Introduction to Symbolic Computation

Release 1.7.6

Jan Verschelde

Jun 20, 2023

Contents

CHAPTER 0

Preface

This document contains the lecture notes for the course MCS 320, introduction to symbolic computation, at the University of Illinois at Chicago.

The course was inspired by the book of A. Heck, introduction to Maple, [Heck96]. From 2001 till 2014, the course was offered, using Maple, about once every academic year. Since Spring 2015, Maple was replaced by SageMath. Release 1.0.0 of the lecture notes dates to 2 May 2019, after teaching the fifth semester with SageMath.

The course provides an introduction to computer algebra via practical experimentation in a computer lab. The sections in the lecture notes correspond to SageMath sessions executed during 50 minutes class meetings. Exercises and quizzes are vital. Many of the assignments listed at the end of each lecture are former exam questions. Absent from the lecture notes are the descriptions of the computer projects.

This course is the first course on the computational track, followed by Numerical Analysis and Introduction to Industrial Math & Computation. The main outcome of this course is that students are confident with the use of mathematical software.

1. While software is obviously tied to programming, this is not a programming but a computational course. A programming problem is solved by a correctly working program. Given a correctly working program, a computational problem starts with the setup of the input and ends with the interpretation of the output.
2. Formulating the solution of a computational problem is done in a notebook, where coding cells are interleaved with cells documenting the work flow. Computational literacy is expressed in the describing the solution of a computational problem.
3. The purpose of computing is insight not numbers (as Richard Hamming said) and the correct computational perspective on problems leads to a better understanding of mathematics.

The development of the lecture notes benefited from *Sage for Undergraduates* by Gregory Bard [Bard15] William Stein's *Sage for Power Users* [Stein12], and the *Computational Mathematics with SageMath* [Zimm18] by Paul Zimmerman et al.

References

CHAPTER 1

Part I: First Steps

The first part of the course consists of the first 9 lectures. One strength of computer algebra is the wide range of number types and arithmetic that go beyond what is offered by the computer arithmetic. In addition to exact rational and arbitrary multiprecision floating-point arithmetic, we cover complex and algebraic numbers. For the latter, we introduce the basics of expressions, and in particular polynomials with coefficients over any number field.

1.1 Lecture 1: Welcome!

In this first lecture we define computer algebra and sketch the organization of the course. The most important aspect of this lecture is to get started with the notebook interface in SageMath.

We can run SageMath in a terminal window, in the cell server, in a notebook, or in the cloud. Our running computation involves the symbol `pi` which SageMath recognizes as the mathematical constant π. The lecture ends with finding out which system performs a particular calculation.

1.1.1 What is Computer Algebra?

Computer Algebra is the discipline that studies the algorithms for Symbolic Computation. In Symbolic Computation, one computes with symbols, rather than with numbers. In this course we are mostly concerned with the practical aspects of Symbolic Computation, in particular its implementation and its application to solve practical mathematical problems.

The lecture notes are organized in four parts:

1. First steps with SageMath
2. Polynomials and rational expressions
3. Functions and Calculus
4. More Advanced SageMath
5. Some Components of SageMath

1.1.2 What is SageMath?

The original name Sage was short for Software for Algebra, Geometry, and Experimentation. Later, SAGE became Sage and now we refer to SageMath. Below are some important points about SageMath:

- SageMath is released as free and open source software under the terms of the GNU General Public License.
- As scripting language, SageMath uses Python. SageMath has a large, active developers community.
- By design, SageMath does not reinvent the wheel, but bundles various powerful free and open source mathematical software systems, such as GAP, Maxima, R, Singular, sympy, numpy, scipy, etc. This design principle is visualized in Fig. 1.1. The figure is copied from [BS10].

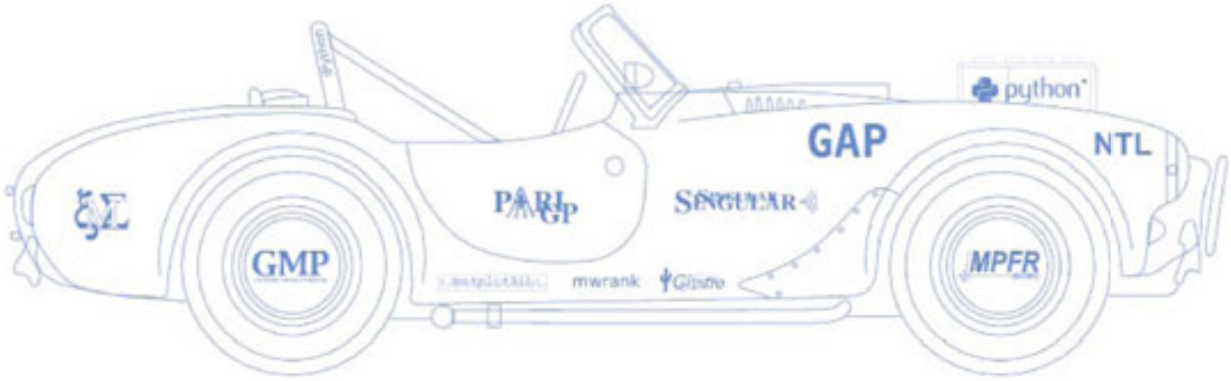

Fig. 1.1: The motto of SageMath: building the car instead of reinventing the wheel.

- Try it without installing, via the SageMath Cell Server: https://sagecell.sagemath.org/, or with CoCalc: https://cocalc.com.

1.1.3 Running SageMath

There are two main modes to run SageMath:

- In a terminal window, which is similar as using the command line interface to interactive software systems. This works fine for short calculations, when the right commands are knows in advance.
- The notebook interface runs in a web browser. The help gives direct access to the reference manual and other documentation. The notebook interface is natural with cloud computing.

A terminal session with SageMath

```
sage: pi
pi
sage: sin(pi)
0
sage: '%.14e' % pi
'3.14159265358979e+00'
sage: type(_)
<type 'str'>
sage: type(pi)
<type 'sage.symbolic.expression.Expression'>
sage: p = plot(sin,(-pi,+pi))
sage: p
```

For a more interesting plot, consider the sine function with a decaying amplitude, as defined by `exp(-x^2)*sin(8*x)`.

As a computer algebra system, Sage is mainly for symbolic computation. For its numerical computations, we are not limited to the hardware machine arithmetic. For example, to see the decimal expansion of π with 20 decimal places:

```
sage: pi20 = pi.n(digits=20)
sage: pi20
3.1415926535897932385
sage: sin(pi20)
6.5640070857470010853e-22
sage: type(pi20)
<type 'sage.rings.real_mpfr.RealNumber'>
```

Unlike the exact value for π, the value of `sin(pi20)` is no longer zero.

In the SageMath Cell Server to plot the sine function, we can enter:

```
var('x')
plot(sin(x), (-pi, pi)).show()
```

When running the notebook interface, we can document our computations with text that follows the # symbol.

1.1.4 Which systems in SageMath are used?

Sage is built out of several systems. If you are curious to see which software package executed your calculation, you could proceed as follows:

```
sage: from sage.misc.citation import get_systems
sage: get_systems('Rational(0.75)')
['MPFR']
sage: get_systems('RealField(10)(pi)')
['MPFR', 'ginac']
sage: import sage.misc.citation
sage: help(sage.misc.citation)
sage: sage.misc.citation.systems.keys()
```

1.1.5 Assignments

1. Go to http://www.sagemath.org/tour.html and take the Feature tour.

   After taking this tour, write a couple of sentences (one paragraph long) about what interests you most.

2. Create an account on CoCalc, at https://cocalc.com, and make a notebook with calculations from this lecture.

3. If there is sufficient disk space available on your home computer or laptop, consider installing SageMath from source. On Linux systems, you should not do this installation as root, but as an ordinary user.

4. Which system executes the statement `pi.n(digits=20)`?

1.1.6 References

1.2 Lecture 2: the notebook – structuring and documenting work flow

We can use the notebook as a calculator or as a computer program, but such use does not take the full advantage of all capabilities of the notebook. The SageMath project is separate from the Jupyter project [KRPetal16].

If you compiled SageMath from source, then you are likely starting in a Terminal window. To launch the program with a Jupyter notebook, type

```
sage --notebook='ipython'
```

at the command prompt in a Terminal window.

1.2.1 the Jupyter notebook in CoCalc

To use the Jupyter notebook in CoCalc, in the creation of a new file, choose for the Jupyter notebook among the Select the type options, see Fig. 1.2.

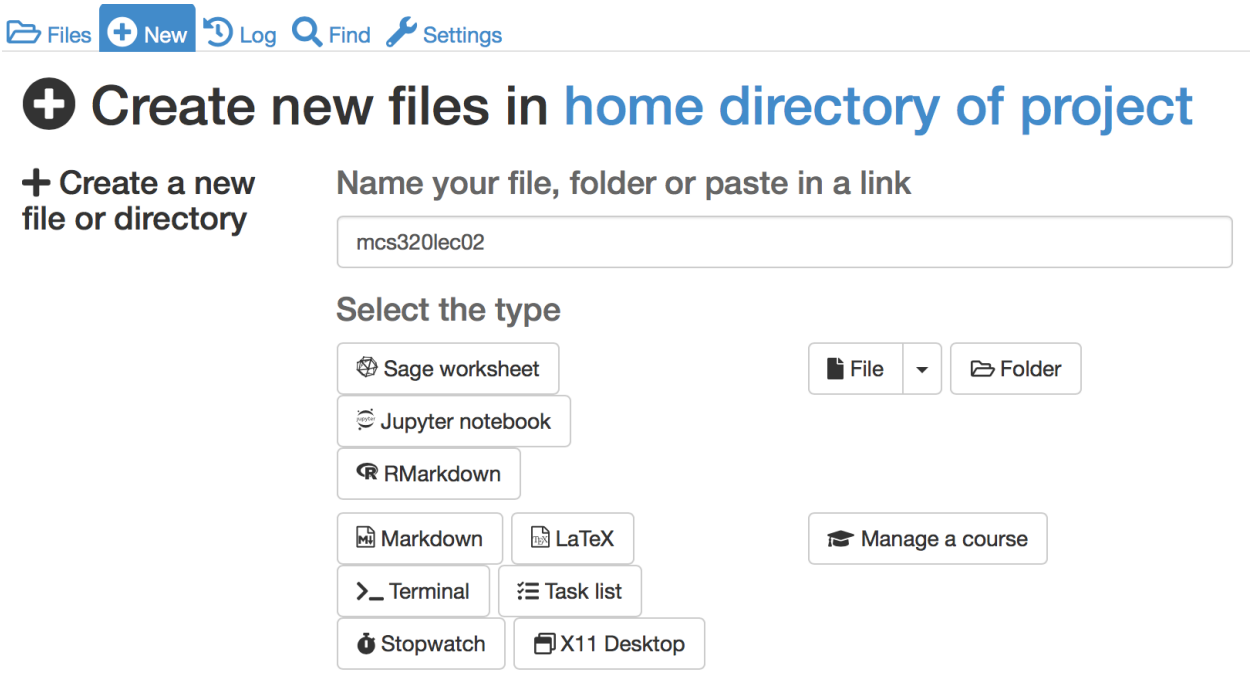

Fig. 1.2: Selecting the Jupyter notebook in CoCalc.

