 10 terms. How does the sequence vary? What is the sequence limit? From which value of n does $u_n \in \left]0, 10^{-8}\right[$ hold?

1. To define the term of order n, we use a symbolic function. We then compute the first 10 terms by hand (loops will be introduced in Chapter 3):

```
sage: u(n) = n^100 / 100^n
sage: u(1.);u(2.);u(3.);u(4.);u(5.);u(6.);u(7.);u(8.);u(9.);u(10.)
0.0100000000000000
1.26765060022823e26
5.15377520732011e41
1.60693804425899e52
7.88860905221012e59
6.53318623500071e65
3.23447650962476e70
2.03703597633449e74
2.65613988875875e77
1.00000000000000e80
```

   We could quickly conclude that u_n tends to infinity...

2. To get an idea of the variation of the sequence, we can draw the graph of the function $n \to u_n$ (cf. Figure 2.2).

```
sage: plot(u(x), x, 1, 40)
Graphics object consisting of 1 graphics primitive
```

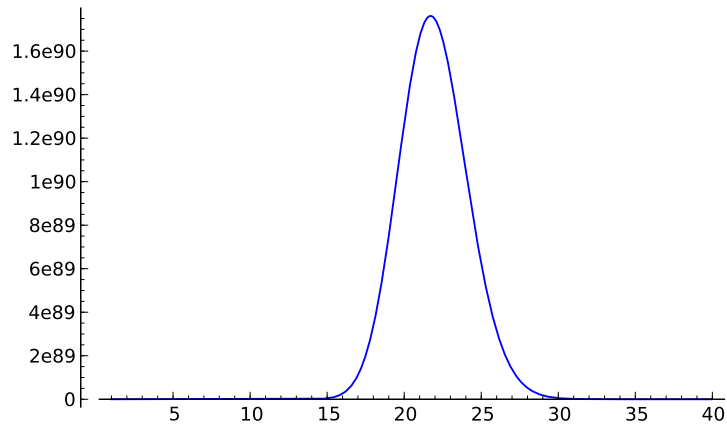


FIGURE 2.2 – Graph of $x \mapsto x^{100}/100^x$.

   We then conjecture that the sequence decreases from index 22 onwards.

```
sage: v(x) = diff(u(x), x); sol = solve(v(x) == 0, x); sol
[x == 50/log(10), x == 0]
sage: floor(sol[0].rhs())
21
```

   The sequence is thus increasing up to index 21, then decreasing after index 22.

Functions and operators	
Derivative	`diff(f(x), x)`
n-th derivative	`diff(f(x), x, n)`
Antiderivative	`integrate(f(x), x)`
Numerical integration	`integral_numerical(f(x), a, b)`
Symbolic summation	`sum(f(i), i, imin, imax)`
Limit	`limit(f(x), x=a)`
Taylor expansion	`taylor(f(x), x, a, n)`
Power series expansion	`f.series(x==a, n)`
Graph of a function	`plot(f(x), x, a, b)`

TABLE 2.4 – Useful functions in analysis.

3. We then compute the limit:

```
sage: limit(u(n), n=infinity)
0
sage: n0 = find_root(u(n) - 1e-8 == 0, 22, 1000); n0
105.07496210187252
```

Since the sequence decreases from index 22 onwards, we deduce that starting from index 106, the sequence always lies in the interval $\left]0, 10^{-8}\right[$.

2.3.4 Power Series Expansions (*)

To compute a power series expansion of order n at x_0, the command to use is `f(x).series(x==x0, n)`.

Let us determine the power series expansion of the following functions:

$$a)\quad (1+\arctan x)^{\frac{1}{x}} \quad \text{of order 3, at } x_0 = 0;$$

$$b)\quad \ln(2\sin x) \quad \text{of order 3, at } x_0 = \tfrac{\pi}{6}.$$

```
sage: ((1+arctan(x))^(1/x)).series(x==0, 3)
```
$(e) + (-\frac{1}{2}\,e)x + (\frac{1}{8}\,e)x^2 + \mathcal{O}\left(x^3\right)$

```
sage: (ln(2*sin(x))).series(x==pi/6, 3)
```
$(\sqrt{3})(-\frac{1}{6}\,\pi + x) + (-2)(-\frac{1}{6}\,\pi + x)^2 + \mathcal{O}\left(-\frac{1}{216}\,(\pi - 6\,x)^3\right)$

To extract the regular part of a power series expansion obtained by `series`, we call the `truncate` method:

```
sage: (ln(2*sin(x))).series(x==pi/6, 3).truncate()
```
$-\frac{1}{18}\,(\pi - 6\,x)^2 - \frac{1}{6}\,\sqrt{3}(\pi - 6\,x)$

The `taylor` command provides asymptotic expansions too. For example, let us see how the function $(x^3+x)^{\frac{1}{3}} - (x^3-x)^{\frac{1}{3}}$ behaves around $+\infty$:

```
sage: taylor((x**3+x)**(1/3) - (x**3-x)**(1/3), x, infinity, 2)
2/3/x
```

Exercise 3 (Computing a symbolic limit)**.** Let f be $\mathcal{C}^3$ around $a \in \mathbb{R}$. Compute

$$\lim_{h \to 0} \frac{1}{h^3} \left(f(a+3h) - 3f(a+2h) + 3f(a+h) - f(a) \right).$$

Generalisation?

EXAMPLE. (*Machin's formula*) Prove the following formula:

$$\frac{\pi}{4} = 4 \arctan \frac{1}{5} - \arctan \frac{1}{239}.$$

The astronomer John Machin (1680-1752) used this formula and the series expansion of arctan to compute 100 decimal digits of π in 1706.

We first notice that $4 \arctan \frac{1}{5}$ and $\frac{\pi}{4} + \arctan \frac{1}{239}$ admit the same tangent:

```
sage: tan(4*arctan(1/5)).simplify_trig()
120/119
sage: tan(pi/4+arctan(1/239)).simplify_trig()
120/119
```

Since the real numbers $4 \arctan \frac{1}{5}$ and $\frac{\pi}{4} + \arctan \frac{1}{239}$ are both in the open interval $]0, \pi[$, they are equal. To obtain an approximation of π, we thus proceed as follows:

```
sage: f = arctan(x).series(x, 10); 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(); pi.n()
3.14159268240440
3.14159265358979
```

Exercise 4 (A formula due to Gauss)**.** The following formula required 20 pages of factorisation tables in the edition of Gauss' works (cf. *Werke*, ed. Königl. Ges. d. Wiss. Göttingen, vol. 2, p. 477-502):

$$\frac{\pi}{4} = 12 \arctan \frac{1}{38} + 20 \arctan \frac{1}{57} + 7 \arctan \frac{1}{239} + 24 \arctan \frac{1}{268}.$$

1. Define $\theta = 12 \arctan \frac{1}{38} + 20 \arctan \frac{1}{57} + 7 \arctan \frac{1}{239} + 24 \arctan \frac{1}{268}$. Verify with Sage that $\tan \theta = 1$.
2. Justify the inequality: $\forall x \geq 0, \arctan x \leq x$. Deduce Gauss' formula.
3. Approximate the arctan function by its Taylor expansion of order 21 at 0, and deduce a new approximation of π.

2.3.5 Series (*)

The commands introduced earlier can be used to perform computations on series. Let us give some examples.

EXAMPLE. (*Evaluation of the Riemann zeta function*)

```
sage: k = var('k')
sage: sum(1/k^2, k, 1, infinity),\
....: sum(1/k^4, k, 1, infinity),\
....: sum(1/k^5, k, 1, infinity)
```

$\left(\frac{1}{6}\,\pi^2, \frac{1}{90}\,\pi^4, \zeta(5)\right)$

EXAMPLE. (*A formula due to Ramanujan*) Using the first 12 terms of the following series, we give an approximation of π and we compare it with the value given by Sage.

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{+\infty} \frac{(4k)! \cdot (1103 + 26390\,k)}{(k!)^4 \cdot 396^{4k}}.$$

```
sage: s = 2*sqrt(2)/9801*(sum((factorial(4*k)) * (1103+26390*k) /
....:     ((factorial(k)) ^ 4 * 396 ^ (4 * k)) for k in (0..11)))
sage: (1/s).n(digits=100)
3.1415926535897932384626433832795028841971693993751058209 74...
sage: (pi-1/s).n(digits=100).n()
-4.36415445739398e-96
```

We notice that the partial sum of the first 12 terms already yields 95 significant digits of π!

EXAMPLE. (*Convergence of a series*) Let us study the convergence of the series

$$\sum_{n\geq 0} \sin\left(\pi\sqrt{4\,n^2+1}\right).$$

To get an asymptotic expansion of the general term, we use the 2π-periodicity of the sine function, so that the sine argument tends to 0:

$$u_n = \sin\left(\pi\sqrt{4\,n^2+1}\right) = \sin\left[\pi\left(\sqrt{4\,n^2+1} - 2n\right)\right].$$

We can then apply the `taylor` function to this new expression of the general term:

```
sage: n = var('n'); u = sin(pi*(sqrt(4*n^2+1)-2*n))
sage: taylor(u, n, infinity, 3)
```

$\frac{\pi}{4\,n} - \frac{6\,\pi+\pi^3}{384\,n^3}$

We deduce $u_n \sim \frac{\pi}{4\,n}$. Therefore, by comparison with the series defining the Riemann zeta function, the series $\sum_{n\geq 0} u_n$ diverges.

Exercise 5 (Asymptotic expansion of a sequence)**.** It is easy to show (for example, using a bijection) that for all $n \in \mathbb{N}$, the equation $\tan x = x$ has exactly one solution x_n in the interval $[n\pi, n\pi + \frac{\pi}{2}[$. Give an asymptotic expansion of x_n to order 6 in $+\infty$.

2.3.6 Derivatives

The `derivative` function (with alias `diff`) computes the derivative of a symbolic expression or function.

```
sage: diff(sin(x^2), x)
2*x*cos(x^2)
sage: function('f')(x); function('g')(x); diff(f(g(x)), x)
f(x)
g(x)
D[0](f)(g(x))*diff(g(x), x)
sage: diff(ln(f(x)), x)
diff(f(x), x)/f(x)
```

2.3.7 Partial Derivatives (*)

The `derivative` (or `diff`) command also computes iterated or partial derivatives.

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

EXAMPLE. Let us check that the following function is harmonic[3]:
$$f(x,y) = \tfrac{1}{2}\ln(x^2+y^2) \text{ for all } (x,y) \neq (0,0).$$

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

Exercise 6 (A counter-example due to Peano to Schwarz' theorem)**.** Let f be the function from $\mathbb{R}^2$ to $\mathbb{R}$ defined by:

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

Does $\partial_1\partial_2 f(0,0) = \partial_2\partial_1 f(0,0)$ hold?

2.3.8 Integrals

To compute an indefinite or definite integral, we use `integrate` as a function or method (or its alias `integral`):

```
sage: sin(x).integral(x, 0, pi/2)
1
sage: integrate(1/(1+x^2), x)
arctan(x)
```

[3] A function f is said *harmonic* when its Laplacian $\Delta f = \partial_1^2 f + \partial_2^2 f$ is zero.

```
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 call last):
...
ValueError: Integral is divergent.
```

EXAMPLE. Let us compute, for $x \in \mathbb{R}$, the integral $\varphi(x) = \displaystyle\int_0^{+\infty} \frac{x \cos u}{u^2 + x^2}\, \mathrm{d}u$.

```
sage: u = var('u'); f = x * cos(u) / (u^2 + x^2)
sage: assume(x>0); f.integrate(u, 0, infinity)
1/2*pi*e^(-x)
sage: forget(); assume(x<0); f.integrate(u, 0, infinity)
-1/2*pi*e^x
```

We thus have: $\forall x \in \mathbb{R}^*, \quad \varphi(x) = \frac{\pi}{2} \cdot \mathrm{sgn}(x) \cdot e^{-|x|}$.

To compute numerically an integral on an interval, we have at our disposal the `integral_numerical` function, which returns a pair, whose first value is the approximation of the integral, while the second value is an estimate of the corresponding error.

```
sage: integral_numerical(sin(x)/x, 0, 1)
(0.946083070367183, 1.0503632079297087e-14)
sage: g = integrate(exp(-x**2), x, 0, infinity)
sage: g, g.n()
(1/2*sqrt(pi), 0.886226925452758)
sage: approx = integral_numerical(exp(-x**2), 0, infinity)
sage: approx
(0.8862269254527568, 1.714774436012769e-08)
sage: approx[0]-g.n()
-1.11022302462516e-15
```

Exercise 7 (The BBP formula (*))**.** Let us establish by a symbolic computation the BBP formula (or Bailey-Borwein-Plouffe formula); this formula directly gives the n-th digit of π in radix 2 (or 16) without computing the previous digits, and with very little memory usage and time. For $N \in \mathbb{N}$, let us define

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

1. Consider the function $f : t \longmapsto 4\sqrt{2} - 8t^3 - 4\sqrt{2}t^4 - 8t^5$. For $N \in \mathbb{N}$, express the following integral as a function of S_N:

$$I_N = \int_0^{1/\sqrt{2}} f(t) \left(\sum_{n=0}^{N} t^{8n} \right) \mathrm{d}t.$$

Usual functions on vectors	
Vector construction	`vector`
Cross product	`cross_product`
Scalar product	`dot_product`
Norm of a vector	`norm`

TABLE 2.5 – Vector computations.

2. For $N \in \mathbb{N}$, let us define $J = \displaystyle\int_0^{1/\sqrt{2}} \frac{f(t)}{1-t^8}\,\mathrm{d}t$. Prove $\lim\limits_{N\to+\infty} S_N = J$.
3. Prove the BBP formula:

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

This fabulous formula was found on September 19, 1995 by Simon Plouffe, in collaboration with David Bailey and Peter Borwein. Thanks to computation derived from the BBP formula, the 4 000 000 000 000 000-th digit of π in radix 2 was computed in 2001.

2.4 Basic Linear Algebra (*)

In this section, we describe the basic useful functions in linear algebra: first operations on vectors, then on matrices. For more details, we refer the reader to Chapter 8 for symbolic linear algebra, and to Chapter 13 for numerical linear algebra.

2.4.1 Solving Linear Systems

To solve a linear system, we can use the `solve` function, already seen above.

Exercise 8 (Polynomial approximation of the sine function)**.** Determine the polynomial of degree at most 5 which approximates best, in the least squares sense, the sine function on the interval $[-\pi, \pi]$:

$$\alpha_5 = \min\left\{\int_{-\pi}^{\pi} |\sin x - P(x)|^2\,\mathrm{d}x \;\middle|\; P \in \mathbb{R}_5[x]\right\}.$$

2.4.2 Vector Computations

The basic functions for manipulating vectors are summarised in Table 2.5.

We can use those functions to deal with the following exercise.

Exercise 9 (Gauss' problem)**.** Consider a satellite in orbit around the Earth, and assume we know three points of its orbit: A_1, A_2 and A_3. Determine from these three points the orbit parameters of this satellite.

Let us denote O the centre of the Earth. The points O, A_1, A_2 and A_3 are clearly in the same plane, namely the plane defined by the satellite orbit. The satellite orbit is

an ellipse of which O is a focal point. We can choose as coordinate system $(O; \overrightarrow{\imath}, \overrightarrow{\jmath})$ in such a way that the ellipse equation in polar coordinates is $r = \frac{p}{1-e\cos\theta}$ where e is the ellipse eccentricity, and p its parameter. We will note $\overrightarrow{r_i} = \overrightarrow{OA_i}$ and $r_i = \|\overrightarrow{r_i}\|$ for $i \in \{1, 2, 3\}$. We then consider the three following vectors deduced from A_1, A_2 and A_3:

$$\begin{aligned}
\overrightarrow{D} &= \overrightarrow{r_1} \wedge \overrightarrow{r_2} + \overrightarrow{r_2} \wedge \overrightarrow{r_3} + \overrightarrow{r_3} \wedge \overrightarrow{r_1}, \\
\overrightarrow{S} &= (r_1 - r_3) \cdot \overrightarrow{r_2} + (r_3 - r_2) \cdot \overrightarrow{r_1} + (r_2 - r_1) \cdot \overrightarrow{r_3}, \\
\overrightarrow{N} &= r_3 \cdot (\overrightarrow{r_1} \wedge \overrightarrow{r_2}) + r_1 \cdot (\overrightarrow{r_2} \wedge \overrightarrow{r_3}) + r_2 \cdot (\overrightarrow{r_3} \wedge \overrightarrow{r_1}).
\end{aligned}$$

1. Show that $\overrightarrow{\imath} \wedge \overrightarrow{D} = -\frac{1}{e}\overrightarrow{S}$ and deduce the ellipse eccentricity.
2. Show that $\overrightarrow{\imath}$ is colinear with the vector $\overrightarrow{S} \wedge \overrightarrow{D}$.
3. Show that $\overrightarrow{\imath} \wedge \overrightarrow{N} = -\frac{p}{e}\overrightarrow{S}$ and deduce the ellipse parameter p.
4. Compute the half major axis a of the ellipse in term of the parameter p and the eccentricity e.
5. *Numerical application:* in the plane with a Cartesian coordinate system, we consider the following points:

$$A_1 \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \quad A_2 \begin{pmatrix} 2 \\ 2 \end{pmatrix}, \quad A_3 \begin{pmatrix} 3{,}5 \\ 0 \end{pmatrix}, \quad O \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

   Determine numerically the characteristics of the unique ellipse having O as focal point and passing through the three points A_1, A_2 and A_3.

2.4.3 Matrix Computations

To construct a matrix, what you want is the `matrix` function, which allows to optionally specify the base ring (or field):

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

To find a particular solution of the matrix equation $Ax = b$ (resp. $xA = b$), we call the `solve_right` function (resp. `solve_left`). To find *all* the solutions, we should add to that particular solution the general solution of the associated homogeneous equation. To solve a homogeneous equation $Ax = 0$ (resp. $xA = 0$), we use the `right_kernel` (resp. `left_kernel`) function, as in the following exercise.

Exercise 10 (Basis of vector subspace)**.**

1. Determine a basis of the space of solutions of the linear homogeneous system corresponding to the matrix:

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

2. Determine a basis of the space F generated by the columns of A.
3. Characterise F by one or several equations.

Usual functions on matrices	
Construction of a matrix	`matrix`
Solving a matrix equation	`solve_right, solve_left`
Right and left kernel	`right_kernel, left_kernel`
Row echelon form	`echelon_form`
Column-generated vector space	`column_space`
Row-generated vector space	`row_space`
Matrix concatenation	`block_matrix`

Matrix reduction	
Eigenvalues of a matrix	`eigenvalues`
Eigenvectors of a matrix	`eigenvectors_right`
Jordan normal form reduction	`jordan_form`
Minimal polynomial	`minimal_polynomial`
Characteristic polynomial	`characteristic_polynomial`

TABLE 2.6 – Matrix computations.

Exercise 11 (A matrix equation)**.** Let us recall the factorisation lemma for linear maps. Let E, F, G be $\mathbb{K}$-vector spaces of finite dimension. Let $u \in \mathcal{L}(E, F)$ and $v \in \mathcal{L}(E, G)$. Then the following assertions are equivalent:

i) there exists $w \in \mathcal{L}(F, G)$ such that $v = w \circ u$,

ii) $\mathrm{Ker}(u) \subset \mathrm{Ker}(v)$.

We search all solutions to this problem in a concrete case. Let

$$A = \begin{pmatrix} -2 & 1 & 1 \\ 8 & 1 & -5 \\ 4 & 3 & -3 \end{pmatrix} \quad \text{and} \quad C = \begin{pmatrix} 1 & 2 & -1 \\ 2 & -1 & -1 \\ -5 & 0 & 3 \end{pmatrix}.$$

Determine all solutions $B \in \mathcal{M}_3(\mathbb{R})$ of the equation $A = BC$.

2.4.4 Reduction of a Square Matrix

To study the eigenvalues and eigenvectors of a matrix, all functions from Table 2.6 are available. Those functions will be detailed in Chapter 8. We only give here a few simple examples of their usage.

EXAMPLE. Is the matrix $A = \begin{pmatrix} 2 & 4 & 3 \\ -4 & -6 & -3 \\ 3 & 3 & 1 \end{pmatrix}$ diagonalisable? Triangularisable?

We start by defining the matrix A by giving the base field (`QQ`=$\mathbb{Q}$), then we determine its eigenvalues and eigenvectors.

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

The minimal polynomial of A admits a simple root and a double root; thus A is not diagonalisable. However, its minimal polynomial being split into linear factors, A is triangularisable.

```
sage: A.eigenvectors_right()
```
$[(1, [(1, -1, 1)], 1), (-2, [(1, -1, 0)], 2)]$

```
sage: A.jordan_form(transformation=True)
```
$$\left(\left(\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & -2 & 1 \\ 0 & 0 & -2 \end{array}\right), \begin{pmatrix} 1 & 1 & 1 \\ -1 & -1 & 0 \\ 1 & 0 & -1 \end{pmatrix}\right)$$

EXAMPLE. Let us diagonalise the matrix $A = \begin{pmatrix} 1 & -1/2 \\ -1/2 & -1 \end{pmatrix}$. We first try the `jordan_form` function:

```
sage: A = matrix(QQ, [[1,-1/2],[-1/2,-1]])
sage: A.jordan_form()
Traceback (most recent call last):
...
RuntimeError: Some eigenvalue does not exist in Rational Field.
```

A small difficulty appears here: the eigenvalues are not all rational.

```
sage: A = matrix(QQ, [[1,-1/2],[-1/2,-1]])
sage: A.minimal_polynomial()
x^2 - 5/4
```

We therefore have to change the base field.

```
sage: R = QQ[sqrt(5)]
sage: A = A.change_ring(R)
sage: A.jordan_form(transformation=True, subdivide=False)
```
$$\left(\begin{pmatrix} \frac{1}{2}sqrt_5 & 0 \\ 0 & -\frac{1}{2}sqrt_5 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ -sqrt_5+2 & sqrt_5+2 \end{pmatrix}\right)$$

This is to be interpreted as:

$$\left(\begin{pmatrix} \frac{1}{2}\sqrt{5} & 0 \\ 0 & -\frac{1}{2}\sqrt{5} \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ -\sqrt{5}+2 & \sqrt{5}+2 \end{pmatrix}\right)$$

EXAMPLE. Let us diagonalise the matrix $A = \begin{pmatrix} 2 & \sqrt{6} & \sqrt{2} \\ \sqrt{6} & 3 & \sqrt{3} \\ \sqrt{2} & \sqrt{3} & 1 \end{pmatrix}$.

Here, we have to work in an extension of degree 4 of the field $\mathbb{Q}$, for example as follows.

```
sage: K.<sqrt2> = NumberField(x^2 - 2)
sage: L.<sqrt3> = K.extension(x^2 - 3)
sage: A = matrix(L, [[2, sqrt2*sqrt3, sqrt2],  \
....:                [sqrt2*sqrt3, 3, sqrt3],  \
....:                [sqrt2, sqrt3, 1]])
```

```
sage: A.jordan_form(transformation=True)
```

$$\left(\begin{pmatrix} 6 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 & 0 \\ \frac{1}{2}\sqrt{2}\sqrt{3} & 0 & 1 \\ \frac{1}{2}\sqrt{2} & -\sqrt{2} & -\sqrt{3} \end{pmatrix}\right)$$

3 Programming and Data Structures

The two preceding chapters introduced mathematical computations using one-line commands, but Sage also allows programs with sequences of instructions.

The Sage computer algebra system is in fact an extension of the Python[1] computer language, and allows, with a few exceptions, to use the Python programming constructs.

The commands described in the previous chapters show that it is not necessary to know the Python language to use Sage; this chapter explains how to use the Python programming structures within Sage. Since we only present basic programming, this chapter can be skipped by the reader fluent in Python; the examples are chosen among the most classical ones encountered in mathematics, so that the reader can quickly grasp the Python programming constructs, by analogy with known programming languages.

This chapter presents in particular the paradigm of *structured programming* with loops and tests, then describes functions dealing with lists and other data structures.

3.1 Syntax

3.1.1 General Syntax

The instructions are generally processed line by line. Python considers the sharp symbol "`#`" as the beginning of a comment, until the end of the line.

The semi-colon "`;`" separates several instructions written on the same line:

[1] The Sage version considered here uses Python 2.7, which slightly differs from Python 3.

Python language keywords	
`while`, `for...in`, `if...elif...else`	loops and tests
`continue`, `break`	early exit from a code block
`try...except...finally`, `raise`	deal with and raise exceptions
`assert`	debugging condition
`pass`	no-effect statement
`def`, `lambda`	definition of a function
`return`, `yield`	return of a value
`global`, `del`	scope and deleting variables and functions
`and`, `not`, `or`	boolean operations
`print`	text output
`class`, `with`	object-oriented and context programming
`from...import...as`	library access
`exec...in`	dynamic code evaluation

TABLE 3.1 – General syntax of the Sagecode.

```
sage: 2*3; 3*4; 4*5          # one comment, 3 results
6
12
20
```

In the terminal, a command can be written on several lines by putting a backslash "\" before each end of line, this character being ignored:

```
sage: 123 + \
....: 345
468
```

An identifier — i.e., a variable or function name, etc. — is formed from letters, digits and the underline symbol "_", and cannot start with a digit. The user identifiers should differ from the language keywords, given in Table 3.1, and which form the core of the Python language. The list of keywords is available by:

```
sage: import keyword; keyword.kwlist
['and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del',
'elif', 'else', 'except', 'exec', 'finally', 'for', 'from',
'global', 'if', 'import', 'in', 'is', 'lambda', 'not', 'or', 'pass',
'print', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

In addition to these keywords, we have the constants `None` (empty value, named `NULL` in other languages), `True` and `False`, and several functions predefined by Python or Sage like `len`, `cos` and `integrate`. It is better not to use these as variable names, otherwise some functionalities might no longer be available. The interpreter knows some additional commands, like `quit` to exit the Sage session. We will discover other commands like `time` or `timeit` later in this book.

Some symbols have a special meaning in Sage. They are explained in Table 3.2.

Sage special symbols and their main uses	
, ;	argument and instruction separators
:	beginning of an instruction block
.	decimal point, accessing an object field
=	assignment of a value to a variable
+ - * /	basic arithmetic operations
^ **	power
// %	quotient and remainder of Euclidean division
+= -= *= /= **=	arithmetic operations with modification of a variable
== != <> is	equality tests
< <= > >=	comparisons
& | ^^ << >>	set operations and bitwise logical operations
#	comment (until end of line)
[...]	construction of a list, accessing an element by its index
(...)	function or method call, immutable tuples
{...:...}	dictionary construction
\	special character escape (and linear algebra)
@	applying a decorator to a function
? ??	help and source code access
_ __ ___	last three results

TABLE 3.2 – General Sage syntax (following).

3.1.2 Function Calls

To evaluate a function, its arguments should be put inside parentheses — for example `cos(pi)` — or in the function call without argument `reset()`. However, the parentheses are superfluous for a command: the instructions `print 6*7` and `print(6*7)` are equivalent[2]. The name of a function without argument nor parenthesis represents the function itself and performs no computation.

3.1.3 More About Variables

As seen previously, Sage denotes the assignment of a value to a variable by the equal sign "`=`". The expression to the right of the equal sign is first evaluated, then its value is saved in the variable whose name is on the left. Thus we have:

```
sage: y = 3; y = 3 * y + 1; y = 3 * y + 1; y
31
```

The three first assignments change the value of the variable `y` without any output, the last command prints the final value of `y`.

The `del x` command discards the value assigned to the variable `x`, and the function call `reset()` recovers the initial Sage state.

Several variables can be assigned *simultaneously*, which differs from *sequential* assignments `a = b; b = a`:

```
sage: a, b = 10, 20 # (a, b) = (10, 20) and [10, 20] are also possible
```

[2]In Python 3, `print` is a function and thus requires parentheses. This behaviour can be obtained with `from __future__ import print_function`.

```
sage: a, b = b, a
sage: a, b
(20, 10)
```

The assignment `a, b = b, a` is equivalent to swapping the values of `a` and `b` using an auxiliary variable:

```
sage: temp = a; a = b; b = temp    # equivalent to: a, b = b, a
```

The following trick swaps the values of `a` and `b` without any auxiliary variable, using additions and subtractions:

```
sage: x, y = var('x, y'); a = x ; b = y
sage: a, b
(x, y)
sage: a = a + b ; b = a - b ; a = a - b
sage: a, b
(y, x)
```

The instruction `a = b = c = 0` assigns the same value, here 0, to several variables; the instructions `x += 5` and `n *= 2` are respectively equivalent to `x = x+5` and `n = n*2`.

The comparison between two objects is performed by the double equal sign "`==`":

```
sage: 2 + 2 == 2^2, 3 * 3 == 3^3
(True, False)
```

3.2 Algorithmics

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:

- an example of atomic instruction is the assignment of a value to a variable (cf. §1.2.4), or a result output;
- a composed instruction, like a loop or a conditional, is made up from several instructions, themselves atomic or composed.

3.2.1 Loops

Enumeration Loops. An enumeration loop performs the same computation for all integer values of an index $k \in \{a, \ldots, b\}$: the following example[3] outputs the beginning of the multiplication table by 7:

[3] When using Sage in a terminal, such a block of instructions must be ended by an additional empty line, which will be implicit in the whole book. This is not necessary when using Sage through a web browser.

```
sage: for k in [1..5]:
....:     print(7*k)  # block containing a single instruction
7
14
21
28
35
```

The colon symbol ":" at the end of the first line starts the instruction block, which is evaluated for each successive value 1, 2, 3, 4 and 5 of the variable `k`. At each iteration, Sage outputs the product $7k$ via the `print` command.

In this example, the repeated instruction block contains a single instruction (namely `print`), which is indented to the right with respect to the `for` keyword. A block with several instructions has its instructions written one below the other, with the same indentation.

The block positioning is important: the two programs below, which differ in the indentation of a single line, yield different results.

```
sage:  S = 0                      sage:  S = 0
sage:  for k in [1..3]:           sage:  for k in [1..3]:
...      S = S+k                  ...      S = S+k
sage:  S = 2*S                    ...      S = 2*S
sage:  S                          sage:  S
```

On the left the instruction `S = 2*S` is executed only once at the end of the loop, while on the right it is executed at every iteration, which explains the different results:

$$S = (0+1+2+3)\cdot 2 = 12 \qquad S = ((((0+1)\cdot 2)+2)\cdot 2+3)\cdot 2 = 22.$$

This kind of loop will be useful to compute a given term of a recurrence, cf. the examples at the end of this section.

The syntax `for k in [a..b]` for an enumeration loop is the simplest one and can be used without any problem for 10^4 or 10^5 iterations; its drawback is that it explicitly constructs the list of all possible values of the loop variable before executing the iteration block, however it manipulates Sage integers of type `Integer` (see §5.3.1). Several `..range` functions allow iterations with two possible choices. The first choice is: either construct the list of possible values before starting the loop, or determine those values along with the loop iterations. The second choice is between Sage integers[4] (`Integer`) and Python integers (`int`), those two integer types having slightly different properties. In case of doubt, the `[a..b]` form should be preferred.

While Loops. The other kind of loops are the *while* loops. Like the enumeration loops, they execute a certain number of times the same sequence of instructions;

[4] The commands `srange`, `sxrange` and `[...]` also work on rational and floating-point numbers: try `[pi,pi+5..20]` for example.

Iterations functions of the `..range` form for `a`, `b`, `c` integers	
`for k in [a..b]: ...`	constructs the list of Sage integers $a \leq k \leq b$
`for k in srange (a, b): ...`	constructs the list of Sage integers $a \leq k < b$
`for k in range (a, b): ...`	constructs a list of Python integers (`int`)
`for k in xrange (a, b): ...`	enumerates Python integers (`int`) without explicitly constructing the corresponding list
`for k in sxrange (a, b): ...`	enumerates Sage integers without constructing a list
`[a,a+c..b], [a..b, step=c]`	Sage integers $a, a+c, a+2c, \ldots$ as long as $a+kc \leq b$
`..range (b)`	equivalent to `..range (0, b)`
`..range (a, b, c)`	sets the iteration increment to c instead of 1

TABLE 3.3 – The different enumeration loops.

however, here the number of repetitions is not known *a priori*, but depends on a condition.

The *while* loop, as its name says, executes instructions while a given condition is fulfilled. The following example computes the sum of the squares of non-negative integers whose exponential is less or equal to 10^6, i.e., $1^2 + 2^2 + \cdots + 13^2$:

```
sage: S = 0 ; k = 0          #        The sum S starts to 0
sage: while e^k <= 10^6:     #            e^13 <= 10^6 < e^14
....:     S = S + k^2        #          accumulates the squares k^2
....:     k = k + 1
sage: S
819
```

The last instruction returns the value of the variable `S` and outputs the result:

$$S = \sum_{\substack{k\in\mathbb{N} \\ e^k \leq 10^6}} k^2 = \sum_{k=0}^{13} k^2 = 819, \qquad e^{13} \approx 442413 \leq 10^6 < e^{14} \approx 1202604.$$

The above instruction block contains two assignments: the first one accumulates the new term, and the second one moves to the next index. Those two instructions are indented in the same way inside the `while` loop structure.

The following example is another typical example of *while* loop. For a given number $x \geq 1$, it seeks the unique integer $n \in \mathbb{N}$ satisfying $2^{n-1} \leq x < 2^n$, i.e., the smallest integer with $x < 2^n$. The program below compares x to 2^n, whose value is successively 1, 2, 4, 8, etc.; it performs this computation for $x = 10^4$:

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

As long as the condition $2^n \leq x$ is satisfied, this program computes the new values $n+1$ and $2^{n+1} = 2 \cdot 2^n$ of the two variables `n` and `u`, and stores them in place of n and 2^n. The loop ends when the condition is no longer fulfilled, i.e., when

$x < 2^n$:

$$x = 10^4, \qquad \min\{n \in \mathbb{N} \mid x < 2^n\} = 14, \qquad 2^{13} = 8192, \qquad 2^{14} = 16384.$$

Note that the body of a *while* loop is never executed when the condition is false at the first test.

As seen above, small command blocks can be typed on a single line after the colon ":", without creating a new indented block starting at the next line.

Aborting a loop execution

The `for` and `while` loops repeat a given number of times the same instructions. The `break` command inside a loop interrupts it before its end, and the `continue` command goes directly to the next iteration. Those commands thus allow — among other things — to check the terminating condition at every place in the loop.

The four examples below determine the smallest positive integer x satisfying $\log(x+1) \leq x/10$. The first program (top left) uses a `for` loop with at most 100 tries which terminates once the first solution is found; the second program (top right) looks for the smallest solution and might not terminate if the condition is never fulfilled; the third (bottom left) is equivalent to the first one with a more complex loop condition; finally the fourth (bottom right) has an unnecessarily complex structure, whose unique goal is to exhibit the `continue` command. In all cases the final value `x` is 37.0.

```
for x in [1.0..100.0]:
  if log(x+1)<=x/10:  break
```

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

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

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

The `return` command (which ends the execution of a function and defines its result, cf. §3.2.3) offers yet another way to abort early from an instruction block.

Application to Sequences and Series. The `for` loop enables us to easily compute a given term of a recurrent sequence. Consider for example the sequence (u_n) defined by

$$u_0 = 1, \qquad \forall n \in \mathbb{N}, \quad u_{n+1} = \frac{1}{1+u_n^2}.$$

The following program yields a numerical approximation of u_n for $n = 20$; the variable `U` is updated at each loop iteration to change from u_{n-1} to u_n according

to the recurrence relation. The first iteration computes u_1 from u_0 for $n = 1$, the second one likewise from u_1 to u_2 when $n = 2$, and the last of the n iterations updates U from u_{n-1} to u_n:

```
sage: U = 1.0                # or U = 1. or U = 1.000
sage: for n in [1..20]:
....:   U = 1 / (1 + U^2)
sage: U
0.682360434761105
```

The same program with the integer `U = 1` instead of the floating-point number `U = 1.0` on the first line will perform exact computations on rational numbers; then u_{10} becomes a rational number with several hundreds digits, and u_{20} has hundreds of thousands digits. Exact computations are useful when rounding errors accumulate in numerical approximations. Otherwise, *by hand* or *with the computer*, the computations on numerical approximations of a dozen digits are faster than those on integers or rational numbers of thousand digits or more.

The sums or products admitting recurrence formulas are computed the same way:

$$S_n = \sum_{k=1}^{n} (2k)(2k+1) = 2 \cdot 3 + 4 \cdot 5 + \cdots + (2n)(2n+1),$$
$$S_0 = 0, \qquad S_n = S_{n-1} + (2n)(2n+1) \quad \text{for } n \in \mathbb{N} - \{0\}.$$

The following programming method follows that of recurrent sequences; starting from 0, we add successive terms for $k = 1$, $k = 2$, ..., until $k = n$:

```
sage: S = 0 ; n = 10
sage: for k in [1..n]:
....:     S = S + (2*k) * (2*k+1)
sage: S
1650
```

This example highlights a general method to compute a sum; however, in this simple case, a symbolic computation yields the general answer:

```
sage: n, k = var('n, k') ; res = sum(2*k*(2*k+1), k, 1, n)
sage: res, factor(res)    # result expanded, factorised
(4/3*n^3 + 3*n^2 + 5/3*n, 1/3*(4*n + 5)*(n + 1)*n)
```

Those results might also be obtained with the *pen and pencil* method from well-known sums:

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}, \qquad \sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6},$$
$$\sum_{k=1}^{n} 2k\,(2k+1) = 4\sum_{k=1}^{n} k^2 + 2\sum_{k=1}^{n} k = \frac{2\,n\,(n+1)(2n+1)}{3} + n(n+1)$$
$$= \frac{n(n+1)\big((4n+2)+3\big)}{3} = \frac{n(n+1)(4n+5)}{3}.$$

Example: Approximation of Sequence Limits. While the enumeration loop is well suited to compute a given term of a sequence or series, the *while* loop is adapted to approximate numerically the limit of a sequence.

If a sequence $(a_n)_{n\in\mathbb{N}}$ converges to $\ell \in \mathbb{R}$, the terms a_n are *close* to ℓ for n *large enough*. It is thus possible to approximate ℓ by a given term a_n, and the mathematical problem reduces to finding a bound for the error $|\ell - a_n|$. This bound is trivial for two sequences $(u_n)_{n\in\mathbb{N}}$ and $(v_n)_{n\in\mathbb{N}}$ such that

$$\begin{cases} (u_n)_{n\in\mathbb{N}} \text{ is increasing,} \\ (v_n)_{n\in\mathbb{N}} \text{ is decreasing,} \\ \lim\limits_{n\to+\infty} v_n - u_n = 0. \end{cases} \tag{3.1}$$

In this case,

$$\begin{cases} \text{the two sequences converge to the same limit } \ell, \\ \forall p \in \mathbb{N} \quad u_p \le \lim\limits_{n\to+\infty} u_n = \ell = \lim\limits_{n\to+\infty} v_n \le v_p, \\ \qquad \left|\ell - \frac{u_p+v_p}{2}\right| \le \frac{v_p-u_p}{2}. \end{cases}$$

A mathematical analysis shows that the two following sequences satisfy the above properties and converge to $\sqrt{ab}$ when $0 < a < b$:

$$u_0 = a, \qquad v_0 = b > a, \qquad u_{n+1} = \frac{2\,u_n v_n}{u_n + v_n}, \qquad v_{n+1} = \frac{u_n + v_n}{2}.$$

The common limit of these two sequences is called arithmetic-harmonic mean since the arithmetic mean of a and b is the average $(a+b)/2$, and the harmonic mean is the inverse of the average inverse: $1/h = (1/a + 1/b)/2 = (a+b)/(2ab)$. The following program checks the limit for given numerical values:

```
sage: U = 2.0; V = 50.0
sage: while V-U >= 1.0e-6:      # 1.0e-6 stands for 1.0*10^-6
....:   temp = U
....:   U = 2 * U * V / (U + V)
....:   V = (temp + V) / 2
sage: U, V
(9.99999999989256, 10.0000000001074)
```

The values u_{n+1} and v_{n+1} depend on u_n and v_n; for this reason the main loop of this program introduces an auxiliary variable `temp` to correctly compute the new values u_{n+1}, v_{n+1} of `U`, `V` from the previous values u_n, v_n. The two left blocks below define the same sequences, while the right one builds two other sequences:

```
temp = 2*U*V/(U+V)
V = (U+V)/2
U = temp
```

```
U,V = 2*U*V/(U+V),(U+V)/2
```

(parallel assignment)

```
U = 2*U*V/(U+V)
V = (U+V)/2
```

$$u'_{n+1} = \frac{2u'_n v'_n}{u'_n + v'_n}$$
$$v'_{n+1} = \frac{u'_{n+1} + v'_n}{2}$$

The series $S_n = \sum_{k=0}^{n} (-1)^k a_k$ is *alternating* as soon as the sequence $(a_k)_{k\in\mathbb{N}}$ is decreasing and tends to zero. Since S is alternating, the two subsequences

$(S_{2n})_{n\in\mathbb{N}}$ and $(S_{2n+1})_{n\in\mathbb{N}}$ satisfy Eq. (3.1), with common limit say ℓ. Hence the sequence $(S_n)_{n\in\mathbb{N}}$ also converges to ℓ and we have $S_{2p+1} \leq \ell = \lim_{n\to+\infty} S_n \leq S_{2p}$.

The following program illustrates this result for the sequence $a_k = 1/k^3$ from $k = 1$, by storing in two variables `U` and `V` the partial sums S_{2n} and S_{2n+1} enclosing the limit:

```
sage: U = 0.0          # the sum S0 is empty, of value zero
sage: V = -1.0         # S1 = -1/1^3
sage: n = 0            # U and V contain S(2n) and S(2n+1)
sage: while U-V >= 1.0e-6:
....:   n = n+1              # n += 1 is equivalent
....:   U = V + 1/(2*n)^3    # going from S(2n-1) to S(2n)
....:   V = U - 1/(2*n+1)^3 # going from S(2n) to S(2n+1)
sage: V, U
(-0.901543155458595, -0.901542184868447)
```

The main loop increases the value of n until the two terms S_{2n} and S_{2n+1} are close enough. The two variables `U` and `V` contain two consecutive terms; the loop body computes S_{2n} from S_{2n-1}, and then S_{2n+1} from S_{2n}, whence the crossed assignments to `U` and `V`.

The program halts when two consecutive terms S_{2n+1} and S_{2n} surrounding the limit are close enough, the approximation error — without taking into account rounding errors — satisfies then $0 \leq a_{2n+1} = S_{2n} - S_{2n+1} \leq 10^{-6}$.

Programming these five alternating series is similar:

$$\sum_{n\geq 2} \frac{(-1)^n}{\log n}, \qquad \sum_{n\geq 1} \frac{(-1)^n}{n}, \qquad \sum_{n\geq 1} \frac{(-1)^n}{n^2},$$
$$\sum_{n\geq 1} \frac{(-1)^n}{n^4}, \qquad \sum_{n\geq 1} (-1)^n e^{-n\ln n} = \sum_{n\geq 1} \frac{(-1)^n}{n^n}.$$

The terms of those series converge more or less rapidly to 0, thus the limit approximations require more or fewer computations.

Looking for a precision of 3, 10, 20 or 100 digits on the limits of these series consists in solving the following inequalities:

$$1/\log n \leq 10^{-3} \iff n \geq e^{(10^3)} \approx 1.97 \cdot 10^{434}$$

$$1/n \leq 10^{-3} \iff n \geq 10^3 \qquad 1/n \leq 10^{-10} \iff n \geq 10^{10}$$
$$1/n^2 \leq 10^{-3} \iff n \geq \sqrt{10^3} \approx 32 \qquad 1/n^2 \leq 10^{-10} \iff n \geq 10^5$$
$$1/n^4 \leq 10^{-3} \iff n \geq (10^3)^{1/4} \approx 6 \qquad 1/n^4 \leq 10^{-10} \iff n \geq 317$$
$$e^{-n\log n} \leq 10^{-3} \iff n \geq 5 \qquad e^{-n\log n} \leq 10^{-10} \iff n \geq 10$$

$$1/n^2 \leq 10^{-20} \iff n \geq 10^{10} \qquad 1/n^2 \leq 10^{-100} \iff n \geq 10^{50}$$
$$1/n^4 \leq 10^{-20} \iff n \geq 10^5 \qquad 1/n^4 \leq 10^{-100} \iff n \geq 10^{25}$$
$$e^{-n\log n} \leq 10^{-20} \iff n \geq 17 \qquad e^{-n\log n} \leq 10^{-100} \iff n \geq 57$$

In the simplest cases solving these inequalities yields an index n from which the value S_n is close enough to the limit ℓ, and then a `for` enumeration loop is

possible. However, when it is not possible to solve the inequality $a_n \leq 10^{-p}$, a `while` loop is necessary.

Numerical approximations of some of the above limits are too expensive, in particular when the index n gets as large as 10^{10} or 10^{12}. A mathematical study can sometimes determine the limit or approach it by other methods, like for the series giving values of the Riemann zeta function:

$$\lim_{n\to+\infty} \sum_{k=1}^{n} \frac{(-1)^k}{k^3} = -\frac{3}{4}\,\zeta(3), \qquad \text{with } \zeta(p) = \lim_{n\to+\infty} \sum_{k=1}^{n} \frac{1}{k^p},$$

$$\lim_{n\to+\infty} \sum_{k=1}^{n} \frac{(-1)^k}{k} = -\log 2, \qquad \lim_{n\to+\infty} \sum_{k=1}^{n} \frac{(-1)^k}{k^2} = -\frac{\pi^2}{12},$$

$$\lim_{n\to+\infty} \sum_{k=1}^{n} \frac{(-1)^k}{k^4} = -\frac{7\pi^4}{6!}.$$

Sage is able to compute symbolically some of these series, and determine a 1200-digit numerical approximation of $\zeta(3)$ in a few seconds, by doing far fewer operations than the 10^{400} ones required by the definition:

```
sage: k = var('k');                    sum((-1)^k/k, k, 1, +oo)
-log(2)
sage: sum((-1)^k/k^2, k, 1, +oo),  sum((-1)^k/k^3, k, 1, +oo)
(-1/12*pi^2, -3/4*zeta(3))
sage: -3/4 * zeta (N(3, digits = 1200))
-0.90154267736969571404980362113358749307373971925537416134\
4
20366650637865433973481763984190520700144360964936834644553\
9
56386899699900496241033229762790592512109045633721202005003\
9
...
01999549265288929706980408015180833590815343731070535991927\
1
79897015140616356032852450242460506051977442139028914505453\
8
90196121635914683781391659806428667225534381770353976017030626\
2
```

3.2.2 Conditionals

Another important instruction is the conditional (or test), which enables us to execute some instructions depending on the result of a boolean condition. The structure of the conditional and two possible syntaxes are:

```
if a condition:
    an instruction sequence
```

```
if a condition:
    an instruction sequence
else:
    another instruction sequence
```

The Syracuse sequence is defined using a parity condition:

$$u_0 \in \mathbb{N} - \{0\}, \qquad u_{n+1} = \begin{cases} u_n/2 & \text{if } u_n \text{ is even} \\ 3u_n + 1 & \text{if } u_n \text{ is odd.} \end{cases}$$

The Collatz conjecture says — with no known proof in 2017 — that for all initial values $u_0 \in \mathbb{N} - \{0\}$, there exists a rank n for which $u_n = 1$. The next terms are then 4, 2, 1, 4, 2, etc. The computation of each term of this sequence requires a parity test. This condition is checked within a *while* loop, which determines the smallest $n \in \mathbb{N}$ satisfying $u_n = 1$:

```
sage: u = 6 ; n = 0
sage: while u != 1:    # the test u <> 1 is also possible
....:   if u % 2 == 0: # the operator % yields the remainder
....:     u = u//2     # //: Euclidean division quotient
....:   else:
....:     u = 3*u+1
....:   n = n+1
sage: n
8
```

Checking whether u_n is even is done by comparing to 0 the remainder of the Euclidean division of u_n by 2. The variable `n` at the end of the block is the number of iterations. The loop ends as soon as $u_n = 1$; for example if $u_0 = 6$ then $u_8 = 1$ and $8 = \min\{p \in \mathbb{N} | u_p = 1\}$:

$p =$	0	1	2	3	4	5	6	7	8	9	10	$\cdots$
$u_p =$	6	3	10	5	16	8	4	2	1	4	2	$\cdots$

Verifying step-by-step the correct behaviour of the loop can be done using a *spy-instruction* `print(u, n)` inside the loop body.

The `if` instruction also allows nested tests in the `else` branch using the `elif` keyword. The two following structures are thus equivalent:

```
if a condition cond1:                          if cond1:
     an instruction sequence inst1                 inst1
else:                                          elif cond2:
    if a condition cond2:                          inst2
         an instruction sequence inst2         elif cond3:
    else:                                          inst3
        if a condition cond3:                  else:
             an instruction sequence inst3         instn
        else:
             in other cases instn
```

Like for loops, small instruction sequences may be put after the colon on the same line, rather than in an indented block below.

3.2.3 Procedures and Functions

General Syntax. As in other computer languages, the Sage user can define her/his own procedures or functions, using the `def` command whose syntax is detailed below. In this book, we call a *function* (resp. *procedure*) a sub-program

with zero, one or several arguments, which returns (resp. does not return) a result. Let us define the function $(x, y) \mapsto x^2 + y^2$:

```
sage: def fct2 (x, y):
....:   return x^2 + y^2
sage: a = var('a')
sage: fct2 (a, 2*a)
5*a^2
```

The function evaluation ends with the `return` command, whose argument, here $x^2 + y^2$, is the result of the function.

A procedure is like a function, but does not return any value, and without any `return` instruction the instruction body of the procedure is evaluated until its end. In fact a procedure returns the `None` value, which means "nothing".

By default, all variables appearing in a function are considered local variables. Local variables are created at each function call, destroyed at the end of the function, and do not interact with other variables of the same name. In particular, global variables are not modified by the evaluation of a function having local variables of the same name:

```
sage: def foo (u):
....:   t = u^2
....:   return t*(t+1)
sage: t = 1 ; u = 2
sage: foo(3), t, u
(90, 1, 2)
```

It is possible to modify a global variable from within a function, with the `global` keyword:

```
sage: a = b = 1
sage: def f(): global a; a = b = 2
sage: f(); a, b
(2, 1)
```

Consider again the computation of the arithmetic-harmonic mean of two positive numbers:

```
sage: def AHmean (u, v):
....:     u, v = min(u, v), max(u, v)
....:     while v-u > 2.0e-8:
....:         u, v = 2*u*v/(u+v), (u+v)/2
....:     return (u+v) / 2
```

```
sage: AHmean (1., 2.)
1.41421356237309
sage: AHmean                        # corresponds to a function
  <function AHmean at ...>
```

The `AHmean` function has two parameters `u` and `v` which are local variables, whose initial values are those of the function arguments; for example with `AHmean (1., 2.)` the function body begins with $u = 1.0$ and $v = 2.0$.

The structured programming paradigm recommends to have the `return` statement at the very end of the function body. However, it is possible to put it in the middle of the instruction block, then the following instructions will not be executed. And the function body might contain several `return` occurrences.

Translating the mathematician's viewpoint into the computer suggests the use of functions that return results from their arguments, instead of procedures that output those results with a `print` command. The Sage computer algebra system is itself built on numerous functions like `exp` or `solve`, which return a result, for example a number, an expression, a list of solutions, etc.

Iterative and Recursive Methods. As we have seen above, a user-defined function is a sequence of instructions. A function is called *recursive* when during its evaluation, it calls itself with different parameters. The factorial sequence is a toy example of recursive sequence:

$$0! = 1, \qquad (n+1)! = (n+1)\,n! \quad \text{for all } n \in \mathbb{N}.$$

The two following functions yield the same result for a nonnegative integer argument n; the first function uses the iterative method with a `for` loop, while the second one is a *word-by-word* translation of the above recursive definition:

```
sage: def fact1 (n):
....:     res = 1
....:     for k in [1..n]: res = res*k
....:     return res
```

```
sage: def fact2 (n):
....:     if n == 0: return 1
....:     else: return n*fact2(n-1)
```

The Fibonacci sequence is a recurrent relation of order 2 since u_{n+2} depends on u_n and u_{n+1}:

$$u_0 = 0, \qquad u_1 = 1, \qquad u_{n+2} = u_{n+1} + u_n \quad \text{for all } n \in \mathbb{N}.$$

The function `fib1` below applies an iterative scheme to compute terms of the Fibonacci sequence: the variables `U` and `V` store the two previous values before computing the next one:

```
sage: def fib1 (n):
....:   if n == 0 or n == 1: return n
....:   else:
....:     U = 0 ; V = 1 # the initial terms u0 and u1
....:     for k in [2..n]: W = U+V ; U = V ; V = W
....:     return V
sage: fib1(8)
```

```
21
```

The `for` loop applies the relation $u_n = u_{n-1} + u_{n-2}$ from $n = 2$. Note: a parallel assignment `U,V = V,U+V` in place of `W=U+V ; U=V ; V=W` would avoid the need of an auxiliary variable `W`, and would translate the order-1 vectorial recurrence $X_{n+1} = f(X_n)$ with $f(a, b) = (b, a + b)$, for $X_n = (u_n, u_{n+1})$. Those iterative methods are efficient, however programming them requires to manually deal with variables corresponding to different terms of the sequence.

On the contrary, the recursive function `fib2` follows more closely the mathematical definition of the Fibonacci sequence, which makes its programming and understanding easier:

```
sage: def fib2 (n):
....:   if 0 <= n <= 1: return n      # for n = 0 or n = 1
....:   else: return fib2(n-1) + fib2(n-2)
```

The result of this function is the value returned by the conditional statement: either 0 or 1 respectively for $n = 0$ and $n = 1$, otherwise the sum `fib2(n-1)+fib2(n-2)`; each branch of the test consists of a `return` instruction.

This method is however less efficient since several computations are duplicated. For example `fib2(5)` evaluates `fib2(3)` and `fib2(4)`, which are in turn evaluated in the same manner. Therefore, Sage computes twice `fib2(3)` and three times `fib2(2)`. This recursive process ends by the evaluation of either `fib2(0)` or `fib2(1)`, of value 0 or 1, and the evaluation of `fib2(n)` eventually consists in computing u_n by adding u_n ones, and u_{n-1} zeroes. The total number of additions performed to compute u_n is thus $u_{n+1} - 1$. This number grows very quickly, and no computer is able to compute u_{100} this way.

Other methods are also possible, for example remembering the intermediate terms using the decorator `@cached_function`, or using properties of matrix powers: the following paragraph shows how to compute the millionth term of this sequence. For example, compare the efficiency of the function `fib2` defined above with the following one, for example on $n = 30$:

```
sage: @cached_function
sage: def fib2a (n):
....:   if 0 <= n <= 1: return n
....:   else: return fib2a(n-1) + fib2a(n-2)
```

3.2.4 Example: Fast Exponentiation

The naive method for computing a^n for $n \in \mathbb{N}$ performs n multiplications by a using a `for` loop:

```
sage: a = 2; n = 6; res = 1          # 1 is the product neutral element
sage: for k in [1..n]: res = res*a
sage: res                            # the value of res is 2^6
64
```

Integer powers often arise in mathematics and computer science; this paragraph discusses a general method to compute a^n in a much faster way than the naive method. The sequence $(u_n)_{n\in\mathbb{N}}$ below satisfies $u_n = a^n$; this follows by induction from the equalities $a^{2k} = (a^k)^2$ and $a^{k+1} = a\,a^k$:

$$u_n = \begin{cases} 1 & \text{if } n = 0, \\ u_{n/2}^2 & \text{if } n \text{ is even positive}, \\ a\,u_{n-1} & \text{if } n \text{ is odd.} \end{cases} \tag{3.2}$$

For example, for $n = 11$:

$$u_{11} = a\,u_{10}, \quad u_{10} = u_5^2, \quad u_5 = a\,u_4, \quad u_4 = u_2^2,$$
$$u_2 = u_1^2, \quad u_1 = a\,u_0 = a;$$

therefore:

$$u_2 = a^2, \quad u_4 = u_2^2 = a^4, \quad u_5 = a\,a^4 = a^5,$$
$$u_{10} = u_5^2 = a^{10}, \quad u_{11} = a\,a^{10} = a^{11}.$$

The computation of u_n only involves terms u_k with $k \in \{0, ..., n-1\}$, and is thus well performed in a finite number of operations.

This example also shows that u_{11} is obtained after the evaluation of 6 terms u_{10}, u_5, u_4, u_2, u_1 and u_0, which performs 6 multiplications only. In general, the computation of u_n requires between $\log n/\log 2$ and $2\log n/\log 2$ multiplications. Indeed, u_n is obtained from u_k, $k \le n/2$, with one or two additional steps, according to the parity of n. This method is thus much faster than the naive one when n is large: about twenty products for $n = 10^4$ instead of 10^4 products:

indices k:	10 000	5 000	2 500	1 250	625	624	312	156	78
	39	38	19	18	9	8	4	2	1

However, this method is not always the best one; the following operations using b, c, d and f perform 5 products to compute a^{15}, whereas the above method — using u, v, w, x and y — requires 6 products, without counting the initial product $a \cdot 1$:

$b = a^2 \quad c = ab = a^3 \quad d = c^2 = a^6 \quad f = cd = a^9 \quad df = a^{15}$: 5 products;
$u = a^2 \quad v = au = a^3 \quad w = v^2 = a^6$
$x = aw = a^7 \quad y = x^2 = a^{14} \quad ay = a^{15}$: 6 products.

The recursive function `pow1` uses the recurrent sequence (3.2) to compute a^n:

```
sage: def pow1 (a, n):
....:   if n == 0: return 1
....:   elif n % 2 == 0: b = pow1 (a, n//2); return b*b
....:   else: return a * pow1(a, n-1)
```

```
sage: pow1 (2, 11)                # result is 2^11
```

```
2048
```

The number of operations performed by this function is the same as a computation by hand using (3.2). In the case n is even, if the instructions `b = pow1(a, n//2);return b*b` would be replaced by `pow1(a, n//2)*pow1(a, n//2)`, Sage would perform much more computations because, like for the recursive function `fib2` for the Fibonacci sequence, some calculations would be duplicated. We would then have of the order of n products, i.e., as many as with the naive method.

Note that instead of `b = pow1(a, n//2);return b*b`, we could write `return pow1(a*a, n//2)`.

The program below performs the same computation of a^n using an iterative method:

```
sage: def pow2 (u, k):
....:   v = 1
....:   while k != 0:
....:     if k % 2 == 0: u = u*u ; k = k//2
....:     else: v = v*u ; k = k-1
....:   return v
```

```
sage: pow2 (2, 10)                  # result is 2^10
1024
```

The fact that `pow2(a, n)` returns a^n is shown by verifying that after each iteration the values of the variables `u`, `v` and `k` satisfy $v\,u^k = a^n$, for whatever parity of k. Before the first iteration $v = 1$, $u = a$ and $k = n$; after the last one $k = 0$, thus $v = a^n$.

The successive values of the integer variable `k` are nonnegative, and they form a decreasing sequence. Hence this variable can only take a finite number of values before being zero and terminating the loop.

Despite their apparent differences — `pow1` is recursive, while `pow2` is iterative — those two functions express almost the same algorithm: the only difference is that a^{2k} is evaluated as $(a^k)^2$ in `pow1`, and as $(a^2)^k$ in `pow2`, through the update of the variable `u`.

The method presented here is not limited to the computation of a^n where a is a number and n a positive integer, it applies to any associative law (which is needed to preserve usual properties of iterated products). For instance, by replacing the integer 1 by the $m \times m$ unit matrix $\mathbf{1}_m$, the two above functions would evaluate powers of square matrices. Those functions show how to efficiently implement the power operator "^" upon multiplication, and are similar to the method implemented within Sage.

For example, using powers of matrices enables us to compute much larger terms of the Fibonacci sequence:

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}, \qquad X_n = \begin{pmatrix} u_n \\ u_{n+1} \end{pmatrix}, \qquad AX_n = X_{n+1}, \qquad A^n X_0 = X_n.$$

The corresponding Sage program fits in two lines, and the wanted result is the first coordinate of the matrix product $A^n X_0$, which effectively works for $n = 10^7$; the `fib3` and `fib4` programs are equivalent, and their efficiency comes from the fact that Sage implements a fast exponentiation method:

```
sage: def fib3 (n):
....:   A = matrix ([[0, 1], [1, 1]]) ; X0 = vector ([0, 1])
....:   return (A^n*X0)[0]
```

```
sage: def fib4 (n):
....:   return (matrix([[0,1], [1,1]])^n * vector([0,1]))[0]
```

3.2.5 Input and Output

The `print` instruction is the main output command. By default, its arguments are printed one after the other, separated by spaces, with a newline after the command:

```
sage: print 2^2, 3^3, 4^4 ; print 5^5, 6^6
4 27 256
3125 46656
```

A comma at the end tells the next `print` instruction to continue on the same line:

```
sage: for k in [1..10]: print '+', k,
+ 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
```

To print results without intermediate spaces, we can transform them into a character string using the `str(..)` function, and concatenate strings with the "`+`" operator:

```
sage: print 10, 0.5 ; print 10+0.5 ; print 10.0, 5
10 0.500000000000000
10.5000000000000
10.0000000000000 5
sage: print 10+0, 5 ; print(str(10)+str(0.5))
10 5
100.500000000000000
```

The last section of this chapter discusses in more detail character strings.

The `print` command is also able to format the output: the following example prints a table of fourth powers using the `%.d` placeholder and the `%` operator:

```
sage: for k in [1..6]: print('%2d^4 = %4d' % (k, k^4))
 1^4 =    1
 2^4 =   16
 3^4 =   81
 4^4 =  256
 5^4 =  625
 6^4 = 1296
```

The `%` operator replaces the expressions to its right in the character string to its left, in place of the placeholders like `%2d` or `%.4f`. In the above example the `%4d` specifier adds some left padding spaces to the string representing k^4, to get at least four characters. Likewise, the `%.4f` placeholder in `'pi = %.4f' % n(pi)` outputs `pi = 3.1416` with four digits after the decimal point.

In a terminal, the `raw_input('message')` command prints the text `message`, waits a keyboard input validated by the ⟨Enter⟩ key, and returns the user-given character string.

3.3 Lists and Other Data Structures

This section discusses some data structures available in Sage: character strings, lists — either mutable or immutable —, sets and dictionaries.

3.3.1 List Creation and Access

The list in computer science and the n-tuple in mathematics allow the enumeration of mathematical objects. In a pair — with $(a, b) \neq (b, a)$ — and an n-tuple, each object has its own position, contrary to a set.

A list is defined by surrounding its elements with square brackets `[...]`, separated by commas. Assigning the triple $(10, 20, 30)$ to the variable `L` is done as follows, and the empty list is defined as:

```
sage: L = [10, 20, 30]
sage: L
[10, 20, 30]
sage: []                    # [] is the empty list
[]
```

The list indices are increasing from 0, 1, 2, etc. The element of index k of a list L is accessed simply by `L[k]`, in mathematical terms this corresponds to the canonical projection on the k-th coordinate. The number of elements of a list is given by the `len` function[5]:

```
sage: L[1], len(L), len([])
(20, 3, 0)
```

Modifying an element is done the same way, by simply assigning the corresponding index. Hence the following command modifies the third term of the list, whose index is 2:

```
sage: L[2] = 33
sage: L
[10, 20, 33]
```

Negative indices access end-of-list elements, `L[-1]` referring to the last one:

[5]The output of `len` is a Python integer of type `int`, to get a Sage integer we write `Integer(len(..))`.

```
sage: L = [11, 22, 33]
sage: L[-1], L[-2], L[-3]
(33, 22, 11)
```

The command `L[p:q]` extracts the sub-list `[L[p], L[p+1], ..., L[q-1]]`, which is empty if $q \leq p$. Negative indices allow to reference the last terms of the list; finally `L[p:]` is equivalent to `L[p:len(L)]`, and `L[:q]` to `L[0:q]`:

```
sage: L = [0, 11, 22, 33, 44, 55]
sage: L[2:4]
[22, 33]
sage: L[-4:4]
[22, 33]
sage: L[2:-2]
[22, 33]
sage: L[:4]
[0, 11, 22, 33]
sage: L[4:]
[44, 55]
```

Similarly to the `L[n] = ...` command which modifies an element of the list, the assignment `L[p:q] = [...]` substitutes all elements between index p included and index q excluded:

```
sage: L = [0, 11, 22, 33, 44, 55, 66, 77]
sage: L[2:6] = [12, 13, 14]          # substitutes [22, 33, 44, 55]
```

Therefore, `L[:1] = []` and `L[-1:] = []` delete respectively the first and last term of a list, and likewise `L[:0] = [a]` and `L[len(L):] = [a]` insert the element a respectively in front and in tail of the list. More generally the following equalities hold:

$$\texttt{L} = [\ell_0, \ell_1, \ell_2, \ldots, \ell_{n-1}] = [\ell_{-n}, \ell_{1-n}, \ldots \ell_{-2}, \ell_{-1}] \qquad \text{with } n = \texttt{len(L)},$$
$$\ell_k = \ell_{k-n} \quad \text{for } 0 \leq k < n, \qquad \ell_j = \ell_{n+j} \quad \text{for } -n \leq j < 0.$$

The operator `in` checks whether a list contains a given element, while "`==`" compares two lists elementwise. The two sub-lists below with positive or negative indices are equal:

```
sage: L = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
sage: L[3:len(L)-5] == L[3-len(L):-5]
True
sage: [5 in L, 6 in L]
[True, False]
```

While we have considered so far lists with integer elements, list elements can be any Sage object: numbers, expressions, other lists, etc.

3.3.2 Global List Operations

The addition operator "+" concatenates two lists, and the multiplication operator "*", together with an integer, performs an iterated concatenation:

```
sage: L = [1, 2, 3] ; L + [10, 20, 30]
[1, 2, 3, 10, 20, 30]
sage: 4 * [1, 2, 3]
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The concatenation of the two sub-lists `L[:k]` and `L[k:]` reconstructs the original list. This explains why the left bound `p` of a sub-list `L[p:q]` is included, while the right bound `q` is excluded:

$$\begin{aligned} \texttt{L} &= \texttt{L[:k]+L[k:]} = [\ell_0, \ell_1, \ell_2, \ldots, \ell_{n-1}] \\ &= [\ell_0, \ell_1, \ell_2, \ldots, \ell_{k-1}] + [\ell_k, \ell_{k+1}, \ell_{k+2}, \ldots, \ell_{n-1}]. \end{aligned}$$

This property is shown in the following example:

```
sage: L = 5*[10, 20, 30] ; L[:3]+L[3:] == L
True
```

The operator made from two points ".." makes it easy to construct integer lists without explicitly enumerating all elements, and can be mixed with isolated elements:

```
sage: [1..3, 7, 10..13]
[1, 2, 3, 7, 10, 11, 12, 13]
```

We explain below how to build the image of a list under a function, and a sub-list of a list. The corresponding functions are `map` and `filter`, together with the `[..for..x..in..]` construction. Mathematics often involve lists made by applying a function f to its elements:

$$(a_0, a_1, \ldots, a_{n-1}) \mapsto (f(a_0), f(a_1), \ldots, f(a_{n-1})).$$

The `map` command builds this "map": the following example applies the trigonometric function cos to a list of usual angles:

```
sage: map (cos, [0, pi/6, pi/4, pi/3, pi/2])
[1, 1/2*sqrt(3), 1/2*sqrt(2), 1/2, 0]
```

A user-defined function — with `def` — or a lambda-expression might also be used as first argument of `map`; the following command is equivalent to the above, using the function $t \mapsto \cos t$:

```
sage: map (lambda t: cos(t), [0, pi/6, pi/4, pi/3, pi/2])
[1, 1/2*sqrt(3), 1/2*sqrt(2), 1/2, 0]
```

The `lambda` command is followed by the parameters separated by commas, and the colon must be followed by exactly one expression, which is the function result (without the `return` keyword).

A `lambda` expression may contain a test, whence the following functions are equivalent:

```
fctTest1 = lambda x:  res1 if cond else res2
def fctTest2 (x):
  if cond:  return res1
  else:  return res2
```

As a consequence, the three following `map` commands are equivalent, the composition $N \circ \cos$ being expressed in different ways:

```
sage: map (lambda t: N(cos(t)), [0, pi/6, pi/4, pi/3, pi/2])
[1.00000000000000, 0.866025403784439, 0.707106781186548,
0.500000000000000, 0.000000000000000]
```

```
sage: map (N, map (cos, [0, pi/6, pi/4, pi/3, pi/2]))
[1.00000000000000, 0.866025403784439, 0.707106781186548,
0.500000000000000, 0.000000000000000]
```

```
sage: map (compose(N, cos), [0, pi/6, pi/4, pi/3, pi/2])
[1.00000000000000, 0.866025403784439, 0.707106781186548,
0.500000000000000, 0.000000000000000]
```

The `filter` command builds the sub-list of the elements satisfying a given condition. To get all integers in $1, ..., 55$ that are prime:

```
sage: filter (is_prime, [1..55])
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53]
```

The test condition might be defined inside the `filter` command, as in the following example which finds by exhaustive search all fourth roots of 7 modulo the prime 37; this equation has four solutions 3, 18, 19 and 34:

```
sage: p = 37 ; filter (lambda n: n^4 % p == 7, [0..p-1])
[3, 18, 19, 34]
```

Another way to build a list is using the *comprehension* form `[..for..x..in..]`; both commands below enumerate odd integers from 1 to 31:

```
sage: map(lambda n:2*n+1, [0..15])
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31]
sage: [2*n+1 for n in [0..15]]
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31]
```

The comprehension command is independent of the `for` loop. Associated with the `if` condition, it yields an equivalent construction to `filter`:

```
sage: filter (is_prime, [1..55])
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53]
sage: [p for p in [1..55] if is_prime(p)]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53]
```

In the two following examples, we combine the `if` and `filter` tests with the comprehension `for` to determine a list of primes congruent to 1 modulo 4, then a list of squares of prime numbers:

```
sage: filter (is_prime, [4*n+1 for n in [0..20]])
[5, 13, 17, 29, 37, 41, 53, 61, 73]
sage: [n^2 for n in [1..20] if is_prime(n)]
[4, 9, 25, 49, 121, 169, 289, 361]
```

In the first case the `is_prime` test is performed after the computation of $4n+1$, while in the second one the primality test is done before the computation of the square n^2.

The `reduce` function operates by associativity from left to right on the elements of a list. Let us define the following operation, say $\star$:

$$x \star y = 10x + y, \qquad \text{then } ((1 \star 2) \star 3) \star 4 = (12 \star 3) \star 4 = 1234.$$

The first argument of `reduce` is a two-parameter function, the second one is the list of its arguments:

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

A third optional argument gives the image of an empty list:

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

This third argument usually corresponds to the neutral element of the operation that is applied. The following example computes a product of odd integers:

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

The Sage functions `add`[6] and `prod` apply directly the `reduce` operator to compute sums and products; the three examples below yield the same result. The list form enables us to add an optional second argument which stands for the neutral element, 1 for the product and 0 for the sum, or a unit matrix for a matrix product:

```
sage: prod ([2*n+1 for n in [0..9]], 1) # a list with for
654729075
sage: prod ( 2*n+1 for n in [0..9])     # without a list
654729075
sage: prod (n for n in [0..19] if n%2 == 1)
654729075
```

The function `any` associated to the `or` operator, and the function `all` to the `and` operator, have similar syntax. Their evaluation terminates as soon as the result `True` or `False` obtained for one term avoids the evaluation of the next terms:

[6] Do not confuse `add` with `sum`, which looks for a symbolic expression of a sum.

```
sage: def fct (x): return 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
```

In contrast, the construction of the list `[fct(x) for x in [2, 1, 0]]` and the command `all([fct(x) for x in [2, 1, 0]])` produce an error because all terms are evaluated, including the last one with $x = 0$.

Nesting several `for` operators enables us to construct the cartesian product of two lists, or to define lists of lists. As seen in the following example, the leftmost `for` operator corresponds to the outermost loop:

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

The order therefore differs from that obtained by constructing a list of lists using nested `for` comprehensions:

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

The `map` command with several lists as arguments takes one element of each list in turn:

```
sage: map (lambda x, y: [x, y], [1..3], [6..8])
[[1, 6], [2, 7], [3, 8]]
```

Finally with the `flatten` command, we can concatenate lists on one or several levels:

```
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)             # equivalent to flatten (L, max_level = 3)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

These elementary list operations are quite useful in other parts of Sage; the following example computes the first successive derivatives of $x\,e^x$; the first argument of `diff` is the expression to differentiate, and the following argument is the derivation variable, or in the case of several arguments the variables with respect to which the expression should be successively differentiated:

```
sage: x = var('x')
sage: factor(diff(x*exp(x), [x, x]))
(x + 2)*e^x
sage: map(lambda n: factor(diff(x*exp(x), n*[x])), [0..6])
[x*e^x, (x + 1)*e^x, (x + 2)*e^x, (x + 3)*e^x, (x + 4)*e^x,
(x + 5)*e^x, (x + 6)*e^x]
sage: [factor (diff (x*exp(x), n*[x])) for n in [0..6]]
```

```
[x*e^x, (x + 1)*e^x, (x + 2)*e^x, (x + 3)*e^x, (x + 4)*e^x,
(x + 5)*e^x, (x + 6)*e^x]
```

The `diff` command admits more than one syntax. The parameters after the function f can be a list of variables, an enumeration of variables, or a variable and an order of derivation:

```
diff(f(x), x, x, x),   diff(f(x), [x, x, x]),   diff(f(x), x, 3).
```

We can also use `diff(f(x), 3)` for functions of one variable. The above results are a direct consequence of Leibniz' formula for iterated derivatives of a 2-term product, given the fact that the derivatives of order 2 or more of x are zero:

$$(xe^x)^{(n)} = \sum_{k=0}^{n} \binom{n}{k} x^{(k)} (e^x)^{(n-k)} = (x+n)e^x.$$

3.3.3 Main Methods on Lists

The `reverse` method reverts the order of elements in a list, and the `sort` method transforms the given list into a sorted one:

```
sage: L = [1, 8, 5, 2, 9] ; 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]
```

Both methods modify the list `L` in-place, the initial list being lost.

A first optional argument of `sort` enables us to choose the order relation, in form of a two-parameter function `Order(x, y)`. The returned value of this function must have the type `int` of the Python integers; it is negative, zero or positive, for example -1, 0 or 1, when $x \prec y$, $x = y$ or $x \succ y$, respectively. The transformed list $(x_0, x_1, \ldots, x_{n-1})$ satisfies $x_0 \preceq x_1 \preceq \cdots \preceq x_{n-1}$.

The lexicographic order of two number lists of same length is similar to the alphabetic order and is defined as follows, ignoring the first equal terms:

$$\begin{aligned} & P = (p_0, p_1, \ldots p_{n-1}) \prec_{\text{lex}} Q = (q_0, q_1, \ldots q_{n-1}) \\ \iff & \exists r \in \{0, \ldots, n-1\} \quad (p_0, p_1, \ldots, p_{r-1}) = (q_0, q_1, \ldots, q_{r-1}) \text{ and } p_r < q_r. \end{aligned}$$

The following function compares two lists of equal lengths. Despite the *a priori* infinite loop `while True`, the `return` commands ensure the termination, together with the finite length. The result is -1, 0 or 1 according to $P \prec_{\text{lex}} Q$, $P = Q$ or $P \succ_{\text{lex}} Q$:

```
sage: def alpha (P, Q):     # len(P) = len(Q) by hypothesis
....:   i = 0
....:   while True:
....:     if i == len(P): return int(0)
....:     elif P[i] < Q[i]: return int(-1)
....:     elif P[i] > Q[i]: return int(1)
....:     else: i = i+1
```

```
sage: alpha ([2, 3, 4, 6, 5], [2, 3, 4, 5, 6])
1
```

The following command sorts a list of lists of same length using the lexicographic order. The `alpha` function using the same order as used by Sage to compare two lists, the command `L.sort()` without optional argument is thus equivalent:

```
sage: L = [[2, 2, 5], [2, 3, 4], [3, 2, 4], [3, 3, 3],\
....: [1, 1, 2], [1, 2, 7]]
sage: L.sort (cmp = alpha) ; L
[[1, 1, 2], [1, 2, 7], [2, 2, 5], [2, 3, 4], [3, 2, 4], [3, 3, 3]]
```

The homogeneous lexicographic order first compares terms according to their weight, where the weight is the sum of coefficients, and only in the case of equal weights resorts to the lexicographic order:

$$\begin{aligned} & P = (p_0, p_1, \dots p_{n-1}) \prec_{\text{lexH}} Q = (q_0, q_1, \dots q_{n-1}) \\ \iff & \sum_{k=0}^{n-1} p_k < \sum_{k=0}^{n-1} q_k \text{ or } \Big(\sum_{k=0}^{n-1} p_k = \sum_{k=0}^{n-1} q_k \text{ and } P \prec_{\text{lex}} Q \Big). \end{aligned}$$

This function implements the homogeneous lexicographic order:

```
sage: def homogLex (P, Q):
....:  sp = sum (P) ; sq = sum (Q)
....:  if sp < sq: return int(-1)
....:  elif sp > sq: return int(1)
....:  else: return alpha (P, Q)
```

```
sage: homogLex ([2, 3, 4, 6, 4], [2, 3, 4, 5, 6])
-1
```

The Sage function `sorted` is a function in the mathematical sense: it takes as first argument a list and returns the corresponding sorted list, without modifying its argument, unlike `sort`.

Sage provides other methods on lists, to insert an element at the tail, to append a list at the end, to count the number of occurrences of an element:

`L.append(x)`	is equivalent to `L[len(L):] = [x]`
`L.extend(L1)`	is equivalent to `L[len(L):] = L1`
`L.insert(i, x)`	is equivalent to `L[i:i] = [x]`
`L.count(x)`	is equivalent to `len ([t for t in L if t == x])`

The commands `L.pop(i)` and `L.pop()` remove the element of index i, or the last one, and return the removed element; their behaviour is described by these two functions:

```
def pop1 (L, i):
  a = L[i]
  L[i:i+1] = []
  return a
```

```
def pop2 (L):
  return pop1 (L, len(L)-1)
```

In addition, `L.index(x)` returns the index of the first element equal to x, and `L.remove(x)` removes the first element equal to x. These commands raise an error when x is not in the list. Finally, the command `del L[p:q]` is equivalent to `L[p:q] = []`, and `del L[i]` removes the ith element.

Contrary to what happens in several other computer languages, these functions modify in-place the list `L`, without creating a new list.

3.3.4 Examples of List Manipulations

The following example constructs the list of even terms and the list of odd terms of a given list. This first solution goes twice through the list, and thus performs the parity tests twice:

```
sage: def fct1(L):
....:   return [filter (lambda n: n % 2 == 0, L),
....:           filter (lambda n: n % 2 == 1, L)]
```

```
sage: fct1([1..10])
[[2, 4, 6, 8, 10], [1, 3, 5, 7, 9]]
```

The second function below goes only once through the list, and constructs the two result lists element by element:

```
sage: def fct2 (L):
....:   res0 = [] ; res1 = []
....:   for k in L:
....:     if k%2 == 0: res0.append(k) # or res0[len(res0):] = [k]
....:     else: res1.append(k)        # or res1[len(res1):] = [k]
....:   return [res0, res1]
```

This program replaces the `for` loop and the auxiliary variables by a recursive call and an additional parameter:

```
sage: def fct3a (L, res0, res1):
....:   if L == []: return [res0, res1]
....:   elif L[0]%2 == 0: return fct3a(L[1:], res0+[L[0]], res1)
....:   else: return fct3a (L[1:], res0, res1+[L[0]])
```

```
sage: def fct3 (L): return fct3a (L, [], [])
```

The parameters `res0` and `res1` contain the first element already treated, and the parameter list `L` has one term less at each recursive call.

The second example below extracts all maximal non-decreasing sequences of a list of numbers. Three variables are used, the first one `res` keeps track of all non-decreasing sequences already obtained, the `start` variable is the starting index of the current sub-sequence, and `k` is the loop index:

```
sage: def subSequences (L):
....:   if L == []: return []
....:   res = [] ; start = 0 ; k = 1
```

```
....:   while k < len(L):    # 2 consecutive terms are defined
....:     if L[k-1] > L[k]:
....:       res.append (L[start:k]) ; start = k
....:     k = k+1
....:   res.append (L[start:k])
....:   return res
```

```
sage: subSequences([1, 4, 1, 5])
[[1, 4], [1, 5]]
sage: subSequences([4, 1, 5, 1])
[[4], [1, 5], [1]]
```

The loop body deals with the kth element of the list. If the condition is fulfilled, the current non-decreasing sub-sequence ends, and we start a new sub-sequence, otherwise the current sub-sequence is extended by one term.

After the loop body, the `append` instruction adds to the final result the current sub-sequence, which contains at least one element.

3.3.5 Character Strings

Character strings are delimited by single or double quotes, `'...'` or `"..."`. Strings delimited by single quotes may contain double quotes, and vice versa. Strings can also be delimited by triple quotes `'''...'''`: in that case they may span several lines and contain single or double quotes.

```
sage: S = 'This is a character string.'
```

The escape character is the \ symbol, which allows to include end of lines by `\n`, quotes by `\"` or `\'`, tabulations by `\t`, the backslash character by `\\`. Character strings may contain characters with accents, and more generally any Unicode character:

```
sage: S = 'This is a déjà-vu example.'; S
'This is a d\xc3\xa9j\xc3\xa0-vu example.'
sage: print(S)
This is a déjà-vu example.
```

The comparison of two character strings is performed according to the internal encoding of each character. The length of a string is given by the `len` function, and the concatenation of strings is performed by the addition and multiplication symbols "`+`" and "`*`".

Accessing sub-strings of `S` is done as for lists using square brackets `S[n]`, `S[p:q]`, `S[p:]` and `S[:q]`, the result being a character string. The language forbids the replacement of an initial string by such an assignment, for this reason character strings are *immutable*.

The `str` function converts its argument into a character string. The `split` method cuts a given string at spaces:

```
sage: S='one two three four five six seven'; L=S.split(); L
['one', 'two', 'three', 'four', 'five', 'six', 'seven']
```

The very extensive Python library `re` may also be used to search sub-strings, words and regular expressions.

3.3.6 Shared or Duplicated Data Structures

A list in square brackets `[...]` can be modified by assigning some of its elements, by a change of the number of elements, or by methods like `sort` or `reverse`.

Assigning a list to a variable does not duplicate the data structure, which is shared. In the following example the lists `L1` and `L2` remain identical: they correspond to two *aliases* of the same object, and modifying one of them is visible on the other one:

```
sage: L1 = [11, 22, 33] ; L2 = L1
sage: L1[1] = 222 ; L2.sort() ; L1, L2
([11, 33, 222], [11, 33, 222])
sage: L1[2:3] = []; L2[0:0] = [6, 7, 8]
sage: L1, L2
([6, 7, 8, 11, 33], [6, 7, 8, 11, 33])
```

In contrast, the `map`, `filter` and `flatten` functions duplicate the data structures; so do the list construction by `L[p:q]` or `[..for..if..]`, and the concatenation by `+` and `*`.

In the above example, replacing on the first line `L2 = L1` by one of the next six commands completely changes the following results, since modifications on one list do not propagate to the other one. The two structures become independent, the two lists are distinct even if they have the same value; for example the assignment `L2 = L1[:]` copies the sub-list of `L1` from the first to last term, and thus fully duplicates the structure of `L1`:

```
L2 = [11, 22, 33]  L2 = copy(L1)  L2 = L1[:]
L2 = []+L1         L2 = L1+[]     L2 = 1*L1
```

Checking for shared data structures can be done in Sage using the `is` binary operator; if the answer is true, all modifications will have a side effect on both variables:

```
sage: L1 = [11, 22, 33] ; L2 = L1 ; L3 = L1[:]
sage: [L1 is L2, L2 is L1, L1 is L3, L1 == L3]
[True, True, False, True]
```

Copy operations on lists operate on one level only. As a consequence, modifying an element in a list of lists has a side effect despite the list copy at the outer level:

```
sage: La = [1, 2, 3] ; L1 = [1, La] ; L2 = copy(L1)
sage: L1[1][0] = 5          # [1, [5, 2, 3]] for L1 and L2
sage: [L1 == L2, L1 is L2, L1[1] is L2[1]]
[True, False, True]
```

The following instruction duplicates a list on two levels:

```
sage: map (copy, L)
```

whereas the `deepcopy` function recursively duplicates Python objects at all levels:

```
sage: La = [1, 2, 3] ; L1 = [1, La] ; L2 = deepcopy(L1)
sage: L1[1][0] = 5; [L1 == L2, L1 is L2, L1[1] is L2[1]]
[False, False, False]
```

The inverse lexicographic order is defined from the lexicographic order on n-tuples by reversing the order on each element:

$$\begin{aligned} & P = (p_0, p_1, \dots p_{n-1}) \prec_{\text{lexInv}} Q = (q_0, q_1, \dots q_{n-1}) \\ \iff & \exists r \in \{0, \dots, n-1\}, \quad (p_{r+1}, \dots, p_{n-1}) = (q_{r+1}, \dots, q_{n-1}) \text{ and } p_r > q_r. \end{aligned}$$

Programming this inverse lexicographic order might be done using the above-defined `alpha` function, which implements the lexicographic order. We have to copy the lists P and Q to perform the inversion without modifying the lists. More precisely the `lexInverse` function reverts the n-tuples by `reverse`, and returns the opposite of the Python integer corresponding to the wanted comparison: $-(P_1 \prec_{\text{lex}} Q_1)$:

```
sage: def lexInverse (P, Q):
....:   P1 = copy(P) ; P1.reverse()
....:   Q1 = copy(Q) ; Q1.reverse()
....:   return - alpha (P1, Q1)
```

The changes made on a list given as argument of a function are performed on the original list, since the functions do not copy arguments which are lists. Thus a function that would perform `P.reverse()`, in place of `P1 = copy(P)` and `P1.reverse()`, would modify definitively the list P; this *side effect* is usually not wanted.

The variable P is a local variable of the function, independent from any other global variable also called P, but this has nothing to do with modifications made to a list given as argument of the function.

The lists in Python and Sage are implemented as dynamic tables, contrary to Lisp and OCaml where lists are defined by a head t and a tail list Q. The Lisp command `cons(t,Q)` returns a list with head t without modifying the list Q, whereas in Python, adding an element e to a dynamic table T via `T.append(e)` modifies the table T. Both representations have advantages and drawbacks, and switching from one to the other is possible, however the efficiency of a given algorithm might greatly vary from one representation to the other.

3.3.7 Mutable and Immutable Data Structures

Lists enable us to construct and manipulate elements that can be modified: they are called *mutable* data structures.

Python also allows to define immutable objects. The immutable data structure corresponding to lists is called *sequence* or *tuple*, and is denoted with parentheses `(...)` instead of square brackets `[...]`. A tuple with only one element is defined by adding a comma after this element, to distinguish it from mathematical parentheses.

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

The operations on tuples are essentially the same as those on lists, for example `map` constructs the image of a tuple by a function, `filter` extracts a sub-sequence. In all cases the result is a list, and the `for` comprehension transforms a tuple in list:

```
sage: S1 = (1, 4, 9, 16, 25); [k for k in S1]
[1, 4, 9, 16, 25]
```

The `zip` command groups several lists or tuples term-by-term, and is equivalent to the following `map` command:

```
sage: L1 = [0..4]; L2 = [5..9]
sage: zip(L1, L2)
[(0, 5), (1, 6), (2, 7), (3, 8), (4, 9)]
sage: map(lambda x, y:(x, y), L1, L2)
[(0, 5), (1, 6), (2, 7), (3, 8), (4, 9)]
```

3.3.8 Finite Sets

Contrary to lists, the set data structure only keeps track of whether an element is present or absent, without considering its position or number of repetitions. Sage constructs finite sets via the `Set` function, applied to the list of its elements. The result is output with curly brackets:

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

The operator `in` checks whether a set contains a given element, and Sage allows the union of sets by `+` or `|`, the intersection by `&`, the set difference by `-`, and the symmetric difference using `^^`:

```
sage: E = Set([1, 2, 4, 8, 2, 2, 2]); F = Set([7, 5, 3, 1])
sage: 5 in E, 5 in F, E + F == F | E
(False, True, True)
sage: E & F, E - F, E ^^ F
({1}, {8, 2, 4}, {2, 3, 4, 5, 7, 8})
```

The `len(E)` command gives the cardinality of such a finite set. The operations `map`, `filter` and `for..if...` apply to sets as well as tuples, and yield lists as results. Accessing a given element is done via `E[k]`. The commands below construct in two different ways the list of elements of a set:

```
sage: E = Set([1, 2, 4, 8, 2, 2, 2])
sage: [E[k] for k in [0..len(E)-1]], [t for t in E]
([8, 1, 2, 4], [8, 1, 2, 4])
```

The following function checks whether E is a subset of F, using the union operator:

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

Contrary to lists, sets are immutable, and thus cannot be modified; their elements must also be immutable. Sets of tuples or sets of sets are thus possible, but not sets of lists:

```
sage: Set([Set([]), Set([1]), Set([2]), Set([1, 2])])
{{1, 2}, {}, {2}, {1}}
sage: Set([    (),     (1, ),    (2, ),    (1, 2) ])
{(1, 2), (2,), (), (1,)}
```

The following function scans all subsets of a set recursively:

```
sage: def Parts (EE):
....:   if EE == Set([]): return Set([EE])
....:   else:
....:     return withOrWithout (EE[0], Parts(Set(EE[1:])))
```

```
sage: def withOrWithout (a, E):
....:   return Set (map (lambda F: Set([a])+F, E)) + E
```

```
sage: Parts(Set([1, 2, 3]))
{{3}, {1, 2}, {}, {2, 3}, {1}, {1, 3}, {1, 2, 3}, {2}}
```

The `withOrWithout(a, E)` function call takes a set E of subsets, and constructs the set twice as large made from those subsets, and those subsets added (in the set union sense) with a. The recursive construction starts with a set with one element $E = \{\emptyset\}$.

3.3.9 Dictionaries

Last but not least, Python, and thus Sage, provides the notion of dictionary. Like a phone book, a dictionary associates a value to a given key.

The keys of a dictionary might be of any immutable type: numbers, characters strings, tuples, etc. The syntax is like lists, using assignments from the empty dictionary `dict()` which can be written `{}` too:

```
sage: D={}; D['one']=1; D['two']=2; D['three']=3; D['ten']=10
sage: D['two'] + D['three']
5
```

The above example shows how to add an entry (key,value) to a dictionary, and how to access the value associated to a given key via `D[...]`.

The operator `in` checks whether a key is in a dictionary, and the commands `del D[x]` or `D.pop(x)` erase the entry of key x in this dictionary.

The following example demonstrates how a dictionary can be used to represent a function on a finite set:

$$E = \{a_0, a_1, a_2, a_3, a_4, a_5\}, \quad \begin{array}{lll} f(a_0) = b_0, & f(a_1) = b_1, & f(a_2) = b_2, \\ f(a_3) = b_0, & f(a_4) = b_3, & f(a_5) = b_3. \end{array}$$

Methods on dictionaries are comparable to those on other enumerated data structures. The program below implements the above function, and gives the input set E and the output set $\mathrm{Im}\, f = f(E)$ via the methods `keys` and `values`:

```
sage: D = {'a0':'b0', 'a1':'b1', 'a2':'b2', 'a3':'b0',\
....: 'a4':'b3', 'a5':'b3'}
sage: E  = Set(D.keys()) ; Imf = Set(D.values())
sage: Imf == Set(map (lambda t:D[t], E))     # is equivalent
True
```

This last command directly translates the mathematical definition $\mathrm{Im}\, f = \{f(x)|x \in E\}$. Dictionaries may also be constructed from lists or pairs `[`*key*`,` *value*`]` via the following command:

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

The two following commands, applied to the keys or to the dictionary itself are, by construction, equivalent to `D.values()`:

```
map (lambda t:D[t], D)    map (lambda t:D[t], D.keys())
```

The following test on the number of distinct values determines if the function represented by `D` is injective, `len(D)` being the number of dictionary entries:

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

The first two commands below build the image $f(F)$ and the preimage $f^{-1}(G)$ of subsets F and G of a function defined by the dictionary D; the last one constructs the dictionary `DR` corresponding to the inverse function f^{-1} of f, assumed to be bijective:

```
sage: Set([D[t] for t in F])
sage: Set([t for t in D if D[t] in G])
sage: DR = dict((D[t], t) for t in D)
```

4 Graphics

Drawing a function of one or two variables, or a series of data, makes it easier to grasp a mathematical or physical phenomenon, and helps us make conjectures. In this chapter, we illustrate the graphical capabilities of Sage using several examples.

4.1 2D Graphics

Several definitions of a plane curve are possible: as the graph of a function, from a parametric system, using polar coordinates, or by an implicit equation. We detail these four cases, and give some examples of data visualisation.

4.1.1 Graphical Representation of a Function

To draw the graph of a symbolic or Python function on an interval $[a, b]$, we use `plot(f(x), a, b)` or the alternative syntax `plot(f(x), x, a, b)`.

```
sage: plot(x * sin(1/x), x, -2, 2, plot_points=500)
```

Among the numerous options of the `plot` command, we mention the following:

- `plot_points` (default value 200): minimal number of computed points;
- `xmin` and `xmax`: interval bounds over which the function is displayed;
- `color`: colour of the graph, either a RGB triple, a character string such as `'blue'`, or an HTML colour like `'#aaff0b'`;
- `detect_poles` (default value `False`): enables to draw a vertical asymptote at poles of the function;

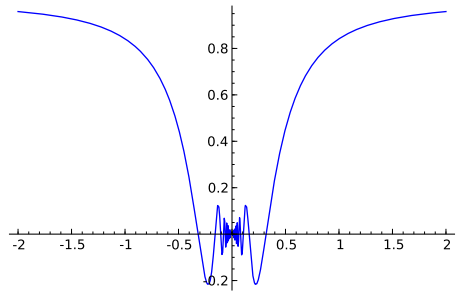


FIGURE 4.1 – Graph of $x \mapsto x \sin \frac{1}{x}$.

- `alpha`: line transparency;
- `thickness`: line thickness;
- `linestyle`: style of the line, either dotted with `':'`, dash-dotted with `'-.'`, or solid with the default value `'-'`.

To visualise the graph, we assign the graphical object to a variable, say `g`, then we use the `show` command; in addition we can give bounds for the y-axis (`g.show(ymin=-1, ymax=3)`) or choose the aspect ratio (`g.show(aspect_ratio=1)` to have equal scales for x and y).

The graph obtained may be exported using the `save` command into several formats defined by the suffixes `.pdf`, `.png`, `.ps`, `.eps`, `.svg` and `.sobj`:

```
g.save(name, aspect_ratio=1, xmin=-1, xmax=3, ymin=-1, ymax=3)
```

To include such a figure in a LaTeX document using the `includegraphics` command, one should use the `eps` suffix (encapsulated PostScript) if the document is to be compiled with `latex`, and the `pdf` suffix (to be preferred to `png`, to obtain a better resolution) if the document is to be compiled with `pdflatex`.

Let us draw on the same graphics the sine function and its first Taylor polynomials at 0.

```
sage: def p(x, n):
....:     return(taylor(sin(x), x, 0, n))
sage: xmax = 15 ; n = 15
sage: g = plot(sin(x), x, -xmax, xmax)
sage: for d in range(n):
....:   g += plot(p(x, 2 * d + 1), x, -xmax, xmax,\
....:     color=(1.7*d/(2*n), 1.5*d/(2*n), 1-3*d/(4*n)))
sage: g.show(ymin=-2, ymax=2)
```

We can also create an animation, to see how the Taylor polynomials approximate better and better the sine function when their degree increases. To keep the animation, it suffices to save it in the `gif` format.

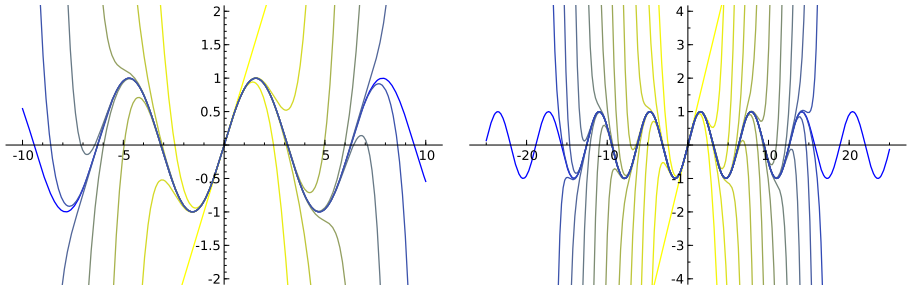


FIGURE 4.2 – Some Taylor polynomials of the sine function at 0.

```
sage: a = animate([[sin(x), taylor(sin(x), x, 0, 2*k+1)]\
....:     for k in range(0, 14)], xmin=-14, xmax=14,\
....:     ymin=-3, ymax=3, figsize=[8, 4])
sage: a.show(); a.save('path/animation.gif')
```

Let us return to the `plot` function to demonstrate, as an example, the Gibbs phenomenon. We draw the partial sum of order 20 of the square wave function.

```
sage: f2(x) = 1; f1(x) = -1
sage: f = piecewise([[(-pi,0),f1],[(0,pi),f2]])
sage: S = f.fourier_series_partial_sum(20,pi)
sage: g = plot(S, x, -8, 8, color='blue')
sage: saw(x) = x - 2 * pi * floor((x + pi) / (2 * pi))
sage: g += plot(saw(x) / abs(saw(x)), x, -8, 8, color='red')
sage: g
```

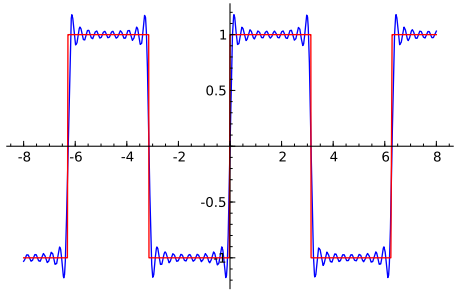


FIGURE 4.3 – Fourier series expansion of the square wave function.

In the code above, `f` is a piecewise function on $[-\pi;\pi]$, defined with the `piecewise` instruction. To extend `f` by 2π-periodicity, the simplest solution is to give an expression valid for any real number, such as `saw(x)/abs(saw(x))`. The sum of the 20 first terms of the Fourier series is:

$$S = \frac{4}{\pi}\left(\sin(x) + \frac{\sin(3x)}{3} + \frac{\sin(5x)}{5} + \cdots + \frac{\sin(19\pi)}{19}\right).$$

4.1.2 Parametric Curve

Parametric curves ($x = f(t)$, $y = g(t)$) may be visualised using the command `parametric_plot((f(t), g(t)), (t, a, b))`, where $[a, b]$ is the interval over which the parameter t ranges.

Let us show the parametric curve defined by the equations:

$$\begin{cases} x(t) = \cos(t) + \dfrac{1}{2}\cos(7t) + \dfrac{1}{3}\sin(17t), \\ y(t) = \sin(t) + \dfrac{1}{2}\sin(7t) + \dfrac{1}{3}\cos(17t). \end{cases}$$

```
sage: t = var('t')
sage: x = cos(t) + cos(7*t)/2 + sin(17*t)/3
sage: y = sin(t) + sin(7*t)/2 + cos(17*t)/3
sage: g = parametric_plot((x, y), (t, 0, 2*pi))
sage: g.show(aspect_ratio=1)
```

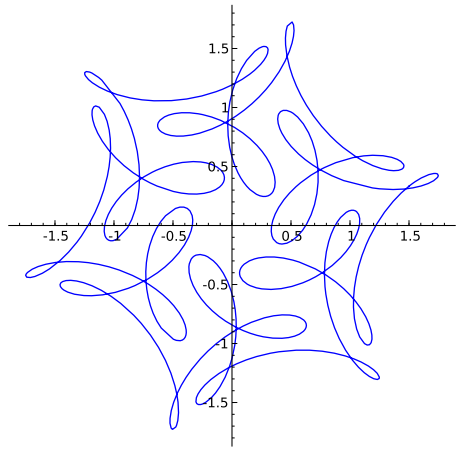


FIGURE 4.4 – Parametric curve of equation $x(t) = \cos(t) + \frac{1}{2}\cos(7t) + \frac{1}{3}\sin(17t)$, $y(t) = \sin(t) + \frac{1}{2}\sin(7t) + \frac{1}{3}\cos(17t)$.

4.1.3 Curve in Polar Coordinates

Curves in polar coordinates $\rho = f(\theta)$, where the parameter θ spans the interval $[a, b]$, may be drawn by the command `polar_plot(rho(theta),(theta,a,b))`.

For example, let us see graphically the rose-curves with polar equation $\rho(\theta) = 1 + e \cdot \cos n\theta$ when $n = 20/19$ and $e \in \{2, 1/3\}$.

```
sage: t = var('t'); n = 20/19
sage: g1 = polar_plot(1+2*cos(n*t),(t,0,n*36*pi),plot_points=5000)
sage: g2 = polar_plot(1+1/3*cos(n*t),(t,0,n*36*pi),plot_points=5000)
```

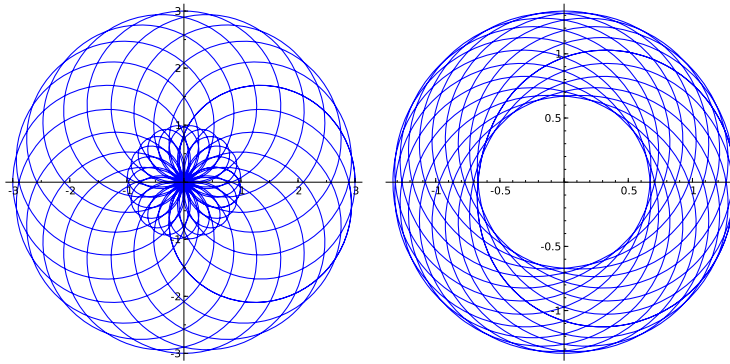


FIGURE 4.5 – Rose-curves of equation $\rho(\theta) = 1 + e \cdot \cos n\theta$.

```
sage: g1.show(aspect_ratio=1); g2.show(aspect_ratio=1)
```

Exercise 12. Draw the family of Pascal conchoids of polar equation $\rho(\theta) = a + \cos\theta$ when the parameter a varies from 0 to 2 by steps of 0.1.

4.1.4 Curve Defined by an Implicit Equation

To draw a curve given by an implicit equation, you need to call the function `implicit_plot(f(x, y), (x, a, b), (y, c, d))`; however, the `complex_plot` command may also be used, which enables us to draw in colour the level set of a two-variable function. Let us draw the curve given by the implicit equation $\mathcal{C} = \left\{ z \in \mathbb{C}\ ,\ \left|\cos(z^4)\right| = 1 \right\}$.

```
sage: z = var('z')
sage: g1 = complex_plot(abs(cos(z^4))-1,
....:                   (-3,3), (-3,3), plot_points=400)
sage: f = lambda x, y : (abs(cos((x + I * y) ** 4)) - 1)
sage: g2 = implicit_plot(f, (-3, 3), (-3, 3), plot_points=400)
sage: g1.show(aspect_ratio=1); g2.show(aspect_ratio=1)
```

4.1.5 Data Plot

To construct a bar graph, two distinct functions are available. On the one hand, `bar_chart` takes as input an integer list and draws vertical bars whose height is given by the list elements (in the given order). The `width` option enables us to choose the bar width.

```
sage: bar_chart([randrange(15) for i in range(20)])
sage: bar_chart([x^2 for x in range(1,20)], width=0.2)
```

On the other hand, to draw the histogram of a random variable from a list of floating-point numbers, we use the `plot_histogram` function. The list values are first sorted and grouped into intervals (the number of intervals is given by the

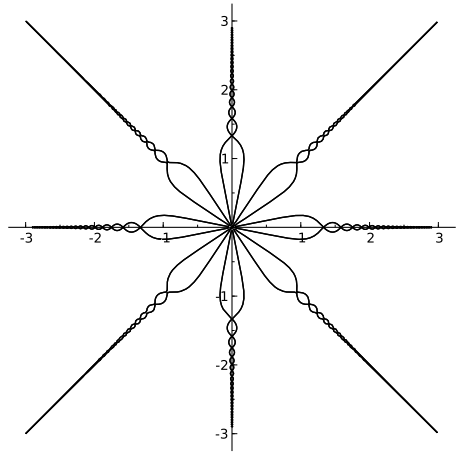


FIGURE 4.6 – Curve `g2` defined by the equation $\left|\cos(z^4)\right| = 1$.

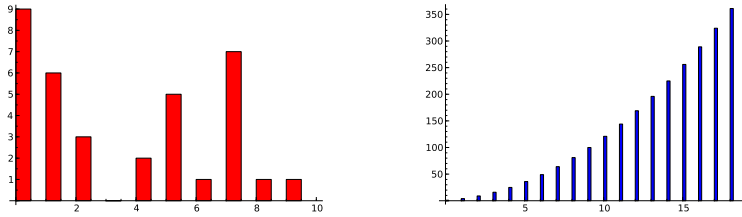
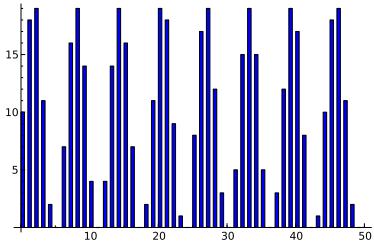


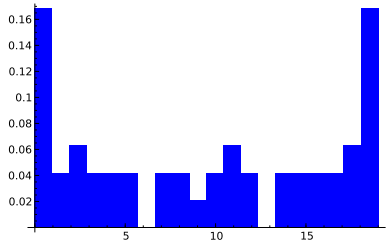
FIGURE 4.7 – Bar graphs.

option `bins` whose default value is 50), the height of each bar being proportional to the number of corresponding values.

```
sage: liste = [10 + floor(10*sin(i)) for i in range(100)]
sage: bar_chart(liste)
sage: finance.TimeSeries(liste).plot_histogram(bins=20)
```



(a) Plot with `bar_chart`.

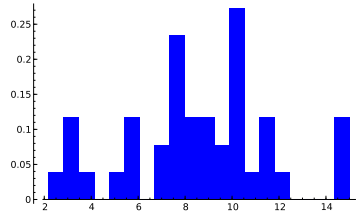


(b) Plot with `plot_histogram`.

It often arises that the list of data values we want to study is stored in a spreadsheet format. The Python `csv` package enables us to import such data

stored in the `csv` format. For example, let us assume that we want to plot the histogram of grades of a school class, which are in column 3 of the file `exam01.csv`. To extract the grades from this column, we will use the following instructions (in general, the first lines of such a file contain text, therefore we deal with potential non-matching lines with the `try` keyword):

```
sage: import csv
sage: reader = csv.reader(open("exam01.csv"))
sage: grades = []; list = []
sage: for line in reader:
....:     grades.append(line[2])
....: for i in grades:
....:     try:
....:         f = float(i)
....:     except ValueError:
....:         pass
....:     else:
....:         list.append(f)
sage: finance.TimeSeries(list).plot_histogram(bins=20)
```



To draw a list of linked points (resp. non-linked), we use the `line(p)` (resp. `point(p)` or `points(p)`) command, `p` being a list of 2-element lists (or tuples) giving abscissa and ordinate of the points.

EXAMPLE. (*Random walk*) Starting from the origin O, a particle moves a distance ℓ every t seconds, in a random direction, independently of the preceding moves. Let us draw an example of particle trajectory. The red line goes from the initial to the final position.

```
sage: n, l, x, y = 10000, 1, 0, 0; p = [[0, 0]]
sage: for k in range(n):
....:     theta = (2 * pi * random()).n(digits=5)
....:     x, y = x + l * cos(theta), y + l * sin(theta)
....:     p.append([x, y])
sage: g1 = line([p[n], [0, 0]], color='red', thickness=2)
sage: g1 += line(p, thickness=.4); g1.show(aspect_ratio=1)
```

EXAMPLE. (*Uniformly distributed sequences*) Given a real sequence $(u_n)_{n\in\mathbb{N}^*}$, we construct the polygonal line whose successive vertices are the points in the complex plane

$$z_N = \sum_{n\leq N} e^{2i\pi u_n}.$$

If the sequence is uniformly distributed modulo 1, the polygonal line should behave like a random walk, and thus not go too far from the origin. Hence we can conjecture the uniform distribution modulo 1 from the graphical aspect of the polygonal line. Let us study the following cases:

- $u_n = n\sqrt{2}$ and $N = 200$,
- $u_n = n\ln(n)\sqrt{2}$ and $N = 10000$,

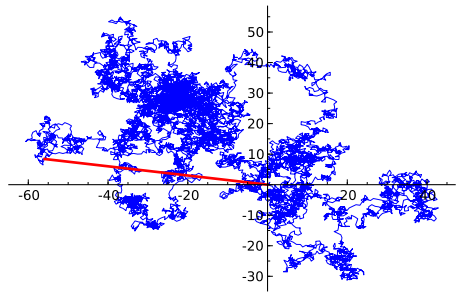


FIGURE 4.8 – Random walk.

- $u_n = \lfloor n \ln(n) \rfloor \sqrt{2}$ and $N = 10000$,
- $u_n = p_n \sqrt{2}$ and $N = 10000$ (here p_n is the n-th prime).

Figure 4.9 is obtained as follows (here for $u_n = n\sqrt{2}$):

```
sage: length = 200; n = var('n')
sage: u = lambda n: n * sqrt(2)
sage: z = lambda n: exp(2 * I * pi * u(n)).n()
sage: vertices = [CC(0, 0)]
sage: for n in range(1, length):
....:     vertices.append(vertices[n - 1] + CC(z(n)))
sage: line(vertices).show(aspect_ratio=1)
```

We see that the curve 4.9a is amazingly regular, which suggests that the uniform distribution of $n\sqrt{2}$ modulo 1 is deterministic. In the case of $u_n = n \ln(n)\sqrt{2}$, the values apparently seem random modulo 1. However, the associated curve 4.9b is remarkably well structured. The curve 4.9c has the same kind of structure as the second one. Finally, the curve 4.9d shows the completely different nature of primes modulo $1/\sqrt{2}$: the spirals have disappeared and the aspect looks very similar to a random walk u_n (Figure 4.8). It thus looks as though "prime numbers make use of all the randomness they are given..."

For a detailed interpretation of these curves, we refer the reader to the book (in French) *Les nombres premiers* of Gérald Tenenbaum and Michel Mendès France [TMF00].

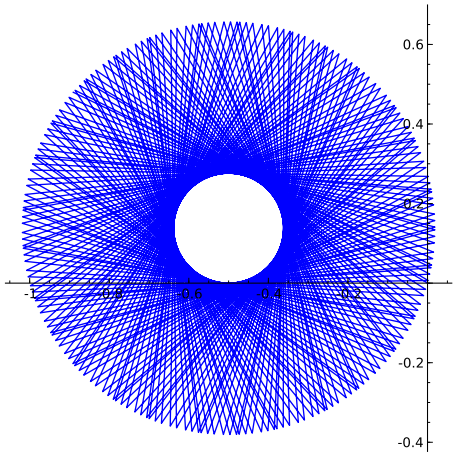
Exercise 13 (Drawing terms of a recurrent sequence)**.** We consider the sequence $(u_n)_{n \in \mathbb{N}}$ defined by:

$$\begin{cases} u_0 = a, \\ \forall n \in \mathbb{N},\ u_{n+1} = \left| u_n^2 - \frac{1}{4} \right|. \end{cases}$$

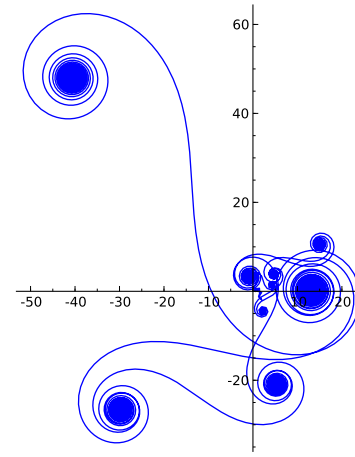
Represent graphically the behaviour of the sequence by constructing a list of points $\big[[u_0, 0], [u_0, u_1], [u_1, u_1], [u_1, u_2], [u_2, u_2], \dots\big]$, with $a \in \{-0.4, 1.1, 1.3\}$.

4.1.6 Displaying Solutions of Differential Equations

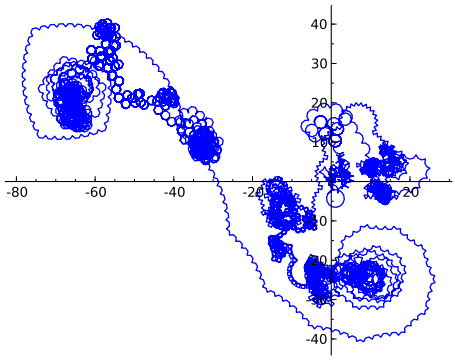
We can combine the above commands to represent solutions of differential equations or systems. To solve symbolically an ordinary differential equation, one calls



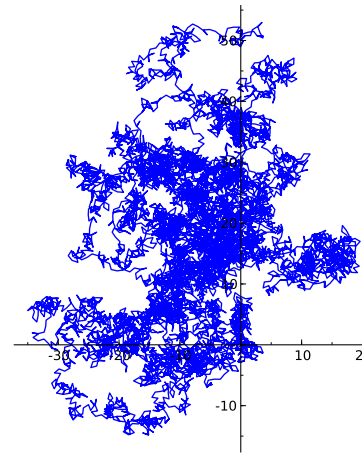
(a) Case $u_n = n\sqrt{2}$.



(b) Case $u_n = n\ln(n)\sqrt{2}$.



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



(d) Case $u_n = p_n\sqrt{2}$.

FIGURE 4.9 – Uniformly distributed sequences.

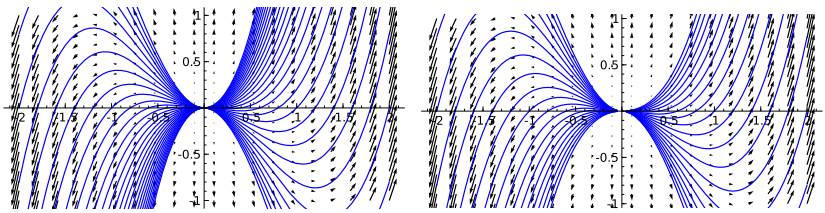
the `desolve` function, which is studied in more detail in Chapter 10. To solve a differential equation numerically, Sage provides several tools: `desolve_rk4` (which uses the same syntax as `desolve`, and which is enough to solve differential equations at undergraduate level), `odeint` (which calls the SciPy package), and finally `ode_solver` (which calls the GSL library, and whose use is detailed in Section 14.2). The functions `desolve_rk4` and `odeint` return a list of points, which is easy to draw using the `line` command; we will use them in this section to draw numerical solutions.

EXAMPLE. (*First-order linear differential equation*) Let us draw the integral curves of the differential equation $xy' - 2y = x^3$.