Jupyter notebooks are stored in files with the extension `ipynb`. In the CoCalc environment, to save a notebook to a file on your computer, select the first `Notebook (.ipynb)...` option of the `Download as...` menu of the `File` menu, available at the top left corner, see Fig. 1.3.

Be aware that many browsers will (sometimes by default) add the extension `.txt` to the downloaded file, which then causes problems when loading the downloaded file in a SageMath session or when uploading to the cloud.

Note the `Print preview...` option, shown in the menu in Fig. 1.3. Selecting this option converts the notebook into an HTML file. The HTML file can then be printed, just as you would print out any web page in your browser.

The opposite to a download is an upload, useful if we want to bring a Jupyter notebook from our computer into CoCalc. To upload, click on the `Upload` button, shown at the the top right in Fig. 1.4.

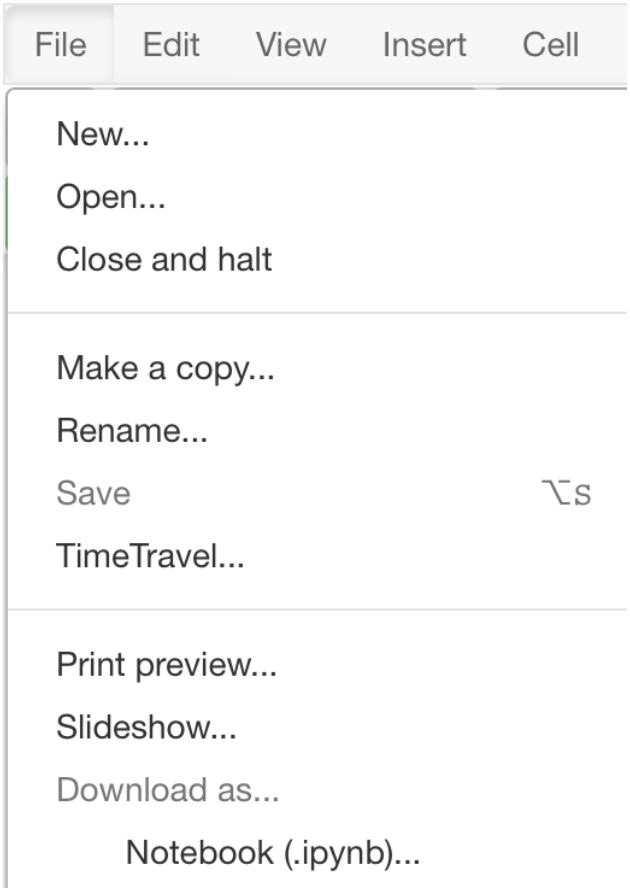

Fig. 1.3: Saving a Jupyter notebook in CoCalc onto your computer.

Fig. 1.4: The upload button to upload a Jupyter notebook from your computer.

1.2.2 Formatting a Notebook

By default, all cells are of type `Code` and contain instructions for SageMath to execute, as if at the command prompt or as in a script.

To introduce structure in a notebook, we can add section headings. Suppose we would want to place the title *Computing with Symbols* at the start of our notebook.

1. Insert a new cell before the first cell in the notebook, via the `Insert Cell Above` available from the dropdown `Insert` menu at the top of the notebook.
2. Instead of the `Code`, we select `Heading` as the type of the cell. The type of the cell can be modified via the toolbar of the notebook.
3. Once the type of the cell is changed to `Heading`, we type in the title `Computing with Symbols` after the `#` symbol.
4. Executing the cell displays the formatted title.

To add text, we follow the same steps as before, but select the `Markdown` type instead of the default `Code`.

```
What is between dollar signs ($) will be formatted by LaTeX.
Text between two single opening left quotes (``)
and two single closing left quotes will not be formatted.
```

Executing the cell displays the formatted text. To see the original unformatted text, select the `Raw NBConvert`, instead of the default Code.

1.2.3 Exact, Symbolic, and Approximate Computations

We distinguish between three types of computations:

1. An *exact* computation is *free from errors*.

   Example: $\frac{1}{2} + \frac{1}{4} = \frac{3}{4}$.

2. A *symbolic* computation operates on *symbols*.

   Example: $\sin(\pi/3) = \sqrt{3}/2$.

   Note: $\sqrt{3}$ is the symbol for the unevaluated function `sqrt(3)`.

3. An *approximate* computation works *in limited precision.*

   Example: $\sin(3.1415/3) = 0.8660$.

Consider the following computation: `Pi10 = pi.n(digits=10)`

The `Pi10` evaluates to `3.141592654`, an approximation for π, accurate to 10 decimal places.

The statement `delta = pi - Pi10` shows `pi - 3.141592654`, which is a symbolic expression.

Now consider the following two statements:

1. `delta.n(digits=10)` evaluates to `0.0000000000`
2. `delta.n(digits=11)` evaluates to `3.6379788071e-12`

Which outcome is correct? Well, consider:

1. The `0.0000000000` shows that the expression `pi - Pi10` evaluates indeed to zero when the working precision is 10 decimal places.

2. When evaluating `pi - Pi10` with 11 decimal places in the working precision, then the error is `3.6379788071e-12`.

Both outcomes are correct with regard to the working precision.

1.2.4 Assignments

1. Copy the assignments of the first lecture in the Markdown cells of a Jupyter notebook. Use `Exercise 1`, `Exercise 2`, etc as the headers in front of each assignment.
2. Add cells to the Jupyter notebook of the first assignment. Each added cell contains your solution of an exercise.

References

1.3 Lecture 3: SageMath as a Calculator – getting Help

In this lecture we show how to use the SageMath notebook as a calculator and explore the extensive help facilities.

If we compute with arbitrary multiprecision, then we must raise the precision of the field to compute the rounding errors correctly. We illustrate this with the computation of a numerical approximation to π, accurate up to 30 decimal places. We explore the help system of SageMath, priming the continued fractions of the next lecture.

1.3.1 Getting Started

We run the notebook in multicell mode. In the command line interface, the commands are executed in sequence, in the notebook interface, we can execute commands out of order.

We enter mathematical expressions in a cell. The expressions are executed by clicking on the evaluate button which appears below each cell. For example:

```
34^34
```

With the underscore we get the result from the previous computation. Note that *previous* refers to time, not location. To experience this, we add in the next cell the underscore, and then in the following cell:

```
2*3
```

When we evaluate we see the result 6 appear and when we evaluate the cell with the underscore that is located above the cell 2*3, then we will see the result change from the value of 34^34 into 6. Consider the next cell with content

```
pi
pi - _
```

Repeatedly clicking on the evaluate button will give different results each time. Can you explain the differences?

At the command line interface: (but not in the notebook interface) we can use double and triple underscores:

```
sage: 2
2
sage: 3
3
sage: 4
4
```

(continues on next page)

(continued from previous page)

```
sage: ___
2
sage: ___
3
sage: ___
4
sage: __
3
sage: _
3
```

Consider the computation of the difference between pi and a 30 digit numerical approximation for pi. The shortest way to obtain this is via `pi.n(digits=30)`. Another way is via `numerical_approx` as illustrated below:

```
numerical_approx(pi,digits=30)
```

But often we want to compute with a precision of 30 decimal places. Then we compute with a `RealField` of prescribed precision. For a 30-digit approximation of pi we then type in a cell:

```
R = RealField(30*log(10,2))
pi30 = R(pi)
strpi30 = str(pi30)
print('%d-digit approximation for pi : %s' % (len(strpi30),strpi30))
```

With `log(10,2)` we compute the binary logarithm of 10, because the argument of `RealField` is expressed in bits. After the creation of `R`, we convert the symbol for the mathematical constant pi into a 30-digit approximation for pi. This approximation is stored in the variable `pi30`. The conversion to the string representation `str(pi30)` is used to check that indeed we do have 30 decimal places, as in confirmed by the output produced by the cell.

As simple computation of the difference between pi and pi30 can be done as

```
R(pi - pi30)
```

which returns `0.00000000000000000000000000000`

The problem with the previous computation is that the precision of the RealField was not large enough. To make the difference with 40 decimal places:

```
RealField(40*log(10,2))(pi-pi30)
```

yields 1.69568553528388823010177990083205496387e-31. Thus we verified that `pi30` has the desired accuracy of 30 decimal places.

Next we illustrate the difference between precision and accuracy:

```
pi40 = RealField(40*log(10,2))(pi30)
```

we have made a variable `pi40` with a working precision of 40 decimal places. But because it takes the value of `pi30`, the accuracy of `pi40` is no more than 30 decimal places.

1.3.2 Getting Help

To see the documentation for a specific command, in a cell we can type

```
help(RealField)
```

To look for a specific topic, we can use the `search_doc` as illustrated below:

```
search_doc("numerical approximation")
```

Then the user gets redirected to the extensive online documentation, shown in Fig. 1.5 below.

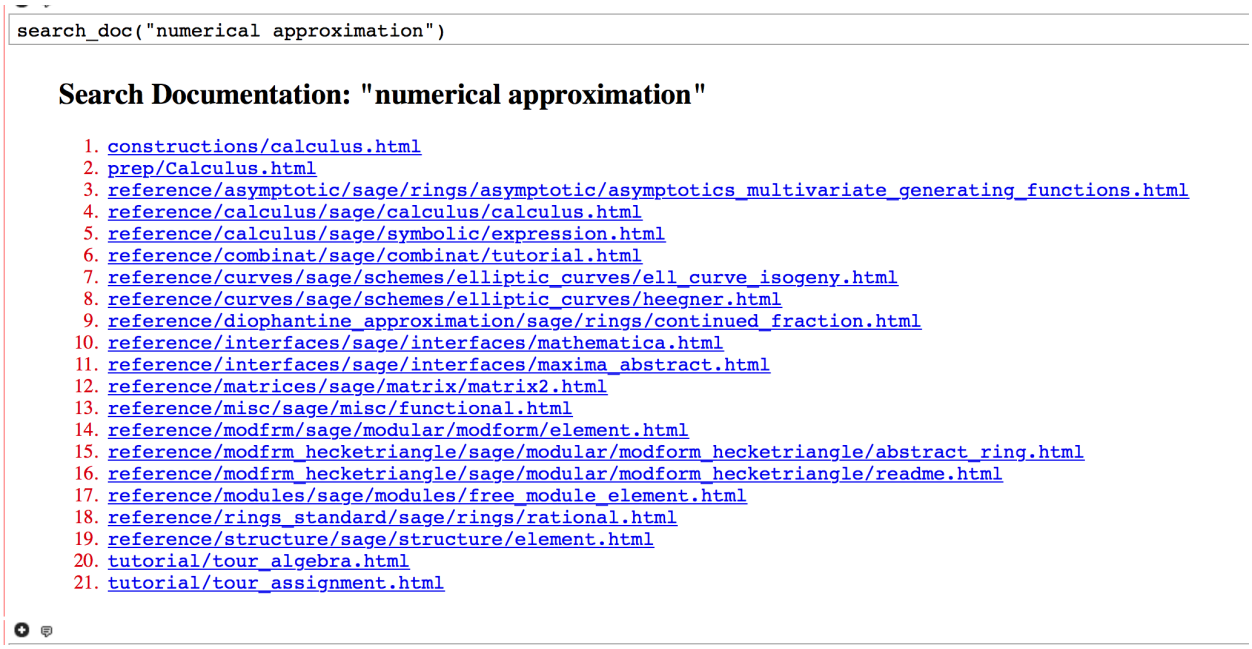

Fig. 1.5: The documentation in SageMath consists of tutorials, the reference manual, constructions (how do I construct .. in Sage?), and PREP tutorials aimed at undergraduate students.

The list in Fig. 1.5 was produced with version 8.6 of SageMath. As the documentation has grown, the output with current versions is much larger.

1.3.3 The Packages in SageMath

There are many packages in SageMath where we can find pi:

```
print(type(pi))
print(type(math.pi))
import sympy
print(type(sympy.pi))
import numpy
print(type(numpy.pi))
```

The second and fourth type is the ordinary `float` whereas the first and third type are symbolic.

To work with the pi as defined by sympy:

```
from sympy import pi as sympypi
print(sin(sympypi))
print(sympypi.evalf(30))
```

The first number we see is `0` followed by a 30-digit approximation for pi.

1.3.4 Assignments

In assignments where you are asked to explain, please formulate your answers with complete sentences.

1. Explain the differences between pi and Pi in SageMath. Illustrate the differences with the appropriate commands.
2. Verify that SageMath knows that the exponential of the natural logarithm of 2 equals 2.
3. The mathematical constant *e* is just as pi a transcendental number. Show that SageMath knows the constant *e* by taking its natural logarithm. Calculate a 30-digit approximation for *e*. Verify the accuracy of this approximation by calculating its difference with the exact value.
4. Consider the following sequence of commands:

   ```
   R = RealField(30*log(10,2))
   print(tan(R(pi/2)))
   print(R(tan(pi/2)))
   ```

   Why do these two commands give different answers? Which of the two commands gives the correct answer? Also compare `R(tan(pi))` with `tan(R(pi))` and explain the differences.
5. The decomposition of 2015 as a product of prime numbers is `5 * 13 * 31`. Use the help system of SageMath to find the command to compute the factorization of a natural number.
6. Explain the difference between 0 and 0.000 and give two good examples of SageMath calculations that result in 0 and 0.000.

1.4 Lecture 4: Exact and Floating-Point Numbers

The capability to compute exactly and in high precision is one of the main features of computer algebra. In this lecture we illustrate how to compute exactly with integer and rational numbers, and to compute with multiprecision arithmetic.

We can approximate a transcendental number such as π as a floating-point number up to any number of decimal places. Instead of a floating-point approximation, we may approximate with a continued fraction expansion. In Sage we can compute continued fractions up to a certain number of terms or up to a given number of bits in the precision. The convergents of a continued fraction gives us a sequence of increasingly more accurate rational approximations for π. We cover machine precision. Irrational and algebraic numbers are defined. We introduce the reverse operation to computing an approximation of an irrational number. Given an approximation for a number and a tolerance, we may find the smallest polynomial that has this number as a root. For example, with sympy we compute $\sqrt{2}$ from an approximation for the $\sqrt{2}$, as $\sqrt{2}$ is a solution of $x^2 - 2$.

1.4.1 Integer and Rational Numbers

The division of two integer numbers gives a rational number. Selecting its numerator and denominator is straightforward:

```
a = 3/4
print type(a)
print numerator(a), denominator(a)
```

Any irrational or transcendental can be approximated with continued fractions. A continued fraction is defined by a list of natural numbers, which are called convergents. For example:

```
x = sqrt(2)
c = continued_fraction(x)
print c
show(c)
```

which prints `[1; 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, ...]`

and the output of `show(c)` is displayed in Fig. 1.6.

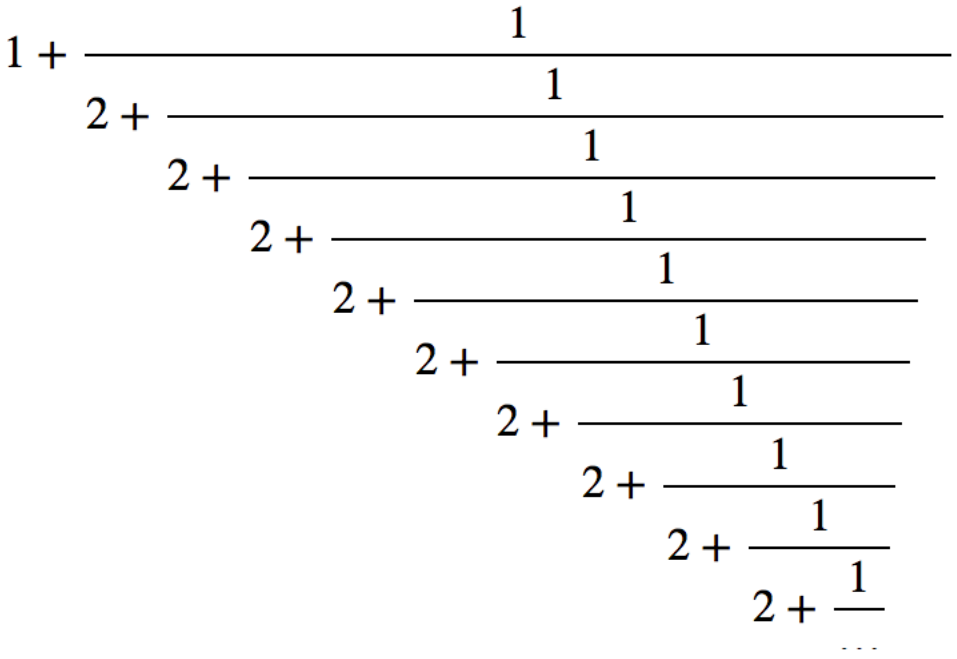

Fig. 1.6: The continued fraction expansion of $\sqrt{2}$.

We can verify that

$$1 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2}}}}$$

approximates $\sqrt{2}$ up to 3 correct decimal places:

```
f = 1 + 1/(2 + 1/(2 + 1/(2 + 1/2)))
print(f)
print(float(f), float(x))
```

The continued fraction defines *convergents* which are the consecutive best rational approximations.

```
c.convergents()
```

The output is a `lazy list` which shows only the first three numbers. To compute the first five convergents, one takes a slice of length 5 and then converts that slice to a list:

```
v = c.convergents()
v[:5].list()
```

Each convergent is a rational approximation. The output of the last command is `[1, 3/2, 7/5, 17/12, 41/29]` which is a list of increasingly more accurate approximations converging to $\sqrt{2}$.

The quickest way to get a rational approximation for the square root of 2, accurate to 2 decimal places, goes as follows

```
QQ(sqrt(2).n(digits=2))
```

With a list comprehension, we can then compute the sequence of increasingly more accurate rational approximations for `sqrt(2)`.

```
[QQ(sqrt(2).n(digits=k)) for k in range(2,10)]
```

Observe that this sequence is different from the convergents, because the list above contains rational approximations with a prescribed accuracy (of `k` decimal places), whereas the convergents give the sequence of the next best rational approximations.

1.4.2 Floating-Point Numbers

The default floating-point type is `float` as in Python. The machine precision is defined as the *smallest* positive number we can add to 1.0 and see a difference from 1.0. A double float has 53 bits in its fraction, a single float has a fraction of 22 bits long. We can verify the machine precision calculation with the predefined constants of the numpy package.

```
print('machine precision (double float) :', float(2^(-52)))
print('machine precision (single float) :', float(2^(-23)))
import numpy
from numpy import finfo
print(finfo(float).eps)
print(finfo(numpy.float32).eps)
```

To compute with higher than double precision, we can make a RealField. Since the argument is in bits, we multiply with the number of decimal places we want in our precision with the 2-logarithm of 10. For 50 decimals, we do

```
nbits = 50*log(10,2)
print('number of bits for 50 decimal places :', int(nbits))
R = RealField(nbits)
print('50-digit approximation of sqrt(2) =', R(x))
```

We will use this field to verify the accuracy of the convergents calculating in R with a list comprehension

```
c = continued_fraction(sqrt(2))
v = c.convergents()
a = v[:20].list()
print(['%.2e' % R(sqrt(2)-p) for p in a])
```

The numbers are displayed in scientific format and we see the exponent gradually decreasing till -15, which indicates that the last convergent is accurate up to 15 decimal places.

A number as $\sqrt{2}$ is an irrational number. But it is not a transcendental number because $\sqrt{2}$ is a root of $x^2 - 2 = 0$ and therefore we say that $\sqrt{2}$ is an *algebraic number*. With the `nsimplify` of `sympy`, given an approximation for a root, we can reconstruct the polynomial and thus find the closest exact number of the algebraic number.

```
asqrt2 = numerical_approx(sqrt(2),10)
print(asqrt2)
from sympy import nsimplify
print(nsimplify(asqrt2,tolerance=0.001,full=True))
```

The last statement prints `sqrt(2)`, obtained from a 10-bit approximation `1.4` for $\sqrt{2}$.

1.4.3 Assignments

1. Explain the outcome of `3^4^5`. In particular, what is the order of execution of the two exponentiation operations?
2. Write $5^{4^3} - 1$ as a product of prime numbers.
3. The greatest common divisor of two integer numbers a and b can be written as a linear combination (with integer coefficients k and ℓ) of a and b: $\gcd(a, b) = ka + \ell b$.