```
sage: x = var('x'); y = function('y')
sage: DE = x*diff(y(x), x) == 2*y(x) + x^3
sage: desolve(DE, [y(x),x])
(_C + x)*x^2
sage: sol = []
sage: for i in srange(-2, 2, 0.2):
....:     sol.append(desolve(DE, [y(x), x], ics=[1, i]))
....:     sol.append(desolve(DE, [y(x), x], ics=[-1, i]))
sage: g = plot(sol, x, -2, 2)
sage: y = var('y')
sage: g += plot_vector_field((x, 2*y+x^3), (x,-2,2), (y,-1,1))
sage: g.show(ymin=-1, ymax=1)
```

To decrease the computation time, it would be better here to define "by hand" the general solution of the equation, and to create a list of particular solutions (as done in the solution of Exercise 14), instead of solving the differential equation several times with different initial conditions. We could also compute a numerical solution of this equation (with the `desolve_rk4` function) to draw its integral curves:

```
sage: x = var('x'); y = function('y')
sage: DE = x*diff(y(x), x) == 2*y(x) + x^3
sage: g = Graphics()                # creates an empty graph
sage: for i in srange(-2, 2, 0.2):
....:     g += line(desolve_rk4(DE, y(x), ics=[1, i],\
```



(a) Symbolic solution. (b) Numerical solution.

FIGURE 4.10 – Integral curves of $xy' - 2y = x^3$.

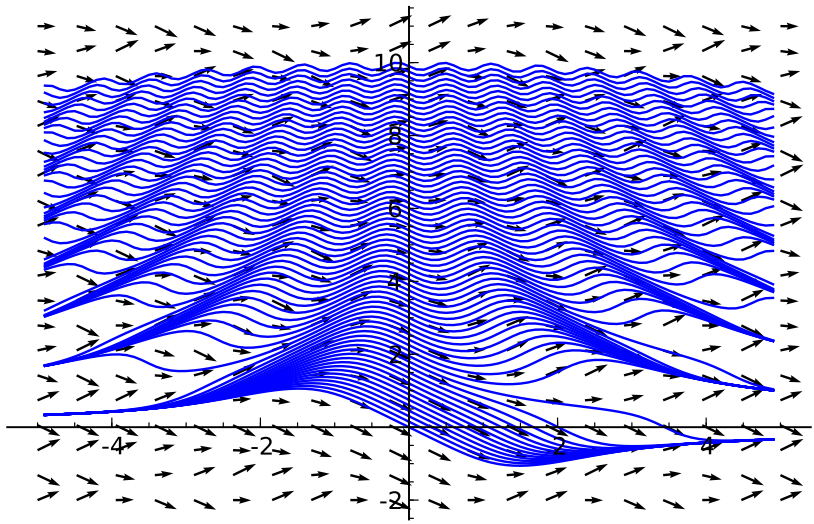


FIGURE 4.11 – Integral curves of $y'(t) + \cos(y(t) \cdot t) = 0$.

```
....:                           step=0.05, end_points=[0,2]))
....:     g += line(desolve_rk4(DE, y(x), ics=[-1, i],\
....:                           step=0.05, end_points=[-2,0]))
sage: y = var('y')
sage: g += plot_vector_field((x, 2*y+x^3), (x,-2,2), (y,-1,1))
sage: g.show(ymin=-1, ymax=1)
```

As seen in the above example, the `desolve_rk4` function takes as input a differential equation (or the right-hand side f of the equation in explicit form $y' = f(y, x)$), the name of the unknown function, the initial conditions, the step and interval where a solution is sought. The optional argument `output` enables us to specify the type of output: the default value `'list'` returns a list (which is useful if we want to combine graphics as in our example), `'plot'` outputs the graph of the solution, and `'slope_field'` adds the graphs of slopes of integral curves.

Exercise 14. Draw the integral curves of the equation $x^2y' - y = 0$, for $-3 \leq x \leq 3$ and $-5 \leq y \leq 5$.

Let us now give an example of the `odeint` function from the SciPy package.

EXAMPLE. (*First-order non-linear differential equation*) Let us draw the integral curves of the equation $y'(t) + \cos(y(t) \cdot t) = 0$.

```
sage: import scipy; from scipy import integrate
sage: f = lambda y, t: - cos(y * t)
sage: t = srange(0, 5, 0.1); p = Graphics()
sage: for k in srange(0, 10, 0.15):
....:       y = integrate.odeint(f, k, t)
....:       p += line(zip(t, flatten(y)))
```

```
sage: t = srange(0, -5, -0.1); q = Graphics()
sage: for k in srange(0, 10, 0.15):
....:       y = integrate.odeint(f, k, t)
....:       q += line(zip(t, flatten(y)))
sage: y = var('y')
sage: v = plot_vector_field((1, -cos(x*y)), (x,-5,5), (y,-2,11))
sage: g = p + q + v; g.show()
```

The `odeint` function takes as argument the right-hand side f of the differential equation $y' = f$ (assumed to be in explicit form), one or more initial conditions, and the interval where a solution is sought; it returns an array of type `numpy.ndarray` that one converts using the `flatten` command[1] already seen in §3.3.2. The obtained list is then combined with the array `t` using the `zip` command, and the approximate solution is displayed. To add the vector fields tangent to the integral curves, we have used the `plot_vector_field` command.

EXAMPLE. (*Lotka-Volterra predator-prey model*) We wish to represent graphically the variation of a set of prey and predators evolving according to a system of Lotka-Volterra equations:

$$\begin{cases} \dfrac{du}{dt} = au - buv, \\ \dfrac{dv}{dt} = -cv + dbuv, \end{cases}$$

where u is the number of preys (for example rabbits), v is the number of predators (for example foxes). In addition, the parameters a, b, c, d describe the evolution of the populations: a is the natural growth of rabbits without foxes to eat them, b is the decrease of rabbits when foxes kill them, c is the decrease of foxes without any rabbit to eat, and finally d indicates how many rabbits are needed for a new fox to appear.

```
sage: import scipy; from scipy import integrate
sage: a, b, c, d = 1., 0.1, 1.5, 0.75
sage: def dX_dt(X, t=0):                # returns the population variation
....:     return [a*X[0] - b*X[0]*X[1], -c*X[1] + d*b*X[0]*X[1]]
sage: t = srange(0, 15, .01)                                  # time scale
sage: X0 = [10, 5]         # initial conditions: 10 rabbits and 5 foxes
sage: X = integrate.odeint(dX_dt, X0, t)            # numerical solution
sage: rabbits, foxes =  X.T                 # shortcut for X.transpose()
sage: p = line(zip(t, rabbits), color='red')  # number of rabbits graph
sage: p += text("Rabbits",(12,37), fontsize=10, color='red')
sage: p += line(zip(t, foxes), color='blue')            # idem for foxes
sage: p += text("Foxes",(12,7), fontsize=10, color='blue')
sage: p.axes_labels(["time", "population"]); p.show(gridlines=True)
```

The instructions above show the evolution of the number of rabbits and foxes with time (Figure 4.12, left), and those below the vector field (Figure 4.12, right):

[1] We could also use the NumPy `ravel` function, which avoids creating a new object, and thus optimises the memory usage.

```
sage: n = 11;  L = srange(6, 18, 12 / n); R = srange(3, 9, 6 / n)
sage: CI = zip(L, R)                    # list of initial conditions
sage: def g(x,y):
....:     v = vector(dX_dt([x, y]))  # for a nicer graph, we
....:     return v/v.norm()          # normalise the vector field
sage: x, y = var('x, y')
sage: q = plot_vector_field(g(x, y), (x, 0, 60), (y, 0, 36))
sage: for j in range(n):
....:     X = integrate.odeint(dX_dt, CI[j], t)        # resolution
....:     q += line(X, color=hue(.8-float(j)/(1.8*n))) # graph plot
sage: q.axes_labels(["rabbits","foxes"]); q.show()
```

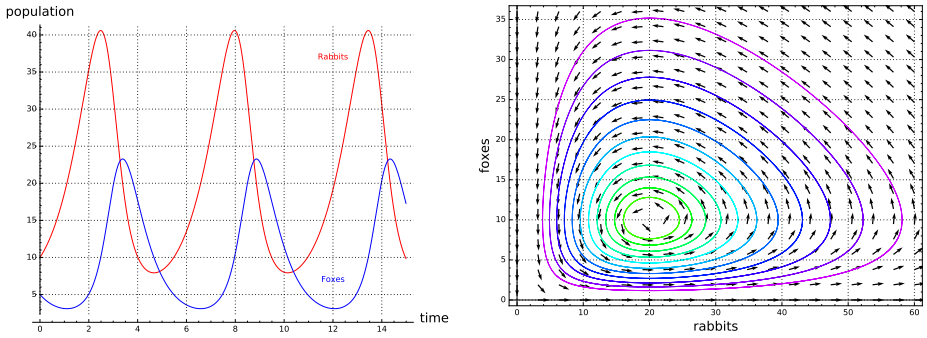


FIGURE 4.12 – Study of a predator-prey system.

Exercise 15 (Predator-prey model)**.** Recreate the left-hand graph of Figure 4.12 using `desolve_system_rk4` instead of `odeint`.

Exercise 16 (An autonomous differential system)**.** Draw the integral curves of the following differential system:

$$\begin{cases} \dot{x} = y, \\ \dot{y} = 0.5y - x - y^3. \end{cases}$$

Exercise 17 (Flow around a cylinder with Magnus effect)**.** We combine a simple flow around a cylinder of radius a to a vortex of parameter α, which modifies the orthoradial velocity component. We work in a coordinate system centered on the cylinder, with cylindrical coordinates in the plane $z = 0$, i.e., in polar coordinates. The velocity components are then:

$$v_r = v_0 \cos(\theta) \left(1 - \frac{a^2}{r^2}\right) \quad \text{and} \quad v_\theta = -v_0 \sin(\theta) \left(1 + \frac{a^2}{r^2}\right) + 2\frac{\alpha a v_0}{r}.$$

The flow lines (which are identical to trajectories, since the flow is stationary) are parallel to the velocity. We search a parametric expression of the flow lines; we have thus to solve the differential system:

$$\frac{dr}{dt} = v_r \quad \text{and} \quad \frac{d\theta}{dt} = \frac{v_\theta}{r}.$$

By using coordinates scaled by the radius a of the cylinder, we may assume $a = 1$. Draw the flow lines for $\alpha \in \{0.1, 0.5, 1, 1.25\}$.

The Magnus effect was proposed to build propulsion systems made from large vertical rotating cylinders able to produce a longitudinal thrust when the wind is perpendicular to the ship (this was the case of the Baden-Baden rotor ship built by Anton Flettner, which crossed the Atlantic in 1926).

4.1.7 Evolute of a Curve

We now give an example of drawing the evolute of a parametric arc (let us recall that the evolute is the envelope of the normals of a curve, or equivalently, the locus of centres of curvature).

EXAMPLE. (*Evolute of the parabola*) Let us find the equation of the evolute of the parabola $\mathcal{P}$ of equation $y = x^2/4$, and show on the same graph the parabola $\mathcal{P}$, some normals to $\mathcal{P}$ and its evolute.

To determine a system of parametric equations $(x(t), y(t))$ of the evolute of a family of lines Δ_t defined by cartesian equations of the form $\alpha(t)X + \beta(t)Y = \gamma(t)$, we express the fact that the line Δ_t is tangent to the envelope at $(x(t), y(t))$:

$$\alpha(t)x(t) + \beta(t)y(t) = \gamma(t), \tag{4.1}$$
$$\alpha(t)x'(t) + \beta(t)y'(t) = 0. \tag{4.2}$$

The derivative of Equation (4.1), combined with (4.2), yields the system:

$$\alpha(t)\ x(t) + \beta(t)\ y(t) = \gamma(t), \tag{4.1}$$
$$\alpha'(t)x(t) + \beta'(t)y(t) = \gamma'(t). \tag{4.3}$$

In our case, the normal (N_t) to the parabola $\mathcal{P}$ in $M(t, t^2/4)$ has normal vector $\overrightarrow{v} = (1, t/2)$ (which is tangent to the parabola); it thus has for equation:

$$\begin{pmatrix} x - t \\ y - t^2/4 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ t/2 \end{pmatrix} = 0 \iff x + \frac{t}{2}y = t + \frac{t^3}{8},$$

in other words, $(\alpha(t), \beta(t), \gamma(t)) = (1, t/2, t + t^3/8)$. We can then solve the preceding system with the `solve` function:

```
sage: x, y, t = var('x, y, t')
sage: alpha(t) = 1; beta(t) = t / 2; gamma(t) = t + t^3 / 8
sage: env = solve([alpha(t) * x + beta(t) * y == gamma(t),\
....:     diff(alpha(t), t) * x + diff(beta(t), t) * y == \
....:     diff(gamma(t), t)], [x,y])
```

$$\left[\left[x = -\frac{1}{4}\,t^3, y = \frac{3}{4}\,t^2 + 2\right]\right]$$

This gives a parametric representation of the normal envelope:

$$\begin{cases} x(t) = -\frac{1}{4}t^3, \\ y(t) = 2 + \frac{3}{4}t^2. \end{cases}$$

We can then answer the given question, by drawing some normals to the parabola (more precisely, we draw line segment $[M, M + 18\overrightarrow{n}]$ where $M(u, u^2/4)$ is a point on $\mathcal{P}$ and $\overrightarrow{n} = (-u/2, 1)$ a normal vector to $\mathcal{P}$):

```
sage: f(x) = x^2 / 4
sage: p = plot(f, -8, 8, rgbcolor=(0.2,0.2,0.4))  # the parabola
sage: for u in srange(0, 8, 0.1):      # normals to the parabola
....:     p += line([[u, f(u)], [-8*u, f(u) + 18]], thickness=.3)
....:     p += line([[-u, f(u)], [8*u, f(u) + 18]], thickness=.3)
sage: p += parametric_plot((env[0][0].rhs(),env[0][1].rhs()),\
....:     (t, -8, 8),color='red')              # draws the evolute
sage: p.show(xmin=-8, xmax=8, ymin=-1, ymax=12, aspect_ratio=1)
```

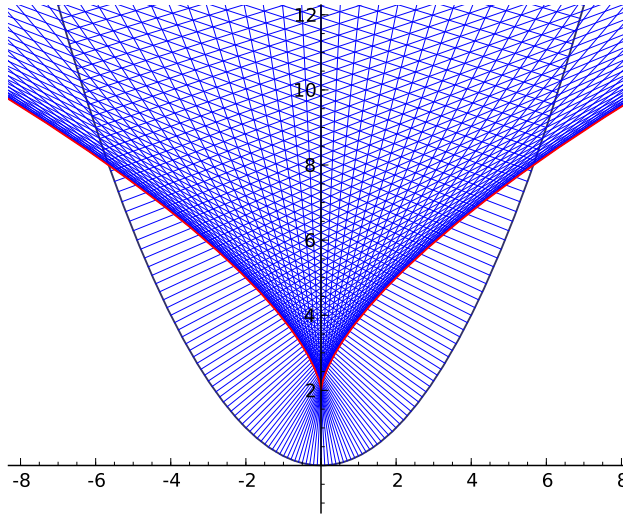


FIGURE 4.13 – The parabola evolute.

As recalled above, the evolute of a curve is also the locus of its centres of curvature. Using the `circle` function, let us draw some osculating circles of the parabola. The centre of curvature Ω at a point $M_t = (x(t), y(t))$ of the curve has coordinates:

$$x_\Omega = x + -y' \frac{x'^2 + y'^2}{x'y'' - x''y'}, \qquad \text{and} \qquad y_\Omega = y + x' \frac{x'^2 + y'^2}{x'y'' - x''y'},$$

and the radius of curvature[2] at M_t is:

$$R = \frac{(x'^2 + y'^2)^{\frac{3}{2}}}{x'y'' - x''y'}.$$

```
sage: t = var('t'); p = 2
sage: x(t) = t; y(t) = t^2 / (2 * p); f(t) = [x(t), y(t)]
sage: df(t) = [x(t).diff(t), y(t).diff(t)]
sage: d2f(t) = [x(t).diff(t, 2), y(t).diff(t, 2)]
```

[2] We consider here the algebraic radius of curvature, which can be negative.

Type of drawing	
Graph of a function	`plot`
Parametric curve	`parametric_plot`
Curve defined by a polar equation	`polar_plot`
Curve defined by an implicit equation	`implicit_plot`
Level set of a complex function	`complex_plot`
Empty graphical object	`Graphics()`
Integral curves of a differential equation	`odeint, desolve_rk4`
Bar graph, bar chart	`bar_chart`
Histogram of a statistical sequence	`plot_histogram`
Polygonal chain	`line`
Cloud of points	`points`
Circle	`circle`
Polygon	`polygon`
Text	`text`

TABLE 4.1 – Summary of 2D graphical functions.

```
sage: T(t) = [df(t)[0] / df(t).norm(), df[1](t) / df(t).norm()]
sage: N(t) = [-df(t)[1] / df(t).norm(), df[0](t) / df(t).norm()]
sage: R(t) = (df(t).norm())^3 / (df(t)[0]*d2f(t)[1]-df(t)[1]*d2f(t)[0])
sage: Omega(t) = [f(t)[0] + R(t)*N(t)[0], f(t)[1] + R(t)*N(t)[1]]
sage: g = parametric_plot(f(t), (t,-8,8), color='green',thickness=2)
sage: for u in srange(.4, 4, .2):
....:     g += line([f(t=u), Omega(t=u)], color='red', alpha = .5)
....:     g += circle(Omega(t=u), R(t=u), color='blue')
sage: g.show(aspect_ratio=1,xmin=-12,xmax=7,ymin=-3,ymax=12)
```

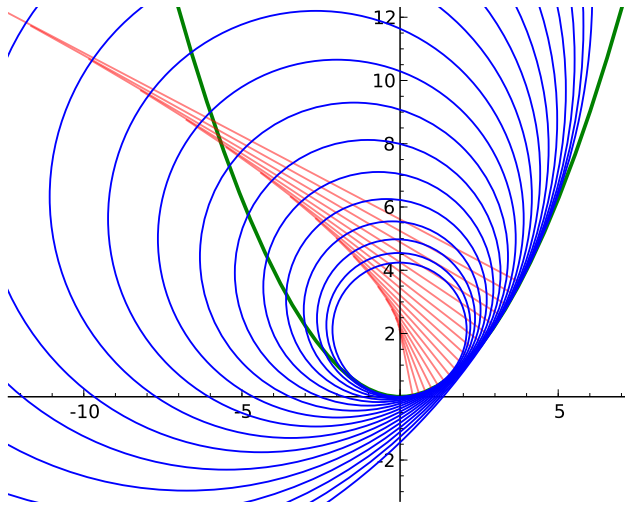


FIGURE 4.14 – Osculating circles of the parabola.

Table 4.1 gives a summary of the functions detailed in this section. It also contains the `text` command which enables us to add a character string in a graph, and the `polygon` command to plot polygons.

4.2 3D Curves

Sage provides the `plot3d(f(x,y),(x,a,b),(y,c,d))` command to display surfaces in 3-dimensions. The surface obtained may then be visualised via the *Jmol* application; the *Tachyon 3D Ray Tracer* or *three.js* can be used alternatively with the option `viewer='tachyon'` or `viewer='threejs'` of the `show` command. Here is a first example of parametric surface (Figure 4.15):

```
sage: u, v = var('u, v')
sage: h = lambda u,v: u^2 + 2*v^2
sage: plot3d(h, (u,-1,1), (v,-1,1), aspect_ratio=[1,1,1])
```

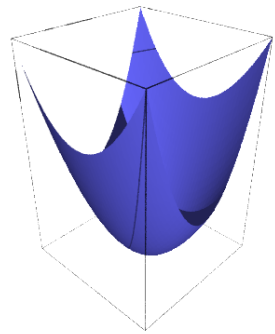


FIGURE 4.15 – The parametric surface $(u, v) \mapsto u^2 + 2v^2$.

Displaying the surface corresponding to a 2-variable function helps us to study that function, as will be seen in the following example.

EXAMPLE. (*A discontinuous function whose directional derivatives exist everywhere!*) Study the existence in $(0, 0)$ of the directional derivatives and the continuity of the function f from $\mathbb{R}^2$ to $\mathbb{R}$ defined by:

$$f(x,y) = \begin{cases} \frac{x^2 y}{x^4+y^2} & \text{if } (x,y) \neq (0,0), \\ 0 & \text{if } (x,y) = (0,0). \end{cases}$$

For $H = \begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix}$, the function $\varphi(t) = f(tH) = f(t\cos\theta, t\sin\theta)$ is differentiable in $t = 0$ for any value of θ; indeed,

```
sage: f(x, y) = x^2 * y / (x^4 + y^2)
sage: t, theta = var('t, theta')
sage: limit(f(t * cos(theta), t * sin(theta)) / t, t=0)
cos(theta)^2/sin(theta)
```

Hence f has well-defined directional derivatives in any direction at the point $(0,0)$. To better visualise the surface corresponding to f, we can first look for some level sets; for example the level set of value $\frac{1}{2}$:

```
sage: solve(f(x,y) == 1/2, y)
[y == x^2]
sage: a = var('a'); h = f(x, a*x^2).simplify_rational(); h
a/(a^2 + 1)
```

Along the parabola of equation $y = ax^2$, except at the origin, f has thus a constant value $f(x, ax^2) = \frac{a}{1+a^2}$. We then display the function $h\colon a \longmapsto \frac{a}{1+a^2}$:

```
sage: plot(h, a, -4, 4)
```

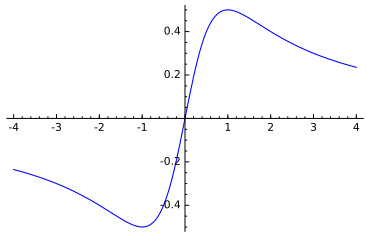


FIGURE 4.16 – A vertical cut of the surface under study.

The function h has its maximum at $a = 1$ and its minimum at $a = -1$. The restriction of f to the parabola of equation $y = x^2$ corresponds to the level set at "height" $\frac{1}{2}$; conversely, the restriction to the parabola of equation $y = -x^2$ corresponds to the bottom of the "thalweg" at height $-\frac{1}{2}$. In conclusion, arbitrarily close to the point $(0,0)$, we can find points where f takes as value $\frac{1}{2}$, or respectively $-\frac{1}{2}$. As a consequence, f is not continuous at the origin.

```
sage: p = plot3d(f(x,y),(x,-2,2),(y,-2,2),plot_points=[150,150])
```

We might also draw horizontal planes to display the level sets of this function with:

```
sage: for i in range(1,4):
....:  p += plot3d(-0.5 + i / 4, (x, -2, 2), (y, -2, 2),\
....:              color=hue(i / 10), opacity=.1)
```

Among the other 3D graphical commands, `implicit_plot3d` allows us to display surfaces defined by an implicit equation of the form $f(x,y,z) = 0$. Let us display for example the Cassini surface (Figure 4.18a) defined by the implicit equation: $\left(a^2 + x^2 + y^2\right)^2 = 4\,a^2x^2 + z^4$.

```
sage: x, y, z = var('x, y, z'); a = 1
sage: h = lambda x, y, z:(a^2 + x^2 + y^2)^2 - 4*a^2*x^2-z^4
sage: implicit_plot3d(h, (x,-3,3), (y,-3,3), (z,-2,2),\
....:                 plot_points=100)
```

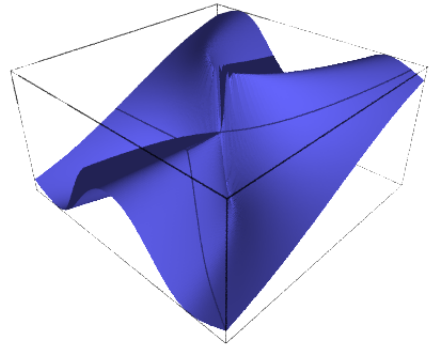
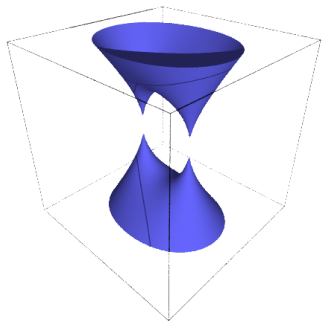


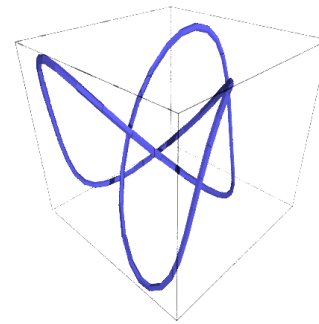
FIGURE 4.17 – The surface corresponding to $f\colon (x,y) \mapsto \frac{x^2 y}{x^4+y^2}$.

Finally, let us give an example of 3-dimensional curve (Figure 4.18b) with the `line3d` command:

```
sage: line3d([(-10*cos(t)-2*cos(5*t)+15*sin(2*t),\
....:          -15*cos(2*t)+10*sin(t)-2*sin(5*t),\
....:          10*cos(3*t)) for t in srange(0,6.4,.1)],radius=.5)
```



(a) The Cassini surface.



(b) A knot in space.

FIGURE 4.18 – Surface and curve in 3D.

5 Computational Domains

Writing mathematics on paper or on the blackboard requires a compromise between ease of notations and rigour. The same holds for the day-to-day use of a computer algebra system. Sage tries to give this choice to the user, by letting her/him specify, more or less rigorously, the computational domains: what is the nature of the considered objects, in which sets do they live, which operations can be applied to them?

5.1 Sage is Object-Oriented

Python and Sage use heavily the object-oriented programming paradigm. Even though this remains relatively transparent in common use, it is useful to know a little about this paradigm, which is quite natural in a mathematical context.

5.1.1 Objects, Classes and Methods

The object-oriented programming paradigm consists in modelling each physical or abstract entity one wishes to manipulate by a programming language construction called an *object*. In most cases, as in Python, each object is an instance of a *class*. For example, the rational number 12/35 is represented by an object which is an instance of the `Rational` class:

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

Note that this class is really associated to the object 12/35, and not to the variable `o` in which it is stored:

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

Let us be more precise. An *object* is a part of the computer memory which stores the required information to represent the corresponding entity. The *class* in turn defines two things:

1. the *data structure* of an object, i.e., how the information is organised in memory. For example, the `Rational` class specifies that a rational number like 12/35 is represented by two integers: its numerator and its denominator;

2. its *behaviour*, in particular the available *operations* on this object: how to obtain the numerator of a rational number, how to compute its absolute value, how to multiply or add two rational numbers. Each of these operations is implemented by a *method* (here respectively `numer`, `abs`, `__mul__`, `__add__`).

To factor an integer, we will thus call the `factor` method with the following syntax:

```
sage: o = 720
sage: o.factor()
2^4 * 3^2 * 5
```

which we can read as follows: "take the value of `o` and apply to it the `factor` method, without any other argument". Under the hood, Python performs the following computation:

```
sage: type(o).factor(o)
2^4 * 3^2 * 5
```

From left to right: "request from the class of `o` (`type(o)`) the factorisation method (`type(o).factor`), and apply it to `o`".

Please note that we can apply this method not only to a variable, but also directly to a value:

```
sage: 720.factor()
2^4 * 3^2 * 5
```

and thus we can chain the operations, from left to right. Here, we first take the numerator of a rational number, then we factor this numerator:

```
sage: o = 720 / 133
sage: o.numerator().factor()
2^4 * 3^2 * 5
```

To make the user's life easier, Sage also provides a *function* `factor`, so that `factor(o)` is a shortcut for `o.factor()`. It is the case for several common functions, and it is possible to add our own shortcuts, as illustrated in the following exercise.

Exercise 18. Build a shortcut `ndigits` so that `ndigits(o)` calls the `ndigits` method of the object `o`.

5.1.2 Objects and Polymorphism

Almost all Sage operations are *polymorphic*, i.e., they apply to several kinds of objects. For example, whatever the nature of the object `o` that we want to "factor", we will use the same notation `o.factor()` (or its shortcut `factor(o)`). The computations to be performed however differ to factor an integer or a polynomial! They also differ if the polynomial has rational coefficients, or coefficients in a finite field. The object class determines the version of the `factor` code that will be called.

Similarly, like the usual mathematical notation, the product of two objects `a` and `b` can always be denoted `a*b`, even if the algorithm used differs in each case[1]. Here is a product of two integers:

```
sage: 3 * 7
21
```

a product of two rational numbers, obtained by multiplying the numerators and denominators, then reducing the fraction:

```
sage: (2/3) * (6/5)
4/5
```

a product of two complex numbers, using the relation $i^2 = -1$:

```
sage: (1 + I)  *  (1 - I)
2
```

some commutative products of two formal expressions:

```
sage: (x + 2) * (x + 1)
(x + 2)*(x + 1)
sage: (x + 1) * (x + 2)
(x + 2)*(x + 1)
```

Apart from the notation simplicity, this form of polymorphism enables us to write *generic* programs which apply to any object having the involved operations (here multiplication):

```
sage: def fourth_power(a):
....:     a = a * a
....:     a = a * a
....:     return a
```

```
sage: fourth_power(2)
16
sage: fourth_power(3/2)
81/16
```

[1] For a binary operation like the product, the selection of the appropriate method is slightly more complex than what was described above. Indeed, we might deal with mixed operations like the sum $2 + 3/4$ of an integer and of a rational number. In this case, 2 will be converted in the rational $2/1$, and the addition of two rationals will be called. The rules that describe which operand must be converted, and how it should be converted, are part of the *coercion model*.

```
sage: fourth_power(I)
1
sage: fourth_power(x+1)
(x + 1)^4
sage: M = matrix([[0,-1],[1,0]]); M
[ 0 -1]
[ 1  0]
sage: fourth_power(M)
[1 0]
[0 1]
```

5.1.3 Introspection

Python objects, and therefore Sage objects, have some *introspection* features. This means that, during execution, we can "ask" an object for its class, its methods, etc., and manipulate the obtained informations using the usual constructions of the programming language. For instance, the class of an object `o` is itself a Python object, and we can obtain it using `type(o)`:

```
sage: t = type(5/1); t
<type 'sage.rings.rational.Rational'>
sage: t == type(5)
False
```

We see here that the expression `5/1` constructs the rational number 5, which differs — as Python object — from the integer 5!

The introspection tools also give access to the factorisation on-line help from an object of integer type:

```
sage: o = 720
sage: o.factor?
Docstring:
     Return the prime factorization of this integer as a formal
     Factorization object.
...
```

and even to the source code of that function:

```
sage: o.factor??
...
def factor(self, algorithm='pari', proof=None, ...)
      ...
      if algorithm == 'pari':
          ...
      elif algorithm in ['kash', 'magma']:
          ...
```

Avoiding some technical details, we see here that Sage delegates the integer factorisation to other tools (PARI/GP, Kash, or Magma).

In the same vein, we can use automatic completion to interactively "ask" an object `o` which operations can be applied to it:

```
sage: o.n<tab>
o.n                    o.nbits                o.ndigits
o.next_prime           o.next_prime_power     o.next_probable_prime
o.nth_root             o.numerator            o.numerical_approx
```

Once again, it is a form of introspection.

5.2 Elements, Parents, Categories

5.2.1 Elements and Parents

In the preceding section, we have seen the concept of *class* of an object. In practice, it is enough to know that this notion exists; we rarely have to explicitly look for the type of an object. However, Sage introduces another concept closer to mathematics: the *parent* of an object, that we will detail now.

Assume for example that we want to know if an element a is *invertible*. The answer does not only depend on the element itself, but also on the mathematical set A it belongs to (and its potential inverse). For example, the number 5 is not invertible in the set $\mathbb{Z}$ of integers, since its inverse $1/5$ is not an integer:

```
sage: a = 5; a
5
sage: a.is_unit()
False
```

However, it is invertible in the set of rational numbers:

```
sage: a = 5/1; a
5
sage: a.is_unit()
True
```

Sage gives two different answers to that question since, as seen in the above section, the objects 5 and $5/1$ have different classes.

In some object-oriented computer algebra systems, like MuPAD or Axiom, the mathematical set X to which x belongs (here $\mathbb{Z}$ or $\mathbb{Q}$) is simply the class of x. Sage follows the approach of the Magma system, and defines the set X by another object attached to x, called its *parent*:

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

We can obtain these two sets with the following shortcuts:

```
sage: ZZ
Integer Ring
```

```
sage: QQ
Rational Field
```

and use them to easily *convert* an element from one set to the other, when it makes sense:

```
sage: QQ(5).parent()
Rational Field
sage: ZZ(5/1).parent()
Integer Ring
sage: ZZ(1/5)
Traceback (most recent call last):
  ...
TypeError: no conversion of this rational to integer
```

More generally, the `P(x)` syntax — where `P` is a parent — tries to convert the object `x` into an element of `P`. We show four different instances of 1: as integer $1 \in \mathbb{Z}$, as rational number $1 \in \mathbb{Q}$, as real floating-point $1.0 \in \mathbb{R}$ or complex floating-point $1.0 + 0.0i \in \mathbb{C}$:

```
sage: ZZ(1), QQ(1), RR(1), CC(1)
(1, 1, 1.00000000000000, 1.00000000000000)
```

Exercise 19. Find two Sage objects having the same type and different parents. Then find two Sage objects having the same parent and different types.

5.2.2 Constructions

The parents being themselves first-class objects, we can apply operations to them. For example, one can construct the cartesian product $\mathbb{Q}^2$:

```
sage: cartesian_product([QQ, QQ])
The Cartesian product of (Rational Field, Rational Field)
```

find $\mathbb{Q}$ as the fraction field of $\mathbb{Z}$:

```
sage: ZZ.fraction_field()
Rational Field
```

construct the ring of polynomials in x with coefficients in $\mathbb{Z}$:

```
sage: ZZ['x']
Univariate Polynomial Ring in x over Integer Ring
```

Using an incremental approach, we can construct complex algebraic structures like the 3×3 matrix space with polynomial coefficients on a finite field:

```
sage: Z5 = GF(5); Z5
Finite Field of size 5
sage: P = Z5['x']; P
Univariate Polynomial Ring in x over Finite Field of size 5
sage: M = MatrixSpace(P, 3, 3); M
```

```
Full MatrixSpace of 3 by 3 dense matrices over
Univariate Polynomial Ring in x over Finite Field of size 5
```

and draw a random element from this domain:

```
sage: M.random_element()
[2*x^2 + 3*x + 4 4*x^2 + 2*x + 2     4*x^2 + 2*x]
[           3*x   2*x^2 + x + 3     3*x^2 + 4*x]
[     4*x^2 + 3 3*x^2 + 2*x + 4         2*x + 4]
```

5.2.3 Further Reading: Categories

In general, a parent does not itself have a parent, but a *category* that indicates its properties:

```
sage: QQ.category()
Join of Category of number fields and Category of quotient fields and
    Category of metric spaces
```

Sage knows that $\mathbb{Q}$ is a field:

```
sage: QQ in Fields()
True
```

and thus, for instance, an additive and commutative group (see Figure 5.1):

```
sage: QQ in CommutativeAdditiveGroups()
True
```

Since $\mathbb{Q}$ is a field, $\mathbb{Q}[x]$ is a Euclidean ring:

```
sage: QQ['x'] in EuclideanDomains()
True
```

All these properties are used to provide rigorous and efficient computations on elements of these sets.

5.3 Domains with a Normal Form

Let us now browse some of the parents we will encounter in Sage.

We have seen in §2.1 how important normal forms[2] can be in computer algebra, since they allow to determine if two objects are mathematically equal in comparing their normal form representations. Each of the fundamental parents presented in this section corresponds to a *domain with normal form*, i.e., a set of mathematical objects having a normal form. This allows Sage to represent without any ambiguity the elements of each of these parents[3].

[2] In this book we use both *canonical form* and *normal form* to mean that two objects are mathematically identical if their canonical (or normal) forms are equal. Sometimes *normal form* is meant as a weaker notion, where only zero is assumed to have a unique representation.

[3] Most of the other parents available in Sage correspond to domains with a normal form, but not all of them. It also happens that, for efficiency reasons, Sage *represents* elements in normal form only when explicitly requested.

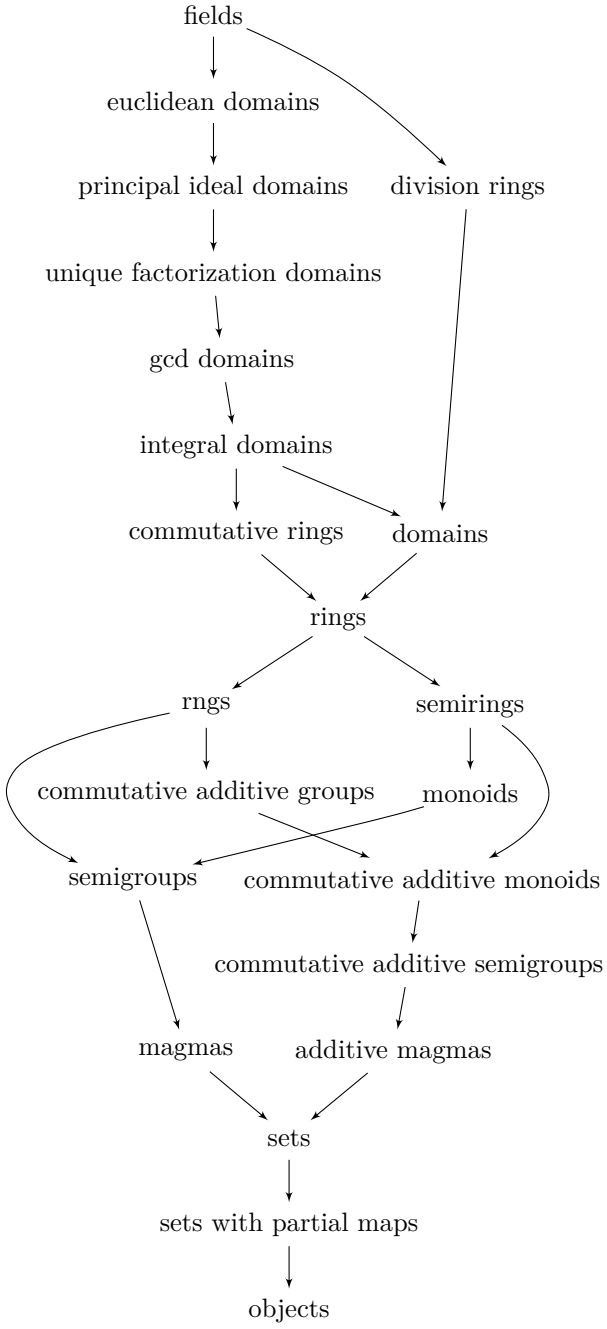


FIGURE 5.1 – A short part of the category graph in Sage.

Some basic Python types	
Python integers	`int`
Python floating-point numbers	`float`
Booleans (*true*, *false*)	`bool`
Character strings	`str`
Basic numerical domains	
Integers $\mathbb{Z}$	`ZZ` *or* `IntegerRing()`
Rational numbers $\mathbb{Q}$	`QQ` *or* `RationalField()`
Floating-point numbers with p bits	`Reals(p)` *or* `RealField(p)`
Complex floating-point numbers with p bits	`Complexes(p)` *or* `ComplexField(p)`
Rings and finite fields	
Integers modulo n, $\mathbb{Z}/n\mathbb{Z}$	`Integers(n)` *or* `IntegerModRing(n)`
Finite field $\mathbb{F}_q$	`GF(q)` *or* `FiniteField(q)`
Algebraic numbers	
Algebraic numbers $\bar{\mathbb{Q}}$	`QQbar` *or* `AlgebraicField()`
Real algebraic numbers	`AA` *or* `AlgebraicRealField()`
Number fields $\mathbb{Q}[x]/\langle p\rangle$	`NumberField(p)`
Symbolic computation	
Matrices $m \times n$ with coefficients in A	`MatrixSpace(A, m, n)`
Polynomials $A[x, y]$	`A['x,y']` *or* `PolynomialRing(A, 'x,y')`
Series $A[[x]]$	`A[['x']]` *or* `PowerSeriesRing(A, 'x')`
Symbolic expressions	`SR`

TABLE 5.1 – Main domains and parents.

5.3.1 Elementary Domains

We call *elementary computational domains* (or simply elementary domains) the classical sets of constants, with no variable: integers, rational numbers, floating-point numbers, booleans, integers modulo n...

Integers. The integers are represented in radix two internally, and printed by default in radix ten. As seen above, the Sage integers are objects of the class `Integer`. Their parent is the ring $\mathbb{Z}$:

```
sage: 5.parent()
Integer Ring
```

The integers are always in normal form; their equality is thus easy to check. As a consequence, to be able to represent integers in factorised form, the `factor` command needs a specific class:

```
sage: type(factor(4))
<class 'sage.structure.factorization_integer.IntegerFactorization'>
```

The `Integer` class is specific to Sage: by default, Python uses integers of type `int`. In general, the conversion from `Integer` to `int` — or vice versa — is automatic, but it might be necessary to convert explicitly by

```
sage: int(5)
5
sage: type(int(5))
<type 'int'>
```

or conversely

```
sage: Integer(5)
5
sage: type(Integer(5))
<type 'sage.rings.integer.Integer'>
```

Rational Numbers. The normal form property extends to rational numbers, elements of `QQ`, which are always represented in reduced form. Therefore, in the command

```
sage: factorial(99) / factorial(100) - 1 / 50
-1/100
```

the factorials are first evaluated, then the obtained fraction 1/100 is put into reduced form. Sage then constructs the rational number 1/50, performs the subtraction, then reduces again the result (there is nothing to do here).

Floating-Point Numbers. Real numbers cannot all be exactly represented in a finite format. Their numerical values are approximated by floating-point numbers, which will be discussed in more detail in Chapter 11.

Within Sage, floating-point numbers are encoded in binary radix. As a consequence, the floating-point number corresponding to the input `0.1` slightly differs from 1/10, since 1/10 is not exactly representable in binary! Each floating-point number has its own precision. The parent of floating-point numbers with p-bit significand is denoted `Reals(p)`, which for the default precision ($p = 53$) is also denoted `RR`. As for integers, Sage floating-point numbers differ from their Python analogue.

When they appear in a sum, product or quotient containing also integers or rational numbers, floating-point numbers are "contagious"; the complete expression is then evaluated as a floating-point number:

```
sage: 72/53 - 5/3 * 2.7
-3.14150943396227
```

Likewise, when the argument of some usual function is a floating-point number, the result is again a floating-point number:

```
sage: cos(1), cos(1.)
(cos(1), 0.540302305868140)
```

The `numerical_approx` method (or its alias `n`) evaluates numerically the remaining expressions. An optional argument allows us to set the number of significant digits used for this evaluation. Here is for example π with 50 significant digits:

```
sage: pi.n(digits=50)     # variant: n(pi,digits=50)
3.1415926535897932384626433832795028841971693993751
```

Complex Floating-Point Numbers. Similarly, the floating-point approximations of complex numbers with precision p are elements of `Complexes(p)` — or its alias `ComplexField(p)` —, or `CC` with the default precision of 53 bits. For example, we can construct a complex floating-point number and compute its argument by

```
sage: z = CC(1,2); z.arg()
1.10714871779409
```

Complex symbolic expressions

The imaginary unit i (denoted `I` or `i`), already encountered in the preceding chapters, is not an element of `CC`, but a symbolic expression (see §5.4.1):

```
sage: I.parent()
Symbolic Ring
```

We can use it to define a complex floating-point number with an explicit conversion:

```
sage: (1.+2.*I).parent()
Symbolic Ring
sage: CC(1.+2.*I).parent()
Complex Field with 53 bits of precision
```

In the world of symbolic expressions, the methods `real`, `imag` and `abs` give respectively the real part, the imaginary part and the modulus of a complex number:

```
sage: z = 3 * exp(I*pi/4)
sage: z.real(), z.imag(), z.abs().canonicalize_radical()
(3/2*sqrt(2), 3/2*sqrt(2), 3)
```

Booleans. Logic expressions also form a computational domain with normal form, but the class of boolean values is a basic type without specific parent in Sage. The two normal forms are `True` and `False` (or `true` and `false`):

```
sage: a, b, c = 0, 2, 3
sage: a == 1 or (b == 2 and c == 3)
True
```

In tests and loops, the conditions built from the operators `or` and `and` are evaluated lazily from left to right. This means that the evaluation of a condition `or` ends as soon as the first `True` value is encountered, without evaluating the rightmost terms; similarly with `and` and `False`. Hence the following divisibility test of b by a does not produce any error even if $a = 0$:

```
sage: a = 0; b = 12; (a == 0 and b == 0) or (a != 0 and b % a == 0)
```

The operator `not` takes precedence over `and`, which in turn takes precedence over `or`, the equality and comparison tests having precedence over all boolean operators. The two following tests are thus equivalent to the above one:

```
sage: ((a == 0) and (b == 0)) or ((a != 0) and (b % a == 0))
sage: a == 0 and b == 0 or not a == 0 and b % a == 0
```

In addition, Sage allows multiple equality or inequality tests, exactly like in mathematics:

$$x \leq y < z \leq t \quad \text{encoded by} \quad \texttt{x <= y < z <= t}$$
$$x = y = z \neq t \qquad\qquad \texttt{x == y == z != t}$$

In the simple cases, these tests are automatically performed; otherwise we call the `bool` command to force the evaluation:

```
sage: x, y = var('x, y')
sage: bool( (x-y)*(x+y) == x^2-y^2 )
True
```

Integers Modulo n. To define an integer modulo n, we first build its parent, the ring $\mathbb{Z}/n\mathbb{Z}$:

```
sage: Z4 = IntegerModRing(4); Z4
Ring of integers modulo 4
sage: m = Z4(7); m
3
```

As in the case of floating-point numbers, the computations involving m are done modulo 4 via automatic conversions. In the following example, 3 and 1 are automatically converted in elements of $\mathbb{Z}/4\mathbb{Z}$:

```
sage: 3 * m + 1
2
```

When p is prime, we can also choose to build $\mathbb{Z}/p\mathbb{Z}$ as a field:

```
sage: Z3 = GF(3); Z3
Finite Field of size 3
```

Both `IntegerModRing(n)` and `GF(p)` are domains with a normal form: the reduction modulo n or p are done automatically. The computations in rings and finite fields are detailed in Chapter 6.

5.3.2 Compound Domains

From well-defined constants, some classes of symbolic objects with variables and having a normal form can be constructed. The most important such classes are matrices, polynomials, rational functions and truncated power series.

The corresponding parents are parameterised by their coefficient domain. For example, matrices with integer coefficients differ from matrices with coefficients in $\mathbb{Z}/n\mathbb{Z}$, and the corresponding computation rules are automatically applied, without requiring an explicit call to a function reducing integers modulo n.

Part II of this book is mainly dedicated to these objects.

Matrices. The normal form[4] of a matrix is obtained when all its coefficients are themselves in normal form. As a consequence, a matrix defined over a field or ring with normal form is automatically in normal form:

```
sage: a = matrix(QQ, [[1,2,3],[2,4,8],[3,9,27]])
sage: (a^2 + 1) * a^(-1)
[  -5 13/2  7/3]
[   7    1 25/3]
[   2 19/2   27]
```

The `matrix` function call is a shortcut. Internally, Sage builds the corresponding parent, here the space of 3×3 matrices with coefficients in $\mathbb{Q}$ (which has normal form), then uses it to construct the matrix:

```
sage: M = MatrixSpace(QQ,3,3); M
Full MatrixSpace of 3 by 3 dense matrices over Rational Field
sage: a = M([[1,2,3],[2,4,8],[3,9,27]])
sage: (a^2 + 1) * a^(-1)
[  -5 13/2  7/3]
[   7    1 25/3]
[   2 19/2   27]
```

The operations on symbolic matrices are described in Chapter 8, and on numerical matrices in Chapter 13.

Polynomials and Fractions. Like matrices, polynomials in Sage "know" the type of their coefficients. Their parents are polynomial rings like $\mathbb{Z}[x]$ or $\mathbb{C}[x, y, z]$, presented in detail in Chapters 7 and 9, and which can be built as follows:

```
sage: P = ZZ['x']; P
Univariate Polynomial Ring in x over Integer Ring
sage: F = P.fraction_field(); F
Fraction Field of Univariate Polynomial Ring in x over Integer Ring
sage: p = P(x+1) * P(x); p
x^2 + x
sage: p + 1/p
```

[4] Do not confuse this concept of normal form with the normal forms of a matrix viewed as a linear transformation, which will be discussed in Chapter 8.

```
(x^4 + 2*x^3 + x^2 + 1)/(x^2 + x)
sage: parent(p + 1/p)
Fraction Field of Univariate Polynomial Ring in x over Integer Ring
```

As we will see in §5.4.2, there is no optimal representation for polynomials and fractions. The elements of polynomial rings are represented in expanded form. These rings do therefore have a normal form as soon as the coefficients themselves belong to a domain with normal form.

These polynomials differ from the polynomial expressions (`Symbolic Ring`) we have seen in Chapter 2, which do not have a well-defined coefficient type, neither a parent reflecting such a type. The latter give an alternative to "true" polynomials, which can be useful, for example, to mix polynomials and other mathematical expressions. However, contrary to polynomial rings, when we work with such expressions, we have to explicitly call a *reduction command* like `expand` to put them in normal form (if such a form exists).

Power Series. Truncated power series are objects of the form

$$a_0 + a_1\, x + a_2\, x^2 + \cdots + a_n\, x^n + \mathcal{O}(x^{n+1})$$

used for example to represent Taylor expansions, and whose usage in Sage is described in §7.5. The parent of series in x, truncated at order n, and with coefficients in A, is the ring $A[[x]]$, build with `PowerSeriesRing(A, 'x', n)`.

Like polynomials, truncated power series have an analogue in the world `SR` of symbolic expressions. The corresponding command to reduce to normal form is `series`.

```
sage: f = cos(x).series(x == 0, 6); 1 / f
```

$$\frac{1}{1+(-\frac{1}{2})x^2+\frac{1}{24}x^4+O(x^6)}$$

```
sage: (1 / f).series(x == 0, 6)
```

$$1 + \tfrac{1}{2}x^2 + \tfrac{5}{24}x^4 + O\left(x^6\right)$$

Algebraic Numbers. An algebraic number is defined as root of a polynomial. When the polynomial degree is 5 or more, in general it is not possible to explicitly write its roots in terms of the operations $+, -, \times, /, \sqrt{\cdot}$. However, many computations involving the roots can be performed successfully without any other information than the polynomial itself.

```
sage: k.<a> = NumberField(x^3 + x + 1); a^3; a^4+3*a
-a - 1
-a^2 + 2*a
```

This book does not describe in detail how to play with algebraic numbers in Sage, however several examples can be found in Chapters 7 and 9.

5.4 Expressions vs Computational Domains

Several approaches are thus possible for manipulating objects like polynomials within Sage. We can consider them as particular symbolic expressions, as in the first chapters, or introduce a given ring of polynomials and compute with its elements. To conclude this chapter, we briefly describe the parent of symbolic expressions, the `SR` domain, then we demonstrate through several examples how important it is to control the domain of computations, and the differences between both approaches.

5.4.1 Symbolic Expressions as a Computational Domain

Symbolic expressions themselves form a computational domain. In Sage, their parent is the *symbolic ring*:

```
sage: parent(sin(x))
Symbolic Ring
```

that can also be obtained with:

```
sage: SR
Symbolic Ring
```

The properties of this ring are rather fuzzy; it is commutative:

```
sage: SR.category()
Category of commutative rings
```

and the computation rules assume roughly speaking that all symbolic variables are in $\mathbb{C}$.

The form of expressions in `SR` (polynomials, fractions, trigonometric expressions) being not apparent in their class or parent, the result of a computation often requires some manual transformations to obtain the desired form (see §2.1), by using for example `expand`, `combine`, `collect` and `simplify`. To use these functions well we have to know which kind of transformation they perform, to which sub-classes[5] of symbolic expressions these transformations apply, and which of these sub-classes have a normal form. In particular, the blind use of the `simplify` command can yield wrong results. Some variants of `simplify` allow then to precisely describe the transformation to apply.

5.4.2 Examples: Polynomials and Normal Forms

Let us build the ring $\mathbb{Q}[x_1, x_2, x_3, x_4]$ of polynomials in 4 variables:

```
sage: R = QQ['x1,x2,x3,x4']; R
Multivariate Polynomial Ring in x1, x2, x3, x4 over Rational Field
sage: x1, x2, x3, x4 = R.gens()
```

The elements of R are automatically put in expanded form:

[5] In the sense of subset, and not of Python class.

```
sage: x1 * (x2 - x3)
x1*x2 - x1*x3
```

which, as we have seen, is a normal form. In particular, the test to zero in R is trivial:

```
sage: (x1+x2)*(x1-x2) - (x1^2 - x2^2)
0
```

An expanded form is not always optimal. For example, if we build the Vandermonde determinant $\prod_{1\leq i<j\leq n}(x_i - x_j)$:

```
sage: prod( (a-b) for (a,b) in Subsets([x1,x2,x3,x4],2) )
x1^3*x2^2*x3 - x1^2*x2^3*x3 - x1^3*x2*x3^2 + x1*x2^3*x3^2
+ x1^2*x2*x3^3 - x1*x2^2*x3^3 - x1^3*x2^2*x4 + x1^2*x2^3*x4
+ x1^3*x3^2*x4 - x2^3*x3^2*x4 - x1^2*x3^3*x4 + x2^2*x3^3*x4
+ x1^3*x2*x4^2 - x1*x2^3*x4^2 - x1^3*x3*x4^2 + x2^3*x3*x4^2
+ x1*x3^3*x4^2 - x2*x3^3*x4^2 - x1^2*x2*x4^3 + x1*x2^2*x4^3
+ x1^2*x3*x4^3 - x2^2*x3*x4^3 - x1*x3^2*x4^3 + x2*x3^2*x4^3
```

we obtain $4! = 24$ terms. The same construct with an expression from `SR` remains under factored form, and is much more compact and readable:

```
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) )
-(x1 - x2)*(x1 - x3)*(x1 - x4)*(x2 - x3)*(x2 - x4)*(x3 - x4)
```

In addition, a factored representation allows faster gcd computations. However, it would be unwise to put automatically every polynomial into factored form, even if this is also a normal form, since the factorisation is computationally expensive, and makes additions costly.

In general, depending on the kind of computation, the optimal representation of an element is not always its normal form (if it exists). This leads computer algebra systems to a compromise with expressions. Some basic simplifications, like the reduction of fractions or the multiplication by zero, are done automatically; the other transformations are left to the user with the provided specialised commands.

5.4.3 Example: Polynomial Factorisation

Let us consider the factorisation of the following polynomial expression:

```
sage: x = var('x')
sage: p = 54*x^4+36*x^3-102*x^2-72*x-12
sage: factor(p)
6*(x^2 - 2)*(3*x + 1)^2
```

Is this answer satisfying? It is indeed a factorisation of p, however its completeness heavily depends on the context! For now, Sage considers `p` as a symbolic expression, which happens to be polynomial. Sage cannot know if we wish to factor p as a

product of polynomials with integer coefficients, or with rational coefficients (for example).

To take full control, we will make it clear in which mathematical set (i.e., computational domain) p lives. To start, let us consider p as a polynomial with integer coefficients. We thus define the ring $R = \mathbb{Z}[x]$ of these polynomials:

```
sage: R = ZZ['x']; R
Univariate Polynomial Ring in x over Integer Ring
```

Then we convert p in this ring:

```
sage: q = R(p); q
54*x^4 + 36*x^3 - 102*x^2 - 72*x - 12
```

The output seems identical, however q knows it is an element of R:

```
sage: parent(q)
Univariate Polynomial Ring in x over Integer Ring
```

As a consequence, its factorisation is uniquely defined:

```
sage: factor(q)
2 * 3 * (3*x + 1)^2 * (x^2 - 2)
```

Let us proceed similarly in the rational field:

```
sage: R = QQ['x']; R
Univariate Polynomial Ring in x over Rational Field
sage: q = R(p); q
54*x^4 + 36*x^3 - 102*x^2 - 72*x - 12
sage: factor(q)
(54) * (x + 1/3)^2 * (x^2 - 2)
```

In this new context, the factorisation is again well-defined, but different from the previous one.

Let us now compute a complete factorisation over the complex numbers. A first solution is to allow a numerical approximation of complex numbers with 16 bits of precision:

```
sage: R = ComplexField(16)['x']; R
Univariate Polynomial Ring in x over Complex Field
with 16 bits of precision
sage: q = R(p); q
54.00*x^4 + 36.00*x^3 - 102.0*x^2 - 72.00*x - 12.00
sage: factor(q)
(54.00) * (x - 1.414) * (x + 0.3333)^2 * (x + 1.414)
```

Another solution is to extend the field of rational numbers, e.g., adding $\sqrt{2}$.

```
sage: R = QQ[sqrt(2)]['x']; R
Univariate Polynomial Ring in x over Number Field in sqrt2
with defining polynomial x^2 - 2
sage: q = R(p); q
```

```
54*x^4 + 36*x^3 - 102*x^2 - 72*x - 12
sage: factor(q)
(54) * (x - sqrt2) * (x + sqrt2) * (x + 1/3)^2
```

Finally, maybe we want that coefficients be considered modulo 5?

```
sage: R = GF(5)['x']; R
Univariate Polynomial Ring in x over Finite Field of size 5
sage: q = R(p); q
4*x^4 + x^3 + 3*x^2 + 3*x + 3
sage: factor(q)
(4) * (x + 2)^2 * (x^2 + 3)
```

5.4.4 Synthesis

In the preceding examples, we have shown how the user might control the level of rigour in her/his computations.

On the one hand, she/he can use symbolic expressions. These expressions live in the ring `SR`. They offer several methods (presented in Chapter 2) which apply well to some sub-classes of expressions, like polynomial expressions. When we recognise to which classes a given expression belongs to, this helps to know which functions could be applied. The simplification of expressions is a particular problem where this recognition is crucial. The main classes of expression are defined to take into account this simplification issue, and we will prefer this approach in the rest of this book.

On the other hand, the user can *construct* a parent which will explicitly define the computational domain. It is especially interesting when this parent has a *normal form*: i.e., when two objects are mathematically equal if and only if they have the same representation.

As a summary, the main advantage of symbolic expressions (`SR`) is their ease of use: no explicit declaration of the computational domain, easy addition of new variables or functions, easy change of the computational domain (for example when one takes the sine of a polynomial expression), use of all possible calculus tools (integration, etc.). The advantages of explicitly defining the computational domain are in the first place pedagogical, more rigorous computations[6], the automatic normal form transformation (which can also be a drawback!), and the easy access to advanced constructions that would be difficult with symbolic expressions (computations in a finite field or an algebraic extension of $\mathbb{Q}$, in a non-commutative ring, etc.).

[6]Sage is not a *certified* computer algebra system: a bug is thus always possible; however, there will be no use of implicit assumption.

Part II

Algebra and Symbolic Computation

God made the integers, all else is the work of man.
Leopold Kronecker (1823 - 1891)

6 Finite Fields and Elementary Number Theory

This chapter describes the use of Sage for elementary number theory, for working with objects related to finite fields (§6.1), for primality testing (§6.2) and integer factorisation (§6.3); we will also discuss some applications (§6.4).

6.1 Finite Fields and Rings

Finite rings and fields are basic objects, both in number theory and throughout computer algebra. Indeed, many algorithms in computer algebra involve computations over finite fields, where one can exploit the information obtained using techniques such as Hensel lifting, or reconstruction using the Chinese Remainder Theorem. As an example, we can mention the Cantor-Zassenhaus algorithm for factoring univariate polynomials with integer coefficients, which begins by factoring the polynomial over a finite field.

6.1.1 The Ring of Integers Modulo n

In Sage, the ring $\mathbb{Z}/n\mathbb{Z}$ of integers modulo n is defined using the constructor `IntegerModRing` (or, more simply, `Integers`). All objects constructed using this constructor and those derived from them are systematically reduced modulo n, and so have a canonical (or normal) form: that is to say, two variables representing the same value modulo n also have the same internal representation. In certain very special situations, it may be more efficient to delay these reductions modulo n; for example, if one multiplies matrices with such coefficients, one would then rather work with integers, and carry out the reductions modulo n "by hand"

using `a % n`. Note that the modulus n does not appear explicitly in the displayed value:

```
sage: a = IntegerModRing(15)(3); b = IntegerModRing(17)(3); a, b
(3, 3)
sage: a == b
False
```

One consequence of this is that when one uses "cut-and-paste" to copy integers modulo n, one loses information about n. Given a variable whose value is an integer modulo n, one can recover information about n using the methods `base_ring` or `parent`, and the value of n using the method `characteristic`:

```
sage: R = a.parent(); R
Ring of integers modulo 15
sage: R.characteristic()
15
```

The basic operations (addition, subtraction and multiplication) are overloaded for integers modulo n, and call the appropriate functions; also, integers are converted automatically when one of the operands is an integer modulo n:

```
sage: a + a, a - 17, a * a + 1, a^3
(6, 1, 10, 12)
```

For inversion, $1/a \bmod n$, or division, $b/a \bmod n$, Sage carries out the operation if possible; otherwise, i.e., when a and n have a nontrivial common factor, a `ZeroDivisionError` is raised:

```
sage: 1/(a+1)
4
sage: 1/a
Traceback (most recent call last):
  ...
ZeroDivisionError: Inverse does not exist.
```

To obtain the value of a as an integer from its residue $a \bmod n$, one can use the method `lift` or even `ZZ`:

```
sage: z = a.lift(); y = ZZ(a); y, type(y), y == z
(3, <type 'sage.rings.integer.Integer'>, True)
```

The *additive order* of a modulo n is the smallest integer $k > 0$ such that $ka = 0 \bmod n$. It is equal to $k = n/g$, where $g = \gcd(a, n)$, and is given by the method `additive_order` (we will see later that one can also use `Mod` or `mod` to define integers modulo n):

```
sage: [Mod(x,15).additive_order() for x in range(0,15)]
[1, 15, 15, 5, 15, 3, 5, 15, 15, 5, 3, 15, 5, 15, 15]
```

The *multiplicative order* of a modulo n, for a coprime[1] to n, is the smallest integer $k > 0$ such that $a^k = 1 \bmod n$. (If a had a common divisor p with n, then

[1] "coprime" and "relatively prime" are synonymous.

$a^k \bmod n$ would be a multiple of p for all k.) If this multiplicative order equals $\varphi(n)$, which is the order of the multiplicative group modulo n, one says that a is a *generator* of this group. Thus for $n = 15$, there is no generator, since the maximal order is $4 < 8 = \varphi(15)$:

```
sage: [[x, Mod(x,15).multiplicative_order()]
....:   for x in range(1,15) if gcd(x,15) == 1]
[[1, 1], [2, 4], [4, 2], [7, 4], [8, 4], [11, 2], [13, 4], [14, 2]]
```

Here is an example with $n = p$ prime, where 3 is a generator:

```
sage: p = 10^20 + 39; mod(2,p).multiplicative_order()
50000000000000000019
sage: mod(3,p).multiplicative_order()
100000000000000000038
```

An important operation on $\mathbb{Z}/n\mathbb{Z}$ is *modular exponentiation*, which means to calculate $a^e \bmod n$. The RSA crypto-system relies on this operation. To calculate $a^e \bmod n$, the most efficient algorithms require of the order of $\log e$ multiplications or squarings modulo n. It is crucial to reduce all calculations modulo n systematically, and not compute a^e first as an integer, as the following example shows:

```
sage: n = 3^100000; a = n-1; e = 100
sage: %timeit (a^e) % n
5 loops, best of 3: 387 ms per loop
sage: %timeit power_mod(a,e,n)
125 loops, best of 3: 3.46 ms per loop
```

6.1.2 Finite Fields

Finite fields[2] are defined using the constructor `FiniteField`, or more simply `GF`. As well as constructing *prime fields* `GF(p)` with p prime, one can construct *non-prime finite fields* `GF(q)` with $q = p^k$, where p is prime and $k > 1$ an integer. As with rings, objects created in such a field have a canonical representation, and reduction is carried out at each arithmetic operation. Finite fields have the same properties as rings (§6.1.1), with in addition the possibility of inverting each non-zero element:

```
sage: R = GF(17); [1/R(x) for x in range(1,17)]
[1, 9, 6, 13, 7, 3, 5, 15, 2, 12, 14, 10, 4, 11, 8, 16]
```

A non-prime finite field $\mathbb{F}_{p^k}$ with p prime and $k > 1$ is isomorphic to the quotient ring of polynomials in $\mathbb{F}_p[x]$ modulo a monic irreducible polynomial f of degree k. In this case, Sage will provide a name for the *generator* of the field, that is, the variable x, or the user can provide a name:

[2]The finite field with q elements is either denoted $\mathbb{F}_q$, or GF(q) (where "GF" stands for "Galois Field"). Here we will use the notation $\mathbb{F}_q$ for the mathematical object, and the notation GF(q) in Sage code.

```
sage: R = GF(9,name='x'); R
Finite Field in x of size 3^2
```

Here, Sage has automatically chosen the polynomial f:

```
sage: R.polynomial()
x^2 + 2*x + 2
```

Field elements are thus represented by polynomials in the generator x, $a_{k-1}x^{k-1} + \cdots + a_1 x + a_0$, with coefficients a_i which are elements of $\mathbb{F}_p$:

```
sage: Set([r for r in R])
{0, 1, 2, x, x + 1, x + 2, 2*x, 2*x + 1, 2*x + 2}
```

One can also make Sage use a specific irreducible polynomial f:

```
sage: Q.<x> = PolynomialRing(GF(3))
sage: R2 = GF(9, name='x', modulus=x^2+1); R2
Finite Field in x of size 3^2
```

Be careful: even though the two fields `R` and `R2` created above are both isomorphic to $\mathbb{F}_9$, Sage provides no isomorphism between them automatically:

```
sage: p = R(x+1); R2(p)
Traceback (most recent call last):
  ...
TypeError: unable to coerce from a finite field other than the prime
    subfield
```

6.1.3 Rational Reconstruction

The problem of *rational reconstruction* is a useful application of modular methods. Given a residue a modulo m, it involves finding a "small" rational number x/y such that $x/y \equiv a \bmod m$. If one knows that such a small rational number exists, instead of computing x/y directly as a rational number, one may instead compute x/y modulo m, which gives the residue a, and then one recovers x/y via rational reconstruction. This second approach is often more efficient, since one has replaced computations with rationals, possibly involving costly gcd calculations, by modular calculations.

LEMMA. Let $a, m \in \mathbb{N}$, with $0 < a < m$. There exists at most one pair of coprime integers $x, y \in \mathbb{Z}$ such that $x/y \equiv a \bmod m$ with $0 < |x|, y \leq \sqrt{m/2}$.

Such a pair x, y does not always exist: for example, take $a = 2$ and $m = 5$. The rational reconstruction algorithm is based on the extended Euclidean algorithm. The extended gcd of m and a computes a sequence of integers $a_i = \alpha_i m + \beta_i a$, where the a_i are decreasing, and the coefficients α_i, β_i increase in absolute value. It therefore suffices to stop as soon as $|a_i|, |\beta_i| \leq \sqrt{m/2}$, and the solution is then $x/y = a_i/\beta_i$. This algorithm is implemented in the Sage function `rational_reconstruction`, which returns x/y when a solution exists, raising an error if not:

```
sage: rational_reconstruction(411,1000)
```

```
-13/17
sage: rational_reconstruction(409,1000)
Traceback (most recent call last):
  ...
ArithmeticError: rational reconstruction of 409 (mod 1000) does not
    exist
```

To illustrate the use of rational reconstruction, consider the computation of the Harmonic numbers $H_n = 1 + 1/2 + \cdots + 1/n$. A naive calculation using rational numbers would be as follows:

```
sage: def harmonic(n):
....:     return add([1/x for x in range(1,n+1)])
```

Now we know that H_n can be written in the form p_n/q_n with integers p_n, q_n, where $q_n = \mathrm{lcm}(1, 2, \ldots, n)$. We also know that $H_n \leq \log n + 1$, which allows us to bound p_n. This leads to the following function, which finds H_n using modular arithmetic and rational reconstruction:

```
sage: def harmonic_mod(n,m):
....:     return add([1/x % m for x in range(1,n+1)])
sage: def harmonic2(n):
....:     q = lcm(range(1,n+1))
....:     pmax = RR(q*(log(n)+1))
....:     m = ZZ(2*pmax^2)
....:     m = ceil(m/q)*q + 1
....:     a = harmonic_mod(n,m)
....:     return rational_reconstruction(a,m)
```

In this example, the function `harmonic2` is no more efficient than the original function `harmonic`, but it illustrates the method. It is not always necessary to know a rigorous bound for x and y, as a rough estimate "by eye" will suffice, provided that one is able to verify easily that x/y is the correct solution.

One can generalise the method of rational reconstruction to handle numerators x and denominators y of different sizes; see for example Section 5.10 of the book [vzGG03].

6.1.4 The Chinese Remainder Theorem

Another useful application of modular arithmetic involves the use of the *Chinese Remainder Theorem*, or CRT, commonly called "Chinese remaindering". Given two coprime moduli m and n, and two residue classes $a \bmod m$ and $b \bmod n$, we seek an integer x such that $x \equiv a \bmod m$ and $x \equiv b \bmod n$. The Chinese Remainder Theorem enables us to recover x uniquely modulo the product mn. To see how this works, one deduces from $x \equiv a \bmod m$ that x has the form $x = a + \lambda m$ with $\lambda \in \mathbb{Z}$. Substituting into $x \equiv b \bmod n$, one obtains $\lambda \equiv \lambda_0 \bmod n$, where $\lambda_0 = (b - a)/m \bmod n$. Hence $x = x_0 + \mu nm$, where $x_0 = a + \lambda_0 m$, and μ is an arbitrary integer.

Here we have presented the simplest variant of the Chinese Remainder Theorem. One can also consider the case of several moduli $m_1, m_2, \ldots, m_k$. The Sage command for finding x_0, given a, b, m, n, is `crt(a,b,m,n)`:

```
sage: a = 2; b = 3; m = 5; n = 7; lambda0 = (b-a)/m % n; a + lambda0 * m
17
sage: crt(2,3,5,7)
17
```

Let us return to the computation of H_n. We first compute $H_n \bmod m_i$ for $i = 1, 2, \ldots, k$, and then obtain $H_n \bmod m_1 \cdots m_k$ by Chinese remaindering, finally recovering the value of H_n by rational reconstruction:

```
sage: def harmonic3(n):
....:     q = lcm(range(1,n+1))
....:     pmax = RR(q*(log(n)+1))
....:     B = ZZ(2*pmax^2)
....:     a = 0; m = 1; p = 2^63
....:     while m < B:
....:         p = next_prime(p)
....:         b = harmonic_mod(n,p)
....:         a = crt(a,b,m,p)
....:         m = m*p
....:     return rational_reconstruction(a,m)
sage: harmonic(100) == harmonic3(100)
True
```

The Sage function `crt` may also be used when the moduli m and n are not coprime. If $g = \gcd(m, n)$, then a solution exists if and only if $a \equiv b \bmod g$:

```
sage: crt(15,1,30,4)
45
sage: crt(15,2,30,4)
Traceback (most recent call last):
  ...
ValueError: No solution to crt problem since gcd(30,4) does not divide
    15-2
```

A more complicated application of the Chinese Remainder Theorem is given in Exercise 23.

6.2 Primality

Testing whether an integer is prime is a fundamental operation for a symbolic computer software package. Even if the user is not aware of it, such tests are carried out thousands of times per second by the software. For example, to factor a polynomial in $\mathbb{Z}[x]$, one starts by factoring it in $\mathbb{F}_p[x]$ for some prime number p, and one must therefore find a suitable prime.

Useful commands	
Ring of integers modulo n	`IntegerModRing`(n)
Finite field with q elements	`GF`(q)
Pseudo-primality test	`is_pseudoprime`(n)
Primality test	`is_prime`(n)

Table 6.1 – Review.

There are two main classes of primality test. The most efficient are *pseudo-primality* tests, and are in general based on forms of Fermat's Little Theorem, which says that if p is prime, then every integer a with $0 < a < p$ is an element of the multiplicative group $(\mathbb{Z}/p\mathbb{Z})^*$, and hence $a^{p-1} \equiv 1 \bmod p$. One uses small values of a $(2, 3, \ldots)$ to speed up the computation of $a^{p-1} \bmod p$. If $a^{p-1} \not\equiv 1 \bmod p$, then p is certainly not prime. If $a^{p-1} \equiv 1 \bmod p$, one cannot conclude either that p is or is not prime; we say that p is a (Fermat) *pseudo-prime to base* a. The intuition is that an integer p which is a pseudo-prime to many bases has a greater chance of being prime (but see below). Pseudo-primality tests share the property that when they return the verdict `False`, the number is certainly composite, whereas when they return `True`, no definite conclusion is possible.

The second class consists of *true primality* tests. These tests always return a correct answer, but can be less efficient than pseudo-primality tests, especially for numbers that are pseudo-primes to many bases, and in particular for actual primes. Many software packages only provide pseudo-primality tests, despite the name of the corresponding function (`isprime`, for example) sometimes leading the user to believe that a true primality test is provided. Sage provides two different functions: `is_pseudoprime` for pseudo-primality, and `is_prime` for true primality:

```
sage: p = previous_prime(2^400)
sage: %timeit is_pseudoprime(p)
625 loops, best of 3: 1.07 ms per loop
sage: %timeit is_prime(p)
5 loops, best of 3: 485 ms per loop
```

We see in this example that the primality test is more costly; when possible, therefore, one prefers to use `is_pseudoprime`.

Some primality testing algorithms provide a *certificate*, which allows an independent subsequent verification of the result, often more efficiently than the test itself. Sage does not provide such a certificate in the current release, but one can construct one using Pocklington's Theorem:

Theorem. Let $n > 1$ be an odd integer such that $n - 1 = FR$, with $F \geq \sqrt{n}$. If for each prime factor p of F, there exists a such that $a^{n-1} \equiv 1 \bmod n$ and $a^{(n-1)/p} - 1$ is coprime to n, then n is prime.

Consider for example $n = 2^{31} - 1$. The factorisation of $n - 1$ is $2 \cdot 3^2 \cdot 7 \cdot 11 \cdot 31 \cdot 151 \cdot 331$. One can take $F = 151 \cdot 331$, and $a = 3$ satisfies the condition for both

factors $p = 151$ and $p = 331$. Hence it suffices to prove the primality of 151 and 331 in order to deduce that n is prime. This test uses modular exponentiation in an important way.

Carmichael numbers

Carmichael numbers are composite integers n that are pseudo-primes to all bases coprime to n. Fermat's Little Theorem is insufficient to distinguish these from primes, however many bases are tested. The smallest Carmichael number is $561 = 3 \cdot 11 \cdot 17$. A Carmichael number must have at least three prime factors: for suppose that $n = pq$ is a Carmichael number, with p, q primes and $p < q$; by definition of Carmichael numbers, if a is a primitive root modulo q then $a^{n-1} \equiv 1$ modulo n implies that the same congruence also holds modulo q, and hence that $n-1$ is a multiple of $q-1$. Then n must be of the form $q + \lambda q(q-1)$, since it is a multiple of q and $n-1$ is a multiple of $q-1$; now $n = pq$ implies $p = \lambda(q-1)+1$, which contradicts $p < q$. If $n = pqr$, then n is a Carmichael number if $a^{n-1} \equiv 1 \bmod p$, and similarly modulo q and r, since then the Chinese Remainder Theorem implies that $a^{n-1} \equiv 1 \bmod n$. So a sufficient condition is that $n-1$ is divisible by each of $p-1$, $q-1$ and $r-1$:

```
sage: [560 % (x-1) for x in [3,11,17]]
[0, 0, 0]
```

Exercise 20. Write a Sage function to count the Carmichael numbers $n = pqr \leq N$, with p, q, r distinct odd primes. How many do you find for $N = 10^4, 10^5, 10^6, 10^7$? (Richard Pinch has counted 20138200 Carmichael numbers less than 10^{21}.)

Finally, in order to repeat an operation on all prime numbers in an interval, it is better to employ the construction `prime_range`, which constructs a table of primes using a sieve, than to simply use a loop with `next_probable_prime` or `next_prime`:

```
sage: def count_primes1(n):
....:     return add([1 for p in range(n+1) if is_prime(p)])
sage: %timeit count_primes1(10^5)
5 loops, best of 3: 674 ms per loop
```

The function is faster if one uses `is_pseudoprime` instead of `is_prime`:

```
sage: def count_primes2(n):
....:     return add([1 for p in range(n+1) if is_pseudoprime(p)])
sage: %timeit count_primes2(10^5)
5 loops, best of 3: 256 ms per loop
```

In this example, it is worth using a loop rather than constructing a list of 10^5 elements, and again `is_pseudoprime` is faster than `is_prime`:

```
sage: def count_primes3(n):
....:     s = 0; p = 2
```

```
....:     while p <= n: s += 1; p = next_prime(p)
....:     return s
sage: %timeit count_primes3(10^5)
5 loops, best of 3: 49.2 ms per loop
sage: def count_primes4(n):
....:     s = 0; p = 2
....:     while p <= n: s += 1; p = next_probable_prime(p)
....:     return s
sage: %timeit count_primes4(10^5)
5 loops, best of 3: 48.6 ms per loop
```

Using the iterator `prime_range` is faster still:

```
sage: def count_primes5(n):
....:     s = 0
....:     for p in prime_range(n): s += 1
....:     return s
sage: %timeit count_primes5(10^5)
125 loops, best of 3: 2.67 ms per loop
```

6.3 Factorisation and Discrete Logarithms

One says that an integer a is a square, or a quadratic residue, modulo n if there exists x such that $a \equiv x^2 \bmod n$. If not, one says that a is a quadratic non-residue[3] modulo n. When $n = p$ is prime, there is a test to decide efficiently whether a is a quadratic residue, using the computation of the Jacobi symbol of a and p, denoted $(a|p)$, which takes the values $\{-1, 0, 1\}$, where $(a|p) = 0$ when a is a multiple of p, and $(a|p) = 1$ (respectively, $(a|p) = -1$) when a is (respectively, is not) a square modulo p. The complexity of computing the Jacobi symbol $(a|n)$ is essentially the same as that of computing the gcd of a and n, namely $O(M(\ell) \log \ell)$ where ℓ is the size of n, and $M(\ell)$ is the cost of multiplying two integers of size ℓ. However, implementations of Jacobi symbols — as of gcds — do not all have this complexity (here, `a.jacobi(n)` computes $(a|n)$):

```
sage: p = (2^42737+1)//3; a = 3^42737
sage: %timeit a.gcd(p)
125 loops, best of 3: 4.3 ms per loop
sage: %timeit a.jacobi(p)
25 loops, best of 3: 26.1 ms per loop
```

When n is composite, finding solutions to $x^2 \equiv a \bmod n$ is as hard as factorising n. Moreover, the Jacobi symbol, which is relatively simple to compute, only gives partial information: if $(a|n) = -1$ then there is no solution, since the existence of a solution implies $(a|p) = 1$ for all prime factors p of n, hence $(a|n) = 1$; but $(a|n) = +1$ does not imply that a is a square modulo n when n is composite.

[3] This terminology is traditional, though "non-quadratic residue" would be more logical.

Let n be a positive integer, let g be a *generator* of the multiplicative group modulo n (we assume here that n is such that this group is cyclic), and let a be coprime to n. By definition of the fact that g is a generator, there is an integer x such that $g^x = a \bmod n$. The *discrete logarithm problem* consists of finding such an integer x. The `log` method gives a solution to this problem:

```
sage: p = 10^10+19; a = mod(17,p); a.log(2)
6954104378
sage: mod(2,p)^6954104378
17
```

The best known algorithms for computing discrete logarithms have the same order of complexity, as a function of the size of n, as those for factoring n. However, the current implementation of discrete logarithms in Sage is not very efficient:

```
sage: p = 10^37+43; a = mod(17,p)
sage: time r = a.log(2)
CPU times: user 1min 32s, sys: 64 ms, total: 1min 32s
Wall time: 1min 34s
```

Aliquot sequences

The *aliquot sequence* associated to a positive integer n is the recurrent sequence (s_k) defined by: $s_0 = n$ and $s_{k+1} = \sigma(s_k) - s_k$, where $\sigma(s_k)$ is the sum of the positive divisors of n, i.e., s_{k+1} is the sum of the *proper* divisors of s_k, excluding s_k itself. The iteration stops when $s_k = 1$, so s_{k-1} is prime, or when the sequence (s_k) enters a cycle. For example, starting from $n = 30$ one obtains:

$$30, 42, 54, 66, 78, 90, 144, 259, 45, 33, 15, 9, 4, 3, 1.$$

When the cycle has length one, we say that the starting integer is *perfect*, for example $6 = 1 + 2 + 3$ and $28 = 1 + 2 + 4 + 7 + 14$ are perfect. When the cycle has length two, the two integers in the cycle are called *amicable* and form an *amicable pair*, for example 220 and 284. When the cycle has length three or more, the integers in the cycle are called *sociable*.

Exercise 21. Calculate the aliquot sequence starting with 840, take the 5 first and 5 last terms, and draw the graph of $\log_{10} s_k$ as a function of k (you can use the function `sigma`).

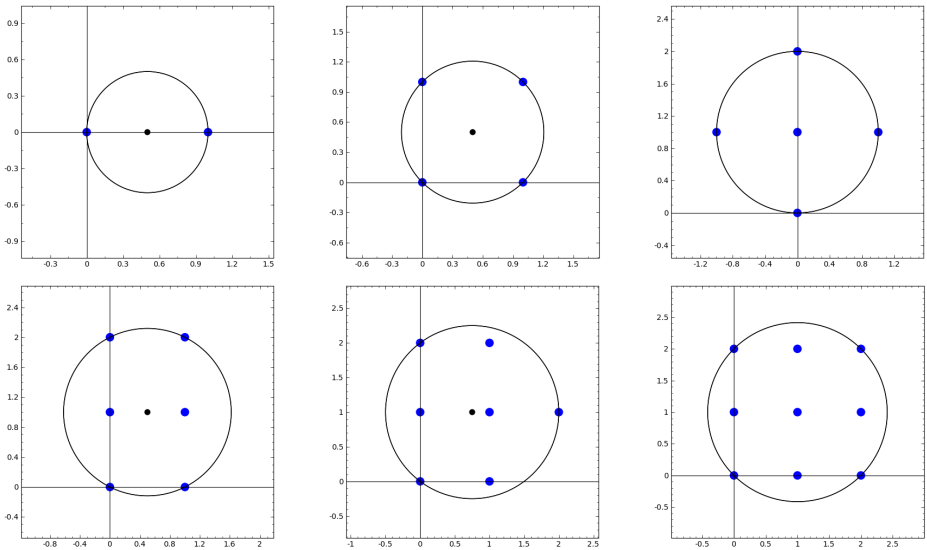
6.4 Applications

6.4.1 The Constant δ

The constant δ is a two-dimensional generalisation of Euler's constant γ. It is defined as follows:

$$\delta = \lim_{n\to\infty} \left(\sum_{k=2}^{n} \frac{1}{\pi r_k^2} - \log n \right), \tag{6.1}$$

where r_k is the radius of the smallest closed disc in the affine plane $\mathbb{R}^2$ containing at least k points of $\mathbb{Z}^2$. For example, $r_2 = 1/2$, $r_3 = r_4 = \sqrt{2}/2$, $r_5 = 1$, $r_6 = \sqrt{5}/2$, $r_7 = 5/4$, and $r_8 = r_9 = \sqrt{2}$:



Exercise 22 (Masser-Gramain constant)**.** 1. Write a function which takes as input a positive integer k, and returns the radius r_k and the centre (x_k, y_k) of a minimal disc, of radius r_k, containing at least k points of $\mathbb{Z}^2$. You may assume that $r_k < \sqrt{k/\pi}$.

2. Write a function which draws the circle with centre (x_k, y_k) and radius r_k, together with $m \geq k$ points of $\mathbb{Z}^2$, as above.

3. Using the bounding inequalities

$$\frac{\sqrt{\pi(k-6)+2}-\sqrt{2}}{\pi} < r_k < \sqrt{\frac{k-1}{\pi}}, \tag{6.2}$$

calculate an approximation of δ with an error at most 0.3.

6.4.2 Computation of a Multiple Integral

This application was inspired by the article [Bea09]. Let k and $n_1, n_2, \ldots, n_k$ be non-negative integers. We wish to compute the integral

$$I = \int_V x_1^{n_1} x_2^{n_2} \cdots x_k^{n_k} \, \mathrm{d}x_1 \, \mathrm{d}x_2 \ldots \mathrm{d}x_k,$$

where the domain of integration is defined by $V = \{x_1 \geq x_2 \geq \cdots \geq x_k \geq 0, x_1 + \cdots + x_k \leq 1\}$. For example, for $k = 2$, $n_1 = 3$, $n_2 = 5$, one finds the value

$$I = \int_{x_2=0}^{1/2} \int_{x_1=x_2}^{1-x_2} x_1^3 x_2^5 \, \mathrm{d}x_1 \, \mathrm{d}x_2 = \frac{13}{258048}.$$

Exercise 23. Given that I is a rational number, develop an algorithm using rational reconstruction and/or the Chinese Remainder Theorem to calculate I. Implement the algorithm in Sage, and apply it to the case $[n_1, \ldots, n_{31}] =$

$$[9, 7, 8, 11, 6, 3, 7, 6, 6, 4, 3, 4, 1, 2, 2, 1, 1, 1, 2, 0, 0, 0, 3, 0, 0, 0, 0, 1, 0, 0, 0].$$

7

Polynomials

This chapter will discuss univariate polynomials and related objects, mainly rational functions and formal power series. We will first see how to perform with Sage some transformations like the Euclidean division of polynomials, factorisation into irreducible polynomials, root isolation, or partial fraction decomposition. All these transformations will take into account the ring or field where the polynomial coefficients live: Sage enables us to compute in polynomial rings $A[x]$, in their quotient $A[x]/\langle P(x)\rangle$, in fraction fields $K(x)$ or in formal power series rings $A[[x]]$ for a whole set of base rings.

Operations on polynomials also have some unexpected applications. How would you automatically guess the next term of the sequence

$$1, 1, 2, 3, 8, 11, 39...?$$

For example, you could use the Padé approximation of rational functions, presented in Section 7.4.3! How could you get a series expansion of the solutions of the equation $e^{xf(x)} = f(x)$? An answer can be found in Section 7.5.3.

We assume in general that the reader is used to playing with polynomials and rational functions at the first year university level. However, we will discuss more advanced subjects. How to prove that the solutions of the equation $x^5 - x - 1$ cannot be expressed by radicals? It suffices to compute its Galois group, as explained in Section 7.3.4. The corresponding parts are not used elsewhere in this book, and the reader may skip them. Finally, this chapter gives a few examples with algebraic and p-adic numbers.

Here we will focus on polynomials with one variable, called univariate polynomials. Multivariate polynomials are discussed in Chapter 9.

Playing with polynomial rings, $R = A[x]$	
construction (dense repr.)	`R.<x> = A[]` *or* `R.<x> = PolynomialRing(A)`
e.g. $\mathbb{Z}[x]$, $\mathbb{Q}[x]$, $\mathbb{R}[x]$, $\mathbb{Z}/n\mathbb{Z}[x]$	`ZZ['x']`, `QQ['x']`, `RR['x']`, `Integers(n)['x']`
construction (sparse repr.)	`R.<x> = PolynomialRing(A, sparse=True)`
accessing the base ring A	`R.base_ring()`
accessing the variable x	`R.gen()` *or* `R.0`
tests (integral, noetherian...)	`R.is_integral_domain()`, `R.is_noetherian()`, ...

TABLE 7.1 – Polynomial rings.

7.1 Polynomial Rings

7.1.1 Introduction

We have seen in Chapter 2 how to perform computations on *symbolic expressions*, elements of the "symbolic ring" `SR`. Some methods available for these expressions, for example `degree`, are suited for polynomials:

```
sage: x = var('x'); p = (2*x+1)*(x+2)*(x^4-1)
sage: print("{} is of degree {}".format(p, p.degree(x)))
(x^4 - 1)*(2*x + 1)*(x + 2) is of degree 6
```

In some computer algebra systems, like Maple or Maxima, representing polynomials as particular symbolic expressions is the usual way to play with them. Like Axiom, Magma or MuPAD, Sage also lets you manipulate polynomials in a more algebraic way, and "knows" how to compute in rings like $\mathbb{Q}[x]$ or $\mathbb{Z}/4\mathbb{Z}[x, y, z]$.

Hence, to reproduce the above example in a well-defined polynomial ring, we assign to the Python variable `x` the *unknown of the polynomial ring in x with rational coefficients*, given by `polygen(QQ, 'x')`, instead of the *symbolic variable x* returned[1] by `var('x')`:

```
sage: x = polygen(QQ, 'x'); p = (2*x+1)*(x+2)*(x^4-1)
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
```

We notice that the polynomial is automatically expanded. The "algebraic" polynomials are always represented in normal form. This is a crucial difference with respect to the polynomials in `SR`. In particular, when two algebraic polynomials are mathematically equal, their computer representation is the same, and a comparison coefficient by coefficient is enough to check their equality.

The available functions on algebraic polynomials are much wider and more efficient that those on (polynomial) symbolic expressions.

7.1.2 Building Polynomial Rings

Polynomials in Sage, like many other algebraic objects, generally have coefficients in a commutative ring. This is the point of view of this book; however, most of

[1] A little difference here: while `var('x')` is equivalent to `x = var('x')` in interactive use, `polygen(QQ, 'x')` alone does not change the value of the Python variable `x`.

the examples will have coefficients in a field. In the whole chapter, the letters A and K respectively correspond to a commutative ring and to a field.

The first step to perform a computation in an algebraic structure R is often to build R itself. We build $\mathbb{Q}[x]$ with

```
sage: R = PolynomialRing(QQ, 'x')
sage: x = R.gen()
```

The `'x'` on the first line is a character string, which is the name of the indeterminate, or *generator* of the ring. The `x` on the second line is a Python variable in which one stores the generator; using the same name makes the code easier to read. The object stored in the variable `x` represents the polynomial $x \in \mathbb{Q}[x]$. Its parent (the *parent* of a Sage object is the algebraic structure "from which it comes", see §5.1) is the ring `QQ['x']`:

```
sage: x.parent()
Univariate Polynomial Ring in x over Rational Field
```

The polynomial $x \in \mathbb{Q}[x]$ is considered different from $x \in A[x]$ for a base ring $A \neq \mathbb{Q}$, and also different from those, like $t \in \mathbb{Q}[t]$, whose indeterminate has a different name.

The expression `PolynomialRing(QQ, 't')` might also be written `QQ['t']`. We often combine this abbreviation with the construction `S.<g> = ...`, which simultaneously assigns a structure to the variable `S` and its generator to the variable `g`. The construction of the ring $\mathbb{Q}[x]$ and of its indeterminate then reduces to `R.<x> = QQ[]`. The form `x = polygen(QQ, 'x')` seen above is equivalent to

```
sage: x = PolynomialRing(QQ, 'x').gen()
```

Let us mention that we can choose between several memory representations when we construct a polynomial ring. The differences between representations are discussed in §7.6.

Exercise 24 (Variables and indeterminates)**.**

1. How would you define `x` and `y` to obtain the following results?

```
sage: x^2 + 1
y^2 + 1
sage: (y^2 + 1).parent()
Univariate Polynomial Ring in x over Rational Field
```

2. After the instructions

```
sage: Q.<x> = QQ[]; p = x + 1; x = 2; p = p + x
```

what is the value of `p`?

Polynomials with polynomial coefficients

In Sage, we can define polynomial rings with coefficients in any commutative ring, including another polynomial ring. But beware that rings $A[x][y]$ constructed this way differ from the true polynomial rings with several variables like $A[x, y]$. The latter, presented in Chapter 9, are better suited for usual computations. Indeed, working in $A[x][y][\dots]$ introduces an asymmetry between the variables.

However, in some cases we precisely want to have one main variable, and the other variables as parameters. The `polynomial` method of multivariate polynomials allows us to isolate one variable, more or less like the `collect` method of symbolic expressions. For example, to compute the reciprocal of a given polynomial with respect to one of its variables:

```
sage: R.<x,y,z,t> = QQ[]; p = (x+y+z*t)^2
sage: p.polynomial(t).reverse()
(x^2 + 2*x*y + y^2)*t^2 + (2*x*z + 2*y*z)*t + z^2
```

Here, `p.polynomial(t)` creates a univariate polynomial in the variable `t` and with coefficients in `QQ[x,y,z]`, to which we then apply the `reverse` method.

The other conversions between $A[x, y, \dots]$ and $A[x][y][\dots]$ work as expected:

```
sage: x = polygen(QQ); y = polygen(QQ[x], 'y')
sage: p = x^3 + x*y + y + y^2; p
y^2 + (x + 1)*y + x^3
sage: q = QQ['x,y'](p); q
x^3 + x*y + y^2 + y
sage: QQ['x']['y'](q)
y^2 + (x + 1)*y + x^3
```

7.1.3 Polynomials

Creation and Basic Arithmetic. After the instruction `R.<x> = QQ[]`, the expressions constructed from `x` and rational constants with operations `+` and `*` are elements of $\mathbb{Q}[x]$. For example, in `p = x + 2`, Sage automatically determines that the values of the variable `x` and the integer 2 can both be seen as elements of $\mathbb{Q}[x]$. The addition routine of polynomials in $\mathbb{Q}[x]$ is thus called; it builds and returns the polynomial $x + 2 \in \mathbb{Q}[x]$.

Another way to build a polynomial is to enumerate its coefficients:

```
sage: def rook_polynomial(n, var='x'):
....:     return ZZ[var]([binomial(n, k)^2 * factorial(k)
....:                                     for k in (0..n) ])
```

The above function constructs polynomials whose coefficient of x^k is the number of ways to put k rooks on an $n \times n$ chessboard, so that two rooks cannot capture each other; this explains the name of the function. The parentheses after `ZZ[var]` force the conversion of a given object into an element of this ring. The

Accessing data, syntactic operations	
indeterminate x	`p.variables()`, `p.variable_name()`
coefficient of x^k	`p[k]`
leading coefficient	`p.leading_coefficient()`
degree	`p.degree()`
list of coefficients	`p.list()` *or* `p.coefficients(sparse=False)`
list of *non-zero* coefficients	`p.coefficients()`
dictionary degree $\mapsto$ coefficient	`p.dict()`
tests (monic, constant...)	`p.is_monic()`, `p.is_constant()`, ...
Basic arithmetic	
operations $p+q$, $p-q$, $p\times q$, p^k	`p + q`, `p - q`, `p * q`, `p^k`
substitution $x := a$	`p(a)` *or* `p.subs(a)`
derivative	`p.derivative()` *or* `p.diff()`
Transformations	
transformation of coefficients	`p.map_coefficients(f)`
change of base ring $A[x] \to B[x]$	`p.change_ring(B)` *or* `B['x'](p)`
reciprocal polynomial	`p.reverse()`

TABLE 7.2 – Basic operations on polynomials $p, q \in A[x]$.

conversion of a list $[a_0, a_1, \dots]$ into an element of `ZZ['x']` yields the polynomial $a_0 + a_1\, x + \cdots \in \mathbb{Z}[x]$.

Global View on Polynomial Operations. The elements of a polynomial ring are represented by Python objects from the class `Polynomial`, or from derived classes. The main operations[2] available for these objects are summarised in Tables 7.2 to 7.5. For example, we query the degree of a polynomial with the `degree` method. Similarly, `p.subs(a)` or simply `p(a)` yields the value of p at the point a, but also computes the composition $p \circ a$ when a itself is a polynomial, and more generally evaluates a polynomial of $A[x]$ at an element of an A-algebra:

```
sage: p = R.random_element(degree=4) # a random polynomial
sage: p
-4*x^4 - 52*x^3 - 1/6*x^2 - 4/23*x + 1
sage: p.subs(x^2)
-4*x^8 - 52*x^6 - 1/6*x^4 - 4/23*x^2 + 1
sage: p.subs(matrix([[1,2],[3,4]]))
[-375407/138  -273931/69]
[ -273931/46  -598600/69]
```

We will come back to the content of the last two tables in Sections 7.2.1 and 7.3.

[2] There are many other operations. Those tables omit functions which are too advanced, some specialised variants of methods we mention, and numerous methods common to all ring elements, and even to all Sage objects, which have no particular interest on polynomials. Note however that some specialised methods (for example `p.rescale(a)`, equivalent to `p(a*x)`) are often more efficient than more general methods that could replace them.

Change of Ring. The exact list of available operations, their meaning and their efficiency heavily depend on the base ring. For example, the polynomials in `GF(p)['x']` have a method `small_roots` which returns their small roots with respect to the characteristic p; those in `QQ['x']` do not have such a method, since it makes no sense. The `factor` method exists for all polynomials, but raises an exception `NotImplementedError` for polynomials with coefficients in `SR` or in $\mathbb{Z}/4\mathbb{Z}$. This exception means that this operation is not available in Sage for this kind of object, despite having a mathematical meaning.

It is very useful to be able to juggle the different rings of coefficients on which we might consider a given polynomial. Applied to a polynomial in $A[x]$, the method `change_ring(B)` returns its image in $B[x]$, when a natural method to convert the coefficients exists. The conversion is often given by a canonical morphism from A to B: in particular, `change_ring` might be used to extend the base ring to gain additional algebraic properties. Here for example, the polynomial p is irreducible over the rationals, but it factors on $\mathbb{R}$:

```
sage: x = polygen(QQ)
sage: p = x^2 - 16*x + 3
sage: p.factor()
x^2 - 16*x + 3
sage: p.change_ring(RDF).factor()
(x - 15.810249675906654) * (x - 0.18975032409334563)
```

The `RDF` domain is that of "machine floating-point numbers", and is discussed in Chapter 11. The obtained factorisation is approximate; it is not enough to recover the original polynomial. To represent real roots of polynomials with integer coefficients in a way that enables exact computations, we use the domain `AA` of real algebraic numbers. We will see some examples in the following sections.

The same method `change_ring` allows to reduce a polynomial in $\mathbb{Z}[x]$ modulo a prime number:

```
sage: p.change_ring(GF(3))
x^2 + 2*x
```

Conversely, if $B \subset A$ and if the coefficients of p are in fact in B, we also call `change_ring` to recover p in $B[x]$.

Iteration. More generally, one often needs to apply a given transformation to all coefficients of a polynomial. The method `map_coefficients` is designed for this. Applied to a polynomial $p \in A[x]$ with parameter a function f, it returns the polynomial obtained by applying f to all *non-zero* coefficients of p. In general, f is an anonymous function defined using the `lambda` construction (see §3.3.2). Here is for example how one can compute the conjugate of a polynomial with complex coefficients:

```
sage: QQi.<myI> = QQ[I]    # myI is the i of QQi, I that of SR
sage: R.<x> = QQi[]; p = (x + 2*myI)^3; p
x^3 + 6*I*x^2 - 12*x - 8*I
sage: p.map_coefficients(lambda z: z.conjugate())
```

```
x^3 - 6*I*x^2 - 12*x + 8*I
```

Here, we can also write `p.map_coefficients(conjugate)`, since `conjugate(z)` has the same effect as `z.conjugate` for $z \in \mathbb{Q}[i]$. Calling explicitly a method of the object `z` is more robust: the code then works for all objects having a `conjugate()` method, and only for those.

Operations on polynomial rings

The parents of polynomial objects, i.e., the rings $A[x]$, are themselves first class Sage objects. Let us briefly see how to use them.

A first family of methods enables us to construct particular polynomials, to draw random ones, or to enumerate families, here those of degree exactly 2 over $\mathbb{F}_2$:

```
sage: list(GF(2)['x'].polynomials(of_degree=2))
[x^2, x^2 + 1, x^2 + x, x^2 + x + 1]
```

We will call some of these methods in the examples of the next sections, to build objects on which we will work. Chapter 15 explains more generally how to enumerate finite sets with Sage.

Secondly, the system "knows" some basic facts for each polynomial ring. We can check whether a given object is a ring, if it is noetherian:

```
sage: A = QQ['x']
sage: A.is_ring() and A.is_noetherian()
True
```

or if $\mathbb{Z}$ is a sub-ring of $\mathbb{Q}[x]$, and for which values of n the ring $\mathbb{Z}/n\mathbb{Z}$ is integral:

```
sage: ZZ.is_subring(A)
True
sage: [n for n in range(20)
....:     if Integers(n)['x'].is_integral_domain()]
[0, 2, 3, 5, 7, 11, 13, 17, 19]
```

These capabilities largely rely on the Sage *category* system (see also §5.2.3). Polynomial rings belong to a number of "categories", like the category of sets, that of Euclidean rings, and many more:

```
sage: R.categories()
[Category of euclidean domains,
 Category of principal ideal domains,
 ...
 Category of sets with partial maps, Category of objects]
```

This reflects that any polynomial ring is also a set, a Euclidean domain, and so on. The system can thus automatically transfer to polynomial rings the general properties of objects from these different categories.

Divisibility and Euclidean division	
divisibility test $p \mid q$	`p.divides(q)`
multiplicity of a divisor $q^k \mid p$	`k = p.valuation(q)`
Euclidean division $p = qd + r$	`q, r = p.quo_rem(d)` *or* `q = p//d, r = p%d`
pseudo-division $a^k p = qd + r$	`q, r, k = p.pseudo_divrem(d)`
greatest common divisor	`p.gcd(q)`, `gcd([p1, p2, p3])`
least common multiple	`p.lcm(q)`, `lcm([p1, p2, p3])`
extended gcd $g = up + vq$	`g, u, v = p.xgcd(q)` *or* `xgcd(p, q)`
"Chinese remainder" $c \equiv a \bmod p$, $c \equiv b \bmod q$	`c = crt(a, b, p, q)`

Miscellaneous	
interpolation $p(x_i) = y_i$	`p = R.lagrange_polynomial([(x1,y1), ...])`
content of $p \in \mathbb{Z}[x]$	`p.content()`

TABLE 7.3 – Polynomial arithmetic.

7.2 Euclidean Arithmetic

Apart from the sum and product, the most elementary operations on polynomials are the Euclidean division and the greatest common divisor computation. The corresponding operators and methods (Table 7.3) mimic those on integers. However, quite often, these operations are hidden by an additional abstraction layer: quotient of rings (§7.2.2) where each arithmetic operation involves an implicit Euclidean division, rational functions (§7.4) whose normalisation implies some gcd computations...

7.2.1 Divisibility

Divisions. The Euclidean division works in a field, and more generally in a commutative ring when the leading coefficient of the divisor is invertible, since this coefficient is the only one from the base ring by which it is required to divide:

```
sage: R.<t> = Integers(42)[]; (t^20-1) % (t^5+8*t+7)
22*t^4 + 14*t^3 + 14*t + 6
```

When the leading coefficient is not invertible, we can still define a *pseudo Euclidean division* (pseudo-division for short): let A be a commutative ring, $p, d \in A[x]$, and a the leading coefficient of d. Then there exists two polynomials $q, r \in A[x]$, with $\deg r < \deg d$, and an integer $k \leq \deg p - \deg d + 1$ such that

$$a^k p = qd + r.$$

The pseudo-division is given by the `pseudo_divrem` method.

To perform an exact division, we also use the Euclidean quotient operator `//`. Indeed, dividing by a non-constant polynomial with `/` returns a result of type rational function (see §7.4), or fails when this makes no sense:

```
sage: ((t^2+t)//t).parent()
```

```
Univariate Polynomial Ring in t over Ring of integers modulo 42
sage: (t^2+t)/t
Traceback (most recent call last):
...
TypeError: self must be an integral domain.
```

Exercise 25. Usually, in Sage, polynomials in $\mathbb{Q}[x]$ are represented on the monomial basis $(x^n)_{n\in\mathbb{N}}$. Chebyshev polynomials T_n, defined by $T_n(\cos\theta) = \cos(n\theta)$, form a family of orthogonal polynomials and thus a basis of $\mathbb{Q}[x]$. The first Chebyshev polynomials are

```
sage: x = polygen(QQ); [chebyshev_T(n, x) for n in (0..4)]
[1, x, 2*x^2 - 1, 4*x^3 - 3*x, 8*x^4 - 8*x^2 + 1]
```

Write a function taking as input an element of $\mathbb{Q}[x]$ and returning the coefficients of its decomposition in the basis $(T_n)_{n\in\mathbb{N}}$.

Exercise 26 (Division by increasing powers)**.** Let $n \in \mathbb{N}$ and $u, v \in A[x]$, with $v(0)$ invertible. Then a unique pair (q, r) of polynomials exists in $A[x]$ with $\deg q \leq n$ such that $u = qv + x^{n+1}r$. Write a function which computes q and r by an analogue of the Euclidean division algorithm. How would you perform this computation in the easiest way, using available Sage functions?

GCD. Sage is able to compute the gcd of polynomials over a field, thanks to the Euclidean structure of $K[x]$, but also on some other rings, including the integers:

```
sage: S.<x> = ZZ[]; p = 2*(x^10-1)*(x^8-1)
sage: p.gcd(p.derivative())
2*x^2 - 2
```

We can prefer the more symmetric expression `gcd(p,q)`, which yields the same result as `p.gcd(q)`. It is though slightly less natural in Sage since it is not a general mechanism: `gcd(p,q)` calls a function of two arguments, defined manually in the source code of Sage, and which calls in turn `p.gcd`. Only some usual methods have such an associated function.

The *extended gcd*, i.e., the computation of a Bézout relation

$$g = \gcd(p, q) = ap + bq, \qquad g, p, q, a, b \in K[x]$$

is given by `p.xgcd(q)`:

```
sage: R.<x> = QQ[]; p = x^5-1; q = x^3-1
sage: print("the gcd is %s = (%s)*p + (%s)*q" % p.xgcd(q))
the gcd is x - 1 = (-x)*p + (x^3 + 1)*q
```

The `xgcd` method also exists for polynomials in `ZZ['x']`, but beware: since $\mathbb{Z}[x]$ is not a principal ideal ring, the result is in general not a Bézout relation ($ap + bq$ might be an integer multiple of the gcd)!

7.2.2 Ideals and Quotients

Ideals of $A[x]$. The ideals of polynomial rings, and the quotients by these ideals, are represented by Sage objects built from the polynomial ring by the methods `ideal` and `quo`. The product of a tuple of polynomials by a polynomial ring is interpreted as an ideal:

```
sage: R.<x> = QQ[]
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
```

We can multiply ideals, and reduce a polynomial modulo an ideal:

```
sage: J2 = R.ideal(x^5 + 2)
sage: ((3*x+5)*J1*J2).reduce(x^10)
421/81*x^6 - 502/81*x^5 + 842/81*x - 680/81
```

The reduced polynomial remains in this case an element of `QQ['x']`. Another way is to construct the quotient by an ideal and project the elements on it. The parent of the projected element is then in the quotient ring. The `lift` method of the quotient elements converts them back into the initial ring.