   In Sage this is achieved with the command `xgcd`. Look in the help page of this command to compute the coefficients of the linear combination of the greatest common divisor of 12214 and 2012. Give the command you type in to find these coefficients and also give the command(s) to verify the result.
4. What is the difference in Sage between $1/3 + 1/3 + 1/3$ and $1.0/3 + 1.0/3 + 1.0/3$? Explain.
5. Consecutive rational approximations for π are 3, 22/7, 333/106, 355/113, ... In this sequence, what is the next more accurate rational approximation for π? How many decimal places are correct in this next rational approximation?
6. Explain the difference between $1.0 + 10^{-20}$ and $1 + 10^{-20}$. How can you make Sage return the same *correct* value of these two sums?
7. The golden ratio is defined as $r = \frac{1+\sqrt{5}}{2}$. Give all Sage commands

   (i) to compute a rational approximation for r accurate with three decimal places;

   (ii) to show that the accuracy of this approximation is indeed three decimal places;

   (iii) to compute a sequence of the first ten consecutive rational approximations, where the k-th number of the sequence is accurate with k decimal places.
8. Consider `r = 1.2599.` Find the algebraic number that is closest to `r`.
9. Consider $\sqrt{3}$. Compute the first ten terms in the continued fraction expansion. Write the last element in the corresponding list of convergents.

   Compare the floating-point approximation of this rational approximation for $\sqrt{3}$ with the floating-point approximation for $\sqrt{3}$. How many decimal places in the rational approximation are correct?

   Give the floating-point approximation of the rational approximation for $\sqrt{3}$. Write only those decimals that are correct.

1.5 Lecture 5: Complex and Algebraic Numbers

We have integer, rational, and irrational numbers. Among the irrational numbers, we distinguish between the algebraic and the transcendental numbers. An algebraic number is a root of a polynomial with integer coefficients. For a transcendental number, there does not exist a polynomial with integer coefficients that has that number as one of its roots. Complex numbers are special type algebraic numbers and merit separate treatment.

Sage knows the symbol `I` as the imaginary unit. While we may define `I` as the square root of negative one, this definition has its problems, as we then see when we cover algebraic numbers. A complex number is a special algebraic number and arises from the polynomial x^2+1, which as no real root, or equivalently, x^2+1 does not factor over the field of real numbers. The outcome of the `factor` command on a polynomial in one variable depends on the choice of coefficient field. We introduce finite fields, multiplication tables, and field extensions.

1.5.1 Complex Numbers

We could define the imaginary unit as the square root of -1 and we see that Sage represents sqrt(-1) with the symbol I.

```
print(sqrt(-1))
print(type(I))
print(I^2)
```

The `I` has type `sage.symbolic.expression.Expression` and its square evaluates to -1.

In rectangular coordinates, complex numbers have a real part and an imaginary part.

```
x = 2/3 + 5*I; print(' x =', x, '|x| =', abs(x))
y = complex(2/3,5); print('y =', y, '|x| =', abs(y))
print('x has type', type(x))
print('y has type', type(y))
```

While `x` is considers as an expression, the `y` is the `complex` type of Python. As an expression, the real and imaginary part of a complex number may be any number type, whereas Python's complex number is a tuple of two hardware floats.

An alternative to representing a complex number via its real and imaginary part (rectangular coordinates) is the polar representation as its absolute value and an argument. We first must make sure the number belongs to the Complex Double Field, abbreviated by CDF.

```
ax = CDF(x).argument(); rx = abs(x)
print(ax, rx, 'x =', complex(rx*(cos(ax) + I*sin(ax))))
```

and we recognize the rectangular representation of `x` as we defined `x` above.

Complex numbers are introduced so every polynomial with real coefficients of degree d has d roots, counted with multiplicities. By default, we get complex roots if we solve a polynomial equation. If we want to see the multiplicities of the roots, we must turn on the `multiplicities` flag.

```
var('z')
solve(z^2 + 4 == 0, z)
solve(z^2 - 2*z + 1 == 0, z, multiplicities=True)
```

Defining I as the square root of -1 has its problems, as we will try to illustrate next.

```
a = (-1 + I)^2; b = (1 - I)^2;
```

If we apply the `sqrt()` function to `a` and `b` we would expect to see `-1 + I` and `1 - I` respectively. Instead we get `sqrt(-2*I)` in both cases. Why? Because, if we take a square root, we are solving a quadratic equation.

```
aq = z^2 - a == 0; bq = z^2 - b == 0;
```

The solution is $\pm\sqrt{-2I}$. With the option `all` we get to see all square roots:

```
sqrt(a, all=True)
```

In contrast, we can also prevent evaluation, after coercing the argument of `sqrt()` to a symbolic ring, denoted by SR.

```
sqrt(SR(4), hold=True)
```

1.5.2 Algebraic Numbers

A square root is an algebraic number. The smallest polynomial that has an algebraic number as its root is the minimal polynomial. The command

```
sqrt(2).minpoly()
```

returns $x^2 - 2$ as the minimal polynomial of $\sqrt{2}$.

The outcome of the factoring of a polynomial depends on the number field.

```
print(factor(z^2 - 1))
print(factor(z^2 - 2))
```

The first command will print `(z + 1)*(z - 1)` whereas the second polynomial does factor and we get `z^2 - 2` returned. If a polynomial does not factor over a specific number field (the default number field is the field of rational numbers), then we say that the polynomial is *irreducible over that specific number field*.

We can define a number field where `c` is the square root of 2, we can compute modulo the minimal polynomial of sqrt(2), that is `c^2 - 2 = 0`. We use `c^2 = 2` to simplify expressions in `c`.

```
K.<c> = NumberField(x^2 - 2); print K
print(c^2 - 2)
print(c^3)
```

The defining polynomial of the number field `K` is `x^2 - 2`. In this number field, the expression `c^2 - c` evaluates to zero and `c^3` simplifies to `2*c`. If we now consider a polynomial ring over the number field K, then the polynomial x^2 - 2 will factor.

```
R.<x> = PolynomialRing(K)
p = x^2 - 2
factor(p)
```

On return we see `(x - c) * (x + c)`.

In coding and cryptography we often work with finite fields.

```
k = GF(8,'c'); print(k)
e = [a for a in k]; print(e)
```

The `GF` is an abbreviation for Galois Field. This makes `k` a field of size 2^3 with elements

```
[0, c, c^2, c + 1, c^2 + c, c^2 + c + 1, c^2 + 1, 1]
```

In a field every element has a multiplicative inverse. We can see this in table of multiplications.

```
for x in e: [x*y for y in e]
```

The multiplication table is

```
[0, 0, 0, 0, 0, 0, 0, 0]
[0, c^2, c + 1, c^2 + c, c^2 + c + 1, c^2 + 1, 1, c]
[0, c + 1, c^2 + c, c^2 + c + 1, c^2 + 1, 1, c, c^2]
[0, c^2 + c, c^2 + c + 1, c^2 + 1, 1, c, c^2, c + 1]
[0, c^2 + c + 1, c^2 + 1, 1, c, c^2, c + 1, c^2 + c]
[0, c^2 + 1, 1, c, c^2, c + 1, c^2 + c, c^2 + c + 1]
[0, 1, c, c^2, c + 1, c^2 + c, c^2 + c + 1, c^2 + 1]
[0, c, c^2, c + 1, c^2 + c, c^2 + c + 1, c^2 + 1, 1]
```

Observe that, except for the first row, there is 1 in every row. Except for the first column, there is 1 in every column. To verify, let us look at the 1 in the second row.

```
print e[1], '* (', e[-2], ') =', e[1]*e[-2]
```

The output is `c * ( c^2 + 1 ) = 1`.

We can factor over finite fields. We make a polynomial ring with coefficients in the finite field we just constructed:

```
R.<x> = PolynomialRing(k)
p = x^3 + x + 1
factor(p)
```

which leads to `(x + c) * (x + c^2) * (x + c^2 + c)`. Note that over the default field, the polynomial is irreducible.

```
q = z^3 + z + 1; factor(q)
```

To verify that the polynomial $p = x^4 + 3x + 4$ does not factor over the finite field of five elements, we do

```
R.<x> = PolynomialRing(GF(5))
p = x^4 + 3*x + 4; print factor(p)
```

We can then extend the field of five elements with a root of p as follows

```
F.<a> = GF(5).extension(p)
e = [n for n in F]; print e
```

Because the degree of p is four, every element in the field extension `F` can be expressed as a polynomial of degree three or less, because every fourth power of a simplifies via $a^4 = 2a + 1$. We have five choices for every coefficient of a degree three polynomials and since we can make those choices independently, we have a field with 5^4 elements. Over this field extension, we have a factorization.

```
R.<y> = PolynomialRing(F)
q = y^4 + 3*y + 4; print(factor(q))
```

1.5.3 Assignments

1. The complex number z in polar representation is given by the radius (absolute value) $r = 3$ and angle (argument) $\theta = \pi/3$. Compute the exact (no floating-point) value of z in the form $a + bI$.
2. Compute the polar representation of $1 - 2I$.

   Verify that the polar representation you computed corresponds to $1 - 2I$.
3. Execute `solve(x^3 - 5 == 0, x)` and show that all solutions of $x^3 - 5 = 0$ are of the form $\sqrt[3]{5}e^{I\theta}$, with θ being either 0 or $\pm\frac{2}{3}\pi$.
4. Take the complex number $z = 1 + I$ and compute $\sqrt{1/z}$ and $1/\sqrt{z}$. Are the results symbolically the same? Are the results numerically the same? Give reasons for your answers, illustrated with the appropriate instructions.
5. Give the instruction(s) to show that the polynomial $p = x^4 + 3x + 4$ is irreducible over the finite field with 5 elements. Declare z to be a root of p and express z^{13} as a polynomial in z of degree 4 (or less).
6. Compute all irreducible polynomials of degree 2 over a field with 3 elements.

1.6 Lecture 6: Symbols, Variables, and References

While Sage works with dynamic typing in a similar fashion as Python, sometimes we must declare variables explicitly with `var` as in `var('x')`.

Every variable in Sage has a type. We distinguish between names of variables and the objects variables refer to. Putting quotes around a variable name prevents an evaluation and we see how this may be used to make connections between variables. We cover the (partial) evaluation of expressions as needed in the verification of the solutions of a general cubic equation.

1.6.1 Expressions and Names

Sage export mathematical constants, such as `pi`. We can work with `pi` as a variable and assign any value to `pi`. For example:

```
print(pi, type(pi))
pi = 3.14
print(pi, type(pi))
```

The first statement shows that `pi` is an expression. After the assignment, `pi` refers to the value 3.14 which is of type `sage.rings.real_mpfr.RealLiteral`. Did we then loose the value of pi?

With `restore` (we could also call the restore the unassign operation) we can get back the original value of `pi`.

```
restore('pi')
print(pi, type(pi))
print(numerical_approx(pi, digits=30))
```

Mind the quotes in the argument of `restore`, without the quotes we would take the value `pi` refers to. After executing the `restore` we see that `pi` is again an expression. And sure enough, we can see as many digits of `pi` as we like.

The quotes are of course a general construction from Python where everything put between quotes is a string. A value can be stored as a string and then later evaluated with eval.

```
x = 3.14
name = 'x'
print(x, name, eval(name))
```

1.6.2 Verification of Solutions

By default, with we solve an equation with``solve`` we receive a list of expressions.

```
var('z')
equ = z**2 - 3 == 0
sols = solve(equ, z); print(sols)
print(type(sols[0]))
```

We have the option to return a list of dictionaries.

```
sols = solve(equ, z, solution_dict=True); print(sols)
```

Now we see `[{z: -sqrt(3)}, {z: sqrt(3)}]` and `sols` is a list of dictionaries. As key for each dictionary we have the variable name and its corresponding value is the value of the solution of the equation. For example, to select the value of the first solution, using as key the variable name `z`, we can proceed as follows

```
print(sols[0], type(sols[0]))
print(sols[0][z])
```

The last command prints the value `-sqrt(3)`. The dictionary is useful to substitute the value of the variable in the equation we solved, for verification purposes.

```
equ.substitute(sols[0])
```

and we see `0 == 0`.

Suppose we were to assign to `z`. Then we can no longer access the dictionary as directly as before, because `z` now refers to a value, but via `keys()` we retrieve the unevaluated variable with the corresponding value. But the substitution still works.

```
z = 3; print('z = ', z)
print(sols[0].keys(), sols[0].values())
equ.substitute(sols[0])
```

Even though we have lost the use of `z` as a general variable, its former value as a solution is still contained in the dictionary of solutions `sols`.

How do we see the current value to which z refers to?

```
print(eval(str(sols[0].keys())))
```

This will show the list `[3]`. Recall that lists in Python allow to work with shared references.

1.6.3 Evaluation of Expressions

We can express the roots of a polynomial of degree two with symbolic coefficients. Similar formulas exist for a polynomial of degree three. To start over, we clear all the variables in our worksheet with the `reset()`.

```
reset()
var('x, a, b, c')
p = x^3 + a*x^2 + b*x + c
s = solve(p == 0, x, solution_dict=True)
print(s)
```

The formulas look complicated. Let us check a specific example. We want to verify the solution for specific values of the coefficients. An easy choice for the coefficients are the numbers 1.0, 2.0, 3.0 (of type float). Recall that Python allows for simultaneous or tuple assignment.

```
(a, b, c) = (1.0, 2.0, 3.0)
print(a, b, c)
print(p)
print(s[0])
```

The outcome is not what we wanted and expected. Even as we see the specific values for `a`, `b`, and `c` printed, the polynomial still shows up in its original symbolic form $x^3 + ax^2 + bx + c$, and so does its solution. If we were to retype the expression for the polynomial again, then the coefficients would be evaluated, but this is tedious and we do not want to retype the complicated expressions for the solutions.

How to force the evaluation of the coefficients in p and the solution without retyping the polynomial p? We can evaluate an expression.

```
print(p(x=x,a=a,b=b,c=c))
s0 = s[0][x](a=a,b=b,c=c); print(s0)
```

Now we see the polynomial `x^3 + x^2 + 2.00000000000000*x + 3.00000000000000` and a numerical value for the solution.

We can then evaluate the expression `p` at `s0`.

```
print(p(x=s0,a=a,b=b,c=c))
```

To verify whether the value of the expression at the solution will evaluate to zero, we convert to the complex floating point type.

```
print(complex(p(x=s0,a=a,b=b,c=c)))
```

and we see that the value is close enough to the machine precision.

1.6.4 References and Shared Values

We first reset all variables with `reset()`. Then we make variables to share the same value.

```
reset()
var('x, y, z')
x = y
y = z
z = 3
print(x, y, z)
```

What is printed is the sequence `y z 3` which means that `x` refers to `y`, `y` refers to `z`, and `z` refers to `3`. The connections between the variables are shown in Fig. 1.7.

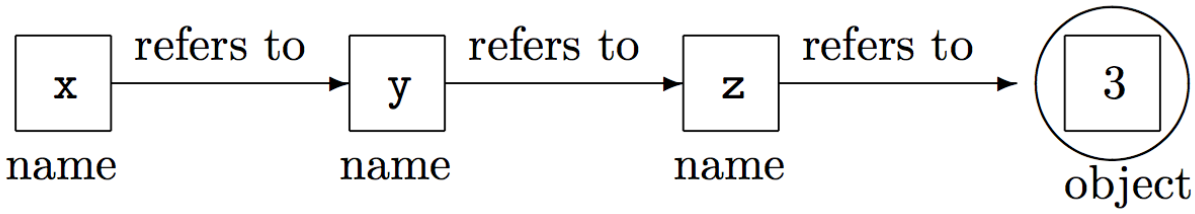

Fig. 1.7: names of variables as references to other variables or values

We can track those references with the `eval()` command. The `eval()` takes a string as argument. With repeated evals we can get from x to 3.

```
ex = eval(str(x)); print x
ey = eval(str(ex)); print ey
print(eval(str(eval(str(x)))))
```

The first `eval` shows `y`, the second one `3`, just as the third nested application of `eval`.

Observe that the order of the assignments matters.

```
reset()
var('x, y, z')
z = 3
y = z
x = y
print(x, y, z)
```

Because the right hand side in an assignment operation gets evaluated, all variables will receive the same value 3. To prevent the evaluation of the right hand side, we use quotes.

```
reset()
var('x, y, z')
z = 3
y = 'z'
x = 'y'
print(x, y, z)
```

The sequence that is printed is `y, z, 3`.

1.6.5 Assignments

1. For variables `a` and `b` consider the expression

$$\frac{1}{a+b\sqrt{2}}$$

We want to compute expressions for the coefficients of the inverse of $a+b\sqrt{2}$, written in the same format, that is: as $c+d\sqrt{2}$ for some variables `c` and `d`.

1. Set up the equations on the coefficients `c` and `d` such that

$$\left(a+b\sqrt{2}\right)\left(c+d\sqrt{2}\right) = 1.$$

2. Apply `solve` to the equations to find expressions for `c` and `d` in function of the `a` and `b` of the original expression.
3. Verify your solution by making the product $\left(a+b\sqrt{2}\right)\left(c+d\sqrt{2}\right)$ and check if the product equals 1 for the computed expressions for `c` and `d`.

2. Execute the statements `reset(); a, b = var('a, b'); b = a; a = 2` and explain the relationships between the variables `a` and `b`. Give the Sage commands and their output to illustrate your explanation.
3. Execute the statements `reset(); a, b = var('a, b'); a = 3` in a cell. What is the next statement so `print(a, b)` shows `3, a` as `b` refers to `a`, as `a` refers to `3`.

1.7 Lecture 7: Number Types and Functions to Store Data

Every object in Sage has a type. The type of an object determines the operations that can be performed on the object. The main data types in Python are lists, dictionaries, and tuples.

The most basic number types in Sage have short abbreviations, they are `ZZ`, `QQ`, `RR`, and `CC`, for the integers, rationals, reals, and complex numbers. We see how to explicitly fix the type of a number with so-called type coercing. The ability to choose random numbers is often very useful. We see how to select left and right hand sides of equations. The lecture ends with a specific application of default parameters of functions that enables us to store data in functions.

1.7.1 Coercing to the Basic Number Types

The main number types are integers, rationals, reals, and complex numbers, respectively denoted by `ZZ`, `QQ`, `RR`, and `CC`. To convert from one type to the other is to coerce.

```
print(ZZ)
```

We can convert a string in hexadecimal format or octal format into decimal notation.

```
a = ZZ('0x10'); print(a)
b = ZZ('010'); print(b)
```

Given the list of coefficients, we can evaluate a number in any base. The following instruction

```
ZZ(142).digits(10)
```

returns the list `[2, 4, 1]` as these are the digits of 142 in the decimal number system , written backwards, starting at the least significant digit. Writing 142 in full is $1 \times 10^2 + 4 \times 10^1 + 2 \times 10^0$. Passing 3 as the argument of `digits()`, we decompose 142 in a number system of base 3, as shown below:

```
c = ZZ(142).digits(3)
print(c)
print(ZZ(c, base=3))
```

The last instruction above takes the coefficients in the list `[1, 2, 0, 2, 1]` and evaluates the coefficients as $1 \times 3^0 + 2 \times 3^1 + 0 \times 3^2 + 2 \times 3^3 + 1 \times 3^4 = 142$.

A quick way to make a rational representation of a floating-point number is via type coercing to `QQ`, the ring of rational numbers.

```
print(QQ)
x = numerical_approx(pi, digits=20)
y = QQ(x); print(y)
```

Note the we cannot coerce `pi` directly to a rational number.

```
z = RR(y); print(z)
print(QQ(z))
```

Observe the difference between the value `21053343141/6701487259` of `y` and the value `245850922/78256779` of `QQ(RR(y))`. This is because the precision of `RR` is the same 53 bit as the hardware double floats.

```
print(RR)
print(RDF)
print(RR == RDF)
```

Although `RR` has a precision of 53 bits, it is not the same as the `RealDoubleField` which is abbreviated as `RDF`. Generating random numbers makes the distinction between the fields `RR` and `RDF` a bit more explicit.

The machine precision is the smallest positive number we can add to 1.0 and still make a difference. We can compute the machine precision as follows:

```
eps = 2.0^(-RR.precision()+1)
print(eps)
```

Note that the `2.0` in the formula for `eps` is necessary, writing `2` instead of `2.0` would have resulted in an exact rational number, not an element of `RR`. We see the value for `eps` again in the following calculation:

```
a = 1.0 + eps
a - 1.0
```

For any real floating-point number `x` smaller than `eps`, the result of `1.0 + x` would have remained `1.0`.

For any `x` belonging to some real number field, we can compute a nearby rational approximation with a given bound on the denominator. For example `x.nearby_rational(max_denominator=1000)` returns a rational approximation for `x` where the denominator is smaller than the given bound of `1000`. To compute a sequence of consecutively larger rational approximations, each time allowing a denominator that is ten times larger than the previous one in the sequence, we can run the following command.

```
[x.nearby_rational(max_denominator=10^k) for k in range(1, 11)]
```

The `nearby_rational` method gives the third type of rational approximations for real numbers. The first two we covered were

1. Rational approximations with prescribed accuracy.
2. Convergents of the continued fraction representation of the real number.

The application of the method `hex()` on a real number shows the hexadecimal expansion of the number.

```
onetenth = 0.1
onetenth.hex()
```

The output is `0x1.999999999999ap-4` which does not suggest a finite expansion. Casting `onetenth` in a `RealField` of a higher precision (e.g.: one hundred bits) will confirm that in binary, the representation of `0.1` cannot be exact. Of course, in the decimal notation `0.1` agrees exactly with $1/10$.