```
sage: B = R.quo((3*x+5)*J1*J2) # quo automatically names 'xbar' which is
sage: B(x^10)                  #   the generator of B image of x
421/81*xbar^6 - 502/81*xbar^5 + 842/81*xbar - 680/81
sage: B(x^10).lift()
421/81*x^6 - 502/81*x^5 + 842/81*x - 680/81
```

If K is a field, then the ring $K[x]$ is principal: the ideals are represented during computations by a generator, all this being an algebraic language for the operations seen in §7.2.1. Its principal advantage is that quotient rings can be easily used in new constructions, here that of $\big(\mathbb{F}_5[t]/\langle t^2+3\rangle\big)[x]$:

```
sage: R.<t> = GF(5)[]; R.quo(t^2+3)['x'].random_element()
(3*tbar + 1)*x^2 + (2*tbar + 3)*x + 3*tbar + 4
```

Sage also allows building non principal ideals like in $\mathbb{Z}[x]$, however the available operations are then limited — except in case of multivariate polynomials over a field, which are the subject of Chapter 9.

Exercise 27. We define the sequence $(u_n)_{n\in\mathbb{N}}$ with the initial conditions $u_n = n+7$ for $0 \le n < 1000$, and the linear recurrence relation

$$u_{n+1000} = 23u_{n+729} - 5u_{n+2} + 12u_{n+1} + 7u_n \qquad (n \ge 0).$$

Compute the last five digits of $u_{10^{10000}}$. *Hint:* we might look at the algorithm from §3.2.4. However, this algorithm is too expensive when the order of the recurrence is large. Introduce a clever quotient of polynomial rings to avoid this issue.

Construction of ideals and quotient rings $Q = R/J$	
ideal $\langle u, v, w\rangle$	`R.ideal(u, v, w)` *or* `(u, v, w)*R`
reduction of p modulo J	`J.reduce(p)` *or* `p.mod(J)`
quotient ring R/J, $R/\langle p\rangle$	`R.quo(J)`, `R.quo(p)`
ring whose quotient gave Q	`Q.cover_ring()`
isomorphic number field	`Q.number_field()`

Elements of $K[x]/\langle p\rangle$	
lift (section of $R \twoheadrightarrow R/J$)	`u.lift()`
minimal polynomial	`u.minpoly()`
characteristic polynomial	`u.charpoly()`
matrix	`u.matrix()`
trace	`u.trace()`

TABLE 7.4 – Ideals and quotients.

Algebraic Extensions. An important special case is the quotient of $K[x]$ by an irreducible polynomial to build an algebraic extension of K. The number fields, finite extensions of $\mathbb{Q}$, are represented by the objects `NumberField`, distinct from the quotients of `QQ['x']`. When this makes sense, the method `number_field` of a quotient of polynomial rings returns the corresponding number field. The interface of number fields, more complete than that of quotient rings, is beyond the scope of this book. The non-prime finite fields $\mathbb{F}_{p^k}$, built as algebraic extensions of the prime finite fields $\mathbb{F}_p$, are described in §6.1.

7.3 Factorisation and Roots

A third level after the elementary operations and the Euclidean arithmetic concerns the decomposition of a polynomial into a product of irreducible factors, or factorisation. It is maybe where computer algebra is the most useful!

7.3.1 Factorisation

Irreducibility Test. On the algebraic side, the simplest question about the factorisation of a polynomial is whether it is irreducible. Naturally, the answer depends on the base ring. The method `is_irreducible` tells if a polynomial is irreducible in its parent ring. For example, the polynomial $3x^2 - 6$ is irreducible over $\mathbb{Q}$, but not over $\mathbb{Z}$ (why?):

```
sage: R.<x> = QQ[]; p = 3*x^2 - 6
sage: p.is_irreducible(), p.change_ring(ZZ).is_irreducible()
(True, False)
```

Factorisation. The factorisation of an *integer* of hundreds or thousands of digits is a very hard problem. In contrast, factoring a *polynomial* of degree 1000

on $\mathbb{Q}$ or $\mathbb{F}_p$ — for small p — needs only a few seconds[3]:

```
sage: p = QQ['x'].random_element(degree=1000)
sage: %timeit p.factor()
1 loop, best of 3: 2.45 s per loop
```

Here ends the algorithmic similarity between polynomials and integers we have seen in preceding sections.

Like the irreducibility test, the factorisation is performed on the base ring. For example, the factorisation of a polynomial over the integers contains a constant part, itself split into prime factors, and a product of primitive polynomials, i.e., whose coefficients are coprime:

```
sage: x = polygen(ZZ); p = 54*x^4+36*x^3-102*x^2-72*x-12
sage: p.factor()
2 * 3 * (3*x + 1)^2 * (x^2 - 2)
```

Sage is able to factor polynomials on various rings — rational, complex (approximate), finite fields and number fields in particular:

```
sage: for A in [QQ, ComplexField(16), GF(5), QQ[sqrt(2)]]:
....:     print(str(A) + ":")
....:     print(A['x'](p).factor())
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:
(54) * (x - sqrt2) * (x + sqrt2) * (x + 1/3)^2
```

The result of a decomposition into irreducible factors is not a polynomial (since the polynomials are always in normal form, i.e., in expanded form!), but an object `f` of type `Factorization`. We obtain the `i`th factor with `f[i]`, and we get back the polynomial with `f.expand()`. The `Factorization` objects also provide methods like `gcd` and `lcm` which have the same meaning as for polynomials, but work on the factored forms.

Square-Free Decomposition. Despite its good theoretical and practical complexity, the full factorisation of a polynomial is an expensive operation. The square-free decomposition is a weaker factorisation, much easier to obtain — some gcd computations are enough — and which already brings a lot of information.

Let $p = \prod_{i=1}^{r} p_i^{m_i} \in K[x]$ be a polynomial that splits into a product of irreducible factors over a field K of characteristic zero. We say that p is *square-free* if all its factors p_i have multiplicity $m_i = 1$, i.e., if the roots of p in an

[3] On the theoretical side, we know how to factor in $\mathbb{Q}[x]$ in polynomial time, and in $\mathbb{F}_p[x]$ in probabilistic polynomial time, whereas we do not know whether integers can be factored in polynomial time.

Factorisation	
irreducibility test	`p.is_irreducible()`
factorisation	`p.factor()`
square-free factorisation	`p.squarefree_decomposition()`
square-free part $p/\gcd(p,p')$	`p.radical()`

Roots	
roots in A, in D	`p.roots()`, `p.roots(D)`
real roots	`p.roots(RR)`, `p.real_roots()`
complex roots	`p.roots(CC)`, `p.complex_roots()`
isolation of real roots	`p.roots(RIF)`, `p.real_root_intervals()`
isolation of complex roots	`p.roots(CIF)`
resultant	`p.resultant(q)`
discriminant	`p.discriminant()`
Galois group (p irreducible)	`p.galois_group()`

TABLE 7.5 – Factorisation and roots.

algebraic closure of K are simple. A *square-free decomposition* is a factorisation into a product of square-free and coprime factors:

$$p = f_1 f_2^2 \dots f_s^s \qquad \text{where} \qquad f_m = \prod_{m_i = m} p_i.$$

Hence, the square-free decomposition splits the irreducible factors of p by multiplicity. The *square-free part* $f_1 \dots f_s = p_1 \dots p_r$ of p is the polynomial with simple roots which has the same roots as p, disregarding multiplicities.

7.3.2 Root Finding

The computation of the roots of a polynomial may be performed in several ways: Do we want real or complex roots? Roots in another domain? Do we want exact or approximate roots? With or without multiplicities? In a guaranteed or heuristic way? The `roots` method of a polynomial returns by default the roots in its base ring, in the form of a list of pairs (root, multiplicity):

```
sage: R.<x> = ZZ[]; p = (2*x^2-5*x+2)^2 * (x^4-7); p.roots()
[(2, 2)]
```

With a parameter, `roots(D)` returns the roots in the domain D, here the rational roots, and approximations of the ℓ-adic roots for $\ell = 19$:

```
sage: p.roots(QQ)
[(2, 2), (1/2, 2)]
sage: p.roots(Zp(19, print_max_terms=3))
[(7 + 16*19 + 17*19^2 + ... + O(19^20), 1),
 (12 + 2*19 + 19^2 + ... + O(19^20), 1),
 (10 + 9*19 + 9*19^2 + ... + O(19^20), 2),
 (2 + O(19^20), 2)]
```

This works for a large number of domains, with more or less efficiency.

In particular, selecting for D the field of algebraic numbers `QQbar` or that of real algebraic numbers `AA` enables us to compute exactly the complex or real roots of a polynomial with rational coefficients:

```
sage: roots = p.roots(AA); roots
[(-1.626576561697786?, 1), (0.500000000000000?, 2),
(1.626576561697786?, 1), (2.000000000000000?, 2)]
```

Sage plays transparently for the user with different representations of algebraic numbers. One encodes each $\alpha \in \bar{\mathbb{Q}}$ by its minimal polynomial together with a sufficiently accurate interval to distinguish α from the other roots. Therefore, despite their output, the returned roots are not just approximate values. They can be reused in exact computations:

```
sage: a = roots[0][0]^4; a.simplify(); a
7
```

Here, we have raised the first root found to the fourth power, then forced Sage to simplify the result to make it clear it equals the integer 7.

A variant of the exact resolution is to simply *isolate* the roots, i.e., determine intervals containing exactly one root each, by giving as domain `D` that of the real intervals `RIF` or complex intervals `CIF`. Among the other useful domains in the case of a polynomial with rational coefficients, let us mention `RR`, `CC`, `RDF`, `CDF`, which all correspond to approximate numerical roots, and the number fields `QQ[alpha]`.The specific methods `real_roots`, `complex_roots` and (for some base rings) `real_root_intervals` offer additional options or give slightly different results from the `roots` method. The numerical approximation and isolation of roots is discussed in more detail in §12.2.

7.3.3 Resultant

In a unique factorisation domain, the existence of a common non-constant factor between two polynomials is characterised by the nullity of their *resultant* $\mathrm{Res}(p, q)$, which is a polynomial in their coefficients. A major advantage of the resultant compared to the gcd is that it *specialises* well under ring morphisms. For example, the polynomials $x - 12$ and $x - 20$ are coprime in $\mathbb{Z}[x]$, but the nullity of their resultant

```
sage: x = polygen(ZZ); (x-12).resultant(x-20)
-8
```

modulo n shows that they have a common factor in $\mathbb{Z}/n\mathbb{Z}$ if and only if n divides 8.

Let $p = \sum_{i=0}^{m} p_i x^i$ and $q = \sum_{i=0}^{n} q_i x^i$ be two non constant polynomials in

$A[x]$, with $p_m, q_n \neq 0$. The resultant of p and q is defined by

$$\mathrm{Res}(p,q) = \begin{vmatrix} p_m & \cdots & & \cdots & p_0 & & \\ & \ddots & & & & \ddots & \\ & & p_m & \cdots & \cdots & & p_0 \\ q_n & \cdots & & q_0 & & & \\ & \ddots & & & \ddots & & \\ & & \ddots & & & \ddots & \\ & & & & q_n & \cdots & q_0 \end{vmatrix}. \tag{7.1}$$

It is the determinant, in suitable bases, of the linear map

$$\begin{array}{rcl} A_{n-1}[x] \times A_{m-1}[x] & \to & A_{m+n-1}[x] \\ u, v & \mapsto & up + vq \end{array}$$

where $A_k[x] \subset A[x]$ is the sub-module of polynomials of degree at most k. If p and q split into linear factors, their resultant may also be expressed in terms of differences of their roots:

$$\mathrm{Res}(p,q) = p_m^n q_n^m \prod_{i,j} (\alpha_i - \beta_j), \quad \begin{cases} p = p_m(x-\alpha_1)\dots(x-\alpha_m) \\ q = q_n(x-\beta_1)\dots(x-\beta_n). \end{cases}$$

The specialisation property mentioned above follows from the definition (7.1): if $\varphi : A \to A'$ is a ring morphism, the application of which to p and q keeps their degrees unchanged, i.e., such that $\varphi(p_m) \neq 0$ and $\varphi(q_n) \neq 0$, then we have

$$\mathrm{Res}(\varphi(p), \varphi(q)) = \varphi(\mathrm{Res}(p,q)).$$

As a consequence, $\varphi(\mathrm{Res}(p,q))$ vanishes when $\varphi(p)$ and $\varphi(q)$ share a common factor. We have seen above an example of this phenomenon, with φ the canonical projection from $\mathbb{Z}$ to $\mathbb{Z}/n\mathbb{Z}$.

The most common usage of the resultant concerns the case where the base ring itself is a polynomial ring: $p, q \in A[x]$ with $A = K[a_1, \dots, a_k]$. In particular, given $\alpha_1, \dots, \alpha_k \in K$, let us consider the specialisation

$$\begin{array}{rccl} \varphi : & B[a_1, \dots, a_k] & \to & K \\ & q(a_1, \dots, a_k) & \mapsto & q(\alpha_1, \dots, \alpha_k). \end{array}$$

We see that the resultant $\mathrm{Res}(p,q)$ vanishes at $(\alpha_1, \dots, \alpha_k)$ if and only if the specialisations $\varphi(p), \varphi(q) \in K[x]$ share a common factor, *assuming that* one of the leading terms of p and q does not vanish in $(\alpha_1, \dots, \alpha_k)$.

For example, the discriminant of $p \in \mathbb{Q}[x]$ of degree m is defined by

$$\mathrm{disc}(p) = (-1)^{m(m-1)/2} \mathrm{Res}(p, p')/p_m.$$

This definition generalises the classical discriminants of degree two and three polynomials:

```
sage: R.<a,b,c,d> = QQ[]; x = polygen(R); p = a*x^2+b*x+c
sage: p.resultant(p.derivative())
-a*b^2 + 4*a^2*c
sage: p.discriminant()
b^2 - 4*a*c
sage: (a*x^3 + b*x^2 + c*x + d).discriminant()
b^2*c^2 - 4*a*c^3 - 4*b^3*d + 18*a*b*c*d - 27*a^2*d^2
```

Since the discriminant of p is, up to a normalisation, the resultant of p and its derivative, it vanishes if and only if p has a multiple root in $\mathbb{C}$.

7.3.4 Galois Group

The Galois group of an irreducible polynomial $p \in \mathbb{Q}[x]$ is an algebraic object which describes some of the "symmetries" of the roots of p. It is a central object in the theory of algebraic equations. In particular, the equation $p(x) = 0$ is solvable by radicals — i.e., its roots can be expressed from coefficients of p using the four operations and the nth root — if and only if the Galois group of p is *solvable*.

Sage allows the computation of the Galois group of polynomials with rational coefficients of moderate degree, and performs several operations on the obtained groups. Both Galois theory and the group theory functionalities of Sage go beyond the scope of this book. Let us simply apply without more explanations Galois' theorem on the solvability by radicals. The following computation[4] shows that the roots of $x^5 - x - 1$ cannot be expressed using radicals:

```
sage: x = polygen(QQ); G = (x^5 - x - 1).galois_group(); G
Transitive group number 5 of degree 5
sage: G.is_solvable()
False
```

It is one of the simplest examples of this situation, since polynomials of degree less than or equal to 4 are always solvable by radicals, as well as obviously those of the form $x^5 - a$. By looking at the generators of G seen as a permutation group, we recognise that $G \simeq \mathfrak{S}_5$, which can be easily verified:

```
sage: G.gens()
[(1,2,3,4,5), (1,2)]
sage: G.is_isomorphic(SymmetricGroup(5))
True
```

7.4 Rational Functions

7.4.1 Construction and Basic Properties

The division of two polynomials (on an integral ring) produces a rational function. Its parent is the fraction field of the polynomial ring, obtained with `Frac(R)`:

[4] This computation requires a table of finite groups which is not in the default installation of Sage, but we can upload and automatically install it with the command `sage -i database_gap` (it might be needed to restart Sage after the installation).

Rational functions	
fraction field $K(x)$	`Frac(K['x'])`
numerator	`r.numerator()`
denominator	`r.denominator()`
simplification (modifies `r`)	`r.reduce()`
partial fraction decomposition	`r.partial_fraction_decomposition()`
rational reconstruction of s mod m	`s.rational_reconstruct(m)`

Truncated power series	
ring $A[[t]]$	`PowerSeriesRing(A, 'x', default_prec=n)`
ring $A((t))$	`LaurentSeriesRing(A, 'x', default_prec=n)`
coefficient $[x^k]\, f(x)$	`f[k]`
truncation	`x + O(x^n)`
precision	`f.prec()`
derivative, antiderivative (vanishes at 0)	`f.derivative()`, `f.integral()`
usual operations $\sqrt{f}$, $\exp f$, ...	`f.sqrt()`, `f.exp()`, ...
reciprocal ($f \circ g = g \circ f = x$)	`g = f.reverse()`
solution of $y' = ay + b$	`a.solve_linear_de(precision, b)`

TABLE 7.6 – Objects constructed from polynomials.

```
sage: x = polygen(RR); r = (1 + x)/(1 - x^2); r.parent()
Fraction Field of Univariate Polynomial Ring in x over Real
Field with 53 bits of precision
sage: r
(x + 1.00000000000000)/(-x^2 + 1.00000000000000)
```

We see that the simplification is not automatic. This is because `RR` is an *inexact* ring, i.e., its elements are approximations of mathematical objects. The `reduce` method puts the fraction in reduced form. It does not return a new object, but modifies the existing fraction:

```
sage: r.reduce(); r
1.00000000000000/(-x + 1.00000000000000)
```

On an exact ring, in contrast, rational functions are automatically reduced.

The operations on rational functions are analogous to those on polynomials. Those having a meaning in both cases (substitution, derivative, factorisation...) may be used in the same manner. Table 7.6 enumerates some other useful methods. The partial fraction decomposition and the rational reconstruction deserve some explanations.

7.4.2 Partial Fraction Decomposition

Sage computes the partial fraction decomposition of a rational function a/b in `Frac(K['x'])` from the factorisation of b in `K['x']`. It is therefore the partial fraction decomposition on K. The result contains a polynomial part p and a list of rational functions whose denominators are powers of irreducible factors of b:

```
sage: R.<x> = QQ[]; r = x^10 / ((x^2-1)^2 * (x^2+3))
sage: poly, parts = r.partial_fraction_decomposition()
sage: poly
x^4 - x^2 + 6
sage: for part in parts: part.factor()
(17/32) * (x - 1)^-1
(1/16) * (x - 1)^-2
(-17/32) * (x + 1)^-1
(1/16) * (x + 1)^-2
(-243/16) * (x^2 + 3)^-1
```

We have thus obtained the partial fraction decomposition on the rationals

$$r = \frac{x^{10}}{(x^2-1)^2(x^2+3)} = x^4 - x^2 + 6 + \frac{\frac{17}{32}}{x-1} + \frac{\frac{1}{16}}{(x-1)^2} - \frac{\frac{17}{32}}{x+1} + \frac{\frac{1}{16}}{(x+1)^2} - \frac{\frac{243}{16}}{x^2+3}.$$

This is also clearly the partial fraction decomposition of r on the real numbers.

However, on the complex numbers, the denominator of the last term is not irreducible, hence the rational function can be further decomposed. We can compute the partial fraction decomposition on the complex numbers numerically:

```
sage: C = ComplexField(15)
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)
    ,
4.385*I/(x - 1.732*I), (-4.385*I)/(x + 1.732*I),
(-0.5312)/(x + 1.000), 0.06250/(x^2 + 2.000*x + 1.000)])
```

We obtain the exact decomposition on $\mathbb{C}$ in the same manner, by replacing `C` by `QQbar`. Doing the computation on `AA`, we would get the decomposition on the reals, even when all real roots of the denominator are not rational.

7.4.3 Rational Reconstruction

As for integers in §6.1.3, the rational reconstruction also exists for polynomials with coefficients in $A = \mathbb{Z}/n\mathbb{Z}$. Given $m, s \in A[x]$, the command

```
sage: s.rational_reconstruct(m, dp, dq)
```

computes when possible polynomials $p, q \in A[x]$ such that

$$qs \equiv p \mod m, \qquad \deg p \le d_p, \quad \deg q \le d_q.$$

For simplicity, let us restrict ourselves to the case where n is prime. Such a relation with q and m coprime implies $p/q = s$ in $A[x]/\langle m\rangle$, which explains the "rational reconstruction" name.

The rational reconstruction problem translates into a linear system on the coefficients of p and q, and a simple dimension argument shows that a non-trivial solution exists as soon as $d_p + d_q \ge \deg m - 1$. A solution with q and m coprime does not always exist (for example, the solutions of $p \equiv qx \mod x^2$ with $\deg p \le 0$, $\deg q \le 1$ are the constant multiples of $(p, q) = (0, x)$), but `rational_reconstruct` looks rather for solutions q coprime to m.

Padé Approximants. The case $m = x^n$ is called Padé approximant. A Padé approximant of type $(k, n-k)$ of a formal power series $f \in K[[x]]$ is a rational function $p/q \in K(x)$ such that $\deg p \leq k-1$, $\deg q \leq n-k$, $q(0) = 1$, and $p/q = f + O(x^n)$. We then have $p/q \equiv f \bmod x^n$.

Let us start with a symbolic example. The following commands compute a Padé approximant of the series $f = \sum_{i=0}^{\infty} (i+1)^2 x^i$ with coefficients in $\mathbb{Z}/101\mathbb{Z}$:

```
sage: A = Integers(101); R.<x> = A[]
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, 1, 3); num/den
(100*x + 100)/(x^3 + 98*x^2 + 3*x + 100)
```

By expanding back into power series the rational function found, we see that not only the terms correspond up the term in x^5, but even the next term "is correct"!

```
sage: S = PowerSeriesRing(A, 'x', 7); S(num)/S(den)
1 + 4*x + 9*x^2 + 16*x^3 + 25*x^4 + 36*x^5 + 49*x^6 + O(x^7)
```

Indeed, f itself is a rational function: we have $f = (1+x)/(1-x)^3$. The truncated expansion `f6`, together with bounds on the degrees of the numerator and denominator, is enough to represent it without any ambiguity. From this point of view, the computation of Padé approximants is the converse of the series expansion of power series: it allows us to go back from this alternative representation to the usual one as quotient of two polynomials.

An Analytic Example. Historically, Padé approximants do not come from this kind of symbolic reasoning, but from the approximation theory of analytic functions. Indeed, the Padé approximants of the series expansion of an analytic function often approximate the function better than series truncations. When the degree of the denominator is large enough, Padé approximants can even give good approximations outside the convergence disc of the series. We sometimes say that they "swallow the poles". Figure 7.1, which shows the convergence of the approximants of type $(2k, k)$ of the tangent function around 0, illustrates this phenomenon.

Although `rational_reconstruct` is restricted to polynomials on $\mathbb{Z}/n\mathbb{Z}$, it is possible to use it to compute Padé approximants with rational coefficients, and obtain that figure. The simplest way is to first perform the rational reconstruction modulo a large enough prime:

```
sage: x = var('x'); s = tan(x).taylor(x, 0, 20)
sage: p = previous_prime(2^30); ZpZx = Integers(p)['x']
sage: Qx = QQ['x']
```

```
sage: num, den = ZpZx(s).rational_reconstruct(ZpZx(x)^10,4,5)
sage: num/den
(1073741779*x^3 + 105*x)/(x^4 + 1073741744*x^2 + 105)
```

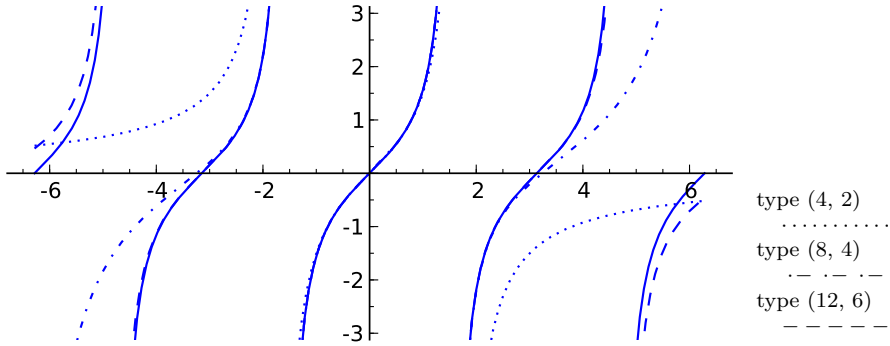


FIGURE 7.1 – The tangent function and some Padé approximants on $[-2\pi, 2\pi]$.

then to lift the solution found. The following function lifts an element a from $\mathbb{Z}/p\mathbb{Z}$ into an integer of absolute value at most $p/2$.

```
sage: def lift_sym(a):
....:     m = a.parent().defining_ideal().gen()
....:     n = a.lift()
....:     if n <= m // 2: return n
....:     else: return n - m
```

We then get:

```
sage: Qx(map(lift_sym, num))/Qx(map(lift_sym, den))
(-10*x^3 + 105*x)/(x^4 - 45*x^2 + 105)
```

When the wanted coefficients are too large for this technique, we can perform the computation modulo several primes, and apply the "Chinese Remainder Theorem" to obtain a solution with integer coefficients, as explained in §6.1.4. Another possibility is to compute a recurrence relation with constant coefficients which is satisfied by the series coefficients. This computation is almost equivalent to a Padé approximant (see Exercise 28), but the Sage function `berlekamp_massey` is able to perform it on any field.

Let us make the preceding computation more automatic, by writing a function which directly computes the approximant with rational coefficients, under favorable assumptions:

```
sage: def mypade(pol, n, k):
....:     x = ZpZx.gen();
....:     n,d = ZpZx(pol).rational_reconstruct(x^n, k-1, n-k)
....:     return Qx(map(lift_sym, n))/Qx(map(lift_sym, d))
```

It then suffices to call `plot` on the results of this function (converted into elements of `SR`, since `plot` is not able to draw directly the graph of an "algebraic" rational function) to obtain the graph of Figure 7.1:

```
sage: add(
....:     plot(expr, -2*pi, 2*pi, ymin=-3, ymax=3,
....:             linestyle=sty, detect_poles=True, aspect_ratio=1)
....:     for (expr, sty) in [
....:         (tan(x), '-'),
....:         (SR(mypade(s, 4,  2)), ':' ),
....:         (SR(mypade(s, 8,  4)), '-.'),
....:         (SR(mypade(s, 12, 6)), '--') ])
```

The following exercises demonstrate two other classical applications of the rational reconstruction.

Exercise 28. 1. Show that if $(u_n)_{n\in\mathbb{N}}$ satisfies a linear recurrence with constant coefficients, then the power series $\sum_{n\in\mathbb{N}} u_n z^n$ is a rational function. How would you interpret the numerator and denominator?

2. Guess the next terms of the sequence

$$1, 1, 2, 3, 8, 11, 34, 39, 148, 127, 662, 339, 3056, 371, 14602, -4257, \ldots,$$

by using `rational_reconstruct`. Find again the result with the `berlekamp_massey` function.

Exercise 29 (Cauchy interpolation)**.** Find a rational function $r = p/q \in \mathbb{F}_{17}(x)$ such that $r(0) = -1$, $r(1) = 0$, $r(2) = 7$, $r(3) = 5$, with p of minimal degree.

7.5 Formal Power Series

A formal power series is a power series considered as a simple sequence of coefficients, without considering convergence. More precisely, if A is a commutative ring, we call formal power series of indeterminate x with coefficients in A the formal sums $\sum_{n=0}^{\infty} a_n x^n$ where (a_n) is any sequence of elements of A. Together with the natural addition and multiplication operations

$$\sum_{n=0}^{\infty} a_n x^n + \sum_{n=0}^{\infty} b_n x^n = \sum_{n=0}^{\infty} (a_n + b_n) x^n,$$
$$\Big(\sum_{n=0}^{\infty} a_n x^n\Big)\Big(\sum_{n=0}^{\infty} b_n x^n\Big) = \sum_{n=0}^{\infty} \Big(\sum_{i+j=n} a_i b_j\Big) x^n,$$

the formal power series constitute a ring named $A[[x]]$.

In a computer algebra system, these series are useful to represent analytic functions for which we have no closed form. As always, the computer performs some computations, but it is the user's responsibility to give them a mathematical meaning. In particular, she/he should make sure that the considered series are convergent (if needed).

Formal power series also appear frequently in combinatorics, in the form of generating series. We will see such an example in §15.1.2.

7.5.1 Operations on Truncated Power Series

The ring $\mathbb{Q}[[x]]$ of formal power series is constructed by

```
sage: R.<x> = PowerSeriesRing(QQ)
```

or in short `R.<x> = QQ[[]]`[5]. The elements of `A[['x']]` are truncated power series, i.e., objects of the form

$$f = f_0 + f_1\, x + \cdots + f_{n-1}\, x^{n-1} + O(x^n).$$

They play the role of approximations of infinite "mathematical" series, much like elements of `RR` are approximations of real numbers. The `A[['x']]` ring is thus an inexact ring.

Each series has its own order of truncation[6] and the precision automatically follows through computations:

```
sage: R.<x> = QQ[[]]
sage: f = 1 + x + O(x^2); g = 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)
```

Series with infinite precision do exist, they correspond exactly to polynomials:

```
sage: (1 + x^3).prec()
+Infinity
```

A default precision is used when it is necessary to truncate an exact result. It is given at the ring creation, or afterwards with the `set_default_prec` method:

```
sage: R.<x> = PowerSeriesRing(Reals(24), default_prec=4)
sage: 1/(1 + RR.pi() * x)^2
1.00000 - 6.28319*x + 29.6088*x^2 - 124.025*x^3 + O(x^4)
```

As a consequence of the above, it is not possible to test the mathematical equality between two series. This is an important difference between these objects and the other classes of objects seen in this chapter. Sage thus considers two elements of `A[['x']]` as equal as soon as they match up to the *smallest* of their precisions:

```
sage: R.<x> = QQ[[]]
sage: 1 + x + O(x^2) == 1 + x + x^2 + O(x^3)
True
```

Warning: this implies that the test `O(x^2) == 0` returns true.

The basic arithmetic operations on series work as for polynomials. We also have some usual functions, for example `f.exp()` when $f(0) = 0$, as well as the

[5] Or from $\mathbb{Q}[x]$, by `QQ['x'].completion('x')`.

[6] In some sense, this is the main difference between a polynomial modulo x^n and a series truncated at order n: the operations on these two objects are analogous, but the elements of $A[[x]]/\langle x^n \rangle$ have all the same "precision".

derivative and antiderivative functions. Hence, an asymptotic expansion when $x \to 0$ of

$$\frac{1}{x^2} \exp\Big(\int_0^x \sqrt{\frac{1}{1+t}}\, \mathrm{d}t\Big)$$

is given by

```
sage: (1/(1+x)).sqrt().integral().exp() / x^2 + O(x^4)
x^-2 + x^-1 + 1/4 + 1/24*x - 1/192*x^2 + 11/1920*x^3 + O(x^4)
```

Here, only terms up to x^3 appear in the result, since `+ O(x^4)` explicitly asks to truncate to order 4. However, the intermediate computations are performed to the default precision 20, which we can check by omitting the `O(x^4)` term. To get even more terms, we can increase the precision of intermediate computations.

This example also demonstrates that if $f, g \in K[[x]]$ and $g(0) = 0$, the quotient f/g yields an object of type *formal Laurent series*. Contrary to the Laurent series in complex analysis, of the form $\sum_{n=-\infty}^{\infty} a_n x^n$, the formal Laurent series are sums of the form $\sum_{n=-N}^{\infty} a_n x^n$, with a finite number of terms of negative exponent. This restriction is mandatory for the product of two formal series: without it, each product coefficient would be the sum of an infinite series.

7.5.2 Solutions of an Equation: Series Expansions

Given a differential equation whose exact solutions are too complex to compute or to deal with, or simply which does not admit a closed-form solution, an alternative is often to look for solutions in the form of series expansions. We usually first determine solutions of the equation in the space of formal power series, and if necessary we conclude using a convergence argument that the constructed series solutions make sense analytically. Sage may be of great help for the first step.

Let us consider for example the differential equation

$$y'(x) = \sqrt{1+x^2}\, y(x) + \exp(x), \qquad y(0) = 1.$$

This equation has a unique formal power series solution, whose first terms might be computed by

```
sage: (1+x^2).sqrt().solve_linear_de(prec=6, b=x.exp())
1 + 2*x + 3/2*x^2 + 5/6*x^3 + 1/2*x^4 + 7/30*x^5 + O(x^6)
```

Moreover, Cauchy's theorem on the existence of solutions to linear differential equations with analytic coefficients ensures that this series converges for $|x| < 1$: its sum thus provides an analytic solution on the complex unit disc.

This approach is not limited to differential equations. The functional equation $e^{xf(x)} = f(x)$ is more complex, at least since it is not linear. Nevertheless, this is a fixed-point equation, we can try to refine a (formal) solution iteratively:

```
sage: S.<x> = PowerSeriesRing(QQ, default_prec=5)
sage: f = S(1)
sage: for i in range(5):
....:     f = (x*f).exp()
....:     print(f)
```

```
1 + x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + O(x^5)
1 + x + 3/2*x^2 + 5/3*x^3 + 41/24*x^4 + O(x^5)
1 + x + 3/2*x^2 + 8/3*x^3 + 101/24*x^4 + O(x^5)
1 + x + 3/2*x^2 + 8/3*x^3 + 125/24*x^4 + O(x^5)
1 + x + 3/2*x^2 + 8/3*x^3 + 125/24*x^4 + O(x^5)
```

What happens here? The solutions of $e^{xf(x)} = f(x)$ in $\mathbb{Q}[[x]]$ are the fixed points of the transform $\Phi : f \mapsto e^{xf}$. If a sequence of iterates of the form $\Phi^n(a)$ converges, its limit is necessarily a solution to the equation. Conversely, let us write $f(x) = \sum_{n=0}^{\infty} f_n\, x^n$, and let us expand in series both sides:

$$\begin{aligned} \sum_{n=0}^{\infty} f_n\, x^n &= \sum_{k=0}^{\infty} \frac{1}{k!} \Big(x \sum_{j=0}^{\infty} f_j\, x^j \Big)^k \\ &= \sum_{n=0}^{\infty} \Big(\sum_{k=0}^{\infty} \frac{1}{k!} \sum_{\substack{j_1,\dots,j_k \in \mathbb{N} \\ j_1+\cdots+j_k = n-k}} f_{j_1} f_{j_2} \cdots f_{j_k} \Big)\, x^n. \end{aligned} \tag{7.2}$$

Ignoring the details of the formula, the important fact is that f_n might be computed from the preceding coefficients $f_0, \dots, f_{n-1}$, as we see by isolating the coefficients on both sides. Hence, each iteration of Φ yields a new correct term.

Exercise 30. Compute the series expansion to order 15 of $\tan x$ near zero, from the differential equation $\tan' = 1 + \tan^2$.

7.5.3 Lazy Power Series

The fixed-point phenomenon motivates the introduction of a new kind of formal power series called *lazy* power series. They are not truncated series, but infinite series; the "lazy" adjective means that coefficients are computed on demand only. As a counterpart, we can only represent series whose coefficients are computable: essentially, combinations of basic series and some solutions of equations for which relations like (7.2) exist. For example, the series `lazy_exp` defined by

```
sage: L.<x> = LazyPowerSeriesRing(QQ)
sage: lazy_exp = x.exponential(); lazy_exp
O(1)
```

is an object which contains in its internal representation all the information needed to compute the series expansion of $\exp x$ to any order. Its output is initially `O(1)` since no coefficient was computed so far. If we ask for the coefficient of x^5, the corresponding computation is performed, and the computed coefficients are stored in memory:

```
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)
```

Let us go back to the equation $e^{xf(x)} = f(x)$ to see how it can be solved with lazy series. We first try to reproduce the above computation in the ring `QQ[['x']]`:

```
sage: f = L(1)  # the constant lazy series 1
sage: for i in range(5):
....:     f = (x*f).exponential()
....:     f.compute_coefficients(5) # forces the computation
....:     print(f)                  # of the first coefficients
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)
```

The obtained expansions are of course the same as above[7]. However the value of `f` at each iteration is now an infinite series, whose coefficients can be computed on demand. All these intermediate series are kept in memory. The computation of each one is automatically done at the required precision in order to yield, for example, the coefficient of x^7 in the last iterate when one asks for it:

```
sage: f[7]
28673/630
```

With the code of §7.5.2, accessing `f[7]` would have raised an error, since the index 7 is larger than the truncation order of the series `f`.

However, the value returned by `f[7]` is the coefficient of x^7 in the iterate $\Phi^5(1)$, and not in the solution! The power of lazy series is the possibility to directly get the limit, by defining f itself as a lazy series:

```
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]
```

The iterative computation did "work" thanks to the relation (7.2). Under the hood, Sage deduces from the recursive definition `f.define((x*f).exponential())` a similar formula, which enables it to compute coefficients by recurrence.

7.6 Computer Representation of Polynomials

A given mathematical object — the polynomial p, with coefficients in A — might be encoded in very different ways on a computer. While the result of a mathematical operation on p is clearly independent of the representation, the corresponding

[7] We observe however that Sage sometimes has incoherent conventions: the `exp` method for truncated series is now called `exponential`, and `compute_coefficients(5)` computes the coefficients up to order 5 included, whereas `default_prec=5` gave series truncated after the coefficient of x^4.

Sage objects might behave differently. The choice of representation impacts the possible operations, the exact form of their results, and particularly the efficiency of the computations.

Dense or Sparse Representation. Two principal ways exist for representing polynomials. In a *dense* representation, the coefficients of $p = \sum_{i=0}^{n} p_i\, x^i$ are stored in a table $[p_0, \ldots, p_n]$ indexed by the exponents. A *sparse* representation only stores the non-zero coefficients: the polynomial is encoded by a set of pairs exponent-coefficient (i, p_i), stored in a list, or better, in a dictionary indexed by the exponents (see §3.3.9).

For polynomials that really are dense, i.e., whose coefficients are mostly non-zero, the dense representation uses less memory and enables faster computations. It saves the encoding of the exponents and of the internal data structures of the dictionary: it only stores what is strictly necessary, the coefficients. Moreover, accessing an element and iterating on elements are faster in a table than in a dictionary. Conversely, the sparse representation enables us to efficiently compute with polynomials that we could not even store in memory with a dense representation:

```
sage: R = PolynomialRing(ZZ, 'x', sparse=True)
sage: p = R.cyclotomic_polynomial(2^50); p, p.derivative()
(x^562949953421312 + 1, 562949953421312*x^562949953421311)
```

As shown by the preceding example, the representation is a characteristic of the polynomial ring, chosen at its construction. The "dense" polynomial $x \in \mathbb{Q}[x]$ and the "sparse" polynomial $x \in \mathbb{Q}[x]$ thus have different parents. The default representation of univariate polynomials is dense. The option `sparse=True` of `PolynomialRing` enables us to build a polynomial ring with sparse representation.

In addition, some details of the representation vary according to the kind of coefficients. The same holds for the code used to perform basic operations. Indeed, Sage provides a *generic* polynomial implementation which works on any commutative ring, but also optimised variants for some particular types of coefficients. These variants bring some additional features, and above all are much more efficient than the generic version. They call for this purpose some specialised external libraries, like FLINT or NTL in the case of $\mathbb{Z}[x]$.

To complete huge computations successfully, it is very important to work whenever possible in polynomial rings with efficient implementations. The help page output by `p?` for a polynomial `p` indicates which implementation it uses. The choice of the implementation often depends on the base ring and the representation. The `implementation` option of `PolynomialRing` enables us to choose a particular implementation when several are possible.

Symbolic Expressions. The symbolic expressions discussed in Chapters 1 and 2 (i.e., the elements of `SR`) provide a third representation of polynomials. They are a natural choice when a computation mixes polynomials and more diverse expressions, as it is often the case in analysis. The flexibility they offer is sometimes useful even in a fully algebraic context. For example, the polynomial

A little bit of theory

To get the best out of fast operations on polynomials, it is good to have an idea of their algorithmic complexity. We briefly discuss this for the reader with some algorithmic knowledge. We limit ourselves to the case of dense polynomials.

Additions, subtractions and other direct operations on coefficients are performed in linear time with respect to the degrees of the considered polynomials. Their practical efficiency thus depends essentially on the easy access to the coefficients, and therefore on the internal data structure.

The critical operation is multiplication. Indeed, not only is this a basic arithmetic operation, but other operations use algorithms whose complexity depends essentially on that of multiplication. For example, given two polynomials of degree at most n, we can compute their Euclidean division at the cost of $O(1)$ multiplications, or their gcd at that of $O(\log n)$ multiplications.

Good news: we know how to multiply polynomials in quasi-linear time. More precisely, the best known complexity over any ring is $O(n \log n \log\log n)$ operations in the base ring. It relies on generalisations of the famous Schönhage-Strassen algorithm, which attains the same complexity for integer multiplication. By comparison, the method used by hand to multiply polynomials requires of the order of n^2 operations.

In practice, the fast multiplication algorithms are competitive for large enough degrees, as well as corresponding methods for the division. The libraries called by Sage for some kinds of coefficients use such advanced algorithms: this explains why Sage is able to efficiently work with polynomials of huge degree on some coefficient rings.

$(x+1)^{10^{10}}$, once expanded, is dense, but it is not necessary (nor desirable!) to expand it in order to differentiate it or evaluate it numerically.

Beware however: as opposed to algebraic polynomials, symbolic polynomials (in `SR`) are not attached to a particular polynomial ring, and are not put in canonical form. A given polynomial might have a lot of different forms, it is the user's responsibility to perform the needed conversions between them. In the same vein, the `SR` domain groups together all symbolic expressions, without any distinction between polynomials and other expressions, but we can explicitly check whether a given symbolic expression `f` is polynomial in the variable `x` by `f.is_polynomial(x)`.

Mathematics is the art of reducing any problem to linear algebra.

William Stein

8 Linear Algebra

This chapter deals with exact and symbolic linear algebra, i.e., linear algebra over rings specific to computer algebra, such as $\mathbb{Z}$, finite fields, or polynomial rings. Numerical linear algebra, based on fixed precision approximate arithmetic, is presented in Chapter 13. We first present constructions on matrices and their vector spaces together with basic operations (§8.1), then various computations on these matrices, gathered in two groups: operations related to Gaussian elimination and left equivalence transformations (§8.2.1-§8.2.2), and computations related to eigenvalues, eigenspaces and similarity transformations (§8.2.3).

The reader may refer to the books of von zur Gathen and Gerhard [vzGG03], and the Ph. D. thesis of Storjohann [Sto00] for further details on the notions presented in this chapter.

8.1 Elementary Constructs and Manipulations

8.1.1 Spaces of Vectors and Matrices

As is the case for polynomials, vectors and matrices are handled as algebraic objects belonging to a space. This is a vector space when the coefficients are elements of a field, or a free module when the coefficients are elements of a ring.

The space $\mathcal{M}_{2,3}(\mathbb{Z})$ and the vector space $(\mathbb{F}_{3^2})^3$ are constructed by:

```
sage: MS = MatrixSpace(ZZ,2,3); MS
Full MatrixSpace of 2 by 3 dense matrices over Integer Ring
sage: VS = VectorSpace(GF(3^2,'x'),3); VS
Vector space of dimension 3 over Finite Field in x of size 3^2
```

Matrix space	
construct	`MS = MatrixSpace(K, nrows, ncols)` *or* `MS = Mat(...)`
construct (sparse matrix)	`MS = MatrixSpace(K, nrows, ncols, sparse = True)`
base ring K	`MS.base_ring()`
extending the ring	`MS.base_extend(B)`
changing the ring	`MS.change_ring(B)`
group generated	`MatrixGroup([A,B])`
basis of the vector space	`MS.basis()` *or* `MS.gens()`

Matrix constructs	
zero matrix	`MS()` *or* `MS.zero()` *or* `zero_matrix(K,nrows,ncols)`
matrix from coefficients	`MS([1,2,3,4])` *or* `matrix(K,2,2,[1,2,3,4])` *or* `matrix(K,[[1,2],[3,4]])`
identity matrix	`MS.one()` *or* `MS.identity_matrix()` *or* `identity_matrix(K,n)`
random matrix	`MS.random_element()` *or* `random_matrix(K,nrows,ncols)`
Jordan block	`jordan_block(x,n)`
block matrix	`block_matrix([A,1,B,0])` *or* `block_diagonal_matrix(A,B)`

Elementary manipulations	
accessing a coefficient	`A[2,3]` *or* `A[2][3]`
last row, third column	`A[-1,:]`, `A[:,2]`
first four even columns	`A[:,0:8:2]`

submatrices	`A[3:4,2:5]`, `A[:,2:5]`, `A[:4,2:5]`
	`A.matrix_from_rows([1,3])`
	`A.matrix_from_columns([2,5])`
	`A.matrix_from_rows_and_columns([1,3],[2,5])`
	`A.submatrix(i,j,nrows,ncols)`

row concatenation	`A.stack(B)`
column concatenation	`A.augment(B)`

TABLE 8.1 – Constructs for matrices and their spaces.

A generating family for these spaces, namely the canonical basis, is obtained by the methods `MS.gens()` or `MS.basis()`.

```
sage: B = MatrixSpace(ZZ,2,3).basis()
sage: list(B)
```

$$\left[\begin{pmatrix}1&0&0\\0&0&0\end{pmatrix},\begin{pmatrix}0&1&0\\0&0&0\end{pmatrix},\begin{pmatrix}0&0&1\\0&0&0\end{pmatrix},\begin{pmatrix}0&0&0\\1&0&0\end{pmatrix},\begin{pmatrix}0&0&0\\0&1&0\end{pmatrix},\begin{pmatrix}0&0&0\\0&0&1\end{pmatrix}\right]$$

One can conveniently access its elements by row and column number:

```
sage: B[1,2]
```

$$\begin{pmatrix}0&0&0\\0&0&1\end{pmatrix}$$

Matrix Groups. One can also define groups and subgroups in the space of matrices. The general linear group of degree n over a field K, denoted by $\mathrm{GL}_n(K)$, is the group consisting of all invertible $n \times n$ matrices in $\mathcal{M}_{n,n}(K)$. It is

constructed in Sage with the command `GL(n,K)`. The special linear group $\mathrm{SL}_n(K)$, consisting of the elements of $\mathrm{GL}_n(K)$ with determinant one, is constructed with the command `SL(n,K)`.

The construction `MatrixGroup([A,B,...])` returns the group generated by the matrices in the list argument, all of which need to be invertible.

```
sage: A = matrix(GF(11), 2, 2, [1,0,0,2])
sage: B = matrix(GF(11), 2, 2, [0,1,1,0])
sage: MG = MatrixGroup([A,B])
sage: MG.cardinality()
200
sage: identity_matrix(GF(11),2) in MG
True
```

8.1.2 Vector and Matrix Construction

Matrices and vectors can naturally be generated as elements of their space, by providing the list of their coefficients. For matrices, they are listed in a row major mode:

```
sage: MS = MatrixSpace(ZZ,2,3); A = MS([1,2,3,4,5,6]); A
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

The empty constructor `MS()` returns the zero matrix, and so does the method `MS.zero()`. Several specialised constructors produce the most common matrices, as for example `random_matrix`, `identity_matrix`, `jordan_block` (see Table 8.1). In particular, one can construct matrices and vectors using the `matrix` and `vector` constructors, without having to construct the related space beforehand. By default, a matrix is built over the ring of integers $\mathbb{Z}$ and has dimension 0×0.

```
sage: a = matrix(); a.parent()
Full MatrixSpace of 0 by 0 dense matrices over Integer Ring
```

Of course, one can also specify the coefficient domain and the dimensions, to form a zero matrix or a matrix with prescribed coefficients provided in a list.

```
sage: a = matrix(GF(8,'x'),3,4); a.parent()
Full MatrixSpace of 3 by 4 dense matrices over Finite Field
in x of size 2^3
```

The constructor `matrix` also accepts as argument objects that have a natural transformation into a matrix. For instance, it can be used to generate the adjacency matrix of a graph, with coefficients in $\mathbb{Z}$.

```
sage: g = graphs.PetersenGraph()
sage: m = matrix(g); m; m.parent()
[0 1 0 0 1 1 0 0 0 0]
[1 0 1 0 0 0 1 0 0 0]
[0 1 0 1 0 0 0 1 0 0]
[0 0 1 0 1 0 0 0 1 0]
[0 0 1 0 1 0 0 0 1 0]
```

```
[1 0 0 1 0 0 0 0 0 1]
[1 0 0 0 0 0 0 1 1 0]
[0 1 0 0 0 0 0 0 1 1]
[0 0 1 0 0 1 0 0 0 1]
[0 0 0 1 0 1 1 0 0 0]
[0 0 0 0 1 0 1 1 0 0]
Full MatrixSpace of 10 by 10 dense matrices over Integer Ring
```

Block Matrices. The function `block_matrix` allows to define a matrix by blocks from several submatrices.

```
sage: A = matrix([[1,2],[3,4]])
sage: block_matrix([[A,-A],[2*A, A^2]])
```

$$\left(\begin{array}{cc|cc} 1 & 2 & -1 & -2 \\ 3 & 4 & -3 & -4 \\ \hline 2 & 4 & 7 & 10 \\ 6 & 8 & 15 & 22 \end{array}\right)$$

By default, this structure is square by blocks but the number of block rows or columns can be specified by the optional arguments `nrows` and `ncols` respectively. Whenever it makes sense, a scalar coefficient, such as `0` or `1`, is interpreted as a block, namely a zero block or the identity block, with conforming dimensions.

```
sage: A = matrix([[1,2,3],[4,5,6]])
sage: block_matrix([1,A,0,0,-A,2], ncols=3)
```

$$\left(\begin{array}{cc|ccc|cc} 1 & 0 & 1 & 2 & 3 & 0 & 0 \\ 0 & 1 & 4 & 5 & 6 & 0 & 0 \\ \hline 0 & 0 & -1 & -2 & -3 & 2 & 0 \\ 0 & 0 & -4 & -5 & -6 & 0 & 2 \end{array}\right)$$

In the special case of block diagonal matrices, the list of the diagonal blocks is simply passed to the constructor `block_diagonal_matrix`.

```
sage: A = matrix([[1,2,3],[0,1,0]])
sage: block_diagonal_matrix(A, A.transpose())
```

$$\left(\begin{array}{ccc|cc} 1 & 2 & 3 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 3 & 0 \end{array}\right)$$

The block structure is only a display feature and Sage treats the matrix as any other matrix. This display mode can be disabled by providing the argument `subdivide=False` to the `block_matrix` constructor.

8.1.3 Basic Manipulations and Arithmetic on Matrices

Indexing and Accessing Coefficients. Coefficients and submatrices are accessed in a unified way through the square bracket operator `A[i,j]`, following

the usual Python conventions. Row and column indices i and j can be integers (in order to access a coefficient) or intervals of the form `1:3` (recall that indices are zero-based in Python and intervals are always inclusive on their lower end and exclusive on their upper end). The interval ":" without bounds corresponds to the entirety of the possible indices in the dimension considered. Notation `a:b:k` lists all indices between a and $b-1$ by steps of k. Lastly, negative indices are also valid and allow one to iterate from the end of the index space. Thus `A[-2,:]` refers to the second to last row of matrix `A`. These access patterns to submatrices are available for both read and write operations. For instance, a given column can be modified as follows:

```
sage: A = matrix(3,3,range(9))
sage: A[:,1] = vector([1,1,1]); A
```

$$\begin{pmatrix} 0 & 1 & 2 \\ 3 & 1 & 5 \\ 6 & 1 & 8 \end{pmatrix}$$

The step increment k can also be negative, in order to iterate in decreasing order.

```
sage: A[::-1], A[:,::-1], A[::2,-1]
```

$$\left(\begin{pmatrix} 6 & 1 & 8 \\ 3 & 1 & 5 \\ 0 & 1 & 2 \end{pmatrix}, \begin{pmatrix} 2 & 1 & 0 \\ 5 & 1 & 3 \\ 8 & 1 & 6 \end{pmatrix}, \begin{pmatrix} 2 \\ 8 \end{pmatrix}\right)$$

Extracting a Submatrix. In order to extract a submatrix from a list of rows or column indices, not necessarily contiguous, one can use the methods `A.matrix_from_rows`, `A.matrix_from_columns` or in the more general setting the method `A.matrix_from_rows_and_columns`.

```
sage: A = matrix(ZZ,4,4,range(16)); A
```

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{pmatrix}$$

```
sage: A.matrix_from_rows_and_columns([0,2,3],[1,2])
```

$$\begin{pmatrix} 1 & 2 \\ 9 & 10 \\ 13 & 14 \end{pmatrix}$$

Alternatively, when the row and column indices are contiguous, one can also use the method `A.submatrix(i,j,m,n)` forming the submatrix of dimension $m \times n$ whose upper left coefficient is at position (i, j) in `A`.

Basic operations	
transpose, conjugate	`A.transpose()` , `A.conjugate()`
scalar product	`a*A`
sum, product, k-th power, inverse	`A + B`, `A * B`, `A^k`, `A^-1` *or* `~A`

TABLE 8.2 – Basic operations and matrix arithmetic.

Embedding and Extension. The method `base_extend` of a matrix space makes it possible to embed a matrix space into another matrix space with the same dimensions but over an extension of the base ring. This operation is however only valid for a field or a ring extension. In order to change the ring of a matrix space, following a ring morphism (when it exists), one uses instead the method `change_ring`.

```
sage: MS = MatrixSpace(GF(3),2,3)
sage: MS.base_extend(GF(9,'x'))
Full MatrixSpace of 2 by 3 dense matrices over Finite Field
in x of size 3^2
sage: MS = MatrixSpace(ZZ,2,3)
sage: MS.change_ring(GF(3))
Full MatrixSpace of 2 by 3 dense matrices over Finite Field of size 3
```

Mutability and Caching. By default, matrix objects are mutable, which means that one can modify their members (namely their coefficients) after their construction. In order to protect the matrix against modification, one can make it immutable with the function `A.set_immutable()`. It is then still possible to create mutable copies of this matrix with the function `copy(A)`. Remark that the caching mechanism for the computed results, such as the rank, the determinant, etc., is always active, regardless of the mutability status.

8.1.4 Basic Operations on Matrices

Arithmetic operations on matrices are done with the usual operators `+,-,*,^`. The inverse of a matrix `A` is obtained equivalently by `A^-1` or `~A`. For a scalar `a` and a matrix `A`, the operation `a*A` corresponds to the scalar multiplication of the matrix space. For any other operation where a scalar `a` is provided in place of a matrix (as for instance in the operation `a+A`), this scalar is interpreted as the corresponding scalar matrix aI_n if $a \neq 0$ if dimensions permit it. Elementwise product of two matrices is achieved by the method `elementwise_product`.

8.2 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 transformation or the dimension of a subspace all boil down to operations on the corresponding matrices that will reveal the property.

These transformations most often correspond to changes of basis, which from the matrix point of view translate into equivalence transformations: $B = PAQ^{-1}$, where P and Q are invertible matrices. Two matrices are equivalent if such a transformation from one to another exists. One can then form equivalence classes for this relation and define normal forms that characterise each equivalence class in a unique manner. In the following, we will present most matrix computations in Sage, from the viewpoint of two instances of these transformations:

- The left equivalence transformations, of the form $B = UA$, revealing characteristic properties for families of vectors, such as their rank (the number of linearly independent vectors), the determinant (the volume of the parallelepiped formed by the family of vectors), the rank profile (the first set of vectors forming a basis of the space spanned by the family)... Gaussian elimination is the key tool for these transformations and the reduced echelon form is the corresponding normal form (or the Hermite form over $\mathbb{Z}$).

- Similarity transformations, of the form $B = UAU^{-1}$, which reveal characteristic properties of the matrices representing endomorphisms, like eigenvalues, eigenspaces, minimal and characteristic polynomials... The Jordan or Frobenius form, according to the underlying domain, will be normal forms for these transformations.

The Gram-Schmidt orthogonalisation process leads to another decomposition based on left equivalence transformations, changing a matrix into a set of orthogonal vectors.

8.2.1 Gaussian Elimination, Echelon Form

Gaussian Elimination and Left Equivalence. Gaussian elimination is a building block operation in computational linear algebra, as it gives access to a matrix representation, a product of triangular factors, which is both better suited for computations, e.g., solving linear systems, and which reveals fundamental properties such as the rank, the rank profile, the determinant, etc. The basic operations used to define Gaussian elimination are the elementary row operations:

- permuting two rows: $L_i \leftrightarrow L_j$,

- adding a multiple of a row to another: $L_i \leftarrow L_i + sL_j$.

From a matrix point of view, these transformations correspond to the left multiplication by respectively a transposition matrix $T_{i,j}$ and by a transvection

Gaussian elimination and applications	
row transvection	`add_multiple_of_row(i,j,s)`
column transvection	`add_multiple_of_column(i,j,s)`
row, column transposition	`swap_rows(i1,i2)`, `swap_columns(j1,j2)`
reduced row echelon form, immutable	`echelon_form`
reduced row echelon form, in-place	`echelonize`
invariant factors	`elementary_divisors`
Smith normal form	`smith_form`
determinant, rank	`det`, `rank`
minors of order k	`minors(k)`
column, row rank profile	`pivots`, `pivot_rows`
left-hand side system solving ($x^tA = b^t$)	`b/A` *or* `A.solve_left(b)`
right-hand side system solving ($Ax = b$)	`A\b` *or* `A.solve_right(b)`
image space	`image`
left kernel	`kernel` *or* `left_kernel`
right kernel	`right_kernel`
kernel in the base ring	`integer_kernel`

Spectral decomposition	
minimal polynomial	`minimal_polynomial` *or* `minpoly`
characteristic polynomial	`characteristic_polynomial` *or* `charpoly`
Krylov iterates on the left-hand side	`maxspin(v)`
eigenvalues	`eigenvalues`
left, right eigenvectors	`eigenvectors_left`, `eigenvectors_right`
left, right eigenspaces	`eigenspaces_left`, `eigenspaces_right`
diagonalisation	`eigenmatrix_left`, `eigenmatrix_right`
Jordan block $J_{a,k}$	`jordan_block(a,k)`

TABLE 8.3 – Matrix computations.

matrix $C_{i,j,s}$ defined by:

$$T_{i,j} = \begin{array}{c} \begin{matrix} & & i & & j & & \end{matrix} \\ \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 0 & & 1 & & \\ & & & \ddots & & & \\ & & 1 & & 0 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{bmatrix} \end{array}, C_{i,j,s} = \begin{array}{c} \begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & s & \\ & & & \ddots & & \\ & & & & 1 & \\ & & & & & \ddots \end{bmatrix} \begin{matrix} \\ \\ i \\ \\ j \\ \\ \end{matrix} \end{array}.$$

These matrices all have determinant 1 or -1. As a consequence, multiplying on the left by any product of these matrices, is a volume preserving change of basis, namely preserving the determinant (up to the sign). In Sage, a transvection is achieved by the method `add_multiple_of_row(i,j,s)`, and a transposition by the method `swap_rows(i,j)`.

For a given column vector $x = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$ whose k-th coefficient is invertible, the Gauss transform is the composition of the transvections C_{i,k,ℓ_i} for $i = k+1 \dots m$, with $\ell_i = -\frac{x_i}{x_k}$ (the order is irrelevant, since they all commute with each other). The corresponding matrix is the following:

$$G_{x,k} = C_{k+1,k,\ell_{k+1}} \times \cdots \times C_{m,k,\ell_m} = \begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & \ell_{k+1} & \ddots & & \\ & & \ell_{k+2} & & \ddots & \\ & & \vdots & & & \ddots \\ & & \ell_m & & & 1 \end{bmatrix} \begin{matrix} \\ \\ k \\ \\ \\ \\ \end{matrix} .$$

The effect of a Gauss transform $G_{x,k}$ is to eliminate the coefficients of the vector below the pivot x_k.

$$G_{x,k} \begin{bmatrix} x_1 \\ \vdots \\ x_k \\ x_{k+1} \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} x_1 \\ \vdots \\ x_k \\ 0 \\ \vdots \\ 0 \end{bmatrix} .$$

For an $m \times n$ matrix $A = [a_{i,j}]$, the Gaussian elimination algorithm then proceeds iteratively, from the leftmost column to the rightmost column. Assuming that the $k-1$ first columns have already been processed, generating $p \leq k-1$ pivots, the k-th column is then treated as follows:

- find the first invertible coefficient $a_{i,k}$ in the column C_k on a row $i > p$. It is the pivot.
- If no pivot can be found, move on to the next column.
- Apply the transposition $T_{i,p+1}$ on the rows of the matrix, to place the pivot at position $(p+1, k)$.
- Apply the Gauss transform $G_{x,p+1}$, where x is the new k-th column C_k.

This algorithm transforms the matrix A into an upper triangular matrix. More precisely, it will have an echelon form: the leading coefficient of each non-zero row is to the right of that of the preceding row, and all zero rows are on the bottom part of the matrix. The following example traces the execution of this algorithm on a 4×3 matrix.

```
sage: a = matrix(GF(7),4,3,[6,2,2,5,4,4,6,4,5,5,1,3]); a
```

$$\begin{pmatrix} 6 & 2 & 2 \\ 5 & 4 & 4 \\ 6 & 4 & 5 \\ 5 & 1 & 3 \end{pmatrix}$$

```
sage: u = copy(identity_matrix(GF(7),4)); u[1:,0] = -a[1:,0]/a[0,0]
sage: u, u*a
```

$$\left(\begin{pmatrix} 1 & 0 & 0 & 0 \\ 5 & 1 & 0 & 0 \\ 6 & 0 & 1 & 0 \\ 5 & 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 6 & 2 & 2 \\ 0 & 0 & 0 \\ 0 & 2 & 3 \\ 0 & 4 & 6 \end{pmatrix}\right)$$

```
sage: v = copy(identity_matrix(GF(7),4)); v.swap_rows(1,2)
sage: b = v*u*a; v, b
```

$$\left(\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 6 & 2 & 2 \\ 0 & 2 & 3 \\ 0 & 0 & 0 \\ 0 & 4 & 6 \end{pmatrix}\right)$$

```
sage: w = copy(identity_matrix(GF(7),4))
sage: w[2:,1] = -b[2:,1]/b[1,1]; w, w*b
```

$$\left(\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 5 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 6 & 2 & 2 \\ 0 & 2 & 3 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}\right)$$

Gauss-Jordan Elimination. The Gauss-Jordan transformation is similar to the Gauss transformation, simply adding to $G_{x,k}$ the transvections corresponding to the rows of index $i < k$; this has the effect of eliminating all coefficients, above and below the pivot. If in addition each row is divided by its pivot, this leads to the so called reduced echelon form or Gauss-Jordan form. It is a normal form: for every equivalence class, there is a unique such reduced echelon form.

Definition. 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 a 1 and is to the right of the pivot of the row above,
- pivots are the only non-zero elements in their column.

Theorem. For every $m \times n$ 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 U such that $UA = R$.

In Sage, the reduced echelon form is obtained by the methods `echelonize` and `echelon_form`. The former replaces the input matrix by its reduced echelon form, while the latter returns an immutable matrix without modifying the input matrix.

```
sage: A = matrix(GF(7),4,5,[4,4,0,2,4,5,1,6,5,4,1,1,0,1,0,5,1,6,6,2])
sage: A, A.echelon_form()
```

$$\left(\begin{pmatrix} 4 & 4 & 0 & 2 & 4 \\ 5 & 1 & 6 & 5 & 4 \\ 1 & 1 & 0 & 1 & 0 \\ 5 & 1 & 6 & 6 & 2 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 5 & 0 & 3 \\ 0 & 1 & 2 & 0 & 6 \\ 0 & 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}\right)$$

Most variants of the Gaussian elimination algorithm yield a matrix decomposition of great interest for computations: decompositions of the form $A = LU$ for generic matrices, $A = LUP$ for regular matrices, $A = LSP, LQUP, PLUQ$ for matrices with arbitrary rank. In these decompositions, the L matrices are lower triangular (with zeros above the main diagonal), with a diagonal of ones, the U matrices are upper triangular (with zeros below the main diagonal) with a diagonal of invertible elements, and P and Q are permutation matrices. Although these decompositions are less expensive to compute than the reduced echelon form (nearly $\frac{2}{3}n^3$ against $2n^3$ for an $n \times n$ full rank matrix), they do not produce a normal form.

Echelon Form over a Euclidean Ring. Over a Euclidean ring, non-zero coefficients are not necessarily invertible while only the invertible ones can be chosen as pivots in the course of Gaussian elimination. Hence some non-zero columns may not contain any pivot, and elimination would no longer be possible. It is however still possible to define a unimodular transformation (whose determinant is invertible) eliminating the leading coefficient in a row with that of another row, thanks to the extended Euclidean algorithm. Let $A = \begin{bmatrix} a & * \\ b & * \end{bmatrix}$ and $g = \gcd(a, b)$. Let u and v be the Bézout coefficients computed with the extended Euclidean algorithm applied to a and b (such that $g = ua + vb$), and let $s = -b/g, t = a/g$ such that

$$\begin{bmatrix} u & v \\ s & t \end{bmatrix} \begin{bmatrix} a & * \\ b & * \end{bmatrix} = \begin{bmatrix} g & * \\ 0 & * \end{bmatrix}.$$

This transformation is unimodular since $\det \begin{pmatrix} u & v \\ s & t \end{pmatrix} = 1$.

Moreover, as in the Gauss-Jordan elimination, it is also always possible to add multiples of the pivot row to the rows above it in order to reduce the coefficients in the pivot column modulo the pivot g. When iterated over all columns of the matrix, this operation produces the Hermite normal form.

Definition. 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. For any $m \times n$ matrix A over a Euclidean ring, there is a unique $m \times n$ matrix H in Hermite form and an $m \times m$ unimodular matrix U, such that $UA = H$.

Over a field, the Hermite form coincides with the reduced echelon form. Indeed all pivots are then invertible, each non-zero row can be divided by its pivot, making this pivot equal to one. Then reducing the coefficients above each pivot modulo one, means setting them to zero, which produces a reduced echelon form. In Sage, there is therefore a unique method, `echelon_form`, which either returns the Hermite form or the reduced echelon form depending whether the coefficient domain is a Euclidean ring or a field.

For instance, a matrix with integer coefficients yields the following two distinct reduced echelon forms depending on whether the base ring is $\mathbb{Z}$ or $\mathbb{Q}$.

```
sage: a = matrix(ZZ, 4, 6, [2,1,2,2,2,-1,1,2,-1,2,1,-1,2,1,-1,\
....:                       -1,2,2,2,1,1,-1,-1,-1]); a.echelon_form()
```

$$\begin{pmatrix} 1 & 2 & 0 & 5 & 4 & -1 \\ 0 & 3 & 0 & 2 & -6 & -7 \\ 0 & 0 & 1 & 3 & 3 & 0 \\ 0 & 0 & 0 & 6 & 9 & 3 \end{pmatrix}$$

```
sage: a.base_extend(QQ).echelon_form()
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \frac{5}{2} & \frac{11}{6} \\ 0 & 1 & 0 & 0 & -3 & -\frac{8}{3} \\ 0 & 0 & 1 & 0 & -\frac{3}{2} & -\frac{3}{2} \\ 0 & 0 & 0 & 1 & \frac{3}{2} & \frac{1}{2} \end{pmatrix}$$

For matrices over $\mathbb{Z}$, the Hermite form can also be obtained with the method `hermite_form`. With both methods, the transformation matrix U such that $UA = H$ is also returned when the option `transformation=True` is passed.

```
sage: A = matrix(ZZ,4,5,[4,4,0,2,4,5,1,6,5,4,1,1,0,1,0,5,1,6,6,2])
sage: H, U = A.echelon_form(transformation=True); H, U
```

$$\left(\begin{pmatrix} 1 & 1 & 0 & 0 & 2 \\ 0 & 4 & -6 & 0 & -4 \\ 0 & 0 & 0 & 1 & -2 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 1 & -1 \\ 0 & -1 & 5 & 0 \\ 0 & -1 & 0 & 1 \\ 1 & -2 & -4 & 2 \end{pmatrix} \right)$$

Invariant Factors and the Smith Normal Form. When eliminating further the Hermite normal form, using unimodular right transformations (acting on columns) one can then reach a diagonal canonical form, named the Smith normal form. Its diagonal coefficients are the elementary divisors of the matrix. They are totally ordered under the divisibility relation: $s_i \mid s_{i+1}$.

THEOREM. For any $m \times n$ matrix A with coefficients over a principal ideal ring, there exist unimodular matrices U and V of dimensions $m \times m$ and $n \times n$ respectively, and a unique $m \times n$ diagonal matrix S such that $S = UAV$. The coefficients $s_i = S_{i,i}$ for $i \in \{1, \ldots, \min(m, n)\}$ are the elementary divisors of A and satisfy $s_i \mid s_{i+1}, \forall i < \min(m, n)$.

In Sage, the method `elementary_divisors` returns the list of elementary divisors. One can also compute directly the Smith normal form, together with the transformation matrices U and V, with the `smith_form` method.

```
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])
sage: S,U,V = A.smith_form(); S,U,V
```

$$\left(\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -2 & 1 & 0 & 0 \\ 0 & 0 & -2 & -1 \end{pmatrix}, \begin{pmatrix} 0 & -2 & -1 & -5 & 0 \\ 1 & 0 & 1 & -1 & -1 \\ 0 & 0 & 0 & 0 & 1 \\ -1 & 2 & 0 & 5 & 0 \\ 0 & -1 & 0 & -2 & 0 \end{pmatrix}\right)$$

```
sage: A.elementary_divisors()
[1, 1, 3, 6]
sage: S == U*A*V
True
```

Rank, Rank Profile and Pivots. Gaussian elimination reveals many matrix invariants, such as the rank, the determinant (computed as the product of the pivots). These values are accessible with the methods `det` and `rank`. They are cached and therefore are not recomputed when the method is called once again.

More generally, the notion of rank profile is of interest when considering the matrix as a sequence of vectors.

Definition. The column rank profile of an $m \times n$ matrix A of rank r is the lexicographically minimal sequence of r indices of linearly independent columns in A.

The column rank profile can be read directly off the reduced row echelon form, as the sequence of column indices of the pivots. It is obtained by the method `pivots`. When the reduced row echelon form has already been computed, the column rank profile is stored in cache and can be obtained with no additional computation.

The row rank profile is defined similarly, considering the matrix as a sequence of m row vectors. It is obtained by the method `pivot_rows`. It is equivalent to the column rank profile of the transposed matrix.

```
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])
sage: B.transpose().echelon_form()
```

$$\begin{pmatrix} 1 & 0 & 5 & 0 & 3 \\ 0 & 1 & 2 & 0 & 6 \\ 0 & 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

```
sage: B.pivot_rows()
(0, 1, 3)
sage: B.transpose().pivots() == B.pivot_rows()
True
```

8.2.2 Linear System Solving, Image and Nullspace Basis

Linear System Solving. A linear system of equations can be represented by a matrix A and a right-hand side or a left-hand side vector b for systems of the form $Ax = b$ or $x^tA = b^t$ respectively. The methods `solve_right` and `solve_left` solve these systems. One can alternatively use the operators `A\b` and `b/A`. When the system is given by a matrix over a ring, the resolution is systematically performed over the fraction field of this ring (e.g., $\mathbb{Q}$ for the ring $\mathbb{Z}$ or $K(X)$ for $K[X]$). We will see later on how to solve the system over the base ring. The right-hand side in the system equality can be either a vector or a matrix (the latter corresponding to the resolution of several systems with the same matrix).

The system's matrix can be rectangular and the system may have a unique solution, no solution or an infinite number of solutions. In the latter case, the `solve` methods return one of these solutions, zeroing out the coefficients corresponding to linearly dependent columns in the system.

```
sage: R.<x> = PolynomialRing(GF(5),'x')
sage: A = random_matrix(R,2,3); A
```

$$\begin{pmatrix} 3x^2 + x & x^2 + 2x & 2x^2 + 2 \\ x^2 + x + 2 & 2x^2 + 4x + 3 & x^2 + 4x + 3 \end{pmatrix}$$

```
sage: b = random_matrix(R,2,1); b
```

$$\begin{pmatrix} 4x^2 + 1 \\ 3x^2 + 2x \end{pmatrix}$$

```
sage: A.solve_right(b)
```

$$\begin{pmatrix} \frac{4x^3+2x+4}{3x^3+2x^2+2x} \\ \frac{3x^2+4x+3}{x^3+4x^2+4x} \\ 0 \end{pmatrix}$$

```
sage: A.solve_right(b) == A\b
True
```

Image and Kernel. Viewed as a linear transformation Φ, an $m \times n$ matrix A defines two subspaces of K^m and K^n, respectively the image and the kernel of Φ.

The image is the set of all vectors in K^m which are a linear combination of the columns of A. It is given by the method `image`, returning a vector space, with a basis in reduced echelon form.

The kernel is the subspace of K^n formed by all vectors x satisfying $Ax = 0$. A basis of this subspace is useful for describing the set of solutions of an underdetermined linear system, having an infinite number of solutions: if $\overline{x}$ is a solution of $Ax = b$ and V is the kernel of A, then $\overline{x} + V$ is the set of all solutions to the system. This set is computed by the method `right_kernel` returning a vector space with a basis in reduced echelon form. The left kernel is defined

similarly as the set of vectors $x \in K^m$ such that $x^t A = 0$, which is also the right kernel of the transposed matrix of A. It is returned by the method `left_kernel`. By convention the method `kernel` returns the left kernel and the bases are given as matrices of row vectors.

```
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.image()
Vector space of degree 5 and dimension 3 over Rational Field
Basis matrix:
[     1      0      0    1/4 -11/32]
[     0      1      0     -1   -1/8]
[     0      0      1    1/2   1/16]
sage: a.right_kernel()
Vector space of degree 5 and dimension 2 over Rational Field
Basis matrix:
[    1     0     0  -1/3   8/3]
[    0     1  -1/2 11/12   2/3]
```

The notion of kernel extends naturally to the case where coefficients no longer belong to a field, but a ring; it then has the structure of a free module. In particular, for a matrix defined over the fraction field of an integral ring, the kernel in the base ring is obtained with the method `integer_kernel`. For instance, the kernel of a matrix over $\mathbb{Z}$, embedded in the vector space over the field $\mathbb{Q}$ can either be the $\mathbb{Q}$-vector subspace of $\mathbb{Q}^m$ or a free $\mathbb{Z}$-submodule of $\mathbb{Z}^m$.

```
sage: a = matrix(ZZ,5,3,[1,1,122,-1,-2,1,-188,2,1,1,-10,1,-1,-1,-1])
sage: a.kernel()
Free module of degree 5 and rank 2 over Integer Ring
Echelon basis matrix:
[   1  979  -11 -279  811]
[   0 2079  -22 -569 1488]
sage: b = a.base_extend(QQ)
sage: b.kernel()
Vector space of degree 5 and dimension 2 over Rational Field
Basis matrix:
[        1         0  -121/189 -2090/189   6949/63]
[        0         1    -2/189 -569/2079   496/693]
sage: b.integer_kernel()
Free module of degree 5 and rank 2 over Integer Ring
Echelon basis matrix:
[   1  979  -11 -279  811]
[   0 2079  -22 -569 1488]
```

8.2.3 Eigenvalues, Jordan Form and Similarity Transformation

A square matrix A is the representation of a linear operator, an endomorphism, in a given basis. Any change of basis corresponds to a similarity transformation

of the form $B = U^{-1}AU$. The matrix B represents the linear operator in the new basis and the two matrices A and B are then *similar*. The properties of the linear operator, which are independent of the basis of representation, are thus revealed by the study of the similarity invariants of the matrix, namely its properties that remain invariant under similarity.

Among these invariants, the most elementary are the rank and the determinant. Indeed, since the matrices U and U^{-1} are invertible, the rank of $U^{-1}AU$ is the rank of A. Moreover, $\det(U^{-1}AU) = \det(U^{-1})\det(A)\det(U) = \det(U^{-1}U)\det(A) = \det(A)$. Similarly, the characteristic polynomial of the matrix A, defined as $\chi_A(x) = \det(x\mathrm{Id} - A)$ is also invariant under similarity transformation:

$$\det(x\mathrm{Id} - U^{-1}AU) = \det(U^{-1}(x\mathrm{Id} - A)U) = \det(x\mathrm{Id} - A).$$

Consequently, the characteristic values of a matrix, defined as the roots of its characteristic polynomial in its splitting field, are thus also similarity invariants.

By definition, a scalar λ is an eigenvalue of a matrix A if there exists a non-zero vector u such that $Au = \lambda u$. The eigenspace associated with an eigenvalue λ is the set of all vectors u verifying $Au = \lambda u$. It is a linear subspace defined by $E_\lambda = \mathrm{Ker}(\lambda\mathrm{Id} - A)$.

Eigenvalues coincide with characteristic values:

$$\det(\lambda\mathrm{Id} - A) = 0 \Leftrightarrow \dim(\mathrm{Ker}(\lambda\mathrm{Id} - A)) \geq 1 \Leftrightarrow \exists u \neq 0, \lambda u - Au = 0.$$

These two points of view respectively correspond to the algebraic and the geometric approach to eigenvalues. The geometric viewpoint considers the action of the linear operator A on vectors in the ambient space with more precision than in the algebraic viewpoint. In particular, they differ in the notion of multiplicity of an eigenvalue: the algebraic multiplicity is the multiplicity of the root in the characteristic polynomial while the geometric multiplicity is the dimension of the eigenspace associated to the eigenvalue. When the matrix is diagonalisable, these notions are equivalent, but otherwise the geometric multiplicity is less than or equal to the algebraic multiplicity.

The geometric point of view gives finer details on the structure of the matrix. It also helps designing efficient algorithms to compute eigenvalues, eigenvectors and the characteristic and minimal polynomials.

Cyclic Invariant Subspace and Frobenius Normal Form. Let A be an $n \times n$ matrix over a field K and $u \in K^n$ a vector. The vectors $u, Au, A^2u, \ldots A^nu$, called the Krylov sequence, are linearly dependent (as it is a set of $n+1$ vectors of dimension n). Let d be the first index such that A^du is linearly dependent with its predecessors $u, Au, \ldots, A^{d-1}u$. We can write this linear dependence relation as

$$A^du = \sum_{i=0}^{d-1} \alpha_i A^i u.$$

The polynomial $\varphi_{A,u}(x) = x^d - \sum_{i=0}^{d-1} \alpha_i x^i$, satisfying the relation $\varphi_{A,u}(A)u = 0$ is therefore a monic polynomial annihilating the Krylov sequence, of minimal

degree. It is named the *minimal polynomial* of the vector u (with respect to the matrix A). The set of all annihilating polynomials of u forms an ideal of $K[X]$, generated by $\varphi_{A,u}$.

The minimal polynomial of the matrix A is defined as the least degree monic polynomial annihilating the matrix A: $\varphi_A(A) = 0$. In particular, applying the vector u to the right in this equation shows that φ_A is annihilating the Krylov sequence for u. It is therefore necessarily a multiple of the minimal polynomial of u. In addition, one can prove (see Exercise 31) that there always exists a vector $\overline{u}$ such that

$$\varphi_{A,\overline{u}} = \varphi_A. \tag{8.1}$$

When the vector u is chosen at random, the probability that it satisfies Equation (8.1) increases with the size of the field (one shows that it is at least $1 - \frac{n}{|K|}$).

Exercise 31. We will show that there always exists a vector $\overline{u}$ whose minimal polynomial coincides with the minimal polynomial of the matrix.

1. Let $(e_1, \ldots, e_n)$ be a basis of the vector space. Show that φ_A is equal to the least common multiple of the φ_{A,e_i} for all $1 \leq i \leq n$.
2. In the case where φ_A is an irreducible polynomial raised to some power, show that there is an index i_0 such that $\varphi_A = \varphi_{A,e_{i_0}}$.
3. Show that if the minimal polynomials $\varphi_i = \varphi_{A,e_i}$ and $\varphi_j = \varphi_{A,e_j}$ of the vectors e_i and e_j are coprime, then $\varphi_{A,e_i+e_j} = \varphi_i\varphi_j$.
4. Show that if $\varphi_A = P_1P_2$ where P_1 and P_2 are coprime, then there exist two vectors $x_1 \neq 0$ and $x_2 \neq 0$ such that P_i is the minimal polynomial of x_i for $i = 1, 2$.
5. Conclude using the factorisation of φ_A in irreducible factors $\varphi_A = \varphi_1^{m_1} \ldots \varphi_k^{m_k}$.
6. Illustration: let $A = \begin{bmatrix} 0 & 0 & 3 & 0 & 0 \\ 1 & 0 & 6 & 0 & 0 \\ 0 & 1 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 1 & 5 \end{bmatrix}$ be a matrix over GF(7). Compute the degrees of the minimal polynomial of A, of the minimal polynomials of the vectors $u = e_1$ and $v = e_4$ of the canonical basis, and of the vector $u + v$. One can use the method `maxspin(u)` applied to the transpose of the matrix A, returning the maximal sequence of linearly independent Krylov iterates of the vector u.

Let $P = x^k + \sum_{i=0}^{k-1} \alpha_i x^i$ be a monic polynomial of degree k. The companion matrix associated with the polynomial P is the $k \times k$ matrix

$$C_P = \begin{bmatrix} 0 & & & -\alpha_0 \\ 1 & & & -\alpha_1 \\ & \ddots & & \vdots \\ & & 1 & -\alpha_{k-1} \end{bmatrix}.$$

This matrix has the property that P equals its minimal polynomial and its characteristic polynomial.

PROPOSITION. Let K_u be the matrix formed by the d first Krylov iterates of a vector u. Then

$$AK_u = K_u C_{\varphi_{A,u}}.$$

Hence, when $d = n$, the matrix K_u is square and invertible. Therefore, it defines a similarity transformation $K_u^{-1} A K_u = C_{\varphi_{A,u}}$ reducing A to a companion matrix. Now this transformation preserves the determinant and thus also the characteristic polynomial. The coefficients of the characteristic and minimal polynomial (which are identical in this case) can therefore be read off directly from the last column of the companion matrix.

```
sage: A = matrix(GF(97), 4, 4,\
....:             [86,1,6,68,34,24,8,35,15,36,68,42,27,1,78,26])
sage: e1 = identity_matrix(GF(97),4)[0]
sage: U = matrix(A.transpose().maxspin(e1)).transpose()
sage: F = U^-1*A*U; F
```

$$\begin{pmatrix} 0 & 0 & 0 & 83 \\ 1 & 0 & 0 & 77 \\ 0 & 1 & 0 & 20 \\ 0 & 0 & 1 & 10 \end{pmatrix}$$

```
sage: K.<x> = GF(97)[]
sage: P = x^4-sum(F[i,3]*x^i for i in range(4)); P
```

$x^4 + 87x^3 + 77x^2 + 20x + 14$

```
sage: P == A.charpoly()
True
```

In the general case ($d \leq n$) the iterates $u, \ldots, A^{d-1}u$ form a basis of a linear subspace I invariant under the action of the matrix A (i.e., such that $AI \subseteq I$). This subspace is also called cyclic invariant subspace, as these vectors are obtained cyclically by applying the matrix A to the preceding vector. The dimension of this subspace is the degree of the minimal polynomial of u and is therefore bounded by the degree of the minimal polynomial of the matrix A. When the dimension is maximal, the space is generated by the Krylov iterates of the vector constructed in Exercise 31, which we will denote by u_1^*. It is called the first invariant subspace. Let V be the complementary subspace of this first invariant subspace. Computing *modulo* the first invariant subspace, i.e., by considering that two vectors are equal whenever their difference belongs to the first invariant subspace, one can define a second invariant subspace for vectors in this complementary subspace, as well as a minimal polynomial which is called the second similarity invariant. In this case we have a relation of the form:

$$A \begin{bmatrix} K_{u_1^*} & K_{u_2^*} \end{bmatrix} = \begin{bmatrix} K_{u_1^*} & K_{u_2^*} \end{bmatrix} \begin{bmatrix} C_{\varphi_1} & \\ & C_{\varphi_2} \end{bmatrix},$$

where φ_1, φ_2 are the first two similarity invariants and $K_{u_1^*}, K_{u_2^*}$ are the Krylov matrices corresponding to the two cyclic subspaces generated by the vectors u_1^* and u_2^*.

Iteratively, one can build a matrix $K = \begin{bmatrix} K_{u_1^*} & \cdots & K_{u_k^*} \end{bmatrix}$ that is square and invertible, and satisfies

$$K^{-1} A K = \begin{bmatrix} C_{\varphi_1} & & \\ & \ddots & \\ & & C_{\varphi_k} \end{bmatrix}. \tag{8.2}$$

As each u_i^* is annihilated by the φ_j for $j \leq i$, we have that $\varphi_i \mid \varphi_{i-1}$ for any $2 \leq i \leq k$. Equivalently, the sequence of the φ_i is totally ordered for division. One shows that for every matrix there exists a unique sequence of similarity invariants $\varphi_1, \ldots, \varphi_k$. Therefore, the block diagonal matrix $\mathrm{Diag}(C_{\varphi_1}, \ldots, C_{\varphi_k})$, similar to the matrix A and revealing these polynomials, is a normal form, called the rational canonical form or the Frobenius normal form.

THEOREM (Frobenius normal form). For every matrix A over a field, there is a unique matrix $F = \begin{bmatrix} C_{\varphi_1} & & \\ & \ddots & \\ & & C_{\varphi_k} \end{bmatrix}$, with $\varphi_{i+1} \mid \varphi_i$ for all $i < k$, similar to A.

Equation (8.2) shows that one can read off the bases of the invariant subspaces from the transformation matrix K.

REMARK.

$$\begin{aligned} \chi_A(x) &= \det(x\mathrm{Id} - A) = \det(K)\det(x\mathrm{Id} - F)\det(K^{-1}) \\ &= \prod_{i=1}^{k} \det(x\mathrm{Id} - C_{\varphi_i}) = \prod_{i=1}^{k} \varphi_i(x). \end{aligned}$$

Hence, the minimal polynomial φ_1 is a divisor of the characteristic polynomial, which therefore annihilates the matrix A.

In Sage, one can compute the Frobenius normal form over $\mathbb{Q}$ of matrices with coefficients over $\mathbb{Z}$ with the method `frobenius`[1]:

```
sage: A = matrix(ZZ,8,[[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()
```

$$\begin{pmatrix} 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 4 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \end{pmatrix}$$

One can also obtain the list of similarity invariants, by passing `1` as argument. In order to obtain information on the associated invariant subspaces, one passes `2` as argument, which will produce the transformation matrix K. It is a basis of the whole space, decomposed into the direct sum of the invariant subspaces.

```
sage: A.frobenius(1)
```

$$\left[x^4 - x^2 - 4x - 4, x^3 - x^2 - 4, x - 2\right]$$

[1] It is a slight abuse in the interface of the software: although the Frobenius normal form is defined for any matrix over a field, Sage only allows to compute it with integer matrices, implicitly embedding them over $\mathbb{Q}$.

```
sage: F,K = A.frobenius(2)
sage: K
```

$$\begin{pmatrix} 1 & -\frac{15}{56} & \frac{17}{224} & \frac{15}{56} & -\frac{17}{896} & 0 & -\frac{15}{112} & \frac{17}{64} \\ 0 & \frac{29}{224} & -\frac{13}{224} & -\frac{23}{448} & -\frac{17}{896} & -\frac{17}{896} & \frac{29}{448} & \frac{13}{128} \\ 0 & -\frac{75}{896} & \frac{75}{896} & -\frac{47}{896} & 0 & -\frac{17}{896} & -\frac{23}{448} & \frac{11}{128} \\ 0 & \frac{17}{896} & -\frac{29}{896} & \frac{15}{896} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & -\frac{4}{21} & -\frac{4}{21} & -\frac{10}{21} & 0 & 0 & -\frac{2}{21} & 1 \end{pmatrix}$$

```
sage: K^-1*F*K == A
True
```

These results implicitly assume that the matrix A is embedded in the fraction field $\mathbb{Q}$. In order to study the action of the matrix A on the free module $\mathbb{Z}^n$, and make explicit the corresponding decomposition of the module, the method `decomposition` can be used. However, further explanations on this method would go beyond the scope of this book.

Invariant Factors and Similarity Invariants. There is a fundamental property relating the similarity invariants and the invariant factors, mentioned in Section 8.2.1.

THEOREM. The similarity invariants of a matrix A over a field F correspond to the invariant factors of its characteristic matrix $x\mathrm{Id} - A$ over the ring $F[x]$.

The proof of this theorem goes well beyond the scope of this book and we will only illustrate it on the previous example.

```
sage: S.<x> = QQ[]
sage: B = x*identity_matrix(8) - A
sage: B.elementary_divisors()
```

$$\left[1, 1, 1, 1, 1, x-2, x^3-x^2-4, x^4-x^2-4x-4\right]$$

```
sage: A.frobenius(1)
```

$$\left[x^4-x^2-4x-4, x^3-x^2-4, x-2\right]$$

Eigenvalues, Eigenvectors. Considering the factorisation of the minimal polynomial into irreducible factors, $\varphi_1 = \psi_1^{m_1} \dots \psi_s^{m_s}$, then every invariant factor can be written in the form $\varphi_i = \psi_1^{m_{i,1}} \dots \psi_s^{m_{i,s}}$, with multiplicities $m_{i,j} \leq m_j$. One can show that there is a similarity transformation, replacing each companion block C_{φ_i} in the Frobenius normal form by a diagonal block $\mathrm{Diag}(C_{\psi_1^{m_{i,1}}}, \dots, C_{\psi_s^{m_{i,s}}})$. This variant of the Frobenius normal form, called intermediate form, is still composed of companion blocks but each of which now corresponds to an irreducible polynomial raised to some power.

$$F = \begin{bmatrix} \boxed{\begin{matrix} C_{\psi_1^{m_{1,1}}} & & \\ & \ddots & \\ & & C_{\psi_s^{m_{1,s}}} \end{matrix}} & & & \\ & \boxed{\begin{matrix} C_{\psi_1^{m_{2,1}}} & & \\ & \ddots & \\ & & \end{matrix}} & & \\ & & \ddots & \\ & & & \boxed{\begin{matrix} C_{\psi_1^{m_{k,1}}} & & \\ & \ddots & \\ & & \end{matrix}} \end{bmatrix} \tag{8.3}$$

When an irreducible factor ψ_i has degree 1 and multiplicity 1, its companion block is a 1×1 matrix on the main diagonal and thus corresponds to an eigenvalue. When the minimal polynomial splits and is square-free, the matrix is diagonalisable.

The eigenvalues are obtained with the method `eigenvalues`. The methods `eigenvectors_right` and `eigenvectors_left` list for each eigenvalue, the right (respectively left) eigenvectors associated together with the multiplicity of the eigenvalue. Lastly, the eigenspaces together with a basis of eigenvectors are returned by the methods `eigenspaces_right` and `eigenspaces_left`.

```
sage: A = matrix(GF(7),4,[5,5,4,3,0,3,3,4,0,1,5,4,6,0,6,3])
sage: A.eigenvalues()
[4, 1, 2, 2]
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])
```

```
]
```

More concisely, the method `eigenmatrix_right` returns the tuple of the diagonalised matrix and the matrix of the corresponding right eigenvectors. The `eigenmatrix_left` does similarly with the left eigenvectors.

```
sage: A.eigenmatrix_right()
```

$$\left(\begin{pmatrix} 4 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 1 & 0 \\ 5 & 1 & 3 & 0 \\ 5 & 1 & 0 & 1 \\ 1 & 4 & 1 & 1 \end{pmatrix}\right)$$

Jordan Normal Form. When the minimal polynomial splits over the base field, but has factors with multiplicity greater than 1, the intermediate form (8.3) is not diagonal. One can show that there is no similarity transformation making it diagonal, hence the matrix is not diagonalisable. However it can be trigonalised, which means upper triangular, with eigenvalues on the main diagonal. Among all possible such upper triangular matrices, the most reduced one is the Jordan normal form. A Jordan block $J_{\lambda,k}$, associated with an eigenvalue λ and of order k, is the $k \times k$ matrix $J_{\lambda,k}$ given by

$$J_{\lambda,k} = \begin{bmatrix} \lambda & 1 & & \\ & \ddots & \ddots & \\ & & \lambda & 1 \\ & & & \lambda \end{bmatrix}.$$

This matrix plays a similar role as the companion blocks, revealing more precisely the multiplicity of an eigenvalue. Indeed, its characteristic polynomial is $\chi_{J_{\lambda,k}} = (X-\lambda)^k$. Moreover, its minimal polynomial also equals $\varphi_{J_{\lambda,k}} = (X-\lambda)^k$: it is necessarily a multiple of $P = X - \lambda$, now the matrix

$$P(J_{\lambda,k}) = \begin{bmatrix} 0 & 1 & & \\ & \ddots & \ddots & \\ & & 0 & 1 \\ & & & 0 \end{bmatrix}$$

is nilpotent of order k, hence $\varphi_{J_{\lambda,k}} = \chi_{J_{\lambda,k}} = (X-\lambda)^k$. The Jordan normal form corresponds to the intermediate form (8.3), where the companion blocks of the $\psi_j^{m_{i,j}}$ have been replaced by the Jordan blocks $J_{\lambda_j,m_{i,j}}$ (recall that since the minimal polynomial splits, the ψ_j are of the form $X - \lambda_j$). As a consequence, every matrix whose minimal polynomial splits is similar to a Jordan matrix of

the form

$$J = \begin{bmatrix} \boxed{\begin{matrix} J_{\lambda_1,m_{1,1}} & & \\ & \ddots & \\ & & J_{\lambda_s,m_{1,s}} \end{matrix}} & & & \\ & \boxed{\begin{matrix} J_{\lambda_1,m_{2,1}} & \\ & \ddots \end{matrix}} & & \\ & & \ddots & \\ & & & \boxed{\begin{matrix} J_{\lambda_1,m_{k,1}} & \\ & \ddots \end{matrix}} \end{bmatrix}. \tag{8.4}$$

In particular, over any algebraically closed field, such as $\mathbb{C}$, the Jordan normal form always exists. In Sage, the constructor `jordan_block(a,k)` produces the Jordan block $J_{a,k}$. The Jordan normal form is obtained by the method `jordan_form`. The option `transformation=True` makes the method also return the transformation matrix U such that $U^{-1}AU$ is in Jordan normal form.

```
sage: A = matrix(ZZ,4,[3,-1,0,-1,0,2,0,-1,1,-1,2,0,1,-1,-1,3])
sage: A.jordan_form()
```

$$\left(\begin{array}{c|c|cc} 3 & 0 & 0 & 0 \\ \hline 0 & 3 & 0 & 0 \\ \hline 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 2 \end{array}\right)$$

```
sage: J,U = A.jordan_form(transformation=True)
sage: U^-1*A*U == J
True
```

The Jordan normal form is unique up to a permutation of the Jordan blocks. Depending on the bibliographic references, one sometimes imposes that their order respects the order of the similarity invariants, as in equation (8.4). Remark, from the above example, that Sage does not respect this order, since the first similarity invariant (the minimal polynomial) is the polynomial $(X-3)(X-2)^2$.

Primary Normal Form. For the sake of completeness, we should mention a last normal form, generalising the Jordan form in the case where the minimal polynomial does not split. For an irreducible polynomial P of degree k, one defines the Jordan block of multiplicity m as the $km \times km$ matrix

$$J_{P,m} = \begin{bmatrix} C_P & B & & \\ & \ddots & \ddots & \\ & & C_P & B \\ & & & C_P \end{bmatrix}$$

where B is the $k \times k$ matrix whose coefficients are all zero except $B_{k,1} = 1$, and where C_P is the companion matrix associated to the polynomial P (§8.2.3). Note that when $P = X - \lambda$, this definition coincides with the notion of Jordan block

associated with the eigenvalue λ. One shows similarly that the minimal and characteristic polynomials of this matrix are

$$\chi_{J_{P,m}} = \varphi_{J_{P,m}} = P^m.$$

As a consequence, there exists a similarity transformation replacing each companion block ${C_{\psi_j}}^{m_{i,j}}$ in the intermediate form (8.3) with a Jordan block $J_{\psi_j, m_{i,j}}$. The resulting matrix is called the primary form or also the second Frobenius form. It is again a normal form, unique up to a permutation of the diagonal blocks.

The uniqueness of these normal forms is used for instance to test whether two matrices are similar, and in such a case, to produce a similarity transformation from one to the other.

Exercise 32. Write a program testing whether two input matrices A and B are similar and returning the transformation matrix U such that $A = U^{-1}BU$ (one can return `None` instead in the case where the two matrices are not similar).

9 Polynomial Systems

This chapter completes the two preceding ones. The objects we consider are systems of equations in several variables, as in Chapter 8. These equations, as in Chapter 7, are polynomial. Compared to univariate polynomials, those with several variables yield nice mathematical properties, but also new difficulties, related in particular to the fact that the ring $K[x_1, \ldots, x_n]$ is not principal. The theory of Gröbner bases provides tools to overcome this limitation. In the end, we have at our disposal powerful methods to study polynomial systems, with uncountable applications in various domains.

A large part of the chapter only requires basic knowledge on multivariate polynomials. Some parts are however at the level of a commutative algebra course of third or fourth year at university. For more details, a very good and complete reference is the book of Cox, Little and O'Shea [CLO07].

9.1 Polynomials in Several Variables

9.1.1 The Rings $A[x_1, \ldots, x_n]$

We consider here polynomials in several indeterminates or variables, also called multivariate polynomials.

Similarly to other algebraic structures available in Sage, before being able to construct polynomials, we have to define a family of indeterminates living all in the same ring. The syntax is almost the same as with a single variable (cf. §7.1.1):

```
sage: R = PolynomialRing(QQ, 'x,y,z')
sage: x,y,z = R.gens() # gives the tuples of indeterminates
```

or in short:

```
sage: R.<x,y,z> = QQ[]
```

(or even `R = QQ['x,y,z']`). The `PolynomialRing` constructor also allows to create a family of indeterminates with the same name, and integer indices:

```
sage: R = PolynomialRing(QQ, 'x', 10)
```

Assigning the n-tuple returned by `gens` to the variable `x` then allows to easily access the indeterminate x_i via `x[i]`:

```
sage: x = R.gens()
sage: sum(x[i] for i in xrange(5))
x0 + x1 + x2 + x3 + x4
```

The order of the variables matters. The comparison with `==` between `QQ['x,y']` and `QQ['y,x']` returns false, and a given polynomial prints differently if seen as element of the former or of the latter:

```
sage: def test_poly(ring, deg=3):
....:     monomials = Subsets(
....:         flatten([(x,)*deg for x in (1,) + ring.gens()]),
....:         deg, submultiset=True)
....:     return add(mul(m) for m in monomials)
```

```
sage: test_poly(QQ['x,y'])
x^3 + x^2*y + x*y^2 + y^3 + x^2 + x*y + y^2 + x + y + 1
sage: test_poly(QQ['y,x'])
y^3 + y^2*x + y*x^2 + x^3 + y^2 + y*x + x^2 + y + x + 1
sage: test_poly(QQ['x,y']) == test_poly(QQ['y,x'])
True
```

Exercise 33. Explain the behaviour of the `test_poly` function defined above.

More generally, writing polynomials in canonical form requires choosing a way to order their monomials. Ordering them by degree is natural for univariate polynomials, however for multivariate polynomials, no monomial order is satisfactory in all cases. Therefore, Sage allows us to choose between several orders, thanks to the `order` option of `PolynomialRing`. For example, the `deglex` order first ranks monomials according to their total degree, then by lexicographic order of the degrees of indeterminates in case of same total degree:

```
sage: test_poly(PolynomialRing(QQ, 'x,y', order='deglex'))
x^3 + x^2*y + x*y^2 + y^3 + x^2 + x*y + y^2 + x + y + 1
```

The main available orders are described in more detail in §9.3.1. We will see that the choice of the monomial order does not only determine the output, but also matters for some computations.

Exercise 34. Define the ring $\mathbb{Q}[x_2, x_3, \ldots, x_{37}]$ whose indeterminates are indexed by prime numbers less than 40, and the variables `x2`, `x3`, ..., `x37` to access the indeterminates.

Finally, it can be useful, in some cases, to play with multivariate polynomials in *recursive representation*, i.e., seen as elements of a polynomial ring with coefficients that are themselves polynomials (see the sidebar on page 130).

Construction of polynomial rings	
ring $A[x, y]$	`PolynomialRing(A, 'x,y')` *or* `A['x,y']`
ring $A[x_0, \dots, x_{n-1}]$	`PolynomialRing(A, 'x', n)`
ring $A[x_0, x_1, \dots, y_0, y_1, \dots]$	`InfinitePolynomialRing(A, ['x','y'])`
n-tuple of generators	`R.gens()`
1st, 2nd... generator	`R.0`, `R.1`, ...
indeterminates of $R = A[x, y][z][\dots]$	`R.variable_names_recursive()`
conversion $A[x_1, x_2, y] \to A[x_1, x_2][y]$	`p.polynomial(y)`

Access to coefficients	
support, non-zero coefficients	`p.exponents()`, `p.coefficients()`
coefficient of a monomial	`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	
transformation of coefficients	`p.map_coefficients(f)`
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(x=a, y=b)`
homogenisation	`p.homogenize()`
common denominator ($p \in \mathbb{Q}[x, y, \dots]$)	`p.denominator()`

TABLE 9.1 – Multivariate polynomials.

9.1.2 Polynomials

Just as univariate polynomials are in the class `Polynomial`, multivariate polynomials (in rings with a finite number of variables) are in the class `MPolynomial`[1]. For the usual base rings (like $\mathbb{Z}$, $\mathbb{Q}$ or $\mathbb{F}_q$), Sage calls the Singular computer algebra system, which is specialised in fast polynomial computations. In the other cases, a generic and much slower implementation is used.

Multivariate polynomials are always encoded in sparse representation[2]. Why this choice? A dense polynomial with n variables of total degree d contains $\binom{n+d}{d}$ monomials: for $n = d = 10$, it amounts to 184 756 coefficients to store! It is thus very difficult to manipulate large dense polynomials like we do with univariate ones. Besides, even when the polynomials are dense, the supports (the exponents of non-zero monomials) encountered in practice have various forms. If for example a polynomial with n variables, dense up to the total degree $d - 1$, is represented by a rectangular array $d \times \cdots \times d$, for large d, only about one coefficient over $n!$ is non-zero. On the contrary, the sparse representation by dictionary is well adapted to the shape of the support, and also to the monomial order.

[1] Contrary to `Polynomial`, this class is not directly available from the command line: we have to use its full name. For example, we can check whether an object is of type "multivariate polynomial" by `isinstance(p, sage.rings.polynomial.multi_polynomial.MPolynomial)`.

[2] The recursive representation (see sidebar page 130) yields nevertheless partially dense polynomials. In the memory representation of a polynomial from $A[x][y]$, each coefficient of y^k occupies (in general) a space proportional to its degree in x, to which we should add a space proportional to the degree in y for the polynomial itself.

The rings $A[(x_n, y_n, \dots)_{n\in\mathbb{N}}]$

It happens that we do not know, at the beginning of a computation, how many variables will be necessary. This makes the use of `PolynomialRing` rather painful: we first have to compute in a first domain, then extend it and convert all elements each time we want to introduce a new variable.

Polynomial rings with an infinite number of variables provide a more flexible data structure. Their elements can contain variables in one or several infinite families of indeterminates. Each generator of the ring corresponds not only to a single variable, but to a family of variables indexed by integers:

```
sage: R.<x,y> = InfinitePolynomialRing(ZZ, order='lex')
sage: p = mul(x[k] - y[k] for k in range(2)); p
x_1*x_0 - x_1*y_0 - x_0*y_1 + y_1*y_0
sage: p + x[100]
x_100 + x_1*x_0 - x_1*y_0 - x_0*y_1 + y_1*y_0
```

We get back to some usual polynomial ring `PolynomialRing` thanks to the `polynomial` method, which returns the image of an element from `InfinitePolynomialRing` in a sufficiently large ring to contain all elements of the ring with an infinite number of variables which have been produced so far. The obtained ring is generally not the smallest one with this property.

As a counterpart of this facility, these rings are less efficient than the rings `PolynomialRing`. Also, their *ideals* cannot replace those of usual polynomial rings for computations on polynomial systems, which is the main topic of this chapter.

9.1.3 Basic Operations

Let us fix the terminology. Let $R = A[x_1, \dots, x_n]$ be a polynomial ring. We call *monomial* an expression of the form $x_1^{\alpha_1} x_2^{\alpha_2} \cdots x_n^{\alpha_n}$, i.e., a product of indeterminates, and we note it $\boldsymbol{x}^{\boldsymbol{\alpha}}$ in short. The integer n-tuple $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \dots, \alpha_n)$ is the *exponent* of the monomial $\boldsymbol{x}^{\boldsymbol{\alpha}}$. A *term* is a monomial multiplied by an element of A, its *coefficient*.

Since there is no unique way to order the terms, the elements of R do not have, as mathematical objects, a dominant coefficient. However, once an order has been chosen at the ring construction, it is possible and useful to define a leading monomial, the leftmost one in the writing order. The methods `lm`, `lc` and `lt` of a multivariate polynomial return respectively its leading monomial, its leading coefficient, and the term they form together:

```
sage: R.<x,y,z> = QQ[]
sage: p = 7*y^2*x^2 + 3*y*x^2 + 2*y*z + x^3 + 6
sage: p.lt()
7*x^2*y^2
```

The arithmetic operations `+`, `-` and `*`, as well as the methods `coefficients`, `dict`, and several others, work like their univariate variants. Among the small

differences, the square-bracket operator `[]` to extract a coefficient accepts as parameter either a monomial, or its exponent:

```
sage: p[x^2*y] == p[(2,1,0)] == p[2,1,0] == 3
True
```

Likewise, the evaluation of a polynomial requires giving values to all variables, or to make explicit those to substitute:

```
sage: p(0, 3, -1)
0
sage: p.subs(x = 1, z = x^2+1)
2*x^2*y + 7*y^2 + 5*y + 7
```

The `subs` method might also substitute any number of variables at once, see its documentation for advanced examples. The degree might be either total or partial:

```
sage: print("total={d}    (in x)={dx}    partial={ds}"\
....:   .format(d=p.degree(), dx=p.degree(x), ds=p.degrees()))
total=4    (in x)=3    partial=(3, 2, 1)
```

Other constructions get trivial adaptations, for example, the `derivative` method takes as parameter the variable with respect to which we want to differentiate.

9.1.4 Arithmetic

Beyond syntactic and elementary arithmetic operations, available functions in Sage are in general limited to polynomials over a field, and sometimes over $\mathbb{Z}$ or $\mathbb{Z}/n\mathbb{Z}$. For the rest of this chapter, unless otherwise noted, we will consider polynomials over a field.

The Euclidean division of polynomials makes sense only in one variable. In Sage, the `quo_rem` method and the associated operators `//` and `%` remain nonetheless defined for multivariate polynomials. The "division with remainder" they compute satisfies

```
(p//q)*q + (p%q) == p
```

and matches the Euclidean division when p and q depend on one variable only, but it is not a Euclidean division and it is not canonical. It is however useful when the division is exact, or when the divisor is a monomial. In the other cases, we will prefer to `quo_rem` and its variants the `mod` method, described in §9.2.3, which reduces a polynomial modulo an ideal while taking into account the monomial order of the ring:

```
sage: R.<x,y> = QQ[]; p = x^2 + y^2; q = x + y
sage: print("({quo})*({q}) + ({rem}) == {p}".format( \
....:         quo=p//q, q=q, rem=p%q, p=p//q*q+p%q))
(-x + y)*(x + y) + (2*x^2) == x^2 + y^2
sage: p.mod(q)  # is NOT equivalent to p%q
2*y^2
```

Operations on multivariate polynomials	
divisibility $p \mid q$	`p.divides(q)`
factorisation	`p.factor()`
gcd, lcm	`p.gcd(q)`, `p.lcm(q)`
square-free test	`p.is_squarefree()`
resultant $\mathrm{Res}_x(p, q)$	`p.resultant(q, x)`

TABLE 9.2 – Arithmetic.

The methods `divides`, `gcd`, `lcm` or `factor` have the same meaning as in one variable. Since multivariate polynomial rings are not Euclidean in general, the first ones are not available for arbitrary coefficient rings, but work on usual fields, for example on number fields:

```
sage: R.<x,y> = QQ[exp(2*I*pi/5)][]
sage: (x^10 + y^5).gcd(x^4 - y^2)
x^2 + y
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)
```

9.2 Polynomial Systems and Ideals

We now consider the central topic of this chapter. Sections 9.2.1 and 9.2.2 give an overview of the different ways to find and understand the solutions of a system of polynomial equations with the help of Sage. Section 9.2.3 is devoted to ideals associated to these systems. The last sections come back in a more detailed manner on algebraic elimination and system solving tools.

9.2.1 A First Example

Let us revisit the polynomial system from Section 2.2,

$$\begin{cases} x^2yz = 18 \\ xy^3z = 24 \\ xyz^4 = 6. \end{cases} \tag{9.1}$$

The `solve()` function from Sage was only able to find numerical solutions. Let us now see how Sage is able to solve the system exactly, and, with a little help from the user, to find simple closed forms for all solutions[3].

Enumerating Solutions. Let us first translate the problem in more algebraic terms, by constructing the ideal of $\mathbb{Q}[x, y, z]$ generated by the equations:

[3]Our purpose being here to illustrate the tools to solve polynomial systems, we neglect the possibility of reducing (9.1) to linear equations by taking the logarithm!

```
sage: R.<x,y,z> = QQ[]
sage: J = R.ideal(x^2 * y * z - 18,
....:             x * y^3 * z - 24,
....:             x * y * z^4 - 6)
```

As we will see in Section 9.2.3, the following command enables us to check the ideal J is of dimension zero, i.e., the system (9.1) has a finite number of solutions in $\mathbb{C}^3$:

```
sage: J.dimension()
0
```

Once this is established, the first reflex should be to call the method `variety`, which computes all solutions of the system. Without any parameter, it gives the solutions in the base field of the polynomial ring:

```
sage: J.variety()
[{y: 2, z: 1, x: 3}]
```

The solution $(3, 2, 1)$ already found is thus the unique rational solution.

The next step is to enumerate the complex solutions. To perform this exactly, we work in the field of algebraic numbers. We find again the 17 solutions:

```
sage: V = J.variety(QQbar)
sage: len(V)
17
```

Explicitly, the last three have the following form:

```
sage: V[-3:]
[{z: 0.9324722294043558? - 0.3612416661871530?*I,
 y: -1.700434271459229? + 1.052864325754712?*I,
 x: 1.337215067329615? - 2.685489874065187?*I},
{z: 0.9324722294043558? + 0.3612416661871530?*I,
 y: -1.700434271459229? - 1.052864325754712?*I,
 x: 1.337215067329615? + 2.685489874065187?*I},
{z: 1, y: 2, x: 3}]
```

Each solution point is given by a dictionary whose keys are the generators of `QQbar['x,y,z']` (and not of `QQ['x,y,z']`, which explains the short detour below), and the associated coordinates of the point. Except for the rational solution already identified, the first coordinates are all algebraic numbers of degree 16:

```
sage: (xx, yy, zz) = QQbar['x,y,z'].gens()
sage: [ pt[xx].degree() for pt in V ]
[16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 1]
```

Computing with the Solutions and Identifying Their Structure. We have obtained an exact representation of the complex solutions from the system (9.1), however this representation is not really explicit. This is not a problem: having the coordinates as elements of `QQbar` is enough to pursue exact computations on these solutions.

For example, it is not difficult to see that if (x, y, z) is solution of the system (9.1), then so is $(|x|, |y|, |z|)$. Let us build the set of $(|x|, |y|, |z|)$ for (x, y, z) solution:

```
sage: Set(tuple(abs(pt[i]) for i in (xx,yy,zz)) for pt in V)
{(3, 2, 1)}
```

All the values of x (resp. y, z) have thus the same modulus. Even more, we can check that the substitution

$$(x, y, z) \mapsto (\omega x, \omega^9 y, \omega^6 z) \qquad \text{where} \qquad \omega = e^{2\pi i/17} \tag{9.2}$$

leaves the system invariant. In particular, the last coordinates of the solutions are exactly the seventeenth roots of unity, which we check again thanks to the possibility to compute exactly with algebraic numbers:

```
sage: w = QQbar.zeta(17); w  # primitive root of 1
0.9324722294043558? + 0.3612416661871530?*I
sage: Set(pt[zz] for pt in V) == Set(w^i for i in range(17))
True
```

The solutions of the system are therefore the triples $(3\omega, 2\omega^9, \omega^6)$ for $\omega^{17} = 1$. This is much more explicit!

Exercise 35. Look for real solutions (and not only rational ones) of (9.1), to check directly there is only $(x = 3, y = 2, z = 1)$. Find again the substitution (9.2), including the value 17 for the order of ω as root of unity, by a computation with Sage.

We could have reached the same result by examining the minimal polynomials of the coordinates of points of V. We see indeed that a given coordinate has the same minimal polynomial for all solution points, apart from $(3, 2, 1)$. The common minimal polynomial of the third coordinates is nothing else than the cyclotomic 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}
```

Those of the first and second coordinate are respectively $3^{16} \cdot \Phi_{17}(x/3)$ and $2^{16} \cdot \Phi_{17}(x/2)$.

Closed-Form Expressions. Getting an explicit form of the solutions is thus possible using the exponential notation of complex numbers:

```
sage: def polar_form(z):
....:     rho = z.abs(); rho.simplify()
....:     theta = 2 * pi * z.rational_argument()
....:     return (SR(rho) * exp(I*theta))
sage: [tuple(polar_form(pt[i]) for i in [xx,yy,zz]) for pt in V[-3:]]
[(3*e^(-6/17*I*pi), 2*e^(14/17*I*pi), e^(-2/17*I*pi)),
(3*e^(6/17*I*pi), 2*e^(-14/17*I*pi), e^(2/17*I*pi)), (3, 2, 1)]
```

Naturally, if we had had the idea of writing the elements of V in exponential notation, this would have been enough to conclude.

Simplifying the System. A different approach is possible. Instead of looking for solutions, let us try to compute a simpler form of the system itself. The fundamental tools offered by Sage for this purpose are the triangular decomposition and Gröbner bases. We will see later exactly what they compute; let us first use them on this example:

```
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]
```

We obtain in both cases the equivalent system

$$z^{17} = 1 \qquad y = 2z^{10} \qquad x = 3z^3,$$

or $x = 3y^2/4$ for the last equation with `transformed_basis`, i.e., $V = \{(3\omega^3, 2\omega^{10}, \omega) \mid \omega^{17} = 1\}$. This is an immediate parametrisation of the compact form of solutions found manually above.

9.2.2 What Does Solving Mean?

A polynomial system that has solutions often has an infinite number of solutions. The simple equation $x^2 - y = 0$ has an infinite number of solutions in $\mathbb{Q}^2$, not even considering $\mathbb{R}^2$ or $\mathbb{C}^2$. It is therefore not possible to enumerate them. The best we can do is to describe the set of solutions "as explicitly as possible", i.e., compute a representation of it from which we can easily extract useful information. The situation is analogous to linear systems, for which (in the homogeneous case) a basis of the system kernel is a good description of the set of solutions.

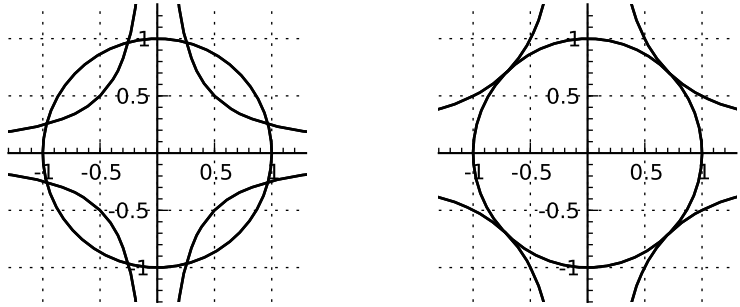
In the particular case where the number of solutions is finite, it becomes possible to "compute them". However, even in that case, do we want to enumerate the solutions in $\mathbb{Q}$, or in a finite field $\mathbb{F}_q$? To find real or complex numerical approximations? Or even, as in the example of last section, to represent them using algebraic numbers, i.e., to compute for example minimal polynomials of their coordinates?

This very example illustrates the fact that other representations of the set of solutions might be much more useful than a list of points, more so when the solutions are numerous. Therefore, enumerating the solutions is not always the best thing to do, even when possible. In the end, we do not really want to compute the solutions, but we want to compute *with* them, to deduce afterwards, according to the given problem, the information we are really interested in. The rest of this chapter investigates several useful tools for this purpose.