1.7.2 Random Numbers

In simulations, we work with random numbers. With the method `random_element()`, we can generate a random integer number, for example of 3 decimal places, between 100 and 999:

```
ZZ.random_element(100, 999)
```

Random real numbers are generated as follows:

```
x = RR.random_element(); print(x, type(x))
y = RDF.random_element(); print(y, type(y))
```

The type of `x` is `sage.rings.real_mpfr.RealNumber` while the type of `y` is `sage.rings.real_double.RealDoubleElement`. The same distinction can be made between the Complex Field `CC` and the Complex Double Field `CDF`.

```
x = CC.random_element(); print(x, type(x))
y = CDF.random_element(); print(y, type(y))
```

To visualize the distribution of numbers, we can plot a bar chart:

```
L = [RR.random_element() for _ in range(100)]
bar_chart(L)
```

Then the output of `bar_chart(L)` is shown in Fig. 1.8.

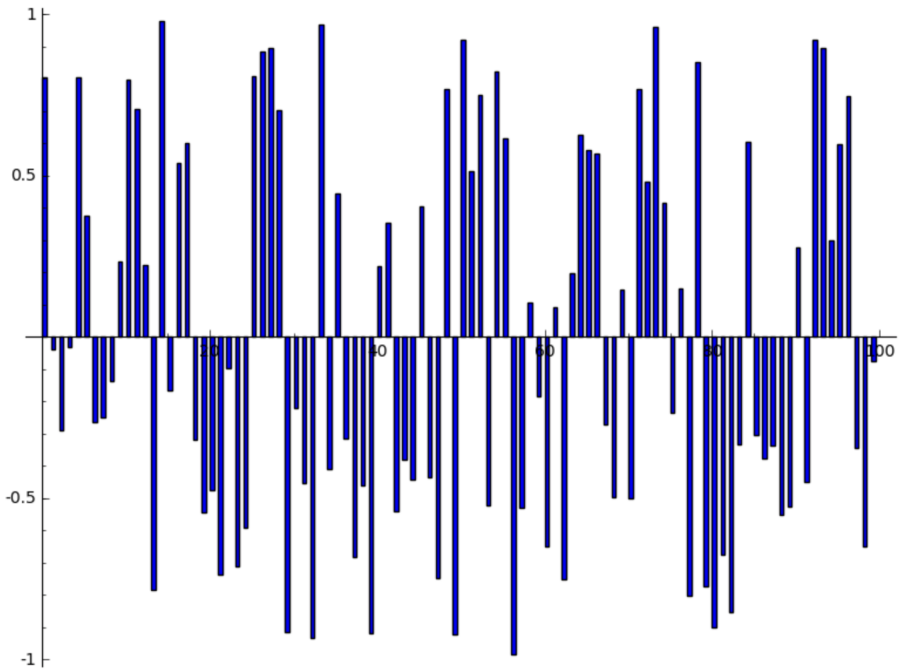

Fig. 1.8: The bar chart of 100 random numbers.

If we sort the numbers, then we can see that the distribution tends to be uniform.

```
L.sort()
bar_chart(L)
```

The sorted numbers are visualized in Fig. 1.9.

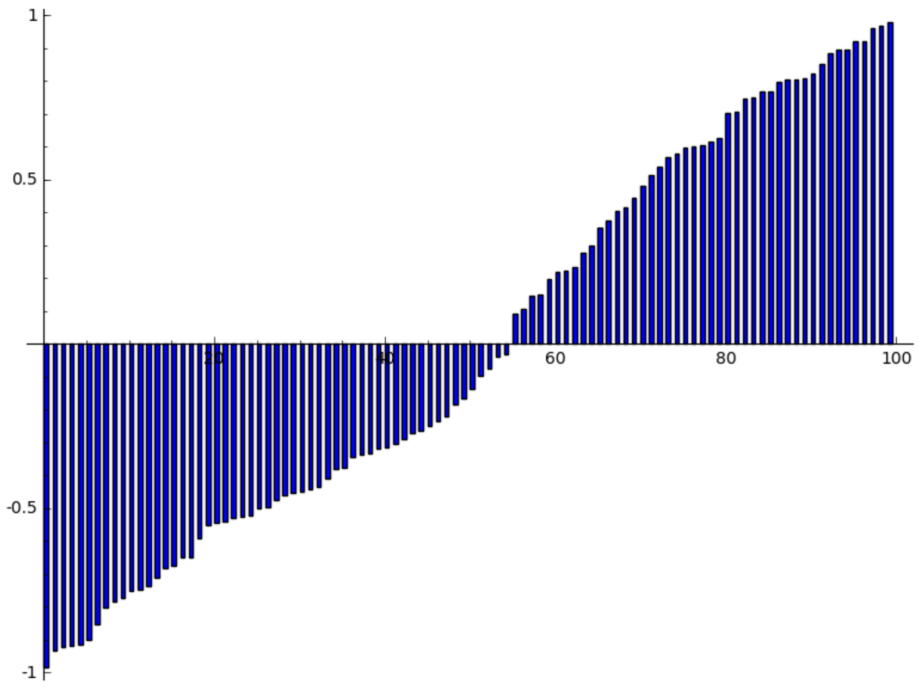

Fig. 1.9: The bar chart of 100 sorted random numbers.

1.7.3 Components of Expressions

We often work with equations.

```
eqn = x**2 + 3*x + 2 == 0
print(type(eqn))
```

The `eqn` is of type `sage.symbolic.expression.Expression`, a type we encountered already many times. Our expression eqn has an operator.

```
print(eqn.operator())
```

The operator is <built-in function eq> and we can select its left and right hand side

```
print(eqn.lhs())
print(eqn.rhs())
```

Alternatives to `lhs()` are `left()` and `left_hand_side()` and instead of `rhs()` we may also use `right()` and `right_hand_side()`.

1.7.4 Storing Data with Functions

The example in this section is taken from the book *Sage for Power Users* by William Stein.

With default arguments in functions, we can store references to objects implicitly. Consider the following function.

```
def our_append(item, L=[]):
    L.append(item)
    print(L)
```

Let us now execute the function a couple times.

```
our_append(1/3)
our_append('1/3')
our_append(1.0/3)
```

We see the following lists printed to screen.

```
[1/3]
[1/3, '1/3']
[1/3, '1/3', 0.333333333333333]
```

To explain what happened, let us print the address of `L` each time. Because we will need to use the address later, we return `id(L)`.

```
def our_append2(item, L=[]):
    L.append(item)
    print(L, id(L))
    return(id(L))
```

We run this function `our_append2` also three times.

```
idL = our_append2(1/3)
idL = our_append2('1/3')
idL = our_append2(1.0/3)
```

and we see the following output

```
[1/3] 4650781440
[1/3, '1/3'] 4650781440
[1/3, '1/3', 0.333333333333333] 4650781440
```

The first time we called `our_append2` without giving an argument for the list `L`, the arguments were evaluated. The effect of `L = []` is that an empty list is created and placed somewhere in memory. Each time the function is called with the default argument of `L`, the same memory location is used. Note that the name `L` does not exist outside the function. Just to check, `print(L)` will result in a `NameError`.

With ctypes we can retrieve the object an address refers to.

```
idL = our_append2(0)
import ctypes
print(ctypes.cast(idL, ctypes.py_object).value)
```

Executing the cell shows

```
[1/3, '1/3', 0.333333333333333, 0] 4650781440
[1/3, '1/3', 0.333333333333333, 0]
```

1.7.5 Assignments

1. Try `QQ.random_element()`. What do you observe? How would you make a random rational number with type coercions?
2. Type `QQ(pi)`. Describe what happens. Is this what you would expect? Write a mathematical explanation.
3. Illustrate how you would generate a random complex number of type `CC`. The number should have absolute value equal to one. *Hint:* think about the polar representation of complex numbers.
4. Consider `x = R100(sqrt(2))` where `R100` is a `RealField` with a precision corresponding to about 100 decimal places.
    1. Compute a list of 10 rational approximations for `x`, starting with a 10 as the first bound on the denominator. The bound of the denominator of the rational approximations equals `10^k` where `k` runs from 1 to 10.
    2. For each approximation in your list, compute the accuracy, that is the relative error for each rational approximation. Write the relative errors in scientific notation.
5. Type `eqn = x^3 + 8.0*x - 3 == 0` and solve this equation. Verify the solutions in the polynomial defined at the left hand side of the equation `eqn` *without retyping the expression* at the left hand side of the equation.

1.8 Lecture 8: Evaluation and Execution

We look at the evaluation of expressions. But first, let us return to modulo arithmetic.

In the basic number types, we forgot to mention the finite rings which we compute modulo a certain number. We return to multiplication tables, which are predefined for finite fields, just as addition tables are. In case we forgot the specific commands, making addition and multiplication tables is a good exercise on double loops in Python, where the inner loop is performed with a for inside a list. In SageMath we can convert an expression into a fast callable object. With the string representation of a fast callable object we can draw the corresponding expression tree that determines the algorithm to evaluate the expression.

1.8.1 Addition and Multiplication Tables

In an earlier lecture we have built the multiplication table of a finite rings. SageMath provides commands for this. To work modulo 3, we define the ring of integers modulo 3.

```
Z3 = Integers(3)
print(Z3)
```

To see all its elements, we convert to a list, simply as `L = list(Z3)`.

The addition table shows all possible additions of any two elements of `Z3`.

```
print(Z3.addition_table())
```

and then we see

```
+  a b c
 +------
a| a b c
b| b c a
c| c a b
```

The multiplication table shows all possible multiplications of any two elements of `Z3`.

```
print(Z3.multiplication_table())
```

and then we see

```
*  a b c
 +------
a| a a a
b| a b c
c| a c b
```

Does this work if we extend `Z3` with an algebraic number? We define an irreducible polynomial with coefficients in `Z3`.

```
P.<x> = PolynomialRing(Z3)
p = x^2 + x + 2
factor(p)
```

As we see the same polynomial `x^2 + x + 2` we cannot factor `p` over `Z3`.

```
K.<a> = Z3.extension(p)
print(K)
```

Then SageMath tells us that `K` is an Univariate Quotient Polynomial Ring in a over Ring of integers modulo 3 with modulus `a^2 + a + 2`.

We cannot simply do `list(K)` to see all elements. We make an explicit loop.

```
L = []
for u in Z3: L = L + [u*a + v for v in Z3]
print(L)
print(len(L))
```

And we see a list of 9 elements. `[0, 1, 2, a, a + 1, a + 2, 2*a, 2*a + 1, 2*a + 2]`. Now we make the multiplication table.

```
for u in L: print [u*v for v in L]
```

and we obtain then

```
[0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 2, a, a + 1, a + 2, 2*a, 2*a + 1, 2*a + 2]
[0, 2, 1, 2*a, 2*a + 2, 2*a + 1, a, a + 2, a + 1]
[0, a, 2*a, 2*a + 1, 1, a + 1, a + 2, 2*a + 2, 2]
[0, a + 1, 2*a + 2, 1, a + 2, 2*a, 2, a, 2*a + 1]
[0, a + 2, 2*a + 1, a + 1, 2*a, 2, 2*a + 2, 1, a]
[0, 2*a, a, a + 2, 2, 2*a + 2, 2*a + 1, a + 1, 1]
```

(continues on next page)

(continued from previous page)

```
[0, 2*a + 1, a + 2, 2*a + 2, a, 1, a + 1, 2, 2*a]
[0, 2*a + 2, a + 1, 2, 2*a + 1, a, 1, 2*a, a + 2]
```

It turns out that `addition_table` and `multiplication_table` are defined when we start with a finite field `GF(3)` instead of `Z3`.