9.2.3 Ideals and Systems

If s polynomials $p_1, \ldots, p_s \in K[\boldsymbol{x}]$ vanish at a point $\boldsymbol{x}$ with coordinates in K or an extension of K, any element of the ideal they generate also vanishes at $\boldsymbol{x}$. It is thus natural to associate to the polynomial system

$$p_1(\boldsymbol{x}) = p_2(\boldsymbol{x}) = \cdots = p_s(\boldsymbol{x}) = 0$$



(a) (S_1) (b) (S_2)

```
sage: opts = {'axes':True, 'gridlines':True, 'frame':False,
....:    'aspect_ratio':1, 'axes_pad':0, 'fontsize':8,
....:    'xmin':-1.3, 'xmax':1.3, 'ymin':-1.3, 'ymax':1.3}
sage: (ideal(J.0).plot() + ideal(J.1).plot()).show(**opts)
```

FIGURE 9.1 – Intersection of two plane curves, see systems (9.3).

the ideal $J = \langle p_1, \dots, p_s \rangle \subset K[\boldsymbol{x}]$. Two polynomial systems generating the same ideal are equivalent in the sense that they share the same solutions. If L is a field containing K, we call *algebraic subvariety* of L^n associated to J the set

$$V_L(J) = \{\boldsymbol{x} \in L^n \mid \forall p \in J, p(\boldsymbol{x}) = 0\} = \{\boldsymbol{x} \in L^n \mid p_1(\boldsymbol{x}) = \cdots = p_s(\boldsymbol{x}) = 0\}$$

of solutions of the system with coordinates in L. Different ideals may have the same associated variety. For example, the equations $x = 0$ and $x^2 = 0$ have the same unique solution in $\mathbb{C}$, although we have $\langle x^2 \rangle \subsetneq \langle x \rangle$. The ideal generated by a polynomial system captures rather the notion of "solutions with multiplicities" (in the algebraic closure of K).

For instance, the following two systems both express the intersection of the unit circle and a curve of equation $\alpha\, x^2 y^2 = 1$, union of two equilateral hyperbolas (see Figure 9.1):

$$(S_1) \begin{cases} x^2 + y^2 = 1 \\ 16\, x^2 y^2 = 1 \end{cases} \qquad (S_2) \begin{cases} x^2 + y^2 = 1 \\ 4\, x^2 y^2 = 1. \end{cases} \tag{9.3}$$

The system (S_1) has eight solutions in $\mathbb{C}$, all with real coordinates. When we deform it into (S_2) by varying the parameter α, the two solutions on each branch of the hyperbola move closer until they match. The system (S_2) has then only four solutions, each one in some sense of "multiplicity two". By decreasing α further, we would have no more real solution, but eight complex solutions.

Computing Modulo an Ideal. Just as for univariate polynomials, Sage allows us to define ideals[4] $J \subset K[\boldsymbol{x}]$, quotient rings $K[\boldsymbol{x}]/J$, and to compute naturally

[4]Warning: the objects `InfinitePolynomialRing` also have an `ideal` method, however it does not have the same meaning as for usual polynomial ring. (An ideal of $K[(x_n)_{n\in\mathbb{N}}]$ has no reason

with elements of these quotient rings. The ideal J_1 associated to (S_1) is built with:

```
sage: R.<x,y> = QQ[]
sage: J = R.ideal(x^2 + y^2 - 1, 16*x^2*y^2 - 1)
```

We can then form the quotient of $K[\boldsymbol{x}]$ by J_1, and project polynomials on it, compute with equivalence classes modulo J_1, and "lift" them into representatives:

```
sage: ybar2 = R.quo(J)(y^2)
sage: [ybar2^i for i in range(3)]
[1, ybar^2, ybar^2 - 1/16]
sage: ((ybar2 + 1)^2).lift()
3*y^2 + 15/16
```

There is a theoretical issue here. The elements of $K[\boldsymbol{x}]/J$ are represented in normal form, which is necessary to be able to check the equality between two elements. However, this normal form is not easy to define, for the reason already mentioned in §9.1.4: the division of a representative of an equivalence class $p + J$ by a principal generator of J, used to compute in $K[x]/J$, has no direct analogue in several variables. Let us admit for now that a normal form nevertheless exists, which depends on the order on the elements chosen at the ring construction, and builds on a particular system of generators of J called Gröbner basis. Section 9.3 at the end of this chapter defines Gröbner bases and shows how to use them in computations. Sage automatically computes Gröbner bases when required; however, these computations are sometimes very expensive, in particular when the number of variables is large, which might make computations in a quotient ring somewhat difficult.

Let us go back to playing with Sage. When $p \in J$, the command `p.lift(J)` rewrites p as a linear combination of generators of J with polynomial coefficients:

```
sage: u = (16*y^4 - 16*y^2 + 1).lift(J); u
[16*y^2, -1]
sage: u[0]*J.0 + u[1]*J.1
16*y^4 - 16*y^2 + 1
```

For any polynomial p, the expression `p.mod(J)` gives the normal form of p modulo J, seen as element of $K[\boldsymbol{x}]$:

```
sage: (y^4).mod(J)
y^2 - 1/16
```

Beware: while `J.reduce(p)` is equivalent to `p.mod(J)`, the variant `p.reduce([p1, p2, ...])` returns a representative of $p + J$ which is not necessarily the normal form (see §9.3.2):

```
sage: (y^4).reduce([x^2 + y^2 - 1, 16*x^2*y^2 - 1])
y^4
```

By combining `p.mod(J)` and `p.lift(J)`, we can decompose a polynomial p into a linear combination of generators of J with polynomial coefficients, plus a remainder which is zero if and only if $p \in J$.

to be finitely generated!) The rest of the chapter does not apply to these objects.

Ideals	
ideal $\langle p_1, p_2\rangle \subset R$	`R.ideal(p1, p2)` *or* `(p1, p2)*R`
sum, product, power	`I + J`, `I * J`, `I^k`
intersection $I \cap J$	`I.intersection(J)`
quotient $I : J = \{p \mid pJ \subset I\}$	`I.quotient(J)`
radical $\sqrt{J}$	`J.radical()`
reduction modulo J	`J.reduce(p)` *or* `p.mod(J)`
section of $R \twoheadrightarrow R/J$	`p.lift(J)`
quotient ring R/J	`R.quo(J)` *or* `R.quotient(J)`
homogenised ideal	`J.homogenize()`

Some predefined ideals	
irrelevant ideal $\langle x_1, \ldots, x_n\rangle$	`R.irrelevant_ideal()`
Jacobian ideal $\langle \partial p/\partial x_i\rangle_i$	`p.jacobian_ideal()`
cyclic roots (9.11)	`sage.rings.ideal.Cyclic(R)`
field equations $x_i^q = x_i$	`sage.rings.ideal.FieldIdeal(GF(q)['x1,x2'])`

TABLE 9.3 – Ideals.

Radical of an Ideal and Solutions. The main correspondence between ideals and varieties lies in Hilbert's theorem of zeros, also known as *Nullstellensatz*. Let $\bar{K}$ be an algebraic closure of K.

THEOREM (*Nullstellensatz*). Let $p_1, \ldots, p_s \in K[\boldsymbol{x}]$, and let $Z \subset \bar{K}^n$ be the set of common zeros to the p_i. A polynomial $p \in K[\boldsymbol{x}]$ vanishes identically on Z if and only if there exists an integer k such that $p^k \in \langle p_1, \ldots, p_s\rangle$.

This result provides an algebraic criterion to check whether a polynomial system has some solutions. The constant polynomial 1 vanishes identically on Z if and only if Z is empty, thus the system $p_1(\boldsymbol{x}) = \cdots = p_s(\boldsymbol{x}) = 0$ has solutions in $\bar{K}$ if and only if the ideal $\langle p_1, \ldots, p_s\rangle$ does not contain 1. For example, the circles of radius 1 centered at $(0, 0)$ and $(4, 0)$ have a complex intersection:

```
sage: 1 in ideal(x^2+y^2-1, (x-4)^2+y^2-1)
False
```

However, after adding the condition $x = y$, the system does not have any solutions. We can give a trivial proof that it is inconsistent by exhibiting as *certificate* a combination of the equations that reduces to $1 = 0$ if they are satisfied. The computation

```
sage: R(1).lift(ideal(x^2+y^2-1, (x-4)^2+y^2-1, x-y))
[-1/28*y + 1/14, 1/28*y + 1/14, -1/7*x + 1/7*y + 4/7]
```

yields in our case the relation

$$\frac{1}{28}\Big((-y+2)(x^2+y^2-1) + (y+2)\big((x-4)^2+y^2-1\big) + (-4x+4y+16)(x-y)\Big) = 1.$$

In terms of ideals, the *Nullstellensatz* says that the set of polynomials vanishing identically on the variety $V_{\bar{K}}(J)$ associated to the ideal J is the *radical* of that ideal, defined by

$$\sqrt{J} = \{p \in K[\boldsymbol{x}] \mid \exists k \in \mathbb{N}, p^k \in J\}.$$

We have

$$V_{\bar{K}}(\sqrt{J}) = V_{\bar{K}}(J)$$

but intuitively, switching to the radical "forgets the multiplicities". Therefore, the ideal J_1 is its own radical (we say it is radical), whereas the ideal J_2 associated to (S_2) satisfies $J_2 \subsetneq \sqrt{J_2}$:

```
sage: J1 = (x^2 + y^2 - 1, 16*x^2*y^2 - 1)*R
sage: J2 = (x^2 + y^2 - 1,  4*x^2*y^2 - 1)*R
sage: J1.radical() == J1
True
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
False
```

Systems, ideals and cryptography

Some specific modules, `sage.rings.polynomial.multi_polynomial_sequence` and `sage.crypto.mq`, provide tools to manipulate polynomial systems by taking into account the particular form of equations, and not only the ideal they generate. This is useful to play with large structured systems, like those found in cryptography. The module `sage.crypto` also defines several polynomial systems associated to classical cryptographic constructions.

Operations on Ideals. It is also possible to compute with ideals themselves. Let us recall the definition of the sum of two ideals:

$$I + J = \{p + q \mid p \in I \text{ and } q \in J\} = \langle I \cup J \rangle.$$

It corresponds geometrically to the intersection of varieties:

$$V(I + J) = V(I) \cap V(J).$$

Hence, the ideal J_1 associated to (S_1) is the sum of $C = \langle x^2 + y^2 - 1 \rangle$ and $H = \langle 16\, x^2 y^2 - 1 \rangle$, which define respectively the circle and the double hyperbola. In Sage:

```
sage: C = ideal(x^2 + y^2 - 1); H = ideal(16*x^2*y^2 - 1)
sage: C + H == J1
True
```

This equality test is also based on computing a Gröbner basis.

Similarly, the intersection, the product and the quotient of ideals satisfy

$$\begin{aligned} I \cap J &= \{p \mid p \in I \text{ and } p \in J\} & V(I \cap J) &= V(I) \cup V(J) \\ I \cdot J &= \langle pq \mid p \in I, q \in J \rangle & V(I \cdot J) &= V(I) \cup V(J) \\ I : J &= \{p \mid pJ \subset I\} & V(I : J) &= \overline{V(I) \setminus V(J)} \end{aligned}$$

and are computed as indicated in Table 9.3. The notation $\bar{X}$ designs here the *Zariski closure* of X, i.e., the smallest algebraic variety containing X. For example, the curve of Figure 9.1a is the set of zeros of polynomials from $C \cap H$, and the quotient $(C \cap H) : \langle 4xy - 1 \rangle$ corresponds to the union of the circle with one of the two hyperbolas:

```
sage: CH = C.intersection(H).quotient(ideal(4*x*y-1)); CH
Ideal (4*x^3*y + 4*x*y^3 + x^2 - 4*x*y + y^2 - 1) of
Multivariate Polynomial Ring in x, y over Rational Field
sage: CH.gen(0).factor()
(4*x*y + 1) * (x^2 + y^2 - 1)
```

However, the curve obtained by removing from $V(H)$ a finite number of points is not an algebraic subvariety, so that:

```
sage: H.quotient(C) == H
True
```

Dimension. To each ideal of $J \subset K[\boldsymbol{x}]$ is also associated a *dimension*, which intuitively corresponds to the maximal "dimension" of the "components" of the variety $V(J)$ over an algebraically closed field[5]. We have for example:

```
sage: [J.dimension() for J in [J1, J2, C, H, H*J2, J1+J2]]
[0, 0, 1, 1, 1, -1]
```

Indeed, $V(J_1)$ and $V(J_2)$ have a finite number of points, $V(C)$ and $V(H)$ are curves, $V(H \cdot J_2)$ is the union of curves and isolated points, and $V(J_1 + J_2)$ is empty. The *zero-dimensional* systems, i.e., those that generate an ideal of dimension zero, or equivalently that have a finite number of solutions (over the algebraic closure of K), will be particularly studied in the rest of the chapter, since they are the systems we can "solve" the most explicitly.

9.2.4 Elimination

In a system of equations, *eliminating* a variable means finding "consequences", or better "all consequences", of the system which are independent of this variable. Said otherwise, we want to find equations satisfied by any solution, but which do not contain the eliminated variable, which makes them often easier to analyse.

[5] We give in §9.3.3 a more rigorous (but not necessarily clearer) definition. We refer the reader to the reference given at the beginning of the chapter for better explanations.

General polynomial systems: elimination, geometry	
elimination ideal $J \cap A[z,t] \subset K[x,y,z,t]$	`J.elimination_ideal(x, y)`
resultant $\mathrm{Res}_x(p,q)$	`p.resultant(q, x)`
dimension	`J.dimension()`
genus	`J.genus()`
Zero-dimensional systems	
solutions in $L \supseteq K$	`J.variety(L)`
dimension over K of the quotient	`J.vector_space_dimension()`
quotient basis	`J.normal_basis()`
triangular decomposition	`J.triangular_decomposition()`

TABLE 9.4 – Solving polynomial systems.

For example, we can eliminate x from the linear system

$$\begin{cases} 2x + \ y - 2z = 0 \\ 2x + 2y + \ z = 1 \end{cases} \tag{9.4}$$

by subtracting the first equation from the second one. It yields $y + 3z = 1$, which shows that any solution triple (x, y, z) of (9.4) is of the form $(x, 1 - 3z, z)$. We can then check that every "partial solution" $(1 - 3z, z)$ lifts to a (unique) solution $(\frac{5z-1}{2}, 1 - 3z, z)$ of (9.4). This illustrates that Gauss' pivoting algorithm solves linear systems by elimination, as opposed to, for example, Cramer's formulas.

Elimination Ideals. In the context of polynomial systems, the "consequences" of equations $p_1(\boldsymbol{x}) = \cdots = p_s(\boldsymbol{x}) = 0$ are elements of the ideal $\langle p_1, \ldots, p_s \rangle$. If J is an ideal of $K[x_1, \ldots, x_n]$, we call k-th *elimination ideal* of J the set

$$J_k = J \cap K[x_{k+1}, \ldots, x_n] \tag{9.5}$$

of elements of J which only contain the $n - k$ last variables. This is an ideal of $K[x_{k+1}, \ldots, x_n]$.

In Sage, the method `elimination_ideal` takes as input the list of variables to eliminate. Beware: it does not return $J_k \subset K[x_{k+1}, \ldots, x_n]$, but the ideal $\langle J_k \rangle$ of $K[x_1, \ldots, x_n]$ it generates. In the case of the linear system (9.4), we find

```
sage: R.<x,y,z> = QQ[]
sage: J = ideal(2*x+y-2*z, 2*x+2*y+z-1)
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
```

Mathematically, we interpret these results as follows: we have $J \cap \mathbb{Q}[y,z] = \langle y + 3z - 1 \rangle \subset \mathbb{Q}[y,z]$ and $J \cap \mathbb{Q}[z] = \mathbb{Q}[z]$, i.e., $\mathbb{Q}[z] \subset J$. (Indeed, the ideal $\langle 0 \rangle$

corresponds to the system reduced to the sole trivial equation $0 = 0$, of which any polynomial is solution.) This is clearly not a recommended way of solving linear systems: the specific tools discussed in Chapter 8 are much more efficient!

For a slightly less trivial example, let us go back to the system (S_1) from Section 9.2.3 (see Figure 9.1a):

```
sage: R.<x,y> = QQ[]
sage: J1 = ideal(x^2 + y^2 - 1, 16*x^2*y^2 - 1)
```

Eliminating y yields an ideal of $\mathbb{Q}[x]$ — therefore principal — generated by a polynomial g whose roots are the abscissas

$$\frac{\pm\sqrt{2 \pm \sqrt{3}}}{2}$$

of the eight solutions of (S_1):

```
sage: g = J1.elimination_ideal(y).gens(); g
[16*x^4 - 16*x^2 + 1]
sage: SR(g[0]).solve(SR(x)) # solves by radicals
[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)]
```

By re-injecting into (S_1) each of the found values of x, we obtain a (redundant) system of equations in y only, which allows to compute the corresponding values of y.

Eliminating = Projecting. The above example shows that eliminating y in a system corresponds geometrically to the *projection* π of the solution variety on a hyperplane of equation $y =$ constant. However, let us now consider separately the ideals $C = \langle x^2 + y^2 - 1 \rangle$ and $H = \langle 16\,x^2y^2 - 1 \rangle$ whose sum is J_1, and, once again, let us eliminate y:

```
sage: C.elimination_ideal(y).gens()
[0]
sage: H.elimination_ideal(y).gens()
[0]
```

Insofar as C is concerned, this is no surprise. The circle $\{(x, y) \in \mathbb{R}^2 \mid x^2 + y^2 = 1\}$ projects on $[-1; 1]$, however it is clear that any value of x can be "re-injected" in the unique equation $x^2 + y^2 - 1 = 0$, and the obtained equation in y has *complex* solutions. The elimination of y in C corresponds to the projection on the first coordinate of the complex circle $\{(x, y) \in \mathbb{C}^2 \mid x^2 + y^2 = 1\}$, which is $\mathbb{C}$ altogether.

The case of H is a bit more intricate. The equation $16\,x^2y^2 = 1$ has no solution, even complex, for $x = 0$. We have then

$$V_{\mathbb{C}}(H \cap \mathbb{Q}[x]) = \mathbb{C} \subsetneq \pi(V_{\mathbb{C}}(H)) = \mathbb{C} \setminus \{0\}.$$

Indeed, the projection of the hyperbola, $\mathbb{C} \setminus \{0\}$, is not an algebraic subvariety. Conclusion: elimination really corresponds to projection (over an algebraically closed field), but it does not compute the exact projection, only the Zariski closure of it.

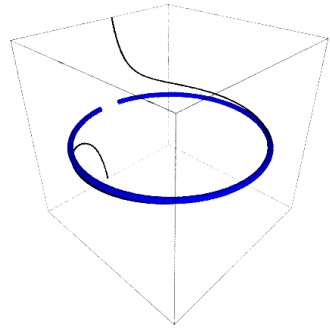


FIGURE 9.2 – A part of the curve in (x, y, t) defined by (9.6) and its projection on the plane $t = 0$.

Applications: Plane Geometry. If $X \subset \mathbb{C}^k$ is given by a rational parametrisation

$$X = \{\big(f_1(\boldsymbol{t}), f_2(\boldsymbol{t}), \dots, f_k(\boldsymbol{t})\big)\}, \qquad f_1, \dots, f_k \in \mathbb{Q}(t_1, \dots, t_n),$$

finding an implicit equation for X consists of projecting the part of $\mathbb{C}^{k+n}$ defined by the equations $x_i = f_i(\boldsymbol{t})$ on the subspace $(x_1, \dots, x_k) \simeq \mathbb{C}^k$. This is an elimination problem. Let us consider the classical parametrisation of the circle

$$x = \frac{1 - t^2}{1 + t^2} \qquad y = \frac{2t}{1 + t^2} \tag{9.6}$$

associated to the expression of $(\sin\theta, \cos\theta)$ in terms of $\tan(\theta/2)$. It translates into polynomial relations defining an ideal of $\mathbb{Q}[x, y, t]$:

```
sage: R.<x,y,t> = QQ[]
sage: Param = R.ideal((1-t^2)-(1+t^2)*x, 2*t-(1+t^2)*y)
```

Let us eliminate t:

```
sage: Param.elimination_ideal(t).gens()
[x^2 + y^2 - 1]
```

We obtain an equation of the circle. We can notice that this equation vanishes at $(x, y) = (-1, 0)$, although the parametrisation (9.6) does not hit that point, since the circle minus a point is not an algebraic subvariety.

Another example: let us draw a few of the circles $(\mathcal{C}_t)$ of equation

$$\mathcal{C}_t : \qquad x^2 + (y - t)^2 = \frac{t^2 + 1}{2} \tag{9.7}$$

using Sage commands (see Figure 9.3):

```
sage: R.<x,y,t> = QQ[]
sage: eq = x^2 + (y-t)^2 - 1/2*(t^2+1)
```

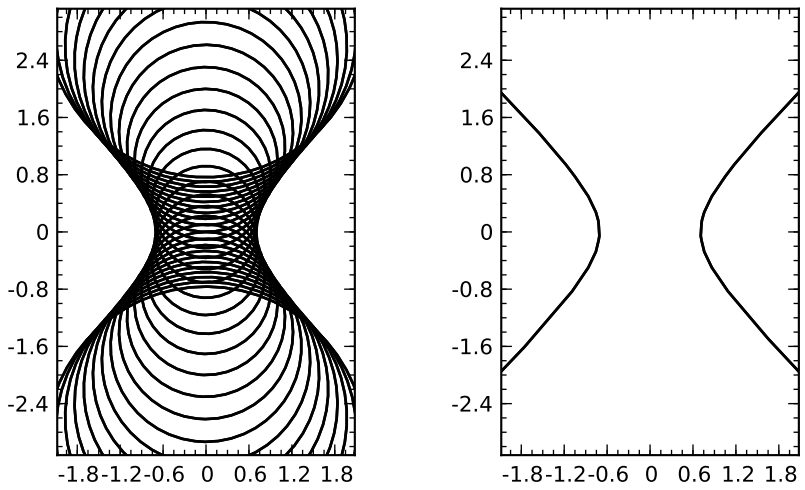


FIGURE 9.3 – A family of circles and its envelope.

```
sage: fig = add((eq(t=k/5)*QQ[x,y]).plot() for k in (-15..15))
sage: fig.show(aspect_ratio=1, xmin=-2, xmax=2, ymin=-3, ymax=3)
```

We see that the *envelope* of the family of circles $(\mathcal{C}_t)$ appears, a "limit curve" tangent to all $\mathcal{C}_t$, which we can describe informally as the set of "intersection points of circles infinitely close" to the family.

More precisely, if f is a differentiable function, and if the curve $\mathcal{C}_t$ is defined by $f(x, y, t) = 0$ for any t, the envelope of $(\mathcal{C}_t)$ is the set of points (x, y) such that

$$\exists t, \qquad f(x, y, t) = 0 \qquad \text{and} \qquad \frac{\partial f}{\partial t}(x, y, t) = 0. \tag{9.8}$$

In the case of circles (9.7), the function $f(x, y, t)$ is a polynomial. Their envelope is the projection on the (x, y) plane of the solutions from (9.8), thus we can determine an equation of it via the following elimination computation:

```
sage: env = ideal(eq, eq.derivative(t)).elimination_ideal(t)
sage: env.gens()
[2*x^2 - 2*y^2 - 1]
```

It remains only to draw the curve found:

```
sage: env.change_ring(QQ[x,y]).plot((x,-2,2),(y,-3,3))
```

Resultant and Elimination. The elimination performed in the preceding examples is implicitly based on Gröbner bases computed automatically by Sage. Yet, we have already encountered in this book another elimination tool: the resultant.

Let us consider two non constant polynomials $p, q \in K[x_1, \ldots, x_n, y]$. We denote by $\mathrm{Res}_y(p, q)$ the resultant of p and q, considered as polynomials in the

Inequalities

Let us consider the triangle with vertices $A = (0,0)$, $B = (1,0)$ and $C = (x,y)$. Assume the angles $\widehat{BAC}$ and $\widehat{CBA}$ are equal, and let us try to prove computationally that we have an isosceles triangle. By introducing the parameter $t = \tan\hat{A} = \tan\hat{B}$, the problem is encoded by the equations $y = tx = t(1-x)$, and we have to show that they imply

$$x^2 + y^2 = (1-x)^2 + y^2.$$

With Sage, we obtain:

```
sage: R.<x,y,t> = QQ[]
sage: J = (y-t*x, y-t*(1-x))*R
sage: (x^2+y^2) - ((1-x)^2+y^2) in J
False
```

This could have been expected: when $x = y = t = 0$, the hypotheses are satisfied, but the conclusion is false! Geometrically, we have to exclude the case of flat triangles, which can have two equal angles without being isosceles.

How to encode the constraint $t \neq 0$? The trick is to introduce an auxiliary variable u, and to force $tu = 1$. The computation becomes:

```
sage: R.<x,y,t,u> = QQ[]
sage: J = (y-t*x, y-t*(1-x), t*u-1)*R
sage: (x^2+y^2) - ((1-x)^2+y^2) in J
True
```

and we now have the expected result. Let us remark by the way that we can simultaneously force several expressions to not vanish with only one auxiliary variable, using an equation like $t_1 t_2 \cdots t_n u = 1$.

single variable y, with coefficients in $K[x_1, \ldots, x_n]$. We have seen in §7.3.3 that it is a polynomial from $K[x_1, \ldots, x_n]$, which vanishes at $\boldsymbol{u} \in K^n$ if and only if $p(u_1, \ldots, u_n, y)$ and $q(u_1, \ldots, u_n, y)$ (which are two polynomials of $K[y]$) have a common zero, except maybe when the leading coefficients (in y) of p and q themselves vanish at $\boldsymbol{u}$.

Some of our elimination computations involving only two polynomials can be replaced by resultants. For example, the equation of the envelope of circles (9.7) is

```
sage: eq.derivative(t).resultant(eq, t)
x^2 - y^2 - 1/2
```

The resultant $\mathrm{Res}_y(p,q)$ is an element of the elimination ideal $\langle p, q\rangle \cap K[x_1, \ldots, x_n]$. Even in the case $n = 1$ (where the elimination ideal is principal), though, and even if the leading coefficients with respect to y of p and q are coprime, the resultant does not necessarily generate the elimination ideal:

```
sage: R.<x,y> = QQ[]
```

```
sage: p = y^2 - x; q = y^2 + x
sage: p.resultant(q, y)
4*x^2
sage: ideal(p, q).elimination_ideal(y)
Ideal (x) of Multivariate Polynomial Ring in x, y over Rational Field
```

9.2.5 Zero-Dimensional Systems

We can deal with many problems just with the computation of elimination ideals, and Sage does not provide other "black-box" tools to solve general polynomial systems. The situation is somewhat different for zero-dimensional systems.

An ideal $J \subset K[\boldsymbol{x}]$ is said to have *dimension zero* when the quotient $K[\boldsymbol{x}]/J$ is a vector space of finite dimension. *Over an algebraically closed field*, it is equivalent to say that the variety $V(J)$ contains a finite number of points. For example, the systems (9.1) and (9.3) generate ideals of dimension zero — we say they are themselves zero-dimensional. On the contrary, the ideal $\langle (x^2+y^2)(x^2+y^2+1)\rangle$ of $\mathbb{Q}[x,y]$ is of dimension 1, despite its only real solution being $(0,0)$:

```
sage: R.<x,y> = QQ[]
sage: ((x^2 + y^2)*(x^2 + y^2 + 1)*R).dimension()
1
```

Zero-dimensional systems can be solved more explicitly than what is possible with the general tools from the previous section. We have already seen several of these methods in practice on the example of §9.2.1.

Enumerating the Solutions. First, having a finite number of solutions enables us to enumerate them, exactly or approximately.

The Sage expression `J.variety(L)` computes the variety $V_L(J)$. It raises an error if J is not zero-dimensional. By default, it looks for solutions with coordinates in the base field of the polynomial ring over which the system is defined. For example, the subvariety of $\mathbb{Q}^n$ defined by J_1 is empty:

```
sage: R.<x,y> = QQ[]
sage: J1 = (x^2 + y^2 - 1, 16*x^2*y^2 - 1)*R
sage: J1.variety()
[]
```

But, like the `roots` method for univariate polynomials, `variety` works for any kind of domain L. The most important case for now is the field of algebraic numbers. We can indeed show that the solutions of a zero-dimensional system with coefficients in K have coordinates in the algebraic closure of K. Therefore, it is possible to compute exactly the complex variety $V_{\mathbb{C}}(J) = V_{\bar{\mathbb{Q}}}(J)$ associated to an ideal $J \subset \mathbb{Q}[\boldsymbol{x}]$:

```
sage: J1.variety(QQbar)[0:2]
[{y: -0.9659258262890683?, x: -0.2588190451025208?},
{y: -0.9659258262890683?, x: 0.2588190451025208?}]
```

Exercise 36. Show that the solutions of (S_1) have coordinates in $\mathbb{Q}[\sqrt{2-\sqrt{3}}]$, and give them in terms of radicals.

Triangular Decomposition. Internally, `J.variety(L)` goes through a *triangular decomposition* of the ideal J. This decomposition is interesting in itself, since it sometimes gives a description of the variety J which is better for the rest of the computation, or even easier to grasp than the output of `variety` (see §9.2.1), particularly in case of numerous solutions.

A polynomial system is called *triangular* when of the following form

$$\begin{cases} p_1(x_1) & := & x_1^{d_1} + a_{1,d_1-1}\, x_1^{d_1-1} + \cdots + a_{1,0} & = & 0 \\ p_2(x_1, x_2) & := & x_2^{d_2} + a_{2,d_2-1}(x_1)\, x_2^{d_2-1} + \cdots + a_{2,0}(x_1) & = & 0 \\ & \vdots & & & \\ p_n(x_1, \ldots, x_n) & := & x_n^{d_n} + a_{n,d_n-1}(x_1, \ldots, x_{n-1})\, x_n^{d_n-1} + \cdots & = & 0 \end{cases}$$

or said otherwise, if each polynomial p_i only involves the variables $x_1, \ldots, x_i$, and is monic in the variable x_i. When a zero-dimensional system has such a form, its resolution reduces to a finite number of univariate polynomial equations to solve: it suffices to find the roots x_1 of p_1, to substitute them into p_2, to then find the roots x_2 of the latter, and so on. This strategy works both when looking for exact and approximate (numerical) solutions.

Not every system is equivalent to a triangular system. Consider for example the ideal J defined by:

```
sage: R.<x,y> = PolynomialRing(QQ, order='lex')
sage: C = ideal(x^2+y^2-1)
sage: D = ideal((x+y-1)*(x+y+1))
sage: J = C + D
```

For an image, see Figure 9.4 (left):

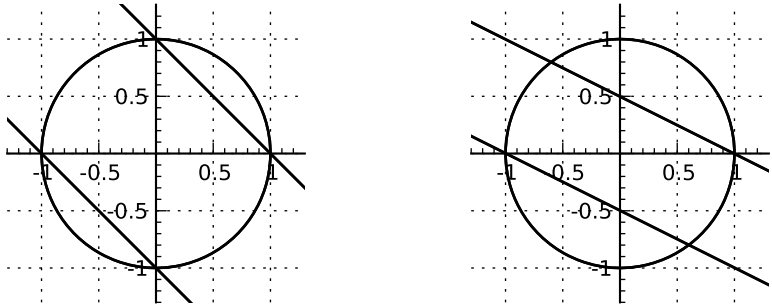
```
sage: opts = {'axes':True, 'gridlines':True, 'frame':False,
....: 'aspect_ratio':1, 'axes_pad':0, 'xmin':-1.3, 'xmax':1.3,
....: 'ymin':-1.3, 'ymax':1.3, 'fontsize': 8}
sage: show(C.plot() + D.plot(), figsize=[2,2], **opts)
```

The variety $V(J)$ contains two points of abscissa 0 but only one point of abscissa -1, and likewise, one point of ordinate -1 against two points of zero ordinate. Hence the ideal J cannot be described by a triangular system.

We can however show that any zero-dimensional ideal can be written as a finite intersection of ideals generated by triangular systems. The `triangular_decomposition` method computes such a decomposition:

```
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]
```

Geometrically, we obtain a representation of the variety $V(J)$ as a union of varieties associated to simpler systems, and often simple enough to give a good description of the solutions.



$\langle x^2+y^2-1, (x+y-1)(x+y+1)\rangle$ $\langle x^2+y^2-1, (x+2y-1)(x+2y+1)\rangle$

FIGURE 9.4 – In each case, the variety associated to the ideal J from the text is the intersection of a circle and the union of two lines.

Some Difficulties. We can rightfully wonder about the interest of the triangular decomposition to enumerate the solutions. After all, given 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 ideal. By substituting its roots into the system, we decrease the number of variables, which allows to iterate the process, until we have completely solved the system.

However, the "substitution" in the system by propagating the partial results might be intricate. Let us slightly modify the preceding system:

```
sage: D = ideal((x+2*y-1)*(x+2*y+1)); J = C + D
sage: J.variety()
[{y: -4/5, x: 3/5}, {y: 0, x: -1}, {y: 0, x: 1}, {y: 4/5, x: -3/5}]
sage: [T.gens() for T in J.triangular_decomposition()]
[[y, x^2 - 1], [25*y^2 - 16, 4*x + 3*y]]
```

The shape of the triangular decomposition remains the same: for each component, we have an equation involving y only, and a second equation enabling to express x in terms of y.

Thus let us eliminate x, to get an equation in y only, which is the product of the two above equations in y:

```
sage: Jy = J.elimination_ideal(x); Jy.gens()
[25*y^3 - 16*y]
```

To find x, it now suffices to substitute the roots of this equation into the equations defining the ideal J. The first equation, $x^2+y^2-1=0$, yields:

```
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)]
```

One of these two values is correct — we have $(-3/5, 4/5) \in V(J)$ — but the other one does not correspond to any solution: we have to check the found values using the second initial equation, $(x+2y-1)(x+2y+1)$, to eliminate it.

The problem becomes harder if one solves the univariate equations numerically, which is sometimes necessary due to the cost of operations on algebraic numbers:

```
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()]
[[(-0.5999999999999999 - 1.306289919090511e-16*I, 1),
 (0.6000000000000001 + 1.3062899190905113e-16*I, 1)],
 [(0.6000000000000001 - 3.1350958058172247e-16*I, 1),
  (2.600000000000001 + 3.135095805817224e-16*I, 1)]]
```

Here, by substituting $y \simeq -0.8$ into the two generators of J, we find two values of x near from 0.6. How to ensure they are approximations of the coordinate x of the same exact solution $(x, y) \simeq (0.6, -0.8)$, and not some spurious roots as in the preceding example? This phenomenon gets trickier when the number of variables and equations grows. However, when the system is triangular, only one equation has to be considered at each substitution step, and since this equation is monic, the numerical approximations do not change the number of solutions.

Let us continue. For the following system, `J.variety()` computes (exactly) a triangular decomposition of J, then finds numerically the real solutions of the obtained system(s). This yields a unique real solution:

```
sage: R.<x,y> = QQ[]; J = ideal([ x^7-(100*x-1)^2, y-x^7+1 ])
sage: J.variety(RealField(51))
[{y: 396340.890166545, x: -14.1660266425312}]
```

Yet, by performing the computation exactly until the end, we see there are three real solutions, and the value of x in the above numerical solution is completely wrong:

```
sage: J.variety(AA)
[{x: 0.00999999900000035?, y: -0.999999999999990?},
{x: 0.01000000100000035?, y: -0.999999999999990?},
{x: 6.3055689986413 85?, y: 396340.8901665450?}]
```

Conclusion: the triangular decomposition does not solve all problems, and we should be careful in the interpretation of approximate computations.

A large number of other methods exist to parametrise and approximate the solutions of zero-dimensional systems, more or less well suited to a given problem, which are not implemented within Sage. Exercise 37 gives an overview of some ideas used.

Advanced mathematics

Sage also provides many functions for commutative algebra and algebraic geometry, which go beyond the scope of this book. We invite the interested reader to explore the documentation of the polynomial ideals, and that of the `sage.schemes` module. Other functionalities are also available through the interfaces of the specialised tools Singular, CoCoA and Macaulay2.

Quotient Algebra. The quotients by zero-dimensional ideals are much easier to manipulate than those by general ideals, since the computations in the quotient algebra reduce to linear algebra in finite dimension.

If $J \subset K[\boldsymbol{x}]$ is a zero-dimensional ideal, the dimension $\dim_K K[\boldsymbol{x}]/J$ of the quotient algebra as a K-vector space bounds the number of points of $V(J)$. (Indeed, for any $u \in V(J)$, there exists a polynomial with coefficients in K which equals 1 at u and 0 at any other point of $V(J)$. Two such polynomials cannot be equivalent modulo J.) We can consider this dimension as the number of solutions "with multiplicity" of the system over the algebraic closure of K. For example, we have noticed that the four solutions of system (S_2) introduced in §9.2.3 are each one the "double" intersection of both curves. This explains the following:

```
sage: len(J2.variety(QQbar)), J2.vector_space_dimension()
(4, 8)
```

The `normal_basis` method computes a list of monomials whose projections on $K[\boldsymbol{x}]/J$ constitute a basis:

```
sage: J2.normal_basis()
[x*y^3, y^3, x*y^2, y^2, x*y, y, x, 1]
```

The returned basis depends on the monomial order chosen at the construction of the polynomial ring; we will describe it more precisely in §9.3.3.

Exercise 37. Let J be a zero-dimensional ideal of $\mathbb{Q}[x, y]$. Let χ_x be the characteristic polynomial of the linear transformation

$$\begin{array}{rccc} m_x : & \mathbb{Q}[x,y]/J & \to & \mathbb{Q}[x,y]/J \\ & p + J & \mapsto & xp + J. \end{array}$$

Compute χ_x in the case $J = J_2 = \langle x^2 + y^2 - 1, 4x^2y^2 - 1\rangle$. Show that every root of χ_x is the abscissa of a point of the variety $V_{\mathbb{C}}(J)$.

9.3 Gröbner Bases

So far, we have used as black boxes the functions provided by Sage for the algebraic elimination and the resolution of polynomial systems. This section introduces some of the underlying mathematical and algorithmic tools. The goal is both to be able to call directly these tools, and to make a wise use of the high-level functions seen before.

The methods used by Sage for computation with ideals and elimination are based on the concept of Gröbner basis. We can consider a Gröbner basis as a multivariate extension of the representation by principal generator of ideals of $K[x]$. The main problem of this section is to define and compute a normal form for the elements of quotient algebras from $K[\boldsymbol{x}]$. Our point of view remains that of the user: we define Gröbner bases, we show how to obtain them with Sage and how they can be useful, but we do not discuss the algorithms used to compute them.

Main monomial orders, with the example of $\mathbb{Q}[x,y,z]$	
`lex`	$\boldsymbol{x}^{\boldsymbol{\alpha}} < \boldsymbol{x}^{\boldsymbol{\beta}} \iff \alpha_1 < \beta_1$ or $(\alpha_1 = \beta_1$ and $\alpha_2 < \beta_2)$ or $\dots$ or $(\alpha_1 = \beta_1, \dots, \alpha_{n-1} = \beta_{n-1}$ and $\alpha_n < \beta_n)$ $x^3 > x^2y > x^2z > x^2 > xy^2 > xyz > xy > xz^2 > xz > x > y^3 > y^2z > y^2 > yz^2 > yz > y > z^3 > z^2 > z > 1$
`invlex`	$\boldsymbol{x}^{\boldsymbol{\alpha}} < \boldsymbol{x}^{\boldsymbol{\beta}} \iff \alpha_n < \beta_n$ or $(\alpha_n = \beta_n$ and $\alpha_{n-1} < \beta_{n-1})$ or $\dots$ or $(\alpha_n = \beta_n, \dots, \alpha_2 = \beta_2$ and $\alpha_1 < \beta_1)$ $z^3 > yz^2 > xz^2 > z^2 > y^2z > xyz > yz > x^2z > xz > z > y^3 > xy^2 > y^2 > x^2y > xy > y > x^3 > x^2 > x > 1$
`deglex`	$\boldsymbol{x}^{\boldsymbol{\alpha}} < \boldsymbol{x}^{\boldsymbol{\beta}} \iff \lvert\boldsymbol{\alpha}\rvert < \lvert\boldsymbol{\beta}\rvert$ or $(\lvert\boldsymbol{\alpha}\rvert = \lvert\boldsymbol{\beta}\rvert$ and $\boldsymbol{x}^{\boldsymbol{\alpha}} <_{\texttt{lex}} \boldsymbol{x}^{\boldsymbol{\beta}})$ $x^3 > x^2y > x^2z > xy^2 > xyz > xz^2 > y^3 > y^2z > yz^2 > z^3 > x^2 > xy > xz > y^2 > yz > z^2 > x > y > z > 1$
`degrevlex`	$\boldsymbol{x}^{\boldsymbol{\alpha}} < \boldsymbol{x}^{\boldsymbol{\beta}} \iff \lvert\boldsymbol{\alpha}\rvert < \lvert\boldsymbol{\beta}\rvert$ or $(\lvert\boldsymbol{\alpha}\rvert = \lvert\boldsymbol{\beta}\rvert$ and $\boldsymbol{x}^{\boldsymbol{\alpha}} >_{\texttt{invlex}} \boldsymbol{x}^{\boldsymbol{\beta}})$ $x^3 > x^2y > xy^2 > y^3 > x^2z > xyz > y^2z > xz^2 > yz^2 > z^3 > x^2 > xy > y^2 > xz > yz > z^2 > x > y > z > 1$

Construction of monomial orders	
object representing a predefined order on n variables	`TermOrder('nom', n)`
matrix order: $\boldsymbol{x}^{\boldsymbol{\alpha}} <_M \boldsymbol{x}^{\boldsymbol{\beta}} \iff M\boldsymbol{\alpha} <_{\texttt{lex}} M\boldsymbol{\beta}$	`TermOrder(M)`
blocks: $\boldsymbol{x}^{\boldsymbol{\alpha}}\boldsymbol{y}^{\boldsymbol{\beta}} < \boldsymbol{x}^{\boldsymbol{\gamma}}\boldsymbol{y}^{\boldsymbol{\delta}} \iff \boldsymbol{\alpha} <_1 \boldsymbol{\gamma}$ or $(\boldsymbol{\alpha} = \boldsymbol{\gamma}, \quad \boldsymbol{\beta} <_2 \boldsymbol{\delta})$	`T1 + T2`

TABLE 9.5 – Monomial orders.

9.3.1 Monomial Orders

A *monomial order* or *admissible order* is a total order on the monomials $\boldsymbol{x}^{\boldsymbol{\alpha}}$ of a polynomial ring, which satisfies

$$\boldsymbol{x}^{\boldsymbol{\alpha}} < \boldsymbol{x}^{\boldsymbol{\beta}} \implies \boldsymbol{x}^{\boldsymbol{\alpha}+\boldsymbol{\gamma}} < \boldsymbol{x}^{\boldsymbol{\beta}+\boldsymbol{\gamma}} \qquad \text{and} \qquad \boldsymbol{\gamma} \neq 0 \implies 1 < \boldsymbol{x}^{\boldsymbol{\gamma}} \tag{9.9}$$

for all exponents $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}$. Equivalently, we can consider $<$ as an order on the exponents $\boldsymbol{\alpha} \in \mathbb{N}^n$ or on the terms $c\,\boldsymbol{x}^{\boldsymbol{\alpha}}$. The leading monomial, leading coefficient and leading term of a polynomial p (see §9.1.3) for the current monomial order are those of largest exponent; we denote them respectively by $\operatorname{lm} p$, $\operatorname{lc} p$ and $\operatorname{lt} p$.

The first condition in (9.9) states that the monomial order should be compatible with products: multiplying by a fixed monomial does not change the order. The second condition implies that $<$ is a well-order, i.e., an infinite sequence of decreasing monomials does not exist. Let us notice that the only monomial order on $K[x]$ is the usual one $x^n > x^{n-1} > \cdots > 1$.

We have seen in §9.1.1 that Sage allows an order to be chosen when defining a polynomial ring via constructions like

```
sage: R.<x,y,z,t> = PolynomialRing(QQ, order='lex')
```

Table 9.5 lists the main predefined monomial orders[6]: `lex` is the lexicographic order of the exponents, `invlex` is the lexicographic order of exponents read

[6]Sage also allows orders (called "local") where 1 is the largest monomial instead of the smallest one. For example, in the order `neglex` on $\mathbb{Q}[x,y,z]$, we have $1 > z > z^2 > z^3 > y >$

from right to left, and `deglex` sorts the monomials first by total degree, then by lexicographic order. The definition of `degrevlex` is slightly more complex: the monomials are sorted by total degree, then by *decreasing* lexicographic order of the exponents *read from the right*. This strange order is nevertheless used by default when we omit the `order` option, since it is more efficient than other orders for some computations.

We generally choose (but not always!) simultaneously the order of variables of the ring and that of monomials such that $x_1 > x_2 > \cdots > x_n$, and we then often speak, for example, of the "`lex` order such that $x > y > z$" instead of the "`lex` order on $K[x, y, z]$". The predefined orders `lex`, `deglex` and `degrevlex` obey that convention; for the `invlex` order on $K[x, y, z]$, it is also the `lex` order such that $z > y > x$, i.e., the `lex` order on $K[z, y, x]$.

9.3.2 Division by a Family of Polynomials

A monomial order $<$ being fixed, let $G = \{g_1, g_2, \ldots, g_s\}$ be a finite set of polynomials from $K[\boldsymbol{x}]$. We denote by $\langle G\rangle = \langle g_1, g_2, \ldots, g_s\rangle$ the ideal of $K[\boldsymbol{x}]$ generated by G.

The division of a polynomial $p \in K[\boldsymbol{x}]$ by G is a multivariate analogue of the Euclidean division in $K[x]$. Like the latter, it associates to p a remainder, given in Sage by the expression `p.reduce(G)`, which is a "smaller" polynomial belonging to the same equivalence class modulo $\langle G\rangle$:

```
sage: ((x+y+z)^2).reduce([x-t, y-t^2, z^2-t])
2*z*t^2 + 2*z*t + t^4 + 2*t^3 + t^2 + t
```

The remainder is obtained by subtracting from p, while possible, multiples of elements of G whose leading term cancels a term of p in the subtraction. Contrary to the univariate case, it might happen that one can thus cancel a term of p, but not the leading one: we then only require to cancel the largest term according to the monomial order.

Formally, for $p \in K[\boldsymbol{x}]$, let us denote by $\mathrm{lt}_G\, p$ the term of p of maximal exponent, which is divisible by a leading term of an element of G. Let us call *elementary reduction* each transformation of the form

$$p \mapsto \tilde{p} = p - c\,\boldsymbol{x}^{\boldsymbol{\alpha}}\, g, \qquad \text{where } g \in G \text{ and } \mathrm{lt}_G\, p = c\,\boldsymbol{x}^{\boldsymbol{\alpha}}\ \mathrm{lt}\, g. \tag{9.10}$$

An elementary reduction leaves unchanged the equivalence class of p modulo $\langle G\rangle$, and makes the largest cancelled monomial of p disappear: we have

$$\tilde{p} - p \in \langle G\rangle \qquad \text{and} \qquad \mathrm{lt}_G\, \tilde{p} < \mathrm{lt}_G\, p.$$

Since $<$ is a well-order, it is not possible to apply to a polynomial an infinite number of successive elementary reductions. Each sequence of elementary reductions ends

$yz > yz^2 > y^2 > y^2z > y^3 > x > xz > xz^2 > xy > xyz > xy^2 > x^2 > x^2z > x^2y > x^3$. The local orders are not well-orders in the sense of definition (9.9), and we do not use them in this book; however the curious reader will complete Table 9.5 by using the `test_poly` function defined in §9.1.1.

thus on a polynomial that cannot be reduced further, and which is the remainder of the division.

Let us notice that this process generalises some familiar elimination methods both for univariate polynomials and linear systems. In one variable, the division of a polynomial p by a singleton $G = \{g\}$ reduces exactly to the Euclidean division of p by g. In the other extreme case of multivariate polynomials, but whose monomials are all of degree 1, it becomes identical to the elementary reduction of the Gauss-Jordan method.

But contrary to what happens in those particular cases, in general, the remainder depends on the choice of the elementary reductions. (We then say that the system (9.10) of rewriting rules is not confluent.) Thus, changing the order in which we give the elements of G leads in the following example to different choices of reduction:

```
sage: R.<x,y> = PolynomialRing(QQ, order='lex')
sage: (g, h) = (x-y, x-y^2);  p = x*y - x
sage: p.reduce([g, h])  # two reductions by h
y^3 - y^2
sage: p.reduce([h, g])  # two reductions by g
y^2 - y
```

Even if the elements of G are considered in a deterministic order (such that the result is unique for given p and G), how to ensure, for example, that the chosen sequence of elementary reductions of p by $\{g, h\}$ will discover the following relation, which shows that $p \in \langle g, h\rangle$?

```
sage: p - y*g + h
0
```

9.3.3 Gröbner Bases

The limitations of multivariate division explain the difficulty mentioned in §9.2.3 to obtain a normal form for the elements of the algebras $K[\boldsymbol{x}]/J$: dividing by the generators of the ideal is not enough... At least in general! Indeed, some particular systems of generators exist for which the division is confluent, and computes a normal form. These systems are called *Gröbner bases*.

Staircases. A pleasant way to grasp Gröbner bases goes through the notion of ideal staircase. Let us attach to each non-zero polynomial from $K[x_1, \ldots, x_n]$ a point of $\mathbb{N}^n$ given by its leading exponent, and let us draw the part $E \subset \mathbb{N}^n$ occupied by an ideal J (see Figures 9.5 to 9.7). The resulting graph (which depends on the monomial order) has a staircase shape: indeed, we have $\boldsymbol{\alpha} + \mathbb{N}^n \subset E$ for any $\boldsymbol{\alpha} \in E$. The elements of $J \setminus \{0\}$ are in the grey zone, above the staircase or at its frontier. The points strictly "under the staircase" correspond exclusively to polynomials from $K[\boldsymbol{x}] \setminus J$, but not all the polynomials from $K[\boldsymbol{x}] \setminus J$ are under the staircase.

For example, in a polynomial of the ideal $\langle x^3, xy^2z, xz^2\rangle$, each monomial, either a leading monomial or not, is multiple of one of the polynomials x^3, xy^2z

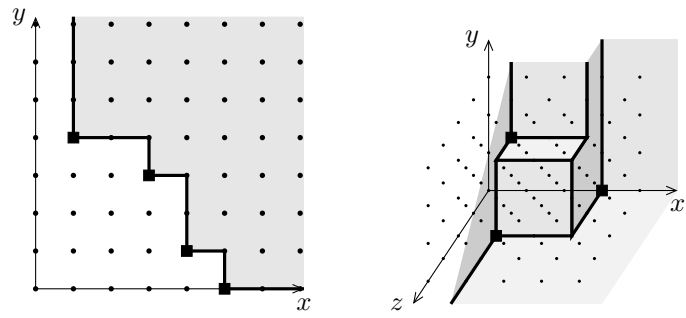


FIGURE 9.5 – Ideal staircases generated by monomials.

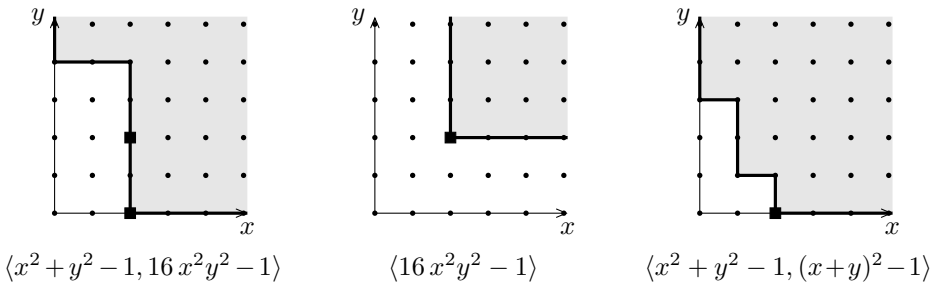


FIGURE 9.6 – Ideal staircases of $\mathbb{Q}[x, y]$ encountered in this chapter. In the three cases, the staircases and the location of generators are the same for the monomial orders `lex`, `deglex` and `degrevlex`.

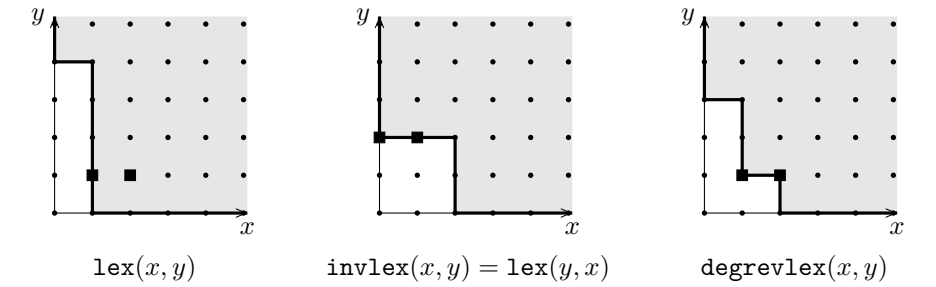


FIGURE 9.7 – Staircases of the ideal $\langle xy + x + y^2 + 1, x^2y + xy^2 + 1\rangle \subset \mathbb{Q}[x, y]$ relative to different monomial orders.

In each diagram, the grey zone corresponds to the leading terms of *elements* of the ideal. The black squares give the location of *generators* used to describe it.

and xz^2. The leading monomials are thus exactly the $x^\alpha y^\beta z^\gamma$ verifying one of the inequalities $(\alpha,\beta,\gamma) \geq (3,0,0)$, $(\alpha,\beta,\gamma) \geq (1,2,1)$ or $(\alpha,\beta,\gamma) \geq (1,0,2)$ component by component (Figure 9.5). A polynomial whose leading exponent does not satisfy these conditions, for example x^2+xz^2 if the monomial order is the lexicographic one with $x > y > z$, cannot belong to that ideal. Some polynomials like $x^3 + x$ are not in the ideal either, despite a leading monomial above the staircases. The situation is analogous for any ideal generated by monomials.

For a random ideal, the staircase structure cannot be easily read on the generators. For instance, by denoting $\boldsymbol{\delta}_1, \ldots, \boldsymbol{\delta}_s$ the leading exponents of the generators, we have $\bigcup_{i=1}^{s}(\boldsymbol{\delta}_i + \mathbb{N}^n) \subsetneq E$ in all examples of Figures 9.6 and 9.7 except the second one. We can nevertheless show that E can always be written as a finite union of sets of the form $\boldsymbol{\alpha} + \mathbb{N}^n$, i.e., intuitively, that the staircase has a finite number of corners. This result is sometimes called Dickson's lemma.

Gröbner Bases. A Gröbner basis is simply a family of generators that captures the shape of the staircase, more precisely that contains a polynomial corresponding to each corner.

Definition. A Gröbner basis of an ideal $J \subset K[\boldsymbol{x}]$ relative to a monomial order $<$ is a finite part G of J such that for any non-zero $p \in J$, there exists $g \in G$ whose leading monomial $\operatorname{lm} g$ (for the order $<$) divides $\operatorname{lm} p$.

Checking whether the generators defining an ideal form a Gröbner basis is done in Sage with the `basis_is_groebner` method. We have already noticed that any set of monomials is a Gröbner basis:

```
sage: R.<x,y> = PolynomialRing(QQ, order='lex')
sage: R.ideal(x*y^4, x^2*y^3, x^4*y, x^5).basis_is_groebner()
True
```

However, the system $\{x^2 + y^2 - 1, 16\,x^2y^2 - 1\}$ which encodes the intersection of the circle and the hyperbolas of Figure 9.1a is not a Gröbner basis:

```
sage: R.ideal(x^2+y^2-1, 16*x^2*y^2-1).basis_is_groebner()
False
```

According to the staircase shape (Figure 9.6), it lacks a polynomial from J_1 of leading monomial y^4.

The reasoning based on Dickson's lemma, mentioned above, shows that every ideal has Gröbner bases[7]. Let us compute Gröbner bases of J_1 and of the other ideals whose staircases are shown in Figure 9.6. In the case of J_1, it yields:

```
sage: R.ideal(x^2+y^2-1, 16*x^2*y^2-1).groebner_basis()
[x^2 + y^2 - 1, y^4 - y^2 + 1/16]
```

The leading monomials x^2 and y^4 appear as expected. Their presence explains the way the staircase closes itself on the axes; we will see it is characteristic of zero-dimensional systems. For the double hyperbola alone, we find:

[7] We can see this result as an effective version of the Hilbert Basis Theorem, which states that ideals from $K[\boldsymbol{x}]$ are generated by a finite number of elements. A classical proof of this theorem is very similar to the construction of a Gröbner basis for the lexicographic order.

```
sage: R.ideal(16*x^2*y^2-1).groebner_basis()
[x^2*y^2 - 1/16]
```

i.e., a multiple of the generator. In general, each singleton is a Gröbner basis by itself. The third example shows that a Gröbner basis might contain more polynomials than a system of generators:

```
sage: R.ideal(x^2+y^2-1, (x+y)^2-1).groebner_basis()
[x^2 + y^2 - 1, x*y, y^3 - y]
```

Due to the simplicity of the previous examples, these three Gröbner bases do not depend much, if at all, on the monomial order. The general situation is quite different. Figure 9.7 represents the staircases associated to the same ideal from $\mathbb{Q}[x, y]$ for three classical monomial orders. Corresponding Gröbner bases are:

```
sage: R_lex.<x,y> = PolynomialRing(QQ, order='lex')
sage: J_lex = (x*y+x+y^2+1, x^2*y+x*y^2+1)*R_lex; J_lex.gens()
[x*y + x + y^2 + 1, x^2*y + x*y^2 + 1]
sage: J_lex.groebner_basis()
[x - 1/2*y^3 + y^2 + 3/2, y^4 - y^3 - 3*y - 1]
```

```
sage: R_invlex = PolynomialRing(QQ, 'x,y', order='invlex')
sage: J_invlex = J_lex.change_ring(R_invlex); J_invlex.gens()
[y^2 + x*y + x + 1, x*y^2 + x^2*y + 1]
sage: J_invlex.groebner_basis()
[y^2 + x*y + x + 1, x^2 + x - 1]
```

```
sage: R_drl = PolynomialRing(QQ, 'x,y', order='degrevlex')
sage: J_drl = J_lex.change_ring(R_drl); J_drl.gens()
[x*y + y^2 + x + 1, x^2*y + x*y^2 + 1]
sage: J_drl.groebner_basis()
[y^3 - 2*y^2 - 2*x - 3, x^2 + x - 1, x*y + y^2 + x + 1]
```

The Gröbner basis for the `lex` order clearly demonstrates the rewriting rule $x = \frac{1}{2}\,y^3 - y^2 - \frac{3}{2}$, thanks to which we can express the elements of the quotient algebra in terms of the variable y only. The stretched out form of the corresponding staircase translates this rule. Similarly, the Gröbner basis for the `invlex` order indicates that we can eliminate powers of y via the equality $y^2 = -xy - x - 1$. We will come back to this at the end of next section.

9.3.4 Gröbner Basis Properties

Gröbner bases are used to implement the operations studied in Section 9.2. We use them in particular to compute normal forms for ideals in polynomial rings and for elements in quotients by these ideals, to eliminate variables in polynomial systems, or to determine characteristics of the solutions such as their dimension.

Reduction	
multivariate division of p by G	`p.reduce(G)`
inter-reduced generators	`J.interreduced_basis()`
Gröbner bases	
Gröbner basis test	`J.basis_is_groebner()`
(reduced) Gröbner basis	`J.groebner_basis()`
change of order towards `lex`	`J.transformed_basis()`
change of order $R_1 \to R_2$	`J.transformed_basis('fglm', other_ring=R2)`

TABLE 9.6 – Gröbner bases.

Division by a Gröbner Basis. Division by a Gröbner basis G of a polynomial from $\langle G \rangle$ cannot end on a non-zero element of $\langle G \rangle$. This is an immediate consequence of the definition: indeed, such an element would be above the staircase associated to $\langle G \rangle$, thus still divisible by G. Hence every element from $\langle G \rangle$ reduces to zero in the division by G. In particular, a Gröbner basis of an ideal J generates J.

Likewise, the division of a polynomial $p \notin J$ by a Gröbner basis of J can only end on a polynomial "under the staircase", moreover two distinct polynomials "under the staircase" belong to different equivalence classes modulo J (since their difference is still "under the staircase"). The division by a Gröbner basis therefore provides a normal form for the elements of the quotient $K[\boldsymbol{x}]/J$, and this holds independently of the order in which we perform the elementary reductions. The normal form of an equivalence class $p + J$ is its unique representative under the staircase, or zero. This is the normal form computed by the operations in the quotient algebra presented in §9.2.3. To continue the example of Figure 9.7, the reduction

```
sage: p = (x + y)^5
sage: J_lex.reduce(p)
17/2*y^3 - 12*y^2 + 4*y - 49/2
```

decomposes into a Gröbner basis computation, followed by a division:

```
sage: p.reduce(J_lex.groebner_basis())
17/2*y^3 - 12*y^2 + 4*y - 49/2
```

The result of a projection onto the quotient is essentially the same:

```
sage: R_lex.quo(J_lex)(p)
17/2*ybar^3 - 12*ybar^2 + 4*ybar - 49/2
```

Naturally, changing the monomial order yields another normal form:

```
sage: R_drl.quo(J_drl)(p)
5*ybar^2 + 17*xbar + 4*ybar + 1
```

The monomials appearing in the normal form correspond to the points under the staircase.

Hence, the ideal J is zero-dimensional if and only if the number of points under its staircase is finite, and this number of points is the dimension of the quotient $K[\boldsymbol{x}]/J$. In this case, the basis returned by the method `normal_basis` described in §9.2.5 is simply the set of monomials under the staircase for the associated monomial order:

```
sage: J_lex.normal_basis()
[y^3, y^2, y, 1]
sage: J_invlex.normal_basis()
[x*y, y, x, 1]
sage: J_drl.normal_basis()
[y^2, y, x, 1]
```

Let us notice that the number of monomials under the staircase is independent of the monomial order.

Dimension. We are now equipped to give a general definition of the dimension of an ideal: J has dimension d when the number of points under the staircase, corresponding to monomials of total degree at most t, is of order t^d when $t \to \infty$. For example, the ideal $\langle 16\,x^2y^2 - 1\rangle$ (Figure 9.6) has dimension 1:

```
sage: ideal(16*x^2*y^2-1).dimension()
1
```

Indeed, the number of monomials m under the staircase such that $\deg_x m + \deg_y m \leq t$ equals $4t - 2$ for $t \geq 3$. Likewise, the two ideals of Figure 9.5 have respectively dimension 1 and 2. We can show that the dimension does not depend on the monomial order, and corresponds — degeneracies excepted — to the "geometric" dimension of the associated variety.

Reduced Bases. A finite set of polynomials containing a Gröbner basis is itself a Gröbner basis, therefore a non-zero ideal has an infinite number of Gröbner bases. A Gröbner basis $G = \{g_1, \ldots, g_s\}$ is called *reduced* when

- the leading coefficients of g_i are all 1 (and $0 \notin G$);
- and no term of g_i is reducible by the rest of the basis $G \setminus \{g_i\}$ in the sense of the rules (9.10).

With a fixed monomial order, each ideal has a unique reduced Gröbner basis. For example, the reduced Gröbner basis of the ideal $\langle 1\rangle$ is the singleton $\{1\}$, whatever the monomial order. The reduced Gröbner bases therefore provide a normal form for all ideals of $K[\boldsymbol{x}]$.

A reduced Gröbner basis is minimal in the sense that, if we remove any element, what remains is no longer a system of generators of the ideal. Concretely, it contains exactly one polynomial per "corner" of the staircase. It can be computed from any Gröbner basis G by replacing each element $g \in G$ by its remainder for the division by $G \setminus \{g\}$, and so on while possible. This is what the `interreduced_basis` method does. The polynomials reducing to zero are erased.

Elimination. The lexicographic monomial orders have the following fundamental property: *if G is a Gröbner basis for the lexicographic order of $J \subset K[x_1, \ldots, x_n]$, then the $G \cap K[x_{k+1}, \ldots, x_n]$ are Gröbner bases for the elimination ideals*[8] $J \cap K[x_{k+1}, \ldots, x_n]$. A lexicographic Gröbner basis splits into blocks, the last one of which depends only on x_n, the penultimate on x_n and x_{n-1}, and so on[9]:

```
sage: R.<t,x,y,z> = PolynomialRing(QQ, order='lex')
sage: J = R.ideal(t+x+y+z-1, t^2-x^2-y^2-z^2-1, t-x*y)
sage: [u.polynomial(u.variable(0)) for u in J.groebner_basis()]
[t + x + y + z - 1,
(y + 1)*x + y + z - 1,
(z - 2)*x + y*z - 2*y - 2*z + 1,
(z - 2)*y^2 + (-2*z + 1)*y - z^2 + z - 1]
```

In this example, the last polynomial of the basis only depends on y and z. It is preceded by a block of two polynomials in x, y and z, and the first polynomial contains all variables. The successive elimination ideals can be seen immediately.

We have seen however (§9.2.5) that the elimination ideals do not provide a perfect description of the ideal. Here, the block of polynomials in z only is empty, thus any value of z, except maybe a finite number, appears as last coordinate of a solution. We are tempted to express the possible values of y for each z thanks to the last equation. We get two values, except for $z = 2$, for which only $y = -1$ works. Only when we get to the preceding equation do we notice that the choice $z = 2$ is contradictory. Inversely, again from the last equation, $y = -1$ implies $z = 2$, and is thus excluded. It finally occurs that none of the leading terms of the polynomials (written in their respective main variable, as in the above Sage output) vanishes for $z \neq 2$.

Exercise 38 (Trigonometric relations)**.** Write $(\sin\theta)^6$ as a polynomial in $u(\theta) = \sin\theta + \cos\theta$ and $v(\theta) = \sin(2\theta) + \cos(2\theta)$.

9.3.5 Computations

We refer the reader interested in algorithms computing Gröbner bases to the reference [CLO07] mentioned in introduction. In addition, the module `sage.rings.polynomial.toy_buchberger` from Sage offers a "pedagogical" implementation of Buchberger's algorithm and various related algorithms, which closely follows their theoretical description.

Let us however keep in mind that computing a Gröbner basis is expensive both in terms of time and memory, and even very expensive in some unlucky cases. Besides, the `groebner_basis` method has several options[10] which allow

[8] For a given k, this is true more generally for any order such that $i \leq k < j \implies x_i > x_j$. Such an order is called a "block order" (see also Table 9.5).

[9] We thus get a "triangular form" of the system formed by the ideal generators, however in a weaker sense with respect to §9.2.5: we cannot say much *a priori* about the number of polynomials in each block or their leading terms.

[10] For more details, see the help page of this method, as well as those of internal methods of the ideal, whose name starts with `_groebner_basis`.

Change of order

The most interesting Gröbner bases are not the easiest to compute: often, the `degrevlex` order is the cheapest, but more useful information can be read on a lexicographic Gröbner basis. Besides, we sometimes need Gröbner bases of the same ideal for different monomial orders.

This motivates the introduction, in addition to general algorithms computing Gröbner bases, of algorithms of "change of order". These algorithms compute a Gröbner basis for a given monomial order from a Gröbner basis of the same ideal for a different order. They are often more efficient than algorithms computing directly a basis for the target order. Thus, a strategy which often wins to compute a lexicographic Gröbner basis is the following: first compute a basis for the `degrevlex` order, then apply an algorithm of change of order. Sage does it automatically in some cases.

The `transformed_basis` method allows us to compute "by hand" Gröbner bases by change of order, when the ideal is zero-dimensional, or when the target order is `lex`. If needed, it first computes a Gröbner basis for the monomial order attached to the polynomial ring.

the expert user to manually choose a Gröbner basis algorithm according to the characteristics of the problem to solve.

Let us consider the ideals $C_n(K) \subset K[x_0, \dots, x_{n-1}]$ defined by:

$$\begin{aligned} C_2(K) &= \langle x_0 + x_1, x_0x_1 - 1\rangle \\ C_3(K) &= \langle x_0 + x_1 + x_2, x_0x_1 + x_0x_2 + x_1x_2, x_0x_1x_2 - 1\rangle \\ &\vdots \\ C_n(K) &= \Big\langle \sum_{i\in\mathbb{Z}/n\mathbb{Z}} \prod_{j=0}^{k} x_{i+j} \Big\rangle_{k=0}^{n-2} + \langle x_0 \cdots x_{n-1} - 1\rangle, \end{aligned} \tag{9.11}$$

and accessible in Sage by commands like:

```
sage: from sage.rings.ideal import Cyclic
sage: Cyclic(QQ['x,y,z'])
Ideal (x + y + z, x*y + x*z + y*z, x*y*z - 1) of
Multivariate Polynomial Ring in x, y, z over Rational Field
```

They are classical test problems to evaluate the efficiency of tools for solving polynomial systems. On a computer where Sage computes the reduced Gröbner basis of $C_6(\mathbb{Q})$ in less than a second:

```
sage: def C(R, n): return Cyclic(PolynomialRing(R, 'x', n))
```

```
sage: %time len(C(QQ, 6).groebner_basis())
CPU times: user 136 ms, sys: 0 ns, total: 136 ms
Wall time: 147 ms
45
```

the computation of that of $C_7(\mathbb{Q})$ does not terminate after a dozen of hours, and uses more than 3 Gb of memory.

Failing to compute the Gröbner basis over the rational numbers, let us try to replace $\mathbb{Q}$ by a finite field $\mathbb{F}_p$. The idea, classical in computer algebra, is to limit the cost of operations on coefficients: those on elements of $\mathbb{F}_p$ take a constant cost, whereas the number of digits of rational numbers tends to increase quite rapidly during computations. We choose p small enough such that computations in $\mathbb{F}_p$ can be done directly with machine integers. It must not however be too small, so that the Gröbner basis on $\mathbb{F}_p$ can share a large part of the structure of that on $\mathbb{Q}$.

For example, with a convenient p, the Gröbner basis of $C_6(\mathbb{F}_p)$ has the same number of elements as that of $C_6(\mathbb{Q})$:

```
sage: p = previous_prime(2^30)
sage: len(C(GF(p), 6).groebner_basis())
45
```

By increasing the size of the system to solve, we see that the influence of the coefficient field on the computing time is far from negligible: the cases $n = 7$ and $n = 8$ become easy to solve.

```
sage: %time len(C(GF(p), 7).groebner_basis())
CPU times: user 1.44 s, sys: 4 ms, total: 1.44 s
Wall time: 1.46 s
209
sage: %time len(C(GF(p), 8).groebner_basis())
CPU times: user 40.7 s, sys: 24 ms, total: 40.7 s
Wall time: 40.9 s
372
```

These examples illustrate also another important phenomenon: the output of a Gröbner basis computation might be much larger than the input. For example, the last computation above shows that any Gröbner basis, reduced or not, of $C_8(\mathbb{F}_p)$ (with this value of p) for the `degrevlex` order counts at least 372 elements, whereas C_8 is generated by only 8 polynomials.

In order to solve this differential equation you look at it till a solution occurs to you.

George Pólya (1887 - 1985)

10 Differential Equations and Recurrences

10.1 Differential Equations

10.1.1 Introduction

If George Pólya's method does not seem very effective, one can appeal to Sage even if the domain of formal resolution of differential equations remains a weakness of many symbolic computation systems. Sage is evolving however by expanding its spectrum of resolution.

One can, if desired, invoke Sage in order to obtain a qualitative study: indeed, its numerical and graphical tools will guide the intuition. This is the subject of Section 14.2 from the chapter on numerical integration. Tools for the graphical study of the solutions are given in Section 4.1.6. Solving methods using series can be found in Section 7.5.2.

One may prefer to solve differential equations exactly. Sage can sometimes help by directly giving a formal answer as we will see in this chapter.

In most cases, it will be necessary to go through a tricky manipulation of these equations to help Sage. It should be kept in mind that the expected solution of a differential equation is a *function* differentiable over a certain interval, but that Sage manipulates *expressions* without a definition domain. The machine will therefore require human intervention to move towards a rigorous solution.

We shall first study generalities on ordinary differential equations of order 1 and some special cases such as linear equations, separable equations, homogeneous equations, a parameter dependent equation (§10.1.2); then more briefly the equations of order 2 and an example of a partial differential equation (§10.1.3). We will end with the use of the Laplace transform (§10.1.4) and finally the resolution of some differential systems (§10.1.5).

An *ordinary differential equation* (ODE) is an equation involving an (unknown) function of a single variable, as well as one or more derivatives, successive or not, of the function.

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* (referred to as PDE) involves several independent variables as well as the partial derivatives of the dependent variable with respect to these independent variables.

Unless otherwise stated, we shall consider in this chapter functions of a real variable.

10.1.2 First-Order Ordinary Differential Equations

Basic Commands. We would like to solve a first-order ODE:

$$F(x, y(x), y'(x)) = 0.$$

We start by defining a variable x and a function y depending on this variable:

```
sage: x = var('x')
sage: y = function('y')(x)
```

Then:

```
sage: desolve(equation, variable, ics = ..., ivar = ...,
....:         show_method = ..., contrib_ode = ...)
```

where:

- `equation` is the differential equation. Equality is designated by `==`, for instance, the equation $y' = 2y + x$ is written `diff(y,x) == 2*y+x`;
- `variable` is the dependent variable, i.e., y in $y' = 2y + x$;
- `ics` is optional and stands for initial conditions. For a first-order equation, write $[x_0,y_0]$ and for a second-order equation write $[x_0,y_0,x_1,y_1]$ or $[x_0,y_0,y'_0]$;
- `ivar` is optional and stands for the independent variable, i.e., x in $y' = 2y+x$. It must be specified if there is more than one independent variable or parameters as in $y' = ay + bx + c$;
- `show_method` is an optional boolean set to false. If true, then Sage returns a pair "[solution, method]", where method is the string describing the method which has been used to get a solution. The method can be one of the following: `linear`, `separable`, `exact`, `homogeneous`, `bernoulli`, `generalized homogeneous`.
- `contrib_ode` is an optional boolean set to false. If true, `desolve` allows to solve Clairaut, Lagrange, Riccati and some other equations. This may take a long time and is thus turned off by default.

First-Order Equations Directly Solved by Sage. We will study in this section how to solve with Sage linear, separable, Bernoulli, Riccati, Lagrange, Clairaut, homogeneous and exact equations.

Linear equations. These are equations of the form

$$y' + P(x)y = Q(x),$$

where P and Q are continuous functions on given intervals.

Example: $y' + 3y = \mathrm{e}^x$.

```
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']
```

Separable equations. These are equations of the form

$$P(x) = y'Q(y),$$

where P and Q are continuous functions on given intervals.

Example: $yy' = x$.

```
sage: desolve(y*diff(y,x) == x, y, show_method=True)
[1/2*y(x)^2 == 1/2*x^2 + _C, 'separable']
```

Caution! Sometimes Sage solves separable equations as exact. Example: $y' = \mathrm{e}^{x+y}$.

```
sage: desolve(diff(y,x) == exp(x+y), y, show_method=True)
[-(e^(x + y(x)) + 1)*e^(-y(x)) == _C, 'exact']
```

Bernoulli equations. These are equations of the form

$$y' + P(x)y = Q(x)y^\alpha,$$

where P and Q are continuous functions on given intervals and $\alpha \notin \{0, 1\}$. Example: $y' - y = xy^4$.

```
sage: desolve(diff(y,x)-y == x*y^4, y, show_method=True)
[e^x/(-1/3*(3*x - 1)*e^(3*x) + _C)^(1/3), 'bernoulli']
```

Homogeneous equations. These are equations of the form

$$y' = \frac{P(x,y)}{Q(x,y)},$$

where P and Q are homogeneous functions of same degree on given intervals.

Example: $x^2y' = y^2 + xy + x^2$.

```
sage: desolve(x^2*diff(y,x) == y^2+x*y+x^2, y, show_method=True)
```

```
[_C*x == e^(arctan(y(x)/x)), 'homogeneous']
```

Solutions are not given explicitly. We will see further on how to deal with these equations in some situations.

EXACT EQUATIONS. These are equations of the form

$$\frac{\partial f}{\partial x}\mathrm{d}x + \frac{\partial f}{\partial y}\mathrm{d}y,$$

where f is a differentiable function of two variables.

Example: $y' = \frac{\cos(y)-2x}{y+x\sin(y)}$ with $f = x^2 - x\cos y + y^2/2$.

```
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']
```

Once again, solutions are not given explicitly.

RICCATI EQUATIONS. These are equations of the form

$$y' = P(x)y^2 + Q(x)y + R(x),$$

where P, Q and R are continuous functions on given intervals.

Example: $y' = xy^2 + \frac{1}{x}y - \frac{1}{x^2}$.

In this case, we set `contrib_ode` to `True` to make Sage use more complex methods.

```
sage: desolve(diff(y,x) == x*y^2+y/x-1/x^2, y,
....:         contrib_ode=True, show_method=True)[1]
'riccati'
```

LAGRANGE AND CLAIRAUT EQUATIONS. When the equation is of the form $y = xP(y') + Q(y')$ where P and Q are $\mathcal{C}^1$ on a given interval, it is a Lagrange equation. When P is the identity function, it is a Clairaut equation. Example: $y = xy' - y'^2$.

```
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']
```

Linear Equations. Let us solve $y' + 2y = x^2 - 2x + 3$:

```
sage: x = var('x'); y = function('y')(x)
```

```
sage: DE = diff(y,x)+2*y == x**2-2*x+3
sage: desolve(DE, y)
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)
```

We can rearrange the output with `expand`:

```
sage: desolve(DE, y).expand()
1/2*x^2 + _C*e^(-2*x) - 3/2*x + 9/4
```

It is thus convenient to use the form `desolve(...).expand()`. Let us check which method has been used:

```
sage: desolve(DE, y, show_method=True)[1]
'linear'
```

Let us add an initial condition, for instance $y(0) = 1$:

```
sage: desolve(DE, y, ics=[0,1]).expand()
1/2*x^2 - 3/2*x - 5/4*e^(-2*x) + 9/4
```

Separable Equations. Let us solve $y' \log(y) = y \sin(x)$:

```
sage: x = var('x'); y = function('y')(x)
sage: desolve(diff(y,x)*log(y) == y*sin(x), y, show_method=True)
[1/2*log(y(x))^2 == _C - cos(x), 'separable']
```

Sage agrees with us: it is a separable equation.

We should assign the solutions in order to use them later on:

```
sage: ed = desolve(diff(y,x)*log(y) == y*sin(x), y); ed
1/2*log(y(x))^2 == _C - cos(x)
```

Here, $y(x)$ is not explicitly given: $\frac{1}{2}\log^2\big(y(x)\big) = \_C - \cos(x)$.

We can get $y(x)$ explicitly using `solve`. Be aware that `ed` is an equation where `y` is a variable:

```
sage: solve(ed, y)
[y(x) == e^(-sqrt(2*_C - 2*cos(x))), y(x) == e^(sqrt(2*_C - 2*cos(x)))]
```

We should take care that `sqrt(2*_C - 2*cos(x))` may cause some problems even if Sage does not warn us. We will then assume that $\_C \geqslant 1$.

To draw the graph of solutions, we need their right-hand side. For instance, in order to get the first solution's right-hand side with $\_C = 5$, we could type:

```
sage: solve(ed, y)[0].substitute(_C==5).rhs()
Traceback (most recent call last):
...
NameError: name '_C' is not defined
```

$\_C$ has not been defined but only introduced by Sage. We can get it through the `variables()` command which gives the variables list:

```
sage: ed.variables()
(_C, x)
```

Only $\_C$ and x are variables, y having been defined as function of the variable x.

```
sage: c = ed.variables()[0]
sage: solve(ed, y)[0].substitute(c == 5).rhs()
e^(-sqrt(-2*cos(x) + 10))
```

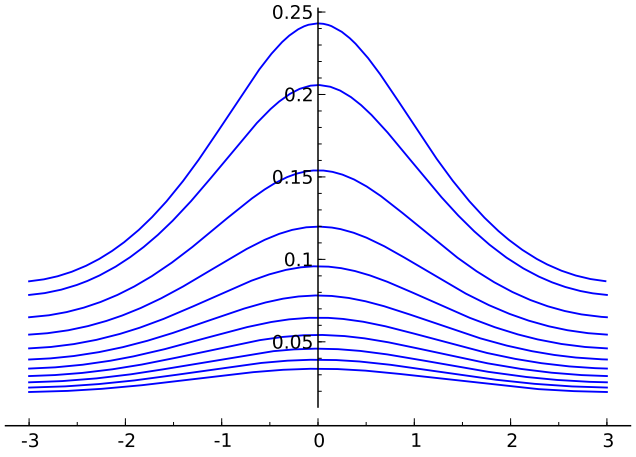
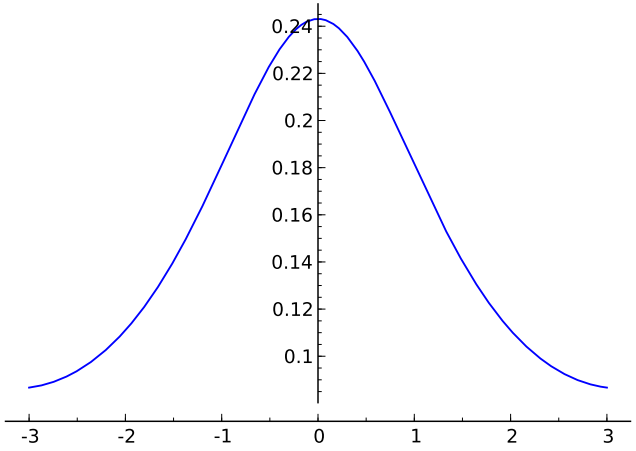


FIGURE 10.1 – Some solutions of $y' \log(y) = y \sin(x)$.

Another example with $\_C = 2$:

```
sage: plot(solve(ed, y)[0].substitute(c == 2).rhs(), x, -3, 3)
```

which gives:



To get several curves, (see Figure 10.1), we use a loop:

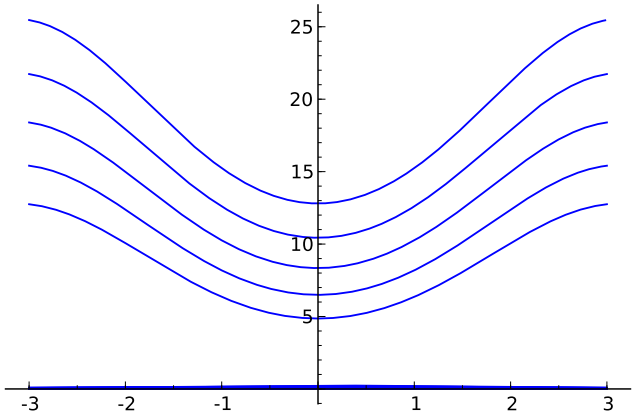
```
sage: P = Graphics()
sage: for k in range(1,20,2):
....:     P += plot(solve(ed, y)[0].substitute(c==1+k/4).rhs(), x, -3, 3)
```

We could have used a double loop in order to get the two solutions:

```
sage: P = Graphics()
sage: for j in [0,1]:
....:     for k in range(1,10,2):
```

```
....:         f = solve(ed,y)[j].substitute(c==2+0.25*k).rhs()
....:         P += plot(f, x, -3, 3)
sage: P
```

but the scales are too different to see both sets of curves:



Exercise 39 (Separable equations)**.** Find the solutions in $\mathbb{R}$ of these separable equations:

1. (E_1) : $\frac{yy'}{\sqrt{1+y^2}} = \sin(x)$;
2. (E_2) : $y' = \frac{\sin(x)}{\cos(y)}$.

Homogeneous Equations. We want to solve the differential equation $xy' = y + \sqrt{y^2 + x^2}$ which is homogeneous since

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{y + \sqrt{y^2 + x^2}}{x} = \frac{N(y,x)}{M(y,x)},$$

with $N(ky, kx) = kN(y, x)$ and $M(ky, kx) = kM(y, x)$.

We just need to introduce the change of variables $y(x) = x \cdot u(x)$ for all real x in order to get a separable equation.

```
sage: u = function('u')(x)
sage: y = x*u
sage: DE = x*diff(y,x) == y + sqrt(x**2 + y**2)
```

Let us change variables in the initial differential equation. The equation being undefined at 0, we solve it first on $]0, +\infty[$ and then on $]-\infty, 0[$.

```
sage: assume(x>0)
sage: desolve(DE, u)
x == _C*e^arcsinh(u(x))
```

We do not get u explicitly. We therefore use Maxima's `ev` command (as in *ev*aluate) with `logarc=True` in order to use the inverse hyperbolic functions as logarithms; u will then be expressed thanks to the `solve` command:

```
sage: S = desolve(DE,u)._maxima_().ev(logarc=True).sage().solve(u); S
[u(x) == -(sqrt(u(x)^2 + 1)*_C - x)/_C]
```

Here, `S` is a list containing a single equation; `S[0]` is therefore the equation itself.

Here we can observe that the equation is still implicitly solved, thus we will ask Sage to solve the equivalent equation:

$$c^2(u^2+1) = (x-uc)^2,$$

via

```
sage: solu = (x-S[0]*c)^2; solu
(_C*u(x) - x)^2 == (u(x)^2 + 1)*_C^2
sage: sol = solu.solve(u); sol
[u(x) == -1/2*(_C^2 - x^2)/(_C*x)]
```

We then just need to go back to y:

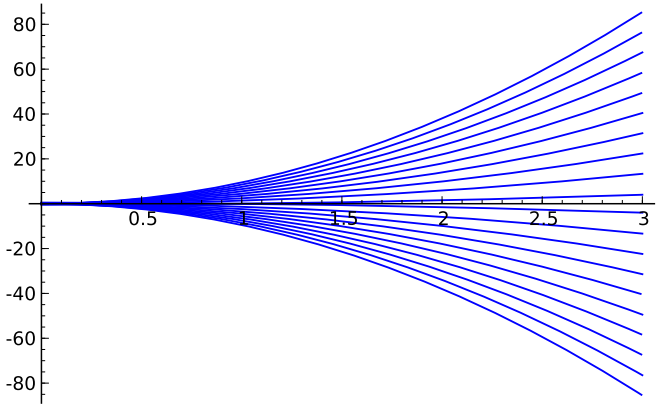
```
sage: y(x) = x*sol[0].rhs(); y(x)
-1/2*(_C^2 - x^2)/_C
```

And here are the explicit solutions!

$$y(x) = \frac{x^2-c^2}{2c}.$$

We then draw the solutions on $]0,+\infty[$, keeping in mind that $\_C$ must be a non-zero constant.

```
sage: c = y(x).variables()[0]
sage: P = Graphics()
sage: for k in range(-19,19,2):
....:     P += plot(y(x).substitute(c == 1/k), x, 0, 3)
sage: P
```



Exercise 40 (Homogeneous differential equations)**.** Solve the following homogeneous equation over $\mathbb{R}$: (E_5) : $xyy' = x^2+y^2$.

A Parametric Equation: the Verhulst Equation. The relative rate of growth of a population is a linearly decreasing function of the population. In order to study this, one can attempt to solve an equation of the form:

$$y' = ay - by^2,$$

with a and b being positive real parameters.

```
sage: x = var('x'); y = function('y')(x); a, b = var('a, b')
sage: DE = diff(y,x) - a*y == -b*y**2
sage: sol = desolve(DE,[y,x]); sol
-(log(b*y(x) - a) - log(y(x)))/a == _C + x
```

As usual, we do not get y explicitly. Let us try to isolate it with `solve`:

```
sage: Sol = solve(sol, y)[0]; Sol
log(y(x)) == _C*a + a*x + log(b*y(x) - a)
```

We still do not have an explicit solution. We group together the terms on the left-hand side and simplify this expression using `simplify_log()`:

```
sage: Sol(x) = Sol.lhs()-Sol.rhs(); Sol(x)
-_C*a - a*x - log(b*y(x) - a) + log(y(x))
sage: Sol = Sol.simplify_log(); Sol(x)
-_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)
```

10.1.3 Second-Order Equations

Linear Ordinary Differential Equations with Constant Coefficients. Let us solve now a second-order linear ordinary differential equation with constant coefficients, for instance:

$$y'' + 3y = x^2 - 7x + 31.$$

Here we use the same syntax as for the first-order equations, the second derivative of y with respect to x is obtained with `diff(y, x, 2)`.

```
sage: x = var('x'); y = function('y')(x)
sage: DE = diff(y,x,2)+3*y == x^2-7*x+31
sage: desolve(DE, y).expand()
1/3*x^2 + _K2*cos(sqrt(3)*x) + _K1*sin(sqrt(3)*x) - 7/3*x + 91/9
```

Let us add initial conditions, for instance $y(0) = 1$ and $y'(0) = 2$:

```
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 $y(0) = 1$ and $y(-1) = 0$:

```
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
```

that is

$$\frac{1}{3}x^2 - \frac{7}{3}x - \frac{82\,\sin(\sqrt{3}x)\cos(\sqrt{3})}{9\,\sin(\sqrt{3})} + \frac{115\,\sin(\sqrt{3}x)}{9\,\sin(\sqrt{3})} - \frac{82}{9}\,\cos(\sqrt{3}x) + \frac{91}{9}.$$

How to Solve a PDE: the Heat Equation. Next we will study the famous heat equation. The temperature z is distributed in a homogeneous rectilinear rod of length ℓ according to the equation (where x is the abscissa along the rod, and t the time):

$$\frac{\partial^2 z}{\partial x^2}(x,t) = C\frac{\partial z}{\partial t}(x,t).$$

This equation will be studied against the following initial conditions:

$$\forall t \in \mathbb{R}^+, \quad z(0,t) = 0 \quad z(\ell,t) = 0 \quad \forall x \in]0;\ell[, \quad z(x,0) = 1.$$

We will seek non-zero solutions of the form:

$$z(x,t) = f(x)g(t).$$

This is the method of separation of variables.

```
sage: x, t = var('x, t'); f = function('f')(x); g = function('g')(t)
sage: z = f*g
sage: eq(x,t) = diff(z,x,2) == diff(z,t); eq(x,t)
g(t)*diff(f(x), x, x) == f(x)*diff(g(t), t)
```

The equation thus becomes:

$$g(t)\frac{\mathrm{d}^2 f(x)}{\mathrm{d}x^2} = f(x)\frac{\mathrm{d}g(t)}{\mathrm{d}t}.$$

Let us divide by $f(x)g(t)$, assumed not to be zero:

```
sage: eqn = eq/z; eqn(x,t)
diff(f(x), x, x)/f(x) == diff(g(t), t)/g(t)
```

We then obtain an equation where each side depends only on one variable:

$$\frac{1}{f(x)}\frac{\mathrm{d}^2 f(x)}{\mathrm{d}x^2} = \frac{1}{g(t)}\frac{\mathrm{d}g(t)}{\mathrm{d}t}.$$

Each side can therefore only be constant. Let us separate equations and introduce a constant k:

```
sage: k = var('k')
sage: eq1(x,t) = eqn(x,t).lhs() == k; eq2(x,t) = eqn(x,t).rhs() == k
```

We solve the equations separately, beginning with the second one:

```
sage: g(t) = desolve(eq2(x,t),[g,t]); g(t)
```

```
_C*e^(k*t)
```

therefore $g(t) = ce^{kt}$ with c a constant. For the first one, we cannot do it directly:

```
sage: desolve(eq1,[f,x])
Traceback (most recent call last):
  ...
TypeError: ECL says: Maxima asks:
Is  k  positive, negative, or zero?
```

Let us use the `assume` mechanism:

```
sage: assume(k>0); desolve(eq1,[f,x])
_K1*e^(sqrt(k)*x) + _K2*e^(-sqrt(k)*x)
```

that is $f(x) = k_1 e^{x\sqrt{k}} + k_2 e^{-x\sqrt{k}}$.

10.1.4 The Laplace Transform

The Laplace transform converts a differential equation with initial conditions into an algebraic equation and the inverse transform then makes it possible to get back to the solution of the differential equation.

If f is a function defined on $\mathbb{R}$ and is identically zero on $]-\infty, 0[$, we call Laplace transform of f the function F defined, under certain conditions, by:

$$\mathcal{L}\big(f(x)\big) = F(s) = \int_0^{+\infty} e^{-sx} f(x)\, dx.$$

Laplace transforms are easily obtained from polynomial, trigonometric, exponential functions, and so on. These transforms have very interesting properties, especially concerning the transform of a derivative: if f' is a piecewise continuous function on $\mathbb{R}_+$ then

$$\mathcal{L}\big(f'(x)\big) = s\mathcal{L}\big(f(x)\big) - f(0),$$

and if f' satisfies the conditions imposed on f:

$$\mathcal{L}\big(f''(x)\big) = s^2\mathcal{L}\big(f(x)\big) - sf(0) - f'(0).$$

Example. We want to solve the differential equation $y'' - 3y' - 4y = \sin(x)$ using the Laplace transform with the initial conditions: $y(0) = 1$ and $y'(0) = -1$. Thus:

$$\mathcal{L}\left(y'' - 3y' - 4y\right) = \mathcal{L}\left(\sin(x)\right),$$

that is:

$$(s^2 - 3s - 4)\mathcal{L}(y) - sy(0) - y'(0) + 3y(0) = \mathcal{L}(\sin(x)).$$

In case we forgot the Laplace transforms of the most common functions, we can use Sage:

```
sage: x, s = var('x, s'); f = function('f')(x)
sage: f(x) = sin(x); f.laplace(x,s)
```

```
x |--> 1/(s^2 + 1)
```

Thus we get an expression of the Laplace transform of y:

$$\mathcal{L}(y) = \frac{1}{(s^2-3s-4)(s^2+1)} + \frac{s-4}{s^2-3s-4}.$$

Let us use Sage to get the inverse transform:

```
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)
```

If one wants to "cheat", one can decompose $X(s)$ into partial fractions first:

```
sage: X(s).partial_fraction()
1/34*(3*s - 5)/(s^2 + 1) + 9/10/(s + 1) + 1/85/(s - 4)
```

And all that remains is to read an inversion table. We can however use the black box `desolve_laplace` which will give the solution directly:

```
sage: x = var('x'); y = function('y')(x)
sage: eq = diff(y,x,x) - 3*diff(y,x) - 4*y - sin(x) == 0
sage: desolve_laplace(eq, y)
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(eq, y, ics=[0,1,-1])
3/34*cos(x) + 1/85*e^(4*x) + 9/10*e^(-x) - 5/34*sin(x)
```

10.1.5 Systems of Linear Differential Equations

A Simple Example of System of First-Order Linear Differential Equations. We want to solve the following system of linear differential equations

$$\begin{cases} y'(x) = A \cdot y(x) \\ y(0) = c \end{cases}$$

knowing that

$$A = \begin{bmatrix} 2 & -2 & 0 \\ -2 & 0 & 2 \\ 0 & 2 & 2 \end{bmatrix}, \qquad y(x) = \begin{bmatrix} y_1(x) \\ y_2(x) \\ y_3(x) \end{bmatrix}, \qquad c = \begin{bmatrix} 2 \\ 1 \\ -2 \end{bmatrix}.$$

We write:

```
sage: x = var('x'); y1 = function('y1')(x)
sage: y2 = function('y2')(x); y3 = function('y3')(x)
sage: y = vector([y1, y2, y3])
sage: A = matrix([[2,-2,0],[-2,0,2],[0,2,2]])
sage: system = [diff(y[i], x) - (A * y)[i] for i in range(3)]
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)]
```

Here the syntax for the initial conditions is: `ics = [x0,y1(x0),y2(x0),y3(x0)]`.

A Matrix with Complex Eigenvalues. Let us consider now

$$A = \begin{bmatrix} 3 & -4 \\ 1 & 3 \end{bmatrix}, \qquad c = \begin{bmatrix} 2 \\ 0 \end{bmatrix}.$$

With Sage:

```
sage: x = var('x'); y1 = function('y1')(x); y2 = function('y2')(x)
sage: y = vector([y1,y2])
sage: A = matrix([[3,-4],[1,3]])
sage: system = [diff(y[i], x) - (A * y)[i] for i in range(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)]
```

that is:

$$\begin{cases} y_1(x) = 2\cos(2x)\mathrm{e}^{3x} \\ y_2(x) = \sin(2x)\mathrm{e}^{3x}. \end{cases}$$

A Second-Order System. We want to solve the following system

$$\begin{cases} y_1''(x) - 2y_1(x) + 6y_2(x) - y_1'(x) - 3y_2'(x) = 0 \\ y_2''(x) + 2y_1(x) - 6y_2(x) - y_1'(x) + y_2'(x) = 0. \end{cases}$$

We reduce to a first-order system by setting

$$u = (u_1, u_2, u_3, u_4) = (y_1, y_2, y_1', y_2').$$

Thus we get:

$$\begin{cases} u_1' = u_3 \\ u_2' = u_4 \\ u_3' = 2u_1 - 6u_2 + u_3 + 3u_4 \\ u_4' = -2u_1 + 6u_2 + u_3 - u_4, \end{cases}$$

That is $u'(x) = A \cdot u(x)$ with

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 2 & -6 & 1 & 3 \\ -2 & 6 & 1 & -1 \end{bmatrix}.$$

With Sage:

```
sage: x = var('x'); u1 = function('u1')(x); u2 = function('u2')(x)
sage: 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 range(4)]
sage: sol = desolve_system(system, [u1, u2, u3, u4])
```

We will only consider the first two coordinates because we need y_1 and y_2, that is u_1 and u_2:

```
sage: sol[0]
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]
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)
```

which can be summarised more concisely as:

$$\begin{cases} y_1(x) = k_1 \mathrm{e}^{2x} + k_2 \mathrm{e}^{-4x} + 3k_3 \\ y_2(x) = k_4 \mathrm{e}^{2x} - k_2 \mathrm{e}^{-4x} + k_3 \end{cases}$$

with k_1, k_2, k_3 and k_4 parameters depending on the initial conditions.

10.2 Recurrence Relations

10.2.1 Recurrences $u_{n+1} = f(u_n)$

Definition. Let $u_{n+1} = f(u_n)$ be a recurrence relation, with $u_0 = a$. We can define the relation naturally by means of a recursive algorithm. Let us consider for instance the logistic map (defined by $x_{n+1} = r x_n (1 - x_n)$):

$$f : x \mapsto 3.83 \cdot x \left(1 - \frac{x}{100\,000}\right) \quad \text{and} \quad u_0 = 20\,000.$$

With Sage:

```
sage: x = var('x'); f = function('f')(x)
sage: f(x) = 3.83*x*(1 - x/100000)
sage: def u(n):
....:     if n==0: return(20000)
....:     else: return f(u(n-1))
```

An iterative definition may be preferred:

```
sage: def v(n):
....:     V = 20000;
....:     for k in [1..n]:
....:         V = f(V)
....:     return V
```

Differential equations	
Variable declaration	`x=var('x')`
Function declaration	`y=function('y')(x)`
Solving an equation	`desolve(equation, y, <options>)`
Solving a system	`desolve_system([eq1, ...], [y1, ...], <options>)`
First-order initial conditions	$[x_0, y(x_0)]$
Second-order initial conditions	$[x_0, y(x_0), x_1, y(x_1)]$
	$[x_0, y(x_0), y'(x_0)]$
System initial conditions	$[x_0, y_1(x_0), y_2(x_0), ...]$
Independent variable	`ivar=x`
Resolution method	`show_method=True`
Call for special methods	`contrib_ode=True`

Laplace transform	
Laplace transform of $f : x \mapsto f(x)$	`f.laplace(x,s)`
Inverse transform of $X(s)$	`X(s).inverse_laplace(s,x)`
Solving an ODE with the Laplace transform	`desolve_laplace(equation,function)`

Miscellaneous commands	
First-order derivative	`diff(y,x)`
Expanding an expression	`expr.expand()`
Getting the variables	`expr.variables()`
Variable substitution	`expr.substitute(var==val)`

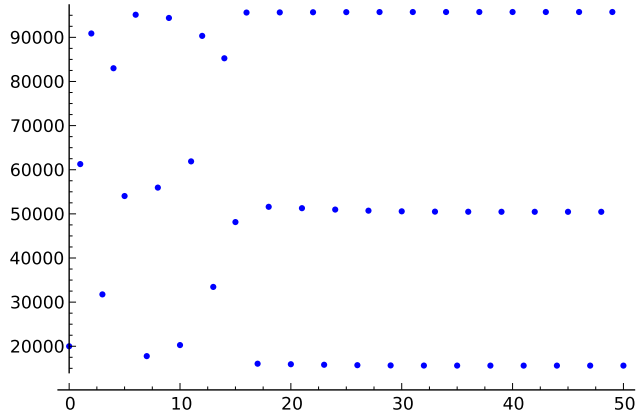
TABLE 10.1 – Useful commands for solving differential equations.

Graphical Representation. Let us plot the coordinates of (k, u_k):

```
sage: def cloud(u,n):
....:     L = [[0,u(0)]];
....:     for k in [1..n]:
....:         L += [[k,u(k)]]
....:     points(L).show()
```

From the following graph, we can assume the existence of three limit points:

```
sage: cloud(u,50)
```



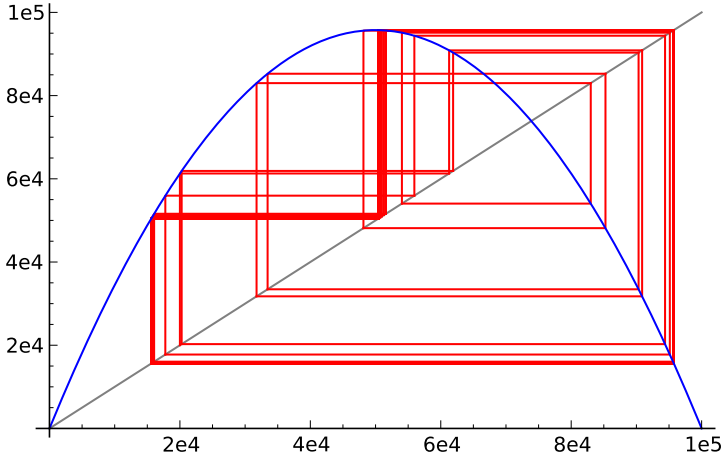
The representation involving the first bisector and the representative curve of f could have been preferred. Since this does not exist natively in Sage, we will build a small procedure that will do the job:

```
sage: def snail(f,x,u0,n,xmin,xmax):
....:     u = u0
....:     P = plot(x, x, xmin, xmax, color='gray')
....:     for i in range(n):
....:         P += line([[u,u],[u,f(u)],[f(u),f(u)]], color = 'red')
....:         u = f(u)
....:     P += f.plot(x, xmin, xmax, color='blue')
....:     P.show()
```

For instance, with the same relation:

```
sage: f(x) = 3.83*x*(1 - x/100000)
sage: snail(f,x,20000,100,0,100000)
```

The three limit points are highlighted:



10.2.2 Linear Recurrences with Rational Coefficients

Sage deals with relations of the following kind:

$$a_k u_{n+k} + a_{k-1} u_{n+k-1} + \cdots + a_1 u_{n+1} + a_0 u_n = 0,$$

the $(a_i)_{0 \leqslant i \leqslant k}$ being an indexed family of rational scalars.

For instance, consider the following relation:

$$u_0 = -1,\ u_1 = 1,\ u_{n+2} = \frac{3}{2} u_{n+1} - \frac{1}{2} u_n.$$

The well known function `rsolve` is not directly accessible. It must be picked up in SymPy, which causes some inconvenience such as syntax changes to declare variables. Here, for example, a preamble is necessary:

```
sage: from sympy import Function
sage: from sympy.abc import n
sage: u = Function('u')
```

The recurrence relation must then be defined as: $a_k u_{n+k} + \cdots + a_0 u_n = 0$. Here $u_{n+2} - \frac{3}{2} u_{n+1} + \frac{1}{2} u_n = 0$:

```
sage: f = u(n+2)-(3/2)*u(n+1)+(1/2)*u(n)
```

Finally, we use `rsolve`, observing how the initial conditions are declared (`u(0):value`, `u(1):value`, etc.):

```
sage: from sympy import rsolve
sage: rsolve(f, u(n), {u(0):-1,u(1):1})
3 - 4*2**(-n)
```

that is $u_n = 3 - \frac{1}{2^{n-2}}$.

10.2.3 Non-Homogeneous Linear Recurrence Relations

Sage also deals with relations of the following kind:

$$a_k(n) u_{n+k} + a_{k-1}(n) u_{n+k-1} + \cdots + a_1(n) u_{n+1} + a_0(n) u_n = f(n),$$

the $(a_i)_{0 \leqslant i \leqslant k}$ being an indexed family of polynomial, rational or hypergeometric functions of n.

The command will depend on the nature of $f(n)$:

- `rsolve_poly` if f is polynomial;
- `rsolve_ratio` if f is rational;
- `rsolve_hyper` if f is hypergeometric.

The coefficients $a_i(n)$ are given as a list $[a_0(n), \ldots, a_{k-1}(n), a_k(n)]$. For example, in order to study the complexity of merge sort, one has to study the following relation:

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

The computation yields:

```
sage: from sympy import rsolve_hyper
sage: from sympy.abc import n
sage: rsolve_hyper([-2,1],2**(n+2),n)
2**n*C0 + 2**(n + 2)*(C0 + n/2)
```

and since $u_0 = 0$ gives `C0=0`, we obtain $u_n = n \cdot 2^{n+1}$.

Part III

Numerical Computation

11 Floating-Point Numbers

In the next chapters, floating-point numbers are at the heart of all computations. It is necessary to study them, as their behaviour follows precise rules.

How can we represent real numbers in a computer? In general, these numbers cannot be coded with a finite amount of information, and thus they cannot be exactly represented. It is necessary to approximate them using a finite amount of memory.

A standard has appeared around an approximation of real numbers with a finite quantity of information: the floating-point representation.

In this chapter, one will find: a basic description of the floating-point numbers and of the different kinds of these numbers available in Sage, and a demonstration of some of their properties. Examples will show some difficulties we encounter when computing with floating-point numbers and some tricks to get around them. We hope that the reader will develop a necessary careful approach. To conclude, we will try to describe some properties which must be fulfilled by numerical methods when they use these numbers.

To go further, the reader should refer to [BZ10] and [Gol91] (available on the internet) or to the book [MBdD+10].

11.1 Introduction

11.1.1 Definition

A set $F(\beta, r, m, M)$ of floating-point numbers is defined by four parameters: a *radix* $\beta \geq 2$, a number r of digits and two signed integers m and M. The elements of $F(\beta, r, m, M)$ are numbers of the form

$$x = (-1)^s \; 0.d_1 d_2 \dots d_r \cdot \beta^j,$$

where the digits d_i are integers verifying $0 \leq d_i < \beta$ for $i > 1$ and $0 < d_1 < \beta$. The amount r of digits is the *precision*: the sign s is 0 or 1; the *exponent* j lies in the range $[m, M]$, and $0.d_1 d_2 \ldots d_r$ is the *significand*.

11.1.2 Properties and Examples

The normalisation $0 < d_1 < \beta$ ensures that all floating-point numbers have the same amount of significant digits. One remarks that, with the convention $d_1 > 0$, the "zero" value cannot be represented: zero has a special representation.

As example, the number denoted by -0.028 in radix 10 (fixed-point representation) will be represented by $-0.28 \cdot 10^{-1}$ (assuming $r \geq 2$ and $m \leq -1 \leq M$). As the radix 2 is well adapted to the binary representation of the computers, we will always have $\beta = 2$ in the different sets of floating-point numbers proposed by Sage, and we will always use this setting in the remainder of this chapter. To give an example, $0.101 \cdot 2^1$ represents the value $5/4$ in the set $F(2, 3, -1, 2)$.

As the only possible value for d_1 when $\beta = 2$ is $d_1 = 1$, d_1 can be omitted in the machine implementation; considering again the set $F(2, 3, -1, 2)$, $5/4$ can be represented in the computer by the 5 bits: 00110, where the leftmost bit represents the + sign, the 2 following bits (01) represent the significand (101), and the last 2 ones at the right represent the exponent (00 encoding the value -1 of the exponent, 01 encoding 0, and so on).

It should be obvious for the reader that the sets $F(\beta, r, m, M)$ only describe a subset of the real numbers. To represent a real number x located between two consecutive numbers in $F(\beta, r, m, M)$, we need a function called *rounding* which will define which number will approximate x: for this we can use the nearest number from x, but other choices are available. The standard imposes that $F(\beta, r, m, M)$ is invariant by the rounding application. The set of numbers which can be represented is bounded, and the floating-point numbers contain the special values $+\infty$, $-\infty$ which represent the infinities (as $1/0$) but also all values greater than the largest positive number which can be represented (or less than the smallest negative number available), and also a representation of indefinite operations like $0/0$.

11.1.3 Standardisation

After some years of trials and errors, the need for a standard did arise, so that identical programs give the same results on different machines. Since 1985, the IEEE-754 standard defines different sets of numbers; among them the 64-bit "double-precision" numbers: the sign s is encoded on 1 bit, the significand on 53 bits (from which only 52 are stored), and the exponent on 11 bits. Numbers are of the form

$$(-1)^s \; 0.d_1 d_2 \ldots d_{53} \cdot 2^{j-1023}.$$

They correspond to the "double" type of the C programming language.

11.2 The Floating-Point Numbers

Sage provides two sorts of floating-point numbers:

1. the "double-precision" numbers as described in §11.1.3: these numbers are provided by the computer's processor; in Sage, they belong to the class `RDF`:

```
sage: xrdf = RDF(3.0)
```

2. floating-point numbers with an arbitrary precision: every instance of the class `RealField` — or `Reals` — defines a set of floating-point numbers with a given precision (and possibly with a given rounding mode: see §11.3.2). For example, to declare a number `x100` with a precision on 100 binary digits, one writes:

```
sage: R100 = RealField(100) # precision: 100 bits.
sage: x100 = R100(3/8); x100
0.37500000000000000000000000000
```

In the set `RealField(p)` numbers are of the form

$$(-1)^s \ 0.d_1 d_2 \dots d_p \cdot 2^e,$$

with $s \in \{0, 1\}$; the significand has p binary digits and e might have 30 binary digits (or more on some computers). An implicit precision is available:

```
sage: Rdefault = RealField()  # default precision of 53 bits
sage: xdefault = Rdefault(2/3)
```

and it is possible to check the precision of all floating-point numbers using the `prec()` method:

```
sage: xrdf.prec()
53
sage: x100.prec()
100
sage: xdefault.prec()
53
```

So, the numbers of the set `RealField()` and those of the set `RDF` have the same precision, but `RealField()` allows much larger exponents. The set `RealField()`, with a precision of 53 bits, is the default type of "real" numbers in Sage:

```
sage: x = 1.0; type(x)
<type 'sage.rings.real_mpfr.RealLiteral'>
sage: x.prec()
53
```

Here, `real_mpfr.RealLiteral` means that the set of numbers to which `x` belongs is implemented by the GNU MPFR library. Let us recall that the type of a variable is automatically defined by the right-hand side in an assignment:

```
sage: x = 1.0          # x belongs to RealField()
sage: x = 0.1e+1       # idem: x belongs to RealField()
sage: x = 1            # x is an integer
sage: x = RDF(1)       # x is a machine double-precision number
sage: x = RDF(1.)      # idem: x is a machine double-precision number
sage: x = RDF(0.1e+1) # idem
sage: x = 4/3          # x is a rational number
sage: R = RealField(20)
sage: x = R(1)         # x is a 20-bit floating-point number
```

and natural conversions from rational numbers are carried out:

```
sage: RDF(8/3)
2.6666666666666665
sage: R100 = RealField(100); R100(8/3)
2.6666666666666666666666666667
```

like conversions between different sets of floating-point numbers:

```
sage: x = R100(8/3)
sage: R = RealField(); R(x)
2.66666666666667
sage: RDF(x)
2.6666666666666665
```

The different sets of floating-point numbers contain the special values `+0`, `-0`, `+infinity`, `-infinity`, and[1] `NaN`:

```
sage: 1.0/0.0
+infinity
sage: RDF(1)/RDF(0)
+infinity
sage: RDF(-1.0)/RDF(0.)
-infinity
```

The special value `NaN` stands for undefined results:

```
sage: 0.0/0.0
NaN
sage: RDF(0.0)/RDF(0.0)
NaN
```

11.2.1 Which Kind of Floating-Point Numbers to Choose?

The arbitrary precision floating-point numbers allow us to compute with a very large precision, whereas the precision is fixed for `RDF` numbers. Computations with the `RealField(n)` numbers use the GNU MPFR software library, while for `RDF` numbers computations are carried out using the floating-point arithmetic of the processor, which is much faster. In §13.2.10 we give a comparison where the

[1] These results follow the IEEE-754 standard.

x	R2(x).ulp()	RDF(x).ulp()	R100(x).ulp()
10^{-30}	3.9e-31	1.75162308041e-46	1.24460305557222834142881281 08e-60
10^{-10}	2.9e-11	1.29246970711e-26	9.18354961579912115600575419 70e-41
10^{-3}	0.00049	2.16840434497e-19	1.54074395550978868244478235 41e-33
1	0.50	2.22044604925e-16	1.57772181044202361082345713 06e-30
10^{3}	510.	1.13686837722e-13	8.07793566946316088741610050 85e-28
10^{10}	4.3e9	1.90734863281e-06	1.35525271560688054250931600 11e-20
10^{30}	3.2e29	1.40737488355e+14	1.00000000000000000000000000

TABLE 11.1 – Distance between floating-point numbers.

efficiency of the processor's floating-point arithmetic is combined with libraries optimised for these numbers. Note that, among the numerical methods we will encounter in the next chapters, most of them only use `RDF` numbers and, whatever we do, a conversion of floating-point numbers to this set will occur.

R2, a Toy Set of Floating-Point Numbers. Arbitrary precision floating-point numbers, apart from being mandatory for large precision computations, enable us to define a class of floating-point numbers which, as they have very low accuracy, demonstrate in an extremal way the properties of floating-point numbers; the set `R2` of numbers with a precision of 2 bits:

```
sage: R2 = RealField(2)
```

11.3 Some Properties of Floating-Point Numbers

11.3.1 These Sets are Full of Gaps

In every set of floating-point numbers, the `ulp()` method (*unit in the last place*) returns the distance from a representable number to the next representable one (in the opposite direction from zero):

```
sage: x2 = R2(1.); x2.ulp()
0.50
sage: xr = 1.; xr.ulp()
2.22044604925031e-16
```

The reader can easily check the value given by `x2.ulp()`.

Table 11.1 gives the size of the interval which separates a given number x — or more exactly $R(x)$ where R is the considered set — from its nearest neighbour (in the opposite direction from zero) for different sets of numbers (`R100` is the set `RealField(100)`), and different values of x.

As expected, the size of the *gaps* between two consecutive numbers grows with the magnitude of the numbers.

Exercise 41 (a somewhat surprising value)**.** Show that `R100(`10^{30}`).ulp()` is exactly `1.00000000000000000000000000000`.

11.3.2 Rounding

How to approach a number which cannot be represented exactly in a set of floating-point numbers? There exist different possibilities to define *rounding*:

- in the direction of the nearest representable number: this is what is done in the set `RDF`, and it is the default behaviour of the sets created by `RealField`. For a number exactly in the middle of two representable numbers, rounding is done at the nearest even significand;
- in the direction of $-\infty$; for this, use `RealField(p,rnd='RNDD')` to obtain this behaviour with a precision of p bits;
- in the direction of zero: `RealField(p,rnd='RNDZ')`;
- in the direction of $+\infty$: `RealField(p,rnd='RNDU')`.

11.3.3 Some Properties

Rounding, which is necessary for the sets of floating-point numbers, gives rise to many unexpected effects. Let us explore some of them:

A Dangerous Phenomenon. Known as *catastrophic cancellation* it is the loss of precision which results from the subtraction of two very close numbers; more exactly, it is an amplification of the errors:

```
sage: a = 10000.0; b = 9999.5; c = 0.1; c
0.100000000000000
sage: a1 = a+c # add a small perturbation to a.
sage: a1-b
0.600000000000364
```

Here, the error `c` introduced on `a` makes the computation imprecise (the last 3 digits are false).

Application: Roots of a Quadratic Equation. Even computing the roots of a second-order equation can cause problems. Let us consider the case $a = 1$, $b = 10^4$, $c = 1$:

```
sage: a = 1.0; b = 10.0^4; c = 1.0
sage: delta = b^2-4*a*c
sage: x = (-b-sqrt(delta))/(2*a); y = (-b+sqrt(delta))/(2*a)
sage: x, y
(-9999.99990000000, -0.000100000001111766)
```

The sum of the roots is right, but not their product:

```
sage: x+y+b/a
0.000000000000000
sage: x*y-c/a
1.11766307320238e-9
```

The error is due to the phenomenon known as *catastrophic cancellation* which appears when we add `-b` and `sqrt(delta)` to compute `y`. Here, we can try to find a better approximation for `y`:

```
sage: y = (c/a)/x; y
-0.000100000001000000
sage: x+y+b/a
0.000000000000000
sage: x*y-c/a
-1.11022302462516e-16
```

We can remark that, due to rounding, the sum of the roots remains correct, but the product is much closer to c/a. The reader can consider all different choices for `a`, `b` and `c` to be convinced that writing a numerically robust program to compute the roots of a quadratic trinomial is far from easy.

The Set of Floating-Point Numbers is not an Additive Group. Actually, addition is not associative. Let us use the set `R2` (with 2 bits of precision):

```
sage: x1 = R2(1/2); x2 = R2(4); x3 = R2(-4)
sage: x1, x2, x3
(0.50, 4.0, -4.0)
sage: x1+(x2+x3)
0.50
sage: (x1+x2)+x3
0.00
```

We can deduce that different orders of computations in a program have some importance on the result!

Recurrences and Sequences of Floating-Point Numbers. Let us consider[2] the recurrence $u_{n+1} = 4u_n - 1$. If $u_0 = 1/3$, the sequence is stationary: $u_i = 1/3$ for all i.