We define an irreducible polynomial with coefficients in `GF(3)`.

```
P.<x> = PolynomialRing(GF(3))
p = x^2 + x + 2
print factor(p)
K.<a> = GF(3).extension(p)
print(K)
```

Now we see that `K` is a `Finite Field in a of size 3^2`. and with a simple `print(list(K))` we can see all its elements:

```
[0, a, 2*a + 1, 2*a + 2, 2, 2*a, a + 2, a + 1, 1]
```

If we want the addition table, we simply do

```
K.addition_table()
```

and we see

```
+  a b c d e f g h i
 +-----------------
a| a b c d e f g h i
b| b f i e g a d c h
c| c i g b f h a e d
d| d e b h c g i a f
e| e g f c i d h b a
f| f a h g d b e i c
g| g d a i h e c f b
h| h c e a b i f d g
i| i h d f a c b g e
```

For the multiplication table, we do

```
print(K.multiplication_table())
```

and then we see

```
*  a b c d e f g h i
 +-----------------
a| a a a a a a a a a
b| a c d e f g h i b
c| a d e f g h i b c
d| a e f g h i b c d
e| a f g h i b c d e
f| a g h i b c d e f
g| a h i b c d e f g
h| a i b c d e f g h
i| a b c d e f g h i
```

1.8.2 Expression Trees

We are interested in the internal structure of expressions.

```
from sage.ext.fast_callable import ExpressionTreeBuilder
etb = ExpressionTreeBuilder(vars=['x','y'])
x = etb.var('x')
y = etb.var('y')
print(x + y)
```

This shows `add(v_0, v_1)`

Other elementary operations are `-`, `*` and `/`

```
print(x - y)
print(x * y)
print(x / y)
```

and we see `sub(v_0, v_1)`, `mul(v_0, v_1)`, and `div(v_0, v_1)`.

Instead of type expression, the expressions involving x and y are of type fast_callable, in a form that can be evaluated fast.

```
s = x + y; print(type(s))
```

We see the type `sage.ext.fast_callable.ExpressionCall`.

Consider the evaluation of a multivariate polynomial in x and y

```
p = x^3 + 4*x*y^2 - 7*y + 2
print(p)
```

This prints `add(sub(add(ipow(v_0, 3), mul(mul(4, v_0), ipow(v_1, 2))), mul(7, v_1)), 2)`. Based on the output of print p we can draw an expression tree.

```
add
 |
 +-- sub
 |    |
 |    +-- add
 |    |    |
 |    |    +-- ipow
 |    |    |    |
 |    |    |    +-- v_0
 |    |    |    |
 |    |    |    +-- 3
 |    |    |
 |    |    +-- mul
 |    |         |
 |    |         +-- mul
 |    |         |    |
 |    |         |    +-- 4
 |    |         |    |
 |    |         |    +-- v_0
 |    |         |
 |    |         +-- ipow
```

(continues on next page)

(continued from previous page)

```
 |    |              |
 |    |              +-- v_1
 |    |              |
 |    |              +-- 2
 |    |
 |    +-- mul
 |         |
 |         +-- 7
 |         |
 |         +-- v_1
 |
 +-- 2
```

An alternative way to draw the expression tree uses `LabelledBinaryTree` and `ascii_art`. We start at the leaves of the expression tree, before the definition of the internal nodes. The string `add(sub(add(ipow(v_0, 3), mul(mul(4, v_0), ipow(v_1, 2))), mul(7, v_1)), 2)` shows there are 9 leaves in the tree.

```
L0 = LabelledBinaryTree([None, None], label='v_0')
L1 = LabelledBinaryTree([None, None], label='3')
L2 = LabelledBinaryTree([None, None], label='4')
L3 = LabelledBinaryTree([None, None], label='v_0')
L4 = LabelledBinaryTree([None, None], label='v_1')
L5 = LabelledBinaryTree([None, None], label='2')
L6 = LabelledBinaryTree([None, None], label='7')
L7 = LabelledBinaryTree([None, None], label='v_1')
L8 = LabelledBinaryTree([None, None], label='2')
```

The `None` and `None` in the first argument of `LabelledBinaryTree` are the left and right children of the tree. The labels of the leaves are the operands in the expression. Then we define the internal nodes.

```
N01 = LabelledBinaryTree([L0, L1], label='ipow')
N23 = LabelledBinaryTree([L2, L3], label='mul')
N45 = LabelledBinaryTree([L4, L5], label='ipow')
N67 = LabelledBinaryTree([L6, L7], label='mul')
N2345 = LabelledBinaryTree([N23, N45], label='mul')
ascii_art(N2345)
```

The children of the internal nodes are the operands and the labels of the nodes are the operators. This shows the expression tree for the monomial `4*x*y^2` in Fig. 1.10.

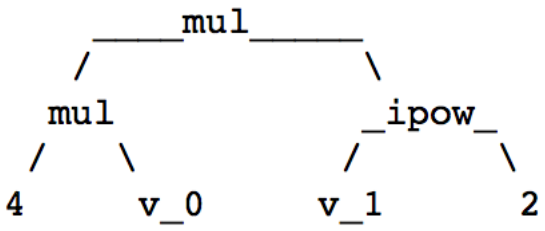

Fig. 1.10: The binary expression tree of `4*x*y^2`.

Then we add the tree `N01` to `N2345`.

```
N012345 = LabelledBinaryTree([N01, N2345], label='add')
ascii_art(N012345)
```

We then see the expression tree for `x^3 + 4*x*y^2` in Fig. 1.11.

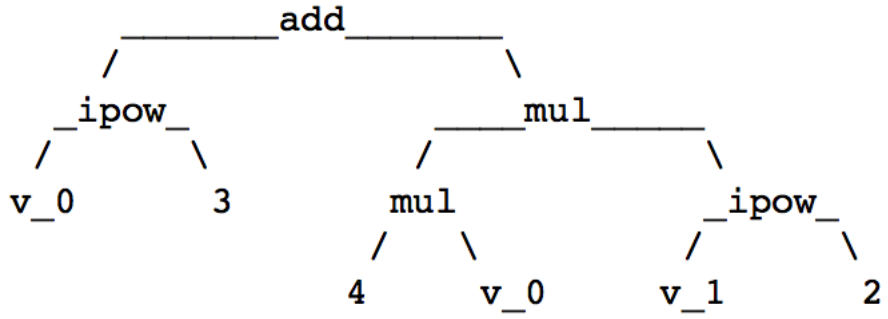

Fig. 1.11: The binary expression tree of `x^3 + 4*x*y^2`.

In the next step, we subtract `7*y`, as defined in node `N67`.

```
N01234567 = LabelledBinaryTree([N012345,N67], label='sub')
ascii_art(N01234567)
```

We then see the expression tree for `x^3 + 4*x*y^2 - 7*y` in Fig. 1.12.

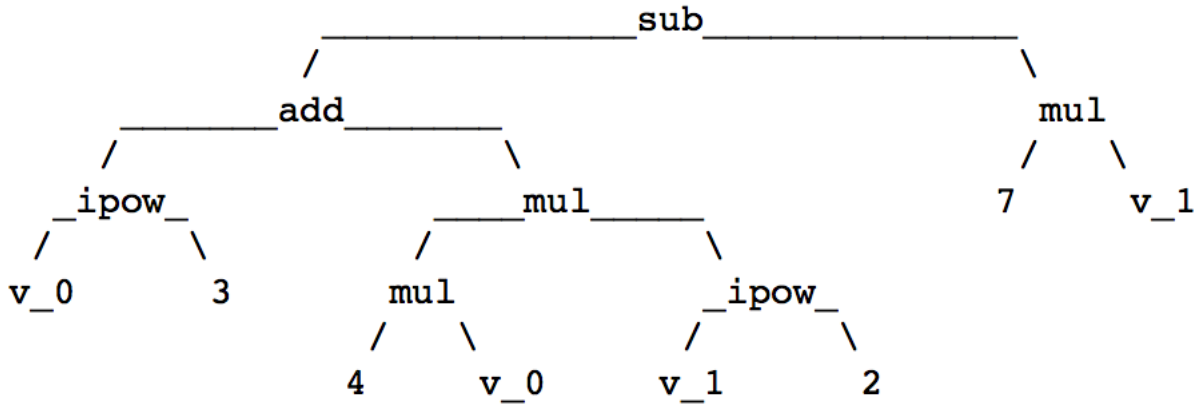

Fig. 1.12: The binary expression tree of `x^3 + 4*x*y^2 - 7*y`.

Finally, we add 2 to the tree, where 2 is defined by the last leaf `L8`.

```
lbt = LabelledBinaryTree([N01234567, L8], label='add')
ascii_art(lbt)
```

The final expression tree is shown in Fig. 1.13.

We can obtain a more lower-level representation of expressions:

```
f = fast_callable(p)
f.op_list()
```

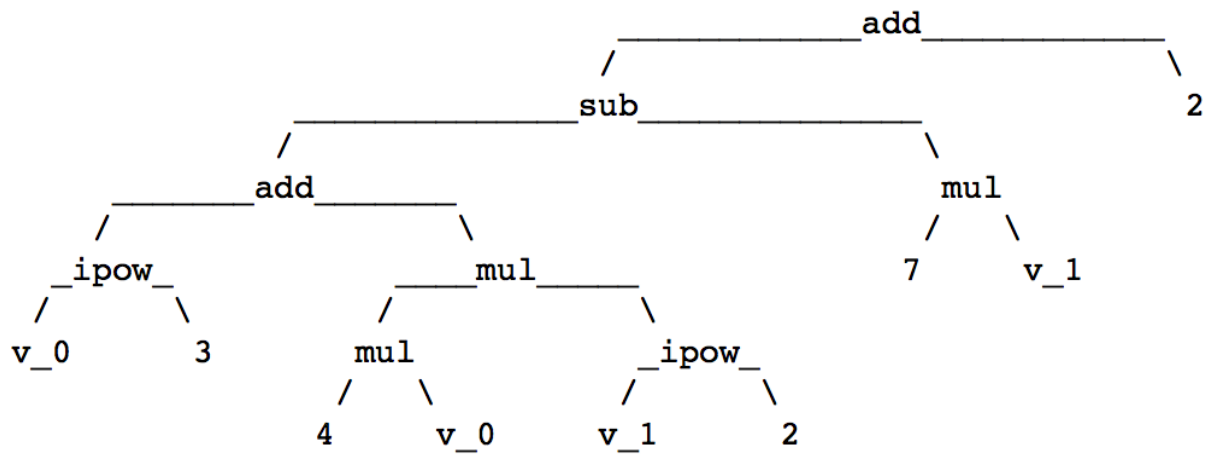

Fig. 1.13: The binary expression tree of `x^3 + 4*x*y^2 - 7*y + 2`.