```
sage: x = RDF(1/3)
sage: for i in range(1,100): x = 4*x-1; print(x)
0.333333333333
0.333333333333
0.333333333333
...
-1.0
-5.0
-21.0
-85.0
-341.0
-1365.0
-5461.0
```

[2]Thanks to Marc Deléglise (Institut Camille Jordan, Lyon, France) for this example.

```
-21845.0
...
```

The computed sequence is diverging! We can observe that this behaviour is natural, as it is a classical instability phenomenon: every error on u_0 is multiplied by 4 at every iteration, and we know that floating-point arithmetic introduces rounding errors, which will be amplified at every iteration.

Now, let us compute the recurrence $u_{n+1} = 3u_n - 1$, with $u_0 = 1/2$. We expect the same problem: the sequence is constant if computed exactly, but every error will be amplified at every iteration.

```
sage: x = RDF(1/2)
sage: for i in range(1,100): x = 3*x-1; print(x)
0.5
0.5
0.5
...
0.5
```

Now, the computed sequence remains constant! How can we explain these two different behaviours? Let us look at the binary representation of u_0 in both cases.

For the first case ($u_{n+1} = 4u_n - 1$, $u_0 = 1/3$), we have:

$$\frac{1}{3} = \frac{1}{4}\sum_{i=0}^{\infty}\frac{1}{4^i} = \frac{1}{4}\sum_{i=0}^{\infty}\frac{1}{2^{2i}},$$

and therefore 1/3 cannot be represented exactly in the set of floating-point numbers we have at our disposal. The reader of this book is invited to repeat the preceding computation in a large precision set like for example `RealField(1000)` to verify that the computed sequence is always diverging. Let us remark that if, in the first program, we replace the line

```
sage: x = RDF(1/3)
```

by

```
sage: x = 1/3
```

then the computations are carried out in the rational numbers and the iterates will remain equal to 1/3. In the second case ($u_{n+1} = 3u_n - 1$, $u_0 = 1/2$), u_0 and 3/2 in radix 2 are respectively 0.1 and 1.1; therefore they are exactly represented, without rounding, in the different sets of floating-point numbers: computation is exact, and the sequence remains constant.

The following exercise shows that a sequence encoded in a set of floating-point numbers may converge to a wrong limit.

Exercise 42 (an example of Jean-Michel Muller)**.** We consider the sequence (cf. [MBdD+10, p. 9]):

$$u_n = 111 - \frac{1130}{u_{n-1}} + \frac{3000}{u_{n-1}u_{n-2}}.$$

It is possible to show that the general solution is of the form:

$$u_n = \frac{\alpha\ 100^{n+1} + \beta\ 6^{n+1} + \gamma\ 5^{n+1}}{\alpha\ 100^{n} + \beta\ 6^{n} + \gamma\ 5^{n}}.$$

1. Choose $u_0 = 2$ and $u_1 = -4$: what are the values of α, β and γ? To which limit is the sequence converging?
2. Write a program which computes the sequence (still with $u_0 = 2$ and $u_1 = -4$) in the set `RealField()` (or in `RDF`). What can we observe?
3. Explain this behaviour.
4. Carry out the same computation with a large precision, in `RealField(5000)` for example. Comment the result.
5. The recurrence is now defined in $\mathbb{Q}$. Program it in the set of rational numbers and comment the result.

The Summation of Numerical Series. We consider a numerical series with a positive general term u_n. The computation of partial sums $\sum_{i=0}^{m} u_i$ in a set of floating-point numbers is perturbed by rounding errors. The reader can enjoy showing that, if u_n tends to 0 when n tends to infinity, and if the partial sums remain in the interval of representable numbers, then after a certain rank m, the sequence $\sum_{i=0}^{m} u_i$ *computed with rounding* is stationary. In short, in the world of floating-point numbers, life is simple: series with positive general terms tending to 0 converge, provided the partial sums do not grow too much!

For example, let us look at the harmonic (diverging) series, with general term $u_n = 1/n$:

```
sage: def sumharmo(p):
....:     RFP = RealField(p)
....:     y = RFP(1.); x = RFP(0.); n = 1
....:     while x <> y:
....:         y = x; x += 1/n; n += 1
....:     return p, n, x
```

Let us test this function with different values for the precision `p`:

```
sage: sumharmo(2)
(2, 5, 2.0)
sage: sumharmo(20)
(20, 131073, 12.631)
```

The reader can verify using a sheet of paper and a pencil that, in our toy set `R2` of floating-point numbers, the function converges in 5 iterations to the value 2.0. Obviously, the result depends on the precision p, and the reader can also verify (always with a pencil...) that for $n > \beta^p$, the computed sum is stationary. However, be careful! With the default precision of 53 bits and doing 10^9 operations per second, it might need $2^{53}/10^9/3600$ hours, that is about 104 days, to reach the stationary value!

Improving the Computation of some Recurrences. With some care, it is possible to improve some results: here is a useful example.

It is common to encounter recurrences of the form:

$$y_{n+1} = y_n + \delta_n,$$

where the numbers δ_n have a small absolute value when compared to y_n: think for instance of the integration of the celestial mechanics ordinary differential equations to simulate the solar system: large values (of the distances, of the velocities) undergo very small perturbations in the long time [HLW02]. Even if it is possible to compute precisely the δ_n terms, rounding errors when doing the additions $y_{n+1} = y_n + \delta_n$ will introduce important errors.

As an example, consider the sequence defined by $y_0 = 10^{13}$, $\delta_0 = 1$ and $\delta_{n+1} = a\delta_n$ with $a = 1 - 10^{-8}$. The standard, naive, programming way to compute y_n is:

```
sage: def iter(y,delta,a,n):
....:     for i in range(0,n):
....:         y += delta
....:         delta *= a
....:     return y
```

As we have chosen rational values for y_0, δ_0 and a, we can compute the exact value of the iterates with Sage:

```
sage: def exact(y,delta,a,n):
....:     return y+delta*(1-a^n)/(1-a)
```

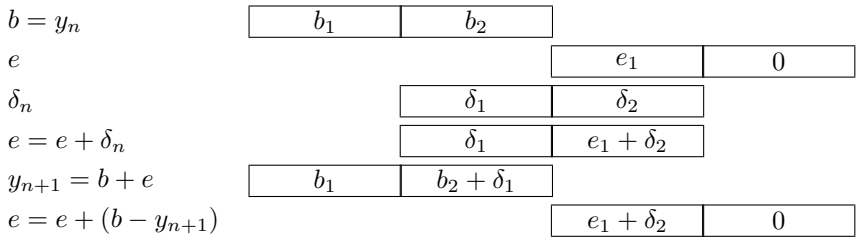
Now, let us compute again 100 000 iterates but with floating-point numbers in `RDF` (for instance), and let us compare the result with the exact value:

```
sage: y0 = RDF(10^13); delta0 = RDF(1); a = RDF(1-10^(-8)); n = 100000
sage: ii = iter(y0,delta0,a,n)
sage: s = exact(10^13,1,1-10^(-8),n)
sage: print("exact - classical summation: %.1f" % (s-ii))
exact - classical summation: -45.5
```

Now, this is the *compensated summation* algorithm:

```
sage: def sumcomp(y,delta,e,n,a):
....:     for i in range(0,n):
....:         b = y
....:         e += delta
....:         y = b+e
....:         e += (b-y)
....:         delta = a*delta # new value of delta
....:     return y
```

To understand the behaviour of this algorithm, let us look at the diagram below (we follow here the presentations of [Hig93] and [HLW02]), where the boxes represent the significand of the numbers. The position of the boxes represent the exponent (the more a box is to the left, the larger its exponent is):



The rounding error accumulates in e, and none of the digits of δ_n is lost, while with the naive method, the digits denoted δ_2 disappear from the computation.

Let us compute again the first 100 000 iterates with the compensated summation method:

```
sage: c = sumcomp(y0,delta0,RDF(0.0),n,a)
sage: print("exact - compensated summation: %.5f" \
            % RDF(s-RR(c).exact_rational()))
exact - compensated summation: -0.00042
```

The absolute error is -45.5 with the naive algorithm, and -0.00042 with the compensated summation! It should also be noted that the relative errors are respectively $4.55 \cdot 10^{-12}$ with the naive method and $4.16 \cdot 10^{-17}$ with the compensated summation.

11.3.4 Complex Floating-Point Numbers

Sage offers two families of complex numbers represented in the computer by pairs of floating-point numbers belonging to the sets we have encountered above:

1. double-precision complex numbers `ComplexDoubleField` (abbreviated `CDF`). These are numbers of the form $x + i \cdot y$ where x and y are both "double-precision" floating-point numbers. They are created like this:

   ```
   sage: x = CDF(2,1.); x
   2.0 + 1.0*I
   sage: y = CDF(20,0); y
   20.0
   ```

   or:

   ```
   sage: z = ComplexDoubleElement(2.,1.); z
   2.0 + 1.0*I
   ```

2. arbitrary precision complex numbers `ComplexField` — or `Complexes`. These are numbers of the form $x + i \cdot y$, where x and y have the same precision of p bits. An instance of the class `ComplexField` creates a set of given precision (53 by default):

   ```
   sage: C = ComplexField(); C(2,3)
   2.00000000000000 + 3.00000000000000*I
   sage: C100 = ComplexField(100); C100(2,3)
   2.0000000000000000000000000000 + 3.0000000000000000000000000000*I
   ```

Of course, computations with these sets face the same rounding problems as with real floating-point numbers.

11.3.5 Methods

We have already seen the `prec` and `ulp` methods. The different sets of numbers we have encountered provide a large amount of methods. Let us give some examples:

- methods which return constants. Examples:

```
sage: R200 = RealField(200); R200.pi()
3.1415926535897932384626433832795028841971693993751058209749
sage: R200.euler_constant()
0.57721566490153286060651209008240243104215933593992359880577
```

- trigonometric functions `sin`, `cos`, `arcsin`, `arccos`, and so on. Example:

```
sage: x = RDF.pi()/2; x.cos() # floating-point approximation of zero!
6.123233995736757e-17
sage: x.cos().arccos() - x
0.0
```

- the logarithms (`log`, `log10`, `log2`, etc.), the hyperbolic functions and their inverses (`sinh`, `arcsinh`, `cosh`, `arccosh`, etc.).

- special functions (`gamma`, `j0`, `j1`, `jn(k)`, and so on).

The reader should look at the Sage documentation to get a complete list of the very large number of available methods. We recall that this list can be obtained in the following way:

```
sage: x = 1.0; x.<tab>
```

For each method, we can get the parameters — if any — and an example of use by typing (here, for the Euler $\Gamma(x)$ function):

```
sage: x.gamma?
```

11.4 Interval and Ball Arithmetic

From what we explained above in this chapter, it should be clear that floating-point numbers only allow us to compute *approximations* of numerical results, and cannot provide proofs. But it turns out that we can compute rigorous *enclosures* of the sought quantities. For example, we generally cannot *prove*, computing with floating-point numbers, that a real number x_0 is the solution of some equation $f(x) = 0$ but we can *prove* that a solution exists in an interval $I = [\underline{x}, \overline{x}]$ or prove that no solution exists in I.

The tools for this are *interval arithmetic* and *ball arithmetic.*

- Given a set $\mathcal{F}$ of floating-point numbers, real interval (or inf-sup) arithmetic computes with intervals $\mathbf{a} = [\underline{a}, \overline{a}] = \{x \in \mathbb{R} : \underline{a} \leq x \leq \overline{a}\}$ (with $\underline{a}$ and $\overline{a} \in \mathcal{F}$). In the following we use bold letters to define an interval, and use the convention that any number $x \in \mathcal{F}$ is represented by the singleton $\mathbf{x} = [x, x]$.
- Real ball arithmetic (or mid-rad interval arithmetic): here we consider numbers $x \in \mathcal{F}$ and an error bound attached to x; balls are sets of the form $\{y \in \mathbb{R} : x - \varepsilon \leq y \leq x + \varepsilon\}$. We will also use bold letters to denote them.

As we will see later these families of sets can be extended to the complex plane.

We just give here a very minimal introduction to this important field; the interested reader should consult [Tuc11] for example (from which we adopt the notations).

Given real valued intervals (or real balls) $\mathbf{a}$ and $\mathbf{b}$, the *four operations* are defined in real valued intervals (or real balls) by:

$$\mathbf{a} * \mathbf{b} = [a * b \text{ for } a \in \mathbf{a} \text{ and } b \in \mathbf{b}],$$

where $*$ stands for one of $+$, $-$, $\times$, $\div$.

Observe that $\mathbf{a} \div \mathbf{b}$ is not yet defined if $0 \in \mathbf{b}$. To get around this, it is necessary to extend $\mathcal{F}$ with the infinite values $\pm\infty$, and to consider infinite intervals in one or both directions. The following table of examples shows the results for all cases where $0 \in \mathbf{b}$, when computing with a class of real intervals:

$\mathbf{a}$	$\mathbf{b}$	Remarks	$\mathbf{a} \div \mathbf{b}$
$[-1, +1]$	$[-2, +3]$	$0 \in \mathbf{a}$	`[-infinity .. +infinity]`
$[-2, -1]$	$[-1, 0]$	$\overline{a} < 0$	`[1.0000000000000000 .. +infinity]`
$[-2, -1]$	$[0, +2]$	$\overline{a} < 0$	`[-infinity .. -0.50000000000000000]`
$[+2, +3]$	$[-3, 0]$	$0 < \underline{a}$	`[-infinity .. -0.66666666666666662]`
$[+2, +3]$	$[0, +4]$	$\underline{a} > 0$	`[0.50000000000000000 .. +infinity]`
$[-2, -1]$	$[0]$	$0 \notin \mathbf{a}$	`[-infinity .. +infinity]`
$[0]$	$[0]$		`[.. NaN ..]`

In the following, we always assume that $\mathcal{F}$ contains $+\infty$ and $-\infty$. Note that the rounding mode must be chosen so as to guarantee the results: thus, when doing arithmetic operations on intervals, one must always round outwards the resulting interval.

11.4.1 Implementation in Sage

Recall that we have arbitrary precision real numbers in Sage (and that different rounding modes are available); then:

- `RealIntervalField(n)` is the set of real intervals with a precision of `n` bits. It is implemented using pairs of numbers from `RealField(n)`. In the default case (53 bits of precision) `RealIntervalField()` can be abbreviated by `RIF`. We can create a `RIF` (or a `RealIntervalField(n)`) object from a numeric expression:

```
sage: r3 = RIF(sqrt(3)); r3
1.732050807568877?
sage: print(r3.str(style="brackets"))
[1.7320508075688769 .. 1.7320508075688775]
```

As $\sqrt{3}$ cannot be exactly represented in `RealField()`, it is represented by an interval which contains the exact value. For printing, the default is to use the "question style", where the "known correct" part of the number, followed by a question mark is printed. The question mark indicates that the preceding digit is possibly wrong by at most ± 1. The "brackets" style prints outward approximations of the minimum and maximum of the interval (bounds are stored in binary: what is printed are decimal values of the bounds rounded in the direction of $-\infty$ and $+\infty$ respectively).

With the following instruction we fix the printing style to "brackets" (replace `brackets` by `question` to go back to the default style):

```
sage: sage.rings.real_mpfi.printing_style = 'brackets'
```

When we create a `RIF` object from a number which can be exactly represented in $\mathcal{F}$, the interval created is a singleton (an interval of diameter 0), as expected:

```
sage: r2 = RIF(2); r2, r2.diameter()
([2.0000000000000000 .. 2.0000000000000000], 0.000000000000000)
```

We can also create intervals of any size by giving two real numbers, and the result is the smallest representable interval which contains them:

```
sage: rpi = RIF(sqrt(2),pi); rpi
[1.4142135623730949 .. 3.1415926535897936]
sage: RIF(0,+infinity)
[0.00000000000000000 .. +infinity]
```

- `RealBallField(n)` is the set of real balls, with a precision of `n` bits. The default case (53 bits of precision) `RealBallField()` can be abbreviated by `RBF`. Some available constructions are:

```
sage: RBF(pi)
[3.141592653589793 +/- 5.61e-16]
sage: RealBallField(100)(pi)
[3.1415926535897932384626433832 +/- 3.83e-30]
```

Observe that, like for intervals, exactly representable numbers produce a ball of radius 0:

```
sage: RBF(2).rad()
0.00000000
```

Some Methods for Intervals and Balls. The following methods do not require further comment:

```
sage: si = sin(RIF(pi))
sage: si.contains_zero()
True
sage: sb = sin(RBF(pi))
sage: sb.contains_zero()
True
```

The standard functions exp, sin, cos, arcsin, arccos, sinh, cosh ... are all defined, and there exist methods to get the centre of an interval, the diameter of an interval or a ball, a bisection method which cuts an interval into two almost equal ones and so on. Note that the middle of an interval is not always in $\mathcal{F}$, and thus it is not always possible to cut an interval into two *equal* ones. Here is an extreme case where the left sub-interval is a singleton, and the right one is identical to the original interval:

```
sage: a = RealIntervalField(30)(1, RR(1).nextabove())
sage: a.bisection()
([1.0000000000 .. 1.0000000000], [1.0000000000 .. 1.0000000019])
```

For the same reason, the centre and diameter might be inexact:

```
sage: b = RealIntervalField(2)(-1,6)
sage: b.center(), b.diameter()
(2.0, 8.0)
```

It is also possible to build `RealBallField()` objects from `RealIntervalField()` intervals and conversely:

```
sage: s = RIF(1,2)
sage: b = RBF(s)
sage: bpi = RBF(pi)
sage: ipi = RIF(bpi)
```

But beware:

```
sage: RIF(RBF(RIF(1,2))) == RIF(1,2)
False
sage: RBF(RIF(RBF(pi))) == RBF(pi)
False
```

Why that? The reason lies in the different implementations of both classes:

- As already said above, `RealIntervalField(n)` objects are represented by pairs of `RealField(n)` numbers, and rely on the `MPFI` library [RR05].
- The implementation of `RealBallField(n)` objects is different: the ball midpoint and its radius are stored, but only the midpoint is stored in full-precision. The radius is represented by a floating-point number with fixed-precision significand and arbitrary-precision exponent. Generally, a few bits suffice for the radius. The implementation in Sage relies on the `Arb` library [Joh13].

Consequently, ball arithmetic is less computationally expensive than inf-sup interval arithmetic and even, at high precision, it is not more expensive than plain floating-point arithmetic.

11.4.2 Computing with Real Intervals and Real Balls

Once we have defined sets of intervals or balls, we must define functions operating on them. For a given interval (or a ball) $\mathbf{x}$, the range of a function f on $\mathbf{x}$ is given by $R(f, \mathbf{x}) = \{f(x) \text{ for } x \in \mathbf{x}\}$. For a continuous function f we just define its interval extension $\mathbf{F}(\mathbf{x})$ as the smallest interval (or ball) included in $\mathcal{F}$ which contains $R(f, \mathbf{x})$.

Examples:

```
sage: E = RIF(-pi/4,pi)
sage: sin(E)
[-0.70710678118654769 .. 1.0000000000000000]
sage: E = RIF(-1,2); exp(E)
[0.36787944117144227 .. 7.3890560989306505]
sage: E = RIF(0,1); log(E)
[-infinity .. -0.00000000000000000]
```

For a set of *standard functions* $(\exp, \sin, \cos, \arcsin, \arccos, \sinh, \cosh, \ldots)$, $\mathbf{F}(\mathbf{x})$ is computed as precisely as possible and is a tight approximation of $R(f, \mathbf{x})$.

But things become less obvious when we compose standard functions and combine them with arithmetic operators. Example:

```
sage: E=RIF(-pi,pi)
sage: f = lambda x: sin(x)/x
sage: f(E)
[-infinity .. +infinity]
```

A mathematician could expect to get the interval $[0, 1]$, extending $s(x) = \sin(x)/x$ by continuity to 1 in 0. But interval arithmetic and ball arithmetic are not substitutes for doing mathematical analysis! Actually for an interval $\mathbf{x}$, the interval $\mathbf{s} = \sin(\mathbf{x})$ is evaluated and then $\mathbf{s} \div \mathbf{x}$ is computed using the rules described above for the division; this explains the result we got: $\sin(x)/x$ does not belong to the set of *standard functions*.

We define the set $\mathcal{E}$ of *elementary functions* as the functions obtained by combining standard functions, constants and variables using arithmetic operations and composition. For example, consider $f(x) = (1 + x^2)\sin(2x + 1)$: this is an elementary function. To evaluate the extension $\mathbf{F}(\mathbf{x})$ of f for a given interval (or ball) $\mathbf{x}$, we evaluate $\mathbf{I}_1 = 2\mathbf{x} + 1$, $\mathbf{I}_2 = 1 + \mathbf{x}^2$, $\mathbf{I}_3 = \sin(\mathbf{I}_1)$ and then $\mathbf{F}(\mathbf{x}) = \mathbf{I}_2.\mathbf{I}_3$ (such a decomposition is generally not unique). From this it is not difficult to deduce that, for an elementary function f, we just have:

$$R(f, \mathbf{x}) \subseteq \mathbf{F}(\mathbf{x}).$$

Moreover, there often exist different possible extensions of a function to intervals: for example let us consider the real valued function $f_1(x) = 1 - x^2$; we

can also write it as $f_2(x) = 1 - x \cdot x$ or $f_3(x) = (1-x) \cdot (1+x)$. It turns out that the extension of these functions to intervals are different. The reader will verify by hand the following results:

```
sage: x = RIF(-1,1)
sage: 1-x^2
[0.00000000000000000 .. 1.0000000000000000]
sage: 1-x*x
[0.00000000000000000 .. 2.0000000000000000]
sage: (1-x)*(1+x)
[0.00000000000000000 .. 4.0000000000000000]
```

Exercise 43. Explain why the outputs of `1-x^2` and `1-x*x` differ.

11.4.3 Some Examples of Applications

Finding Roots by Bisection. Let $f : I \mapsto \mathbb{R}$ be a continuous elementary function. We want to find *all* the zeros of f in I. More precisely, we want to find (tiny) intervals such that their union contains all the zeros of f in I (and with only one root by interval). Recall that for an interval $\mathbf{x} \subset I$, we have $R(f, \mathbf{x}) \subseteq \mathbf{F}(\mathbf{x})$. Conversely, $y \notin \mathbf{F}(\mathbf{x}) \Rightarrow y \notin R(f, \mathbf{x})$. Then we get a simple algorithm: we recursively cut I into sub-intervals until a threshold size is attained; for any sub-interval $\mathbf{s}$, if $0 \notin \mathbf{F}(\mathbf{s})$, we can throw away $\mathbf{s}$ as we know that it cannot contain any root. Moreover, when the bisection is finished, if we know the derivative of f and if it is continuous, we can check that f is monotonic on all the computed intervals, and so obtain a *proof* of the existence of only one root by interval.

Example: find *all* the roots of $\sin(1/x)$ in the interval $[1/64, 1/32]$, with a precision of 100 bits (here, we compute with intervals):

```
sage: def bisect(funct,x,tol,zeros):
....:     if 0 in funct(x):
....:         if x.diameter()>tol:
....:             x1,x2 = x.bisection()
....:             bisect(funct,x1,tol,zeros)
....:             bisect(funct,x2,tol,zeros)
....:         else:
....:             zeros.append(x)
sage: sage.rings.real_mpfi.printing_style = 'question'
sage: fs = lambda x: sin(1/x)
sage: d = RealIntervalField(100)(1/64,1/32)
sage: zeros = []
sage: bisect(fs,d,10^(-25),zeros)
sage: for s in zeros:
....:     s
0.015915494309189533576888377?
0.016753151904410035344093037?
0.017683882565766148418764877?
```

```
0.018724110951987686561045148?
0.019894367886486916971111047?
0.021220659078919378102517835?
0.022736420441699333681269 11?
0.024485375860291590118289809?
0.026525823848649222628147293?
0.028937262380344606503433 42?
sage: dfs = lambda x: -cos(1/x)/x^2
sage: not any([dfs(z).contains_zero() for z in zeros])
True
```

So, $\sin(1/x)$ has exactly 10 roots in the interval $[1/64, 1/32]$: this computation is a *proof*.

Note that Newton's method (see page 270) can be generalised to interval arithmetic [Tuc11].

Proving that a Matrix is not Singular. Let us consider the matrix M of size n with term $M_{i,j} = (1 + \log i)/(i^2 + j^2)$:

```
sage: def NearlySingularMatrix(R,n):
....:     M=matrix(R,n,n)
....:     for i in range(0,n):
....:         for j in range(0,n):
....:             M[i,j]= (1+log(R(1+i)))/((i+1)^2+(j+1)^2)
....:     return M
```

We fix $n = 35$. Let us build M in `RDF` and compute its determinant:

```
sage: n=35
sage: NearlySingularMatrix(RDF,n).det()
0.0
```

So, the determinant seems to be zero. Now we compute with `RBF` balls:

```
sage: NearlySingularMatrix(RBF,n).det().contains_zero()
True
```

We cannot conclude whether the matrix is singular or not, as the computed ball determinant contains zero. Let us compute with balls of growing precision:

```
sage: def tryDet(R,n):
....:     p = 53
....:     z = True
....:     while z:
....:         p += 100
....:         MRF=NearlySingularMatrix(R(p),n)
....:         d = MRF.det()
....:         z = d.contains_zero()
....:     return p,d
sage: tryDet(RealBallField,n)
```

```
(1653, [9.552323592707808e-485 +/- 1.65e-501])
```

For a precision of 1653 bits, we have found a ball $9.552323592707808 \cdot 10^{-485} \pm 1.65 \cdot 10^{-501}$ which contains the exact value of the determinant, and thus the determinant of M is not equal to zero. We have *proven* that the matrix is non singular (note that a more serious program should include, for safety, a limit to the loop on the precision `p`).

We can make the same computation with intervals:

```
sage: tryDet(RealIntervalField,n)
(1653, 9.552323592707808?e-485)
```

Here also we get the same conclusion: M is non singular. Let us compare the computing times:

```
sage: time p,d = tryDet(RealBallField,n)
CPU times: user 4.75 s, sys: 12 ms, total: 4.76 s
Wall time: 4.75 s
sage: time p,d = tryDet(RealIntervalField,n)
CPU times: user 6.62 s, sys: 8 ms, total: 6.63 s
Wall time: 6.62 s
```

As could be expected (1653 bits is a large precision!) `RealBallField` computations are less expensive than `RealIntervalField` ones.

11.4.4 Complex Intervals and Complex Balls

`ComplexIntervalField(n)` and `ComplexBallField(n)` define square boxes in the complex plane, with a precision of `n` bits. The default cases (53 bits of precision) can be called `CIF` and `CBF`. Constructors accept numerical complex quantities:

```
sage: CBF(sqrt(2),pi)
[1.414213562373095 +/- 4.10e-16] + [3.141592653589793 +/- 5.61e-16]*I
sage: CIF(sqrt(2),pi)
1.414213562373095? + 3.141592653589794?*I
sage: CIF(sqrt(2)+pi*I)
1.414213562373095? + 3.141592653589794?*I
sage: CBF(sqrt(2)+pi*I)
[1.414213562373095 +/- 4.10e-16] + [3.141592653589793 +/- 5.61e-16]*I
```

and real intervals or balls:

```
sage: c = CIF(RIF(1,2),RIF(-3,3))
sage: c.real()
[1.0000000000000000 .. 2.0000000000000000]
sage: c.imag()
[-3.0000000000000000 .. 3.0000000000000000]
sage: CBF(RIF(1,2),RIF(-3,3))
[+/- 2.01] + [+/- 3.01]*I
```

Standard functions are defined, and also methods to compute the argument, the norm and so on:

```
sage: ComplexIntervalField(100)(1+I*pi).arg()
1.2626272556789116834444322083 61?
sage: ComplexBallField(100)(1+I*pi).arg()
[1.2626272556789116834444322083 60 +/- 6.60e-30]
sage: ComplexIntervalField(100)(1+I*pi).norm()
10.869604401089358618834490999 9?
```

11.4.5 Usage and Limitations

First, not every computation can be carried out with balls or intervals. For example, finding the roots of a polynomial is not implemented:

```
sage: (x^3+2*x-1).roots(ring=RR)
[(0.453397651516404, 1)]
sage: (x^3+2*x-1).roots(ring=RBF)
...
NotImplementedError: root finding for this polynomial not implemented
```

(and the same happens with `ring=RIF`). Linear algebra also has some limitations: for example matrix inversion, computation of echelon form, minimal polynomial, solution of linear systems are available, but not eigenvalue computations.

Secondly, in case both `RealBallField` and `RealIntervalField` are available for a given computation, which one should we use? There is no definitive answer but the examples above show that when dealing with potentially large intervals, it is easier to use `RealIntervalField`, and that `RealBallField` is faster when computing with numbers tainted by errors.

11.4.6 Interval Arithmetic is Used by Sage

Internally, Sage uses interval arithmetic; for example Sage can compute in the field of algebraic numbers (roots of polynomials in $\mathbb{Z}[x]$). Numbers are represented by their minimal polynomial and computations are exact. But, how can we print the results? How can we get a human understandable result? Interval arithmetic is the answer (see also pages 140 and 275 of this book):

```
sage: x=QQbar(sqrt(3)); x
1.732050807568878?
sage: x.interval(RealIntervalField(100))
1.732050807568877293527446341 51?
```

11.5 Conclusion

All numerical methods implemented in Sage, as those described in the next chapters, have been theoretically studied: this numerical analysis includes studying

Sets of floating-point numbers	
Machine floating-point numbers	`RDF`
Machine complex floating-point numbers	`CDF`
Real numbers with a precision of p bits	`RealField(p)`
Complex numbers with a precision of p bits	`ComplexField(p)`
Real intervals with a precision of p bits	`RealIntervalField(p)`
Complex intervals with a precision of p bits	`ComplexIntervalField(p)`
Real balls with a precision of p bits	`RealBallField(p)`
Complex balls with a precision of p bits	`ComplexBallField(p)`

TABLE 11.2 – A summary of floating-point classes.

the convergence of the iterative methods, the error introduced when simplifying a problem to make it computable, but also the behaviour of the computations in presence of perturbations such as those introduced by the inexact arithmetic of the floating-point numbers.

Let us consider an algorithm $\mathcal{F}$ which, from some data d, computes $x = \mathcal{F}(d)$. This algorithm can only be used if it does not increase the errors on d too much: to a perturbation ε of d corresponds a perturbed solution $x_\varepsilon = \mathcal{F}(d + \varepsilon)$. It is absolutely necessary that the error $x_\varepsilon - x$ introduced depends only moderately on ε (in a continuous way, does not grow too fast, ...): all numerical algorithms must have stability properties to be usable. In Chapter 13, we will explore the stability problems for the algorithms used in numerical linear algebra.

Let us also remark that some computations are definitively not possible in finite precision, like for example the sequence given in Exercise 42: every perturbation, however small, will lead the sequence to converge to a wrong value. This is a typical instability for the solution of a problem: the experimental study of a sequence with floating-point numbers should be performed with great care.

The reader might find that performing computations with floating-point numbers is hopeless, but this opinion should be moderated: an overwhelming part of available computing resources is used to perform computations in these sets of numbers: the approximate solution of partial differential equations, optimisation or signal processing, etc. Floating-point numbers should be used with care, but they did not prevent the development of computing and its applications: rounding errors do not limit the validity of numerical weather forecasts, to give only one obvious example.

Interval arithmetic seems to appear first in the works of Ramon E. Moore in 1966 [Moo66] (even if the idea appears before the computer era), but new software developments (such as `MPFI` and `Arb` used by Sage) as well as some spectacular applications draw attention to it: some famous old mathematical conjectures have proofs which rely at least partially on interval computations. For example, we may mention the Kepler conjecture in 1998 by Thomas C. Hales, or the ternary Goldbach problem by Harald A. Helfgott in 2013.

12
Non-Linear Equations

This chapter explains how to *solve* a non-linear equation using Sage. We first study polynomial equations and show the limitations of the search for exact solutions. We then describe some classical numerical solving methods, while indicating the numerical algorithms implemented in Sage.

12.1 Algebraic Equations

An algebraic equation is an equation of the form $p(x) = 0$, where p is a polynomial in one variable with coefficients in an integral domain A. We say that an element $\alpha \in A$ is a *root* of the polynomial p if $p(\alpha) = 0$.

Let α be an element of A. Polynomial long division of p by $x - \alpha$ yields a constant polynomial r such that

$$p = (x - \alpha)q + r.$$

Upon evaluating this equation at $x = \alpha$, we get $r = p(\alpha)$. So the polynomial $x - \alpha$ divides p if and only if α is a root of p. This remark leads to the notion of *multiplicity* of a root α of the polynomial p: it is the largest integer m such that $(x - \alpha)^m$ divides p. We note that the sum of the multiplicities of the roots of p is less than or equal to the degree of p.

12.1.1 The Method `Polynomial.roots()`

To solve the algebraic equation $p(x) = 0$ means to identify the roots of the polynomial p and their multiplicities. The method `Polynomial.roots()` finds the roots of a polynomial. It takes up to three parameters, all of them optional. The parameter `ring` indicates the ring in which to search for the roots. If we do not

give a value to this parameter, Sage uses the coefficient ring of the polynomial. The boolean parameter `multiplicities` specifies whether the information returned by `Polynomial.roots()` should consist of both roots and multiplicities. The parameter `algorithm` indicates which algorithm to use; the possible values are described below (see §12.2.2).

```
sage: R.<x> = PolynomialRing(RealField(prec=10))
sage: p = 2*x^7 - 21*x^6 + 64*x^5 - 67*x^4 + 90*x^3 \
....: + 265*x^2 - 900*x + 375
sage: p.roots()
[(-1.7, 1), (0.50, 1), (1.7, 1), (5.0, 2)]
sage: p.roots(ring=ComplexField(10), multiplicities=False)
[-1.7, 0.50, 1.7, 5.0, -2.2*I, 2.2*I]
sage: p.roots(ring=RationalField())
[(1/2, 1), (5, 2)]
```

12.1.2 Representation of Numbers

We recall how to denote commonly-used rings with Sage (see §5.2). Integers are represented by objects of the class `Integer`; conversions are performed using the *parent* `ZZ` or the function `IntegerRing()`, which returns the object `ZZ`. Similarly, rational numbers are represented by objects of the class `Rational`; the parent of these objects is the object `QQ`, which is returned by the function `RationalField()`. In both cases Sage uses the arbitrary precision library GMP. Without diving into the inner workings of this library, the integers used by Sage have arbitrary size, limited only by the available memory of the machine on which the software is run.

Several approximate representations of real numbers are available (see Chapter 11). There is `RealField()` — or `Reals()` — for floating-point numbers with a given precision and, in particular, `RR` for 53-bit precision. Representations using machine double precision are afforded by `RDF`, short for `RealDoubleField()`. There is also the class `RIF` — short for `RealIntervalField()` — in which a real number is represented by an interval containing it; the endpoints of this interval are floating-point numbers.

The analogous representations for complex numbers are: `CC`, `CDF`, and `CIF`. Again, to which object is associated a function, namely `ComplexField()`, `ComplexDoubleField()`, and `ComplexIntervalField()`.

The computations performed by `Polynomial.roots()` are exact or approximate depending on the representation of the polynomial's coefficients and (if specified) the value of the parameter `ring`: for instance, given `ZZ` or `QQ`, the computation is exact; given `RR` it is approximate. In the second part of this chapter we describe the algorithm used for the computation of the roots and specify the role of the parameters `ring` and `algorithm` (see §12.2).

Solving algebraic equations is intimately related to the notion of number. The *splitting field* of a nonconstant polynomial p is the smallest field extension of the coefficient field in which p is a product of polynomials of degree 1; one can prove that such an extension always exists. In Sage, we can construct the splitting field

of an irreducible polynomial with the method `Polynomial.root_field()`. We can then compute with the roots *implicitly* contained in the splitting field.

```
sage: R.<x> = PolynomialRing(QQ, 'x')
sage: p = x^4 + x^3 + x^2 + x + 1
sage: K.<alpha> = p.root_field()
sage: p.roots(ring=K, multiplicities=None)
[alpha, alpha^2, alpha^3, -alpha^3 - alpha^2 - alpha - 1]
sage: alpha^5
1
```

12.1.3 The Fundamental Theorem of Algebra

The splitting field of the polynomial with real coefficients $x^2 + 1$ is precisely the field of complex numbers. It is remarkable that every nonconstant polynomial with complex coefficients has at least one complex root: this is the content of the *Fundamental Theorem of Algebra*. Therefore, every nonconstant complex polynomial is a product of polynomials of degree 1. We noted above that the sum of the multiplicities of the roots of a polynomial p is less than or equal to the degree of p. In other words, every polynomial equation of degree n with complex coefficients has n complex roots, counted with multiplicity.

Let's see how the method `Polynomial.roots()` can be used to illustrate this result. In the following example, we construct the ring of polynomials with real coefficients (represented by floating-point numbers with a precision of 53 bits). Then a polynomial of degree less than 15 is picked randomly from this ring. Finally we add the multiplicities of the complex roots computed with the method `Polynomial.roots()` and we compare this sum to the degree of the polynomial.

```
sage: R.<x> = PolynomialRing(RR, 'x')
sage: d = ZZ.random_element(1, 15)
sage: p = R.random_element(d)
sage: p.degree() == sum(r[1] for r in p.roots(CC))
True
```

12.1.4 Distribution of the Roots

We give a curious illustration of the power of the method `Polynomial.roots()`: we plot all points in the complex plane whose corresponding complex number is a root of a polynomial of degree 12 and with coefficients in the set $\{-1, 1\}$. The choice of degree is rather arbitrary, made in order to obtain a sufficiently detailed plot in a short amount of time. The usage of approximate values for the complex numbers is also motivated by performance reasons (see §13).

```
sage: def build_complex_roots(degree):
....:     R.<x> = PolynomialRing(CDF, 'x')
....:     v = []
....:     for c in cartesian_product([[-1, 1]] * (degree + 1)):
....:         v.extend(R(list(c)).roots(multiplicities=False))
....:     return v
```

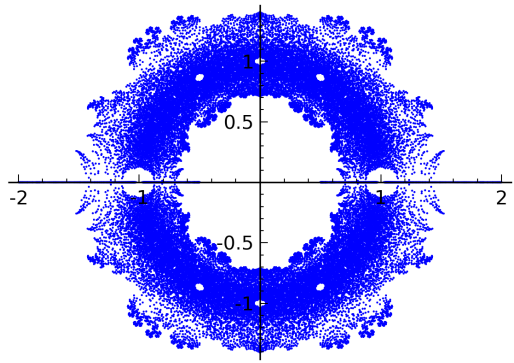


FIGURE 12.1 – Distribution of roots of all degree-12 polynomials with coefficients −1 or 1.

```
sage: data = build_complex_roots(12)
sage: points(data, pointsize=1, aspect_ratio=1)
```

12.1.5 Solvability in Radicals

In certain cases it is possible to compute the exact value of the roots of a polynomial. This occurs for instance when we can express the roots in terms of the coefficients of the polynomial and using radicals (square roots, cubic roots, etc.). We then say that the polynomial is *solvable in radicals*.

To solve in radicals with Sage, we must work with objects of the class `Expression` which represent symbolic expressions. We have seen that integers represented by objects of the class `Integer` have the same *parent*, namely the object `ZZ`. Similarly, the objects of the class `Expression` have the same parent: the object `SR` (short for *Symbolic Ring*); this allows to convert into the class `Expression`.

```
sage: a, b, c, x = var('a, b, c, x')
sage: p = a * x^2 + b * x + c
sage: type(p)
<type 'sage.symbolic.expression.Expression'>
sage: p.parent()
Symbolic Ring
sage: p.roots(x)
[(-1/2*(b + sqrt(b^2 - 4*a*c))/a, 1),
 (-1/2*(b - sqrt(b^2 - 4*a*c))/a, 1)]
```

Degree Strictly Bigger than 2. Algebraic equations of degree 3 and 4 are solvable in radicals. However, it is generally impossible to solve in radicals polynomial equations of degree at least 5 (see §7.3.4). This impossibility leads to investigate numerical solution methods (see §12.2).

```
sage: a, b, c, d, e, f, x = var('a, b, c, d, e, f, x')
sage: p = a*x^5+b*x^4+c*x^3+d*x^2+e*x+f
sage: p.roots(x)
Traceback (most recent call last):
...
RuntimeError: no explicit roots found
```

We use Sage to illustrate a method for solving equations of degree 3 over the field of complex numbers. We start by showing that the general equation of degree 3 can be reduced to the form $x^3 + px + q = 0$.

```
sage: x, a, b, c, d = var('x, a, b, c, d')
sage: P = a * x^3 + b * x^2 + c * x + d
sage: alpha = var('alpha')
sage: P.subs(x = x + alpha).expand().coefficient(x, 2)
3*a*alpha + b
sage: P.subs(x = x - b / (3 * a)).expand().collect(x)
a*x^3 - 1/3*(b^2/a - 3*c)*x + 2/27*b^3/a^2 - 1/3*b*c/a + d
```

To obtain the roots of an equation of the form $x^3+px+q = 0$, we set $x = u+v$.

```
sage: p, q, u, v = var('p, q, u, v')
sage: P = x^3 + p * x + q
sage: P.subs(x = u + v).expand()
u^3 + 3*u^2*v + 3*u*v^2 + v^3 + p*u + p*v + q
```

Let's assume the last expression is zero. We then note that $u^3 + v^3 + q = 0$ is equivalent to $3uv + p = 0$; moreover, if these equalities hold, u^3 and v^3 satisfy an equation of degree two: $(X - u^3)(X - v^3) = X^2 - (u^3 + v^3)X + (uv)^3 = X^2 + qX - p^3/27$.

```
sage: P.subs({x: u + v, q: -u^3 - v^3}).factor()
(3*u*v + p)*(u + v)
sage: P.subs({x: u+v, q: -u^3 - v^3, p: -3 * u * v}).expand()
0
sage: X = var('X')
sage: solve([X^2 + q*X - p^3 / 27 == 0], X, solution_dict=True)
[{X: -1/2*q - 1/18*sqrt(12*p^3 + 81*q^2)},
 {X: -1/2*q + 1/18*sqrt(12*p^3 + 81*q^2)}]
```

The solutions of the equation $x^3 + px + q = 0$ are therefore the sums $u + v$ where u and v are the cube roots of

$$-\frac{\sqrt{4p^3 + 27q^2}\sqrt{3}}{18} - \frac{q}{2} \quad \text{and} \quad \frac{\sqrt{4p^3 + 27q^2}\sqrt{3}}{18} - \frac{q}{2}$$

satisfying $3uv + p = 0$.

12.1.6 The Method `Expression.roots()`

The preceding examples use the method `Expression.roots()`. This method returns a list of exact roots, not guaranteed to be complete. Among the optional parameters of this method, we find the parameters `ring` and `multiplicities` already seen for the method `Polynomial.roots()`. It is important to keep in mind that the method `Expression.roots()` is not restricted to polynomial expressions.

```
sage: e = sin(x) * (x^3 + 1) * (x^5 + x^4 + 1)
sage: roots = e.roots(); len(roots)
9
sage: roots
[(0, 1),
 (-1/2*(1/18*sqrt(23)*sqrt(3) - 1/2)^(1/3)*(I*sqrt(3) + 1) - 1/6*(-I*
     sqrt(3) + 1)/(1/18*sqrt(23)*sqrt(3) - 1/2)^(1/3),
  1),
 (-1/2*(1/18*sqrt(23)*sqrt(3) - 1/2)^(1/3)*(-I*sqrt(3) + 1) - 1/6*(I*
     sqrt(3) + 1)/(1/18*sqrt(23)*sqrt(3) - 1/2)^(1/3),
  1),
 ((1/18*sqrt(23)*sqrt(3) - 1/2)^(1/3) + 1/3/(1/18*sqrt(23)*sqrt(3) -
     1/2)^(1/3),
  1),
 (-1/2*I*sqrt(3) - 1/2, 1),
 (1/2*I*sqrt(3) - 1/2, 1),
 (1/2*I*sqrt(3)*(-1)^(1/3) - 1/2*(-1)^(1/3), 1),
 (-1/2*I*sqrt(3)*(-1)^(1/3) - 1/2*(-1)^(1/3), 1),
 ((-1)^(1/3), 1)]
```

If the parameter `ring` is not specified, the method `roots()` of the class `Expression` delegates the computation of the roots to Maxima, which tries to factor the expression, then solves in radicals each factor of degree strictly less than 5. When the parameter `ring` is specified, the expression is converted into an object of the class `Polynomial` whose coefficients have as parent the object identified by the parameter `ring`; then the result of the method `Polynomial.roots()` is returned. We will describe the algorithm used in this case below (see §12.2.2).

It is also possible to compute with implicit roots, which we can access via the objects `QQbar` and `AA` representing the field of algebraic numbers (see §7.3.2).

Elimination of Multiple Roots. Given a polynomial p with multiple roots, it is possible to construct a polynomial with simple roots (that is, of multiplicity 1), identical to those of p. Hence, when computing the roots of a polynomial, we can assume that these roots are simple. We first prove the existence of the polynomial with simple roots and then show how to construct it. This will give a new illustration of the method `Expression.roots()`.

Let α be a root of the polynomial p with multiplicity $m > 1$. It is a root of the derivative p' with multiplicity $m - 1$. In fact, if $p = (x - \alpha)^m q$ then $p' = (x - \alpha)^{m-1}(mq + (x - \alpha)q')$.

```
sage: alpha, m, x = var('alpha, m, x'); q = function('q')(x)
sage: p = (x - alpha)^m * q
sage: p.derivative(x)
(-alpha + x)^(m - 1)*m*q(x) + (-alpha + x)^m*diff(q(x), x)
sage: simplify(p.derivative(x)(x=alpha))
0
```

Therefore, the gcd of p and p' is the product $\prod_{\alpha\in\Gamma}(x-\alpha)^{m_\alpha-1}$, where Γ is the set of roots of p with multiplicity strictly greater than 1, and m_α is the multiplicity of the root α. If d denotes this gcd, then the quotient p/d has the desired properties.

Note that the degree of the quotient of p by d is strictly less than the degree of p. In particular, if this degree is strictly less than 5, it is possible to express the roots in terms of radicals. The following example illustrates this for a polynomial of degree 13 with rational coefficients.

```
sage: R.<x> = PolynomialRing(QQ, 'x')
sage: p = 128 * x^13 - 1344 * x^12 + 6048 * x^11 - 15632 * x^10 \
....: + 28056 * x^9 - 44604 * x^8 + 71198 * x^7 - 98283 * x^6   \
....: + 105840 * x^5 - 101304 * x^4 + 99468 * x^3 - 81648 * x^2 \
....: + 40824 * x - 8748
sage: d = gcd(p, p.derivative())
sage: (p // d).degree()
4
sage: roots = SR(p // d).roots(multiplicities=False)
sage: roots
[1/2*I*sqrt(3)*2^(1/3) - 1/2*2^(1/3),
 -1/2*I*sqrt(3)*2^(1/3) - 1/2*2^(1/3),
 2^(1/3),
 3/2]
sage: [QQbar(p(alpha)).is_zero() for alpha in roots]
[True, True, True, True]
```

12.2 Numerical Solution

There is a traditional dichotomy in mathematics between the *discrete* and the *continuous*. Numerical analysis bridges this gap to some extent: a major aspect of numerical analysis is the study of questions about real numbers (firmly part of the continuous domain) via an experimental point of view, often assisted by a computer, which belongs to the discrete domain.

Regarding the solution of non-linear equations, numerous natural questions arise beyond the computation of approximate values: how many real roots does a given equation have? How many imaginary, positive, or negative roots are there?

In this section we start to answer such questions in the special case of algebraic equations. Then we describe some of the methods for computing approximate solutions of a non-linear equation.

12.2.1 Location of Solutions of Algebraic Equations

Descartes' Rule. Descartes' rule of signs is the following: the number of positive roots of a polynomial with real coefficients is less than or equal to the number of sign changes in the sequence of coefficients of the polynomial.

```
sage: R.<x> = PolynomialRing(RR, 'x')
sage: p = x^7 - 131/3*x^6 + 1070/3*x^5 - 2927/3*x^4 \
....: + 2435/3*x^3 - 806/3*x^2 + 3188/3*x - 680
sage: l = [c for c in p.coefficients(sparse=False) if not c.is_zero()]
sage: sign_changes = [l[i]*l[i+1] < 0 for i in range(len(l)-1)].count(True)
sage: real_positive_roots = \
....: sum([alpha[1] if alpha[0] > 0 else 0 for alpha in p.roots()])
sage: sign_changes, real_positive_roots
(7, 5)
```

Indeed, let p be a degree-d polynomial with real coefficients and let p' be its derivative. We denote by u and u' the sequences of the signs of the coefficients of the polynomials p and p': we have $u_i = \pm 1$ depending on whether the degree-i coefficient of p is positive or negative. The sequence u' is a simple truncation of u: we have $u'_i = u_{i+1}$ for $0 \leq i < d$. If follows that the number of sign changes of the sequence u is at most one plus the number of sign changes of the sequence u'.

On the other hand, the number of positive roots of p is at most equal to one plus the number of positive roots of p': any interval whose endpoints are roots of p always contains a root of p'.

As Descartes' rule holds for a degree-1 polynomial, the above observations show that it also holds for a degree-2 polynomial, etc.

Moreover, the difference between the number of positive roots and the number of sign changes in the sequence of coefficients is always even.

Isolating the Real Roots of a Polynomial. Given a polynomial with real coefficients, we have seen that it is possible to bound from above the number of roots contained in the interval $[0, +\infty)$. More generally, there are methods for finding the number of roots contained in a given interval.

One such result is Sturm's Theorem. Let p be a degree-d polynomial with real coefficients and let $[a, b]$ be an interval. We construct recursively a sequence of polynomials. We start with $p_0 = p$ and $p_1 = p'$; then p_{i+2} is the negative of the remainder of the polynomial division of p_i by p_{i+1}. By evaluating this sequence of polynomials at the points a and b we get two finite real sequences $(p_0(a), \ldots, p_d(a))$ and $(p_0(b), \ldots, p_d(b))$. Sturm's Theorem is the following: if p has simple roots, $p(a) \neq 0$ and $p(b) \neq 0$, then the number of roots of p contained in the interval $[a, b]$ equals the number of sign changes of the sequence $(p_0(a), \ldots, p_d(a))$ minus the number of sign changes of the sequence $(p_0(b), \ldots, p_d(b))$.

Here is how we can implement this theorem in Sage.

```
sage: def count_sign_changes(p):
....:     l = [c for c in p if not c.is_zero()]
....:     changes = [l[i]*l[i + 1] < 0 for i in range(len(l) - 1)]
....:     return changes.count(True)
```

```
sage: def sturm(p, a, b):
....:     assert p.degree() > 2
....:     assert not (p(a) == 0)
....:     assert not (p(b) == 0)
....:     assert a <= b
....:     remains = [p, p.derivative()]
....:     for i in range(p.degree() - 1):
....:         remains.append(-(remains[i] % remains[i + 1]))
....:     evals = [[], []]
....:     for q in remains:
....:         evals[0].append(q(a))
....:         evals[1].append(q(b))
....:     return count_sign_changes(evals[0]) \
....:            - count_sign_changes(evals[1])
```

Here is an example of usage of this function `sturm()`.

```
sage: R.<x> = PolynomialRing(QQ, 'x')
sage: p = (x - 34) * (x - 5) * (x - 3) * (x - 2) * (x - 2/3)
sage: sturm(p, 1, 4)
2
sage: sturm(p, 1, 10)
3
sage: sturm(p, 1, 200)
4
sage: p.roots(multiplicities=False)
[34, 5, 3, 2, 2/3]
sage: sturm(p, 1/2, 35)
5
```

12.2.2 Iterative Approximation Methods

In this section we illustrate various ways of approximating the solutions of a non-linear equation $f(x) = 0$. There are essentially two approaches to such approximations. The most efficient algorithm mixes the two approaches.

The first approach constructs a sequence of nested intervals that contain a solution of the equation. We control the precision and convergence is guaranteed, but the convergence speed is not always good.

The second approach starts with a given approximate value of one of the solutions of the equation. If the local behaviour of the function f is sufficiently regular, we can compute a new, more accurate approximation to the solution. By recurrence, we get a sequence of approximate values. This approach assumes that we know a first approximation of the desired number. Moreover, its performance depends on a good local behaviour of the function f: we cannot dictate *a priori* the precision of the answer; worse, the convergence of the sequence of approximations is not necessarily guaranteed.

Throughout this section, we consider a non-linear equation $f(x) = 0$ where f is a numerical function defined on the interval $[a, b]$ and continuous on this interval. We assume that the values of f at the endpoints of the interval $[a, b]$ are non-zero and of opposite signs: in other words, the product $f(a)f(b)$ is strictly negative. The continuity of f guarantees the existence of at least one solution of the equation $f(x) = 0$ in the interval $[a, b]$.

For each of the methods we discuss, we experiment with the following function.

```
sage: f(x) = 4 * sin(x) - exp(x) / 2 + 1
sage: a, b = RR(-pi), RR(pi)
sage: bool(f(a) * f(b) < 0)
True
```

We should note that, for this function, the command `solve` is not useful.

```
sage: solve(f(x) == 0, x)
[sin(x) == 1/8*e^x - 1/4]
```

```
sage: f.roots()
Traceback (most recent call last):
 ...
RuntimeError: no explicit roots found
```

The algorithms for solving non-linear equations can be very time-consuming: it is advisable to take certain precautions before proceeding. In particular, we should ensure that solutions exist by studying the continuity and differentiability of the function in question, as well as its sign changes; plotting the graph of the function can be useful here (see Chapter 4).

The Bisection Method. This method is based on the first approach: construct a sequence of nested intervals, each of which containing a solution of the equation $f(x) = 0$.

We find the midpoint c of the interval $[a, b]$. Suppose that $f(c) \neq 0$ (otherwise we have found a solution). Either $f(a)f(c)$ is negative, so that the interval $[a, c]$ must contain a solution of the equation; or $f(c)f(b)$ is negative, so that the interval $[c, b]$ contains a solution of the equation. Therefore, we can construct an interval containing a solution, and whose length is half the length of the interval $[a, b]$. By iterating this construction, we obtain a sequence of intervals with the expected properties. To implement this approach, we define a Python function `intervalgen` as follows.

```
sage: def phi(s, t): return (s + t) / 2
sage: def intervalgen(f, phi, s, t):
....:     msg = 'Wrong arguments: f({0})*f({1})>=0)'.format(s, t)
....:     assert (f(s) * f(t) < 0), msg
....:     yield s
....:     yield t
....:     while True:
....:         u = phi(s, t)
```

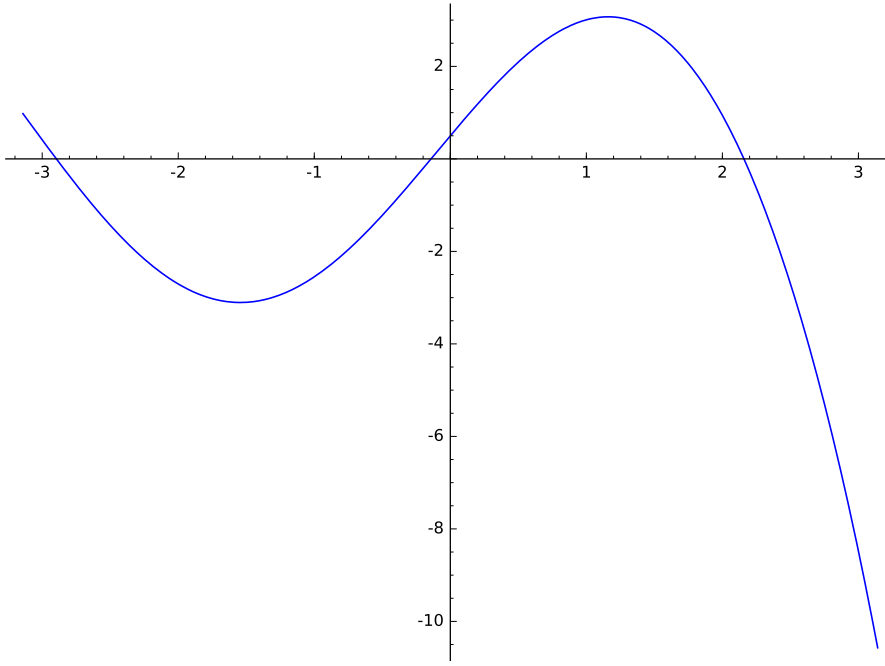


FIGURE 12.2 – Graph of the function f.

```
....:         yield u
....:         if f(u) * f(s) < 0:
....:             t = u
....:         else:
....:             s = u
```

The definition of this function deserves explanation. The keyword `yield` in the definition of `intervalgen` turns it into a *generator* (see §15.2.4). When the method `next()` of a generator is called, if the interpreter sees the keyword `yield`, all local data are saved, the execution is interrupted and the expression immediately at the right of the keyword is returned. The following call of the method `next()` restores the local data that was saved before the interruption and continues from the line following the keyword `yield`. The usage of the keyword `yield` inside an infinite loop (`while True:`) allows the implementation of a recursive sequence via a syntax that resembles closely its mathematical definition. It is possible to stop the execution completely by using the keyword `return`.

The parameter `phi` is a function that specifies the approximation method. For the bisection method, this function computes the midpoint of an interval. In order to test other iterative approximation methods that also use nested intervals, we need to give a new definition of the function `phi` and can then use the function `intervalgen` to construct the corresponding generator.

The parameters `s` and `t` of the function specify the endpoints of the first

interval. The call to `assert` verifies that the function f changes sign between the endpoints of this interval; as we have seen, this guarantees the existence of a solution.

The first two values of the generator are the values of the parameters `s` and `t`. The third value is the midpoint of the corresponding interval. The parameters `s` and `t` then represent the endpoints of the last interval computed. After evaluating f at the midpoint of this interval, we change one of the endpoints of the interval in such a way that the new interval still contains a solution. We agree to take as approximation of the desired solution the midpoint of the last interval computed.

We experiment with the chosen example: here are three approximations obtained by the bisection method applied to the interval $[-\pi, \pi]$.

```
sage: a, b
(-3.14159265358979, 3.14159265358979)
sage: bisection = intervalgen(f, phi, a, b)
sage: bisection.next()
-3.14159265358979
sage: bisection.next()
3.14159265358979
sage: bisection.next()
0.000000000000000
```

In order to compare the different approximation methods, it is useful to automate the computation of approximate solutions to the equation $f(x) = 0$ using the generators defined in Sage for each of the methods. This mechanism should allow us to control the precision and the maximum number of iterations. This role is taken by the function `iterate` defined below.

```
sage: from types import GeneratorType, FunctionType
sage: def checklength(u, v, w, prec):
....:     return abs(v - u) < 2 * prec
sage: def iterate(series, check=checklength, prec=10^-5, maxit=100):
....:     assert isinstance(series, GeneratorType)
....:     assert isinstance(check, FunctionType)
....:     niter = 2
....:     v, w = series.next(), series.next()
....:     while niter <= maxit:
....:         niter += 1
....:         u, v, w = v, w, series.next()
....:         if check(u, v, w, prec):
....:             print('After {0} iterations: {1}'.format(niter, w))
....:             return
....:     print('Failed after {0} iterations'.format(maxit))
```

The parameter `series` must be a generator. We keep the last three values of this generator to verify convergence. This is the role of the parameter `check`: a function that may or may not stop the iterations. By default the function `iterate` uses the function `checklength` which stops the iterations if the last interval computed has length strictly less than twice the parameter `prec`; this

guarantees that the value computed by bisection approximates the solution with an error strictly less than `prec`.

An exception is raised once the number of iterations is larger than the parameter `maxit`.

```
sage: bisection = intervalgen(f, phi, a, b)
sage: iterate(bisection)
After 22 iterations: 2.15847275559132
```

Exercise 44. Modify the function `intervalgen` so that the generator stops if one of the endpoints of the interval is a solution.

Exercise 45. Use the functions `intervalgen` and `iterate` to compute the approximation to a solution of the equation $f(x) = 0$ using nested intervals, each interval being obtained by dividing the previous one at a randomly chosen point.

The False Position Method. This method also uses the first approach: construct a sequence of nested intervals that contain a solution of the equation $f(x) = 0$. However, it employs linear interpolation of the function f for dividing each interval.

More precisely, to divide the interval $[a, b]$, we consider the segment joining the two points on the graph of f with x-coordinates a and b. As $f(a)$ and $f(b)$ have opposite signs, this segment intersects the x-axis, thereby dividing the interval $[a, b]$ into two subintervals. As in the bisection method, we identify the subinterval containing a solution by computing the value of f at the common point of the two subintervals.

The line through the points $(a, f(a))$ and $(b, f(b))$ has equation

$$y = \frac{f(b) - f(a)}{b - a}(x - a) + f(a). \tag{12.1}$$

Since $f(b) \neq f(a)$, this line intersects the x-axis in the point with x-coordinate

$$a - f(a)\frac{b - a}{f(b) - f(a)}.$$

We can therefore test this method as follows.

```
sage: phi(s, t) = s - f(s) * (t - s) / (f(t) - f(s))
sage: falsepos = intervalgen(f, phi, a, b)
sage: iterate(falsepos)
After 8 iterations: -2.89603757331027
```

It is important to note that the sequences constructed by the bisection and false position methods do not necessarily converge to the same solution. By shrinking the interval of study, we recover the positive solutions obtained via the bisection method.

```
sage: a, b = RR(pi/2), RR(pi)
sage: phi(s, t) = t - f(t) * (s - t) / (f(s) - f(t))
sage: falsepos = intervalgen(f, phi, a, b)
```

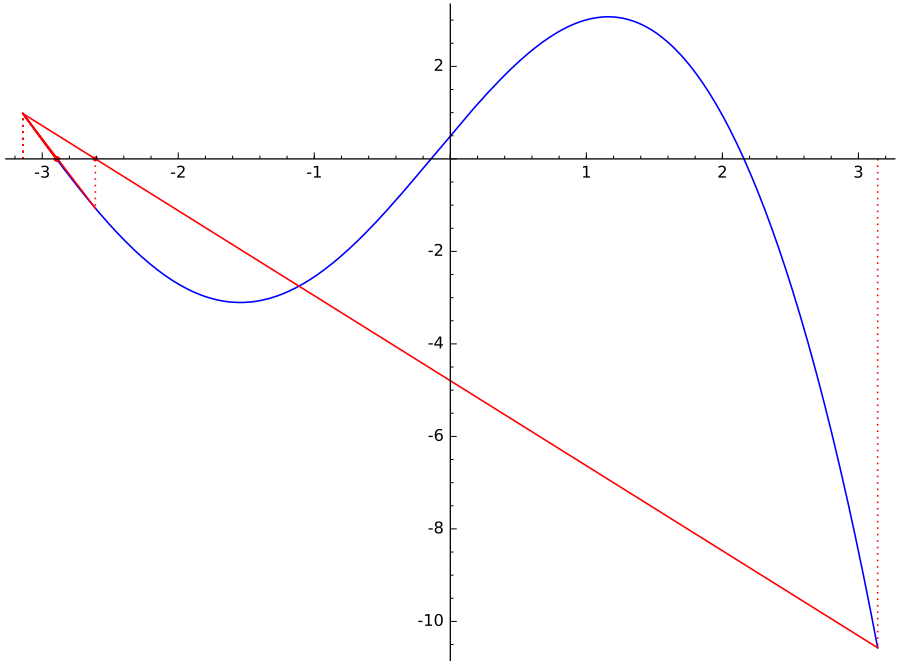


FIGURE 12.3 – The false position method on $[-\pi, \pi]$.

```
sage: phi(s, t) = (s + t) / 2
sage: bisection = intervalgen(f, phi, a, b)
sage: iterate(falsepos)
After 15 iterations: 2.15846441170219
sage: iterate(bisection)
After 20 iterations: 2.15847275559132
```

Newton's Method. Like the false position method, Newton's method uses a linear approximation of the function f. From a graphical point of view, we consider a tangent to the graph of f as approximating this graph.

We assume that f is differentiable and the derivative f' has the same sign in the interval $[a, b]$; hence f is monotone. We also assume that f changes sign in the interval $[a, b]$. The equation $f(x) = 0$ then has a unique solution in this interval; we denote it by α.

Let $u_0 \in [a, b]$. The tangent to the graph of f at the point of x-coordinate u_0 has equation

$$y = f'(u_0)(x - u_0) + f(u_0). \tag{12.2}$$

The coordinates of intersection of this line with the x-axis are

$$(u_0 - f(u_0)/f'(u_0), 0).$$

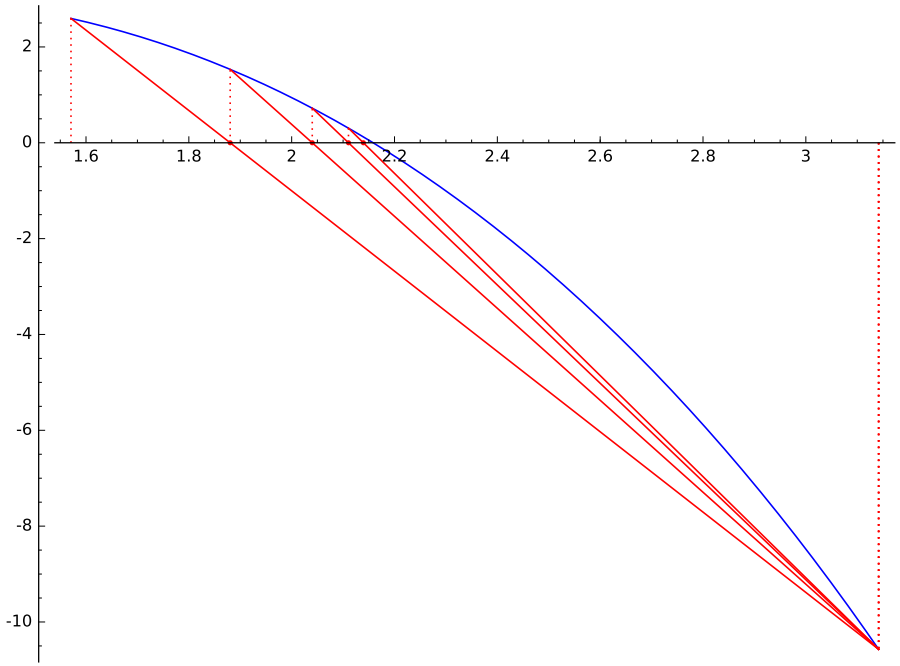


FIGURE 12.4 – The false position method on $[\pi/2, \pi]$.

We denote by φ the function $x \mapsto x - f(x)/f'(x)$. It is defined if f' does not vanish on the interval $[a, b]$. We are interested in the recursive sequence u defined by $u_{n+1} = \varphi(u_n)$.

If the sequence u converges[1], then its limit ℓ satisfies $\ell = \ell - f(\ell)/f'(\ell)$, hence $f(\ell) = 0$; the limit is therefore equal to α, the solution of the equation $f(x) = 0$.

So that our example satisfies the monotonicity hypotheses, we have to shrink the interval of study.

```
sage: f.derivative()
x |--> 4*cos(x) - 1/2*e^x
sage: a, b = RR(pi/2), RR(pi)
```

We define a Python generator `newtongen` representing the recursive sequence that we defined above. Then we define a new convergence test `checkconv` that stops the iterations if the last two computed terms are sufficiently close; of course this test does not guarantee the convergence of the sequence of approximations.

```
sage: def newtongen(f, u):
....:     while True:
....:         yield u
....:         u -= f(u) / f.derivative()(u)
```

[1] A theorem of L. Kantorovich gives a sufficient condition for the convergence of Newton's method.

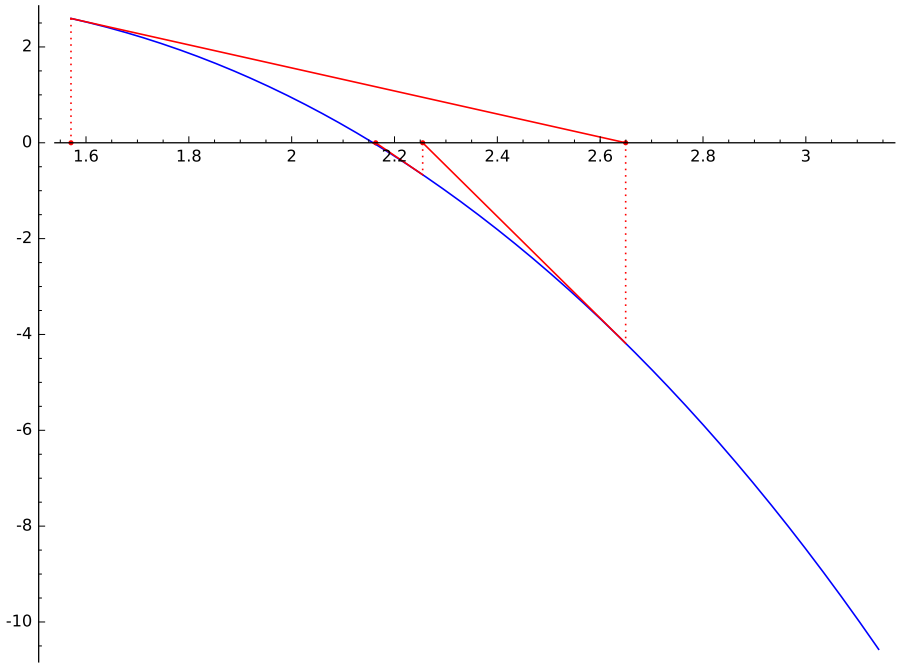


FIGURE 12.5 – Newton's method.

```
sage: def checkconv(u, v, w, prec):
....:     return abs(w - v) / abs(w) <= prec
```

We can now test Newton's method on our example.

```
sage: iterate(newtongen(f, a), check=checkconv)
After 6 iterations: 2.15846852566756
```

The Secant Method. The computation of the derivative in Newton's method can be expensive. It is possible to replace it by a linear interpolation: given two approximations to the solution, hence two points on the graph of f, if the line through these two points intersects the x-axis, we take the x-coordinate of the intersection point as the new approximation. To start the construction, or when this line is parallel to the x-axis, we use Newton's method.

This gives rise to the same iterative formula as the false position method, but applied at different points. Contrary to the false position method, the secant method does not provide an interval that contains a solution.

We define a Python generator that implements this method.

```
sage: def secantgen(f, a):
....:     yield a
....:     estimate = f.derivative()(a)
```

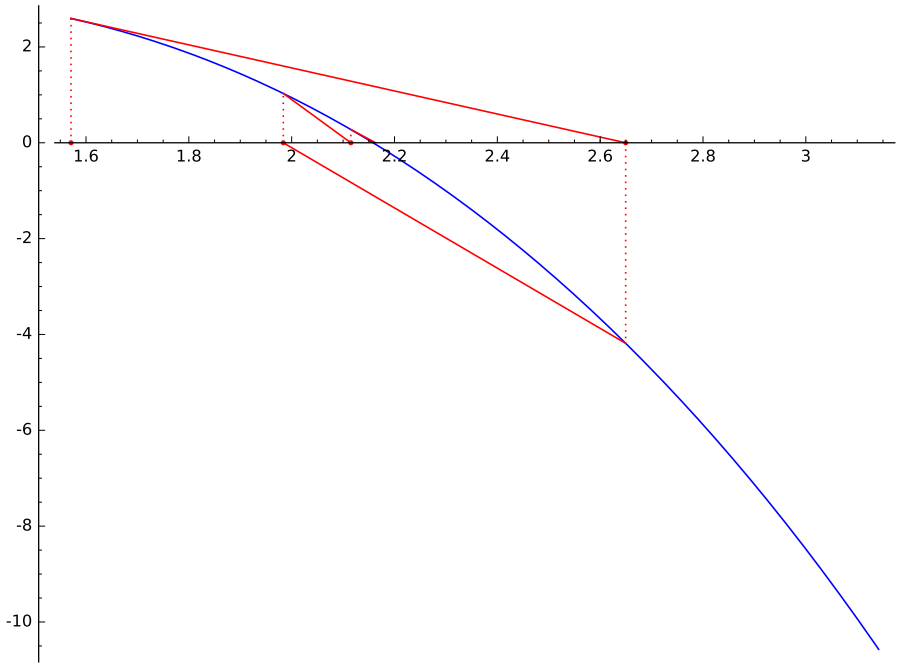


FIGURE 12.6 – The secant method.

```
....:     b = a - f(a) / estimate
....:     yield b
....:     while True:
....:         fa, fb = f(a), f(b)
....:         if fa == fb:
....:             estimate = f.derivative()(a)
....:         else:
....:             estimate = (fb - fa) / (b - a)
....:         a = b
....:         b -= fb / estimate
....:         yield b
```

We can now test the secant method on our example.

```
sage: iterate(secantgen(f, a), check=checkconv)
After 8 iterations: 2.15846852557553
```

Muller's Method. It is possible to extend the secant method by replacing f by polynomial approximations of any degree. For instance, Muller's method[2] uses quadratic approximations.

[2]This is David E. Muller, also known for inventing Reed-Muller codes, and not Jean-Michel Muller mentioned in Chapter 11.

Suppose we have constructed three approximations r, s, and t of the solution to the equation $f(x) = 0$. We consider the Lagrange interpolation polynomial defined by the three points on the graph of f with x-coordinates r, s, and t. It is a second-degree polynomial. We take as new approximation the root of this polynomial that is closest to t. The first three terms of the sequence are just taken to be a, b, and $(a + b)/2$.

We should note that the roots of the polynomials — and hence the computed approximations — can be complex numbers.

The implementation of this method in Sage is not difficult; it can be done similarly to the secant method. However, our implementation uses a data structure that is better suited to enumerating the terms of a recursive sequence.

```
sage: from collections import deque
sage: basering = PolynomialRing(CC, 'x')
sage: def quadraticgen(f, r, s):
....:     t = (r + s) / 2
....:     yield t
....:     points = deque([(r,f(r)), (s,f(s)), (t,f(t))], maxlen=3)
....:     while True:
....:         pol = basering.lagrange_polynomial(points)
....:         roots = pol.roots(ring=CC, multiplicities=False)
....:         u = min(roots, key=lambda x: abs(x - points[2][0]))
....:         points.append((u, f(u)))
....:         yield points[2][0]
```

The module `collections` of the Python standard library provides several data structures. In `quadraticgen`, the class `deque` is used to store the last approximations computed. A `deque` object stores data up to the limit `maxlen` specified at its creation; here the maximum number of stored data is the recurrence order of the sequence of approximations. Once a `deque` object reaches its maximum storage capacity, the method `deque.append()` inserts new data according to the rule "first in, first out".

Note that the iterations of this method do not require the computation of derivatives. Moreover, each iteration only requires one evaluation of the function f.

```
sage: generator = quadraticgen(f, a, b)
sage: iterate(generator, check=checkconv)
After 5 iterations: 2.15846852554764
```

Back to Polynomials. We return to the situation studied at the beginning of the chapter: compute the roots of a polynomial P with real coefficients. We assume that P is monic:

$$P = a_0 + a_1 x + \ldots + a_{d-1} x^{d-1} + x^d.$$

It is easy to check that P is the characteristic polynomial of the *companion* matrix (see §8.2.3):

$$A = \begin{pmatrix} 0 & 0 & 0 & \dots & 0 & -a_0 \\ 1 & 0 & 0 & \dots & 0 & -a_1 \\ 0 & 1 & 0 & \dots & 0 & -a_2 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 1 & -a_{d-1} \end{pmatrix}.$$

Therefore, the roots of the polynomial P are the eigenvalues of the matrix A. We can therefore apply the methods of Chapter 13.

We have seen that the method `Polynomial.roots()` takes up to three parameters, all optional: `ring`, `multiplicities`, and `algorithm`. Assume that a Sage object of the `Polynomial` class has name `p` (so that `isinstance(p, 'Polynomial')` returns `True`). The algorithm used by the command `p.roots()` then depends on the parameters `ring` and `algorithm`, as well as on the coefficient ring of the polynomial, that is `p.base_ring()`.

The algorithm checks whether arithmetic operations in `ring` and `p.base_ring()` are exact. If this is not the case, the approximations to the roots are computed via the library NumPy if `p.base_ring()` is `RDF` or `CDF`, or via the library PARI/GP otherwise (the parameter `algorithm` allows the user to override this and specify which library to use). By looking at the source code of NumPy, we see that the root approximation method used by this library computes the eigenvalues of the companion matrix.

The following command identifies which objects have exact arithmetic operations (the method `Ring.is_exact()` returning `True` in this case).

```
sage: for ring in [ZZ, QQ, QQbar, RDF, RIF, RR, AA, CDF, CIF, CC]:
....:     print("{0:50} {1}".format(ring, ring.is_exact()))
Integer Ring                                       True
Rational Field                                     True
Algebraic Field                                    True
Real Double Field                                  False
Real Interval Field with 53 bits of precision      False
Real Field with 53 bits of precision               False
Algebraic Real Field                               True
Complex Double Field                               False
Complex Interval Field with 53 bits of precision   False
Complex Field with 53 bits of precision            False
```

When the parameter `ring` is `AA` or `RIF`, while `p.base_ring()` is `ZZ`, `QQ` or `AA`, the algorithm calls the function `real_roots()` of the module `sage.rings.polynomial.real_roots`. This function converts the polynomial into the Bernstein basis then uses Casteljau's algorithm (for evaluating polynomials in Berstein basis) and Descartes' rule of signs (see §12.2.1) to isolate the roots.

When the parameter `ring` is `QQbar` or `CIF` and `p.base_ring()` is `ZZ`, `QQ`, `AA` or the Gaussian numbers $\mathbb{Q}[\sqrt{-1}]$, the algorithm passes the computation to NumPy and PARI/GP, whose results are then converted into the appropriate rings.

We refer the reader to the `Polynomial.roots()` documentation, for a comprehensive view of all situations covered by this method.

Rate of Convergence. Consider a convergent sequence of numbers u and let ℓ be its limit. We say that the rate of convergence of the sequence u is *linear* if there exists $K \in (0,1)$ such that

$$\lim_{n\to\infty} \frac{|u_{n+1} - \ell|}{|u_n - \ell|} = K.$$

The rate of convergence of the sequence u is said to be *quadratic* if there exists $K > 0$ such that

$$\lim_{n\to\infty} \frac{|u_{n+1} - \ell|}{|u_n - \ell|^2} = K.$$

Recall that Newton's method constructs a recursive sequence u defined by $u_{n+1} = \varphi(u_n)$, where φ is the function $x \mapsto x - f(x)/f'(x)$. If f is twice differentiable, Taylor's formula for φ with x in a neighbourhood of the root α is

$$\varphi(x) = \varphi(\alpha) + (x-\alpha)\varphi'(\alpha) + \frac{(x-\alpha)^2}{2}\varphi''(\alpha) + O_\alpha((x-\alpha)^3).$$

But $\varphi(\alpha) = \alpha$, $\varphi'(\alpha) = 0$ and $\varphi''(\alpha) = f''(\alpha)/f'(\alpha)$. By substituting in the previous formula and using the definition of the sequence u, we get

$$u_{n+1} - \alpha = \frac{(u_n-\alpha)^2}{2}\frac{f''(\alpha)}{f'(\alpha)} + O_\infty((u_n-\alpha)^3).$$

Therefore, when Newton's method converges, the convergence rate of the sequence is quadratic.

Acceleration of Convergence. Given a convergent sequence whose rate of convergence is linear, it is possible to construct a sequence whose rate of convergence is quadratic. The same technique, applied to Newton's method, is known as Steffensen's method.

```
sage: def steffensen(sequence):
....:     assert isinstance(sequence, GeneratorType)
....:     values = deque(maxlen=3)
....:     for i in range(3):
....:         values.append(sequence.next())
....:         yield values[i]
....:     while True:
....:         values.append(sequence.next())
....:         u, v, w = values
....:         yield u - (v - u)^2 / (w - 2 * v + u)
```

```
sage: g(x) = sin(x^2 - 2) * (x^2 - 2)
sage: sequence = newtongen(g, RR(0.7))
sage: accelseq = steffensen(newtongen(g, RR(0.7)))
sage: iterate(sequence, check=checkconv)
After 17 iterations: 1.41422192763287
```

Solution of non-linear equations	
Approximate roots of a polynomial	`Polynomial.roots()`
Exact roots (not guaranteed to give all of them)	`Expression.roots()`
Approximate real roots	`real_roots()`
Approximate roots via Brent's method	`Expression.find_root()`

TABLE 12.1 – Summary of commands described in this chapter.

```
sage: iterate(accelseq, check=checkconv)
After 10 iterations: 1.41421041980166
```

Note that the rate of convergence is an asymptotic notion: it says nothing about the error $|u_n - \ell|$ for a given n.

```
sage: sequence = newtongen(f, RR(a))
sage: accelseq = steffensen(newtongen(f, RR(a)))
sage: iterate(sequence, check=checkconv)
After 6 iterations: 2.15846852566756
sage: iterate(accelseq, check=checkconv)
After 7 iterations: 2.15846852554764
```

The Method `Expression.find_root()`. We now consider the most general situation: the computation of an approximate solution of the equation $f(x) = 0$. In Sage, this is done using the method `Expression.find_root()`.

The parameters of the method `Expression.find_root()` allow us to specify an interval where the root should be found, the precision of the computation, or the number of iterations. The parameter `full_output` gives access to additional information about the computation, in particular the number of iterations and the number of evaluations of the function.

```
sage: result = (f == 0).find_root(a, b, full_output=True)
sage: result[0], result[1].iterations
(2.1584685255476415, 9)
```

In fact, the method `Expression.find_root()` does not implement an algorithm for finding the solutions of equations: the computation is delegated to the module SciPy. The SciPy functionality used by Sage for solving equations implements Brent's method, which combines three of the methods we discussed above: the bisection method, the secant method, and quadratic interpolation. The two first approximate values are the endpoints of the interval where we are looking for a solution of the equation. The next approximation is obtained by linear interpolation, as in the secant method. In the following iterations, the function is approximated by quadratic interpolation; the x-coordinate of the intersection point of the interpolating curve and the x-axis is the new approximation, unless this x-coordinate is sandwiched between the last two computed approximations, in which case we continue with the bisection method.

The SciPy library does not offer arbitrary precision computations (unless we are satisfied to compute with integers); in fact, the source code of the method `Expression.find_root()` starts by converting the bounds into double precision numbers. On the other hand, all illustrations of solution methods we have given in this chapter work in arbitrary precision, and even symbolically.

```
sage: a, b = pi/2, pi
sage: generator = newtongen(f, a)
sage: generator.next(); generator.next()
1/2*pi
1/2*pi - (e^(1/2*pi) - 10)*e^(-1/2*pi)
```

Exercise 46. Write a generator for Brent's method that works with arbitrary precision.

13 Numerical Linear Algebra

We consider here the numerical side of linear algebra, the symbolic side being described in Chapter 8. The linear algebra numerical analysis and methods are discussed in [TBI97, Sch02]. The book of Golub and Van Loan [GVL12] is a major reference in this domain.

Numerical linear algebra plays a key role in *scientific computing* including the numerical solution of ordinary and partial differential equations, optimisation problems, and signal processing problems.

The numerical resolution of several of these problems, even linear ones, relies on algorithms made from nested loops; at the bottom of these loops, in most cases a linear system is solved. Non-linear algebraic systems are often solved using Newton's method: here again, we have to deal with linear systems. Efficiency and robustness of numerical linear algebra methods are thus crucial.

This chapter is split into three sections: in the first, we discuss the different sources of inexactness of numerical linear algebra computations; the second section (§13.2) discusses, without trying to be exhaustive, the more classical problems (linear system resolution, eigenvalue computation, least squares); in the third section (§13.3) we consider the case of sparse matrices. This last section is not only a guide for the user, but also an introduction to methods which are part of an active research domain.

13.1 Inexact Computations

We consider here classical linear algebra problems (linear system resolution, eigenvalue and eigenvector computation, etc.) which are solved using floating-point (i.e., inexact) computations. The different kinds of floating-point numbers available in Sage are detailed in Chapter 12.

Consider for example the system $Ax = b$ to solve, where A is a matrix with real coefficients. How big error δx do we get if we modify A by δA or b by δb? We give some partial answers in this chapter.

13.1.1 Matrix Norms and Condition Number

Let $A \in \mathbb{R}^{n\times n}$ (or $\mathbb{C}^{n\times n}$). We define on $\mathbb{R}^n$ (or $\mathbb{C}^n$) a norm $\|x\|$, for example the norm $\|x\|_\infty = \max |x_i|$ or $\|x\|_1 = \sum_{i=1}^n |x_i|$, or even the Euclidean norm $\|x\|_2 = (\sum_{i=1}^n |x_i|^2)^{1/2}$; then the quantity

$$\|A\| = \max_{\|x\|=1} \|A\ x\|$$

defines a norm on the set of $n \times n$ matrices. We say it is an *induced norm* with respect to the norm defined on $\mathbb{R}^n$ (or $\mathbb{C}^n$). The *condition number* of a nonsingular matrix A is defined by $\kappa(A) = \|A^{-1}\| \cdot \|A\|$. The fundamental result is that, if A is slightly modified by δA and b by δb, then the solution x of the linear system $Ax = b$ is modified by δx satisfying

$$\frac{\|\delta x\|}{\|x\|} \le \frac{\kappa(A)}{1 - \kappa(A)\|\delta A\|/\|A\|} \left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|b\|} \right).$$

The norms $\|\cdot\|_\infty$ and $\|\cdot\|_1$ are easy to compute: $\|A\|_\infty = \max_{1\le i\le n}(\sum_{j=1}^n |a_{ij}|)$ and $\|A\|_1 = \max_{1\le j\le n}(\sum_{i=1}^n |a_{ij}|)$. On the contrary, the norm $\|\cdot\|_2$ is more difficult to compute, since $\|A\|_2 = \sqrt{\rho(A^t A)}$, the spectral radius ρ of a matrix A being the largest modulus of its eigenvalues.

The Frobenius norm is defined by

$$\|A\|_F = \left(\sum_{i=1}^n \sum_{j=1}^n |a_{ij}|^2 \right)^{1/2}.$$

We easily check that $\|A\|_F^2 = \mathrm{trace}(A^t A)$. Contrary to the above norms, it is not an induced norm.

Computing Matrix Norms in Sage. The matrices in Sage have a `norm(p)` method. According to the value of the argument p it computes:

$p = 1$:	$\|A\|_1$,	$p = 2$:	$\|A\|_2$,
$p =$ `Infinity` :	$\|A\|_\infty$,	$p =$ `'frob'` :	$\|A\|_F$.

This method only works when the matrix coefficients can be converted into complex floating-point numbers `CDF`. Please note that we write `A.norm(Infinity)` but `A.norm('frob')`. With `A.norm()` we obtain the default norm $\|A\|_2$.

Errors and Condition Number: an Example, with Exact and Approximate Computations. Let us use the capability of Sage to perform exact computations when the coefficients are rational. We consider the Hilbert matrix

$$a_{ij} = 1/(i+j-1),\ i,j = 1,\ldots,n.$$

The following program computes[1] exactly the condition number of Hilbert matrices, using the norm $\|\cdot\|_\infty$:

```
sage: def cond_hilbert(n):
....:     A = matrix(QQ, [[1/(i+j-1) for j in [1..n]] for i in [1..n]])
....:     return A.norm(Infinity) * (A^-1).norm(Infinity)
```

Here are the results for a few n:

n	condition number
2	27
4	28375
8	33872791095
16	5.06277e+22
32	1.35711e+47

We see the extremely fast growth of the condition number with respect to n. We can prove that $\kappa(A) \simeq e^{7n/2}$, which indeed grows very quickly. Still computing in the rational field, we take $x = [1,\cdots,1]^t$, $y = Ax$ and we solve the system $\widetilde{A}s = y$ where $\widetilde{A}$ is a slight modification of the matrix A; then we compare the solution s and $x = A^{-1}y$, thus measuring the error introduced by the modification of A:

```
sage: def diff_hilbert(n):
....:     x = vector(QQ,[1 for i in range(0,n)])
....:     A = matrix(QQ, [[1/(i+j-1) for j in [1..n]] for i in [1..n]])
....:     y = A*x
....:     A[n-1,n-1] = (1/(2*n-1))*(1+1/(10^5))   # modifies the matrix
....:     s = A\y
....:     return max(abs(float(s[i]-x[i])) for i in range(0,n))
```

We obtain:

n	error (`diff_hilbert`)
2	3.99984e-05
4	5.97610e-3
8	4.80536
16	67.9885
32	20034.3

The error very rapidly becomes too large with respect to $x_i = 1$.

Let us now perform the same computation with floating-point coefficients. We no longer modify the matrix A, but the floating-point arithmetic will perform

[1] In Sage, there exists a `condition()` method which returns the condition number for various norms, but it exists for matrices with coefficients in `RDF` or `CDF`.

small rounding errors (thus, the matrix A is actually *not* exactly the Hilbert matrix any more). We perform again the above computation: the vector y being first computed by rounding the exact values in RDF, we try to recover x by solving the linear system $Ax = y$:

```
sage: def hilbert_diff(n):
....:     j = var("j")
....:     f = lambda i: sum(1/(i+j-1),j,1,n)
....:     y = vector(RDF, [f(i+1) for i in range(0,n)])
....:     A = matrix(RDF, [[1/(i+j-1) for i in [1..n]] for j in [1..n]])
....:     x = A.solve_right(y)
....:     return max(abs(x[i]-1.0) for i in range(0,n))
```

We also compute the condition number $\kappa(A)$. According to n we obtain:

n	error (`hilbert_diff`)	$\kappa(A)$	$\kappa(A)$ /error
2	6.66134e-16	27.0	2.46716e-17
4	2.38586e-13	28375.0	8.40835e-18
8	3.45957e-07	3.38723e+10	1.02134e-17
16	73.9841	1.00834e+19	7.33725e-18
32	76.6078	1.22870e+19	1.91195e-18

We see for example that for $n = 16$, the error made on the solution (with the infinite norm) is so large that all digits of the result are wrong (with the particular choice of x we have made, $x_i = 1$, the absolute and relative errors coincide).

Remarks. But why then compute with floating-point numbers? The performance is not always the only reason since efficient linear algebra libraries with rational arithmetic exist (for example Linbox, used by Sage); these libraries implement algorithms which are slower than their floating-point equivalent, but could be used for the resolution of moderate size linear systems. However, a second source of inexactness comes from the fact that, in real applications, the coefficients are usually not known (or measured) exactly. For example, solving a non-linear system of equations using Newton's method will naturally involve inexact terms.

Ill-conditioned linear systems (if we except extreme cases like Hilbert matrix) are more the rule than the exception: we often encounter (in physics, chemistry, biology, etc.) systems of ordinary differential equations of the form $du/dt = F(u)$ where the Jacobian matrix $DF(u)$, defined by partial derivatives $\partial F_i(u)/\partial u_j$, defines an ill-conditioned system. The eigenvalues span a very large range, which yields a bad condition number of $DF(u)$; this corresponds to the fact that the system models phenomena of different time scales. Unfortunately, in practice, we have to solve linear systems whose matrix is $DF(u)$.

All the computations (matrix decomposition, computation of eigenelements, convergence of iterative methods) depend on an appropriately defined condition number. We therefore should keep this notion in mind as soon as we perform linear algebra computations with floating-point numbers.

13.2 Dense Matrices

13.2.1 Solving Linear Systems

Methods to Avoid. Using Cramer's formula should (almost) always be avoided. A recurrence reasoning shows that computing the determinant of an $n \times n$ matrix using Cramer's formula requires on the order of $n!$ multiplications and additions. To solve a system of size n, we have to compute $n + 1$ determinants. Consider $n = 20$:

```
sage: n = 20; cost = (n+1)*factorial(n); cost
51090942171709440000
```

we obtain the huge value of 51 090 942 171 709 440 000 multiplies. Assuming our computer performs $3 \cdot 10^9$ multiplications per second, let us find how long the computation would last:

```
sage: v = 3*10^9
sage: print("%3.3f" % float(cost/v/3600/24/365))
540.028
```

The computation would thus require about 540 years! Of course, we can use Cramer's formula to solve a 2×2 system, but not much beyond! All methods used in practice have in common a polynomial cost in the dimension, i.e., of order n^p, with p small ($p = 3$ in general).

Practical Methods. The solution of linear systems $Ax = b$ is in most cases based on a factorisation of the matrix A into a product of two matrices $A = M_1 M_2$, where M_1 and M_2 correspond to "easy" linear systems. To solve $Ax = b$, we thus first solve $M_1 y = b$, then $M_2 x = y$.

For example M_1 and M_2 can be triangular matrices; in this case, once the factorisation is performed, we have to solve two triangular linear systems. The factorisation is much more expensive than solving the triangular linear systems (for example $O(n^3)$ for the LU factorisation against $O(n^2)$ for the triangular linear systems). When several systems with the same matrix have to be solved, we should therefore perform the matrix decomposition only once. Of course, we *never* invert a matrix to solve a linear system, since the inversion requires the matrix factorisation, followed by solving n systems instead of only one.

13.2.2 Direct Resolution

The simplest way to solve a linear system is illustrated by

```
sage: A = matrix(RDF, [[-1,2],[3,4]])
sage: b = vector(RDF, [2,3])
sage: x = A\b; x
(-0.20000000000000018, 0.9000000000000001)
```

Within Sage, matrices have a method `solve_right` to solve linear systems (which is based on the LU decomposition); this method is used above. We could also write:

```
sage: x = A.solve_right(b)
```

The `x = A\b` syntax is similar to what can be found in the Matlab, Octave and Scilab systems.

13.2.3 The LU Decomposition

```
sage: A = matrix(RDF, [[-1,2],[3,4]])
sage: P, L, U = A.LU()
```

This method gives the L and U factors, and the permutation matrix P, such that $A = PLU$ (or equivalently, $P^tA = LU$). The matrix L is lower triangular, with unit diagonal, and U is upper triangular. The matrix P depends on the pivot choices. For this, the strategy described in §8.2.1 consists, at step k, in finding an invertible coefficient a_{ik} in column k, and to use it as pivot. But here, this strategy must be improved, as we compute with floating-point numbers. To demonstrate this, let us consider the following system:

$$\begin{cases} \varepsilon x + y &= 1, \\ x + y &= 2. \end{cases}$$

If we use the coefficient ε of x in the first equation as pivot, we obtain $y = (1-2\varepsilon)/(1-\varepsilon)$ and $x = 1/(1-\varepsilon)$. Let us choose a small number for ε:

```
sage: eps = 1e-16
sage: y = (1-2*eps)/(1-eps)
sage: x = (1-y)/eps
sage: x, y
(1.11022302462516, 1.00000000000000)
```

The solution is very inaccurate! This is a consequence of the fact that, in the set of floating-point numbers we use:

```
sage: 1. + eps == 1.
True
```

as `RR(1).ulp()`$> \varepsilon$ (see §11.3.1). Now, we choose as pivot the coefficient of x in the second equation (that is to say we permute the equations), we get:

$$\begin{cases} x + y &= 2, \\ (1-\varepsilon)y &= 1 - 2\varepsilon. \end{cases}$$

We obtain:

```
sage: y = (1-2*eps)/(1-eps)
sage: x = 2-y
sage: x, y
(1.00000000000000, 1.00000000000000)
```

which is much more acceptable.

To obtain the PLU factorisation of A, the partial pivoting by column algorithm is used: at the first step we choose in the first column the coefficient a_{i1} of maximum absolute value and exchange row i and row 1; then, we fill the first column (except the first element) with zeros using the Gaussian elimination process. The application to the next steps is obvious.

Please note that Sage keeps in memory the factorisation of A: the `A.LU_valid()` command answers `True` if and only if the LU factorisation has already been computed. Moreover, the `A.solve_right(b)` command will only compute the factorisation if required, i.e., if it has not been computed before, or if the matrix A has changed.

Example. Let us create a random matrix of size 1000, and two size-1000 vectors:

```
sage: A = random_matrix(RDF, 1000)
sage: b = vector(RDF, range(1000))
sage: c = vector(RDF, 2*range(500))
```

Let us first solve the system $Ax = b$:

```
sage: %time x = A.solve_right(b)
CPU times: user 132 ms, sys: 4 ms, total: 136 ms
Wall time: 140 ms
```

and now let us solve $Ay = c$:

```
sage: %time y = A.solve_right(c)
CPU times: user 68 ms, sys: 0 ns, total: 68 ms
Wall time: 72.8 ms
```

The second resolution is faster, because it used the LU factorisation computed in the first one.

13.2.4 The Cholesky Decomposition

A symmetric matrix A is said to be *positive definite* if for every non-zero vector x, $x^tAx > 0$. For every symmetric positive definite matrix, there exists a lower triangular matrix C such that $A = CC^t$. This factorisation is called Cholesky decomposition[2]. Within Sage, it is obtained by the `cholesky()` method. In the following example, we construct a matrix A which is almost surely positive definite:

```
sage: m = random_matrix(RDF, 10)
sage: A = transpose(m)*m
sage: C = A.cholesky()
```

[2]Cholesky (1875-1918) did his studies at the French École Polytechnique and was artillery officer; his method was invented to solve geodesic problems.

It should be noted that it is not possible to easily know (without performing the Cholesky decomposition) whether a symmetric matrix is positive definite; if we apply the `cholesky` method to a matrix that is not positive definite, the decomposition will fail and an exception `ValueError` will be raised.

To solve a system $Ax = b$ with the Cholesky decomposition, we proceed as with the LU decomposition. Once the Cholesky decomposition is computed, we call `A.solve_right(b)`. Here again, the decomposition is not recomputed.

Why use the Cholesky decomposition instead of the LU decomposition to solve linear systems with symmetric positive definite matrix? First, the memory necessary to store the factors is halved, due to symmetry, but the Cholesky method is especially interesting for its number of operations: indeed, for a size-n matrix, the Cholesky factorisation costs n square roots, $n(n-1)/2$ divisions, $(n^3-n)/6$ additions and as many multiplications. As a comparison, the LU factorisation costs $n(n-1)/2$ divisions as well, but $(n^3-n)/3$ additions and as many multiplications.

13.2.5 The QR Decomposition

Let $A \in \mathbb{R}^{n\times m}$, with $n \geq m$. We want to find two matrices Q and R such that $A = QR$ where $Q \in \mathbb{R}^{n\times n}$ is orthogonal ($Q^t \cdot Q = I$) and $R \in \mathbb{R}^{n\times m}$ is upper triangular. Of course, once such a decomposition is computed, we can use it to solve linear systems if the matrix A is square and invertible. However, as we will see, the QR decomposition is especially interesting to solve least squares problems, and to compute eigenvalues. We should note that A is not necessarily square. The QR decomposition always exists. Example:

```
sage: A = random_matrix(RDF,6,5)
sage: Q, R = A.QR()
```

Exercise 47 (Perturbing a linear system)**.** Let A be an invertible square matrix, and assume we have computed a decomposition of A (LU, QR, Cholesky, ...). Let u and v be two vectors. We consider the matrix $B = A + uv^t$, and we assume that $1 + v^t A^{-1} u \neq 0$. How to cheaply solve the $Bx = f$ system (i.e., without a factorisation of B)?

Hint: We will use the Sherman and Morrison formula (that we will either prove or assume):

$$(A + uv^t)^{-1} = A^{-1} - \frac{A^{-1}uv^tA^{-1}}{1 + v^tA^{-1}u}.$$

13.2.6 Singular Value Decomposition

Let A be an $n \times m$ matrix with real coefficients. Then two orthogonal matrices $U \in \mathbb{R}^{n\times n}$ and $V \in \mathbb{R}^{m\times m}$ exist, such that

$$U^tAV = \Sigma = \mathrm{diag}(\sigma_1, \sigma_2, \cdots, \sigma_p),$$

where $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_p \geq 0$ (with $p = \min(m, n)$). The numbers $\sigma_1, \ldots, \sigma_p$ are the *singular values* of A.

The matrices U and V are orthogonal ($U \cdot U^t = I$ and $V \cdot V^t = I$) and as a consequence:

$$A = U\Sigma V^t.$$

Example (the computations are inexact due to rounding errors):

```
sage: A = matrix(RDF, [[1,3,2],[1,2,3],[0,5,2],[1,1,1]])
sage: U, Sig, V = A.SVD()
sage: A1 = A - U*Sig*transpose(V); A1
[ 2.220446049250313e-16                  0.0                   0.0]
[3.3306690738754696e-16 -4.440892098500626e-16 -4.440892098500626e-16]
[-9.298117831235686e-16 1.7763568394002505e-15 -4.440892098500626e-16]
[ 4.440892098500626e-16 -8.881784197001252e-16 -4.440892098500626e-16]
```

We can show that the singular values of a matrix A are the square roots of the eigenvalues of A^tA. It is easy to check that, for a square matrix of order n, the Euclidean norm $\|A\|_2$ equals σ_1 and that, if the matrix is non-singular, the condition number of A in the Euclidean norm equals σ_1/σ_n. The rank of A is the integer r defined by:

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > \sigma_{r+1} = \cdots = \sigma_p = 0.$$

13.2.7 Application to Least Squares

We want to solve the over-determined system $Ax = b$ where A is a rectangular matrix with real coefficients, having n rows and m columns with $n > m$. Clearly, this system has no solution in general. We thus consider the minimisation problem of the square Euclidean norm $\|\cdot\|_2$ of the residue:

$$\min_x \|Ax - b\|_2^2.$$

The rank of the matrix A might even be less than m.

Solving the Normal Equations. It is straightforward that the solution satisfies:

$$A^tAx = A^tb.$$

Assuming A of full rank m, we can thus try to form the matrix A^tA and solve the system $A^tAx = A^tb$, for example by computing the Cholesky decomposition of A^tA. This is precisely the origin of Cholesky's method. What is the condition number of A^tA? This is what will influence the accuracy of the results. The singular values of A^tA, which is of dimension $m \times m$, are the squares of the singular values of A; thus the condition number in Euclidean norm is σ_1^2/σ_m^2, the square of the condition number of A, which can be large. We thus prefer the methods based either on the QR decomposition of A, or on its singular value decomposition.

Nevertheless, this method is useful for small systems, with a condition number which is not too bad. Here is the corresponding code:

```
sage: A = matrix(RDF, [[1,3,2],[1,4,2],[0,5,2],[1,3,2]])
sage: b = vector(RDF, [1,2,3,4])
sage: Z = transpose(A)*A
sage: C = Z.cholesky()
sage: R = transpose(A)*b
sage: Z.solve_right(R)
(-1.5000000000000135, -0.5000000000000085, 2.7500000000000213)
```

We should note that Cholesky's decomposition is *cached*, and is used by `Z.solve_right(R)`, without being recomputed.

With the QR decomposition. Assume A of full rank[3]; we consider the QR decomposition of A. Then

$$\|Ax - b\|_2^2 = \|QRx - b\|_2^2 = \|Rx - Q^t b\|_2^2.$$

We have: $R = \begin{bmatrix} R_1 \\ 0 \end{bmatrix}$ where R_1 is $m \times m$ upper triangular and $Q^t b = \begin{bmatrix} c \\ d \end{bmatrix}$ with c of size m. Thus $\|Ax - b\|_2^2 = \|R_1 x - c\|_2^2 + \|d\|_2^2$, and the minimum is obtained by solving the triangular system $R_1 x = c$:

```
sage: A = matrix(RDF, [[1,3,2],[1,4,2],[0,5,2],[1,3,2]])
sage: b = vector(RDF, [1,2,3,4])
sage: Q, R = A.QR()
sage: R1 = R[0:3,0:3]
sage: b1 = transpose(Q)*b
sage: c = b1[0:3]
sage: R1.solve_right(c)
(-1.499999999999999, -0.49999999999999867, 2.749999999999997)
```

Let us compute the condition number of $A^t A$ in infinite norm:

```
sage: Z = A.transpose()*A
sage: Z.norm(Infinity)*(Z^-1).norm(Infinity)
1992.3750000000168
```

The QR method and the method of normal equations give results that agree to within the roundoff level times $\kappa(A^t A)$.

With the Singular Value Decomposition. The singular value decomposition $A = U\Sigma V^t$ also allows us to compute the solution; moreover, we can use it when A is not of full rank. If A is not the zero matrix, Σ has $0 < r \leq m$ positive coefficients σ_i (assumed in decreasing order). We then have:

$$\|Ax - b\|_2^2 = \|U^t A V V^t x - U^t b\|_2^2.$$

Writing $\lambda = V^t x$, and u_i for the columns of U, we have:

$$\|Ax - b\|_2^2 = \sum_{i=1}^{p} (\sigma_i \lambda_i - u_i^t b)^2 + \sum_{i=p+1}^{m} (u_i^t b)^2.$$

[3]We can avoid that condition with a QR method with pivots.

The minimum is thus attained by taking $\lambda_i = (u_i^t b)/\sigma_i$ for $1 \leq i \leq p$ if $\lambda_i \neq 0$, and $\lambda_i = 0$ otherwise. We finally obtain the solution $x = V\lambda$.

Here is the corresponding Sage program:

```
sage: A = matrix(RDF, [[1,3,2],[1,3,2],[0,5,2],[1,3,2]])
sage: B = vector(RDF, [1,2,3,4])
sage: U, Sig, V = A.SVD()
sage: m = A.ncols()
sage: x = vector(RDF, [0]*m)
sage: lamb = vector(RDF, [0]*m)
sage: for i in range(0,m):
....:     s = Sig[i,i]
....:     if s!=0.0:
....:         lamb[i]=U.column(i)*B/s
sage: x = V*lamb; x
(0.2370370370370367, 0.4518518518518521, 0.3703703703703702)
```

Please note that we have chosen here a matrix of rank 2 (we can check with the `A.rank()` command) and thus not of full rank; there are several solutions to the least squares problem, and the above mathematical analysis shows that x is the solution with the smallest Euclidean norm.

Let us look at the singular values:

```
sage: m = 3; [ Sig[i,i] for i in range(0,m) ]
[8.309316833256451, 1.3983038884881154, 0.0]
```

The rank of the matrix A being 2, the third singular value is necessarily 0.

Example. Among the marvelous applications of the singular value decomposition (SVD), here is a problem (known as the *orthogonal Procrustes problem*) which is hard to solve by another method: let A and $B \in \mathbb{R}^{n\times m}$ be the results of an experiment repeated twice. We wonder if B can be *twisted* to A, i.e., if there exists an orthogonal matrix Q such that $A = BQ$. Naturally, the measures A and B contain some noise, whence the problem has no solution in general. We therefore consider the corresponding least squares problem; we are looking for the orthogonal matrix Q which minimises the square of the Frobenius norm:

$$\|A - BQ\|_F^2.$$

We remember that $\|A\|_F^2 = \operatorname{trace}(A^t A)$. Then

$$\|A - BQ\|_F^2 = \operatorname{trace}(A^t A) + \operatorname{trace}(B^t B) - 2\operatorname{trace}(Q^t B^t A) \geq 0,$$

and we have to maximise $\operatorname{trace}(Q^t B^t A)$. We then compute the SVD of $B^t A$: we have $U^t(B^t A)V = \Sigma$. Let us denote by σ_i the singular values, and $Z = V^t Q^t U$. This matrix is orthogonal, and thus all its coefficients are less or equal to 1 in absolute value. Then:

$$\operatorname{trace}(Q^t B^t A) = \operatorname{trace}(Q^t U \Sigma V^t) = \operatorname{trace}(Z\Sigma) = \sum_{i=1}^{m} Z_{ii}\sigma_i \leq \sum_{i=1}^{m} \sigma_i,$$

and the maximum is obtained for $Q = UV^t$.

```
sage: A = matrix(RDF, [[1,2],[3,4],[5,6],[7,8]])
```

B is obtained by adding a random noise to A, and then applying a rotation R:

```
sage: th = 0.7
sage: R = matrix(RDF, [[cos(th),sin(th)],[-sin(th),cos(th)]])
sage: B = (A + 0.1*random_matrix(RDF,4,2)) * transpose(R)
```

```
sage: C = transpose(B)*A
sage: U, Sigma, V = C.SVD()
sage: Q = U*transpose(V)
```

The random noise is small, and Q is close to R as expected:

```
sage: Q
[ 0.7612151656410958  0.6484993998439783]
[-0.6484993998439782  0.7612151656410955]
sage: R
[0.7648421872844885  0.644217687237691]
[-0.644217687237691 0.7648421872844885]
```

Exercise 48 (Square root of a symmetric semi-definite positive matrix)**.** Let A be a symmetric semi-definite positive matrix (i.e., which satisfies $x^tAx \geq 0$ for any vector x). Show that we can compute a matrix X, also symmetric semi-definite positive, such that $X^2 = A$.

13.2.8 Eigenvalues, Eigenvectors

So far, we have used some direct methods (LU, QR, or Cholesky decompositions), which give a solution in a finite number of operations (the four basic arithmetic operations, and the square root for the Cholesky decomposition). This *cannot hold* for the computation of eigenvalues: indeed (cf. page 294), we can associate to every polynomial a matrix whose eigenvalues are the roots of the polynomial, and we know there is no explicit formula for the roots of a polynomial of degree 5 or more, a formula that a direct method would yield. Also, constructing the characteristic polynomial to compute its roots would be extremely costly (the Faddeev-Le Verrier algorithm allows us to compute the characteristic polynomial of a size-n matrix in $O(n^4)$ operations, which is still considered too expensive). The numerical methods used to compute eigenvalues and eigenvectors are all iterative. Recall also that the singular values of a matrix A are the square roots of the eigenvalues of A^tA: consequently there is no direct method to compute the singular value decomposition.

We will thus build sequences converging towards the eigenvalues (and eigenvectors), and stop the iterations when close enough to the solution[4].

[4]In the examples below, we choose a fixed number of iterations, for simplicity.

The Power Method. Let $A \in \mathbb{C}^{n\times n}$. We choose any norm $\|\cdot\|$ on $\mathbb{C}^n$. Starting from x_0, we consider the vector sequence x_k defined by:

$$x_{k+1} = \frac{Ax_k}{\|Ax_k\|}.$$

If the eigenvalues satisfy $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|$, then the sequence x_k converges towards an eigenvector associated to the dominant eigenvalue λ_1. Moreover, the sequence $\nu_k = x_{k+1}^* x_k$ converges towards $|\lambda_1|$. (The assumption that the eigenvalues have different absolute values can be relaxed.)

```
sage: n = 10
sage: A = random_matrix(RDF,n)
sage: A = A*transpose(A) + diagonal_matrix([-RDF(i) for i in [1..n]])
sage: # A satisfies (almost surely) the hypotheses.
sage: X = vector(RDF, [1 for i in range(0,n)])
sage: lam_old = 0
sage: for i in range(0,100):
....:     Z = A*X
....:     X = Z/Z.norm()
....:     lam =  X*A*X
....:     s = abs(lam - lam_old)
....:     print("{i} s={s} lambda={lam}".format(i=i, s=s, lam=lam))
....:     lam_old = lam
....:     if s < 1.e-10:
....:         break
0 s=2.0567754429 lambda=-2.0567754429
1 s=1.25099951088 lambda=-3.30777495378
2 s=1.09226722941 lambda=-4.40004218319
3 s=0.908251200751 lambda=-5.30829338394
4 s=0.721759096603 lambda=-6.03005248054
...
96 s=1.69451061005e-06 lambda=-8.01819929979
97 s=1.54755954895e-06 lambda=-8.01820084735
98 s=1.41335122805e-06 lambda=-8.0182022607
99 s=1.29078087419e-06 lambda=-8.01820355148
```

Now, let us compute:

```
sage: A.eigenvalues()
[6.587670892594506,
 2.693595372897774,
 -8.018217143995244,
 -7.662621462625094,
 -6.107162561729528,
 1.2180921545902392,
 0.11456793007905457,
 -0.30200998716052146,
```

```
-2.3272481190401657,
-3.52779132563669]
```

We have indeed determined the dominant eigenvalue.

This method might look of little interest, but it will appear again for sparse matrices. It also introduces what follows, which is very useful.

The Inverse Power Method. We assume a known approximation μ of an eigenvalue λ_j (with $\mu, \lambda_j \in \mathbb{C}$). How can we determine an eigenvector associated to λ_j?

We assume that $\forall k \neq j,\ 0 < |\mu - \lambda_j| < |\mu - \lambda_k|$, and thus, λ_j is a simple eigenvalue. We then consider $(A - \mu I)^{-1}$, whose largest eigenvalue is $(\lambda_j - \mu)^{-1}$, and we apply the power method to this matrix.

Let us take for example:

```
sage: A = matrix(RDF, [[1,3,2],[1,2,3],[0,5,2]])
```

By calling the method `A.eigenvalues()`, we find the eigenvalues (rounded to 5 significant digits) `6.3929, 0.56052, -1.9535`. We will search the eigenvector associated to the second eigenvalue, starting from an approximation:

```
sage: A = matrix(RDF,[[1,3,2],[1,2,3],[0,5,2]])
sage: mu = 0.56
sage: AT = A - mu*identity_matrix(RDF,3)
sage: X = vector(RDF,[1 for i in range(0,A.nrows())])
sage: lam_old = 0
sage: for i in range(1,1000):
....:     Z = AT.solve_right(X)
....:     X = Z/Z.norm()
....:     lam = X.dot_product(A*X)
....:     s = abs(lam - lam_old)
....:     print("{i} s={s} lambda={lam}".format(i=i, s=s, lam=lam))
....:     lam_old = lam
....:     if s<1.e-10:
....:         break
1 s=0.56423627407 lambda=0.56423627407
2 s=0.00371649959176 lambda=0.560519774478
3 s=2.9833340176e-07 lambda=0.560519476145
4 s=3.30288019157e-11 lambda=0.560519476112
sage: X
(0.9276845629439007, 0.10329475725387141, -0.3587917847435305)
```

Let us verify that we have calculated an approximation of the selected eigenvalue and of an associated eigenvector:

```
sage: A*X-lam*X
(2.886579864025407e-15, 1.672273430841642e-15, 8.326672684688674e-15)
```

Several remarks can be made:

- we do not compute the inverse of the matrix $A - \mu I$, but we use its LU factorisation, which is computed only once (by the first `solve_right` call);
- we use the iterations to improve an estimation of the selected eigenvalue;
- the convergence is very fast; we can indeed show that (modulo the above hypotheses, and the choice of an initial vector non-orthogonal to the eigenvector q_j associated to λ_j), we have, for the iterates $x^{(i)}$ and $\lambda^{(i)}$:

$$\|x^{(i)} - q_j\| = O\left(\left|\frac{\mu - \lambda_j}{\mu - \lambda_K}\right|^i\right)$$

and

$$\|\lambda^{(i)} - \lambda_j\| = O\left(\left|\frac{\mu - \lambda_j}{\mu - \lambda_K}\right|^{2i}\right),$$

where λ_K is the second closest eigenvalue to μ;

- the condition number of $A - \mu I$ (bounded from below by the ratio between the largest and smallest eigenvalues of $A - \mu I$) is large thus bad; however, we can show that errors are after all not that important!

The QR Algorithm. Let A be a non-singular square matrix. We consider the sequence $A_0 = A$, A_1, A_2, …, A_k, A_{k+1}, … In the raw form of the QR algorithm, going from A_k to A_{k+1} is done as follows:

1. we compute the QR decomposition of A_k: $A_k = Q_k R_k$,
2. we compute $A_{k+1} = R_k Q_k$.

Let us program this method with A a symmetric real matrix:

```
sage: m = matrix(RDF, [[1,2,3,4],[1,0,2,6],[1,8,4,-2],[1,5,-10,-20]])
sage: Aref = transpose(m)*m
sage: A = copy(Aref)
sage: for i in range(0,20):
....:     Q, R = A.QR()
....:     A = R*Q
....:     print(A.str(lambda x: RR(x).str(digits=8)))
[    347.58031    -222.89331    -108.24117  -0.067928252]
[   -222.89331     243.51949     140.96827   0.081743964]
[   -108.24117     140.96827     90.867499 -0.0017822044]
[ -0.067928252   0.081743964 -0.0017822044   0.032699348]
...
[     585.03056 -3.2281469e-13 -6.8752767e-14 -9.9357605e-14]
[-3.0404094e-13       92.914265 -2.5444701e-14 -3.3835458e-15]
[-1.5340786e-39  7.0477800e-25       4.0229095  2.7461301e-14]
[ 1.1581440e-82 -4.1761905e-68  6.1677425e-42    0.032266909]
```

We observe a convergence towards an almost diagonal matrix. The diagonal coefficients are approximations to the eigenvalues of A. Let us check:

```
sage: Aref.eigenvalues()
[585.0305586200212, 92.91426499150643, 0.03226690899408103,
    4.022909479477674]
```

We can prove the convergence if the matrix is Hermitian positive definite. If we have a non-symmetric matrix, we should compute in $\mathbb{C}$, the eigenvalues being *a priori* complex, and, if the method converges, the lower triangular part of A_k tends to zero, while the diagonal tends to the eigenvalues of A.

The QR method requires several improvements to become efficient, in particular because the successive QR decompositions are expensive; among the common refinements, in general we first reduce the matrix A to a simpler form (Hessenberg's form: upper triangular plus the first subdiagonal), which makes the QR decompositions much cheaper; then, to speed-up convergence, we apply cleverly chosen translations $A := A + \sigma I$ (see for example [GVL12]). This is precisely the method used by Sage for dense matrices `CDF` or `RDF`.

In Practice. The above programs are mainly given as pedagogical examples; in practice, we will use the methods provided by Sage, which, whenever possible, call optimised routines from the Lapack library. The interface allows either to get only the eigenvalues, or both the eigenvalues and the corresponding eigenvectors:

```
sage: A = matrix(RDF, [[1,3,2],[1,2,3],[0,5,2]])
sage: eigen_vals, eigen_vects = A.eigenmatrix_right()
sage: eigen_vals
[   6.39294791648918                0.0                 0.0]
[                0.0  0.560519476111939                 0.0]
[                0.0                0.0 -1.9534673926011215]
sage: eigen_vects
[ 0.5424840601106511  0.9276845629439008 0.09834254667424457]
[ 0.5544692861094349 0.10329475725386986  -0.617227053099068]
[ 0.6310902116870117 -0.3587917847435306   0.780614827194734]
```

This example computes the diagonal matrix with eigenvalues, and the eigenvector matrix (whose columns correspond to eigenvectors).

Example (Computing the roots of a polynomial). Given a monic polynomial (with real or complex coefficients) $p(x) = x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$, it is easy to check that the eigenvalues of the companion matrix M, defined by $M_{i+1,i} = 1$ and $M_{i,n-1} = -a_i$, are the roots of p (see §8.2.3), which thus gives a method to approximate these roots:

```
sage: def pol2companion(p):
....:     n = len(p)
....:     m = matrix(RDF,n)
....:     for i in range(1,n):
....:         m[i,i-1]=1
....:     m.set_column(n-1,-p)
....:     return m
```

```
sage: q = vector(RDF,[1,-1,2,3,5,-1,10,11])
sage: comp = pol2companion(q); comp
[  0.0   0.0   0.0   0.0   0.0   0.0   0.0  -1.0]
[  1.0   0.0   0.0   0.0   0.0   0.0   0.0   1.0]
[  0.0   1.0   0.0   0.0   0.0   0.0   0.0  -2.0]
[  0.0   0.0   1.0   0.0   0.0   0.0   0.0  -3.0]
[  0.0   0.0   0.0   1.0   0.0   0.0   0.0  -5.0]
[  0.0   0.0   0.0   0.0   1.0   0.0   0.0   1.0]
[  0.0   0.0   0.0   0.0   0.0   1.0   0.0 -10.0]
[  0.0   0.0   0.0   0.0   0.0   0.0   1.0 -11.0]
sage: roots = comp.eigenvalues(); roots
[0.3475215101190289 + 0.5665505533984981*I, 0.3475215101190289 -
    0.5665505533984981*I,
 0.34502377696179265 + 0.4399870238588275*I, 0.34502377696179265 -
     0.4399870238588275*I,
 -0.5172576143252197 + 0.5129582067889322*I, -0.5172576143252197 -
     0.5129582067889322*I,
 -1.3669971645459291, -9.983578180965276]
```

In this example, the polynomial is represented by the list `q` of its coefficients, from 0 to $n-1$. The polynomial $x^2 - 3x + 2$ would thus be represented by `q=[2,-3]`.

13.2.9 Polynomial Curve Fitting: the Devil is Back

Continuous Version. We would like to approximate the function $f(x)$ by a polynomial $P(x)$ of degree $\leq n$, on the interval $[\alpha, \beta]$. We formulate the least squares problem:

$$\min_{a_0,\ldots,a_n \in \mathbb{R}} J(a_0, \ldots, a_n) = \int_\alpha^\beta (f(x) - \sum_{i=0}^{n} a_i x^i)^2 \, \mathrm{d}x.$$

By differentiating $J(a_0, \ldots, a_n)$ with respect to the coefficients a_i, we find that $a_0, \ldots, a_n$ are solutions of the linear system $Ma = F$ where $M_{i,j} = \int_\alpha^\beta x^i x^j \, dx$ and $F_j = \int_\alpha^\beta x^j f(x) \, dx$. We immediately see by looking at the case $\alpha = 0, \beta = 1$ that M is a Hilbert matrix! However a remedy exists: it suffices to use a basis of orthogonal polynomials (for example, for $\alpha = -1$ and $\beta = 1$ we obtain Legendre polynomials); then M becomes the identity matrix.

Discrete Version. We consider m observations $y_1, \ldots, y_m$ of a phenomenon at points $x_1, \ldots, x_m$. We want to fit a polynomial $\sum_{i=0}^{n-1} a_i x^i$ of degree (at most) $n-1$ among those values, with $n \leq m$. We thus minimise the functional:

$$J(a_0, \ldots, a_{n-1}) = \sum_{j=1}^{m} (\sum_{i=0}^{n-1} a_i x_j^i - y_j)^2.$$

Written like this, the problem will produce a matrix close to a Hilbert matrix and the system will be hard to solve accurately. Yet, we notice that $\langle P, Q\rangle = \sum_{j=1}^{m} P(x_j)\cdot Q(x_j)$ defines a scalar product on polynomials of degree at most $n-1$. We can thus first construct n polynomials of increasing degrees, orthonormal for this scalar product, and then diagonalise the linear system. Remembering[5] that the Gram-Schmidt process reduces to a 3-term recurrence for the computation of orthogonal polynomials, we are looking for the polynomial $P_{n+1}(x)$ under the form $P_{n+1}(x) = xP_n(x) - \alpha_n P_{n-1}(x) - \beta_n P_{n-2}(x)$: the `orthopoly` procedure below performs this computation (we represent here polynomials by the list of their coefficients: for example `[1,-2,3]` corresponds to the polynomial $1 - 2x + 3x^2$).

Horner's scheme to evaluate a polynomial yields the following program:

```
sage: def eval(P,x):
....:     if len(P) == 0:
....:         return 0
....:     else:
....:         return P[0]+x*eval(P[1:],x)
```

We can then encode the scalar product of two polynomials:

```
sage: def pscal(P,Q,lx):
....:     return float(sum(eval(P,s)*eval(Q,s) for s in lx))
```

and the operation $P \leftarrow P + aQ$ for two polynomials P and Q:

```
sage: def padd(P,a,Q):
....:     for i in range(0,len(Q)):
....:         P[i] += a*Q[i]
```

A more careful program should raise an exception when wrongly used; in our case, we use the following exception when $n > m$:

```
sage: class BadParamsforOrthop(Exception):
....:     def __init__(self, degreeplusone, npoints):
....:         self.deg = degreeplusone - 1
....:         self.np = npoints
....:     def __str__(self):
....:         return "degree: " + str(self.deg) + \
....:                " nb. points: " + repr(self.np)
```

The following procedure computes the n orthogonal polynomials:

```
sage: def orthopoly(n,x):
....:     if n > len(x):
....:         raise BadParamsforOrthop(n, len(x))
....:     orth = [[1./sqrt(float(len(x)))]]
....:     for p in range(1,n):
....:         nextp = copy(orth[p-1])
....:         nextp.insert(0,0)
```

[5] Proving it is not very difficult!

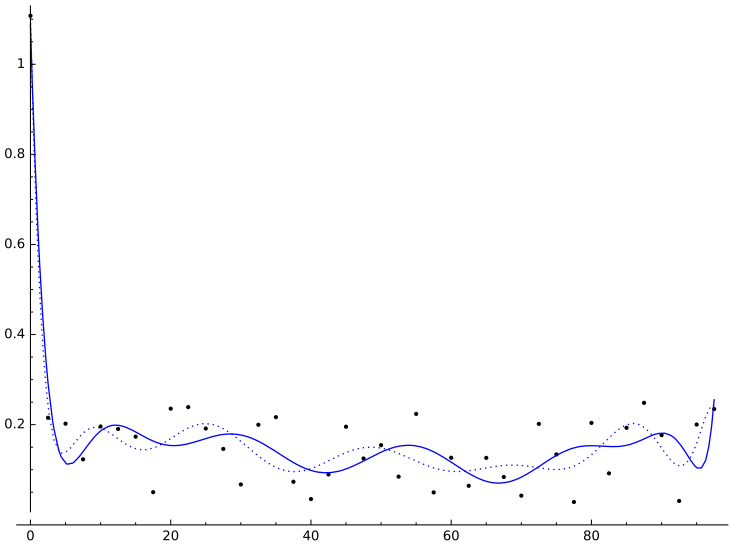


FIGURE 13.1 – Dotted line: naive fitting. Continuous line: orthogonal polynomials fitting.

```
....:         s = []
....:         for i in range(p-1,max(p-3,-1),-1):
....:             s.append(pscal(nextp, orth[i], x))
....:         j = 0
....:         for i in range(p-1,max(p-3,-1),-1):
....:             padd(nextp, -s[j], orth[i])
....:             j += 1
....:         norm = sqrt(pscal(nextp, nextp, x))
....:         nextpn = [nextp[i]/norm for i in range(len(nextp))]
....:         orth.append(nextpn)
....:     return orth
```

Once the orthogonal polynomials $P_0(x), \ldots, P_{n-1}(x)$ are computed, the solution is given by $P(x) = \sum_{i=0}^{n-1} \gamma_i P_i(x)$, with:

$$\gamma_i = \sum_{j=1}^{m} P_i(x_j) y_j,$$

which can be clearly expressed in turn in the monomial basis $1, x, \ldots, x^{n-1}$.

Example ($n = 15$):

```
sage: L = 40
sage: X = [100*float(i)/L for i in range(40)]
sage: Y = [float(1/(1+25*X[i]^2)+0.25*random()) for i in range(40)]
sage: n = 15; orth = orthopoly(n, X)
```

Let us compute the solution coefficients on the basis of orthogonal polynomials:

```
sage: coeff = [sum(Y[j]*eval(orth[i],X[j]) for j in
....:          range(0,len(X))) for i in range(0,n)]
```

We can then translate this result into the monomial basis $1, x, \ldots, x^{n-1}$, for example to draw the graph:

```
sage: polmin = [0 for i in range(0,n)]
sage: for i in range(0,n):
....:     padd(polmin, coeff[i], orth[i])
sage: p = lambda x: eval(polmin, x)
sage: plot(p(x), x, 0, X[len(X)-1])
```

We do not detail here the computation of the naive fitting on the monomial basis x^i, and its graphical representation. We obtain Figure 13.1. The two curves, which correspond to the fitting with a basis of orthogonal polynomials and to the monomial basis, are close; however, if we compute their residue (the value of the functional J) we find 0.1202 for the orthogonal polynomial basis, and 0.1363 for the naive fitting.

13.2.10 Implementation and Efficiency

The computations with matrices having `RDF` coefficients are performed with the processor floating-point unit, those having `RR` coefficients with the GNU MPFR library. Moreover, in the first case, Sage uses NumPy/SciPy, which in turn calls the Lapack library (written in Fortran), this library using the BLAS[6], which are optimised for each processor. We then get for the product of two matrices of size 1000:

```
sage: a = random_matrix(RR, 1000)
sage: b = random_matrix(RR, 1000)
sage: %time _ = a*b
CPU times: user 7min 50s, sys: 508 ms, total: 7min 50s
Wall time: 7min 51s
```

```
sage: c = random_matrix(RDF, 1000)
sage: d = random_matrix(RDF, 1000)
sage: %time _ = c*d
CPU times: user 40 ms, sys: 4 ms, total: 44 ms
Wall time: 45.2 ms
```

whence a ratio of more than 10000 between the multiplication times! (Recall that we compute with the same precision in both cases).

We can also notice the efficiency of computations with matrices having `RDF` coefficients: since the product of two square matrices of size n costs n^3 multiplications (and as many additions), we perform here 10^9 additions and multiplications in 0.045 second; this is about $44 \cdot 10^9$ floating-point operations per second, which corresponds to 44 gigaflops. The processor clock having a frequency of 3.3 Ghz, we thus perform *more* than one operation by clock cycle: this is possible thanks

[6] Basic Linear Algebra Subroutines (matrix-vector products, matrix-matrix products, etc.).

to the almost direct call to the BLAS corresponding routine (which uses the different possibilities to compute in parallel in modern processors cores). There exists an algorithm of cost lower than n^3 to compute the product of two matrices, namely Strassen's algorithm. It is not often used in practice for floating-point computations, as it is more sensitive to roundoff errors as the naive method[Hig93].

13.3 Sparse Matrices

Sparse matrices arise quite often in scientific computing: the sparsity is indeed a wanted property which enables us to solve problems of large size, out of reach with dense matrices.

An approximate definition: we will say that a sequence M_n of matrices, with M_n of size n, is a family of sparse matrices if the number of non-zero coefficients of M_n is $O(n)$.

Clearly, those matrices are encoded in the computer using data structures where only the non-zero elements are stored. By taking into account the sparsity of the matrices, we want of course to save memory to be able to represent large matrices, but also heavily reduce the computation cost.

13.3.1 Where do Sparse Systems Come From?

Boundary Problems. The most common source of sparse linear systems is the discretisation of partial derivative equations. Let us consider for example the Poisson equation (stationary heat equation):

$$-\Delta u = f$$

where $u = u(x, y)$, $f = f(x, y)$,

$$\Delta u := \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}.$$

The equation is considered in the square $[0, 1]^2$, with boundary conditions $u = 0$ on the square border. The one-dimensional analogue is the problem

$$-\frac{\partial^2 u}{\partial x^2} = f, \tag{13.1}$$

with $u(0) = u(1) = 0$.

To approximate the solution of this equation, one of the simplest methods consists in using the finite difference method: we divide the range $[0, 1]$ into a finite number N of intervals of constant width h. We denote by u_i the approximation of u at the point $x_i = ih$. We approximate the derivative of u by $(u_{i+1} - u_i)/h$, and its second derivative by

$$\frac{(u_{i+1} - u_i)/h - (u_i - u_{i-1})/h}{h} = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}.$$

We immediately see that the values $u_0, \ldots, u_N$, approximating u at points ih, satisfy a linear system having only 3 non-zero terms per row (and whose matrix is symmetric positive definite).

In dimension 2, we can stick a grid of step h on the unit square, and we obtain a pentadiagonal system (with $4/h^2$ on the diagonal, and $-1/h^2$ on the first two subdiagonals and on the first two superdiagonals). In dimension 3, if we proceed similarly in a cube, we obtain a system where each row has 7 non-zero coefficients. We therefore indeed get very sparse matrices.

Random Walk on a Large Sparse Graph. We consider a graph where each vertex is linked to a small number of vertices (small with respect to the total number of vertices). For example, we can figure out a graph where the vertices are the internet pages: each page only links to a small number of other pages (which defines the graph edges), but it is clearly a huge graph. A random walk on the graph is defined by a stochastic matrix, i.e., a matrix whose coefficients are reals between 0 and 1, with the sum of coefficients being 1 on each line. We can show that such a matrix A has a dominant eigenvalue equal to 1. The stationary distribution of the random walk is the (left) eigenvector x associated to the dominant eigenvalue, i.e., the vector x satisfying $xA = x$. One of the most dramatic applications is the *Pagerank* algorithm from *Google*, where the vector x is used to balance the search results.

13.3.2 Sparse Matrices in Sage

Sage gives us the chance to work with sparse matrices, by adding `sparse = True` when creating the matrix. It corresponds to a representation as a dictionary (§3.3.9). In addition, computations on large sparse matrices with coefficients in `RDF` or `CDF` are performed by Sage using the SciPy library, which offers its own classes of sparse matrices. In the current situation, there is no interface between the sparse matrices of Sage and those of SciPy. We thus have to directly use the SciPy objects.

The available SciPy classes for sparse matrices are:

- a list-of-lists structure (different however from that used by Sage), which is quite handy to create and modify matrices, the `lil_matrix` class;
- some immutable structures, which store only non-zero coefficients, and which are standard formats in sparse linear algebra (`csr` and `csv` formats).

13.3.3 Solving Linear Systems

For systems of moderate size, we can use a direct method, based on the LU decomposition. We can easily convince ourselves that, in the LU decomposition of a sparse matrix A, the L and U factors usually contain more non-zero terms altogether than A. It is necessary to renumber the unknowns to limit the memory used, as in the SuperLU library used by Sage in a transparent manner:

```
sage: from scipy.sparse.linalg.dsolve import *
```

```
sage: from scipy.sparse import lil_matrix
sage: from numpy import array
sage: n = 200
sage: n2 = n*n
sage: A = lil_matrix((n2, n2))
sage: h2 = 1./float((n+1)^2)
sage: for i in range(0,n2):
....:     A[i,i]=4*h2+1.
....:     if i+1<n2: A[i,int(i+1)]=-h2
....:     if i>0:    A[i,int(i-1)]=-h2
....:     if i+n<n2: A[i,int(i+n)]=-h2
....:     if i-n>=0: A[i,int(i-n)]=-h2
sage: Acsc = A.tocsc()
sage: b = array([1 for i in range(0,n2)])
sage: solve = factorized(Acsc) # LU factorisation
sage: S = solve(b)             # resolution
```

Once we have created the matrix as a `lil_matrix` (warning, this format requires indices of type `int` from Python), we have to convert it to the `csc` format. The above program is not very efficient: the construction of the `lil_matrix` is slow, the `lil_matrix` data structure being quite inefficient. However, the conversion to a `csc` matrix and its factorisation are fast; the following resolution is even faster.

Iterative Methods. The main principle of these methods is to build a sequence converging towards the solution of the $Ax = b$ system. Modern iterative methods rely on the Krylov space K_n, the vector space spanned by $\{b, Ab, \ldots, A^n b\}$. Among the most popular methods, let us mention:

- the conjugate gradient method: it can only be used for systems whose matrix A is symmetric positive definite. In this case $\|x\|_A = \sqrt{x^t Ax}$ is a norm, and the iterate x_n is such that it minimises the error $\|x - x_n\|_A$ between the solution x and x_n for $x_n \in K_n$ (some formulas exist which are easy to program, cf. for example [GVL12]);

- the generalised minimal residual method (GMRES): it is designed for non-symmetric linear systems. At the n-th iteration, the Euclidean residual norm $\|Ax_n - b\|_2$ is minimised for $x_n \in K_n$. We notice that it is a least squares problem.

In practice, these methods are only efficient with *preconditioning*: instead of solving $Ax = b$, we solve $MAx = Mb$ where M is a matrix such that MA has a better condition number than A. The study and discovery of efficient preconditioners is an active branch of numerical analysis, with fruitful developments. As an example, here is the resolution of the system studied above by the conjugate gradient method, where the preconditioner M is the inverse of the diagonal of A. It is a simple but not very efficient preconditioner:

```
sage: b = array([1 for i in range(0,n2)])
```

Most useful commands (dense matrices)	
Solve a linear system	`x = A\b`
LU decomposition	`P, L, U =  A.LU()`
Cholesky decomposition	`C = A.cholesky()`
QR decomposition	`Q, R = A.QR()`
Singular value decomposition	`U, Sig, V = A.SVD()`
Eigenvalues and eigenvectors	`Val, Vect = A.eigenmatrix_right()`

TABLE 13.1 – Summary.

```
sage: m = lil_matrix((n2, n2))
sage: for i in range(0,n2):
....:     m[i,i] = 1./A[i,i]
sage: msc = m.tocsc()
sage: from scipy.sparse.linalg import cg
sage: x = cg(A, b, M = msc, tol=1.e-8)
```

13.3.4 Eigenvalues, Eigenvectors

The Power Method. The power method is particularly well suited for huge sparse matrices; indeed, to implement the algorithm, it is enough to know how to perform matrix-vector and scalar products. As an example, let us come back to random walks on a sparse graph, and let us compute the stationary distribution using the power method:

```
sage: from scipy import sparse
sage: from numpy.linalg import *
sage: from numpy import array
sage: from numpy.random import rand
sage: def power(A,x):              # power iteration
....:     for i in range(0,1000):
....:         y = A*x
....:         z = y/norm(y)
....:         lam = sum(x*y)
....:         s = norm(x-z)
....:         print("{i} s={s} lambda={lam}".format(i=i, s=s, lam=lam))
....:         if s < 1e-3:
....:             break
....:         x = z
....:     return x
sage: n = 1000
sage: m = 5
sage: # build a stochastic matrix of size n
sage: # with m non-zero coefficients per row
sage: A1 = sparse.lil_matrix((n, n))
```

```
sage: for i in range(0,n):
....:     for j in range(0,m):
....:         l = int(n*rand())
....:         A1[l,i] = rand()
sage: for i in range(0,n):
....:     s = sum(A1[i,0:n])
....:     A1[i,0:n] /= s
sage: At = A1.transpose().tocsc()
sage: x = array([rand() for i in range(0,n)])
sage: # compute the dominant eigenvalue
sage: # and the associated eigenvector
sage: y = power(At, x)
0 s=17.0241218112 lambda=235.567796432
1 s=0.39337173784 lambda=0.908668201953
2 s=0.230865716856 lambda=0.967356896036
3 s=0.134156683993 lambda=0.986660315554
4 s=0.0789423487458 lambda=0.995424635219
...
```

When running this example, we might play with measuring the time spent in its different parts, and we will observe that almost all the time is spent in constructing the matrix; computing the transpose is not very expensive; the power iterations themselves take negligible time (about 2% of the total time on the test computer). The representation using list of large matrices is not very efficient, and this kind of problem should be rather solved with compiled languages and appropriate data structures.

13.3.5 More Thoughts on Solving Large Non-Linear Systems

The power method and the methods using the Krylov space share a key property: they only require the computation of matrix-vector products. We do not even need to know the matrix, we only need to know the *action* of the matrix on a vector. This is why these methods are also called "black box" methods. It is thus possible to perform some computations in cases where the matrix is not explicitly known, or when we are unable to compute it. The SciPy methods allow in fact *linear operators* as arguments. We invite the reader to consider the following application, and maybe implement it.

Let $F : \mathbb{R}^n \to \mathbb{R}^n$. We want to solve $F(x) = 0$. We consider Newton's method, where we compute iterates $x_{n+1} = x_n - J(x_n)^{-1} \cdot F(x_n)$, starting from x_0. The Jacobian matrix $J(x_n)$ is the matrix of partial derivatives of F at x_n. In practice, one will successively solve $J(x_n)y_n = F(x_n)$, then compute $x_{n+1} = x_n - y_n$. We thus have to solve a linear system. If F is somewhat hard to compute, then its partial derivatives are in general still harder to compute and to program, and this computation might be almost impossible. We therefore bring automatic differentiation to the rescue: if e_j is the vector from $\mathbb{R}^n$ with all its components 0 except the j-th one equal to 1, then $(F(x + he_j) - F(x))/h$ yields a (good)

approximation of the j-th column of the Jacobian matrix for small h. We thus have to perform $n+1$ evaluations of F to obtain an approximation of J, which is quite expensive if n is large. How about applying an iterative method like Krylov to solve the system $J(x_n)y_n = F(x_n)$? We notice that $J(x_n)V \simeq (F(x_n+hV)-F(x_n))/h$ for h small enough, which avoids the computation of the *whole* matrix. Within SciPy, it is sufficient to define a "linear" operator as being the application $V \to (F(x_n + hV) - F(x_n))/h$. This kind of method is quite often used to solve large non-linear systems. The "matrix" being non-symmetric, we will use for example the GMRES method.

14 Numerical Integration and Differential Equations

This chapter covers the numerical computation of integrals (§14.1) and the numerical resolution of ordinary differential equations (§14.2) with Sage. We recall the theoretical bases of integration methods, then we detail the available functions and their usage (§14.1.1).

We have already seen in §2.3.8 how to compute an integral *symbolically* with Sage; this will only be briefly mentioned in this chapter. This "symbolic-numeric" approach, when it is possible, is one of the strengths of Sage, and should be preferred because the number of numerical computations performed — and thus the number of roundoff errors — is usually less than with purely numerical integration methods.

For differential equations, we give a quick introduction to the classical resolution methods, and after an introductory example (§14.2.1), we describe the functionalities of Sage (§14.2.2).

14.1 Numerical Integration

We consider the numerical integration of real functions; for a function $f : I \longrightarrow \mathbb{R}$, where I is an interval of $\mathbb{R}$, we want to approximate:

$$\int_I f(x)\,\mathrm{d}x.$$

For example, let us compute

$$\int_1^3 \exp(-x^2)\log(x)\,\mathrm{d}x.$$

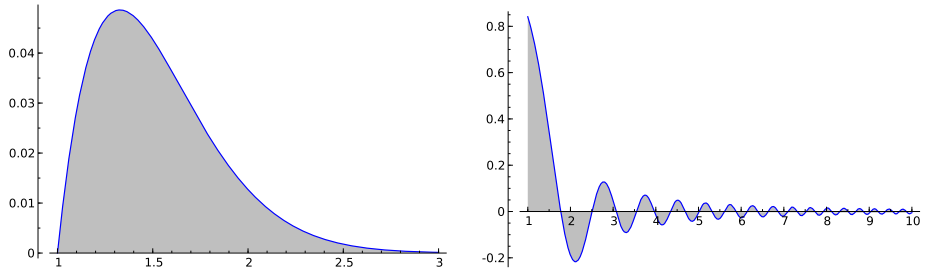


FIGURE 14.1 – The functions $x \mapsto \exp(-x^2)\log(x)$ and $x \mapsto \frac{\sin(x^2)}{x^2}$.

```
sage: x = var('x'); f(x) = exp(-x^2) * log(x)
sage: N(integrate(f, x, 1, 3))
0.035860294991267694
```

```
sage: plot(f, 1, 3, fill='axis')
```

Since the `integrate` function computes a symbolic integral of the given expression, we have to explicitly ask if we want a numerical value.

It is also possible, in principle, to compute integrals on an unbounded interval:

```
sage: N(integrate(sin(x^2)/(x^2), x, 1, infinity))
0.285736646322853 - 6.93889390390723e-18*I
```

```
sage: plot(sin(x^2)/(x^2), x, 1, 10, fill='axis')
```

Several methods exist in Sage to perform numerical integration, and even if their implementations differ technically, they all follow one of the two following principles:

- a polynomial interpolation (in particular the Gauss-Kronrod method);
- a function transformation (double exponential method).

Interpolation Methods. In these methods, we evaluate the function f to integrate at a given number n of well-chosen points $x_1, x_2, \ldots, x_n$, and we deduce an approximation of the integral of f on $[a, b]$ by

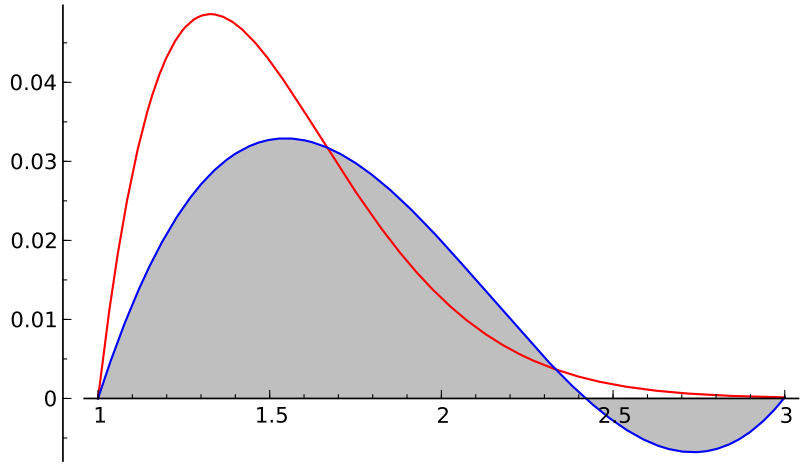
$$\int_a^b f(x)\,\mathrm{d}x \approx \sum_{i=1}^{n} w_i f(x_i).$$

The w_i coefficients are the "weights" of the method, which are determined by the fact that the method should be exact for any polynomial f of degree less or equal to $n-1$. For fixed points (x_i), the weights (w_i) are uniquely determined by this condition. We define the *order* of the method as the maximal degree of

polynomials whose integral is exact; this order is thus at least $n-1$ by construction, but it might be larger.

For example, the family of Newton-Cotes integration methods (which contains the rectangle rule, the trapezoidal rule, Simpson's rule) use equally spaced points on the interval $[a, b]$:

```
sage: fp = plot(f, 1, 3, color='red')
sage: n = 4
sage: interp_points = [(1+2*u/(n-1), N(f(1+2*u/(n-1))))
....:                  for u in xrange(n)]
sage: A = PolynomialRing(RR, 'x')
sage: pp = plot(A.lagrange_polynomial(interp_points), 1, 3, fill='axis')
sage: fp+pp
```



For the interpolation methods, we can consider that we first compute the Lagrange interpolating polynomial of the given function, and that the integral of this polynomial is the chosen approximate value for the integral. These two steps are in fact merged into a formula called "quadrature rule", the Lagrange interpolation polynomial being never explicitly computed. The choice of the interpolation points is crucial for the quality of the polynomial approximation, and equally spaced points do not ensure convergence when the number of points increases (this is called Runge's phenomenon). The corresponding quadrature rule might thus suffer from this problem, illustrated in Figure 14.2.

When we ask Sage to compute a numerical approximation of an integral on an interval $[a, b]$, the quadrature rule is not directly applied to the whole domain: $[a, b]$ is divided into sub-intervals small enough such that the quadrature rule gives enough accuracy (this is called "method composition"). The subdivision strategy might be for example dynamically tuned for the function to integrate: if we denote $I_a^b(f)$ the value of $\int_a^b f(x)\,\mathrm{d}x$ computed by the quadrature rule, we compare

$$I_0 = I_a^b(f)$$

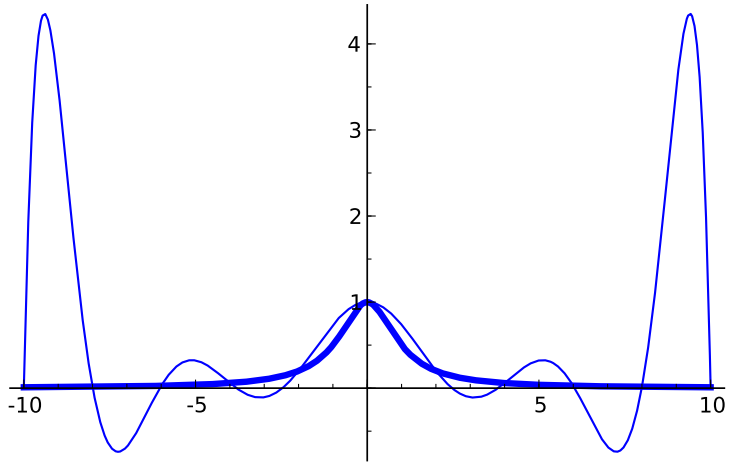


FIGURE 14.2 – Interpolation using a degree-10 polynomial (thin line) of the function $x \mapsto 1/(1+x^2)$ (thick line) at 11 points equally spaced on $[-10, 10]$. Runge's phenomenon appears at endpoints.

with

$$I_1 = I_a^{(a+b)/2}(f) + I_{(a+b)/2}^b(f)$$

and we stop the subdivision process when $|I_0 - I_1|$ is small enough compared to I_0 with the required precision. Here comes into play the method order: for a quadrature rule of order n, dividing the interval in 2 will divide the theoretical error by 2^n, i.e., without taking into account roundoff errors.

One of the interpolation methods available within Sage is the Gauss-Legendre method. In this method, the n integration points are the roots of the Legendre polynomial of degree n (with a translated interval of definition to match the considered range $[a, b]$). The properties of Legendre polynomials, which are orthogonal for the scalar product

$$\langle f, g \rangle = \int_a^b f(x)g(x)\,\mathrm{d}x,$$

imply that the corresponding quadrature rule computes exactly the integrals of polynomials of degree up to $2n-1$, instead of only $n-1$ as in the general case. Moreover, the corresponding integration weights are always positive, which minimises the effect of numerical issues like *cancellation*[1].

To conclude on interpolation methods, the Gauss-Kronrod method with $2n+1$ points is an "extension" of the Gauss-Legendre method with n points:

- n of the $2n+1$ points are the Gauss-Legendre integration points;

[1] This phenomenon happens when a sum of real numbers is significantly smaller (in absolute value) than the summands: each rounding error might then be larger than the final result, which yields a total loss of accuracy. See also §11.3.3.

- the method is exact for any polynomial of degree up to $3n+1$.

We can naively remark that the $3n+2$ unknowns (the $2n+1$ weights and the $n+1$ added points) are *a priori* determined by requiring that the method is of order at least $3n+1$ (which indeed yields $3n+2$ equalities). Beware that the weights associated in the Gauss-Kronrod method to the n Gauss-Legendre points have no reason to coincide with those associated in the original Gauss-Legendre method.

Such an extension method becomes particularly interesting when the main cost of a quadrature rule is the number of evaluations of the function f to integrate (moreover if the integration points and weights are tabulated). The Gauss-Kronrod method being in principle more precise than the Gauss-Legendre method, we can use its result I_1 to validate the result I_0 of the latter method, and obtain an estimate of the error as $|I_1 - I_0|$, while minimising the number of evaluations of f. The reader might compare this strategy, which is particular to the Gauss-Legendre method, with the more general subdivision strategy described on page 308.

Double Exponential Methods. The double exponential (DE) methods rely on a change of variable which transforms a bounded integration range into $\mathbb{R}$, and on the very good accuracy of the trapezoidal rule on $\mathbb{R}$ for analytic functions. For a function f integrable on $\mathbb{R}$, and an integration step h, the trapezoidal rule computes

$$I_h = h \sum_{i=-\infty}^{+\infty} f(hi)$$

as approximate value for $\int_{-\infty}^{+\infty} f(x)\,\mathrm{d}x$. Discovered in 1973 by Takahasi and Mori, the double exponential transform is commonly used by numerical integration software tools. We describe here its main features; an introduction to this transform and its discovery is given in [Mor05]. This article gives in particular an explanation of the surprising good accuracy of the trapezoidal rule, which is optimal in a certain sense, for analytic functions on $\mathbb{R}$.

To compute

$$I = \int_{-1}^{1} f(x)\,\mathrm{d}x,$$

it is possible to use a transform $x = \varphi(t)$ where φ is analytic on $\mathbb{R}$ and satisfies

$$\lim_{t\to-\infty} \varphi(t) = -1, \quad \lim_{t\to\infty} \varphi(t) = 1,$$

and then

$$I = \int_{-\infty}^{\infty} f(\varphi(t))\varphi'(t)\,\mathrm{d}t.$$

Applying the trapezoidal rule to this last expression, we compute

$$I_h^N = h \sum_{k=-N}^{N} f(\varphi(kh))\varphi'(kh)$$

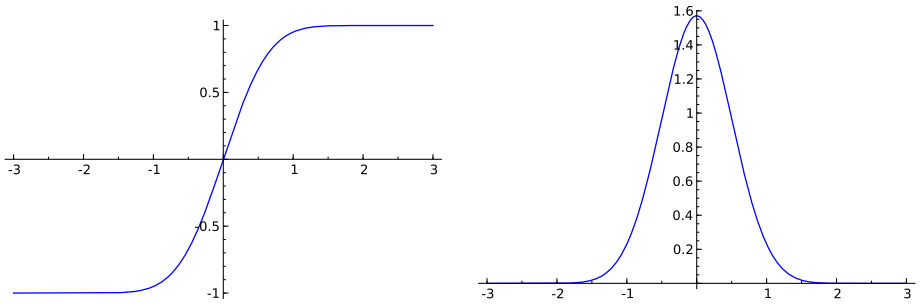


FIGURE 14.3 – The transform $\varphi(t) = \tanh(\frac{\pi}{2}\sinh t)$ used in the double exponential method (left) and the decreasing of $\varphi'(t)$ (right).

for a given step h, and truncating the sum to terms from $-N$ to N. The proposed transform is chosen to be

$$\varphi(t) = \tanh\left(\frac{\pi}{2}\sinh t\right)$$

which yields the formula

$$I_h^N = h\sum_{k=-N}^{N} f\left(\tanh\left(\frac{\pi}{2}\sinh kh\right)\right)\frac{\frac{\pi}{2}\cosh kh}{\cosh^2(\frac{\pi}{2}\sinh kh)}.$$

The name of the method comes from the doubly exponential decreasing of

$$\varphi'(t) = \frac{\frac{\pi}{2}\cosh t}{\cosh^2(\frac{\pi}{2}\sinh t)}$$

when $|t| \to \infty$ (see Figure 14.3). The main idea of the transform is to concentrate the contribution of the function to integrate around 0, which explains the huge decreasing of $\varphi'(t)$ when $|t|$ grows. A compromise should be found between the choice of parameters and of the transform φ: a decreasing more than doubly exponential decreases the truncation error but increases the discretisation error.

Since the discovery of the DE transform, this method is used alone or with other transforms, according to the nature of the integrand, of its singularities and of the integration domain. An example of other transform is the cardinal sine function "sinc":

$$f(x) \approx \sum_{k=-N}^{N} f(kh)S_{k,h}(x)$$

where

$$S_{k,h}(x) = \frac{\sin(\pi(x-kh)/h)}{\pi(x-kh)/h},$$

used together with the double exponential method in [TSM05] to improve the previous methods which relied on a single exponential transform $\varphi(t) = \tanh(t/2)$.

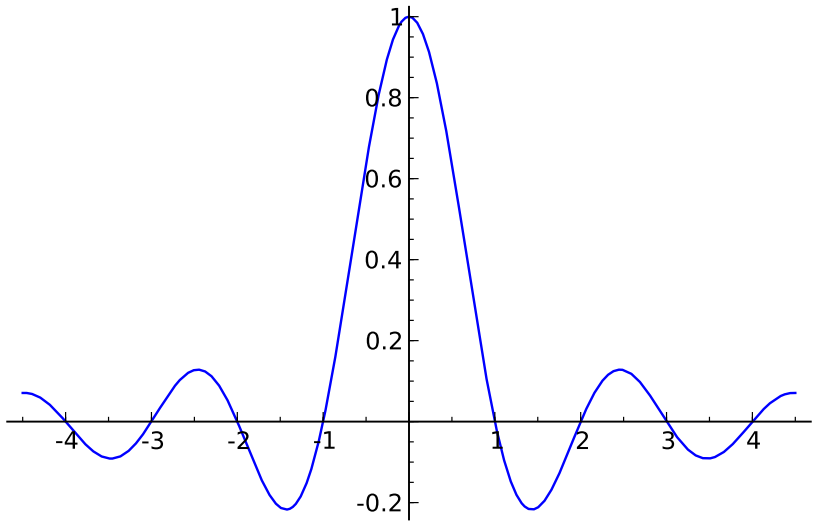


FIGURE 14.4 – The cardinal sine function.

The sinc function is defined by

$$\mathrm{sinc} = \begin{cases} 1 & \text{if } x = 0, \\ \dfrac{\sin(\pi x)}{\pi x} & \text{otherwise,} \end{cases}$$

and its graph is shown in Figure 14.4.

The choice of the transform greatly influences the quality of the result in the case of singularities at the interval bounds (there is no good solution yet in the case of singularities inside the interval). We will see later that in the version of Sage we consider, PARI/GP is the only system providing double exponential transforms allowing to specify the behaviour at interval bounds.

14.1.1 Available Integration Functions

We will now see in more detail the various ways to compute a numerical integral with Sage, through the following examples:

$$\begin{aligned} I_1 &= \int_{17}^{42} \exp(-x^2)\log(x)\,\mathrm{d}x, \quad I_2 = \int_0^1 x\log(1+x)\,\mathrm{d}x = \frac{1}{4}, \\ I_3 &= \int_0^1 \sqrt{1-x^2}\,\mathrm{d}x = \frac{\pi}{4}, \\ I_4 &= \int_0^1 \max(\sin(x),\cos(x))\,\mathrm{d}x = \int_0^{\frac{\pi}{4}} \cos(x)\,\mathrm{d}x + \int_{\frac{\pi}{4}}^1 \sin(x)\,\mathrm{d}x \\ &= \sin\frac{\pi}{4} + \cos\frac{\pi}{4} - \cos 1 = \sqrt{2} - \cos 1, \end{aligned}$$

$$I_5 = \int_0^1 \sin(\sin(x))\,\mathrm{d}x, \quad I_6 = \int_0^\pi \sin(x)\exp(\cos(x))\,\mathrm{d}x = e - \frac{1}{e},$$
$$I_7 = \int_0^1 \frac{1}{1+10^{10}x^2}\,\mathrm{d}x = 10^{-5}\arctan(10^5), \quad I_8 = \int_0^{1,1} \exp(-x^{100})\,\mathrm{d}x,$$
$$I_9 = \int_0^{10} x^2\sin(x^3)\,\mathrm{d}x = \frac{1}{3}(1-\cos(1000)), \quad I_{10} = \int_0^1 \sqrt{x}\,\mathrm{d}x = \frac{2}{3}.$$

We do not give an exhaustive description of the integration functions — which can be found in the on-line help — but only their more common usage.

`N(integrate(...))`. The first numerical method which can be used with Sage is `N(integrate(...))`:

```
sage: N(integrate(exp(-x^2)*log(x), x, 17, 42))
2.5657285006962035e-127
```

It is not guaranteed that the integral will be computed really *numerically* this way. Indeed, the `integrate` command requires a symbolic integration; if this succeeds, then Sage will just evaluate numerically the obtained symbolic expression:

```
sage: integrate(log(1+x)*x, x, 0, 1)
1/4
sage: N(integrate(log(1+x)*x, x, 0, 1))
0.250000000000000
```

`numerical_integral`. On the contrary, the `numerical_integral` function *explicitly* requires a numerical integration of the given function. It calls the GSL library (GNU Scientific Library), which implements the Gauss-Kronrod method for a fixed number n of integration points. The points and weights are precomputed, and the precision is limited to that of machine floating-point numbers (53-bit significand). The result is a pair with the computed approximation and an estimate of the error:

```
sage: numerical_integral(exp(-x^2)*log(x), 17, 42)
(2.5657285006962035e-127, 3.3540254049238093e-128)
```

The error estimate is not a guaranteed bound on the error, but a simple indication of the difficulty to approximate the given integral. In the above example, the error estimate is so large that all digits of the result might be wrong, except the most significant one.

The arguments of `numerical_integral` allow in particular:

- to choose the number of evaluation points (six choices from `rule=1` for 15 points to `rule=6` for 61 points, which is the default value);
- to ask for an adaptive subdivision (default choice), or require a direct application of the composition method on the integration interval (with the option `algorithm='qng'`);
- to bound the number of evaluations of the integrand.

Forbidding GSL to perform an adaptive integration might lead to a loss of accuracy:

```
sage: numerical_integral(exp(-x^100), 0, 1.1)
(0.99432585119150..., 4.0775730...e-09)
sage: numerical_integral(exp(-x^100), 0, 1.1, algorithm='qng')
(0.994327538576531..., 0.016840666914688864)
```

When the `integrate` command does not find an analytic expression for the requested integral, it returns the input integral unchanged:

```
sage: integrate(exp(-x^2)*log(x), x, 17, 42)
integrate(e^(-x^2)*log(x), x, 17, 42)
```

and the numerical evaluation with `N` calls `numerical_integral`. This explains in particular why the precision parameter is ignored in that case:

```
sage: N(integrate(exp(-x^2)*log(x), x, 17, 42), digits=60)
2.5657285006962035e-127
```

but we get:

```
sage: N(integrate(sin(x)*exp(cos(x)), x, 0, pi), digits=60)
2.3504023872876029137647637011912016303114359626681917404591 3
```

because the symbolic integration succeeds in that case.

`sage.calculus.calculus.nintegral`. For the symbolic functions, it is possible to ask Maxima for a numerical approximation of the integral:

```
sage: sage.calculus.calculus.nintegral(sin(sin(x)), x, 0, 1)
(0.430606103120690..., 4.78068810228705...e-15, 21, 0)
```

and it is also possible to directly call the `nintegral` method on an object of type `Expression`:

```
sage: g = sin(sin(x))
sage: g.nintegral(x, 0, 1)
(0.430606103120690..., 4.78068810228705...e-15, 21, 0)
```

Maxima calls the QUADPACK numerical quadrature library, which like GSL is limited to machine floating-point numbers. The `nintegral` method uses an adaptive subdivision strategy of the integration interval, and we might indicate:

- the wanted relative accuracy of the result;
- the maximal number of sub-intervals for the computation.

The output is a tuple:

1. the approximation of the integral;
2. an estimate of the absolute error;
3. the number of evaluations of the integrand;
4. an error code (0 if no problem was encountered, for more details on the other possible values the reader should look at the reference manual with `sage.calculus.calculus.nintegral?`).

gp('intnum(...)'). The PARI/GP calculator, which is available within Sage, also implements a numerical integration command called `intnum`:

```
sage: gp('intnum(x=17, 20, exp(-x^2)*log(x))')
2.5657285005610514829173563961304785900 E-127
```

The `intnum` command uses the double exponential method, but beware, it does not guarantee any correct significant digit of the result!

We might ask for a given precision of the result by modifying the global precision of the PARI/GP interpreter:

```
sage: gp('intnum(x=0, 1, sin(sin(x)))')
0.43060610312069060491237735524846578643
sage: old_prec = gp.set_precision(50)
sage: gp('intnum(x=0, 1, sin(sin(x)))')
0.43060610312069060491237735524846578643360804182200
```

A major bottleneck is that the integrand must be given as a character string, following the PARI/GP syntax, it is thus not possible to integrate arbitrary functions this way.

The `intnum` command allows us to indicate the behaviour of the integrand at the interval bounds. The following example demonstrates the corresponding effect on the result accuracy. Let us integrate $x \mapsto x^{-99/100}$ without any indication:

```
sage: p = gp.set_precision(old_prec) # we reset the default precision
sage: gp('intnum(x=0, 1, x^(-99/100))')
73.62914262423378365
```

If we give the nature of the singularity, i.e., that the function behaves like $x \mapsto x^{-99/100}$ at 0:

```
sage: gp('intnum(x=[0, -99/100], 1, x^(-99/100))')
100.00000000000000000000000000000000000
```

The user is responsible for the exactness of the given behaviour; if we erroneously say that the function behaves like $x \mapsto x^{-1/42}$ at 0, the error will remain large:

```
sage: gp('intnum(x=[0, -1/42], 1, x^(-99/100))')
74.47274932028288503
```

mpmath.quad*. The `mpmath` library is an arbitrary precision numerical library written in Python. It is able to compute with floating-point real and complex numbers, with matrices, and floating-point real intervals.

It provides numerical quadrature functions (the principal one being `quad`) and is available within Sage, after importing it:

```
sage: import mpmath
sage: mpmath.mp.prec = 53
sage: mpmath.quad(lambda x: mpmath.sin(mpmath.sin(x)), [0, 1])
mpf('0.43060610312069059')
```

The mpmath floating-point numbers mpf(...) might be converted in Sage floating-point numbers and vice-versa:

```
sage: a = RDF(pi); b = mpmath.mpf(a); b
mpf('3.1415926535897931')
sage: c = RDF(b); c
3.141592653589793
```

The user might specify the wanted precision either in decimal digits (mpmath.mp.dps) or in bits (mpmath.mp.prec), as with the command N from Sage: N(...,53) or N(...,digits=17).

```
sage: mpmath.mp.prec = 113
sage: mpmath.quad(lambda x: mpmath.sin(mpmath.sin(x)), [0, 1])
mpf('0.43060610312069060491237735524846580 9')
```

The mpmath.quad function might use either the Gauss-Legendre method, or the double exponential method (this latter being used by default). The method to use might be fixed with the functions mpmath.quadgl and mpmath.quadts.[2]

An important limitation of the mpmath integration functions within Sage is that they cannot manipulate arbitrary Sage expressions:

```
sage: mpmath.quad(sin(sin(x)), [0, 1])
Traceback (most recent call last):
...
TypeError: no canonical coercion from <type 'sage.libs.mpmath.ext_main.
    mpf'> to Symbolic Ring
```

The situation is however less dramatic than for PARI/GP which is limited to its own syntax. It is indeed possible to define evaluation and conversion procedures to integrate via mpmath.quad arbitrary functions[3]:

```
sage: g(x) = max_symbolic(sin(x), cos(x))
sage: mpmath.mp.prec = 100
sage: mpmath.quadts(lambda x: g(N(x, 100)), [0, 1])
mpf('0.87391241626303543595797908625 2')
```

The integration of irregular functions (like the above I_4 example) might lead to a significant accuracy loss, even when asking for a large precision:

```
sage: mpmath.mp.prec = 170
sage: mpmath.quadts(lambda x: g(N(x, 190)), [0, 1])
mpf('0.873910907574009752053930059819624763440541483541887 94')
sage: N(sqrt(2) - cos(1), 100)
0.87391125650495533140075211677
```

We get only 5 correct digits here. We can nevertheless help mpmath by suggesting a subdivision of the interval domain (here at the irregular point, cf. Figure 14.5):

[2] Which recalls the name of the transform $\varphi : t \mapsto \tanh(\frac{\pi}{2}\sinh(t))$ which was seen above.

[3] The reader wondering why we used max_symbolic will try with max instead, and will look at the max_symbolic on-line help.

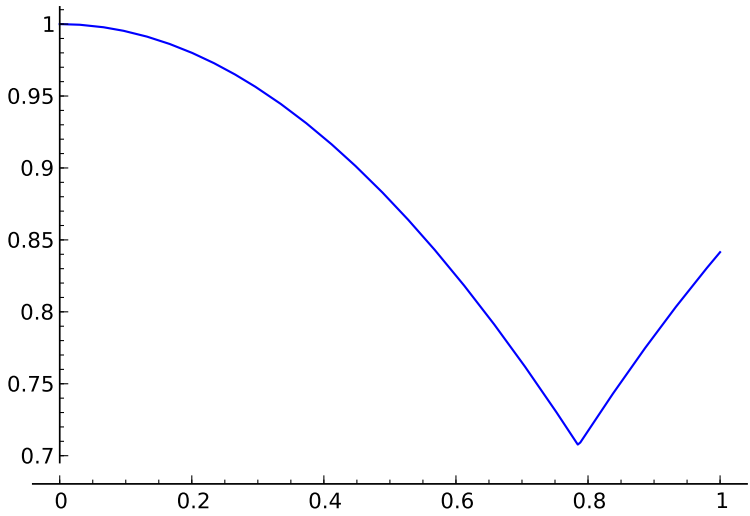


FIGURE 14.5 – The function $x \mapsto \max(\sin(x), \cos(x))$. The irregularity in $\pi/4$ renders numerical integration quite troublesome.

```
sage: mpmath.quadts(lambda x: g(N(x, 170)), [0, mpmath.pi / 4, 1])
mpf('0.87391125650495533140075211676672147483736145475902551')
```

The discontinuous functions (or those having a discontinuous derivative) are a classical "trap" for integration methods; however an automatic subdivision strategy, as described above, can limit the damage.

Exercise 49 (Computation of Newton-Cotes coefficients)**.** We want to compute the coefficients of the Newton-Cotes method with n points, which is not available within Sage. We consider for the sake of simplicity that the interval domain is $I = [0, n-1]$, the integration points being thus $x_1 = 0, x_2 = 1, \ldots, x_n = n-1$. The coefficients (w_i) of the quadrature rule are such that the equation

$$\int_0^{n-1} f(x)\,\mathrm{d}x = \sum_{i=0}^{n-1} w_i f(i) \tag{14.1}$$

is exact for any polynomial f of degree up to $n-1$.

1. We consider for $i \in \{0, \ldots, n-1\}$ the polynomial $P_i(X) = \prod_{\substack{j=0 \\ j \neq i}}^{n-1} (X - x_j)$. By applying Equation (14.1) to P_i, determine w_i in terms of P_i.
2. Deduce a function `NCRule` which associates to n the coefficients of Newton-Cotes' rule with n points on the interval $[0, n-1]$.
3. Show how to apply these weights to an interval $[a, b]$, with a, b any real numbers.
4. Write a function `QuadNC` which computes the integral of a function given on a segment of $\mathbb{R}$ given as parameter. Compare its results with the integration functions available in Sage on the integrals I_1 to I_{10}.

14.1.2 Multiple Integrals

Let us consider the double integral:

$$I = \int_0^1 \int_0^{\sqrt{y}} \exp(y \sin x)\, \mathrm{d}x\, \mathrm{d}y.$$

We first try to reduce it to a simple integral, by looking for a closed form for the inner integral, which fails in this case:

```
sage: y = var('y'); integrate(exp(y*sin(x)), (x, 0, sqrt(y)))
integrate(e^(y*sin(x)), x, 0, sqrt(y))
```

Since Sage does not provide any functionality for multiple integrals, we rewrite the problem as $I = \int_0^1 f(y)\, \mathrm{d}y$, where f is a Sage function, itself calling a numerical integration method:

```
sage: f = lambda y: numerical_integral(lambda x: exp(y*sin(x)),  \
                                       0, sqrt(y))[0]
sage: f(0.0), f(0.5), f(1.0)
(0.0, 0.8414895067661431, 1.6318696084180513)
```

We can now evaluate the integral of f on $[0, 1]$ numerically:

```
sage: numerical_integral(f, 0, 1)
(0.8606791942204567, 6.301207560882073e-07)
```

We might also use `sage.calculus.calculus.nintegral` to compute f:

```
sage: f = lambda y: sage.calculus.calculus.nintegral(exp(y*sin(x)), \
                                                     x, 0, sqrt(y))[0]
sage: numerical_integral(f, 0, 1)
(0.860679194220456..., 6.301207560882096e-07)
```

or even `mpmath.quad`:

```
sage: f = lambda y: RDF(mpmath.quad(lambda x: mpmath.exp(y*mpmath.sin(x)), \
                                    [0, sqrt(y)]))
sage: numerical_integral(f, 0, 1)
(0.8606791942204567, 6.301207561187562e-07)
```

Note that `mpmath` is able to compute multiple integrals directly, even in arbitrary precision, however only on a rectangular domain:

```
sage: mpmath.mp.dps = 60
sage: f = lambda x, y: mpmath.exp(y*mpmath.sin(x))
sage: mpmath.quad(f, [0,1], [0,1])
mpf('1.2839220575523847175438591764632467574166425032518975110871630 5')
```

Sometimes, in particular when the integrand is expensive to compute, we want an answer with at most (say) n^2 evaluations, even if we get a worse accuracy of the result. Here is a solution using the options `algorithm` and `max_points` of `numerical_integral`:

```
sage: def evalI(n):
....:   f = lambda y: numerical_integral(lambda x: exp(y*sin(x)),
....:                0, sqrt(y), algorithm='qng', max_points=n)[0]
....:   return numerical_integral(f, 0, 1, algorithm='qng', max_points=n)
sage: evalI(100)
(0.8606792028826138, 5.553962923506737e-07)
```

14.2 Solving Ordinary Differential Equations Numerically

In this section, we are interested in solving ordinary differential equations numerically. The available functions in Sage are able to deal with systems of the form:

$$\left\{\begin{array}{rcl} \frac{\mathrm{d}y_1}{\mathrm{d}t}(t) &=& f_1(t, y_1(t), y_2(t), \ldots, y_n(t)) \\ \frac{\mathrm{d}y_2}{\mathrm{d}t}(t) &=& f_2(t, y_1(t), y_2(t), \ldots, y_n(t)) \\ \vdots & & \\ \frac{\mathrm{d}y_n}{\mathrm{d}t}(t) &=& f_n(t, y_1(t), y_2(t), \ldots, y_n(t)) \end{array}\right.$$

with known initial conditions $(y_1(0), y_2(0), \ldots, y_n(0))$.

This setting enables us to solve also problems of order larger than 1, by introducing auxiliary variables (see the detailed example in §14.2.1). It does not however allow to represent the system of equations satisfied by Dickman's ρ function:

$$\left\{\begin{array}{rcll} u\rho'(u) + \rho(u-1) &=& 0 & \text{for } u \geqslant 1, \\ \rho(u) &=& 1 & \text{for } 0 \leqslant u \leqslant 1. \end{array}\right.$$

Indeed, the tools to solve ordinary differential equations are not suited to such an equation (called *with delay*).

The "one-step" numerical methods all use the same general principle: for a given step h and known values of $y(t_0)$ and $y'(t_0)$, we compute an approximation of $y(t_0+h)$ from an estimate of $y'(t)$ taken on the interval $[t_0, t_0+h]$. The simplest method consists in the approximation:

$$\begin{array}{rcl} \forall t \in [t_0, t_0+h], \quad y'(t) &\approx& y'(t_0), \\ \displaystyle\int_{t_0}^{t_0+h} y'(t)\,\mathrm{d}t &\approx& hy'(t_0), \\ y(t_0+h) &\approx& y(t_0) + hy'(t_0). \end{array}$$

The approximation of y' by a constant on $[t_0, t_0+h]$ reminds us of the rectangle quadrature rule. The obtained method is of order 1, i.e., the error made after one computation step is $O(h^2)$, assuming f is regular enough. In general a method is of order p if the error made on a step of width h is $O(h^{p+1})$. The value obtained in $t_1 = t_0 + h$ is used as starting point to make one further step, until the desired target.

This order 1 method, called Euler's method, is not renowned for its accuracy (as the rectangle rule for numerical integration), and some higher order methods

exist, for example the Runge-Kutta method of order 2 to solve the equation $y' = f(t, y)$:

$$\begin{aligned} k_1 &= hf(t_n, y(t_n)) \\ k_2 &= hf(t_n + \frac{1}{2}h, y(t_n) + \frac{1}{2}k_1) \\ y(t_{n+1}) &\approx y(t_n) + k_2 + O(h^3). \end{aligned}$$

In this method, we try to evaluate $y'(t_n + h/2)$ to get a better estimate of $y(t_n + h)$.

Some multi-step methods also exist (for example Gear's method): they consist in computing $y(t_n)$ from the already computed values $y(t_{n-1}), y(t_{n-2}), \ldots, y(t_{n-\ell})$, for a given number ℓ of steps. These methods necessarily require an initial phase, before enough values are available.

Similarly to the Gauss-Kronrod quadrature method, some hybrid methods exist for solving differential equations. For example, the Dormand-Prince method computes with the same approximation points a value at orders 4 and 5, the latter being used to estimate the error made for the former. We say it is an adaptive method.

We also distinguish between explicit and implicit methods: in an explicit method, the value of $y(t_{n+1})$ is given by a formula using known values only; for an implicit method we have to solve an equation. Let us consider for example Euler's implicit method:

$$y(t_{n+1}) = y(t_n) + hf(t_{n+1}, y(t_{n+1})).$$

The wanted value $y(t_{n+1})$ appears on both sides of the equation; if the function f is complex enough, we have to solve a non-linear algebraic system, typically using Newton's method (see §12.2.2).

A priori, we expect more accurate results if we decrease the integration step h; in addition to the extra computations it implies, the expected accuracy gain is however counter-balanced by the increased roundoff errors which, at the end, might be significant with respect to the final result.

14.2.1 An Example

Let us consider the Van der Pol oscillator of parameter μ, satisfying the following differential equation:

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2}(t) - \mu(1 - x^2)\frac{\mathrm{d}x}{\mathrm{d}t}(t) + x(t) = 0.$$

Writing $y_0(t) = x(t)$ and $y_1(t) = \frac{\mathrm{d}x}{\mathrm{d}t}$, we get the order 1 system:

$$\left\{ \begin{array}{lcl} \frac{\mathrm{d}y_0}{\mathrm{d}t} &=& y_1, \\ \frac{\mathrm{d}y_1}{\mathrm{d}t} &=& \mu(1 - y_0^2)y_1 - y_0. \end{array} \right.$$

To solve it, we will use a "solver" object provided by the `ode_solver` command:

```
sage: T = ode_solver()
```

A solver object enables us to register the definition and the parameters of the system we want to solve; it gives access to the numerical tools for solving differential equations from the GSL library, already mentioned for numerical quadrature.

The system equations are given in the form of a function:

```
sage: def f_1(t,y,params): return [y[1],params[0]*(1-y[0]^2)*y[1]-y[0]]
sage: T.function = f_1
```

The parameter `y` represents the vector of unknown functions, and we should return the right-hand side vector of the system, in terms of t and an optional parameter (here `params[0]` which represents μ).

Some of the GSL algorithms require the system Jacobian as well (the matrix whose (i, j) term is $\frac{\partial f_i}{\partial y_j}$, and whose last line contains $\frac{\partial f_j}{\partial t}$):

```
sage: def j_1(t,y,params):
....:     return [[0, 1],
....:             [-2*params[0]*y[0]*y[1]-1, params[0]*(1-y[0]^2)],
....:             [0,0]]
sage: T.jacobian = j_1
```

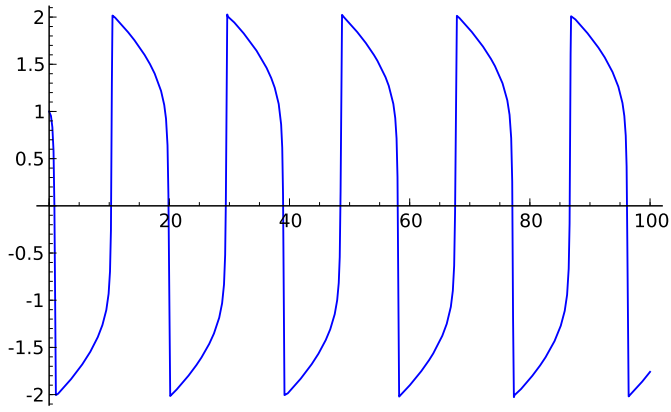
it is now possible to ask for a numerical solution. We choose the algorithm, the interval on which we want the solution, and the number of steps, which determines h:

```
sage: T.algorithm = "rk8pd"
sage: T.ode_solve(y_0=[1,0], t_span=[0,100], params=[10],
....:             num_points=1000)
sage: f = T.interpolate_solution()
```

Here, we have chosen the Runge-Kutta Dormand-Prince algorithm to compute the solution on $[0, 100]$; the initial conditions and the value of the parameters (here one only) are given too: `y_0=[1,0]` means $y_0(0) = 1, y_1(0) = 0$, i.e., $x(0) = 1, x'(0) = 0$.

To show the graph of the solution (we might try `plot(f, 0, 2)` to see the zero derivative in $t = 0$ more clearly):

```
sage: plot(f, 0, 100)
```



14.2.2 Available Functions

We have already mentioned for the solver objects from GSL the `rk8pd` method. Other methods are available:

`rkf45`: Runga-Kutta-Fehlberg, an adaptive method of orders 5 and 4;

`rk2`: adaptive Runge-Kutta of orders 3 and 2;

`rk4`: the classical Runge-Kutta method of order 4;

`rk2imp`: an implicit order 2 Runge-Kutta method with evaluation in the middle of the interval;

`rk4imp`: an implicit order 4 Runge-Kutta method with evaluation at "Gaussian points"[4];

`bsimp`: the implicit Burlisch-Stoer method;

`gear1`: the implicit one-step Gear method;

`gear2`: the implicit two-step Gear method.

For more details on all these methods, we refer the reader to [AP98].

One should note that the GSL limitation to machine floating-point numbers — thus of fixed precision — that we mentioned for the numerical integration also holds for solving differential equations.

Maxima also provides routines to solve differential equations numerically, with its own syntax:

```
sage: t, y = var('t, y')
sage: desolve_rk4(t*y*(2-y), y, ics=[0,1], end_points=[0, 1], step=0.5)
[[0, 1], [0.5, 1.12419127424558], [1.0, 1.461590162288825]]
```

The `desolve_rk4` function uses the order-4 Runge-Kutta method (the same as `rk4` for GSL) and takes as parameters:

- the right-hand side of the equation $y'(t) = f(t, y(t))$, here $y' = ty(2 - y)$;
- the name of the unknown function, here y;
- the initial conditions `ics`, here $t = 0$ and $y = 1$;
- the resolution interval `end_points`, here $[0, 1]$;
- the resolution step, here 0.5.

We omit the similar command `desolve_system_rk4`, already mentioned in Chapter 4, and which applies to a differential system. Maxima is limited to machine precision too.

If we want arbitrary precision solutions, we might use `odefun` from the `mpmath` package:

[4]The roots of the degree-2 Legendre polynomial, which are shifted on the interval $[t, t + h]$, and whose name makes reference to the Gauss-Legendre quadrature rule.

```
sage: import mpmath
sage: mpmath.mp.prec = 53
sage: sol = mpmath.odefun(lambda t, y: y, 0, 1)
sage: sol(1)
mpf('2.7182818284590451')
sage: mpmath.mp.prec = 100
sage: sol(1)
mpf('2.7182818284590452353602874802307')
sage: N(exp(1), 100)
2.7182818284590452353602874714
```

The arguments of the `mpmath.odefun` function are:

- the right-hand sides of the system of equations, in the form of a function $(t, y) \mapsto f(t, y(t))$, here $y' = y$, like for the `ode_solver` function. The dimension of the system is automatically deduced from the dimension of the function return value;
- the initial conditions t_0 and $y(t_0)$, here $y(0) = 1$.

For example for this two-dimensional system

$$\begin{cases} y_1' &= -y_2 \\ y_2' &= y_1 \end{cases}$$

whose solutions are $(\cos(t), \sin(t))$, with initial conditions $y_1(0) = 1$ and $y_2(0) = 0$:

```
sage: mpmath.mp.prec = 53
sage: f = mpmath.odefun(lambda t, y: [-y[1], y[0]], 0, [1, 0])
sage: f(3)
[mpf('-0.98999249660044542'), mpf('0.14112000805986721')]
sage: (cos(3.), sin(3.))
(-0.989992496600445, 0.141120008059867)
```

The `mpmath.odefun` function relies on Taylor's method. For degree p it uses:

$$y(t_{n+1}) = y(t_n) + h\frac{\mathrm{d}y}{\mathrm{d}t}(t_n) + \frac{h^2}{2!}\frac{\mathrm{d}^2y}{\mathrm{d}t^2}(t_n) + \ldots + \frac{h^p}{p!}\frac{\mathrm{d}^py}{\mathrm{d}t^p}(t_n) + O(h^{p+1}).$$

The main question is the computation of the derivatives of y. For this purpose, `odefun` computes approximate values

$$[\widetilde{y}(t_n + h), \ldots, \widetilde{y}(t_n + ph)] \approx [y(t_n + h), \ldots, y(t_n + ph)]$$

using p steps of the less precise method of Euler. We then compute

$$\widetilde{\frac{\mathrm{d}y}{\mathrm{d}t}}(t_n) \approx \frac{\widetilde{y}(t_n + h) - \widetilde{y}(t_n)}{h}, \quad \widetilde{\frac{\mathrm{d}y}{\mathrm{d}t}}(t_n + h) \approx \frac{\widetilde{y}(t_n + 2h) - \widetilde{y}(t_n + h)}{h}$$

then

$$\widetilde{\frac{\mathrm{d}^2y}{\mathrm{d}t^2}}(t_n) \approx \frac{\widetilde{\frac{\mathrm{d}y}{\mathrm{d}t}}(t_n + h) - \widetilde{\frac{\mathrm{d}y}{\mathrm{d}t}}(t_n)}{h},$$

and so on until we obtain estimates of the derivative of y in t_n up to order p.

Care must be taken when we change the floating-point precision of `mpmath`. To illustrate this problem, let us consider again the differential equation $y' = y$ seen above, satisfied by the exp function:

```
sage: mpmath.mp.prec = 10
sage: sol = mpmath.odefun(lambda t, y: y, 0, 1)
sage: sol(1)
mpf('2.7148')
sage: mpmath.mp.prec = 100
sage: sol(1)
mpf('2.7135204235459511323824699502438')
```

The last approximation of exp(1) is quite bad, albeit being computed with 100 bits of precision! The solution function `sol` (an "interpolator" in the `mpmath` jargon) has been computed with 10 bits of precision only, and its coefficients are not recomputed when the precision is changed, which explains the result.

Part IV

Combinatorics

15

Enumeration and Combinatorics

This chapter mainly covers the treatment in Sage of the following combinatorial problems: enumeration (how many elements are there in a set S?), listing (generate all elements of S, or iterate through them), and random selection (choosing an element at random from a set S according to a given distribution, for example the uniform distribution). These questions arise naturally in the calculation of probabilities (what is the probability in poker of obtaining a straight or a four-of-a-kind of aces?), in statistical physics, and also in computer algebra (the number of elements in a finite field), or in the analysis of algorithms. Combinatorics covers a much wider domain (partial orders, representation theory...) for which we only give a few pointers towards the possibilities offered by Sage. Graphs are treated in Chapter 16.

A characteristic of computational combinatorics is the profusion of types of objects and sets that one wants to manipulate. It would be impossible to describe them all or, a fortiori, to implement them all. After some examples (§15.1), this chapter illustrates the underlying method: supplying the basic building blocks for describing common combinatorial sets §15.2, tools for combining them to construct new examples §15.3, and generic algorithms for solving uniformly a large class of problems §15.4. On a first reading, this chapter can be skipped through quickly, pausing at the summaries of the sections §15.1.2 and §15.3.

This is a domain in which Sage has much more extensive capabilities than most computer algebra systems, and it is rapidly expanding; at the same time, it is still quite new, and has many unnecessary limitations and inconsistencies.

15.1 Initial Examples

15.1.1 Poker and Probability

We begin by solving a classic problem: enumerating certain combinations of cards in the game of poker, in order to deduce their probability.

A card in a poker deck is characterised by a suit (hearts, diamonds, spades, or clubs) and a value (2, 3, ..., 10, jack, queen, king, ace). The game is played with a full deck, which is the Cartesian product of the set of suits and the set of values:

$$\text{Cards} = \text{Suits} \times \text{Values} = \{(s, v) \mid s \in \text{Suits} \text{ and } v \in \text{Values}\}.$$

We construct these examples in Sage:

```
sage: Suits = Set(["Hearts", "Diamonds", "Spades", "Clubs"])
sage: Values = Set([2, 3, 4, 5, 6, 7, 8, 9, 10,
....:               "Jack", "Queen", "King", "Ace"])
sage: Cards = cartesian_product([Values, Suits])
```

There are 4 suits and 13 possible values, and therefore $4 \times 13 = 52$ cards in the poker deck:

```
sage: Suits.cardinality()
4
sage: Values.cardinality()
13
sage: Cards.cardinality()
52
```

Draw a card at random:

```
sage: Cards.random_element()
(6, 'Clubs')
```

Draw two cards at random:

```
sage: Set([Cards.random_element(), Cards.random_element()])
{(2, 'Hearts'), (4, 'Spades')}
```

Returning to our main topic, we will be considering a simplified version of poker, in which each player directly draws five cards, which form his *hand*. The cards are all distinct and the order in which they are drawn is irrelevant; a hand is therefore a subset of size 5 of the set of cards. To draw a hand at random, we first construct the set of all possible hands, and then we ask for a randomly chosen element:

```
sage: Hands = Subsets(Cards, 5)
sage: Hands.random_element()
{(4, 'Hearts'), (9, 'Diamonds'), (8, 'Spades'),
 (9, 'Clubs'), (7, 'Hearts')}
```

The total number of hands is given by the number of subsets of size 5 of a set of size 52, which is given by the binomial coefficient $\binom{52}{5}$:

```
sage: binomial(52, 5)
2598960
```

One can also ignore the method of calculation, and simply ask for the size of the set of hands:

```
sage: Hands.cardinality()
2598960
```

The strength of a poker hand depends on the particular combination of cards present. One such combination is the *flush*; this is a hand all of whose cards have the same suit. (In principle, straight flushes should be excluded; this will be the goal of an exercise given below.) Such a hand is therefore characterised by the choice of five values from among the thirteen possibilities, and the choice of one of four suits. We will construct the set of all flushes, so as to determine how many there are:

```
sage: Flushes = cartesian_product([Subsets(Values, 5), Suits])
sage: Flushes.cardinality()
5148
```

The probability of obtaining a flush when drawing a hand at random is therefore:

```
sage: Flushes.cardinality()  / Hands.cardinality()
33/16660
```

or about two in a thousand:

```
sage: 1000.0 * Flushes.cardinality()  / Hands.cardinality()
1.98079231692677
```

We will now attempt a little numerical simulation. The following function tests whether a given hand is a flush or not:

```
sage: def is_flush(hand):
....:     return len(set(suit for (val, suit) in hand)) == 1
```

We now draw 10000 hands at random, and count the number of flushes obtained. (This takes about 10 seconds.)

```
sage: n = 10000
sage: nflush = 0
sage: for i in range(n):
....:     hand = Hands.random_element()
....:     if is_flush(hand):
....:         nflush += 1
sage: print(n, nflush)
10000, 18
```

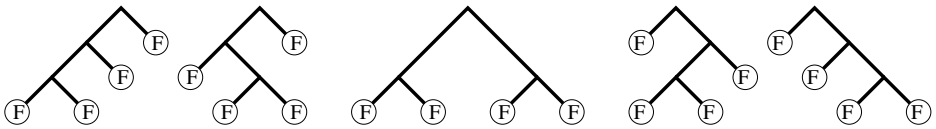


FIGURE 15.1 – The five complete binary trees with four leaves.

Exercise 50. A hand containing four cards of the same value is called a *four of a kind*. Construct the set of four of a kind hands. (Hint: use `Arrangements` to choose a pair of distinct values at random, then choose a suit for the first value.) Calculate the number of four of a kind hands, list them, and then determine the probability of obtaining a four of a kind when drawing a hand at random.

Exercise 51. A hand all of whose cards have the same suit, is called a *straight flush* if those values are consecutive, otherwise it is called a *flush*. Count the number of straight flushes, and then deduce the correct probability of obtaining a flush when drawing a hand at random.

Exercise 52. Calculate the probability of each of the poker hands (see http://en.wikipedia.org/wiki/Poker_hands), and compare them with the results of simulations.

15.1.2 Enumeration of Trees Using Generating Functions

In this section, we discuss the example of complete binary trees, and illustrate in this context many techniques of enumeration in which formal power series play a natural role. These techniques are quite general, and can be applied whenever the combinatorial objects in question admit a recursive definition (grammar) (see §15.4.3 for an automated treatment). The goal is not a formal presentation of these methods; the calculations are rigorous, but most of the justifications will be skipped.

A *complete binary tree* is either a leaf F, or a node to which two complete binary trees are attached (see Figure 15.1).

Exercise 53. Find by hand all complete binary trees with $n = 1, 2, 3, 4, 5$ leaves (see Exercise 61 to find them using Sage).

Our goal is to determine the number c_n of complete binary trees with n leaves (in this section, except when explicitly stated otherwise, "trees" always means complete binary trees). This is a typical situation where one is not only interested in a single set, but in a family of sets, typically parameterised by $n \in \mathbb{N}$.

According to the solution of Exercise 53, the first terms are given by $c_1, \ldots, c_5 = 1, 1, 2, 5, 14$. The simple fact of knowing these few numbers is already very valuable. In fact, this permits research in a gold mine of information: the *Online Encyclopedia of Integer Sequences* http://oeis.org/ (commonly called "Sloane", the name of its principal author), which contains more than 282892 sequences of integers:

```
sage: oeis([1,1,2,5,14])
0: A000108: Catalan numbers: C(n) = binomial(2n,n)/(n+1) = (2n)!/(n!(n
    +1)!). Also called Segner numbers.
1: A120588: G.f. satisfies: 3*A(x) = 2 + x + A(x)^2, with a(0) = 1.
```

```
2: A080937: Number of Catalan paths (nonnegative, starting and ending at
     0, step +/-1) of 2*n steps with all values <= 5.
```

The result suggests that the trees are counted by one of the most famous sequences, the Catalan numbers. Looking through the references supplied by the Encyclopedia, we see that this is really the case: the few numbers above form a digital fingerprint of our objects, which enables us to find, in a few seconds, a precise result from within an abundant literature.

Enumeration Using Generating Series. Our next goal is to recover this result using Sage. Let C_n be the set of trees with n leaves, so that $c_n = |C_n|$; by convention, we will define $C_0 = \emptyset$ and $c_0 = 0$. The set of all trees is then the disjoint union of the sets C_n:

$$C = \biguplus_{n\in\mathbb{N}} C_n \,.$$

Having named the set C of all trees, we can translate the recursive definition of trees into a set-theoretic equation:

$$C \quad \approx \quad \{\mathrm{L}\} \quad \uplus \quad C \times C \,.$$

In words: a tree t (which is by definition in C) is either a leaf (so in $\{\mathrm{L}\}$) or a node to which two trees t_1 and t_2 have been attached, and which we can therefore identify with the pair (t_1, t_2) (in the Cartesian product $C \times C$).

The founding idea of algebraic combinatorics, introduced by Euler in a letter to Goldbach of 1751 to treat a similar problem, is to manipulate all numbers c_n simultaneously, by encoding them as coefficients in a formal power series, called the *generating function* of the c_n's:

$$C(z) = \sum_{n\in\mathbb{N}} c_n z^n \,,$$

where z is a formal variable (which means that we do not have to worry about questions of convergence). The beauty of this idea is that set-theoretic operations ($A \uplus B$, $A \times B$) translate naturally into algebraic operations on the corresponding series ($A(z) + B(z)$, $A(z) \cdot B(z)$), in such a way that the set-theoretic equation satisfied by C can be translated directly into an algebraic equation satisfied by $C(z)$:

$$C(z) = z + C(z) \cdot C(z).$$

Now we can solve this equation with Sage. In order to do so, we introduce two variables, C and z, and we define the equation:

```
sage: C, z = var('C, z'); sys = [ C == z + C*C ]
```

There are two solutions, which happen to have closed forms:

```
sage: sol = solve(sys, C, solution_dict=True); sol
[{C: -1/2*sqrt(-4*z + 1) + 1/2}, {C: 1/2*sqrt(-4*z + 1) + 1/2}]
sage: s0 = sol[0][C]; s1 = sol[1][C]
```

and whose Taylor series begin as follows:

```
sage: s0.series(z, 6)
1*z + 1*z^2 + 2*z^3 + 5*z^4 + 14*z^5 + Order(z^6)
sage: s1.series(z, 6)
1 + (-1)*z + (-1)*z^2 + (-2)*z^3 + (-5)*z^4 + (-14)*z^5 + Order(z^6)
```

The second solution is clearly nonsensical, while the first one gives the expected coefficients. Therefore, we set:

```
sage: C = s0
```

We can now calculate the next terms:

```
sage: C.series(z, 11)
1*z + 1*z^2 + 2*z^3 + 5*z^4 + 14*z^5 + 42*z^6 +
132*z^7 + 429*z^8 + 1430*z^9 + 4862*z^10 + Order(z^11)
```

or calculate, more or less instantaneously, the 100-th coefficient:

```
sage: C.series(z, 101).coefficient(z,100)
227508830794229349661819540395688853956041682601541047340
```

It is unfortunate to have to recalculate everything if at some point we wanted the 101-st coefficient. Lazy power series (see §7.5.3) come into their own here, in that one can define them from a system of equations without solving it, and, in particular, without needing a closed form for the answer. We begin by defining the ring of lazy power series:

```
sage: L.<z> = LazyPowerSeriesRing(QQ)
```

Then we create a "free" power series, which we name, and which we then define by a recursive equation:

```
sage: C = L()
sage: C._name = 'C'
sage: C.define( z + C * C )
```

```
sage: [C.coefficient(i) for i in range(11)]
[0, 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]
```

At any point, one can ask for any coefficient without having to redefine C:

```
sage: C.coefficient(100)
227508830794229349661819540395688853956041682601541047340
```

```
sage: C.coefficient(200)
129013158064429114001222907669676675134349530552728882499 8
108515989014190133483190455345808508477355282757501221889 40
```

Recurrence Relation and Closed-Form Formula. We now return to the closed form of $C(z)$:

```
sage: z = var('z'); C = s0; C
-1/2*sqrt(-4*z + 1) + 1/2
```

The n-th coefficient in the Taylor series for $C(z)$ being given by $\frac{1}{n!}C(z)^{(n)}(0)$, we look at the successive derivatives $C(z)^{(n)}(z)$:

```
sage: derivative(C, z, 1)
1/sqrt(-4*z + 1)
sage: derivative(C, z, 2)
2/(-4*z + 1)^(3/2)
sage: derivative(C, z, 3)
12/(-4*z + 1)^(5/2)
```

This suggests the existence of a simple explicit formula, which we now seek. The following small function returns $d_n = n!\,c_n$:

```
sage: def d(n): return derivative(C, n).subs(z=0)
```

Taking successive quotients:

```
sage: [ (d(n+1) / d(n)) for n in range(1,17) ]
[2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62]
```

we observe that d_n satisfies the recurrence relation $d_{n+1} = (4n-2)d_n$, from which we deduce that c_n satisfies the recurrence relation $c_{n+1} = \frac{(4n-2)}{n+1}c_n$. Simplifying, we find that c_n is the $(n-1)$-th Catalan number:

$$c_n = \text{Catalan}(n-1) = \frac{1}{n}\binom{2(n-1)}{n-1}.$$

We check this:

```
sage: def c(n): return 1/n*binomial(2*(n-1),n-1)
sage: [c(k) for k in range(1, 11)]
[1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]
sage: [catalan_number(k-1) for k in range(1, 11)]
[1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]
```

We can now calculate coefficients much further; here we calculate c_{100000} which has more than 60000 digits:

```
sage: %time cc = c(100000)
CPU times: user 2.34 s, sys: 0.00 s, total: 2.34 s
Wall time: 2.34 s
sage: ZZ(cc).ndigits()
60198
```

Systematic Treatment by Algebraic-Differential Equations. The methods that we have used generalise to all recursively defined objects: the system of set-theoretic equations can be translated into a system of equations on the generating function, which enables the recursive calculation of its coefficients. If the set-theoretic equations are simple enough (for example, if they only involve Cartesian products and disjoint unions), the equation for $C(z)$ is algebraic. This equation has, in general, no closed-form solution. However, using *confinement*, one can deduce a *linear* differential equation that $C(z)$ satisfies. This differential equation, in turn, can be translated into a recurrence relation of fixed length on its coefficients c_n. In this case, the series is called *D-finite*. After the initial calculation of this recurrence relation, the calculation of coefficients is very fast. All these steps are purely algorithmic, and it is planned to port into Sage the implementations that exist in Maple (the gfun and combstruct packages) or MuPAD-Combinat (the decomposableObjects library).

For the moment, we illustrate this general procedure in the case of complete binary trees. The generating function $C(z)$ is a solution to an algebraic equation $P(z, C(z)) = 0$, where $P = P(x, y)$ is a polynomial with coefficients in $\mathbb{Q}$. In the present case, $P = y^2 - y + x$. We formally differentiate this equation with respect to z:

```
sage: x, y, z = var('x, y, z')
sage: P = function('P')(x, y); C = function('C')(z)
sage: equation = P(x=z, y=C) == 0
sage: diff(equation, z)
diff(C(z), z)*D[1](P)(z, C(z)) + D[0](P)(z, C(z)) == 0
```

or, in a more readable format,

$$\frac{dC(z)}{dz}\frac{\partial P}{\partial y}(z, C(z)) + \frac{\partial P}{\partial x}(z, C(z)) = 0.$$

From this we deduce:

$$\frac{dC(z)}{dz} = -\frac{\frac{\partial P}{\partial x}}{\frac{\partial P}{\partial y}}(z, C(z))\,.$$

In the case of complete binary trees, this gives:

```
sage: P = y^2 - y + x; Px = diff(P, x); Py = diff(P, y)
sage: - Px / Py
-1/(2*y - 1)
```

Recall that $P(z, C(z)) = 0$. Thus, we can calculate this fraction mod P and, in this way, express the derivative of $C(z)$ as a *polynomial in* $C(z)$ *with coefficients in* $\mathbb{Q}(z)$. In order to achieve this, we construct the quotient ring $R = \mathbb{Q}(x)[y]/(P)$:

```
sage: Qx = QQ['x'].fraction_field(); Qxy = Qx['y']
sage: R = Qxy.quo(P); R
Univariate Quotient Polynomial Ring in ybar
over Fraction Field of Univariate Polynomial Ring in x
over Rational Field with modulus y^2 - y + x
```

Note: `ybar` is the name of the variable y in the quotient ring; for more information on quotient rings, see §7.2.2. We continue the calculation of this fraction in R:

```
sage: fraction = - R(Px) / R(Py); fraction
(1/2/(x - 1/4))*ybar - 1/4/(x - 1/4)
```

We lift the result to $\mathbb{Q}(x)[y]$ and then substitute z and $C(z)$ to obtain an expression for $\frac{d}{dz}C(z)$:

```
sage: fraction = fraction.lift(); fraction
(1/2/(x - 1/4))*y - 1/4/(x - 1/4)
sage: fraction(x=z, y=C)
2*C(z)/(4*z - 1) - 1/(4*z - 1)
```

or, more legibly,

$$\frac{\partial C(z)}{\partial z} = \frac{1}{1-4z} - \frac{2}{1-4z}C(z)\,.$$

In this simple case, we can directly deduce from this expression a linear differential equation with coefficients in $\mathbb{Q}[z]$:

```
sage: equadiff = diff(C,z) == fraction(x=z, y=C); equadiff
diff(C(z), z) == 2*C(z)/(4*z - 1) - 1/(4*z - 1)
sage: equadiff = equadiff.simplify_rational()
sage: equadiff = equadiff * equadiff.rhs().denominator()
sage: equadiff = equadiff - equadiff.rhs()
sage: equadiff
(4*z - 1)*diff(C(z), z) - 2*C(z) + 1 == 0
```

or, more legibly,

$$(1-4z)\frac{\partial C(z)}{\partial z} + 2C(z) - 1 = 0\,.$$

It is trivial to verify this equation on the closed form:

```
sage: Cf = sage.symbolic.function_factory.function('C')
sage: bool(equadiff.substitute_function(Cf, lambda z: s0(z=z)))
True
```

In the general case, one continues to calculate successive derivatives of $C(z)$. These derivatives are *confined* to the quotient ring $\mathbb{Q}(z)[C]/(P)$, which is of finite dimension $\deg P$ over $\mathbb{Q}(z)$. Therefore, one will eventually find a linear relation among the first $\deg P$ derivatives of $C(z)$. Putting it over a single denominator, we obtain a linear differential equation of degree $\leq \deg P$ with coefficients in $\mathbb{Q}[z]$. By extracting the coefficient of z^n in the differential equation, we obtain the desired recurrence relation on the coefficients; in this case we recover the relation we had already found, based on the closed form:

$$c_{n+1} = \frac{4n-2}{n+1}c_n.$$

After fixing the correct initial conditions, it becomes possible to calculate the coefficients of $C(z)$ recursively:

```
sage: def C(n): return n if n <= 1 else (4*n-6)/n * C(n-1)
sage: [ C(i) for i in range(10) ]
[0, 1, 1, 2, 5, 14, 42, 132, 429, 1430]
```

If n is too large for the explicit calculation of c_n, a sequence asymptotically equivalent to the sequence of coefficients c_n may be sought. Here again, there are generic techniques. The central tool is complex analysis, specifically, the study of the generating function around its singularities. In the present instance, the singularity is at $z_0 = 1/4$ and one would obtain $c_n \sim \frac{4^{n-1}}{n^{3/2}\sqrt{\pi}}$.

Summary. We see here a general phenomenon of computer algebra: the best *data structure* to describe a complicated mathematical object (a real number, a sequence, a formal power series, a function, a set) is often an equation defining the object (or a system of equations, typically with some initial conditions). Attempting to find a closed-form solution to this equation is not necessarily of interest: on the one hand, such a closed form rarely exists (e.g., the problem of solving a polynomial by radicals), and on the other hand, the equation, in itself, contains all necessary information to calculate algorithmically the properties of the object under consideration (e.g., a numerical approximation, the initial terms or elements, an asymptotic equivalent), or to calculate with the object itself (e.g., performing arithmetic on power series). Therefore, instead of solving the equation, we look for the equation that describes the object and is best suited to the problem we want to solve; see also §2.2.2.

As we saw in our example, confinement (for example, in a finite dimensional vector space) is a fundamental tool for studying such equations. This notion of confinement is widely applicable in elimination techniques (linear algebra, Gröbner bases, and their algebraic-differential generalisations). The same tool is central in algorithms for automatic summation and automatic verification of identities (Gosper's algorithm, Zeilberger's algorithm, and their generalisations [PWZ96]; see also the examples in §2.3.1 and Exercise 56).

All these techniques and their many generalisations are at the heart of very active topics of research: automatic combinatorics and analytic combinatorics, with major applications in the analysis of algorithms [FS09]. It is likely, and desirable, that they will be progressively implemented in Sage.

15.2 Common Enumerated Sets

15.2.1 First Example: Subsets of a Set

Fix a set E of size n and consider the subsets of E of size k. We know that these subsets are counted by the binomial coefficients $\binom{n}{k}$. We can therefore calculate the number of subsets of size $k = 2$ of $E = \{1, 2, 3, 4\}$ with the function `binomial`:

```
sage: binomial(4, 2)
6
```

Alternatively, we can *construct* the set $\mathcal{P}_2(E)$ of all subsets of size 2 of E, then ask for its cardinality:

```
sage: S = Subsets([1,2,3,4], 2); S.cardinality()
6
```

Once `S` has been constructed, we can also obtain the list of its elements, select an element at random, or request a typical element:

```
sage: S.list()
[{1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4}]
sage: S.random_element()
{1, 4}
sage: S.an_element()
{2, 3}
```

More precisely, the object `S` models the set $\mathcal{P}_2(E)$ equipped with a fixed order (here, lexicographic order). It is therefore possible to ask for its 5-th element, keeping in mind that, as with Python lists, the first element is numbered zero. (As a shortcut, in this setting, one can also use the notation `S[.]`.)

```
sage: S.unrank(4)
{2, 4}
sage: S[4]
{2, 4}
```

This should be used with care because some sets have a natural indexing other than by $(0, \dots)$.

Conversely, one can calculate the position of an object in this order:

```
sage: s = S([2,4]); S.rank(s)
4
```

Note that `S` is *not* the list of its elements. One can, for example, model the set $\mathcal{P}(\mathcal{P}(\mathcal{P}(E)))$ and calculate its cardinality ($2^{2^{2^4}}$):

```
sage: E = Set([1,2,3,4])
sage: S = Subsets(Subsets(Subsets(E))); S.cardinality()
2003529930406846464979072351560255750447825475569751419265016...736
```

which is roughly $2 \cdot 10^{19728}$:

```
sage: S.cardinality().ndigits()
19729
```

or ask for its 237102124-th element:

```
sage: S.unrank(237102123)
{{{2, 4}, {1, 4}, {}, {1, 3, 4}, {1, 2, 4}, {4}, {2, 3}, {1, 3}, {2}},
 {{1, 3}, {2, 4}, {1, 2, 4}, {}, {3, 4}}}
```

It would be physically impossible to construct explicitly all elements of S, as there are many more of them than there are particles in the universe (estimated at 10^{82}).

Remark: it would be natural in Python to use `len(S)` to ask for the cardinality of `S`. This is not possible because Python requires that the result of `len` be an integer of type `int`; this could cause overflows, and would not permit the return of `Infinity` for infinite sets.

```
sage: len(S)
Traceback (most recent call last):
...
OverflowError: Python int too large to convert to C long
```

15.2.2 Integer Partitions

We now consider another classic problem: given a positive integer n, in how many ways can it be written as a sum $n = i_1 + i_2 + \cdots + i_\ell$ of positive integers? There are two cases to distinguish:

- the order of the elements in the sum is not important, in which case we call $(i_1, \ldots, i_\ell)$ a *partition* of n;
- the order of the elements in the sum is important, in which case we call $(i_1, \ldots, i_\ell)$ a *composition* of n.

We begin with the partitions of $n = 5$; as before, we first construct the set of these partitions:

```
sage: P5 = Partitions(5); P5
Partitions of the integer 5
```

then we ask for its cardinality:

```
sage: P5.cardinality()
7
```

We look at these 7 partitions; the order being irrelevant, the entries are ordered, by convention, in decreasing order.

```
sage: P5.list()
[[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1], [2, 1, 1, 1],
 [1, 1, 1, 1, 1]]
```

The calculation of the number of partitions uses the Rademacher formula (see http://en.wikipedia.org/wiki/Partition_(number_theory)), implemented in `C` and highly optimised, which makes it very fast:

```
sage: Partitions(100000).cardinality()
2749351056977569651267751632098635268817342931598005475820312598430214
7328114964173055050741660736621590157844774296248940493063070200461792
7644930335101160793424571901557189435097253124661084520063695589344642
4871682878983218234500926285383140459702130713067451062441922731123899
9702284408609370935531629697851569569892196108480158600569421098519
```

Partitions of integers are combinatorial objects naturally equipped with many operations. They are therefore returned as objects that are richer than simple lists.

```
sage: P7 = Partitions(7); p = P7.unrank(5); p
[4, 2, 1]
```

```
sage: type(p)
<class 'sage.combinat.partition.Partitions_n_with_category.element_class
    '>
```

For example, they can be represented graphically by a Ferrers diagram:

```
sage: print(p.ferrers_diagram())
****
**
*
```

We leave it to the user to explore by introspection the available operations.

Note that we can also construct a partition directly by:

```
sage: Partition([4,2,1])
[4, 2, 1]
sage: P7([4,2,1])
[4, 2, 1]
```

If one wants to restrict the possible values of the parts $i_1, \ldots, i_\ell$ of the partition, as, for example, when giving change, one can use `WeightedIntegerVectors`. For example, the following calculation:

```
sage: WeightedIntegerVectors(8, [2,3,5]).list()
[[0, 1, 1], [1, 2, 0], [4, 0, 0]]
```

shows that to make 8 dollars using \$2, \$3, and \$5 bills, one can use a \$3 and a \$5, or a \$2 and two \$3's, or four \$2's.

Compositions of integers are manipulated in the same way:

```
sage: C5 = Compositions(5); C5
Compositions of 5
sage: C5.cardinality()
16
sage: C5.list()
[[1, 1, 1, 1, 1], [1, 1, 1, 2], [1, 1, 2, 1], [1, 1, 3],
 [1, 2, 1, 1], [1, 2, 2], [1, 3, 1], [1, 4], [2, 1, 1, 1],
 [2, 1, 2], [2, 2, 1], [2, 3], [3, 1, 1], [3, 2], [4, 1], [5]]
```

The number 16 above seems significant and suggests the existence of a formula. We look at the number of compositions of n ranging from 0 to 9:

```
sage: [ Compositions(n).cardinality() for n in range(10) ]
[1, 1, 2, 4, 8, 16, 32, 64, 128, 256]
```

Similarly, if we consider the number of compositions of 5 by length, we find a line of Pascal's triangle:

```
sage: x = var('x'); sum( x^len(c) for c in C5 )
x^5 + 4*x^4 + 6*x^3 + 4*x^2 + x
```

The above example uses a functionality that we have not seen yet: `C5` being iterable, it can be used like a list in a `for` loop or a comprehension (§15.2.4).

Exercise 54. Prove the formulas suggested by the above examples for the number of compositions of n and the number of compositions of n of length k; investigate by introspection whether Sage uses these formulas for calculating cardinalities.

15.2.3 Some Other Finite Enumerated Sets

Essentially, the principle is the same for all finite sets with which one wants to do combinatorics in Sage; begin by constructing an object that models this set, and then supply appropriate methods, following a uniform interface[1]. We now give a few more typical examples.

Intervals of integers:

```
sage: C = IntegerRange(3, 21, 2); C
{3, 5, ..., 19}
sage: C.cardinality()
9
sage: C.list()
[3, 5, 7, 9, 11, 13, 15, 17, 19]
```

Permutations:

```
sage: C = Permutations(4); C
Standard permutations of 4
sage: C.cardinality()
24
sage: C.list()
[[1, 2, 3, 4], [1, 2, 4, 3], [1, 3, 2, 4], [1, 3, 4, 2],
 [1, 4, 2, 3], [1, 4, 3, 2], [2, 1, 3, 4], [2, 1, 4, 3],
 [2, 3, 1, 4], [2, 3, 4, 1], [2, 4, 1, 3], [2, 4, 3, 1],
 [3, 1, 2, 4], [3, 1, 4, 2], [3, 2, 1, 4], [3, 2, 4, 1],
 [3, 4, 1, 2], [3, 4, 2, 1], [4, 1, 2, 3], [4, 1, 3, 2],
 [4, 2, 1, 3], [4, 2, 3, 1], [4, 3, 1, 2], [4, 3, 2, 1]]
```

Set partitions:

```
sage: C = SetPartitions([1,2,3]); C
Set partitions of {1, 2, 3}
sage: C.cardinality()
5
sage: C.list()
```

[1] Or at least that should be the case; there are still many corners to clean up.

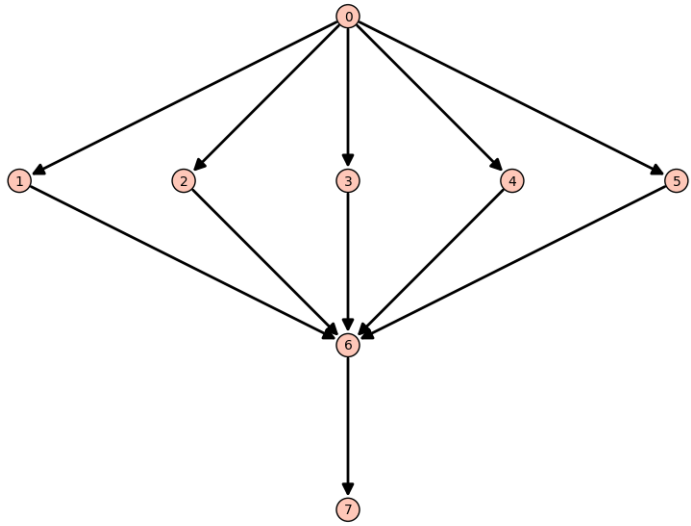


FIGURE 15.2 – A poset on 8 vertices.

```
[{{1, 2, 3}}, {{1}, {2, 3}}, {{1, 3}, {2}}, {{1, 2}, {3}}, {{1}, {2},
    {3}}]
```

Partial orders (posets) on a set of 8 elements, up to isomorphism:

```
sage: C = Posets(8); C
Posets containing 8 elements
sage: C.cardinality()
16999
```

Let us draw one of these posets (see Figure 15.2):

```
sage: show(C.unrank(20))
```

One can iterate through all graphs up to isomorphism. For example, there are 34 simple graphs with 5 vertices (Figure 15.3):

```
sage: len(list(graphs(5)))
34
```

Here is how to draw all those with at most 4 edges (see Figure 15.3):

```
sage: for g in graphs(5, lambda G: G.size() <= 4):
....:     show(g)
```

However, the *set* `C` of these graphs is *not yet* available in Sage; as a result, the following commands are not yet implemented:

```
sage: C = Graphs(5); C.cardinality()
34
sage: Graphs(19).cardinality()
```

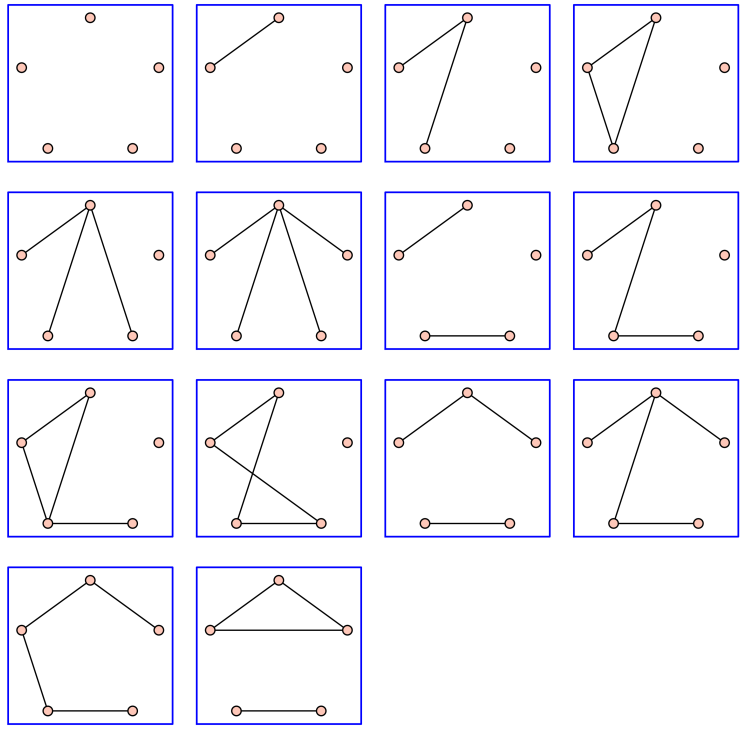


FIGURE 15.3 – The simple graphs with 5 vertices and at most 4 edges.

```
2463780925312500452438300749143276 8
sage: Graphs(19).random_element()
Graph on 19 vertices
```

What we have seen so far also applies, in principle, to finite algebraic structures like the dihedral groups:

```
sage: G = DihedralGroup(4); G
Dihedral group of order 8 as a permutation group
sage: G.cardinality()
8
sage: G.list()
[(), (1,4)(2,3), (1,2,3,4), (1,3)(2,4), (1,3), (2,4), (1,4,3,2), (1,2)
    (3,4)]
```

or the algebra of 2×2 matrices over the finite field $\mathbb{Z}/2\mathbb{Z}$:

```
sage: C = MatrixSpace(GF(2), 2); C.list()
[
[0 0]  [1 0]  [0 1]  [0 0]  [0 0]  [1 1]  [1 0]  [1 0]  [0 1]
[0 0], [0 0], [0 0], [1 0], [0 1], [0 0], [1 0], [0 1], [1 0],
```

```
[0 1]  [0 0]  [1 1]  [1 1]  [1 0]  [0 1]  [1 1]
[0 1], [1 1], [1 0], [0 1], [1 1], [1 1], [1 1]
]
```

```
sage: C.cardinality()
16
```

Exercise 55. List all monomials of degree 5 in three variables (see `IntegerVectors`). Manipulate the ordered set partitions `OrderedSetPartitions` and standard tableaux (`StandardTableaux`).

Exercise 56. List the alternating sign matrices of size 3, 4, and 5, and try to guess the definition (see `AlternatingSignMatrices`). The discovery and proof of the formula for the enumeration of these matrices (see the method `cardinality`), motivated by calculations of determinants in physics, is quite a story. In particular, the first proof, given by Zeilberger in 1992 was automatically produced by a computer program. It was 84 pages long, and required nearly a hundred people to verify it [Zei96].

Exercise 57. Calculate by hand the number of vectors in $(\mathbb{Z}/2\mathbb{Z})^5$, and the number of matrices in $\mathrm{GL}_3(\mathbb{Z}/2\mathbb{Z})$ (that is to say, the number of invertible 3×3 matrices with coefficients in $\mathbb{Z}/2\mathbb{Z}$). Verify your answer with Sage. Generalise to $\mathrm{GL}_n(\mathbb{Z}/q\mathbb{Z})$.

15.2.4 Set Comprehension and Iterators

We will now show some of the possibilities offered by Python for constructing (and iterating through) sets, with a notation that is flexible and close to usual mathematical usage, and in particular the benefits this yields in combinatorics.

We begin by constructing the finite set $\{i^2 \mid i \in \{1, 3, 7\}\}$:

```
sage: [ i^2 for i in [1, 3, 7] ]
[1, 9, 49]
```

and then the same set, but with i running from 1 to 9:

```
sage: [ i^2 for i in range(1,10) ]
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

A Python construction of this form is called *set comprehension*. A clause can be added to keep only those elements with i prime:

```
sage: [ i^2 for i in range(1,10) if is_prime(i) ]
[4, 9, 25, 49]
```

Combining more than one set comprehension, it is possible to construct the set $\{(i, j) \mid 1 \leq j < i < 6\}$:

```
sage: [ (i,j) for i in range(1,6) for j in range(1,i) ]
[(2, 1), (3, 1), (3, 2), (4, 1), (4, 2), (4, 3),
 (5, 1), (5, 2), (5, 3), (5, 4)]
```

or to produce Pascal's triangle:

```
sage: [[binomial(n, i) for i in range(n+1)] for n in range(10)]
```

```
[[1],
 [1, 1],
 [1, 2, 1],
 [1, 3, 3, 1],
 [1, 4, 6, 4, 1],
 [1, 5, 10, 10, 5, 1],
 [1, 6, 15, 20, 15, 6, 1],
 [1, 7, 21, 35, 35, 21, 7, 1],
 [1, 8, 28, 56, 70, 56, 28, 8, 1],
 [1, 9, 36, 84, 126, 126, 84, 36, 9, 1]]
```

The execution of a set comprehension is accomplished in two steps; first an *iterator* is constructed, and then a list is filled with the elements successively produced by the iterator. Technically, an *iterator* is an object with a method `next` that returns a new value each time it is called, until it is exhausted. For example, the following iterator `it`:

```
sage: it = (binomial(3, i) for i in range(4))
```

returns successively the binomial coefficients $\binom{3}{i}$ with $i = 0, 1, 2, 3$:

```
sage: it.next()
1
sage: it.next()
3
sage: it.next()
3
sage: it.next()
1
```

When the iterator is finally exhausted, an exception is raised:

```
sage: it.next()
Traceback (most recent call last):
  ...
StopIteration
```

More generally, an *iterable* is a Python object `L` (a list, a set, ...) over whose elements it is possible to iterate. Technically, the iterator is constructed by `iter(L)`. In practice, the commands `iter` and `next` are used very rarely, since `for` loops and list comprehensions provide a much more pleasant syntax:

```
sage: for s in Subsets(3): s
{}
{1}
{2}
{3}
{1, 2}
{1, 3}
{2, 3}
{1, 2, 3}
```

```
sage: [ s.cardinality() for s in Subsets(3) ]
[0, 1, 1, 1, 2, 2, 2, 3]
```

What is the point of an iterator? Consider the following example:

```
sage: sum( [ binomial(8, i) for i in range(9) ] )
256
```

When it is executed, a list of 9 elements is constructed, and then it is passed as an argument to `sum` to add them up. If, on the other hand, the iterator is passed directly to `sum` (note the absence of square brackets):

```
sage: sum( binomial(8, i) for i in xrange(9) )
256
```

the function `sum` receives the iterator directly, and can short-circuit the construction of the intermediate list. If there is a large number of elements, this avoids allocating a large quantity of memory to fill a list that will be immediately destroyed[2].

Most functions that take a list of elements as input will also accept an iterator (or an iterable) instead. To begin with, one can obtain the list (or the tuple) of elements of an iterator as follows:

```
sage: list(binomial(8, i) for i in xrange(9))
[1, 8, 28, 56, 70, 56, 28, 8, 1]
sage: tuple(binomial(8, i) for i in xrange(9))
(1, 8, 28, 56, 70, 56, 28, 8, 1)
```

We now consider the functions `all` and `any`, which denote respectively the n-ary *and* and *or*:

```
sage: all([True, True, True, True])
True
sage: all([True, False, True, True])
False
sage: any([False, False, False, False])
False
sage: any([False, False, True, False])
True
```

The following example verifies that all primes from 3 to 99 are odd:

```
sage: all( is_odd(p) for p in xrange(3,100) if is_prime(p) )
True
```

A *Mersenne prime* is a prime of the form $2^p - 1$. We verify that, for $p < 1000$, if $2^p - 1$ is prime, then p is also prime:

```
sage: def mersenne(p): return 2^p - 1
sage: [ is_prime(p) for p in range(1000) if is_prime(mersenne(p)) ]
```

[2]Technical detail: `xrange` returns an iterator on $\{0, \ldots, 8\}$ while `range` returns the corresponding list. Starting in Python 3.0, `range` will behave like `xrange`, and `xrange` will no longer be needed.

```
[True, True, True, True, True, True, True, True, True, True,
 True, True, True, True]
```

Is the converse true?

Exercise 58. Try the two following commands and explain the considerable difference in the length of the calculations:

```
sage: all( [ is_prime(mersenne(p)) for p in range(1000) if is_prime(p)] )
False
sage: all(   is_prime(mersenne(p)) for p in range(1000) if is_prime(p)  )
False
```

We now try to find the smallest counter-example. In order to do this, we use the Sage function `exists`:

```
sage: exists( (p for p in range(1000) if is_prime(p)),
....:         lambda p: not is_prime(mersenne(p)) )
(True, 11)
```

Alternatively, we could construct an iterator on the counter-examples:

```
sage: counter_examples = \
....:   (p for p in range(1000)
....:      if is_prime(p) and not is_prime(mersenne(p)))
sage: counter_examples.next()
11
sage: counter_examples.next()
23
```

Exercise 59. What do the following commands do?

```
sage: cubes = [t**3 for t in range(-999,1000)]
sage: exists([(x,y) for x in cubes for y in cubes], lambda (x,y): x+y == 218)
sage: exists(((x,y) for x in cubes for y in cubes), lambda (x,y): x+y == 218)
```

Which of the last two is more economical in terms of time? In terms of memory? By how much?

Exercise 60. Try each of the following commands, and explain the results. Warning: it will be necessary to interrupt the execution of some of them.

```
sage: x = var('x'); sum( x^len(s) for s in Subsets(8) )
```

```
sage: sum( x^p.length() for p in Permutations(3) )
```

```
sage: P = Permutations(5)
sage: all( p in P for p in P )
```

```
sage: for p in GL(2, 2): print(p); print("-----")
```

```
sage: for p in Partitions(3): print(p)
```

```
sage: for p in Partitions(): print(p)
```

```
sage: for p in Primes(): print(p)
```

```
sage: exists( Primes(), lambda p: not is_prime(mersenne(p)) )
```

```
sage: counter_examples = (p for p in Primes()
....:                     if not is_prime(mersenne(p)))
sage: for p in counter_examples: print(p)
```

Operations on Iterators. Python provides numerous tools for manipulating iterators; most of them are in the `itertools` library, which can be imported by:

```
sage: import itertools
```

We will demonstrate some applications, taking as a starting point the permutations of 3:

```
sage: list(Permutations(3))
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

We can list the elements of a set by numbering them:

```
sage: list(enumerate(Permutations(3)))
[(0, [1, 2, 3]), (1, [1, 3, 2]), (2, [2, 1, 3]),
 (3, [2, 3, 1]), (4, [3, 1, 2]), (5, [3, 2, 1])]
```

select only the elements in positions 2, 3, and 4 (analogue of `l[1:4]`):

```
sage: list(itertools.islice(Permutations(3), 1, 4))
[[1, 3, 2], [2, 1, 3], [2, 3, 1]]
```

apply a function to all elements:

```
sage: list(itertools.imap(lambda z: z.cycle_type(), Permutations(3)))
[[1, 1, 1], [2, 1], [2, 1], [3], [3], [2, 1]]
```

or select the elements satisfying a certain condition:

```
sage: list(itertools.ifilter(lambda z: z.has_pattern([1,2]),
....:                        Permutations(3)))
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2]]
```

In all these situations, `attrcall` can be an advantageous alternative to creating an anonymous function:

```
sage: list(itertools.imap(attrcall("cycle_type"), Permutations(3)))
[[1, 1, 1], [2, 1], [2, 1], [3], [3], [2, 1]]
```

Implementation of New Iterators. It is easy to construct new iterators, using the keyword `yield` instead of `return` in a function:

```
sage: def f(n):
....:     for i in range(n):
....:         yield i
```

After the `yield`, execution is not halted, but only suspended, ready to be continued from the same point. The result of the function is therefore an iterator over the successive values returned by `yield`:

```
sage: g = f(4)
sage: g.next()
0
sage: g.next()
1
sage: g.next()
2
sage: g.next()
3
```

```
sage: g.next()
Traceback (most recent call last):
  ...
StopIteration
```

The function could be used as follows:

```
sage: [ x for x in f(5) ]
[0, 1, 2, 3, 4]
```

This model of computation, called *continuation*, is very useful in combinatorics, especially when combined with recursion. (See also §12.2.2 for other applications.) Here is how to generate all words of a given length on a given alphabet:

```
sage: def words(alphabet,l):
....:     if l == 0: yield []
....:     else:
....:         for word in words(alphabet, l-1):
....:             for l in alphabet: yield word + [l]
sage: [ w for w in words(['a','b'], 3) ]
[['a', 'a', 'a'], ['a', 'a', 'b'], ['a', 'b', 'a'], ['a', 'b', 'b'],
 ['b', 'a', 'a'], ['b', 'a', 'b'], ['b', 'b', 'a'], ['b', 'b', 'b']]
```

These words can then be counted by:

```
sage: sum(1 for w in words(['a','b','c','d'], 10))
1048576
```

Counting the words one by one is clearly not an efficient method in this case, since the formula n^ℓ is also available; note, though, that this is not the stupidest possible approach — it does, at least, avoid constructing the entire list in memory.

We now consider Dyck words, which are well-parenthesised words in the letters "(" and ")". The function below generates all Dyck words of a given length (where the length is the number of pairs of parentheses), using the recursive definition which says that a Dyck word is either empty or of the form $(w_1)w_2$ where w_1 and w_2 are Dyck words:

```
sage: def dyck_words(l):
....:     if l == 0: yield ''
....:     else:
....:         for k in range(l):
```

```
....:             for w1 in dyck_words(k):
....:                 for w2 in dyck_words(l-k-1):
....:                     yield '(' + w1 + ')' + w2
```

Here are all Dyck words of length 4:

```
sage: list(dyck_words(4))
['()()()()', '()()(())', '()(())()', '()(()())', '()((()))',
 '(())()()', '(())(())', '((())())', '(((())))', '((()()))',
 '((()()))', '(((()))', '(((())))']
```

Counting them, we recover a well-known sequence:

```
sage: [ sum(1 for w in dyck_words(l)) for l in range(10) ]
[1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]
```

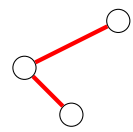
Exercise 61. Construct an iterator on the set C_n of complete binary trees with n leaves (see §15.1.2).

Indication: use `BinaryTree`; in the example below, we construct a leaf and the second tree of Figure 15.1.

```
sage: BT = BinaryTree
sage: BT()
.
sage: t = BT([BT([BT(), BT([BT(),BT()])]), BT()]); t
[[., [., .]], .]
```

Beware that, when drawing a complete binary tree, Sage uses the classical convention of displaying only its *skeleton*, that is the tree with its leaves pruned:

```
sage: view(t)
```



15.3 Constructions

We now see how to construct new sets starting from these building blocks. In fact, we have already begun to do this with the construction of $\mathcal{P}(\mathcal{P}(\mathcal{P}(\{1,2,3,4\})))$ in the previous section, and the example of sets of cards in §15.1.

Consider a large Cartesian product:

```
sage: C = cartesian_product([Compositions(8), Permutations(20)]); C
The Cartesian product of (Compositions of 8, Standard permutations of
    20)
sage: C.cardinality()
311411457046609920000
```

Clearly, it is impractical to construct the list of all elements of this Cartesian product. One can nevertheless manipulate it, for example to generate a random element:

```
sage: C.random_element()
([2, 3, 2, 1], [10, 6, 11, 13, 14, 3, 4, 19, 5, 12, 7, 18, 15, 8, 20, 1,
     17, 2, 9, 16])
```

The construction `cartesian_product` knows the algebraic properties of its arguments. Therefore, in the following example, H is equipped with the usual combinatorial operations and also its structure as a product group.

```
sage: G = DihedralGroup(4)
sage: H = cartesian_product([G,G])
sage: H.cardinality()
64
sage: H in Sets().Enumerated().Finite()
True
sage: H in Groups()
True
```

We now construct the disjoint union of two existing sets:

```
sage: C = DisjointUnionEnumeratedSets([Compositions(4),Permutations(3)])
sage: C
Disjoint union of Family (Compositions of 4, Standard permutations of 3)
sage: C.cardinality()
14
sage: C.list()
[[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [1, 3], [2, 1, 1], [2, 2], [3, 1],
[4], [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

It is also possible to take the union of more than two disjoint sets, or even an infinite number of them. We will now construct the set of all permutations, viewed as the union of the sets P_n of permutations of size n. We begin by constructing the infinite family $F = (P_n)_{n \in N}$:

```
sage: F = Family(NonNegativeIntegers(), Permutations); F
Lazy family (<class 'sage.combinat.permutation.Permutations'>(i))_{i in
    Non negative integers}
sage: F.keys()
Non negative integers
sage: F[1000]
Standard permutations of 1000
```

Now we can construct the disjoint union $\bigcup_{n \in \mathbb{N}} P_n$:

```
sage: U = DisjointUnionEnumeratedSets(F); U
Disjoint union of
Lazy family (<class 'sage.combinat.permutation.Permutations'>(i))_{i in
    Non negative integers}
```

It is an infinite set:

```
sage: U.cardinality()
+Infinity
```

which does not prohibit iteration through its elements, though it will be necessary to interrupt it at some point:

```
sage: for p in U: p
[]
[1]
[1, 2]
[2, 1]
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 1, 2]
...
```

Note: the above set could also have been constructed directly with:

```
sage: U = Permutations(); U
Standard permutations
```

Summary. To summarise, Sage provides a library of common enumerated sets, which can be combined via standard constructions, giving a toolbox that is flexible (but that could still be expanded). It is also possible to add new building blocks to Sage with a few lines (see the code in `Sets().Enumerated().Finite()`). This is made possible by the uniformity of the interfaces and the fact that Sage is based on an object-oriented language. Also, very large or even infinite sets can be manipulated thanks to lazy evaluation strategies (iterators, etc.).

There is no magic to any of this: under the hood, Sage applies the usual rules (for example, that the cardinality of $E \times E$ is $|E|^2$); the added value comes from the capacity to manipulate complicated constructions. The situation is comparable to Sage's implementation of differential calculus: Sage applies the usual rules for differentiation of functions and their compositions, where the added value comes from the possibility of manipulating complicated formulas. In this sense, Sage implements a *calculus* of finite enumerated sets.

15.4 Generic Algorithms

15.4.1 Lexicographic Generation of Lists of Integers

Among the classic enumerated sets, especially in algebraic combinatorics, a certain number are composed of lists of integers of fixed sum, such as partitions, compositions, or integer vectors. These examples can also have further constraints added to them. Here are some examples. We start with the integer vectors with sum 10 and length 3, with parts bounded below by 2, 4 and 2 respectively:

```
sage: IntegerVectors(10, 3, min_part = 2, max_part = 5,
....:                inner = [2, 4, 2]).list()
[[4, 4, 2], [3, 5, 2], [3, 4, 3], [2, 5, 3], [2, 4, 4]]
```

The compositions of 5 with each part at most 3, and with length 2 or 3:

```
sage: Compositions(5, max_part = 3,
....:              min_length = 2, max_length = 3).list()
[[3, 2], [3, 1, 1], [2, 3], [2, 2, 1], [2, 1, 2], [1, 3, 1],
 [1, 2, 2], [1, 1, 3]]
```

The strictly decreasing partitions of 5:

```
sage: Partitions(5, max_slope = -1).list()
[[5], [4, 1], [3, 2]]
```

These sets share the same underlying algorithmic structure, implemented in the more general — and slightly more cumbersome — class `IntegerListsLex`. This class models sets of vectors $(\ell_0, \ldots, \ell_k)$ of non-negative integers, with constraints on the sum and the length, and bounds on the parts and on the consecutive differences between the parts. Here are some more examples:

```
sage: IntegerListsLex(10, length=3, min_part = 2, max_part = 5,
....:                 floor = [2, 4, 2]).list()
[[4, 4, 2], [3, 5, 2], [3, 4, 3], [2, 5, 3], [2, 4, 4]]
```

```
sage: IntegerListsLex(5, min_part = 1, max_part = 3,
....:                 min_length = 2, max_length = 3).list()
[[3, 2], [3, 1, 1], [2, 3], [2, 2, 1], [2, 1, 2], [1, 3, 1],
 [1, 2, 2], [1, 1, 3]]
```

```
sage: IntegerListsLex(5, min_part = 1, max_slope = -1).list()
[[5], [4, 1], [3, 2]]
```

```
sage: list(Compositions(5, max_length=2))
[[5], [4, 1], [3, 2], [2, 3], [1, 4]]
```

```
sage: list(IntegerListsLex(5, max_length=2, min_part=1))
[[5], [4, 1], [3, 2], [2, 3], [1, 4]]
```

The point of the model of `IntegerListsLex` is in the good compromise between generality and efficiency in the iteration. The main algorithm permits iteration through the elements of such a set S in reverse lexicographic order, and in Constant Amortised Time (CAT), except in very degenerate cases; roughly speaking, the time needed to iterate through all elements is proportional to the number of elements, which is optimal. In addition, the memory usage is proportional to the largest element found, which is to say negligible in practice.

This algorithm is based on a very general principle for traversing a decision tree, called *branch and bound*: at the top level, we run through all possible choices for ℓ_0; for each of these choices, we run through all possible choices for ℓ_1, and so on. Mathematically speaking, we have put the structure of a prefix tree on the elements of S: a node of the tree at depth k corresponds to a prefix $\ell_0, \ldots, \ell_k$ of one (or more) elements of S (see Figure 15.4).

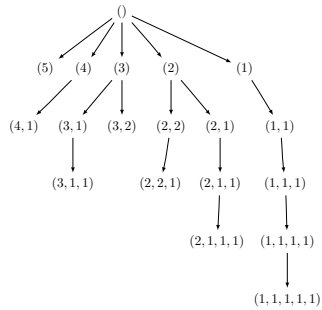


FIGURE 15.4 – The prefix tree of the partitions of 5.

The usual problem with this type of approach is to avoid bad decisions that lead to leaving the prefix tree and exploring dead branches, particularly problematic because the growth of the number of elements is exponential in the depth. It turns out that the constraints listed above are simple enough to guarantee the following property: given a prefix $\ell_0, \ldots, \ell_k$ of S, the set of ℓ_{k+1} such that $\ell_0, \ldots, \ell_{k+1}$ is a prefix of S is either empty or consists of an interval $[a, b]$, and the bounds a and b can be calculated in time linear in the length of the longest element of S having $\ell_0, \ldots, \ell_k$ as a prefix.

15.4.2 Integer Points in Polytopes

Although the algorithm for iteration in `IntegerListsLex` is efficient, its counting algorithm is naive: it just iterates over all the elements.

There is an alternative approach to this problem: modelling the desired lists of integers as the set of integer points of a polytope, that is to say, the set of solutions with integer coordinates of a system of linear inequalities. This is a very general context in which there exist advanced counting algorithms (e.g., Barvinok), which are implemented in libraries like LattE. Iteration does not pose a hard problem in principle. However, there are two limitations that justify the existence of `IntegerListsLex`. The first is theoretical: lattice points in a polytope only allow modelling of problems of a fixed dimension (length). The second is practical: at the moment only the library PALP has a Sage interface, and though it offers multiple capabilities for the study of polytopes, in the present application it only produces a list of lattice points, without providing either an iterator or non-naive counting:

```
sage: A = random_matrix(ZZ, 6, 3, x=7)
sage: L = LatticePolytope(A.rows())
sage: L.points()
M(1, 4, 3),
M(6, 4, 1),
...
M(3, 5, 5)
in 3-d lattice M
```

```
sage: L.points().cardinality()
23
```

Here is how to draw this polytope in 3D (see Figure 15.5):

```
sage: L.plot3d()
```

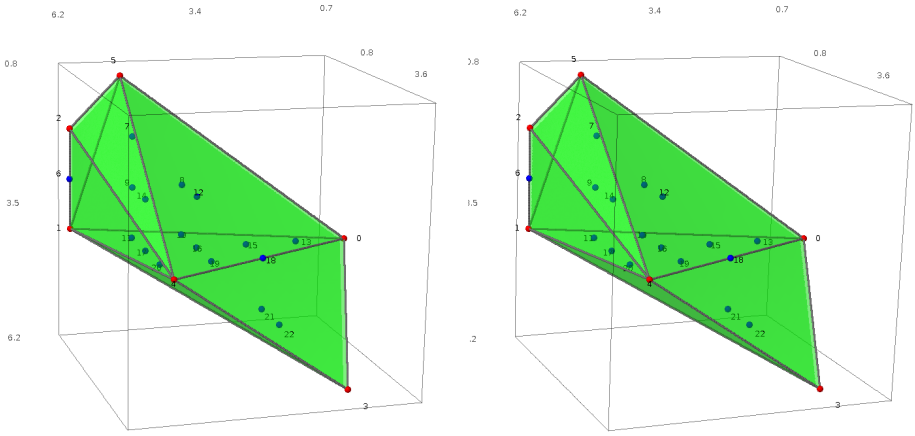


FIGURE 15.5 – The polytope L and its integer points, in cross-eyed stereographic perspective.

15.4.3 Species, Decomposable Combinatorial Classes

In §15.1.2, we showed how to use the recursive definition of binary trees to count them efficiently using generating functions. The techniques we used there are very general, and apply whenever the sets involved can be defined recursively (depending on who you ask, such a set is called a decomposable combinatorial class or, roughly speaking, a combinatorial species). This includes all types of trees, and also permutations, compositions, block diagrams, etc.

Here, we illustrate just a few examples using the Sage library on combinatorial species:

```
sage: from sage.combinat.species.library import *
sage: o = var('o')
```

We begin by redefining the complete binary trees; to do so, we stipulate the recurrence relation directly on the sets:

```
sage: BT = CombinatorialSpecies()
sage: Leaf =  SingletonSpecies()
sage: BT.define( Leaf + (BT*BT) )
```

Now we can construct the set of trees with five nodes, list them, count them...:

```
sage: BT5 = BT.isotypes([o]*5); BT5.cardinality()
```

```
14
sage: BT5.list()
[o*(o*(o*(o*o))), o*(o*((o*o)*o)), o*((o*o)*(o*o)), o*((o*(o*o))*o),
 o*(((o*o)*o)*o), (o*o)*(o*(o*o)), (o*o)*((o*o)*o), (o*(o*o))*(o*o),
 ((o*o)*o)*(o*o), (o*(o*(o*o)))*o, (o*((o*o)*o))*o, ((o*o)*(o*o))*o,
 ((o*(o*o))*o)*o, (((o*o)*o)*o)*o]
```

The trees are constructed using a generic recursive structure; the display is therefore not wonderful. To do better, it would be necessary to provide Sage with a more specialised data structure with the desired display capabilities.

We recover the generating function for the Catalan numbers:

```
sage: g = BT.isotype_generating_series(); g
x + x^2 + 2*x^3 + 5*x^4 + 14*x^5 + O(x^6)
```

which is returned in the form of a lazy power series:

```
sage: g[100]
227508830794229349661819540395688853956041682601541047340
```

We finish with the Fibonacci words, which are binary words without two consecutive "1"s. They admit a natural recursive definition:

```
sage: Eps = EmptySetSpecies(); Z0 = SingletonSpecies()
sage: Z1 = Eps*SingletonSpecies()
sage: FW  = CombinatorialSpecies()
sage: FW.define(Eps + Z0*FW  +  Z1*Eps + Z1*Z0*FW)
```

The Fibonacci sequence is easily recognised here, hence the name:

```
sage: L = FW.isotype_generating_series().coefficients(15); L
[1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
```

```
sage: oeis(L)
0: A000045: Fibonacci numbers: F(n) = F(n-1) + F(n-2) with F(0) = 0 and
    F(1) = 1.
1: A212804: Expansion of (1-x)/(1-x-x^2).
2: A132636: a(n) = Fibonacci(n) mod n^3.
```

This is an immediate consequence of the recurrence relation. One can also generate immediately all Fibonacci words of a given length, with the same limitations resulting from the generic display.

```
sage: FW3 = FW.isotypes([o]*3)
sage: FW3.list()
[o*(o*(o*{})), o*(o*(({}*o)*{})), o*((({}*o)*o)*{}),
 (({}*o)*o)*(o*{}), (({}*o)*o)*(({}*o)*{})]
```

By replacing `o` by 0, `{}*o` by 1 and dropping parentheses as well as the last `{}`, one reads respectively 000, 001, 010, 100 and 101.

15.4.4 Objects up to Isomorphism

We saw in §15.2.3 that Sage could generate graphs and partial orders up to isomorphism. In this section, we describe two typical algorithms to, respectively, generate and count objects up to isomorphism: orderly generation and Pólya enumeration. Our running example will be unlabelled simple graphs.

Graphs up to Isomorphism. We begin by recalling some notions. A graph $G = (V, E)$ is a set V of vertices and a set E of edges connecting these vertices; an edge is described by a pair $\{u, v\}$ of distinct vertices of V. Such a graph is called labelled; its vertices are typically numbered by considering $V = \{1, 2, \ldots, n\}$.

In many problems, the labels on the vertices play no role. Typically a chemist wants to study all possible molecules with a given composition, for example the alkanes with $N = 8$ carbon atoms and $2N + 2 = 18$ hydrogen atoms. We therefore want to find all (connected) graphs consisting of 8 vertices with 4 neighbours, and 18 vertices with a single neighbour. The different carbon atoms, however, are all considered to be identical, and the same for the hydrogen atoms. Our chemist's problem is not artificial; this type of application is actually at the origin of an important part of the research in graph theory on isomorphism problems.

When working by hand on a small graph it is possible, as in the example of §15.2.3, to make a drawing, erase the labels, and "forget" the geometrical information about the location of the vertices in the plane. However, to represent a graph in a computer program, it is necessary to introduce labels on the vertices so as to be able to describe how the edges connect them together. To compensate for the extra information which we have introduced, we then say that two labelled graphs g_1 and g_2 are *isomorphic* if there is a bijection from the vertices of g_1 to those of g_2, which maps the edges of g_1 bijectively to those of g_2; an *unlabelled graph* is then an equivalence class of labelled graphs.

Orderly Generation. We start with the algorithms behind the generation of graphs up to isomorphism. In general, (graph) isomorphism problems are hard; for example, testing if two labelled graphs are isomorphic is computationally expensive. However, the number of graphs, even unlabelled, grows very rapidly, and it turns out to be possible to list unlabelled graphs very efficiently considering their number. For example, the program Nauty can list the 12005168 simple graphs with 10 vertices in seconds.

As in §15.4.1, the general principle of the algorithm is to organise the objects to be enumerated into a tree that one traverses.

For this, in each equivalence class of labelled graphs (that is to say, for each unlabelled graph) one fixes a convenient canonical representative. The following are the fundamental operations:

1. testing whether a labelled graph is canonical;

2. calculating the canonical representative of a labelled graph.

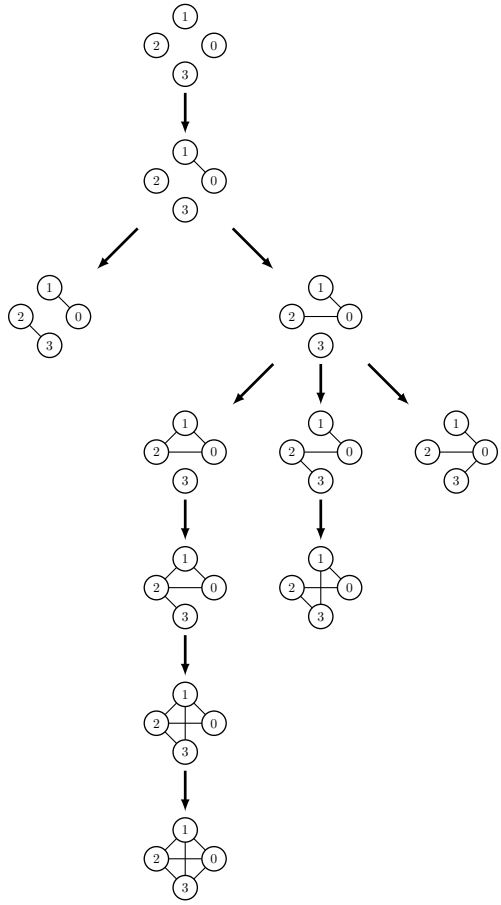


FIGURE 15.6 – The generation tree of simple graphs with 4 vertices.

These unavoidable operations remain expensive; one therefore tries to minimise the number of calls to them.

The canonical representatives are chosen in such a way that, for each canonical labelled graph G, there is a canonical choice of an edge whose removal produces another canonical labelled graph, which is called the father of G. This property implies that it is possible to organise the canonical labelled graphs on a set V of vertices as the nodes of a tree: at the root, the graph with no edges; below it, its unique child, the graph with one edge; then the graphs with two edges, and so on. The set of children of a graph G can be constructed by *augmentation*, adding an edge in all possible ways to G, and then selecting, among those graphs, the ones that are still canonical[3]. Recursively, one obtains all canonical graphs.

In what sense is this algorithm generic? Consider for example planar graphs

[3]In practice, an efficient implementation would exploit the symmetries of G, i.e., its automorphism group, to reduce the number of children to explore, and to reduce the cost of each test of canonicity.

(graphs which can be drawn in the plane without edges crossing): by removing an edge from a planar graph, one obtains another planar graph; so planar graphs form a subtree of the previous tree. To generate them, exactly the same algorithm can be used, selecting only the children that are planar:

```
sage: [len(list(graphs(n, property = lambda G: G.is_planar())))
....:  for n in range(7)]
[1, 1, 2, 4, 11, 33, 142]
```

In a similar fashion, one can generate any family of graphs closed under edge-deletion, and in particular any family characterised by forbidden subgraphs. This includes for example forests (graphs without cycles), bipartite graphs (graphs without odd cycles), etc. This approach can also be applied to generate:

- partial orders, via the bijection with Hasse diagrams (directed graphs without cycles and without edges implied by the transitivity of the order relation);
- lattices, via the bijection with the meet semi-lattice obtained by deleting the maximal vertex; in this case an augmentation by vertices rather than by edges is used.

Pólya Enumeration. We now count graphs on n vertices up to isomorphism, using Pólya enumeration. We remain voluntarily brief and informal; the reader is encouraged to focus on the examples to get an intuition of what is going on. Our purpose is indeed merely to point to this gem of algebraic combinatorics that connects enumerative combinatorics, group theory, symmetric functions, and in fact even some representation theory. For details on the theory behind it, we refer to https://en.wikipedia.org/wiki/Cycle_index.

As above, we fix $V = \{1, 2, \ldots, n\}$ as the set of vertices. In the examples, we take $n = 4$. A labelled graph is now just a subset E of the collection F of the $\binom{n}{2}$ pairs $\{i, j\}$ of vertices:

```
sage: V = [1,2,3,4]
sage: F = Subsets(V, 2); F.list()
[{1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4}]
```

We take the group S_n of all permutations of V:

```
sage: S = SymmetricGroup(V)
```

We want to let S_n act on pairs in F. We start with a function that lets a single permutation act on a pair:

```
sage: def on_pair(sigma, pair):
....:     return Set(sigma(i) for i in pair)
sage: def on_pairs(sigma):
....:     return [on_pair(sigma, e) for e in F]
```

Here it is in action:

```
sage: sigma = S.an_element(); sigma
```

```
(1,2,3,4)
sage: for e in F: print((e, on_pair(sigma, e)))
({1, 2}, {2, 3})
({1, 3}, {2, 4})
({1, 4}, {1, 2})
({2, 3}, {3, 4})
({2, 4}, {1, 3})
({3, 4}, {1, 4})
sage: on_pairs(sigma)
[{2, 3}, {2, 4}, {1, 2}, {3, 4}, {1, 3}, {1, 4}]
```

We can now construct the permutation group G_n acting on pairs, induced by the action of S_n:

```
sage: G = PermutationGroup([ on_pairs(sigma) for sigma in S.gens() ],
....:                      domain=F)
```

We compute the cycle indicator of G_n next. This is a symmetric function that encodes the statistic of the cycle types of the permutations in G_n:

```
sage: Z = G.cycle_index(); Z
1/24*p[1, 1, 1, 1, 1, 1] + 3/8*p[2, 2, 1, 1] + 1/3*p[3, 3] + 1/4*p[4, 2]
```

This gadget is to be interpreted as follows: the term $\frac{1}{4}p_{(4,2)}$ indicates that, in G_4, there are $|G_4| \cdot \frac{1}{4} = 6$ permutations with one cycle of length 4 and one cycle of length 2; indeed:

```
sage: [ sigma for sigma in G if sigma.cycle_type() == [4,2] ]
[({1,2},{2,3},{3,4},{1,4})({1,3},{2,4}),
 ({1,2},{2,4},{3,4},{1,3})({1,4},{2,3}),
 ({1,2},{1,3},{3,4},{2,4})({1,4},{2,3}),
 ({1,2},{1,4},{3,4},{2,3})({1,3},{2,4}),
 ({1,2},{3,4})({1,3},{1,4},{2,4},{2,3}),
 ({1,2},{3,4})({1,3},{2,3},{2,4},{1,4})]
```

The interesting fact about this symmetric function — and this is the content of Pólya's enumeration formula — is that, *when evaluated on an alphabet* $A = (a_1, \ldots, a_n)$*, it returns the generating function by weight for the functions from* E *to a set of size* n *whose elements are weighted by* A. Skimming over the terminology, we illustrate this on our running example. We see a graph G as a function from F to a set with 2 elements, and weight edges with t and non edges with q. The generating function of unlabelled graphs on 4 nodes by number of edges is then:

```
sage: q,t = QQ['q,t'].gens()
sage: p = Z.expand(2, [q,t]); p
q^6 + q^5*t + 2*q^4*t^2 + 3*q^3*t^3 + 2*q^2*t^4 + q*t^5 + t^6
```

The term $2q^2t^4$ means that there are two graphs with four edges (and thus two non edges). The other coefficients can be checked with Figure 15.6. The total number of unlabelled graphs is obtained by further evaluating p at $q = t = 1$:

```
sage: p(q=1,t=1)
11
```

If we are interested in counting multigraphs (graphs with multiple edges) by number of edges instead, the cycle indicator polynomial can be evaluated on the infinite alphabet $A = (1, q, q^2, \dots)$. Infinite alphabets are not yet directly supported by Sage; however this can easily be done by hand since the evaluation of the symmetric powersum p_k on the alphabet A is obtained by encoding A as $1 + q + q^2 + \cdots = \frac{1}{1-q}$ and substituting q^k for q in this formula:

```
sage: q = var('q')
sage: H = sum( c * prod( 1/(1-q^k) for k in partition )
....:          for partition, c in Z )
sage: H
1/3/(q^3 - 1)^2 + 1/4/((q^4 - 1)*(q^2 - 1))
+ 3/8/((q^2 - 1)^2*(q - 1)^2) + 1/24/(q - 1)^6
```

Now, the number of multigraphs with 0 to 19 edges can be obtained by Taylor expansion:

```
sage: H.series(q)
1 + 1*q + 3*q^2 + 6*q^3 + 11*q^4 + 18*q^5 + 32*q^6 + 48*q^7
+ 75*q^8 + 111*q^9 + 160*q^10 + 224*q^11 + 313*q^12 + 420*q^13
+ 562*q^14 + 738*q^15 + 956*q^16 + 1221*q^17 + 1550*q^18 + 1936*q^19
+ Order(q^20)
```

The computation of the cycle index is carried out in a rather efficient way, using group theory algorithms provided by GAP to reduce the calculation of the cycle indicator to a summation over conjugacy classes of the group. The following example counts in a few seconds the number of graphs on $n = 10$ nodes; in that case, G_n contains 10! permutations acting on 45 edges:

```
sage: n = 10
sage: V = range(1,n+1)
sage: F = Subsets(V, 2)
sage: S = SymmetricGroup(V)
sage: G = PermutationGroup([ on_pairs(sigma) for sigma in S.gens() ],
....:                      domain=F)
sage: q,t = QQ['q,t'].gens()
sage: Z = G.cycle_index()
sage: Z.expand(2, [q,t])(q=1,t=1)
12005168
```

Most of the time is spent in computing the conjugacy classes of G_n. One can go much further by exploiting the specific structure of the group: indeed G_n is isomorphic to S_n, the conjugacy classes of which are indexed by integer partitions:

```
sage: n = 20
sage: V = range(1,n+1)
```

```
sage: F = Subsets(V, 2)
sage: S = SymmetricGroup(V)
sage: CC = S.conjugacy_classes(); CC
[...
 Conjugacy class of cycle type [19, 1] in Symmetric group of order 20!
     as a permutation group,
 Conjugacy class of cycle type [20] in Symmetric group of order 20! as a
      permutation group]
```

Now counting the number of graphs with 20 nodes takes just a few seconds:

```
sage: p = SymmetricFunctions(QQ).powersum()
sage: G = PermutationGroup([ on_pairs(sigma) for sigma in S.gens() ],
....:                      domain=F)
sage: Z = p.sum_of_terms([G(on_pairs(c.representative())).cycle_type(),
....:                     c.cardinality()]
....:                    for c in CC) / factorial(n)
sage: Z.expand(2, [q,t])(q=1,t=1)
64549012279579984185616463849074274944O
```

16 Graph Theory

This chapter presents the study of graph theory with Sage, starting with a description of the `Graph` class (§16.1) and its methods (§16.2), then how to use them to solve practical problems (§16.4) or verify theoretical results through experimentation (§16.3).

16.1 Constructing Graphs

16.1.1 Starting from Scratch

We define a graph as a pair (V, E), where V represents a set of vertices and E a set of edges, or unordered pairs of vertices. The graph shown in Figure 16.1 is defined by the set of vertices $\{0, 1, 2, 5, 9, \text{'Madrid'}, \text{'Edinburgh'}\}$ and has edges $(1, 2)$, $(1, 5)$, $(1, 9)$, $(2, 5)$, $(2, 9)$ as well as ('Madrid', 'Edinburgh').

It should come as no surprise that graphs in Sage are represented by the class `Graph`:

```
sage: g = Graph()
```

By default, `g` is an empty graph. The next example demonstrates how to add vertices and edges: whenever an edge is created, the corresponding vertices — if they are not already present in the graph — are silently added. We can observe this process with methods whose purpose is easy to guess:

```
sage: g.order(), g.size()
(0, 0)
sage: g.add_vertex(0)
sage: g.order(), g.size()
(1, 0)
```

```
sage: g.add_vertices([1, 2, 5, 9])
sage: g.order(), g.size()
(5, 0)
sage: g.add_edges([(1,5), (9,2), (2,5), (1,9)])
sage: g.order(), g.size()
(5, 4)
sage: g.add_edge("Madrid", "Edinburgh")
sage: g.order(), g.size()
(7, 5)
```

Adding the edge `(1,2)` is equivalent to adding the edge `(2,1)`. It should also be noted that the methods `add_vertex` and `add_edge` both have "plurals" (`add_vertices` and `add_edges`) that take a list as their argument, allowing a more compact composition (see for example §16.4.1, where we construct a graph having already generated a set of edges).

In general, Sage is not particular about what types of objects may be used as vertices of a graph. In fact it accepts any immutable Python objects, that is any object accepted as a dictionary keyword (cf. §3.3.9). It is, of course, possible to delete the added elements with the `delete_*` methods, and to enumerate the vertices and edges, which we will use often.

```
sage: g.delete_vertex(0)
sage: g.delete_edges([(1,5), (2,5)])
sage: g.order(), g.size()
(6, 3)
sage: g.vertices()
[1, 2, 5, 9, 'Edinburgh', 'Madrid']
sage: g.edges()
[(1, 9, None), (2, 9, None), ('Edinburgh', 'Madrid', None)]
```

The edges of a graph are actually represented in Sage as triples, of which the last entry is a label. Most of the time it is filled with a numeric value — interpreted for example as capacities in flow or connectivity algorithms, or as weights in matching problems — though it may contain any immutable object. By default, the label is `None`.

When we know the vertices and edges of a graph in advance, we can construct it in a compact manner with a dictionary associating each vertex with a list of its neighbours:

```
sage: g = Graph({
....:     0: [],
....:     1: [5, 9],
....:     2: [1, 5, 9],
....:     'Edinburgh': ['Madrid']})
```

As before, we can omit lines corresponding to vertices such as `5`, `9`, or `'Madrid'`, which are already listed as neighbours of other vertices. Likewise, we can specify that `1` is a neighbour of `2` even though `2` does not appear in the list of neighbours of `1`: the edge `(1,2)` is created either way.

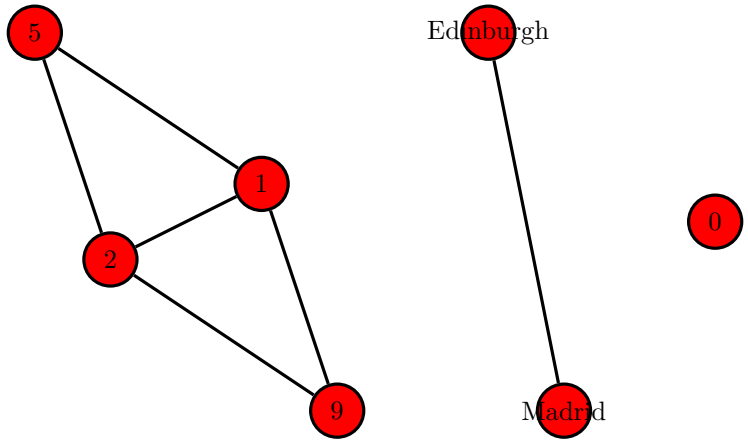


FIGURE 16.1 – A graph whose vertices are integers or character strings.

Exercise 62 (Circulant graphs)**.** A circulant graph parameterised by n, d is a graph of n vertices numbered from 0 to $n-1$ (which we can represent in shape of a circle), such that two vertices u and v are connected by an edge if $u \equiv v + c \bmod n$, with $-d \leq c \leq d$. Write a function which takes the parameters n and d, and returns the associated graph.

16.1.2 Available Constructors

Despite the previous examples, it is quite rare to enter an adjacency table in Sage, or likewise to manually enumerate the edges to create a graph. Most of the time it is more efficient to build them from pre-defined components: the methods in `graphs.*` allow the construction of more than seventy graphs or families of graphs, which we will now introduce. The Chvátal and Petersen graphs, for example, are obtained in Sage with the following lines:

```
sage: P = graphs.PetersenGraph()
sage: C = graphs.ChvatalGraph()
```

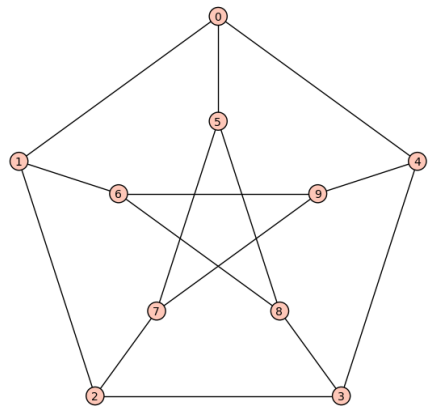
Let us start by describing small graphs — as opposed to graph families that we will encounter later.

Small Graphs. These graphs are most often named after their discoverers, or after an object they resemble (a house, a lollipop, a bull).

Small graphs		
`BullGraph`	`ChvatalGraph`	`ClawGraph`
`DesarguesGraph`	`DiamondGraph`	`DodecahedralGraph`
`FlowerSnark`	`FruchtGraph`	`HeawoodGraph`
`HexahedralGraph`	`HigmanSimsGraph`	`HoffmanSingletonGraph`
`HouseGraph`	`HouseXGraph`	`IcosahedralGraph`
`KrackhardtKiteGraph`	`LollipopGraph`	`MoebiusKantorGraph`
`OctahedralGraph`	`PappusGraph`	`PetersenGraph`
`TetrahedralGraph`	`ThomsenGraph`	

They often appear as counter-examples to certain conjectures, or as smaller graphs satisfying this or that property. The Petersen graph, for example, is non-planar: it contains — simultaneously — two minors forbidden by Kuratowski's theorem (K_5 and $K_{3,3}$). It is a triangle-free graph (its girth is 5), 3-regular, and of chromatic number 3 as well. It is also a vertex-transitive graph. Each of these properties can be determined by Sage with the help of the corresponding methods:

```
sage: P = graphs.PetersenGraph()
sage: P.is_planar()
False
sage: P.minor(graphs.CompleteBipartiteGraph(3,3))
{0: [1], 1: [8], 2: [4], 3: [6, 7, 9], 4: [2, 3], 5: [0, 5]}
sage: P.minor(graphs.CompleteGraph(5))
{0: [1, 6], 1: [0, 5], 2: [2, 7], 3: [4, 9], 4: [3, 8]}
sage: P.girth()
5
sage: P.is_regular(3)
True
sage: P.chromatic_number()
3
sage: P.is_vertex_transitive()
True
sage: P.show()
```



Families of Graphs. The constructors presented here describe families of graphs, each taking one or more arguments (with one exception, `nauty_geng`, which does not describe a specific family of graphs, but rather generates sets of *all* graphs isomorphic to each other; see Section 15.4.4).

In this list we find a generalisation (two, in fact) of the Petersen graph: the Kneser graph. This graph is constructed from two parameters, n and k, and its vertices are the $\binom{n}{k}$ size-k subsets of $\{1, \ldots, n\}$. Two of these sets are adjacent if and only if they are disjoint. The vertices of the Petersen graph correspond to subsets of size $k = 2$ of a set of size $n = 5$:

Families of graphs	
BarbellGraph	BubbleSortGraph
CircularLadderGraph	DegreeSequence
DegreeSequenceBipartite	DegreeSequenceConfigurationModel
DegreeSequenceTree	DorogovtsevGoltsevMendesGraph
FibonacciTree	FuzzyBallGraph
GeneralizedPetersenGraph	Grid2dGraph
GridGraph	HanoiTowerGraph
HyperStarGraph	KneserGraph
LCFGraph	LadderGraph
NKStarGraph	NStarGraph
OddGraph	ToroidalGrid2dGraph
nauty_geng	

```
sage: K = graphs.KneserGraph(5, 2); P = graphs.PetersenGraph()
sage: K.is_isomorphic(P)
True
```

By construction, Kneser graphs are also vertex-transitive. Their chromatic number is exactly $n - 2k + 2$, a surprising result of Lovász proven through the theorem of Borsuk-Ulam — and therefore by topological considerations [Mat03]. Let us check this immediately, with a few examples:

```
sage: all( graphs.KneserGraph(n,k).chromatic_number() == n - 2*k + 2
....:      for n in range(5,9) for k in range(2,floor(n/2)) )
True
```

Exercise 63 (Kneser Graphs)**.** Write a function of two parameters n, k returning the associated Kneser graph, if possible without using the "if" statement.

Basic Graphs. The following graphs are the most common "building blocks" in graph theory: complete graphs, complete bipartites, circulants, paths, cycles, stars, etc. Again, there is one notable exception: `trees`; this method can iterate over the set of all trees of n vertices.

Elementary graphs		
BalancedTree	CirculantGraph	CompleteBipartiteGraph
CompleteGraph	CubeGraph	CycleGraph
EmptyGraph	PathGraph	StarGraph
WheelGraph	trees	

Random Graphs. This last class of graphs is very rich in properties. Among others, we find $G_{n,p}$ and $G_{n,m}$, the two simplest models for random graphs.

Random graphs		
DegreeSequenceExpected	RandomBarabasiAlbert	RandomBipartite
RandomGNM	RandomGNP	RandomHolmeKim
RandomIntervalGraph	RandomLobster	RandomNewmanWattsStrogatz
RandomRegular	RandomShell	RandomTreePowerlaw

The graphs $G_{n,p}$ are defined by an integer n and a real $0 \leq p \leq 1$. We obtain a random graph $G_{n,p}$ on n vertices $\{0, \ldots, n-1\}$ by tossing a coin for each of

the $\binom{n}{2}$ vertex pairs i, j — such that the probability of landing on "heads" is p — and by adding the corresponding edge to the graph for each "heads" result.

We observe that for a fixed graph H, the probability that $G_{n,p}$ contains H as an induced subgraph[1] tends to 1 when $0 < p < 1$ is fixed and n tends to infinity (see §16.3.4):

```
sage: H = graphs.ClawGraph()
sage: def test():
....:     g = graphs.RandomGNP(20,2/5)
....:     return not g.subgraph_search(H, induced=True) is None
sage: sum( test() for i in range(100) ) >= 80
True
```

16.1.3 Disjoint Unions

In addition to these basic building blocks, Sage allows the creation of disjoint unions of graphs by means of two simple but effective operations. *The addition of two graphs* corresponds to their disjoint union:

```
sage: P = graphs.PetersenGraph()
sage: H = graphs.HoffmanSingletonGraph()
sage: U = P + H; U2 = P.disjoint_union(H)
sage: U.is_isomorphic(U2)
True
```

The *product* of a graph G by an integer k returns the disjoint union of k copies of G:

```
sage: C = graphs.ChvatalGraph()
sage: U = 3 * C; U2 = C.disjoint_union(C.disjoint_union(C))
sage: U2.is_isomorphic(U)
True
```

The following line creates a disjoint union of three copies of the Petersen graph and two copies of the Chvátal graph:

```
sage: U = 3*P + 2*C
```

There are many ways to verify this result, none of which should require more than a few lines of code. For example, by ensuring that each connected component is isomorphic to one of the two graphs:

```
sage: all( (CC.is_isomorphic(P) or CC.is_isomorphic(C))
....:       for CC in U.connected_components_subgraphs() )
True
```

or by counting the exact number of subgraphs:

[1] H is an induced subgraph of G if there exists a set $S \subseteq V(G)$ of vertices such that the restriction of G to S (that is, the graph whose vertices are S and whose edges are only those edges of G between vertices in S) is isomorphic to H. We denote such an induced subgraph $G[S]$.

```
sage: sum( CC.is_isomorphic(P)
....:      for CC in U.connected_components_subgraphs() )
3
sage: sum( CC.is_isomorphic(C)
....:      for CC in U.connected_components_subgraphs() )
2
```

Technical Details. It should be noted that the operations of addition and product create copies. This can be a source of overhead in terms of memory and time. As such, modifying `P` or `C` does not in turn cause `U` to be modified. Furthermore, these two operations result in two other losses of information:

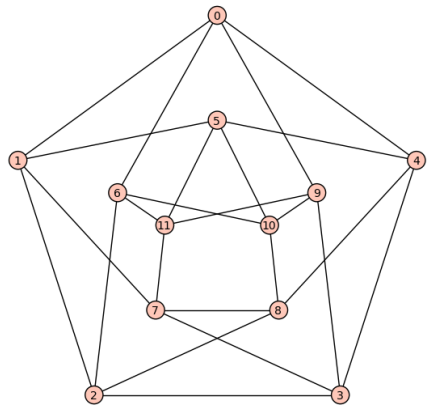
- the vertices of the final graph are re-labeled by integers $\{0, 1, 2, \ldots\}$;
- the positions (in the layout) of the vertices are not preserved in `U`.

The `disjoint_union` method behaves differently: if the graph `g` contains a vertex a, and the graph `h` a vertex b, the graph returned by `g.disjoint_union(h)` contains the vertices `(0, a)` and `(1, b)`. In the case that a or b are not integers, but some other type of object (strings, tuples, ...), use of this method greatly simplifies traversal of the graph resulting from this union.

16.1.4 Graph Visualisation

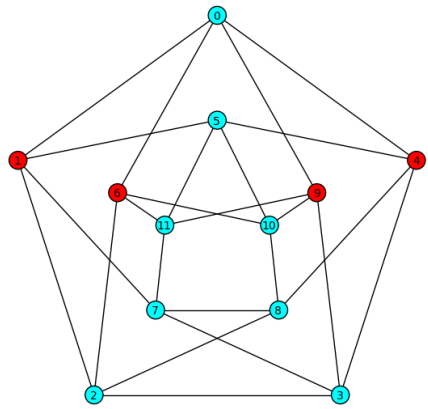
A very useful aspect of the study of graphs under Sage is the ability to visualise them. For a basic, no frills visualisation a single command suffices:

```
sage: C = graphs.ChvatalGraph(); C.show()
```



This is a valuable tool for visualising the results of certain functions. Here, we highlight an independent set:

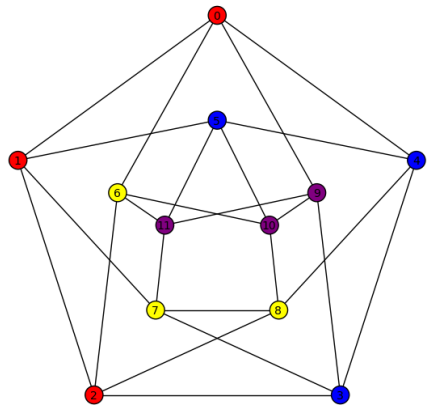
```
sage: C.show(partition = [C.independent_set()])
```



The `partition` argument of the `show` method accepts, as the name indicates, a partitioning of the set of vertices. A colour is assigned to each set in the partition in order to distinguish them visually. An additional colour is assigned to those vertices not belonging to the partition. In our example we thus have two colours in total.

It is of course possible to manually specify the colours we want to assign to the vertices, with the use of a dictionary in a straightforward syntax:

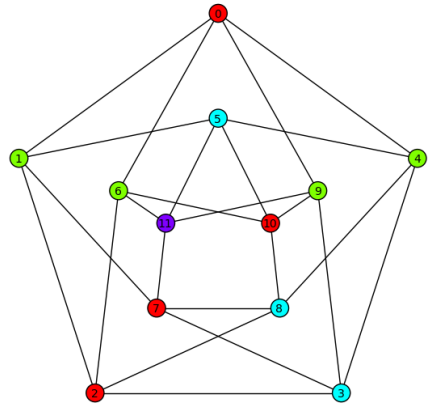
```
sage: C.show(vertex_colors = {
....:   "red" : [0, 1, 2],    "blue" : [3, 4, 5],
....:   "yellow" : [6, 7, 8], "purple" : [9, 10, 11]})
```



Since the colours we desire are not always primary or secondary colours, it is also possible to specify a hexadecimal code, as in HTML. Methods such as `coloring` are useful for situations like this:

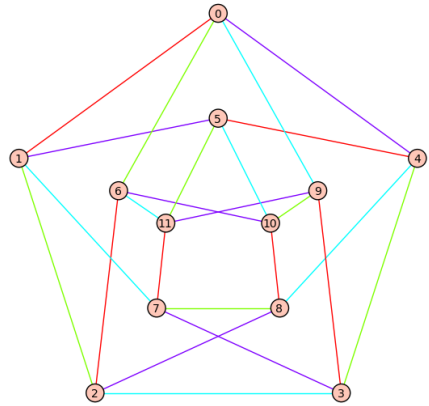
```
sage: C.coloring(hex_colors = True)
{'#00ffff': [3, 8, 5],
   '#7f00ff': [11],
   '#7fff00': [1, 4, 6, 9],
   '#ff0000': [0, 2, 7, 10]}
```

```
sage: C.show(vertex_colors = C.coloring(hex_colors = True))
```



Usage of the `edge_colors` argument is identical:

```
sage: from sage.graphs.graph_coloring import edge_coloring
sage: edge_coloring(C, hex_colors = True)
{'#00ffff': [(0, 6), (1, 5), (2, 8), (3, 4), (7, 11), (9, 10)],
   '#7f00ff': [(0, 4), (1, 7), (2, 6), (3, 9), (5, 11), (8, 10)],
   '#7fff00': [(0, 9), (1, 2), (3, 7), (4, 8), (5, 10), (6, 11)],
   '#ff0000': [(0, 1), (2, 3), (4, 5), (6, 10), (7, 8), (9, 11)]}
sage: C.show(edge_colors = edge_coloring(C, hex_colors = True))
```



Exporting Images. It is also possible to export individual images generated by Sage. The below example draws the complete graphs of $3, 4, ..., 12$ vertices, outputting them to the files `graph0.png`, ..., `graph9.png`.

```
sage: L = [graphs.CompleteGraph(i) for i in range(3,3+10)]
sage: for number, G in enumerate(L):
....:     G.plot().save('/tmp/' + 'graph' + str(number) + '.png')
```

The options of the `show` and `plot` commands are endless, and well-documented. At the very least we should mention the `figsize = 15` option, which specifies the resolution of the image and will prove useful for large graphs.

16.2 Methods of the `Graph` Class

The `Graph` class has more than 250 methods available, excluding those defined exclusively in the `DiGraph` class, or those appearing only in extension modules. This makes Sage an expressive and complete library for graph theory, allowing us to concentrate on the essentials — that is, we can spend less time programming basic functions, and more time on the problems in which we are actually interested.

As when learning any programming language (or library), it is useful to give its list of functions at least one look over in order to familiarise ourselves with its capabilities. This (non-exhaustive) section attempts to introduce the `Graph` methods succinctly. It is advisable for the reader to sacrifice a few minutes to look over the contents of each category of methods — *and even the complete list of methods*: the time will prove infinitely well-spent when faced with graph problems. It is also advisable to have a Sage session open to consult the online documentation for each of the methods presented (for example `g.degree_constrained_subgraph?`), as some accept many options, or have names that are not fully explicit.

16.2.1 Modification of Graph Structure

Of course, much of the `Graph` class is made up of natural methods for defining and modifying graphs — necessary but not particularly remarkable functionality.

Access and modification methods of the `Graph` class

`add_cycle`	`add_edge`	`add_edges`
`add_path`	`add_vertex`	`add_vertices`
`adjacency_matrix`	`allow_loops`	`allow_multiple_edges`
`allows_loops`	`allows_multiple_edges`	`clear`
`delete_edge`	`delete_edges`	`delete_multiedge`
`delete_vertex`	`delete_vertices`	`edge_iterator`
`edge_label`	`edge_labels`	`edges`
`edges_incident`	`get_vertex`	`get_vertices`
`has_edge`	`has_loops`	`has_multiple_edges`
`has_vertex`	`incidence_matrix`	`latex_options`
`loop_edges`	`loop_vertices`	`loops`
`merge_vertices`	`multiple_edges`	`name`
`neighbor_iterator`	`neighbors`	`networkx_graph`
`num_edges`	`num_verts`	`number_of_loops`
`order`	`relabel`	`remove_loops`
`remove_multiple_edges`	`rename`	`reset_name`
`save`	`set_edge_label`	`set_latex_options`
`set_vertex`	`set_vertices`	`size`
`subdivide_edge`	`subdivide_edges`	`vertex_iterator`
`vertices`	`weighted`	`weighted_adjacency_matrix`

16.2.2 Operators

Along the same lines, we find methods that act as *operators*, which return instances of the `Graph` class (or `DiGraph`). For example, the `complement` method applied

to a graph G returns a graph defined on the same set of vertices, such that the edge uv exists if and only if $uv \notin G$. The `subgraph` method obtains from a graph G the sub-graph *induced* by a given set of vertices (see definition page 368), an operation denoted $G[\{v_1, \ldots, v_k\}]$.

Let us check some elementary relationships. The complement of P_5 (denoted $\bar{P_5}$) is a house, and the graphs P_4 and C_5 are self-complementary:

```
sage: P5 = graphs.PathGraph(5); House = graphs.HouseGraph()
sage: P5.complement().is_isomorphic(House)
True
sage: P4 = graphs.PathGraph(4); P4.complement().is_isomorphic(P4)
True
sage: C5 = graphs.CycleGraph(5); C5.complement().is_isomorphic(C5)
True
```

Sage also defines (via the eponymous method) the *line graph* of G — often denoted $L(G)$ — of which the vertices correspond to the edges of G, and wherein two vertices are adjacent if their corresponding edges are incident in G. We also find the definitions of different products of graphs. In each of the following examples, we suppose that G is the product of G_1 and G_2, defined on the set of vertices $V(G_1) \times V(G_2)$. Two vertices $(u, v), (u', v') \in G$ are adjacent if and only if:

CARTESIAN PRODUCT
`cartesian_product`

$$\text{or} \begin{cases} u = u' \text{ and } vv' \in E(G_2) \\ uu' \in E(G_1) \text{ and } v = v' \end{cases}$$

LEXICOGRAPHIC PRODUCT
`lexicographic_product`

$$\text{or} \begin{cases} uu' \in E(G_1) \\ u = u' \text{ and } vv' \in E(G_2) \end{cases}$$

DISJUNCTIVE PRODUCT
`disjunctive_product`

$$\text{or} \begin{cases} uu' \in E(G_1) \\ vv' \in E(G_2) \end{cases}$$

TENSORIAL PRODUCT
`tensor_product`

$$\text{and} \begin{cases} uu' \in E(G_1) \\ vv' \in E(G_2) \end{cases}$$

STRONG PRODUCT
`strong_product`

$$\text{or} \begin{cases} u = u' \text{ and } vv' \in E(G_2) \\ uu' \in E(G_1) \text{ and } v = v' \\ uu' \in E(G_1) \text{ and } vv' \in E(G_2) \end{cases}$$

We can construct a square grid `GridGraph` as the cartesian product of two paths:

```
sage: n = 5; Path = graphs.PathGraph(n)
sage: Grid = Path.cartesian_product(Path)
sage: Grid.is_isomorphic(graphs.GridGraph([n,n]))
True
```

Products, operators, ...		
cartesian_product	categorical_product	complement
copy	disjoint_union	disjunctive_product
kirchhoff_matrix	laplacian_matrix	line_graph
lexicographic_product	strong_product	subgraph
to_directed	to_simple	to_undirected
tensor_product	transitive_closure	transitive_reduction
union		

16.2.3 Graph Traversal and Distances

Sage offers the usual graph traversal methods, such as depth-first and breadth-first search (`depth_first_search` and `breadth_first_search`) which are the basic routines for calculating distances, flow, and connectivity.

It also includes a less classical `lex_BFS` (*lexicographic breadth-first search*), used for example, in the detection of chordal graphs (cf. `is_chordal`). These methods return an ordering of vertices corresponding to the order of their discovery[2]:

```
sage: g = graphs.RandomGNP(10, .6)
sage: list(g.depth_first_search(0))
[0, 8, 5, 4, 9, 2, 3, 7, 6, 1]
sage: list(g.breadth_first_search(0))
[0, 8, 5, 4, 1, 3, 6, 7, 9, 2]
sage: g.lex_BFS(0)
[0, 8, 5, 4, 1, 6, 7, 3, 9, 2]
```

We define with the help of these traversal methods the `shortest_path` method, which is probably the most widely used of all[3]. Sage also allows the calculation of many invariants related to distances:

- `eccentricity`: associate with a vertex v the maximal distance between v and all other vertices of the graph;
- `center`: return a *central* vertex v of the graph — that is, a vertex of minimal eccentricity;
- `radius`: return the eccentricity of a *centre*;
- `diameter`: return the maximal distance between two vertices;
- `periphery`: return a list of vertices of eccentricity equal to the diameter.

[2] While `lex_BFS` returns a list of vertices, the `depth_first_search` and `breadth_first_search` methods are iterators over the vertices, hence the use of `list`.

[3] It should be noted that `shortest_path` does not necessarily call `breadth_first_search`: when the edges of a graph are given with associated distances, implementations of Dijkstra's algorithm (standard or bidirectional) take over.

Distances, traversal		
average_distance	breadth_first_search	center
depth_first_search	diameter	distance
distance_all_pairs	distance_graph	eccentricity
lex_BFS	periphery	radius
shortest_path	shortest_path_all_pairs	shortest_path_length
shortest_path_lengths	shortest_paths	

16.2.4 Flows, Connectivity, Matching

Sage can solve problems of maximum flow (cf. §17.4.3) with the help of the `flow` method[4]. Thanks to the aforementioned traversals, it also contains numerous methods related to connectivity (`is_connected`, `edge_connectivity`, `vertex_connectivity`, `connected_components`, ...) as well as results from Menger's theorem:

Given a graph G and two of its vertices u, v, the following statements are equivalent:

- *the value of the maximum flow between u and v is k (see* `flow`*);*
- *there exist k (but not $k+1$) edge-disjoint paths between u and v (see* `edge_disjoint_paths`*);*
- *there exists a set of k edges in G which, once removed from the graph, disconnect u from v (see* `edge_cut`*).*

The counterparts to these connectivity methods in terms of vertices are `flow` (with the `vertex_bound=True` option), `vertex_cut` and `vertex_disjoint_paths`.

Let us verify, for example, that with (very) high probability the connectivity of a random graph $G_{n,p}$ is equal to its minimum degree:

```
sage: n = 30; p = 0.3; trials = 50
sage: def equality(G):
....:     return G.edge_connectivity() == min(G.degree())
sage: sum(equality(graphs.RandomGNP(n,p)) for i in range(trials))/trials
1
```

We can also obtain the decomposition of a graph into 2-connected components, or its Gomory-Hu tree, respectively with `blocks_and_cut_vertices` and `gomory_hu_tree`.

Since it is one of the fundamental functions of graph theory, we will mention here the `matching` method, which constructs a maximal matching using Edmonds' algorithm. Matchings are also discussed in §16.4.2 and §17.4.2.

[4]Two implementations are available: the first follows the Ford-Fulkerson algorithm, while the second uses linear programming.

Flows, connectivity, ...	
`blocks_and_cut_vertices`	`connected_component_containing_vertex`
`connected_components`	`connected_components_number`
`connected_components_subgraphs`	`degree_constrained_subgraph`
`edge_boundary`	`edge_connectivity`
`edge_cut`	`edge_disjoint_paths`
`edge_disjoint_spanning_trees`	`flow`
`gomory_hu_tree`	`is_connected`
`matching`	`multicommodity_flow`
`vertex_boundary`	`vertex_connectivity`
`vertex_cut`	`vertex_disjoint_paths`

16.2.5 NP-Complete Problems

Sage contains algorithms for exact solutions of certain NP-complete problems. Of course, these problems may demand significant computational resources, but many real-world cases may be easier to solve than their theoretical counter-examples. For example, it is possible to solve the following optimisation problems.

Maximum Cliques and Independent Sets. A maximum clique of a graph is a set of pair-wise adjacent vertices of maximum cardinality (or of non-adjacent vertices in the case of the independent set). One application of this type of problem is presented in §16.4.1. This is the algorithm used by the program Cliquer [NO].

Methods: `clique_maximum`, `independent_set`

Vertex and Edge Colouring. A proper vertex colouring of a graph is an assignment of colours to vertices such that any two adjacent vertices have different colours. Sage has several functions to compute exact colourings, mostly using linear programming or the *Dancing Links* algorithm. The reader will find in §16.3.1 an explanation of a simple, but non-optimal graph colouring algorithm.

Methods: `chromatic_number`, `coloring`, `edge_coloring`[1], `grundy_coloring`[1].

Dominating Set. A set S of vertices of a graph is called dominating if all vertices v in G are neighbours with an element of S—we call a vertex not in the set S *dominated* by elements of the set. The set of all vertices being trivially dominating, the problem is to minimise the size of the set S. This problem is solved by Sage with the help of linear programming.

Method: `dominating_set`

Hamiltonian Cycle, Traveling Salesman. A graph G is called *Hamiltonian* if it contains a cycle that passes once and only once through each of its vertices. Unlike the problem of the *Eulerian* cycle — the cycle that uses each *edge* of G once and only once — this problem is NP-complete, and is solved by Sage using linear programming, in the specific case of the *traveling salesman* problem.

[1]These methods are not directly accessible through the `Graph` class. To access these and other colouring functions, see the `graph_coloring` module.

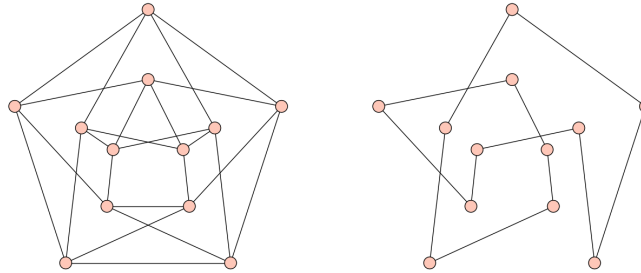


FIGURE 16.2 – The Chvátal graph and one of its Hamiltonian cycles.

We demonstrate the `hamiltonian_cycle` function, which returns a Hamiltonian cycle when such a cycle exists (Figure 16.2):

```
sage: g = graphs.ChvatalGraph(); cycle = g.hamiltonian_cycle()
sage: g.show(vertex_labels = False); cycle.show(vertex_labels = False)
```

Methods: `is_hamiltonian`, `hamiltonian_cycle`, `traveling_salesman_problem`

Miscellaneous Problems. Sage also knows how to calculate the *genus* of a graph (`genus`), *maximum cuts* (`max_cut`), *Steiner trees* (`steiner_tree`), etc. It can also solve existence problems, such as of multi-commodity flows (`multicommodity_flow`), test for the existence of minors (`minor`—find a minor isomorphic to a given graph), or search for subgraphs (`subgraph_search`).

Although the theoretical complexity of these problems is not yet known, algorithms are available to solve the problem of graph isomorphism (`is_isomorphic`) as well as to calculate the automorphism groups of a graph (`automorphism_group`).

NP-complete problems (or similar)	
automorphism_group	characteristic_polynomial
chromatic_number	chromatic_polynomial
coloring	disjoint_routed_paths
edge_coloring	dominating_set
genus	hamiltonian_cycle
independent_set_of_representatives	is_hamiltonian
is_isomorphic	max_cut
minor	multicommodity_flow
multiway_cut	subgraph_search
traveling_salesman_problem	steiner_tree
vertex_cover	

16.2.6 Recognition and Testing of Properties

A number of NP-complete problems have efficient solutions (linear, quadratic, ...) for graphs that belong to special classes. For example, it is trivial to solve the maximum clique problem on a chordal graph, and the complexity is polynomial

(albeit difficult) to compute an optimal vertex colouring on a perfect graph. Sage has algorithms for the recognition of certain elementary classes of graphs: forests (`is_forest`), trees (`is_tree`), bipartite graphs (`is_bipartite`), Eulerian graphs (`is_eulerian`), regular graphs (`is_regular`), etc. It is also possible to identify the following classes.

Chordal Graphs. A graph is said to be *chordal* if it does not have any cycle of size greater than four as an induced subgraph. Equivalently, any chordal graph can be decomposed by sequentially removing those vertices whose neighbourhood is a complete graph (this decomposition order is called a *perfect elimination ordering*). Such graphs can be recognised using a *breadth-first search* (`lex_BFS`).

Method: `is_chordal`

Interval Graphs. Let $\mathcal{I} = \{I_1, \ldots, I_n\}$ be a finite set of intervals of real numbers. From $\mathcal{I}$, we define a graph G of n vertices $\{1, \ldots, n\}$, where two vertices i and j are adjacent if and only if the corresponding intervals I_i and I_j have a non-empty intersection. Interval graphs are those that can be constructed in this fashion. They make up a sub-class of chordal graphs, recognisable in linear time thanks to the structure of PQ-trees.

Method: `is_interval`

Perfect Graphs. A graph G is said to be *perfect* if for all induced subgraphs $G' \subseteq G$ the chromatic number of G' is equal to the maximum size of a clique (i.e., the equality $\chi(G') = \omega(G')$ holds). Although the recognition of these graphs is a polynomial problem, the algorithms are complex and the implementation in Sage uses an exponential algorithm.

Method: `is_perfect`

Vertex-Transitive Graphs. A graph G is said to be *vertex-transitive* if there exists for all pairs of vertices u and v an isomorphism $h : V(G) \mapsto V(G)$ such that $h(u) = v$. Although the theoretical complexity of this problem is not yet established, the implementation available in Sage is quite efficient.

Method: `is_vertex_transitive`

Cartesian Product of Graphs. Only certain graphs can be expressed as the Cartesian product of a sequence $G_1, ..., G_k$ of graphs. It is possible, given a connected graph G, to find the unique such construction with the help of an elegant characterisation result, easily translated into a polynomial algorithm.

Method: `is_cartesian_transitive`

In addition to their characterisation by construction (such as chordal graphs) or by a particular property (such as perfect graphs), a number of classes of graphs are formulated in terms of excluded subgraphs. This is equivalent to saying that a graph G belongs to a class $\mathcal{C}$ if and only if it does not contain a subgraph in $\{G_1, \ldots, G_k\}$. In such cases we can write a recognition algorithm that simply

tests for the existence of each of these subgraphs, which can be accomplished with the `subgraph_search` function.

Recognition and tests of properties		
`is_bipartite`	`is_chordal`	`is_directed`
`is_equitable`	`is_eulerian`	`is_even_hole_free`
`is_forest`	`is_interval`	`is_odd_hole_free`
`is_overfull`	`is_regular`	`is_split`
`is_subgraph`	`is_transitively_reduced`	`is_tree`
`is_triangle_free`	`is_vertex_transitive`	

16.3 Graphs in Action

It is time now to put to use some of the features we have discovered. The following examples are motivated by a practical or theoretical pretext, and are not necessarily the best ways to use Sage to solve the problems they illustrate. They are often brute-force and enumerative, and that is what gives them their charm: their aim is obviously to give clear and approachable examples of using Sage's graph library — form over substance.

16.3.1 Greedy Vertex Colouring of a Graph

To colour the vertices of a graph means to assign to each vertex a colour (here we can agree that an integer will serve as a colour), so that each vertex has a different colour from that of its neighbours. This is of course always possible: it suffices to use as many colours as there are vertices; it is for this reason that the problem of colouring is a minimisation problem: to find, given a graph G, the smallest number $\chi(G)$ of colours that satisfies the aforementioned constraint.

As the calculation of $\chi(G)$ is a problem to which much impressive literature has been devoted, the reader with a more practical mindset will be pleased to find that there exist expeditive ways of getting closer to the optimal colouring than the trivial $|V|$ colours. To them we propose the following algorithm: "Greedy colouring of graph vertices".

We take an arbitrary vertex and assign it a colour, the integer 0. Iteratively, we take a non-coloured vertex and assign it the lowest unused integer not used by its neighbours.

This algorithm only requires a few lines of explanation, and it is the same when implemented in Sage. We apply the algorithm here to a random graph:

```
sage: n = 100; p = 5/n; g = graphs.RandomGNP(n, p)
```

```
sage: # Set of available colours.
sage: # In the worst-case scenario up to n colours suffice
sage: available_colours = Set(range(n))
```

```
sage: # This dictionary contains the colour associated
```

```
sage: # with each vertex of the graph
sage: colour = {}
sage: for u in g:
....:     forbidden = Set([colour[v] for v in g.neighbors(u)
....:                                if v in colour])
....:     colour[u] = min(available_colours - forbidden)
```

```
sage: # Number of colours used
sage: max(colour.values()) + 1
6
```

This is significantly more efficient than using 100 colours. It is easy, however, to improve this algorithm when we notice that it depends on an unknown: the order in which the vertices are selected. In fact, we do not have any control over the ordering, as we just use "`for u in g`" which allowed for rapid development of the program. In Chapter 15, we learned of the rich collection of set types available in Sage, of which `Permutations` was one. Better still, this class has a `random_element` method that we can use:

```
sage: P = Permutations([0,1,2,3]); P.random_element()
[2, 0, 1, 3]
```

We will try to obtain better results by colouring the vertices of our graph 30 times in orders given by random permutations. The result is the following code that we apply to the graph `g` defined previously:

```
sage: available_colours = Set(range(n))
```

```
sage: n_tests = 30
sage: vertices = g.vertices()
sage: P = Permutations(range(n))
sage: best_coloring = {}
sage: best_chromatic_number = +oo
```

```
sage: for t in range(n_tests):
....:     # Random ordering of vertices
....:     p = P.random_element()
....:     colour = {}
....:     for i in range(g.order()):
....:         u = vertices[p[i]]
....:         forbidden = Set([colour[v] for v in g.neighbors(u)
....:                          if v in colour])
....:         colour[u] = min(available_colours - forbidden)
....:     # Update the best colouring
....:     if max(colour.values()) + 1 < best_chromatic_number:
....:         best_coloring = colour
....:         best_chromatic_number = 1 + max(colour.values())
```

```
sage: best_chromatic_number # Number of colours used
```

```
4
```

An improvement, in any case! However, all the machinery for updating the minimum is not necessary. There is no reason to program what is already there. In Python, a large majority of objects — in this case `(int, dict)` pairs — are comparable to each other, in lexicographic order (we compare the first terms — integers —, then the second terms — dictionaries —, which has less significance in this case). By rewriting the first part of our code as a function we obtain the following result:

```
sage: def greedy_coloring(g, permutation):
....:     n = g.order()
....:     available_colours = Set(range(n))
....:     vertices = g.vertices()
....:     colour = {}
....:     for i in range(n):
....:         u = vertices[permutation[i]]
....:         forbidden = Set([colour[v] for v in g.neighbors(u)
....:                          if v in colour])
....:         colour[u] = min(available_colours - forbidden)
....:     return max(colour.values()) + 1, colour
```

With this function defined, performing 50 attempts and returning the minimum is trivial:

```
sage: P = Permutations(range(g.order()))
sage: n_colours, coloration = min(
....:    greedy_coloring(g, P.random_element()) for i in range(50))
sage: n_colours
4
```

To colour a graph using the minimal number of colours, it is preferable to use the `coloring` method. Being that this problem is NP-complete, one should expect longer computation times than with greedy colouring.

Exercise 64 (Optimal order for greedy colouring)**.** The greedy colouring algorithm is capable of colouring a graph with the minimum number of colours possible (i.e., $\chi(G)$) if it iterates through the vertices in the right order. With the help of the `coloring` method, which calculates an optimal colouring, write a function that returns an order of vertices with which the greedy colouring algorithm produces the optimal results.

16.3.2 Generating Graphs Under Constraints

Random graphs $G_{n,p}$ have very interesting connectivity properties. In particular, their minimal cuts are almost certainly the neighbourhood of a vertex: there are therefore cuts such that one of the two connected partitions consists of a single vertex. This may seem unsettling: every set of vertices then defines a cut of greater cardinality than the size of the minimum cut. However, it is possible, with great patience (for very large graphs) and a few lines of Sage to produce somewhat different random graphs. Here is the method we will implement.

Let n and k be two integers, the first representing the number of vertices and the second a connectivity. The algorithm begins with a tree of n vertices; calculate a minimum cut and its two corresponding sets. As long as the minimum cut is of size $k' < k$, we randomly add $k - k'$ edges between the two sets.

As described above, given a pair of sets S and $\bar{S}$, we will need to generate a pair of elements $(s, \bar{s}) \in S \times \bar{S}$. For this we use the constructor `cartesian_product` and the resulting object's `random_element` method.

```
sage: n = 20; k = 4; g = graphs.RandomGNP(n, 0.5)
sage: g = g.subgraph(edges = g.min_spanning_tree())
```

```
sage: while True:
....:     _, edges, [S,Sb] = g.edge_connectivity(vertices = True)
....:     cardinality = len(edges)
....:     if cardinality < k:
....:         CP = cartesian_product([S, Sb])
....:         g.add_edges([CP.random_element()
....:                      for i in range(k - len(edges))])
....:     else:
....:         break
```

And that's it.

16.3.3 Find a Large Independent Set

Although Sage provides a method `Graph.independent_set` that finds a maximal independent set in a graph (set of non-adjacent vertices), nothing prevents us from using funny graph theory results to discover ourselves an independent set. We can read for example in the book *The Probabilistic Method* [AS00] that any graph G has an independent set S such that

$$|S| \geq \sum_{v \in G} \frac{1}{d(v) + 1}$$

where $d(v)$ stands for the degree of v. The proof of this result lies in the following algorithm.

*Let us take a random bijection $n : V \mapsto \{1, \ldots, |V|\}$, associating to each vertex of G a unique integer. Let us now associate to this function an independent set S_n, defined as the set of vertices of G having image less than all their neighbours (*minimal *vertices). Formally, this is written:*

$$S_n = \{v \in G : \forall u \text{ such that } uv \in E(G), n(v) < n(u)\}.$$

This set is by definition an independent set, but how to control its size? It suffices to ask, for each vertex, the frequency with which it appears in the set S_n.

If we consider the set P of bijections from V to $\{1, \dots, |V|\}$, we notice that

$$\begin{aligned}\sum_{n\in P} |S_n| &= \sum_{n\in P}\sum_{v\in G} \text{“1 if } v \text{ is minimal for } n\text{, 0 otherwise”} \\ &= \sum_{v\in G}\Big(\sum_{n\in P} \text{“1 if } v \text{ is minimal for } n\text{, 0 otherwise”}\Big) \\ &= \sum_{v\in G}\frac{|P|}{d(v)+1} = |P|\sum_{v\in G}\frac{1}{d(v)+1}.\end{aligned}$$

As a consequence, such a function corresponds on average to an independent set of size $\sum_{v\in G}\frac{1}{d(v)+1}$. To obtain a set of this size with Sage, we will use random bijections using the `Permutations` class, until we obtain the size promised by the theorem:

```
sage: g = graphs.RandomGNP(40, 0.4)
sage: P = Permutations(range(g.order()))
sage: mean = sum( 1/(g.degree(v)+1) for v in g )
```

```
sage: while True:
....:     n = P.random_element()
....:     S = [v for v in g if all( n[v] < n[u] for u in g.neighbors(v))]
....:     if len(S) >= mean:
....:         break
```

16.3.4 Find an Induced Subgraph in a Random Graph

We will play here with the random graphs $G_{n,p}$, quickly discussed in Section 16.1.2 on graph constructors. As mentioned there, these graphs have the following property.

> *Let H be a graph, and $0 < p < 1$. Then:*
>
> $$\lim_{n\to+\infty} P\left[H \text{ is an induced subgraph of } G_{n,p}\right] = 1$$
>
> *which means that, H and p being fixed, a large random graph $G_{n,p}$ will almost surely contain H as induced subgraph (see Definition page 368).*

Let us reformulate: given a graph H and a large random graph G, it is possible to find a copy of H in G, by iteratively assigning to each vertex v_i of $V(H) = \{v_1, \dots, v_k\}$ a representative $h(v_i)$, where each v_i is a “correct extension” of the already selected vertices. We will thus follow the algorithm:

- associate to v_1 a random vertex $h(v_1) \in G$;
- associate to v_2 a random vertex $h(v_2) \in G$ such that $h(v_1)$ is adjacent to $h(v_2)$ in G if and only if v_1 is adjacent to v_2 in H;
- ...

- after $j < k$ steps, we have associated a representative $h(v_i) \in G$ to every v_i $(i \leq j)$, in such a way that for any $i, i' \leq j, h(v_i)h(v_{i'}) \in E(G)$ if and only if $v_i v_{i'} \in E(H)$. We now associate to v_{j+1} a random vertex $h(v_{j+1})$ such that for any $i \leq j$, $h(v_i)h(v_{j+1}) \in E(G)$ if and only if $v_i v_{j+1} \in E(H)$;
- ...
- after k steps, the subgraph of G induced by the representatives of vertices $v_1, \ldots, v_k$ is a copy of H.

PROPOSITION. When n is large, this strategy works with high probability.

Proof. Let us define $H_j = H[\{v_1, \ldots, v_j\}]$, and let us write $P[H \mapsto_{\text{ind}} G_{n,p}]$ the probability that H is an induced subgraph of $G_{n,p}$. We can roughly bound the probability that H_j (but not H_{j+1}) is an induced subgraph of $G_{n,p}$ in the following manner:

- Given a copy of H_j in some $G_{n,p}$, let us compute the probability that no other vertex can complete the current copy into a copy of H_{j+1}. The probability that a vertex works being

$$p^{d_{H_{j+1}}(v_{j+1})}(1-p)^{j-d_{H_{j+1}}(v_{j+1})} \geq \min(p, 1-p)^j,$$

  the probability that none of the $n - j$ remaining vertices works is at most

$$\big(1 - \min(p, 1-p)^j\big)^{n-j}.$$

- There are in our graph at most $j!\binom{n}{j}$ different copies of H_j (in fact $\binom{n}{j}$ ways to choose a set of j vertices, and $j!$ bijections between these vertices and those of H_j).

Since $0 < p < 1$, we write $0 < \varepsilon = \min(p, 1-p)$; therefore, the probability that H_j (but not H_{j+1}) is an induced subgraph of $G_{n,p}$ is at most, for a fixed $j \leq k$,

$$j!\binom{n}{j}(1-\varepsilon^j)^{n-j} \leq j! n^j (1-\varepsilon^j)^{n-j} = o(1/n)$$

which is asymptotically zero when n grows. Eventually:

$$\begin{aligned} P[H \mapsto_{\text{ind}} G_{n,p}] &\geq 1 - P[H_2 \mapsto_{\text{ind}} G_{n,p}, H_3 \not\mapsto_{\text{ind}} G_{n,p}] \\ &\quad - P[H_3 \mapsto_{\text{ind}} G_{n,p}, H_4 \not\mapsto_{\text{ind}} G_{n,p}] \\ &\quad \cdots \\ &\quad - P[H_{k-1} \mapsto_{\text{ind}} G_{n,p}, H_k \not\mapsto_{\text{ind}} G_{n,p}] \\ P[H \mapsto_{\text{ind}} G_{n,p}] &\geq 1 - \sum_{j \leq k} j! n^j (1-\varepsilon^j)^{n-j} \\ &\geq 1 - k\, o\Big(\frac{1}{n}\Big). \end{aligned}$$

□

Moreover, this proof provides a probabilistic algorithm allowing to find a copy of a given graph H in a large random graph $G_{n,p}$. Although this algorithm does not always find a copy of H if such a copy exists, the probability of success tends to 1 when n goes to infinity.

```
sage: def find_induced(H, G):
....:  # the function from V(H) to V(G) we aim to define:
....:  f = {}
....:  # set of vertices of G not yet used by f:
....:  G_remain = G.vertices()
....:  # set of vertices having no representative yet:
....:  H_remain = H.vertices()
....:  # while the function is not complete:
....:  while H_remain:
....:      v = H_remain.pop(0) # look for the next vertex of H
....:      # and its potential images in G
....:      candidates = [u for u in G_remain if
....:             all([H.has_edge(h,v) == G.has_edge(f_h,u)
....:                    for h, f_h in f.iteritems()])]
....:      # if no candidate is found, we abort immediately
....:      if not candidates:
....:          raise ValueError("No copy of H has been found in G")
....:      # otherwise we select the first candidate
....:      f[v] = candidates[0]
....:      G_remain.remove(f[v])
....:  return f
```

```
sage: H = graphs.PetersenGraph()
sage: G = graphs.RandomGNP(500,0.5)
sage: find_induced(H,G)
{0: 0, 1: 4, 2: 3, 3: 7, 4: 35, 5: 10, 6: 67, 7: 108, 8: 240, 9: 39}
```

To find a copy of a given graph H in a graph G in one line, it is more efficient to call the `Graph.subgraph_search` method.

16.4 Some Problems Solved Using Graphs

16.4.1 A Quiz from the French Journal "Le Monde 2"

We can read in number 609 of "Le Monde 2" the following quiz.

> *What is the size of the largest set $S \subseteq [0, ..., 100]$ which does not contain two integers i, j such that $|i - j|$ is a square?*

The problem can be easily translated into a graph theory problem. The relation "$|i - j|$ *is a square*" being a binary symmetric relation, we start by creating the graph on the set of vertices $[0, \ldots, 100]$ in which two vertices are

adjacent (incompatible) if their difference is a square. We will use for that the `Subsets` class which allows us here to iterate over all size-2 subsets.

```
sage: n = 100; V = range(n+1)
sage: G = Graph()
sage: G.add_edges([
....:     (i,j) for i,j in Subsets(V,2) if is_square(abs(i-j)) ])
```

Since we are looking for a maximal number of "compatible" elements, we might call the `independent_set` method, which returns a maximal subset of non-adjacent elements.

```
sage: G.independent_set()
[4, 6, 9, 11, 16, 21, 23, 26, 28, 33, 38, 43, 50,
56, 61, 71, 76, 78, 83, 88, 93, 95, 98, 100]
```

The answer is thus 24, and not "42". As a consequence, the quiz from "Le Monde 2" was not "the ultimate question of life, the universe, and everything", whose answer should be searched elsewhere.

16.4.2 Task Assignment

We now face the following situation: for an important construction site, ten workers must complete a total of ten tasks. We can associate to each worker a list of tasks they are able to complete. How to distribute the tasks in a optimal way?

Here again, we start by translating the problem into a graph: it will be bipartite, defined on the union $\{w_0, \ldots, w_9\} \cup \{t_0, \ldots, t_9\}$ of workers and tasks, and we will define t_i as adjacent to w_j when w_j is able to complete task t_i.

```
sage: tasks = {0: [2, 5, 3, 7], 1: [0, 1, 4],
....:   2: [5, 0, 4],          3: [0, 1],
....:   4: [8],                5: [2],
....:   6: [8, 9, 7],          7: [5, 8, 7],
....:   8: [2, 5, 3, 6, 4],    9: [2, 5, 8, 6, 1]}
sage: G = Graph()
sage: for i in tasks:
....:   G.add_edges(("w" + str(i), "t" + str(j)) for j in tasks[i])
```

It now only remains to use the `matching` method, which will return a maximal set of tasks that can be performed simultaneously by different people:

```
sage: for task, worker,_ in sorted(G.matching()):
....:   print("{} can be performed by {}".format(task, worker))
t0 can be performed by w2
t1 can be performed by w3
t2 can be performed by w5
t3 can be performed by w8
t4 can be performed by w1
t5 can be performed by w7
t6 can be performed by w9
```

```
t7 can be performed by w0
t8 can be performed by w4
t9 can be performed by w6
```

16.4.3 Plan a Tournament

Given n teams competing in a tournament, in which each team must play against all other teams, how to plan all matches in the best way, knowing that several matches can happen simultaneously?

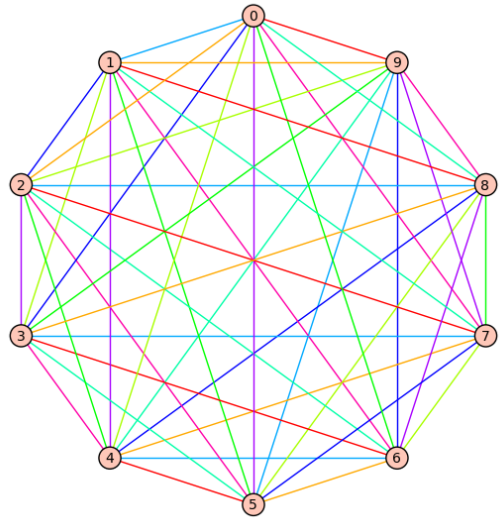
This is a typical case of the *proper vertex colouring* in graph theory. Given a graph G, this problem consists in assigning a colour to each edge so that no vertex touches two edges of same colour. Equivalently, this problem reduces to find an edge partition into pairings (union of disjoint vertices) of minimal cardinality. In the present case, we will try to colour the edges of the complete graph — each one representing the match between the two teams at its ends:

```
sage: n = 10
sage: G = graphs.CompleteGraph(n)
sage: from sage.graphs.graph_coloring import edge_coloring
sage: for day, matches in enumerate(edge_coloring(G)):
....:     print("Matches of day {}: {}".format(day, matches))
Matches of day 0: [(0, 9), (1, 8), (2, 7), (3, 6), (4, 5)]
Matches of day 1: [(0, 2), (1, 9), (3, 8), (4, 7), (5, 6)]
Matches of day 2: [(0, 4), (1, 3), (2, 9), (5, 8), (6, 7)]
Matches of day 3: [(0, 6), (1, 5), (2, 4), (3, 9), (7, 8)]
Matches of day 4: [(0, 8), (1, 7), (2, 6), (3, 5), (4, 9)]
Matches of day 5: [(0, 1), (2, 8), (3, 7), (4, 6), (5, 9)]
Matches of day 6: [(0, 3), (1, 2), (4, 8), (5, 7), (6, 9)]
Matches of day 7: [(0, 5), (1, 4), (2, 3), (6, 8), (7, 9)]
Matches of day 8: [(0, 7), (1, 6), (2, 5), (3, 4), (8, 9)]
```

It would be easy to adapt this solution to the case where the teams do not have to compete with all the others.

For the joy of it, the following image gives a proper edge colouring of the complete graph. A same colour indicates that the corresponding matches happen the same day.

```
sage: g = graphs.CompleteGraph(10)
sage: g.show(edge_colors=edge_coloring(g, hex_colors=True))
```


0
1
2
3
4
5
6
7
8
9

17

Linear Programming

This chapter is devoted to linear programming and mixed integer linear programming, presenting numerous problems that can be solved with these methods. The applications considered mainly come from graph theory, and the elementary ones should be easily understandable without any specific knowledge of this field. As a tool in combinatorics, applying linear programming amounts to understanding how to reformulate a problem of existence or optimisation in terms of linear constraints.

17.1 Definition

A *linear program* is a system of linear equations where the goal is to search for an optimal solution. Formally, it is defined by a matrix $A : \mathbb{R}^m \mapsto \mathbb{R}^n$ and two vectors $b \in \mathbb{R}^n$ and $c \in \mathbb{R}^m$. Solving a linear program then requires to find a vector $x \in \mathbb{R}^m$ which maximises an *objective* function, while satisfying a system of linear constraints, i.e.,

$$c^t x = \max_{y \text{ s.t. } Ay \leq b} c^t y$$

where the relationship $u \leq u'$ between two vectors indicates that the values of u are less than or equal to those of u', componentwise. We will also write:

$$\begin{aligned} &\text{maximise: } c^t x \\ &\text{such that: } Ax \leq b. \end{aligned}$$

A solution to the following linear program is given by $x = 4, y = 0, z = 1.6$:

$$\begin{aligned} \text{max: } & x + y + 3z \\ \text{such that: } & x + 2y \leq 4 \\ & 5z - y \leq 8 \\ & x, y, z \geq 0. \end{aligned}$$

In other words, solving a linear program consists in finding a point which maximises a linear function over a polytope (in this case the preimage $A^{-1}(\leq b)$). These definitions, however, do not yet explain the motivation for using linear programming in combinatorics, which is the main focus of this chapter. We will show how to apply this formalism in order to solve, among others, the knapsack problem (§17.4.1), the matching problem (§17.4.2), or the flow problem (§17.4.3). In §17.5, we will prove the existence of a Hamiltonian cycle by the method of generating constraints.

17.2 Integer Programming

There are bad news coming along with this definition of linear programming: a linear program (LP) can be solved in polynomial time. This is indeed bad news, because this would mean that unless we define LP's of exponential size, we cannot expect to solve NP-complete problems with this method, which would be a disappointment. On a brighter side, it becomes NP-complete to solve a linear program if we are allowed to specify constraints of a different kind: requiring that all or some components of x be integers instead of real values. Such a LP is actually called an integer linear program (ILP) or, if only certain components should be integers, a *Mixed Integer Linear Program* (MILP).

Solving ILP or MILP is known to be NP-complete. Hence, we can expect to find in the MILP framework a wide range of expressivity.

17.3 In Practice

17.3.1 The `MixedIntegerLinearProgram` Class

In Sage, `MixedIntegerLinearProgram` represents ... a MILP! It is also used to solve regular LP's. It has a very small number of methods, meant to define our set of constraints and variables, then to read the solution found by the solvers once computed. It is also possible to export a MILP defined with Sage to a `LP` or `MPS` file — standard formats, understood by most solvers.

For illustration, let us solve the linear program presented in §17.1. We first need to build an object of class `MixedIntegerLinearProgram`,

```
sage: p = MixedIntegerLinearProgram()
```

define the 3 optimisation variables,

```
sage: x, y, z = p['x'], p['y'], p['z']
```

```
sage: p.set_min(x, 0)
sage: p.set_min(y, 0)
sage: p.set_min(z, 0)
```

the objective function,

```
sage: p.set_objective( x + y + 3*z )
```

and finally the constraints:

```
sage: p.add_constraint( x + 2*y <= 4 )
sage: p.add_constraint( 5*z - y  <= 8 )
```

The method `solve` of `MixedIntegerLinearProgram` returns the optimal value of the objective function:

```
sage: p.solve()
8.8
```

We can read an optimal assignment of values for x, y, and z with the method `get_values`:

```
sage: p.get_values(x), p.get_values(y), p.get_values(z)
(4.0, 0.0, 1.6)
```

17.3.2 Variables

The variables associated to an instance of a `MixedIntegerLinearProgram` are objects of type `MIPVariable`, but we will not discuss that any further. In the previous example, the variables were obtained using `p['x']`, which is practical when their number is small. The linear programs that we define next often require associating to the variables a list of objects, such as integers, the vertices of a graph, or even other types of objects. It is then essential to be able to handle vectors of variables, or even dictionaries of variables.

For example, if in our linear program we need to define the variables $x_1, \ldots, x_{15}$, it is easier to make use of the method `new_variable`:

```
sage: x = p.new_variable()
```

It is now possible to define new constraints using our 15 variables:

```
sage: p.add_constraint( x[1] + x[12] - x[14] >= 8 )
```

We point out that it is not necessary to define the "size" of the vector `x`. In fact, `x` accepts without complaining any key of an immutable type (cf. §3.3.7), exactly like a dictionary. Hence, we can write:

```
sage: p.add_constraint( x["i_am_a_valid_index"] + x["a",pi] <= 3 )
```

Let us note that this formalism allows to use variables with multiple indices as well. To define the constraint $\sum_{0 \leq i,j < 4} x_{i,j} \leq 1$ we can write:

```
sage: p.add_constraint( p.sum(
....:       x[i,j] for i in range(4) for j in range(4)) <= 1 )
```

The notation `x[i,j]` is equivalent to the notation `x[(i,j)]`.

Types of Variables. By default, the variables returned by `new_variable` are real. It is possible to define them as binary variables, using the argument `binary = True`, or as integer variables, using `integer = True`. We can then set minimal or maximal bounds for the variables with the help of the methods `set_min` and `set_max`. Moreover, it is possible to change the type of a variable after it has been created, with the help of the methods `set_binary`, `set_integer` or `set_real`.

17.3.3 Infeasible or Unbounded Problems

Certain linear programs do not admit, formally, any solution. In fact, it is ambitious to attempt optimising a function — even a linear one — over an empty set, or conversely, over a domain without enough constraints, such that the objective function is unbounded. In these two cases, Sage will return an exception when the method `solve` is invoked:

```
sage: p = MixedIntegerLinearProgram()
sage: p.set_objective( p[3] + p[2] )
sage: p.add_constraint( p[3] <= 5 )
```

```
sage: p.solve()
Traceback (most recent call last):
  ...
MIPSolverException: GLPK: The LP (relaxation) problem has no dual
    feasible solution
```

```
sage: p.add_constraint( p[2] <= 8 )
sage: p.solve()
13.0
```

```
sage: p.add_constraint( p[3] >= 6 ); p.solve()
Traceback (most recent call last):
  ...
MIPSolverException: GLPK: Problem has no feasible solution
```

Similarly, restricting a variable to the integers can make the domain empty:

```
sage: p = MixedIntegerLinearProgram()
sage: p.set_objective( p[3] )
sage: p.add_constraint( p[3] <= 4.75 ); p.add_constraint( p[3] >= 4.25 )
sage: p.solve()
4.75
sage: p.set_integer(p[3]); p.solve()
Traceback (most recent call last):
  ...
MIPSolverException: GLPK: Problem has no feasible solution
```

In any case, it would be unreasonable to bring a code to a halt whenever the linear program cannot be solved; indeed, the sole objective of certain linear programs is to test the *existence* of a solution, and are in consequence often

infeasible. To handle these scenarios, we will use the classical "*try-except*" Python's mechanism to catch exceptions:

```
sage: try:
....:     p.solve()
....:     print("The problem has a solution!")
....: except:
....:     print("The problem is infeasible!")
The problem is infeasible!
```

17.4 First Applications in Combinatorics

Now that we have discussed the basics, let us consider a more interesting aspect: modelling. In this section we present several optimisation or existence problems: starting with their abstract definition, we continue to model each problem as a MILP, obtaining in a couple of lines of code an algorithm for an NP-complete problem.

17.4.1 Knapsack

The "knapsack problem" is the following: we have in front of us a collection of items having both a weight and a "utility" measured by a real number. We would like to choose some of those objects such that the total weight does not exceed a given constant C, the best way being to optimise the sum of utilities of the objects in the knapsack.

To achieve this, to each object o of a list L we associate a binary variable `taken[o]`, set to 1 if the object is chosen, and 0 otherwise. We are trying to solve the following MILP:

$$\begin{aligned} \text{max:} \quad & \sum_{o \in L} \text{utility}_o \times \text{taken}_o \\ \text{such that:} \quad & \sum_{o \in L} \text{weight}_o \times \text{taken}_o \leq C. \end{aligned}$$

Using Sage, let us assign to our items some cost and utility:

```
sage: C = 1
sage: L = ["Pan", "Book", "Knife", "Flask", "Flashlight"]
sage: w = [0.57,0.35,0.98,0.39,0.08]; u = [0.57,0.26,0.29,0.85,0.23]
sage: weight = {}; utility = {}
sage: for o in L:
....:     weight[o] = w[L.index(o)]; utility[o] = u[L.index(o)]
```

We can now define the MILP itself.

```
sage: p = MixedIntegerLinearProgram()
sage: taken = p.new_variable( binary = True )
```

```
sage: p.add_constraint(
....:   p.sum( weight[o] * taken[o] for o in L ) <= C )
sage: p.set_objective(
....:   p.sum( utility[o] * taken[o] for o in L ) )
sage: p.solve()
1.4199999999999999
sage: taken = p.get_values(taken)
```

We can check that the solution is admissible:

```
sage: sum( weight[o] * taken[o] for o in L )
0.960000000000000
```

Should we take a flask ?

```
sage: taken["Flask"]
1.0
```

Exercise 65 (*Subset Sum*)**.** The combinatorial problem known as *Subset Sum* consists in finding, among a set of integers, a non-empty subset of elements whose sum is zero. Solve this problem with a linear program over the integers for the set $\{28, 10, -89, 69, 42, -37, 76, 78, -40, 92, -93, 45\}$.

17.4.2 Matching

Finding a matching in a graph, amounts to detecting a set of edges which are pairwise disjoint. The empty set being a trivial matching, we focus our attention on maximum matchings: we seek to maximise the number of edges in a matching. Computing the maximum matching is a polynomial problem, which follows from a result of Jack Edmonds [Edm65]. Edmonds' algorithm is based on local improvements and the proof that the algorithm does not halt until a maximum matching is found. This algorithm is not the hardest to implement among those graph theory can offer, though this problem can be modeled with a very simple MILP.

For this task we will need, as in the previous problem, to associate a binary value to each of our objects — the edges of a graph — indicating if the edge belongs to our matching or not.

It is then needed to ensure that two adjacent edges cannot be simultaneously in the matching. Indeed, this looks like a linear constraint: if x and y are two edges of the same graph, and if m_x and m_y are their associated variables, it suffices to require that the inequality $m_x + m_y \leq 1$ is satisfied. Since these two edges will not be found simultaneously in our solution, we are able to write down a linear program which computes the maximum matching. Let us remark that if two edges cannot be in the matching simultaneously, it is because they both have a common vertex v of the graph. It is then simpler to say that at most one edge incident to each vertex v should be taken in the matching, which is again a linear constraint.

$$\text{max: } \sum_{e \in E(G)} m_e$$
$$\text{such that: } \forall v \in V(G), \sum_{\substack{e \in E(G) \\ e=uv}} m_e \leq 1.$$

This problem is readily adapted to a MILP using Sage:

```
sage: g = graphs.PetersenGraph()
sage: p = MixedIntegerLinearProgram()
sage: matching = p.new_variable(binary = True)
```

```
sage: p.set_objective(p.sum(matching[e]
....:                 for e in g.edges(labels = False)))
```

```
sage: for v in g:
....:     p.add_constraint(p.sum(matching[e]
....:           for e in g.edges_incident(v, labels = False)) <= 1)
sage: p.solve()
5.0
```

```
sage: matching = p.get_values(matching)
sage: [e for e, b in matching.iteritems() if b == 1]
[(0, 1), (4, 9), (6, 8), (5, 7), (2, 3)]
```

Exercise 66 (Dominating set)**.** A dominating set in a graph is a set of vertices S such that each vertex which is not in S has at least one neighbour in S. Write a linear program over the integers to find a dominating set whose cardinality is minimal for the Petersen graph.

17.4.3 Flow

In this section we present yet another fundamental algorithm in graph theory: maximum flow! Given a pair of vertices s and t of a *directed* graph G (that is, the edges have a direction, see Figure 17.1), this problem consists in sending from s to t a maximum *flow*, using the edges of G. Each one of these edges has an associated maximal *capacity* — i.e., the maximal flow which can go through it.

The definition of this problem is almost its formulation as a linear program: we are looking for a real value associated to each edge, representing the intensity of flow going through it, under two types of constraints:

- the amount of flow arriving at a vertex (different from s or t) should equal the amount of flow leaving it;
- the flow over an edge cannot exceed its capacity.

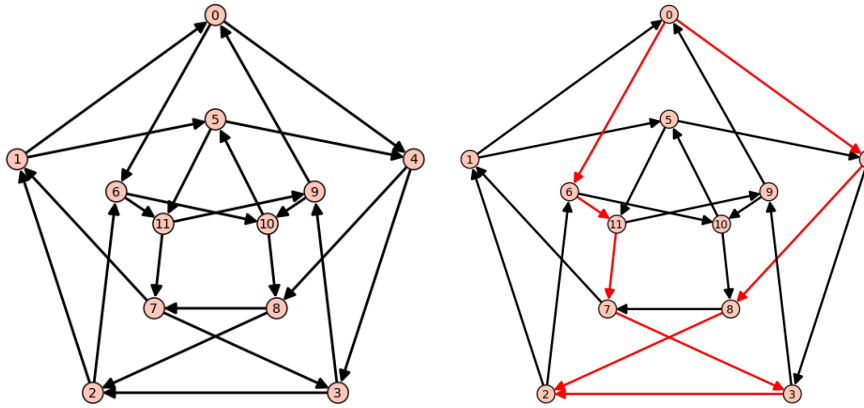


FIGURE 17.1 – A maximum flow problem over Chvátal's graph.

This being said, we are left with the task of maximising the flow leaving s: all of it will end up in t, as the other vertices are sending just as much as they receive. We can model the flow problem with the following linear program (assuming that the capacities of the edges are all equal to 1):

$$\begin{aligned} \text{max:} \quad & \sum_{sv \in E(G)} f_{sv} \\ \text{such that:} \quad & \forall v \in V(G) \backslash \{s,t\}, \sum_{vu \in E(G)} f_{vu} = \sum_{uv \in E(G)} f_{uv} \\ & \forall uv \in E(G), f_{uv} \leq 1. \end{aligned}$$

We will solve the flow problem over an orientation of Chvátal's graph (cf. Figure 17.1), in which all edges have a capacity of 1:

```
sage: g = graphs.ChvatalGraph()
sage: g = g.minimum_outdegree_orientation()
```

```
sage: p = MixedIntegerLinearProgram()
sage: f = p.new_variable()
sage: s, t = 0, 2
```

```
sage: for v in g:
....:     if v == s or v == t: continue
....:     p.add_constraint(
....:           p.sum(f[v,u] for u in g.neighbors_out(v)) ==
....:           p.sum(f[u,v] for u in g.neighbors_in(v)))
```

```
sage: for e in g.edges(labels = False): p.add_constraint( f[e] <= 1 )
```

```
sage: p.set_objective(p.sum( f[s,u] for u in g.neighbors_out(s)))
```

```
sage: p.solve()
2.0
```

17.5 Generating Constraints — Application to the Traveling Salesman Problem

Even though the examples presented in previous sections seem to offer a great deal of expressive power, the "interpretation" of an optimisation problem (or an existence problem) given by its formulation as a linear program is a rather arbitrary choice. The same problem can be solved via different formulations, and the performance among them can differ considerably. We are led to taking advantage of the capacities of MILP solvers in a smarter way, by asking them to solve linear programs without specifying all constraints, and adding only those which are necessary as long as the solution is approached: this technique is indeed essential if the number of constraints is too big for writing them down explicitly when we create the linear program. We are preparing to solve the Hamiltonian cycle problem (a particular case of the *traveling salesman problem*).

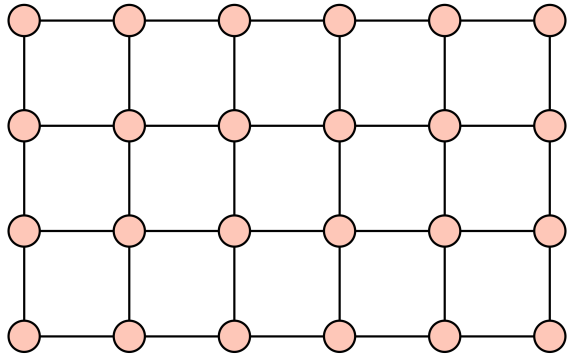


FIGURE 17.2 – A grid of size 4×6 used to test our implementations.

We say that a cycle $C \subseteq E(G)$ which is contained in a graph G is Hamiltonian if it visits all vertices of G. Testing the existence of a Hamiltonian cycle in a given graph is an NP-complete problem: in consequence, we should not expect to solve this problem promptly, although we can still attempt to model it as a linear program. Consider the following initial formulation:

- associate to each edge a binary variable b_e which indicates if the edge is included in the circuit C or not;
- impose that each vertex should have exactly two of its incident edges in C.

Unfortunately, this is not an exact formulation. Indeed, it can happen that the solution obtained with this formulation is a disjoint union of several cycles — each vertex would have two neighbours in C, but it may not be possible to go from vertex v to a vertex u using only the edges which belong to C.

However, it is possible to provide a more complex *and* exact algorithm (known as Miller-Tucker-Zemlin's formulation):

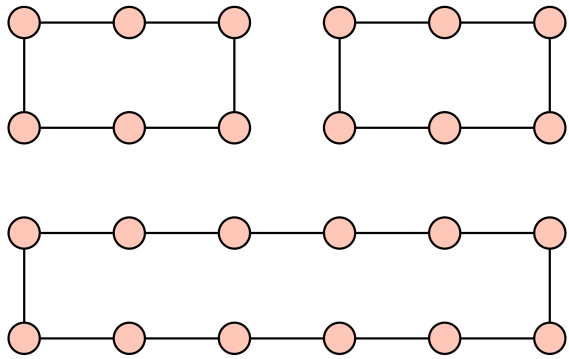


FIGURE 17.3 – A possible solution of the inexact formulation.

- to each vertex v of the graph, we associate an integer i_v representing the stage at which the cycle C is visiting it, with $i_{v_0} = 0$ for a fixed vertex v_0;
- to each edge uv of G we associate two binary variables b_{uv} and b_{vu} indicating if the edge belongs to the cycle C (and to know in which *direction* this edge is being used);
- we impose that each vertex should have an outgoing and incoming edge in C;
- an edge uv belongs to C only if $i_u < i_v$ (the edges go in the increasing sense with respect to the order in which they are visited).

We can rewrite this algorithm in terms of linear equations in a few lines:

$$
\begin{aligned}
\text{max: } & \text{no objective function} \\
\text{such that: } & \forall u \in V(G), \sum_{uv \in E(G)} b_{uv} = 1 \\
& \forall u \in V(G), \sum_{uv \in E(G)} b_{vu} = 1 \\
& \forall uv \in E(G \backslash v_0), && i_u - i_v + |V(G)| b_{uv} \leq |V(G)| - 1 \\
& && i_v - i_u + |V(G)| b_{vu} \leq |V(G)| - 1 \\
& \forall v \in V(G), i_v \leq |V(G)| - 1 \\
& b_{uv} \text{ is a binary variable} \\
& i_v \text{ is an integer variable.}
\end{aligned}
$$

In this formulation, there is a coefficient $|V(G)|$, which often indicates that the solver will not be able to efficiently solve the linear program. Therefore, we will use an alternative modelling approach for the Hamiltonian cycle. Consider the following simple observation: if there exists a Hamiltonian cycle C in our graph, then there exists for every proper subset S of vertices at least two edges

of C, which enter or leave S. If we denote by $\bar{S}$ the set of edges having exactly one end in S, then we obtain the following formulation (identifying the variables b_{uv} and b_{vu}):

$$\begin{aligned} \text{max: } & \text{no objective function} \\ \text{such that: } & \forall u \in V(G), \sum_{uv \in E(G)} b_{uv} = 2 \\ & \forall S \subseteq V(G), \emptyset \neq S \neq V(G), \sum_{e \in \bar{S}} v_e \geq 2 \\ & b_{uv} \text{ is a binary variable.} \end{aligned}$$

It would be unlikely that we can directly use the previous formulation to solve a Hamiltonian cycle problem, even for a small grid such as the one with $4 \times 6 = 24$ elements: the constraints over the sets S would be $2^{24} - 2 = 16\,777\,214$. On the other hand, the *branch-and-bound* method (or *branch-and-cut*) used by linear inequality solvers is well adapted to generating constraints *during* the resolution of the linear program[1]. Generating constraints for the Hamiltonian cycle problem corresponds to the following steps:

- create a linear program — without objective function — having one binary variable per edge;
- for each vertex add a constraint imposing a degree 2;
- solve the linear program;
- while the current solution is not a Hamiltonian cycle (it is then a subgraph having several connected components), let S be one of its connected components, and add the constraint imposing that at least two edges leave S.

Fortunately for us, it is algorithmically fast to verify that the current solution is invalid and to generate the corresponding constraint. Using the method of generating constraints with Sage, here is how we can solve the Hamiltonian cycle problem over our grid:

```
sage: g = graphs.Grid2dGraph(4, 6)
sage: p = MixedIntegerLinearProgram()
sage: b = p.new_variable(binary = True)
```

To avoid the difference between the variables `b[u,v]` and `b[v,u]`, it is convenient to create a lambda-function replacing the pair x, y with the set $\{x, y\}$:

```
sage: B = lambda x,y : b[frozenset([x,y])]
```

[1]This means that it is possible, once the linear program is solved, to add an additional constraint and to solve the new linear program using some of the results obtained during the previous computation.

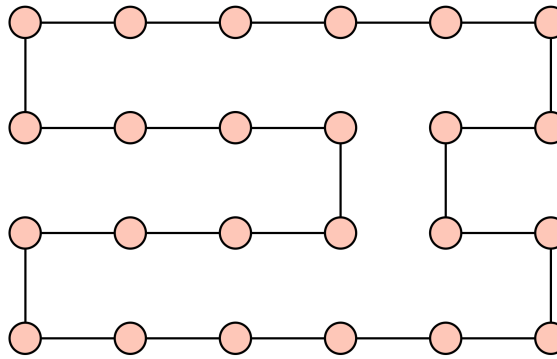


FIGURE 17.4 – A Hamiltonian cycle computed by generating constraints.

Let us now add the degree constraints:

```
sage: for u in g:
....:   p.add_constraint( p.sum( B(u,v) for v in g.neighbors(u) ) == 2 )
```

It is now time to compute the first solution of our problem and to create the graph representing it,

```
sage: p.solve()
0.0
sage: h = Graph()
sage: h.add_edges( [(u,v) for u, v in g.edges(labels = False)
....:                 if p.get_values(B(u,v)) == 1.0 ] )
```

then we begin our iterations:

```
sage: while not h.is_connected():
....:     S = h.connected_components()[0]
....:     p.add_constraint(
....:            p.sum( B(u,v) for u,v
....:                 in g.edge_boundary(S, labels = False))
....:                >= 2)
....:     zero = p.solve()
....:     h = Graph()
....:     h.add_edges( [(u,v) for u,v in
....:                   g.edges(labels = False)
....:                if p.get_values(B(u,v)) == 1.0 ] )
```

In less than a dozen iterations (an interesting economy of computations with respect to $2^{24} - 2$) we obtain an admissible solution (cf. Figure 17.4). In terms of performance, this solution method exceeds that of Miller-Tucker-Zemlin's formulation. When we implement both linear programs in Sage, for a random graph $\mathcal{G}_{35,0.3}$ the computation times are the following:

```
sage: g = graphs.RandomGNP(35, 0.3)
```

```
sage: %time MTZ(g)
CPU times: user 51.52 s, sys: 0.24 s, total: 51.76 s
Wall time: 52.84 s
sage: %time constraint_generation(g)
CPU times: user 0.23 s, sys: 0.00 s, total: 0.23 s
Wall time: 0.26 s
```

Annexes

A Answers to Exercises

A.1 First Steps

Exercise 1 page 14. The command `SR.var('u')` creates the symbolic variable u and assigns it to the computer variable `u`. The computer variable `u` receives twice in a row its current value plus one, that is $u+1$, then $u+2$ (where u remains the symbolic variable).

A.2 Analysis and Algebra

Exercise 2 page 28. (*Computing a sum by recurrence*)

```
sage: n = var('n'); pmax = 4; s = [n + 1]
sage: for p in [1..pmax]:
....:     s += [factor(((n+1)^(p+1) - sum(binomial(p+1, j) * s[j]
....:           for j in [0..p-1])) / (p+1))]
sage: s
```

Thus we obtain:

$$\sum_{k=0}^{n} k = \frac{1}{2}\,(n+1)n, \qquad \sum_{k=0}^{n} k^2 = \frac{1}{6}\,(n+1)(2\,n+1)n,$$
$$\sum_{k=0}^{n} k^3 = \frac{1}{4}\,(n+1)^2 n^2, \qquad \sum_{k=0}^{n} k^4 = \frac{1}{30}\,(n+1)(2\,n+1)\big(3\,n^2+3\,n-1\big)n.$$

Exercise 3 page 31. (*Computing a symbolic limit*) To answer this question, we use a symbolic function, and we compute its Taylor polynomial at 0 up to order 3:

```
sage: x, h, a = var('x, h, a'); f = function('f')
sage: g(x) = taylor(f(x), x, a, 3)
sage: phi(h) = (g(a+3*h) - 3*g(a+2*h) + 3*g(a+h) - g(a)) / h^3; phi(h)
diff(f(a), a, a, a)
```

The function g differs from f by a rest which is negligible compared to h^3; thus the function `phi` differs from the quotient by $o(1)$; consequently `phi` has the wanted limit at 0. As a conclusion,

$$\lim_{h\to 0} \frac{1}{h^3}\big(f(a+3h) - 3f(a+2h) + 3f(a+h) - f(a)\big) = f'''(a).$$

This formula allows an approximate computation of the third derivative of f without doing any derivation.

We can conjecture that the formula can be generalised in the following form:

$$\lim_{h\to 0} \frac{1}{h^n}\left(\sum_{k=0}^{n}(-1)^{n-k}\binom{n}{k} f(a+kh)\right) = f^{(n)}(a).$$

To verify this formula for larger values of n, we can easily adapt the preceding computation:

```
sage: n = 7; x, h, a = var('x h a'); f = function('f')
sage: g(x) = taylor(f(x), x, a, n)
sage: sum((-1)^(n-k) * binomial(n,k) * g(a+k*h) for k in (0..n)) / h^n
diff(f(a), a, a, a, a, a, a, a)
```

Exercise 4 page 31. (*A formula due to Gauss*)

1. We use successively `trig_expand` and `trig_simplify`:

   ```
   sage: theta = 12*arctan(1/38) + 20*arctan(1/57) \
   ....:       + 7*arctan(1/239) + 24*arctan(1/268)
   sage: tan(theta).trig_expand().trig_simplify()
   1
   ```

2. The arc-tangent function is concave on $[0, +\infty[$, thus $\forall x \geq 0, \arctan x \leq x$.

   ```
   sage: 12*(1/38) + 20*(1/57) + 7*(1/239) + 24*(1/268)
   37735/48039
   ```

   From this we deduce:

   $$\begin{aligned}\theta &= 12\arctan\frac{1}{38} + 20\arctan\frac{1}{57} + 7\arctan\frac{1}{239} + 24\arctan\frac{1}{268}\\ &\leq 12\cdot\frac{1}{38} + 20\cdot\frac{1}{57} + 7\cdot\frac{1}{239} + 24\cdot\frac{1}{268}\\ &= \frac{37735}{48039} < \frac{\pi}{2}.\end{aligned}$$

   Thus $0 < \theta < \pi/2$; also (cf. question 1) $\tan\theta = 1 = \tan\frac{\pi}{4}$ and tan is injective on $[0, \pi/2[$. We deduce that $\theta = \frac{\pi}{4}$.

3. We substitute the Taylor polynomial in Gauss formula:

```
sage: x = var('x'); f(x) = taylor(arctan(x), x, 0, 21)
sage: approx = 4 * (12 * f(1/38) + 20 * f(1/57)
....:           + 7 * f(1/239) + 24 * f(1/268))
sage: approx.n(digits = 50); pi.n(digits = 50)
3.1415926535897932384626433832795028851616168852864
3.1415926535897932384626433832795028841971693993751
sage: approx.n(digits = 50) - pi.n(digits = 50)
9.6444748591132486785420917537404705292978817080880e-37
```

Exercise 5 page 32. (*Asymptotic expansion of a sequence*) One can easily show that $x_n \sim n\pi$, that is $x_n = n\pi + o(n)$.

We inject this equality into the following equation, which we obtain from $\arctan x + \arctan(1/x) = \pi/2$:

$$x_n = n\pi + \frac{\pi}{2} - \arctan\left(\frac{1}{x_n}\right).$$

We then inject the asymptotic expansions of x_n obtained in this equation, and so on (method of successive refinements).

As we know that at each step, an order-p expansion allows to get an order-$(p+2)$ expansion, we obtain, in four steps, an expansion at order 6. Anticipating on Chapter 3, we can program these four steps into a loop:

```
sage: n = var('n'); phi = lambda x: n*pi + pi/2 - arctan(1/x)
sage: x = n*pi
sage: for i in range(4):
....:     x = taylor(phi(x), n, infinity, 2*i); x
```

Finally, we obtain:

$$x_n = \frac{1}{2}\,\pi + \pi n - \frac{1}{\pi n} + \frac{1}{2}\,\frac{1}{\pi n^2} - \frac{1}{12}\,\frac{3\,\pi^2+8}{\pi^3 n^3} + \frac{1}{8}\,\frac{\pi^2+8}{\pi^3 n^4} - \frac{1}{240}\,\frac{15\,\pi^4+240\,\pi^2+208}{\pi^5 n^5} + \frac{1}{96}\,\frac{3\,\pi^4+80\,\pi^2+208}{\pi^5 n^6} + O\left(\frac{1}{n^7}\right).$$

Exercise 6 page 33. (*A counter-example to Schwarz theorem due to Peano*) The partial applications $f(x,0)$ and $f(0,x)$ are identically zero in $(0,0)$; without any computation we deduce that $\partial_1 f(0,0) = \partial_2 f(0,0) = 0$. Then we compute the second order partial derivative in $(0,0)$:

```
sage: h = var('h'); f(x, y) = x * y * (x^2 - y^2) / (x^2 + y^2)
sage: D1f(x, y) = diff(f(x,y), x); limit((D1f(0,h) - 0) / h, h=0)
-1
sage: D2f(x, y) = diff(f(x,y), y); limit((D2f(h,0) - 0) / h, h=0)
1
sage: g = plot3d(f(x, y), (x, -3, 3), (y, -3, 3))
```

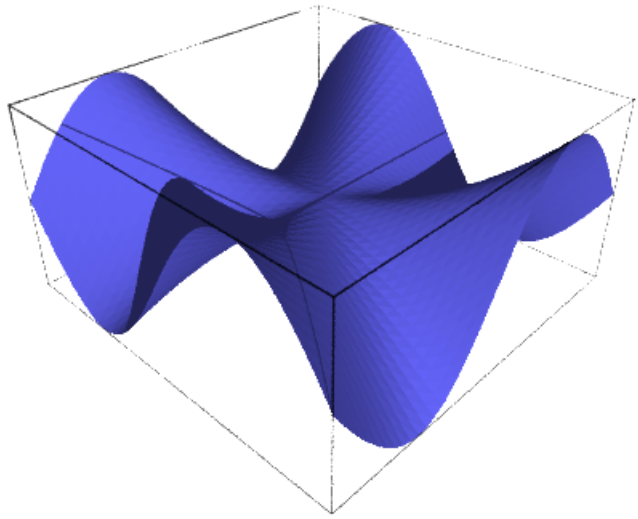


FIGURE A.1 – The Peano surface.

We deduce that $\partial_1\partial_2 f(0,0) = 1$ and $\partial_2\partial_1 f(0,0) = -1$. Thus, we get a counter-example to Schwarz theorem (Figure A.1).

Exercise 7 page 34. (*The BBP formula*)

1. Let us first compare

$$u_n = \int_0^{1/\sqrt{2}} f(t)\cdot t^{8n}\,\mathrm{d}t \quad \text{and} \quad v_n = \left(\tfrac{4}{8n+1} - \tfrac{2}{8n+4} - \tfrac{1}{8n+5} - \tfrac{1}{8n+6}\right)\left(\tfrac{1}{16}\right)^n.$$

```
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)
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
```

We deduce that $u_n = v_n$. By the linearity of the integral, we get:

$$I_N = \int_0^{1/\sqrt{2}} f(t)\cdot\left(\sum_{n=0}^{N} t^{8n}\right)\mathrm{d}t = \sum_{n=0}^{N} u_n = \sum_{n=0}^{N} v_n = S_N.$$

2. The radius of convergence of the power series $\sum_{n\geq 0} t^{8n}$ is 1, thus it converges on the interval $\left[0, \frac{1}{\sqrt{2}}\right]$. We can interchange integration and limit on this

interval:

$$\begin{aligned}\lim_{N\to\infty} S_N &= \lim_{N\to\infty} \int_0^{1/\sqrt{2}} f(t)\cdot\left(\sum_{n=0}^{N} t^{8n}\right)\mathrm{d}t \\ &= \int_0^{1/\sqrt{2}} f(t)\cdot\left(\sum_{n=0}^{\infty} t^{8n}\right)\mathrm{d}t \\ &= \int_0^{1/\sqrt{2}} f(t)\cdot\frac{1}{1-t^8}\,\mathrm{d}t = J.\end{aligned}$$

3. Now, we compute J:

```
sage: t = var('t'); 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)
```

To simplify this expression, we must tell Sage to combine the sum of logarithms:

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

At the end, we get the formula:

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

With this formula, we get another way to approximate π:

```
sage: l = sum(v(n) for n in (0..40)); l.n(digits=60)
3.14159265358979323846264338327950288419716939937510581474759
sage: pi.n(digits=60)
3.14159265358979323846264338327950288419716939937510582097494
sage: print("%e" % (l-pi).n(digits=60))
-6.227358e-54
```

Exercise 8 page 35. (*Polynomial approximation of the sine function*) We provide the vector space $\mathcal{C}^\infty([-\pi,\pi])$ with the dot product $\langle f\,|\,g\rangle = \int_{-\pi}^{\pi} fg$. The wanted polynomial is the orthogonal projection of the sine function on the vector subspace $\mathbb{R}_5[X]$. Finding this polynomial reduces to the solution of a linear system: indeed, P is the projection of the sine function if and only if the function $(P-\sin)$ is orthogonal to every vector of the canonical basis of $\mathbb{R}_5[X]$. Here is the Sage code:

```
sage: x = var('x'); ps = lambda f, g : integral(f * g, x, -pi, pi)
sage: n = 5; a = var('a0, a1, a2, a3, a4, a5')
sage: P = sum(a[k] * x^k for k in (0..n))
sage: equ = [ps(P - sin(x), x^k) for k in (0..n)]
sage: sol = solve(equ, a)
```

```
sage: P = sum(sol[0][k].rhs() * x^k for k in (0..n)); P
105/8*(pi^4 - 153*pi^2 + 1485)*x/pi^6 - 315/4*(pi^4 - 125*pi^2 +
1155)*x^3/pi^8 + 693/8*(pi^4 - 105*pi^2 + 945)*x^5/pi^10
sage: g = plot(P,x,-6,6,color='red') + plot(sin(x),x,-6,6,color='blue')
sage: g.show(ymin = -1.5, ymax = 1.5)
```

The wanted polynomial is:

$$P = \frac{105}{8}\,\frac{\pi^4 - 153\,\pi^2 + 1485}{\pi^6}x - \frac{315}{4}\,\frac{\pi^4 - 125\,\pi^2 + 1155}{\pi^8}x^3 + \frac{693}{8}\,\frac{\pi^4 - 105\,\pi^2 + 945}{\pi^{10}}x^5.$$

Then, we can plot the sine function and its orthogonal projection to see the quality of this polynomial approximation (Figure A.2).

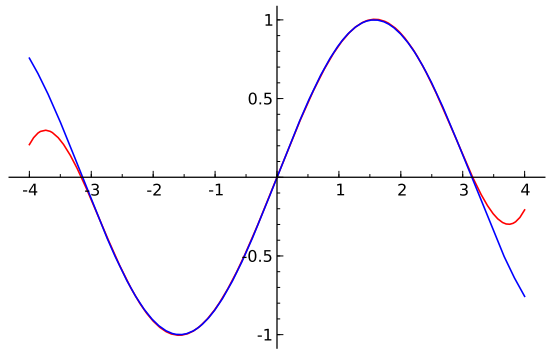


FIGURE A.2 – Least square approximation of the sine function.

Exercise 9 page 35. (*Gauss' problem*) Let us first prove formally the relations; then we will perform the numerical application. We first define the vectors $\overrightarrow{r_i}$:

```
sage: p, e = var('p, e')
sage: theta1, theta2, theta3 = var('theta1, theta2, theta3')
sage: r(theta) = p / (1 - e * cos(theta))
sage: r1 = r(theta1); r2 = r(theta2); r3 = r(theta3)
sage: R1 = vector([r1 * cos(theta1), r1 * sin(theta1), 0])
sage: R2 = vector([r2 * cos(theta2), r2 * sin(theta2), 0])
sage: R3 = vector([r3 * cos(theta3), r3 * sin(theta3), 0])
```

- We verify that $\overrightarrow{S} + e \cdot \left(\overrightarrow{\imath} \wedge \overrightarrow{D}\right)$ is the null vector:

```
sage: D = R1.cross_product(R2)+R2.cross_product(R3)+R3.cross_product(R1)
sage: S = (r1 - r3) * R2 + (r3 - r2) * R1 + (r2 - r1) * R3
sage: i = vector([1, 0, 0]); V =  S + e * i.cross_product(D)
sage: V.simplify_full()
(0, 0, 0)
```

from which we get the wanted relation. We deduce $e = \frac{\left\| \overrightarrow{S} \right\|}{\left\| \overrightarrow{\imath} \wedge \overrightarrow{D} \right\|} = \frac{\left\| \overrightarrow{S} \right\|}{\left\| \overrightarrow{D} \right\|}$, as $\overrightarrow{D}$ is normal to the orbit plane, and thus to $\overrightarrow{\imath}$.

- Then we verify that $\overrightarrow{\imath}$ is colinear with $\overrightarrow{S} \wedge \overrightarrow{D}$:

```
sage: S.cross_product(D).simplify_full()[1:3]
(0, 0)
```

The result shows that the second and third components are zero.

- We also verify that $p \cdot \overrightarrow{S} + e \cdot \left(\overrightarrow{\imath} \wedge \overrightarrow{N} \right)$ is the null vector,

```
sage: N = r3 * R1.cross_product(R2) + r1 * R2.cross_product(R3)\
....:   + r2 * R3.cross_product(R1)
sage: W = p * S + e * i.cross_product(N)
sage: W.simplify_full()
(0, 0, 0)
```

from which we get the required relation. We deduce that:

$$p = e\frac{\left\| \overrightarrow{\imath} \wedge \overrightarrow{N} \right\|}{\left\| \overrightarrow{S} \right\|} = e\frac{\left\| \overrightarrow{N} \right\|}{\left\| \overrightarrow{S} \right\|} = \frac{\left\| \overrightarrow{N} \right\|}{\left\| \overrightarrow{D} \right\|},$$

as $\overrightarrow{N}$ is normal to the orbit plane, and thus to $\overrightarrow{\imath}$.

- From a classical property of conic curves, we have $a = \frac{p}{1-e^2}$.
- Now, let us perform the numerical application:

```
sage: R1=vector([0,1,0]); R2=vector([2,2,0]); R3=vector([3.5,0,0])
sage: r1 = R1.norm(); r2 = R2.norm(); r3 = R3.norm()
sage: D = R1.cross_product(R2) + R2.cross_product(R3) \
....:   + R3.cross_product(R1)
sage: S = (r1 - r3) * R2 + (r3 - r2) * R1 + (r2 - r1) * R3
sage: N = r3 * R1.cross_product(R2) + r1 * R2.cross_product(R3) \
....:   + r2 * R3.cross_product(R1)
sage: e = S.norm() / D.norm(); p = N.norm() / D.norm()
sage: a = p/(1-e^2); c = a * e; b = sqrt(a^2 - c^2)
sage: X = S.cross_product(D); i = X / X.norm()
sage: phi = atan2(i[1], i[0]) * 180 / pi.n()
sage: print("%.3f %.3f %.3f %.3f %.3f %.3f" % (a, b, c, e, p, phi))
2.360 1.326 1.952 0.827 0.746 17.917
```

Thus, we finally find:

$$a \approx 2.360, \quad b \approx 1.326, \quad c \approx 1.952, \quad e \approx 0.827, \quad p \approx 0.746, \quad \varphi \approx 17.917.$$

The inclination of the ellipse major axis is 17.92 degrees.

Exercise 10 page 36. (*Basis of vector subspace*)

1. The set $\mathcal{S}$ of the solutions of the homogeneous system associated to A is a vector subspace of $\mathbb{R}^5$. We obtain the dimension and a basis of $\mathcal{S}$ with the function `right_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()
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]
```

So, $\mathcal{S}$ is the vector plane generated by the two above vectors (read them line by line, as below).

2. We extract from the given generating family a basis of the wanted vector space as follows. We reduce the matrix A (made with the columns u_1, u_2, u_3, u_4, u_5) row-by-row until we get the Hermite form:

```
sage: H = A.echelon_form(); H
```

$$\begin{pmatrix} 1 & 0 & 4 & 0 & -3 \\ 0 & 1 & 2 & 0 & 7 \\ 0 & 0 & 0 & 1 & -5 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Let $F = \mathrm{Vect}(u_1, u_2, u_3, u_4, u_5)$ be the family of column vectors of A. It is a vector subspace of $\mathbb{R}^4$. Looking at H, we observe that the pivots belong to columns 1, 2 and 4. More precisely, we have:

$$\begin{cases} (u_1, u_2, u_4) \text{ is a free family,} \\ u_3 = 4u_1 + 2u_2, \\ u_5 = -3u_1 + 7u_2 - 5u_4. \end{cases}$$

Thus $F = \mathrm{Vect}(u_1, u_2, u_3, u_4, u_5) = \mathrm{Vect}(u_1, u_2, u_4)$ is generated by the free family (u_1, u_2, u_4); thus (u_1, u_2, u_4) is a basis of F. More directly, we could also use the `column_space` method:

```
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]
```

3. Now, we are looking for equations of the generated subspace: we reduce the matrix A, augmented by a right-hand side, computing with Sage in a ring of polynomials with four variables:

```
sage: S.<x, y, z, t> = QQ[]
sage: C = matrix(S, 4, 1, [x, y, z, t])
sage: B = block_matrix([A, C], ncols=2)
```

```
sage: C = B.echelon_form()
sage: C[3,5]*350
```

$-1139x + 63y + 120z + 350t$

We deduce that F is the hyperplane of $\mathbb{R}^4$ defined by

$$-1139\,x + 63\,y + 120\,z + 350\,t = 0.$$

It is also possible to get this equation by computing the left kernel of A, which gives the coordinates of the linear forms defining F (here, there is only one form):

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

A basis of the hyperplane defined by this linear form is given by the following vectors already obtained by `A.column_space()`:

```
sage: matrix(K.0).right_kernel()
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]
```

Exercise 11 page 37. (*A matrix equation*) Let us first define the A and C matrices:

```
sage: A = matrix(QQ, [[-2, 1, 1], [8, 1, -5], [4, 3, -3]])
sage: C = matrix(QQ, [[1, 2, -1], [2, -1, -1], [-5, 0, 3]])
```

As the equation $A = BC$ is linear, the set of solutions is an affine subspace of $\mathcal{M}_3(\mathbb{R})$. We search for a particular solution of our equation.

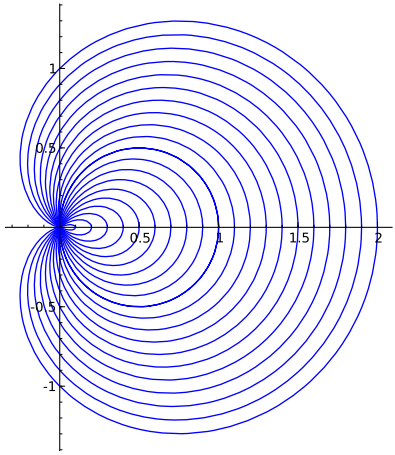
```
sage: B = C.solve_left(A); B
[ 0 -1  0]
[ 2  3  0]
[ 2  1  0]
```

Then, we determine the general form of the solutions, that is to say, the left kernel of C:

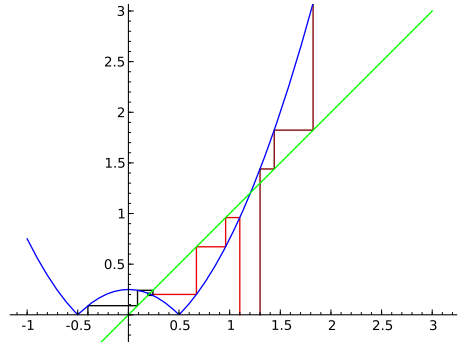
```
sage: C.left_kernel()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[1 2 1]
```

Then, we obtain the general form of the solutions of our equation:

```
sage: x, y, z = var('x, y, z'); v = matrix([[1, 2, 1]])
sage: B = B + (x*v).stack(y*v).stack(z*v); B
```



(a) Pascal conchoids.



(b) Terms of a recurrent sequence.

```
[      x 2*x - 1       x]
[  y + 2 2*y + 3       y]
[  z + 2 2*z + 1       z]
```

It is easy to check the result:

```
sage: A == B*C
True
```

In conclusion, the set of the solutions is a 3-dimensional affine subspace:

$$\left\{ \begin{pmatrix} x & 2\,x-1 & x \\ y+2 & 2\,y+3 & y \\ z+2 & 2\,z+1 & z \end{pmatrix} \;\middle|\; (x,y,z) \in \mathbb{R}^3 \right\}.$$

A.4 Graphics

Exercise 12 page 79. (*Pascal conchoids*)

```
sage: t = var('t'); liste = [a + cos(t) for a in srange(0, 2, 0.1)]
sage: g = polar_plot(liste, (t, 0, 2 * pi)); g.show(aspect_ratio = 1)
```

Exercise 13 page 82. (*Drawing the terms of a recurrent sequence*)

```
sage: f = lambda x: abs(x**2 - 1/4)
sage: def liste_pts(u0, n):
....:     u = u0; liste = [[u0,0]]
....:     for k in range(n):
....:         v, u = u, f(u)
....:         liste.extend([[v,u], [u,u]])
....:     return(liste)
sage: g = line(liste_pts(1.1, 8), rgbcolor = (.9,0,0))
```

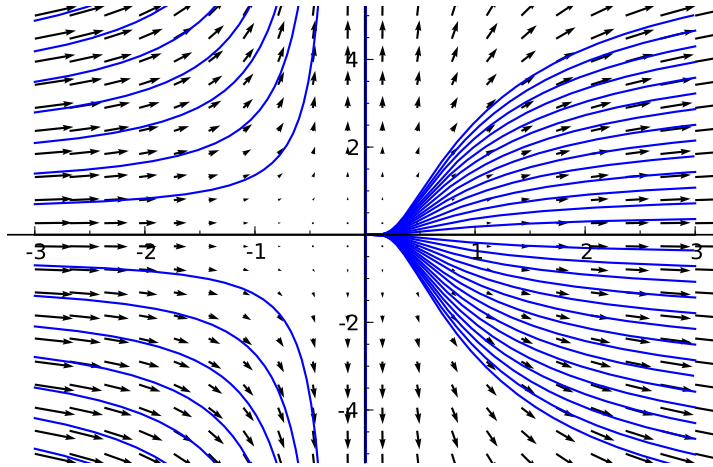


FIGURE A.3 – Integral curves of $x^2y' - y = 0$.

```
sage: g += line(liste_pts(-.4, 8), rgbcolor = (.01,0,0))
sage: g += line(liste_pts(1.3, 3), rgbcolor = (.5,0,0))
sage: g += plot(f, -1, 3, rgbcolor = 'blue')
sage: g += plot(x, -1, 3, rgbcolor = 'green')
sage: g.show(aspect_ratio = 1, ymin = -.2, ymax = 3)
```

Exercise 14 page 85. (*First order differential equation, resolved*)

```
sage: x = var('x'); y = function('y')
sage: DE = x^2 * diff(y(x), x) - y(x) == 0
sage: desolve(DE, y(x))
_C*e^(-1/x)
sage: g = plot([c*e^(-1/x) for c in srange(-8, 8, 0.4)], (x, -3, 3))
sage: y = var('y')
sage: g += plot_vector_field((x^2, y), (x,-3,3), (y,-5,5))
sage: g.show()
```

Exercise 15 page 87. (*Predator-prey model*)

```
sage: from sage.calculus.desolvers import desolve_system_rk4
sage: f = lambda x, y: [a*x-b*x*y,-c*y+d*b*x*y]
sage: x, y, t = var('x, y, t')
sage: a, b, c, d = 1., 0.1, 1.5, 0.75
sage: P = desolve_system_rk4(f(x,y), [x,y],\
....:         ics=[0,10,5], ivar=t, end_points=15)
sage: Ql = [[i,j] for i,j,k in P]; p = line(Ql, color='red')
sage: p += text("Rabbits", (12,37), fontsize=10, color='red')
sage: Qr = [[i,k] for i,j,k in P]; p += line(Qr, color='blue')
sage: p += text("Foxes", (12,7), fontsize=10, color='blue')
sage: p.axes_labels(["time", "population"])
```

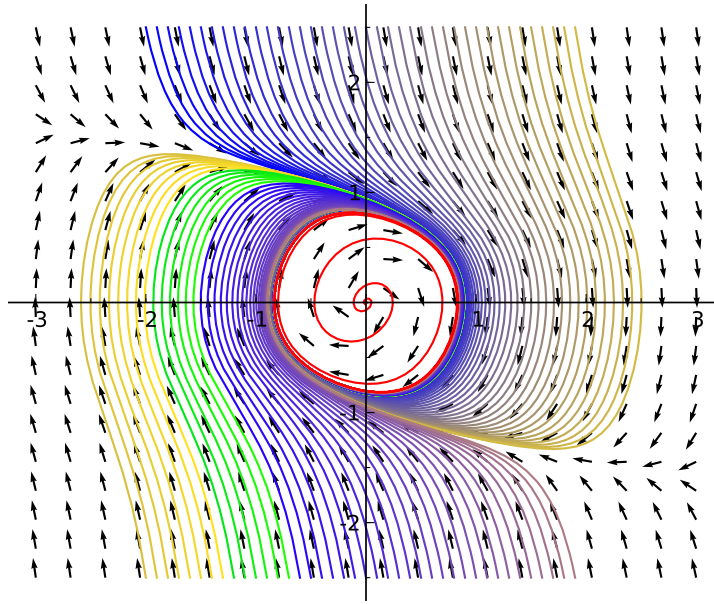


FIGURE A.4 – An autonomous differential system.

```
sage: p.show(gridlines = True)
```

One can also redo the right-hand graphic of Figure 4.12:

```
sage: n = 10;  L = srange(6, 18, 12 / n); R = srange(3, 9, 6 / n)
sage: def g(x,y): v = vector(f(x, y)); return v / v.norm()
sage: q = plot_vector_field(g(x, y), (x, 0, 60), (y, 0, 36))
sage: for j in range(n):
....:     P = desolve_system_rk4(f(x,y), [x,y],
....:           ics=[0,L[j],R[j]], ivar=t, end_points=15)
....:     Q = [[j,k] for i,j,k in P]
....:     q += line(Q, color=hue(.8-j/(2*n)))
sage: q.axes_labels(["rabbits", "foxes"]); q.show()
```

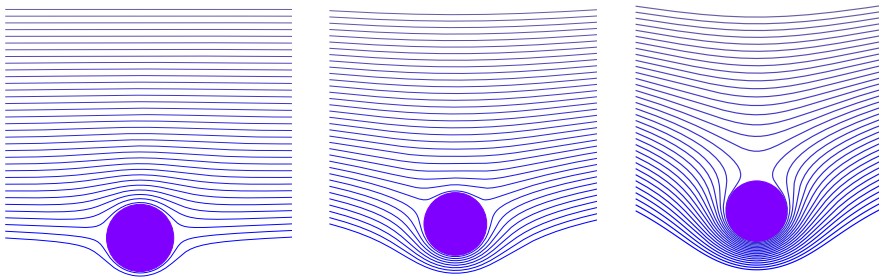
Exercise 16 page 87. (*An autonomous differential system*)

```
sage: from scipy import integrate
sage: def dX_dt(X, t=0): return [X[1], 0.5*X[1] - X[0] - X[1]^3]
sage: t = srange(0, 40, 0.01);  x0 = srange(-2, 2, 0.1); y0 = 2.5
sage: CI = [[i, y0] for i in x0] + [[i, -y0] for i in x0]
sage: def g(x,y): v = vector(dX_dt([x, y])); return v / v.norm()
sage: x, y = var('x, y'); n = len(CI)
sage: q = plot_vector_field(g(x, y), (x, -3, 3), (y, -y0, y0))
sage: for j in xrange(n):
....:     X = integrate.odeint(dX_dt, CI[j], t)
....:     q += line(X, color=(1.7*j/(4*n),1.5*j/(4*n),1-3*j/(8*n)))
```

```
sage: X = integrate.odeint(dX_dt, [0.01,0], t)
sage: q += line(X, color = 'red'); q.show()
```

Exercise 17 page 87. (*Flow around a cylinder and the Magnus effect*)

```
sage: from scipy import integrate
sage: t = srange(0, 40, 0.2)
sage: n = 35; CI_cart = [[4, .2 * i] for i in range(n)]
sage: CI = map(lambda x: [sqrt(x[0]^2+x[1]^2),\
....:      pi - arctan(x[1]/x[0])], CI_cart)
sage: for alpha in [0.1, 0.5, 1, 1.25]:
....:     dX_dt = lambda X, t=0: [cos(X[1])*(1-1/X[0]^2), \
....:              -sin(X[1]) * (1/X[0]+1/X[0]^3) + 2*alpha/X[0]^2]
....:     q = circle((0, 0), 1, fill=True, rgbcolor='purple')
....:     for j in range(n):
....:         X = integrate.odeint(dX_dt, CI[j], t)
....:         Y = [[u[0]*cos(u[1]), u[0]*sin(u[1])] for u in X]
....:         q += line(Y, xmin = -4, xmax = 4, color='blue')
....:     q.show(aspect_ratio = 1, axes = False)
```



(a) Case $\alpha = 0.1$. (b) Case $\alpha = 0.5$. (c) Case $\alpha = 1$.

FIGURE A.5 – The Magnus effect.

The solutions corresponding to $\alpha = 0.1, 0.5, 1$ are shown on Figure A.5.

A.5 Computational Domains

Exercise 18 page 96.

```
sage: def ndigits(x): return x.ndigits()
sage: o = 720; ndigits(o)
3
```

If the object `x` does not have a method `ndigits`, we obtain an error:

```
sage: ndigits("abcd")
...
```

```
AttributeError: 'str' object has no attribute 'ndigits'
```

Exercise 19 page 100. The floating-point numbers `RealField(p)`, with a precision of p bits, all share the same type, but as their parent contains the information about precision, the parents of `RealField(p)` and `RealField(q)` differ:

```
sage: a = Reals(17)(pi); b = Reals(42)(pi)
sage: type(a) == type(b)
True
sage: parent(a), parent(b)
(Real Field with 17 bits of precision, Real Field with 42 bits of
    precision)
```

It is more difficult to find two objects with the same parent but different types. Here is an example:

```
sage: a = 0.1; b = 0.1*1
sage: type(a), type(b)
(<type 'sage.rings.real_mpfr.RealLiteral'>, <type 'sage.rings.real_mpfr.
    RealNumber'>)
sage: parent(a) == parent(b)
True
```

This example needs some explanation. The type `RealLiteral` shows that Sage keeps a in the form of a character string, not doing any floating-point number conversion which would lose its exact value, as $1/10$ cannot be represented exactly in binary. On the contrary, as b is the result of a computation, it is converted into a floating-point number, with the default precision of 53 bits. One can observe the difference by computing $a - 1/10$ and $b - 1/10$ with a precision of 100 bits:

```
sage: Reals(100)(a)-1/10
0.000000000000000000000000000
sage: Reals(100)(b)-1/10
5.5511151231257629805955278152e-18
```

Here is a more advanced example where objects with the same parent have different possible implementations:

```
sage: E = CombinatorialFreeModule(QQ, [1,2,3])
sage: H = Hom(E,E); H.rename("H")
sage: C = E.category(); C
Category of finite dimensional vector spaces with basis over Rational
    Field
sage: phi1 = E.module_morphism(on_basis=lambda i: E.term(i), codomain=E)
sage: phi2 = E.module_morphism(on_basis=lambda i: E.term(i),
....:                          triangular="lower", codomain=E)
sage: phi3 = E.module_morphism(diagonal=lambda i: 1, codomain=E,
....:                          category=C)
sage: phi1.parent() == phi2.parent() == phi3.parent() == H
```

```
True
sage: type(phi1)
<class 'sage.modules.with_basis.morphism.
    ModuleMorphismByLinearity_with_category'>
sage: type(phi2)
<class 'sage.modules.with_basis.morphism.
    TriangularModuleMorphismByLinearity_with_category'>
sage: type(phi3)
<class 'sage.modules.with_basis.morphism.
    DiagonalModuleMorphism_with_category'>
```

A.6 Finite Fields and Number Theory

Exercise 20 page 122. We assume $n = pqr$ with $p < q < r$. Necessarily $p^3 \leq n$, thus the main function becomes:

```
sage: def enum_carmichael(N, verbose=True):
....:     p = 3; s = 0
....:     while p^3 <= N:
....:         s += enum_carmichael_p(N, p, verbose); p = next_prime(p)
....:     return s
```

where the function `enum_carmichael_p` counts Carmichael numbers multiple of p, which are of the form $a + \lambda m$ with λ a non-negative integer, $a = p$ and $m = p(p-1)$, since n should be multiple of p, and $n - 1$ multiple of $p - 1$:

```
sage: def enum_carmichael_p (n, p, verbose):
....:     a = p; m = p*(p-1); q = p; s = 0
....:     while p*q^2 <= n:
....:         q = next_prime(q)
....:         s += enum_carmichael_pq(n, a, m, p, q, verbose)
....:     return s
```

The function `enum_carmichael_pq` counts Carmichael numbers multiple of pq, which are of the form $a' + \mu m'$ with μ a non-negative integer, where $a' \equiv a \bmod m$, $a' \equiv q \bmod q(q-1)$, and m' is multiple both of $m = p(p-1)$ and $q(q-1)$. We use the `crt` function to solve simultaneous modular constraints, while eliminating cases where there is no solution, otherwise Sage would give an error. We also require $a' > pq^2$ to have $r > q$:

```
sage: def enum_carmichael_pq(n,a,m,p,q,verbose):
....:     if (a-q) % gcd(m,q*(q-1)) <> 0: return 0
....:     s = 0
....:     a = crt (a, q, m, q*(q-1)); m = lcm(m,q*(q-1))
....:     while a <= p*q^2: a += m
....:     for t in range(a, n+1, m):
....:         r = t // (p*q)
....:         if is_prime(r) and t % (r-1) == 1:
....:             if verbose:
```

```
....:             print((p*q*r, factor(p*q*r)))
....:         s += 1
....:     return s
```

With these functions, we obtain:

```
sage: enum_carmichael(10^4)
(561, 3 * 11 * 17)
(1105, 5 * 13 * 17)
(2465, 5 * 17 * 29)
(1729, 7 * 13 * 19)
(2821, 7 * 13 * 31)
(8911, 7 * 19 * 67)
(6601, 7 * 23 * 41)
7
sage: enum_carmichael(10^5, False)
12
sage: enum_carmichael(10^6, False)
23
sage: enum_carmichael(10^7, False)
47
```

Exercise 21 page 124. We start by writing a function `aliq` computing the aliquot sequence starting from n, and stopping as soon as one reaches 1 or a cycle:

```
sage: def aliq(n):
....:     l = [n]
....:     while n <> 1:
....:         n = sigma(n) - n
....:         if n in l: break
....:         l.append(n)
....:     return l
sage: l = aliq(840)
sage: len(l), l[:5], l[-5:]
(748, [840, 2040, 4440, 9240, 25320], [2714, 1606, 1058, 601, 1])
```

```
sage: p = points([(i, log(l[i])/log(10)) for i in range(len(l))])
```

See the graph in Figure A.6.

Exercise 22 page 125. (*Masser-Gramain constant*) For question 1, let C be the border circle of a smallest disk. Without loss of generality, we can assume the origin O is on the circle — indeed, there is at least one point of $\mathbb{Z}^2$ on the circle, otherwise the disk is not optimal. We can also assume that the circle center lies in the first quadrant (by rotating the disk if needed by a multiple of $\pi/2$ around O). We will admit that we also have two points A and B of the first quadrant on the circle, thus the circle C includes the triangle OAB. The bound $r_k < \sqrt{k/\pi}$ allows us to bound the points A and B, since their distance to O is at most $2\sqrt{k/\pi}$. We can assume that one of A and B, for example A, lies in the second octant (if both

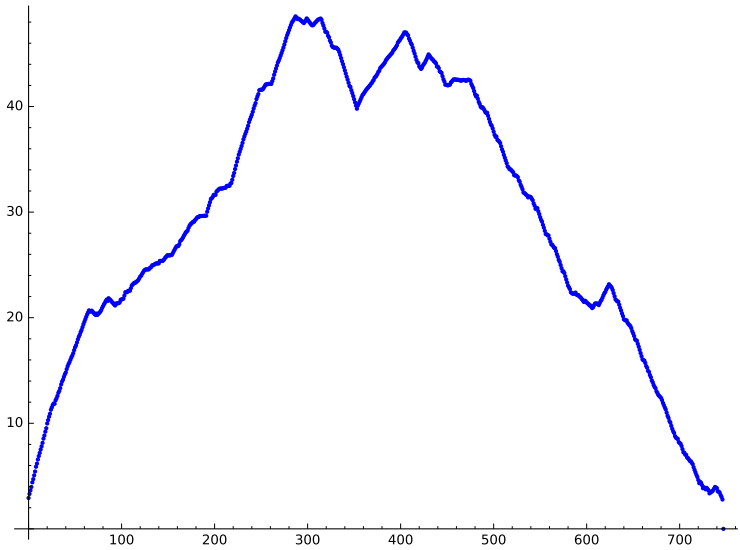


FIGURE A.6 – The graph of aliquot sequence 840.

lie in the first octant, by symmetry with respect to the line $x = y$ we can bring them back in the second octant). We can also assume that the angle in A of the triangle OAB is acute (by exchanging A and B if needed, and after a symmetry with respect to the line $x = y$ if they lie in different octants). The abscissa of A therefore satisfies $x_A < \sqrt{2k/\pi}$, its ordinate satisfies $x_A \leq y_A < \sqrt{4k/\pi - x_A^2}$. For the point B, we have $0 \leq x_B < 2\sqrt{k/\pi}$, and $0 \leq x_A y_B + y_A x_B \leq x_A^2 + y_A^2$ (acute angle in A). This yields the following code, where the `rk_aux` routine computes the number of points in the disk centered in $(x_c/d, y_c/d)$, and of radius $\sqrt{r_2}/d$, where x_c, y_c, d, r_2 are all integers.

```
sage: def rk_aux(xc, yc, d, r2):
....:     s = 0
....:     xmin = ceil((xc - sqrt(r2))/d)
....:     xmax = floor((xc + sqrt(r2))/d)
....:     for x in range(xmin,xmax+1):
....:         r3 = r2 - (d*x-xc)^2 # (d*y-yc)^2 <= r2 - (d*x-xc)^2
....:         ymin = ceil((yc - sqrt(r3))/d)
....:         ymax = floor((yc + sqrt(r3))/d)
....:         s += ymax + 1 - ymin
....:     return s
```

```
sage: def rk(k): # returns (r_k^2, xc, yc)
....:     if k == 2: return 1/4, 1/2, 0
....:     dmax = (2*sqrt(k/pi)).n(); xamax = (sqrt(2*k/pi)).n()
....:     sol = (dmax/2)^2, 0, 0, 0
....:     for xa in range(0, floor(xamax)+1):
....:         # if xa=0, ya > 0 since A should differ from O
....:         yamin = max(xa, 1)
```

```
....:         for ya in range(yamin, floor(sqrt(dmax^2-xa^2))+1):
....:             xbmin = 0 # we want xb*ya <= xa^2+ya^2
....:             if xa == 0:
....:                 xbmin = 1 # O, A, B should not be aligned
....:             xbmax = min(floor(dmax), floor((xa*xa+ya*ya)/ya))
....:             for xb in range(xbmin, xbmax+1):
....:                 ybmax = floor(sqrt(dmax^2-xb^2))
....:                 if xa > 0: # we want xb*ya+yb*xa <= xa^2+ya^2
....:                     tmp = floor((xa*xa+ya*ya-xb*ya)/xa)
....:                     ybmax = min(ybmax, tmp)
....:                 # if xb=0, yb > 0 since B should differ from O
....:                 ybmin = 0
....:                 if xb == 0:
....:                     ybmin = 1
....:                 for yb in range(ybmin,ybmax+1):
....:                     d = 2*abs(xb*ya - xa*yb)
....:                     if d <> 0:
....:                         ra2 = xa^2+ya^2; rb2 = xb^2+yb^2
....:                         xc = abs(ra2*yb - rb2*ya)
....:                         yc = abs(rb2*xa - ra2*xb)
....:                         r2 = ra2*rb2*((xa-xb)^2+(ya-yb)^2)
....:                         m = rk_aux(xc,yc,d,r2)
....:                         if m >= k and r2/d^2 < sol[0]:
....:                             sol = r2/d^2, xc/d, yc/d
....:     return sol
```

```
sage: for k in range(2,10): print((k, rk(k)))
(2, (1/4, 1/2, 0))
(3, (1/2, 1/2, 1/2))
(4, (1/2, 1/2, 1/2))
(5, (1, 0, 1))
(6, (5/4, 1/2, 1))
(7, (25/16, 3/4, 1))
(8, (2, 1, 1))
(9, (2, 1, 1))
```

For question 2, a solution is the following:

```
sage: def plotrk(k):
....:     r2, x0, y0 = rk(k); r = n(sqrt(r2))
....:     var('x, y')
....:     c = implicit_plot((x-x0)^2+(y-y0)^2-r2,
....:         (x, x0-r-1/2, x0+r+1/2),(y, y0-r-1/2, y0+r+1/2))
....:     center = points([(x0,y0)], pointsize=50, color='black')
....:     # we want (i-x0)^2+(j-y0)^2 <= r2
....:     # thus |i-x0| <= r and |j-y0| <= r2 - (i-x0)^2
....:     l = [(i, j) for i in range(ceil(x0-r), floor(x0+r)+1)
....:                 for j in range(ceil(y0-sqrt(r^2-(i-x0)^2)),
....:                                floor(y0+sqrt(r2-(i-x0)^2))+1)]
....:     d = points(l, pointsize=100)
....:     return (c+center+d).show(aspect_ratio=1, axes=True)
```

Question 3 requires a little more work. Let us write $S_{i,j} = \sum_{k=i}^{j} 1/(\pi r_k^2)$. From the upper bound (6.2) for r_k, we obtain $r_k^2 < (k-1)/\pi$, thus $1/(\pi r_k^2) > 1/(k-1)$, and $S_{n,N} > \sum_{k=n}^{N} 1/(k-1) > \int_n^{N+1} \mathrm{d}k/k = \log((N+1)/n)$.

The lower bound (6.2) gives $1/(\pi r_k^2) < 1/k + 2/k^{3/2}$ for $k \geq 407$, which leads for $n \geq 407$ to $S_{n,N} < \sum_{k=n}^{N}(1/k + 2/k^{3/2}) < \int_{n-1}^{N}(1/k + 2/k^{3/2})\mathrm{d}k = \log(N/(n-1)) + 4/\sqrt{n-1} - 4/\sqrt{N}$, thus:

$$S_{2,n-1} + \log(1/n) \leq \delta \leq S_{2,n-1} + \log(1/(n-1)) + 4/\sqrt{n-1}.$$

```
sage: def bound(n):
....:     s = sum(1/pi/rk(k)[0] for k in range(2,n+1))
....:     return float(s+log(1/n)), float(s+log(1/(n-1))+4/sqrt(n-1))
sage: bound(60)
(1.7327473659779615, 2.2703101282176377)
```

We deduce $1.73 < \delta < 2.28$, thus the approximation $\delta \approx 2.00$ with an error bounded by 0.28.

Exercise 23 page 126. We use here the same notations as in Beauzamy's article. We write $s_i = 1 - x_i - \cdots - x_k$ with $s_{k+1} = 1$. We must thus have $x_1 + \cdots + x_{i-1} \leq s_i$, and in particular $x_2 \leq x_1 \leq s_2$. Let us define

$$C_1 = \int_{x_1=x_2}^{s_2} x_1^{n_1} \mathrm{d}x_1 = \frac{1}{n_1+1}(s_2^{n_1+1} - x_2^{n_1+1}).$$

```
sage: x1, x2, s2 = var('x1, x2, s2')
sage: n1 = 9; C1 = integrate(x1^n1, x1, x2, s2); C1
1/10*s2^10 - 1/10*x2^10
```

Then we have $x_3 \leq x_2 \leq s_3 = s_2 + x_2$, thus by replacing s_2 by $s_3 - x_2$ in C_1, and by integrating for x_2 from x_3 to $s_3/2$ — since $x_1 + x_2 \leq s_3$ and $x_2 \leq x_1$ — we get:

```
sage: x3, s3 = var('x3, s3')
sage: n2 = 7; C2 = integrate(C1.subs(s2=s3-x2)*x2^n2, x2, x3, s3/2); C2
44923/229417943040*s3^18 - 1/80*s3^10*x3^8 + 1/9*s3^9*x3^9 - 9/20*s3^8*x3^10
+ 12/11*s3^7*x3^11 - 7/4*s3^6*x3^12 + 126/65*s3^5*x3^13 - 3/2*s3^4*x3^14
+ 4/5*s3^3*x3^15 - 9/32*s3^2*x3^16 + 1/17*s3*x3^17
```

and so on. At each iteration C_i is an homogeneous polynomial in x_{i+1} and s_{i+1}, with rational coefficients and of total degree $n_1 + \ldots + n_i + i$. For the last variable, we integrate between $x_k = 0$ and $x_k = 1/k$.

By assuming known bounds on the numerator and denominator of I, we can compute I modulo p for different prime numbers not dividing the denominator of I, and deduce by the Chinese Remainder Theorem the value of I modulo the product of those prime numbers, and finally using rational reconstruction the exact value of I.

A.7 Polynomials

Exercise 24 page 129.

1. For example, but there exist many other solutions, we can take

```
sage: x = polygen(QQ, 'y'); y = polygen(QQ, 'x')
```

   Let us recall the difference, in Sage, between Python variables and mathematical variables. Python variables are names, used for programming; they only denote a position in the memory. The mathematical variables, and polynomials are part of them, are completely different by nature: they are Sage objects which *can be stored* in Python variables. If we create an indeterminate called `'x'`, we are not forced to store it in the Python variable `x` — and we can also store `'y'` there.

2. We first assign the indeterminate `'x'` of the polynomials with rational coefficients to the Python variable `x`. Then, the expression `x+1` evaluates as the polynomial $x + 1 \in \mathbb{Q}[x]$ which we assign to the variable `p`. After that we assign the integer 2 to the variable `x`. This operation has no effect on `p`, which keeps the value $x + 1$; this x is the indeterminate: it has nothing to do with the Python variable whose value is now 2. Now `p+x` evaluates as $x + 3$, and thus, the final value of `p` is $x + 3$.

Exercise 25 page 135. A simple solution is to carry out successive Euclidean divisions by the Chebyshev polynomials by increasing powers: if the polynomial degree is n, we set $p = c_n T_n + R_{n-1}$ with $c_n \in \mathbb{Q}$ and $\deg R_{n-1} \leq n - 1$, then $R_{n-1} = c_{n-1} T_{n-1}$ and so on.

In the Sage program below, instead of returning the computed coefficients c_n as a simple list, we prefer to build a symbolic expression where the polynomial T_n is represented as an "inert" function (that is to say kept in a non-evaluated form) `T(n,x)`.

```
sage: T = sage.symbolic.function_factory.function('T', nargs=2)
sage: def to_chebyshev_basis(pol):
....:     (x,) = pol.variables()
....:     res = 0
....:     for n in xrange(pol.degree(), -1, -1):
....:         quo, pol = pol.quo_rem(chebyshev_T(n, x))
....:         res += quo * T(n, x)
....:     return res
```

Let us test this function. To check the results, we can just substitute the function which computes the Chebyshev polynomials into our "inert" function `T`, and then expand the result:

```
sage: p = QQ['x'].random_element(degree=6); p
4*x^6 + 4*x^5 + 1/9*x^4 - 2*x^3 + 2/19*x^2 + 1
sage: p_cheb = to_chebyshev_basis(p); p_cheb
1/8*T(6, x) + 1/4*T(5, x) + 55/72*T(4, x) + 3/4*T(3, x) +
2713/1368*T(2, x) + T(1, x) + 1069/456*T(0, x)
```

```
sage: p_cheb.substitute_function(T, chebyshev_T).expand()
4*x^6 + 4*x^5 + 1/9*x^4 - 2*x^3 + 2/19*x^2 + 1
```

Exercise 26 page 135. A direct translation of the algorithm in Sage gives something like:

```
sage: def mydiv(u, v, n):
....:     v0 = v.constant_coefficient()
....:     quo = 0; rem = u
....:     for k in xrange(n+1):
....:         c = rem[0]/v0
....:         rem = (rem - c*v) >> 1  # shifting the coefficients
....:         quo += c*x^k
....:     return quo, rem
```

(One can apply this function to larger examples and measure the execution time and try to make the code more efficient, keeping the same algorithm.)

But the division by increasing powers up to order n is the series expansion of the rational function u/v truncated at order $n+1$. Using the division of formal power series (see §7.5), we can compute the division by increasing powers like in the following function.

```
sage: def mydiv2(u, v, n):
....:     x = u.parent().gen()
....:     quo = (u / (v + O(x^(n+1)))).polynomial()
....:     rem = (u - quo*v) >> (n+1)
....:     return quo, rem
```

The line `quo = ...` uses the fact that, if we add $O(\cdot)$ to a polynomial, it is converted into a truncated series and that the default precision used when dividing a polynomial by a series is that of the divisor.

Exercise 27 page 136. First of all, $u_{10^{10000}}$ has about 10^{10000} expected digits. Computing it entirely is absolutely out of reach. But as we are only interested by the last five digits, it is not a real problem: we will compute everything modulo 10^5. The fast exponentiation method as presented in §3.2.4 needs tens of thousands of matrix multiplications, the matrix size being 1000×1000, with coefficients in $\mathbb{Z}/10^5\mathbb{Z}$. Each of these matrix products costs about one billion multiplications, or less with a fast algorithm. It is not impossible, but a test with only one multiplication suggests that the full computation with Sage would take at least one hour:

```
sage: Mat = MatrixSpace (IntegerModRing(10^5), 1000)
sage: m1, m2 = (Mat.random_element() for i in (1,2))
sage: %time p = m1*m2
CPU times: user 48 ms, sys: 4 ms, total: 52 ms
Wall time: 54.3 ms
```

It is possible to do much better from the algorithmic point of view. Let us denote S the shift operator $(a_n)_{n\in\mathbb{N}} \mapsto (a_{n+1})_{n\in\mathbb{N}}$. The equation satisfied by $u =$

$(u_n)_{n\in\mathbb{N}}$ can be rewritten as $P(S)\cdot u = 0$, with $P(x) = x^{1000} - 23x^{729} + 5x^2 - 12x - 7$. For all N (especially $N = 10^{100}$), the u_N term is the first one in the sequence $S^N \cdot u$. Let R be the remainder of the Euclidean division of x^N by P. As $P(S) \cdot u = 0$, we have $S^N \cdot u = R(S) \cdot u$. Thus it is only necessary to compute the image of x^N in $(\mathbb{Z}/10^5\mathbb{Z})[x]/\langle P(x)\rangle$. We obtain the following code which, on the same machine, computes the result in less than twenty seconds:

```
sage: Poly.<x> = Integers(10^5)[]
sage: P = x^1000 - 23*x^729 + 5*x^2 - 12*x - 7
sage: Quo.<s> = Poly.quo(P)
sage: op = s^(10^10000)
sage: add(op[n]*(n+7) for n in range(1000))
63477
```

The last five digits are 63477. The difference in computing time between both methods grows rapidly with the order of the recurrence.

Exercise 28 page 147.

1. Let us assume that $a_s u_{n+s} + a_{s-1}u_{n+s-1} + \cdots + a_0 u_n = 0$ for all $n \geq 0$, and let us denote $u(z) = \sum_{n=0}^{\infty} u_n z^n$. Let $Q(z) = a_s + a_{s-1}z + \cdots + a_0 z^s$. Then

$$S(z) = Q(z)\,u(z) = \sum_{n=0}^{\infty} (a_s u_n + a_{s-1}u_{n-1} + \cdots + a_0 u_{n-s}) z^n,$$

   with the convention that $u_n = 0$ for $n < 0$. The coefficient of z^n in $S(z)$ is zero for $n \geq s$, thus $S(z)$ is a polynomial and $u(z) = S(z)/Q(z)$. The denominator $Q(z)$ is the reciprocal polynomial of the characteristic polynomial of the recurrence, and the numerator encodes the initial conditions.

2. The first coefficients are enough to guess an order-3 recurrence satisfied by the given coefficients. Using `rational_reconstruct` we obtain a rational function. By computing a series expansion of this rational function we recover all given coefficients and the possible next ones:

```
sage: p = previous_prime(2^30); ZpZx.<x> = Integers(p)[]
sage: s = ZpZx([1, 1, 2, 3, 8, 11, 34, 39, 148, 127, 662, 339])
sage: num, den = s.rational_reconstruct(x^12, 6, 6)
sage: S = ZpZx.completion(x)
sage: map(lift_sym, S(num)/S(den))
[1, 1, 2, 3, 8, 11, 34, 39, 148, 127, 662, 339, 3056, 371,
14602, -4257, 72268, -50489, 369854, -396981]
```

   (The `lift_sym` function is defined in the chapter of the book. All of the first 20 coefficients of the sequence are much lower than 2^{29}, so that we can allow us to unroll the recurrence modulo a prime near 2^{30}, then lift up the result in $\mathbb{Z}$, rather than the converse.)

   With `berlekamp_massey`, we directly obtain the characteristic polynomial of the recurrence with coefficients in $\mathbb{Z}$:

```
sage: berlekamp_massey([1, 1, 2, 3, 8, 11, 34, 39, 148, 127])
x^3 - 5*x + 2
```

Then we verify that all coefficients satisfy $u_{n+3} = 5u_{n+1} - 2u_n$, and from there we guess the missing coefficients $72268 = 5 \cdot 14602 - 2 \cdot 371$, $-50489 = 5 \cdot (-4257) - 2 \cdot 14602$ and so forth.

Exercise 29 page 147. We first construct a polynomial of degree 3 which satisfies the given interpolation conditions:

```
sage: R.<x> = GF(17)[]
sage: pairs = [(0,-1), (1,0), (2,7), (3,5)]
sage: s = R(QQ['x'].lagrange_polynomial(pairs)); s
6*x^3 + 2*x^2 + 10*x + 16
sage: [s(i) for i in range(4)]
[16, 0, 7, 5]
```

We reduce the exercise to a problem of rational reconstruction:

$$p/q \equiv s \mod x(x-1)(x-2)(x-3).$$

As s cannot be inverted modulo $x(x-1)(x-2)(x-3)$ (recall that $s(1) = 0$), there is no solution with a constant p. With $\deg p = 1$, we find:

```
sage: s.rational_reconstruct(mul(x-i for i in range(4)), 1, 2)
(15*x + 2, x^2 + 11*x + 15)
```

Exercise 30 page 150. We proceed as in the example: we rewrite the equation $\tan x = \int_0^x (1 + \tan^2 t)\,\mathrm{d}t$ and we search for a fixed point, starting from the initial condition $\tan(0) = 0$.

```
sage: S.<x> = PowerSeriesRing(QQ)
sage: t = S(0)
sage: for i in range(7): # here t is correct up to degree 2i+1
....:     # with O(x^15) we prevent the truncation order to grow
....:     t = (1+t^2).integral() + O(x^15)
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)
```

A.8 Linear Algebra

Exercise 31 page 171. (*Minimal polynomial of a vector*)

1. φ_A is an annihilating polynomial for all vectors e_i of the canonical basis. It is therefore a multiple of all φ_{A,e_i}. Let ψ be the least common multiple of the φ_{A,e_i}. It verifies $\psi|\varphi_A$. Moreover, $\psi(A) = \begin{bmatrix}\psi(A)e_1 & \dots & \psi(A)e_n\end{bmatrix} = 0$ is annihilating the matrix A. Hence $\varphi_A|\psi$. Since these polynomials are both monic, they are equal.

2. In this case all φ_{A,e_i} are of the form χ^{ℓ_i}, where χ is an irreducible polynomial. From the previous question, φ_A coincides with the polynomial χ^{ℓ_i} having the largest multiplicity ℓ_i.

3. Let φ be an annihilating polynomial for the vector $e = e_i + e_j$ and let $\varphi_1 = \varphi_{A,e_i}, \varphi_2 = \varphi_{A,e_j}$. We have $\varphi_2(A)\varphi(A)e_i = \varphi_2(A)\varphi(A)e - \varphi(A)\varphi_2(A)e_j = 0$. Hence $\varphi_2\varphi$ is annihilating the vector e_i and is therefore divisible by φ_1. Now since φ_1 and φ_2 are coprime, we have $\varphi_1|\varphi$. Similarly one shows that $\varphi_2|\varphi$, thus φ is a multiple of $\varphi_1\varphi_2$. Now $\varphi_1\varphi_2$ is annihilating e, thus $\varphi = \varphi_1\varphi_2$.

4. P_1 and P_2 being coprime, there exist two polynomials α and β such that $1 = \alpha P_1 + \beta P_2$. Thus for any vector x, we have $x = \alpha(A)P_1(A)x + \beta(A)P_2(A)x = x_2 + x_1$, where $x_1 = \beta(A)P_2(A)x$ and $x_2 = \alpha(A)P_1(A)x$. As $\varphi_A = P_1P_2$, P_1 is annihilating $x_1 = \beta(A)P_2(A)x$ (similarly P_2 is annihilating x_2). If for any vector x, $x_1 = 0$, then βP_2 is annihilating A and is therefore a multiple of P_1P_2, hence $1 = P_1(\alpha + \gamma P_2)$, which implies $\deg P_1 = 0$. Therefore, there exists a nonzero x_1 such that P_1 is an annihilating polynomial of x_1. We will now show that P_1 is minimal for x_1: let $\tilde{P}_1$ be an annihilating polynomial of x_1. Then $\tilde{P}_1(A)P_2(A)x = P_2(A)\tilde{P}_1(A)x_1 + \tilde{P}_1(A)P_2(A)x_2 = 0$, hence $\tilde{P}_1P_2$ is a multiple of $\varphi_A = P_1P_2$. Hence $P_1|\tilde{P}_1$, and P_1 is therefore the minimal polynomial of x_1. The reasoning is identical for x_2.

5. For each factor $\varphi_i^{m_i}$, there exists a vector x_i for which $\varphi_i^{m_i}$ is the minimal polynomial and the vector $x_1 + \cdots + x_k$ has minimal polynomial φ_A.

6. One first computes the minimal polynomial of the matrix A.

```
sage: A = matrix(GF(7),[[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: P = A.minpoly(); P
x^5 + 4*x^4 + 3*x^2 + 3*x + 1
sage: P.factor()
(x^2 + 2*x + 2) * (x^3 + 2*x^2 + x + 4)
```

It has maximal degree.

```
sage: e1 = identity_matrix(GF(7),5)[0]
sage: e4 = identity_matrix(GF(7),5)[3]
sage: A.transpose().maxspin(e1)
[(1, 0, 0, 0, 0), (0, 1, 0, 0, 0), (0, 0, 1, 0, 0)]
sage: A.transpose().maxspin(e4)
[(0, 0, 0, 1, 0), (0, 0, 0, 0, 1)]
sage: A.transpose().maxspin(e1 + e4)
[(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)]
```

The method `maxspin` iterates a vector *on the left*. We therefore apply it on the transpose of the matrix so as to produce the list of linearly independent Krylov iterates from the vectors e_1 and e_4. The minimal polynomial of e_1 thus has degree 3, that of e_4 has degree 2, and that of $e_1 + e_4$ has degree 5.

Note that the shape of the matrix implies that the vectors e_1 and e_4 produce iterates that are also vectors of the canonical basis. This form is also called the Frobenius normal form (see §8.2.3). It describes how the

matrix decomposes the space into invariant cyclic subspaces generated by vectors of the canonical basis.

Exercise 32 page 178. (*Test whether two matrices are similar*)

```
sage: def Similar(A, B):
....:     F1, U1 = A.frobenius(2)
....:     F2, U2 = B.frobenius(2)
....:     if F1 == F2:
....:         return True, ~U2*U1
....:     else:
....:         return False, F1 - F2
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]])
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]])
sage: A = (U^-1 * B * U).change_ring(ZZ)
sage: ok, V = Similar(A, B); ok
True
sage: V
[                1    2824643/1601680   -6818729/1601680
    -43439399/11211760  73108601/11211760]
[                0      342591/320336     -695773/320336
    -2360063/11211760  -10291875/2242352]
[                0     -367393/640672      673091/640672
    -888723/4484704   15889341/4484704]
[                0     661457/3203360    -565971/3203360
    13485411/22423520 -69159661/22423520]
[                0   -4846439/3203360    7915157/3203360
    -32420037/22423520 285914347/22423520]
sage: ok, V = Similar(2*A, B); ok
False
```

A.9 Polynomial Systems

Exercise 33 page 180. Given a polynomial ring, the `test_poly` function returns the sum of all monomials of total degree bounded by the value of the parameter `deg`. Its code is quite compact, and deserves some explanations.

The first instruction constructs and assigns to the local variable `monomials` a set (represented by a specific object `SubMultiset`, see §15.2) of lists, each one having `deg` elements, whose product corresponds to a term of the polynomial:

```
sage: ring = QQ['x,y,z']; deg = 2
sage: tmp1 = [(x,)*deg for x in (1,) + ring.gens()]; tmp1
[(1, 1), (x, x), (y, y), (z, z)]
sage: tmp2 = flatten(tmp1); tmp2
[1, 1, x, x, y, y, z, z]
```

```
sage: monomials = Subsets(tmp2, deg, submultiset=True); monomials
SubMultiset of [y, y, 1, 1, z, z, x, x] of size 2
sage: monomials.list()
[[y, y], [y, 1], [y, z], [y, x], [1, 1], [1, z], [1, x], [z, z], [z, x],
     [x, x]]
```

For this purpose, we start by adding 1 to the tuple of indeterminates, replace each element of the result by a tuple of `deg` copies of itself, and group these tuples in a list. Let us notice the syntax `(x,)` which denotes a tuple with only one element, as well as the operators `+` and `*` for the concatenation and repetition of tuples. The obtained list of tuples is transformed by the `flatten` command into a list containing exactly `deg` times each indeterminate and the constant 1. The `Subsets` function with the option `submultiset=True` then computes the subsets of cardinality `deg` of the multiset (set with repetitions) of elements from this list. The `monomials` object is iterable: thus `(mul(m) for m in monomials)` is a Python generator which iterates over the built monomials by passing to `mul` the lists representing the subsets. This generator is finally given to `add`.

The last line could be replaced by `add(map(mul, monomials))`. We might also write `((1,) + ring.gens())*deg` to simplify the expression `[(x,)*deg for x in (1,) + ring.gens()]`.

Exercise 34 page 180. An example of the help page `PolynomialRing?` suggests a solution: to obtain a non-trivial family of indeterminates — here, indexed by prime numbers — we give to `PolynomialRing` a list built by comprehension (see §3.3.2):

```
sage: ['x%d' % n for n in [2,3,5,7]]
['x2', 'x3', 'x5', 'x7']
sage: R = PolynomialRing(QQ, ['x%d' % n for n in primes(40)])
sage: R.inject_variables()
Defining x2, x3, x5, x7, x11, x13, x17, x19, x23, x29, x31, x37
```

The `inject_variables` method initialises the Python variables `x2`, `x3`, ..., each one containing the corresponding generator of `R`.

Exercise 35 page 186. We check that $(3, 2, 1)$ is the only real root, for example with

```
sage: R.<x,y,z> = QQ[]
sage: J = R.ideal(x^2*y*z-18, x*y^3*z-24, x*y*z^4-6)
sage: J.variety(AA)
[{x: 3, z: 1, y: 2}]
```

or with

```
sage: V = J.variety(QQbar)
sage: [u for u in V if all(a in AA for a in u.values())]
[{z: 1, y: 2, x: 3}]
```

A substitution $(x, y, z) \mapsto (\omega^a x, \omega^b y, \omega^c z)$ with $\omega^k = 1$ keeps the system invariant if and only if (a, b, c) is a root modulo k of the homogeneous linear system of matrix

```
sage: M = matrix([ [p.degree(v) for v in (x,y,z)]
....:              for p in J.gens()]); M
[2 1 1]
[1 3 1]
[1 1 4]
```

By computing its determinant

```
sage: M.det()
17
```

we see that $k = 17$ works. It just remains to find a non-zero kernel element:

```
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]
```

Exercise 36 page 198. It is almost trivial:

```
sage: L.<a> = QQ[sqrt(2-sqrt(3))]; L
Number Field in a with defining polynomial x^4 - 4*x^2 + 1
sage: R.<x,y> = QQ[]
sage: J1 = (x^2 + y^2 - 1, 16*x^2*y^2 - 1)*R
sage: J1.variety(L)
[{y: 1/2*a^3 - 2*a, x: -1/2*a}, {y: 1/2*a^3 - 2*a, x: 1/2*a},
 {y: -1/2*a, x: 1/2*a^3 - 2*a}, {y: -1/2*a, x: -1/2*a^3 + 2*a},
 {y: 1/2*a, x: 1/2*a^3 - 2*a}, {y: 1/2*a, x: -1/2*a^3 + 2*a},
 {y: -1/2*a^3 + 2*a, x: -1/2*a}, {y: -1/2*a^3 + 2*a, x: 1/2*a}]
```

Thus, for example, we have for the fifth solution above:

$$x = \frac{1}{2}\,(2-\sqrt{3})^{3/2} - 2\sqrt{2-\sqrt{3}}, \qquad y = \frac{1}{2}\sqrt{2-\sqrt{3}}\,.$$

Exercise 37 page 202. We have seen how to obtain a basis B of the $\mathbb{Q}$-vector space $\mathbb{Q}[x,y]/J_2$:

```
sage: R.<x,y> = QQ[]; J2 = (x^2+y^2-1, 4*x^2*y^2-1)*R
sage: basis = J2.normal_basis(); basis
[x*y^3, y^3, x*y^2, y^2, x*y, y, x, 1]
```

We then compute the image of B by m_x, and we deduce the matrix of m_x in the basis B:

```
sage: xbasis = [(x*p).reduce(J2) for p in basis]; xbasis
[1/4*y, x*y^3, 1/4, x*y^2, -y^3 + y, x*y, -y^2 + 1, x]
sage: mat = matrix([ [xp[q] for q in basis]
....:                for xp in xbasis])
sage: mat
```

```
[  0   0   0   0   0 1/4   0   0]
[  1   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0 1/4]
[  0   0   1   0   0   0   0   0]
[  0  -1   0   0   0   1   0   0]
[  0   0   0   0   1   0   0   0]
[  0   0   0  -1   0   0   0   1]
[  0   0   0   0   0   0   1   0]
```

The polynomial χ_x and its roots are then given by (see Chapters 2 and 8):

```
sage: charpoly = mat.characteristic_polynomial(); charpoly
x^8 - 2*x^6 + 3/2*x^4 - 1/2*x^2 + 1/16
sage: solve(SR(charpoly), SR(x))
[x == -1/2*sqrt(2), x == 1/2*sqrt(2)]
```

We see on this example that the roots of χ are the abscissas of the points of $V(J_2)$.

For a random ideal J, let us assume $\chi(\lambda) = 0$ with $\lambda \in \mathbb{C}$. Thus λ is an eigenvalue of m_x. Let $p \in \mathbb{Q}[x, y] \setminus J$ a representative of an eigenvector associated to λ: we have $xp = \lambda p + q$ for a given $q \in J$. Since $p \notin J$, we can find $(x_0, y_0) \in V(J)$ such that $p(x_0, y_0) \neq 0$, and we have then

$$(x_0 - \lambda)\, p(x_0, y_0) = q(x_0, y_0) = 0,$$

thus $\lambda = x_0$.

Exercise 38 page 211. The expressions $\sin\theta$, $\cos\theta$, $\sin(2\theta)$ and $\cos(2\theta)$ are related by the classical trigonometric formulas

$$\sin^2\theta + \cos^2\theta = 1, \quad \sin(2\theta) = 2(\sin\theta)(\cos\theta), \quad \cos(2\theta) = \cos^2\theta - \sin^2\theta.$$

To simplify the notations, let us write $c = \cos\theta$ and $s = \sin\theta$. The ideal

$$\langle u - (s + c), v - (2sc + c^2 - s^2), s^2 + c^2 - 1\rangle$$

from $\mathbb{Q}[s, c, u, v]$ translates the definitions of $u(\theta)$ and $v(\theta)$ from the exercise, together with the relation between sine and cosine. For a monomial order which eliminates s and c first, the canonical form of s^6 modulo this ideal gives the wanted result.

```
sage: R.<s, c, u, v> = PolynomialRing(QQ, order='lex')
sage: Rel = ideal(u-(s+c), v-(2*s*c+c^2-s^2), s^2+c^2-1)
sage: Rel.reduce(s^6)
1/16*u^2*v^2 - 3/8*u^2*v + 7/16*u^2 + 1/8*v^2 - 1/8*v - 1/8
```

A.10 Differential Equations and Recurrences

Exercise 39 page 221. (*Separable variable equations*)

1. Let us use the same method as in Section 10.1.2:

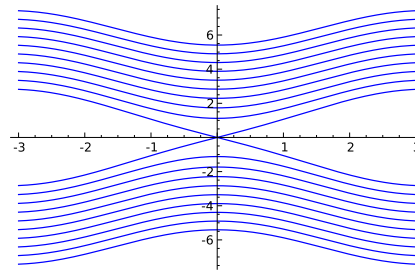
```
sage: x = var('x')
```

```
sage: y = function('y')(x)
sage: ed = (desolve(y*diff(y,x)/sqrt(1+y^2) == sin(x),y)); ed
sqrt(y(x)^2 + 1) == _C - cos(x)
```

The same problem appears. We impose that `_C-cos(x)` is positive:

```
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)]
```

```
sage: P = Graphics()
sage: for j in [0,1]:
....:   for k in range(0,20,2):
....:     P += plot(sol[j].substitute(c==2+0.25*k).rhs(),x,-3,3)
sage: P
```



2. Same method:

```
sage: sol = desolve(diff(y,x)==sin(x)/cos(y), y, show_method=True)
sage: sol
[sin(y(x)) == _C - cos(x), 'separable']
sage: solve(sol[0],y)
[y(x) == -arcsin(-_C + cos(x))]
```

Exercise 40 page 222. (*Homogeneous equations*) We verify that the equation $xyy' = x^2 + y^2$ defined on $]0, +\infty[$ and on $]-\infty, 0[$ is homogeneous, then we try to solve it by the change of unknown function indicated in the example treated in Section 10.1.2.

```
sage: x = var('x')
sage: y = function('y')(x)
sage: id(x) = x
sage: u = function('u')(x)
sage: d = diff(u*id,x)
sage: DE = (x*y*d == x**2+y**2).substitute(y == u*id)
sage: eq = desolve(DE,u)
```

```
sage: sol = solve(eq,u)
sage: sol
[u(x) == -sqrt(2*_C + 2*log(x)), u(x) == sqrt(2*_C + 2*log(x))]
sage: Y = [x*sol[0].rhs() , x*sol[1].rhs()]
sage: Y[0]
-sqrt(2*_C + 2*log(x))*x
```

We can add conditions on x (with `assume`) to remember that the equation is not defined in 0.

A.11 Floating-Point Numbers

Exercise 41 page 239. We propose two solutions.

1. Let us make the calculation without the methods of the class `RealField` which give the significand and the exponent of a number. We first check that $2^{99} < 10^{30} < 2^{100}$ (we remark that $10^{30} = (10^3)^{10} \approx (2^{10})^{10}$).

```
sage: R100=RealField(100)
sage: x=R100(10^30)
sage: x>2^99
True
sage: x<2^100
True
```

Then, we compute the significand:

```
sage: e=2^100
sage: s1=10^30
sage: significand=[]
sage: nbdigits=0 # number of significant digits
sage: while s1>0:
....:     e/=2
....:     if e<=s1:
....:         significand.append(1)
....:         s1-=e
....:     else:
....:         significand.append(0)
....:     nbdigits+=1
sage: print(significand)
[1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0,
 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0,
 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1,
 1, 1, 1, 0, 1, 0, 1, 0, 0, 1]
sage: print("number of significant digits: " + str(nbdigits))
number of significant digits: 70
```

All the binary digits of the significand beyond the seventieth are zero. Thus we add 2^{-100} to the significand to obtain the nearest number from 10^{30}, and we get the result: the value of `x.ulp()` is $2^{-100} \cdot 2^{100} = 1$.

2. Using the method `sign_mantissa_exponent()` of the class `RealField`, we obtain directly:

```
sage: R100=RealField(100)
sage: x=R100(10^30)
sage: s,m,e = x.sign_mantissa_exponent()
sage: s,m,e
(1, 1000000000000000000000000000000, 0)
```

The command `m.binary()` reveals that we get the same significand in both cases.

Exercise 42 page 242.

1. Let us calculate the values of α, β and γ in the formula:

$$u_n = \frac{\alpha\ 100^{n+1} + \beta\ 6^{n+1} + \gamma\ 5^{n+1}}{\alpha\ 100^n + \beta\ 6^n + \gamma\ 5^n}. \tag{A.1}$$

Why not use Sage for this? We use the values of u_0, u_1 and u_2 to obtain a system of equations where the unknowns are α, β and γ and then, we solve it. Let us define the general solution:

```
sage: var("u0 u1 u2 alpha beta gamma n")
(u0, u1, u2, alpha, beta, gamma, n)
sage: recurrence = lambda a,b:  111-1130/a+3000/(a*b)
sage: gener1 = lambda n: (alpha*100^n+beta*6^n+gamma*5^n)
sage: solGen =  lambda n: gener1(n+1)/gener1(n)
```

We calculate u_2 as a function of u_1 and u_0 to get the system:

```
sage: u2 = recurrence(u1,u0)
sage: s = [u2==solGen(2),u1==solGen(1),u0==solGen(0)]
sage: t = [s[i].substitute(u0=2,u1=-4) for i in range(0,3)]
```

then, we solve it:

```
sage: solve(t,alpha,beta,gamma)
[[alpha == 0, beta == -3/4*r1, gamma == r1]]
```

This shows that γ can have any value.

We must verify that we really got the general solution, that is to say that equation (A.1) is verified for all n:

```
sage: alpha=0
sage: beta = -3/4*gamma
sage: final=solGen(n)-recurrence(solGen(n-1),solGen(n-2))
sage: final.simplify_full()
0
```

Since we can take any value for γ, let us take $\gamma = 4$ and then we have $\beta = -3$ and $\alpha = 0$.

2. Now, we define a procedure which implements the recurrence, using exact coefficients so that we can reuse it with numbers of different precisions:

```
sage: def recur(x1,x0):
....:     return 111 - 1130/x1 + 3000/(x0*x1)
```

Let us take the initial conditions in `RealField()`, so that the calculation takes place in this domain:

```
sage: u0 = 2.
sage: u1 = -4.
sage: for i in range(1,25):
....:     x = recur(u1,u0)
....:     print((i, x))
....:     u0 = u1
....:     u1 = x
(1, 18.5000000000000)
(2, 9.37837837837838)
(3, 7.80115273775217)
(4, 7.15441448097533)
(5, 6.80678473692481)
(6, 6.59263276872179)
..................
(23, 99.9999986592167)
(24, 99.9999999193218)
```

Clearly, the sequence converges to 100!

3. The explanation of this behaviour is just that the terms u_{n-1} and u_{n-2} are computed with a rounding error, and thus, the formula (A.1) no longer defines the general solution of the recurrence.

Let us search for the stationary values of the recurrence:

```
sage: var("x")
x
sage: solve(x==recurrence(x,x),x)
[x == 100, x == 5, x == 6]
```

We observe that there are 3 stationary values: 100, 5 and 6. The convergence to 100, as observed in the presence of rounding errors can be explained by stability considerations for these 3 values (but this is out of the scope of this exercise).

4. Increasing the precision does not change the limit; the sequence always converges to 100:

```
sage: RL = RealField(5000)
sage: u0 = RL(2)
sage: u1 = RL(-4)
```

```
sage: for i in range(1,2500):
....:     x = recur(u1,u0)
....:     u0 = u1
....:     u1= x
sage: x
100.000000000000000000000000000000000000000000000000000000...
```

If only one of the u_i is not computed exactly, then the sequence diverges (α is not equal to zero).

5. With very few modifications to the program, we can initialise `u0` and `u1` as integers (and do all computations in $\mathbb{Q}$):

```
sage: u0 = 2
sage: u1 = -4
sage: for i in range(1,2500):
....:     x = recur(u1,u0)
....:     u0 = u1
....:     u1 = x
sage: float(x)
6.0
```

We find the expected value 6.0, but if we print `x`, we see the huge quantity of information used for the calculation (printing not reproduced here!). If we print `x-6`, we verify that the limit of the sequence is not attained: there is no reason for the memory size of `x` to decrease if we continue the iterations.

Exercise 43 page 251. Let I be an interval. If $f(x) = 1 - x^2$, we use the function $x \mapsto x^2$ extended to the interval I (see page 250). But when we compute the extension of $g(x) = 1 - x \cdot x$ to I we compute the interval $I_2 = \{y \cdot z, y \in I, z \in I\}$. Final results can differ:

```
sage: f = lambda x: x^2
sage: g = lambda x: x*x
sage: sage.rings.real_mpfi.printing_style = 'brackets'
sage: I = RIF(-1,1)
sage: f(I)
[0.00000000000000000 .. 1.0000000000000000]
sage: g(I)
[-1.0000000000000000 .. 1.0000000000000000]
```

A.12 Non-Linear Equations

Exercise 44 page 269. We have seen that the keyword `return` terminates the execution of the function. It is sufficient to test whether `f(u)` is zero. To avoid evaluating `f(u)` twice we store its value in a variable. We obtain the following function:

```
sage: def intervalgen(f, phi, s, t):
....:     assert (f(s) * f(t) < 0), \
....:            'Wrong arguments: f(%s) * f(%s) >= 0)'%(s, t)
....:     yield s
....:     yield t
....:     while 1:
....:         u = phi(s, t)
....:         yield u
....:         fu = f(u)
....:         if fu == 0:
....:             return
....:         if fu * f(s) < 0:
....:             t = u
....:         else:
....:             s = u
```

Let us test this function with an equation having a known solution, for example an affine function:

```
sage: f(x) = 4 * x - 1
sage: a, b = 0, 1
sage: phi(s, t) = (s + t) / 2
sage: list(intervalgen(f, phi, a, b))
[0, 1, 1/2, 1/4]
```

Exercise 45 page 269. The function `phi` passed as parameter to `intervalgen` establishes the point where we must divide the interval. It is sufficient to define the function:

```
sage: f(x) = 4 * sin(x) - exp(x) / 2 + 1
sage: a, b = RR(-pi), RR(pi)
sage: def phi(s, t): return RR.random_element(s, t)
sage: random = intervalgen(f, phi, a, b)
sage: iterate(random, maxit=10000)
After 19 iterations: 2.15848379485564
```

Exercise 46 page 278. It is natural to try to compute in `PolynomialRing(SR, 'x')`, which has a `roots()` method:

```
sage: basering.<x> = PolynomialRing(SR, 'x')
sage: p = x^2 + x
sage: p.roots(multiplicities=False)
[-1, 0]
```

We therefore obtain the following code.

```
sage: from collections import deque
sage: basering = PolynomialRing(SR, 'x')
sage: q, method = None, None
```

```
sage: def quadraticgen(f, r, s):
....:     global q, method
....:     t = r - f(r) / f.derivative()(r)
....:     method = 'newton'
....:     yield t
....:     pts = deque([(p, f(p)) for p in (r, s, t)], maxlen=3)
....:     while True:
....:         q = basering.lagrange_polynomial(pts)
....:         roots = [r for r in q.roots(multiplicities=False) \
....:                           if CC(r).is_real()]
....:         approx = None
....:         for root in roots:
....:             if (root - pts[2][0]) * (root - pts[1][0]) < 0:
....:                 approx = root
....:                 break
....:             elif (root - pts[0][0]) * (root - pts[1][0]) < 0:
....:                 pts.pop()
....:                 approx = root
....:                 break
....:         if approx:
....:             method = 'quadratic'
....:         else:
....:             method = 'dichotomy'
....:             approx = (pts[1][0] + pts[2][0]) / 2
....:         pts.append((approx, f(approx)))
....:         yield pts[2][0]
```

Now, it is possible to print the first terms of the sequence defined by Brent's method. But, these computations need a relatively long computing time (and produce an output too large for a single page of this book).

```
sage: basering = PolynomialRing(SR, 'x')
sage: a, b = pi/2, pi
sage: f(x) = 4 * sin(x) - exp(x) / 2 + 1
sage: generator = quadraticgen(f, a, b)
sage: generator.next()
1/2*pi - (e^(1/2*pi) - 10)*e^(-1/2*pi)
```

Running the code above, a patient reader can visualise the arcs of parabola used in the first terms of the sequence.

```
sage: generator = quadraticgen(f, a, b)
sage: g = plot(f, a, b, rgbcolor='blue')
sage: g += point((a, 0), rgbcolor='red', legend_label='0')
sage: g += point((b, 0), rgbcolor='red', legend_label='1')
sage: data = {'2': 'blue', '3': 'violet', '4': 'green'}
sage: for l, color in data.iteritems():
....:     u = RR(generator.next())
....:     print(u, method)
```

```
....:     g += point((u, 0), rgbcolor=color, legend_label=l)
....:     if method == 'quadratic':
....:         q = sum([c*x^d for d, c in enumerate(q.list())])
....:         g += plot(q, 0, 5, rgbcolor=color)
2.64959209030252 newton
2.17792417785922 quadratic
2.15915701206506 quadratic
sage: g.show()
```

A.13 Numerical Linear Algebra

Exercise 47 page 286. From the Sherman–Morrison formula , the solution of $Bx = f$ is equivalent to the solution of $Ax = \sigma(I + uv^tA^{-1})f$ with $\sigma = (1 + v^tA^{-1}u)^{-1}$. Then we proceed in the following way.

1. We compute w solution of $Aw = u$, then $\sigma = (1 + v^tw)^{-1}$.

2. We compute z solution of $Az = f$, then $g = v^tz$ (which is a scalar).

3. Then, we calculate $h = \sigma(f - gu)$ and we solve $Ax = h$; x is actually the solution of $Bx = f$.

We remark that we have solved 3 linear systems with the matrix A, which is factorised. All together we have solved 6 linear systems, each of them being triangular. The cost of each of these resolutions is about n^2 operations, much less than the cost of a factorisation, which is about n^3 operations. To verify the Sherman–Morrison formula it is sufficient to right multiply the right-hand side of the formula by $A + uv^t$ and then to verify that the result is equal to the identity matrix.

Exercise 48 page 290. We consider the Cholesky factorisation $A = CC^t$, and the singular value decomposition of C: $C = U\Sigma V^t$. Then, $X = U\Sigma U^t$. Indeed: $A = CC^t = (U\Sigma V^t)(V\Sigma U^t) = U\Sigma U^tU\Sigma U^t = X^2$.

Let us construct a random symmetric positive definite matrix:

```
sage: m = random_matrix(RDF,4)
sage: a = transpose(m)*m
sage: c = a.cholesky()
sage: U,S,V = c.SVD()
sage: X = U*S*transpose(U)
```

Now, we verify that $X^2 - a$ is zero (modulo rounding errors):

```
sage: M = (X*X-a)
sage: all(abs(M[i,j]) < 10^-14
....:     for i in range(4) for j in range(4) )
True
```

A.14 Numerical Integration

Exercise 49 page 316. (*Computation of Newton-Cotes coefficients*)

1. We remark that the degree of P_i is $n-1$ (and thus, formula (14.1) can be applied) and that $P_i(j) = 0$ for $j \in \{0, \dots, n-1\}$ and $j \neq i$, we deduce that

$$\int_0^{n-1} P_i(x)\,\mathrm{d}x = w_i P_i(i)$$

that is

$$w_i = \frac{\int_0^{n-1} P_i(x)\,\mathrm{d}x}{P_i(i)}.$$

2. Then it is easy to deduce how to compute the weights:

```
sage: x = var('x')
sage: def NCRule(n):
....:     P = prod([x - j for j in xrange(n)])
....:     return [integrate(P / (x-i), x, 0, n-1) \
....:             / (P/(x-i)).subs(x=i) for i in xrange(n)]
```

3. With a simple change of variable, we get:

$$\int_a^b f(x)\,\mathrm{d}x = \frac{b-a}{n-1}\int_0^{n-1} f\left(a + \frac{b-a}{n-1}u\right)\mathrm{d}u.$$

4. Applying the preceding formula, we find the following program:

```
sage: def QuadNC(f, a, b, n):
....:     W = NCRule(n)
....:     ret = 0
....:     for i in xrange(n):
....:         ret += f(a + (b-a)/(n-1)*i) * W[i]
....:     return (b-a)/(n-1)*ret
```

Before comparing this method with others from the precision point of view, we can verify that it does not return inconsistent results:

```
sage: QuadNC(lambda u: 1, 0, 1, 12)
1
sage: N(QuadNC(sin, 0, pi, 10))
1.99999989482634
```

Let us compare our method with GSL on integrals I_2 and I_3:

```
sage: numerical_integral(x * log(1+x), 0, 1)
(0.25, 2.7755575615628914e-15)
sage: N(QuadNC(lambda x: x * log(1+x), 0, 1, 19))
0.250000000000001
sage: numerical_integral(sqrt(1-x^2), 0, 1)
```

```
(0.785398167726482..., 9.042725224567119...e-07)
sage: N(pi/4)
0.785398163397448
sage: N(QuadNC(lambda x: sqrt(1-x^2), 0, 1, 20))
0.784586419900198
```

We remark that the precision of the result depends on the amount of used points:

```
sage: [N(QuadNC(lambda x: x * log(1+x), 0, 1, n) - 1/4)
....:  for n in [2, 8, 16]]
[0.0965735902799726, 1.17408932933522e-7, 2.13449050101566e-13]
sage: [N(QuadNC(lambda x: sqrt(1-x^2), 0, 1, n) - pi/4)
....:  for n in [2, 8, 16]]
[-0.285398163397448, -0.00524656673640445, -0.00125482109302663]
```

A more interesting comparison between the different numerical integration methods in Sage and our method `QuadNC` would need to turn it into an adaptive method. Adaptive methods automatically subdivide the integration interval like what is done by `numerical_integral`.

A.15 Enumeration and Combinatorics

Exercise 50 page 330. (*Probability to draw a four-of-a-kind*) Let us build the set of fours-of-a-kind:

```
sage: Suits = FiniteEnumeratedSet(
....:            ["Hearts", "Diamonds", "Spades", "Clubs"])
sage: Values  = FiniteEnumeratedSet([2, 3, 4, 5, 6, 7, 8, 9, 10,
....:                                "Jack", "Queen", "King", "Ace"])
sage: FourOfaKind = cartesian_product([Arrangements(Values,2), Suits])
```

We have used `FiniteEnumeratedSet` instead of `Set` in order to specify the order of suits and of values, and thus of fours-of-a-kind:

```
sage: FourOfaKind.list()
[([2, 3], 'Hearts'),
 ([2, 3], 'Diamonds'),
...
 (['Ace', 'King'], 'Clubs')]
```

The above list starts with a four-of-a-kind of 2 with a 3 of hearts, and ends with a four-of-a-kind of aces with a king of clubs. There are 624 fours-of-a-kind in total:

```
sage: FourOfaKind.cardinality()
624
```

Relatively to the number of hands, we obtain a probability of 1 over 4165 to get a four-of-a-kind when we draw a hand at random:

```
sage: Cards = cartesian_product([Values, Suits])
sage: Hands = Subsets(Cards, 5)
sage: FourOfaKind.cardinality() / Hands.cardinality()
1/4165
```

Exercise 51 page 330. (*Probability to draw a flush and a straight flush*) To choose a straight flush, we have to choose its smallest card (between 1 and 10) and its suit. There are thus 40 straight flushes.

```
sage: StraightFlush = cartesian_product([range(1, 11), Suits])
sage: StraightFlush.cardinality()
40
```

It thus remains 5108 (non-straight) flushes:

```
sage: AllFlush = cartesian_product([Subsets(Values,5),Suits])
sage: AllFlush.cardinality() - StraightFlush.cardinality()
5108
```

Finally the probability to get a flush by drawing a hand at random is about 2 over 1000:

```
sage: _ / Hands.cardinality()
1277/649740
sage: float(_)
0.001965401545233478
```

It would be nicer to perform the above computation on sets — instead of dealing with cardinalities —, by explicitly building the set of flushes as the difference between `AllFlush` and `StraightFlush`. Yet there is no efficient generic algorithm to compute the difference $A \setminus B$ of two sets: without any additional information, the best we can do is to scan all elements of A, and check if they are in B. In the above computation, we have used the fact that B is included in A, which Sage cannot guess *a priori*. Another difficulty, however easier to handle, is that elements of A and B should be represented in the same way.

Exercise 52 page 330. We will here only deal with the case of the full hand, made of a three-of-a-kind and a pair. Let us first write a function checking for a full hand. For a shorter writing, we use the following method allowing us to count the repetitions of letters in a word:

```
sage: Word(['a','b','b','a','a','b','a']).evaluation_dict()
{'a': 4, 'b': 3}
```

```
sage: def is_full_hand(hand):
....:     suits = Word([value for (value, suit) in hand])
....:     repetitions = sorted(suits.evaluation_dict().values())
....:     return repetitions == [2,3]
sage: is_full_hand({(5, 'Diamonds'), (6, 'Diamonds'), (6, 'Hearts'),
....:               (5, 'Spades'), (1, 'Spades')})
False
```

```
sage: is_full_hand({(3, 'Clubs'), (3, 'Spades'), (3, 'Hearts'),
....:               (2, 'Clubs'), (2, 'Spades')})
True
```

We now estimate experimentally the proportion of full hands. More generally, the following function estimates the proportion of elements of the finite set `S` satisfying `predicate`. It assumes that the set has a `random_element` method with uniform distribution.

```
sage: def estimate_proportion(S, predicate, n):
....:     count = 0
....:     for i in range(n):
....:         if predicate(S.random_element()):
....:             count += 1
....:     return count/n
```

```
sage: float(estimate_proportion(Hands, is_full_hand, 10000))
0.0014
```

Let us now perform the computation symbolically. To identify a full hand, we have to choose a pair of distinct values, one for the three-of-a-kind, one for the pair, and a set of three suits for the three-of-a-kind, and two suits for the pair:

```
sage: FullHands = cartesian_product([Arrangements(Values, 2),
....:     Subsets(Suits, 3), Subsets(Suits, 2)])
```

Here is, for example, a full hand with a three-of-a-kind of twos, and a pair of threes:

```
sage: FullHands.first()
([2, 3], {'Hearts', 'Spades', 'Diamonds'}, {'Hearts', 'Diamonds'})
```

The probability to draw a full hand is:

```
sage: float(FullHands.cardinality() / Hands.cardinality())
0.0014405762304921968
```

Exercise 53 page 330. *(Counting by hand complete binary trees)* There is one complete binary tree with one leaf, and one with two leaves. For $n = 3, 4$ and 5 leaves, we find respectively 2, 5 and 14 trees (for $n = 4$, see Figure 15.1).

Exercise 54 page 340. The compositions of n with k parts have a one-to-one correspondence with subsets of size $k-1$ of $\{1, \ldots, n-1\}$: to the set $\{i_1, i_2, \ldots, i_{k-1}\}$ where $0 < i_1 < i_2 < \cdots < i_{k-1} < n$, we associate the composition $(i_1, i_2 - i_1, \ldots, n - i_{k-1})$, and reciprocally. We deduce the enumeration formulas: there are 2^{n-1} compositions of n, and among these $\binom{n-1}{k-1}$ compositions with k parts.

```
sage: n=6
sage: Compositions(n).cardinality(); 2^(n-1)
32
32
sage: n=6; k=3
```

```
sage: Compositions(n, length=k).cardinality(); binomial(n-1, k-1)
10
10
```

To find back if these formulas are used, we can look at the code of the `cardinality` command:

```
sage: C = Compositions(n)
sage: C.cardinality??
```

In the second case below, the name of the method used internally, namely `_cardinality_from_iterator`, yields the answer: the cardinality is computed — inefficiently — by iterating over all compositions.

```
sage: C = Compositions(5,length=3)
sage: C.cardinality
<bound method IntegerListsLex...._cardinality_from_iterator ...>
```

Exercise 55 page 343. Some examples:

```
sage: IntegerVectors(5,3).list()
[[5, 0, 0], [4, 1, 0], [4, 0, 1], [3, 2, 0], [3, 1, 1], [3, 0, 2],
 ...
 [0, 4, 1], [0, 3, 2], [0, 2, 3], [0, 1, 4], [0, 0, 5]]
```

```
sage: OrderedSetPartitions(3).cardinality()
13
sage: OrderedSetPartitions(3).list()
[[{1}, {2}, {3}], [{1}, {3}, {2}], [{2}, {1}, {3}], [{3}, {1}, {2}],
 ...
 [{1, 2}, {3}], [{1, 3}, {2}], [{2, 3}, {1}], [{1, 2, 3}]]
sage: OrderedSetPartitions(3,2).random_element()
[{1, 3}, {2}]
```

```
sage: StandardTableaux([3,2]).cardinality()
5
sage: StandardTableaux([3,2]).an_element()
[[1, 3, 5], [2, 4]]
```

Exercise 56 page 343. For small sizes, we obtain the permutation matrices:

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

The first negative sign appears for $n = 3$:

```
sage: list(AlternatingSignMatrices(3))
```

```
[
[1 0 0]  [0 1 0]  [1 0 0]  [ 0  1  0]  [0 0 1]  [0 1 0]  [0 0 1]
[0 1 0]  [1 0 0]  [0 0 1]  [ 1 -1  1]  [1 0 0]  [0 0 1]  [0 1 0]
[0 0 1], [0 0 1], [0 1 0], [ 0  1  0], [0 1 0], [1 0 0], [1 0 0]
]
```

By looking at examples for a larger n, we can see that it consists of all matrices with coefficients in $\{-1, 0, 1\}$ such that, on each row and column, the non-zero coefficients alternate between 1 and -1, starting and ending with 1.

Exercise 57 page 343. There are 2^5 vectors in $(\mathbb{Z}/2\mathbb{Z})^5$:

```
sage: GF(2)^5
Vector space of dimension 5 over Finite Field of size 2
sage: _.cardinality()
32
```

To build an invertible 3×3 matrix with coefficients in $\mathbb{Z}/2\mathbb{Z}$, it suffices to choose a first non-zero row vector ($2^3 - 1$ choices), then a second vector independent from the first one ($2^3 - 2$ choices), then a third one independent from the first two ($2^3 - 2^2$ choices). This gives:

```
sage: (2^3-2^0)*(2^3-2^1)*(2^3-2^2)
168
```

And indeed:

```
sage: GL(3,2)
General Linear Group of degree 3 over Finite Field of size 2
sage: _.cardinality()
168
```

The same reasoning yields the general formula, which is naturally expressed in terms of the q-factorial:

$$\prod_{k=0}^{n-1}(q^n - q^k) = q^{n(n-1)/2}(q-1)^n[n]_q!$$

Thus:

```
sage: from sage.combinat.q_analogues import q_factorial
sage: q = 2; n = 3
sage: q^(n*(n-1)/2) * (q-1)^n * q_factorial(n,q)
168
sage: q = 3; n = 5
sage: GL(n, q).cardinality()
475566474240
sage: q^(n*(n-1)/2) * (q-1)^n * q_factorial(n,q)
475566474240
```

Exercise 58 page 346. In the first case, Python first builds the list of all results before giving it to the `all` function. In the second case, the iterator

gives results incrementally to `all`, which can thus stop the iteration as soon as a counter-example is found.

Exercise 59 page 346. The first line gives the list of the cubes of all integers from -999 to 999. The next two lines search for a pair of cubes whose sum is 218. The last one is faster since it stops as soon as a solution is found.

```
sage: cubes = [t**3 for t in range(-999,1000)]
sage: %time exists([(x,y) for x in cubes for y in cubes],
....:              lambda (x,y): x+y == 218)
CPU times: user 940 ms, sys: 104 ms, total: 1.04 s
Wall time: 1.06 s
(True, (-125, 343))
sage: %time exists(((x,y) for x in cubes for y in cubes),
....:              lambda (x,y): x+y == 218)
CPU times: user 524 ms, sys: 4 ms, total: 528 ms
Wall time: 532 ms
(True, (-125, 343))
```

Moreover, it is more efficient in memory: if n is the length of the list of cubes, the memory used is of order n instead of n^2. This will be visible if one multiplies n by ten.

Exercise 60 page 346.

- Compute the generating function $\sum_{s \subset S} x^{|s|}$ of the subsets of $\{1, \dots, 8\}$ according to their cardinality.
- Compute the generating function of permutations of $\{1, 2, 3\}$ according to their number of inversions.
- Checks the tautology $\forall x \in P, x \in P$ for P being the set of permutations of $\{1, 2, 3, 4, 5\}$. This is a very good test for internal coherence between the iteration and membership functions of a given set. For a matter of fact, it is included in Sage generic tests; see:

  ```
  sage: P = Partitions(5)
  sage: P._test_enumerated_set_contains??
  ```

  The tautology $\forall x \notin P, x \notin P$ would be most useful to complete the membership test. However, we would need to specify the considered universe; and moreover, we would need an iterator on the complement of P in this universe, which is not a usual operation.
- Print all 2×2 invertible matrices over $\mathbb{Z}/2\mathbb{Z}$.
- Print all integer partitions of 3.
- Print all integer partitions (does not terminate!).
- Print all prime numbers (does not terminate!).
- Search for a prime number such that the associated Mersenne number is not prime.

- Iterates over all prime numbers whose associated Mersenne number is not prime.

Exercise 61 page 349. Let us define recursively our iterator:

```
sage: def C(n):
....:     if n == 1:
....:         yield BinaryTree()
....:     elif n > 1:
....:         for k in range(1,n):
....:             for t1 in C(k):
....:                 for t2 in C(n-k):
....:                     yield BinaryTree([t1,t2])
```

Here are the small trees:

```
sage: list(C(1))
[.]
sage: list(C(2))
[[., .]]
sage: list(C(3))
[[., [., .]],
 [[., .], .]]
sage: list(C(4))
[[., [., [., .]]],
 [., [[., .], .]],
 [[., .], [., .]],
 [[., [., .]], .],
 [[[., .], .], .]]
```

We indeed find Catalan's sequence:

```
sage: [len(list(C(n))) for n in range(9)]
[0, 1, 1, 2, 5, 14, 42, 132, 429]
```

A.16 Graph Theory

Exercise 62 page 365. (*Circulant graphs*) Two loops should suffice!

```
sage: def circulant(n,d):
....:     g = Graph(n)
....:     for u in range(n):
....:         for c in range(d):
....:             g.add_edge(u,(u+c)%n)
....:     return g
```

Exercise 63 page 367. (*Kneser Graphs*) The simplest method uses Sage's `Subsets` object. We then enumerate all pairs of vertices to find the adjacencies, albeit with many redundant calculations.

```
sage: def kneser(n,k):
....:     g = Graph()
....:     g.add_vertices(Subsets(n,k))
....:     for u in g:
....:         for v in g:
....:             if not u & v:
....:                 g.add_edge(u,v)
....:     return g
```

However, it is possible to save time by directly enumerating the adjacent vertices.

```
sage: def kneser(n,k):
....:     g = Graph()
....:     sommets = Set(range(n))
....:     g.add_vertices(Subsets(sommets,k))
....:     for u in g:
....:         for v in Subsets(sommets - u,k):
....:             g.add_edge(u,v)
....:     return g
```

Exercise 64 page 381. (*Optimal order for greedy colouring*) The `coloring` method returns a colouring as a list of lists: the list of vertices with colour 0, the list of vertices with colour 1, etc. In order to obtain an optimal ordering of the vertices for greedy colouring, then, it suffices to list the vertices of colour 0 (their order does not matter), following by those of colour 1, and so on! Thus, for the Petersen graph, we obtain:

```
sage: g = graphs.PetersenGraph()
sage: def optimal_order(g):
....:     order = []
....:     for colour_class in g.coloring():
....:         for v in colour_class:
....:             order.append(v)
....:     return order
sage: optimal_order(g)
[1, 3, 5, 9, 0, 2, 6, 4, 7, 8]
```

A.17 Linear Programming

Exercise 65 page 394. (*Subset Sum*) To each element of the set, we associate a binary variable indicating if the element is included or not in the set of sum zero, as well as two constraints:

- The sum of the elements in the set must be zero.
- The set should not be empty.

This can be coded as follows:

```
sage: l = [28, 10, -89, 69, 42, -37, 76, 78, -40, 92, -93, 45]
```

```
sage: p = MixedIntegerLinearProgram()
sage: b = p.new_variable(binary = True)
sage: p.add_constraint(p.sum([ v*b[v] for v in l ]) == 0)
sage: p.add_constraint(p.sum([   b[v] for v in l ]) >= 1)
sage: p.solve()
0.0
sage: b = p.get_values(b)
sage: [v for v in b if b[v] == 1]
[-93, 10, 45, 78, -40]
```

Let us note that it has not been necessary to define an objective function.

Exercise 66 page 395. (*Dominant set*) The constraints of this linear program over the integers correspond to a covering problem: a set S of vertices of a graph is a dominant set if and only if, for any vertex v of the graph we have that $\big(\{v\} \cup N_G(v)\big) \cap S \neq \emptyset$, where $N_G(v)$ denotes the set of neighbours of v in G. We can write the following code:

```
sage: g = graphs.PetersenGraph()
sage: p = MixedIntegerLinearProgram(maximization = False)
sage: b = p.new_variable(binary = True)
sage: for v in g:
....:     p.add_constraint( p.sum([b[u] for u in g.neighbors(v)])
....:                       + b[v] >= 1)
sage: p.set_objective( p.sum([ b[v] for v in g ]) )
sage: p.solve()
3.0
sage: b = p.get_values(b)
sage: [v for v in b if b[v] == 1]
[0, 2, 6]
```

B
Bibliography

[AP98] Uri M. Ascher and Linda R. Petzold, *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. Society for Industrial and Applied Mathematics, 1998, ISBN 0898714128.

[AS00] Noga Alon and Joel H. Spencer, *The Probabilistic Method*. Wiley-Interscience, 2000, ISBN 0471370460.

[Bea09] Bernard Beauzamy, *Robust mathematical methods for extremely rare events*. On-line, 2009. http://www.scmsa.eu/RMM/BB_rare_events_2009_08.pdf, 20 pages.

[BZ10] Richard P. Brent and Paul Zimmermann, *Modern Computer Arithmetic*. Cambridge University Press, 2010, ISBN 0521194693. http://www.loria.fr/~zimmerma/mca/pub226.html.

[CLO07] David Cox, John Little, and Donal O'Shea, *Ideals, Varieties, and Algorithms*. Undergraduate Texts in Mathematics. Springer-Verlag, 3rd edition, 2007, ISBN 0387946802.

[Coh93] Henri Cohen, *A Course in Computational Algebraic Number Theory*. Number 138 in *Graduate Texts in Mathematics*. Springer-Verlag, 1993, ISBN 3540556400.

[CP01] Richard Crandall and Carl Pomerance, *Prime Numbers: A Computational Perspective*. Springer-Verlag, 2001, ISBN 0387947779.

[Edm65] Jack Edmonds, *Paths, trees, and flowers*. Canadian Journal of Mathematics, 17(3):449–467, 1965.

[FS09] Philippe Flajolet and Robert Sedgewick, *Analytic Combinatorics*. Cambridge University Press, 2009, ISBN 0521898065.

[Gol91] David Goldberg, *What every computer scientist should know about floating point arithmetic*. ACM Computing Surveys, 23(1):5–48, 1991.

[GVL12] Gene H. Golub and Charles F. Van Loan, *Matrix computations*, volume 3. JHU Press, 2012.

[Hig93] Nicholas J. Higham, *The accuracy of floating point summation*. SIAM Journal on Scientific Computing, 14(4):783–799, 1993, ISSN 1064-8275.

[HLW02] Ernst Hairer, Christian Lubich, and Gerhard Wanner, *Geometric Numerical Integration*. Springer-Verlag, 2002, ISBN 3662050200.

[HT04] Florent Hivert and Nicolas M. Thiéry, *MuPAD-Combinat, an open-source package for research in algebraic combinatorics*. Séminaire lotharingien de combinatoire, 51:B51z, 2004. http://www.mat.univie.ac.at/~slc/wpapers/s51thiery.html.

[Joh13] F. Johansson, *Arb: a C library for ball arithmetic*. ACM Communications in Computer Algebra, 47(4):166–169, 2013.

[Mat03] Jiří Matoušek, *Using the Borsuk-Ulam Theorem: Lectures on Topological Methods in Combinatorics and Geometry*. Springer-Verlag, 2003, ISBN 3540003625.

[MBdD+10] Jean Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres, *Handbook of floating-point arithmetic*. Birkhäuser, 2010, ISBN 0817647049.

[Moo66] Ramon E Moore, *Interval analysis*, volume 4. Prentice-Hall Englewood Cliffs, 1966.

[Mor05] Masatake Mori, *Discovery of the double exponential transformation and its developments*. Publications of the Research Institute for Mathematical Sciences, 41:897–935, 2005.

[NO] Sampo Niskanen and Patric R. J. Östergård, *Cliquer — routines for clique searching*. http://users.tkk.fi/pat/cliquer.html.

[PWZ96] Marko Petkovšek, Herbert S. Wilf, and Doron Zeilberger, $A = B$. A K Peters Ltd., 1996, ISBN 1568810636.

[RR05] Nathalie Revol and Fabrice Rouillier, *Motivations for an arbitrary precision interval arithmetic and the mpfi library*. Reliable computing, 11(4):275–290, 2005.

[Sch02] M. Schatzman, *Numerical Analysis: A Mathematical Introduction*. Clarendon Press, 2002, ISBN 9780198508526. https://books.google.fr/books?id=3SuNiR1hzxUC.

[Sto00] Arne Storjohann, *Algorithms for Matrix Canonical Forms*. PhD thesis, Institut für Wissenschaftliches Rechnen, ETH-Zentrum, Zürich, Switzerland, November 2000.

[TBI97] Lloyd N. Trefethen and David Bau III, *Numerical linear algebra*, volume 50. SIAM, 1997.

[TMF00] Gérald Tenenbaum and Michel Mendès France, *Les nombres premiers*. Que sais-je ? P.U.F., 2000, ISBN 2130483992.

[TSM05] Ken'ichiro Tanaka, Masaaki Sugihara, and Kazuo Murota, *Numerical indefinite integration by double exponential sinc method*. Mathematics of Computation, 74(250):655–679, 2005.

[Tuc11] Warwick Tucker, *Validated numerics: a short introduction to rigorous computations*. Princeton University Press, 2011.

[vzGG03] Joachim von zur Gathen and Jürgen Gerhard, *Modern Computer Algebra*. Cambridge University Press, 2nd edition, 2003, ISBN 0521826462. http://www-math.uni-paderborn.de/mca.

[Zei96] Doron Zeilberger, *Proof of the alternating sign matrix conjecture*. Electronic Journal of Combinatorics, 3(2), 1996.

C Index

SageMath, or Sage for short, is an open-source mathematical software system based on the Python language. Sage is developed by an international community of hundreds of teachers and researchers, whose aim is to provide an alternative to the commercial products Magma, Maple, Mathematica and Matlab. To reach this goal, Sage relies on many open-source programs, including GAP, Maxima, PARI and various scientific libraries for Python, to which thousands of new functions are added. Sage is freely available and is supported by all modern operating systems.

For high school students, Sage provides a wonderful scientific and graphical calculator. It efficiently supports undergraduate students in their computations in analysis, linear algebra, calculus, etc. For graduate students, researchers and engineers, Sage provides the most recent algorithms and tools for many domains of mathematics. This is why several universities all around the world already use Sage at the undergraduate level, including for student internships.

This book, written by researchers and teachers at different levels (high school, undergraduate, graduate) focuses on the underlying mathematics, which is necessary to efficiently use Sage. In such a way, it is more a mathematical book illustrated by concrete examples with Sage than a reference manual.

The first part of the book is accessible to high school and undergraduate students. The content of the other parts is more suited for graduate students, teachers and researchers.