The output is a list

```
[('load_arg', 0), ('ipow', 3), ('load_const', 4),
('load_arg', 0), 'mul', ('load_arg', 1), ('ipow', 2),
'mul', 'add', ('load_const', 7), ('load_arg', 1),
'mul', 'sub', ('load_const', 2), 'add', 'return'].
```

Observe that, if we view the list as a stack, we have a more explicit description to evaluate the expression. We first push the argument of an operand to the stack, the first argument `v_0` and then the operator `ipow` with operand `3`. The action of pushing an operator to the stack results in popping the needed operands from the stack, evaluating the operation, and then push the outcome of the operation to the stack, so that this outcome can then be used for the next operation.

1.8.3 Assignments

1. Consider computing modulo 5. This corresponds to computing with numbers in $Z_5 = \{0, 1, 2, 3, 4\}$. Print the multiplication table for this set of numbers. Explain how you can see from the multiplication table that every element (except for zero) has a multiplicative inverse. Make a loop to print for every nonzero element its multiplicative inverse.
2. Consider polynomials in x over the modulo 2 integers. List all polynomials of degree at most two and show that $x^2 + x + 1$ is the only polynomial of degree two or less that does not factor.
3. Let p be the polynomial $x^2y^3 - 3x^3 + 2y - 9$. Make a fast callable object for p and print this object. Use the output of the print of the fast callable object to draw the expression tree to evaluate p.
4. Consider the expression $2x^3y^5 - 6x^3 + y^8$. Turn this expression into an object of the class `ExpressionCall` and draw the expression tree with `LabelledBinaryTree()`.

1.9 Lecture 9: Input/Output Formats – Saving and Loading Data

In this lecture we will work in SageMath with persistent data, stored on file. We cover three ways to save data permanently on file. The most basic way uses plain Python files. While the conversion of a SageMath number to string is straightforward, we must be careful to run the preparse command before the application of eval. SageMath objects can be saved and loaded respectively with the save and load commands. The lecture ends with an illustration of pickle.

Data concerns not only numbers, but also includes source code definitions of functions which can be imported into a SageMath session.

1.9.1 Files in Python

At the most basic level, we can use files to store data permanently. As an example, we take a 20-digit approximation for π.

```
x = numerical_approx(pi,digits=20)
x
```

To save the number x refers to we convert it to a string and then write the number to a file in the `/tmp/` directory. The name of the file is `sagexnum.txt`.

```
sx = str(x)
file = open('sagexnum.txt','w')
file.write(sx)
file.close()
```

The file is written in the current directory. To change the current directory, use the `chdir` of the `os` module after importing it via `from os import chdir`. To see the listing of the files in the current directory, use `listdir` of the `os` module.

```
import os
os.listdir('.')
```

We clear everything to make sure `x` is gone.

```
reset()
x
```

Typint `x` then just shows the symbol `x` the reference of `x` to the 20-digit approximation of π is gone.

To retrieve the data, we will open the file for reading.

```
file = open('sagexnum.txt','r')
s = file.readline()
print(s, 'has type', type(s))
```

So far, we have just executed pure Python, and retrieved a string from file. The application of `eval` directly on `s` will give a `float` on return, which is not what we want, given that we stored a 20-digit number on file. Consider the following cell in SageMath.

```
x = preparse(s)
print(x, 'has type', type(x))
y = eval(x)
print(y, 'has type', type(y))
```

Commands in a SageMath cell are interpreted by a language which is Python – for almost all of the time. Each line of code runs automatically through a preparse before execution by the Python interpreter. To see how SageMath differs from Python, we use the `preparse` command. While `s` contains the string representation of the 20-digit floating-point number, that is: `3.1415926535897932385`, the content of the string returned by `preparse` is different. In particular, `x` contains `RealNumber('3.1415926535897932385')` and after `eval(x)` we get an object of type `sage.rings.real_mpfr.RealLiteral` with as content the number `3.14159265358979239`.

1.9.2 Saving and Loading SageMath Objects

The `save()` method and the `load()` function are much more convenient han working directly with Python files. In this section we continue with the `y` from the previous section. If the `y` is lost, just do `y = numerical_approx(pi, digits=20)`. To save a SageMath object, we apply the `save` method to the object. The argument we give to the `save` method is a file name.

```
y.save('sageynum')
```

The result of executing the `save` is that the current directory contains the file `sageynum.sobj`. We can check this asking for a directory listing.

```
import os
os.listdir('.')
```

We could execute `reset('y')` to remove the reference to `y` but we may also declare `y` as a variable.

```
y = var('y')
y
```

Declaring `y` as a variable removes the reference to the object it referred to.

To retrieve a SageMath object from file, we use the `load` function.

```
z = load('sageynum.sobj')
print(z, 'has type', type(z))
```

On return in `z` is `3.14159265358979239` an object of type `sage.rings.real_mpfr.RealNumber`.

While the `save()` and `load()` are perhaps the most convenient ways to work with persistent storage, note that those methods work only for SageMath objects. You cannot save for example a Python list with `save()`.

1.9.3 Pickling Objects

Python has a pickling mechanism which is also called serialization. In this section we continue with the `z` from the previous section. If the `z` is lost, just do `z = numerical_approx(pi, digits=20)`.

```
import pickle
s = pickle.dumps(z)
s
```

The pickled object is a string of a bytes.

```
"b'\\x80\\x03csage.rings.real_mpfr\\n__create__RealNumber_version0\\nq\\x00csage.rings.
↪real_mpfr\\n__create__RealField_version0\\nq\\x01KC\\x89X\\x04\\x00\\x00\\x00RNDNq\\
↪x02\\x87q\\x03Rq\\x04X\\x12\\x00\\x00\\x003.4gvml245kc4d80@0q\\x05K \\x87q\\x06Rq\\x07.
↪'"
```

Now we will write the string s to file, and then later we will delete the string and reset z.

```
file = open('sageznum.txt', 'w')
file.write(s)
file.close()
```

We delete the string s and reset z.

```
del(s)
reset('z')
print(z, s)
```

Printing z results in a NameError.

Now we open the file, read the string from file and then load. We read all lines from file with the method readlines of a file object.

```
file = open('sageznum.txt', 'r')
lines = file.readlines()
lines
```

We join all the elements in lines into one string s.

```
s = ''.join(lines); print(s)
```

The output of print(s) shows the pickled representation of the real number. The type of s is a string and s itself is the string representation of a bytes object. Before we can reconstruct the number z from the pickled object, we evaluate the string to the bytes object.

```
bytelines = eval(s)
print(bytelines, 'has type', type(bytelines))
```

Now we can reconstruct z from the pickled object.

```
z = pickle.loads(bytelines)
print(z, 'has type', type(z))
```

And then we see that z once again is an object of type sage.rings.real_mpfr.RealNumber with value 3.141592653589793239.

1.9.4 Assignments

1. Take a floating-point approximation of $\sqrt{2}$ with 30 decimal places. Assign this approximation to a variable, convert the value to a string, use Python to write the string object to a file, and close the file. Reset the variable that referred to the approximation. Open the same file again with Python, read the string from file, and convert the string into the same SageMath object that was stored. Verify that the value and type of the retrieved object is the same as the original object that was written to file.
2. Take a floating-point approximation of $\sqrt{2}$ with 30 decimal places. Assign this approximation to a variable and use the save command of SageMath to store this approximation to a file. Reset the variable that referred to the approximation. Use the load command of SageMath to retrieve the approxmation from file, verify that the value and type of the retrieved object is the same as the original object that was saved to file.
3. Take a floating-point approximation of $\sqrt{2}$ with 30 decimal places. Assign this approximation to a variable and write the pickled object to file. Reset the variable that referred to the approximation. Read the pickled object

from file and reconstruct the SageMath object. Verify that the value and type of the retrieved object is the same as the original object that was stored to file.

4. We have covered three ways to store a SageMath object to a file. For each of the three ways, list one advantage and one disadvantage. Why and in which circumstances would you prefer one way over the other?

5. Do `x = '3/4'`. Explain the difference between `eval(x)` and `eval(preparse(x))`.

1.10 Lecture 10: Speeding up Python Functions with Vectorization and Cython

This lecture follows Chapter 3 of *Sage for Power Users* by William Stein.

As an interpreted scripting language, Python is slow compared to compiled languages such as C. With Cython the compiled libraries available in Sage can be used. We may view Cython as a compiled variant of Python.

The execution efficiency of Python is often very slow because variables are interpreted as expressions and the default number types are often not the data types supported by fast hardware arithmetic, but in slower arithmetic implemented by software. Cython is a variant of Python that allows to add type declarations, so we may build much faster functions without having to change the logic of the original script. The running example is the application of basic numerical integration to approximate π. The fastest version applies numpy vectorization, although this approach requires a reformulation of the original script.

1.10.1 A Motivating Example

We consider the computation of a floating-point approximation of a sum. Such sums occur in numerical integration.

One way to approximate $\pi/4$ is to compute the area of the unit circle defined by $x^2 + y^2 - 1 = 0$, for x ranging from 0 to 1. The corresponding y coordinate of a point on the unit circle is then $\sqrt{1 - x^2}$.

$$\frac{\pi}{4} = \int_0^1 \sqrt{1 - x^2} \approx \frac{1}{n} \sum_{k=1}^{n} \sqrt{1 - \left(\frac{k}{n}\right)^2}$$

In Python, the formula translates into a one line of code, defined in the function `python_sum_symbolic`.

```
def python_sum_symbolic(n):
    return float( sum(sqrt(1-(k/n)^2) for k in range(1, n+1)) )/n
4*python_sum_symbolic(1000)
```

We see `3.1395554669110264` as an approximation for π.

To benchmark the function, we can use `timeit`. The command `timeit` in Python is good to measure the execution time of small code snippets.

```
timeit('python_sum_symbolic(1000)')
```

The output on an 3.1 GHz Intel Core i7 processor is `5 loops, best of 3:  84.8 ms per loop`. For longer execution times, we better use cputime().

```
t1 = cputime()
python_sum_symbolic(10^4)
ct1 = cputime(t1)
print('time for python_sum_symbolic :', ct1)
```

and we see `time for python_sum_symbolic :  2.694785`.

1.10.2 Executing in Pure Python

The first reason that the function is so slow is because we use the symbolic `sqrt` function. We will use the sqrt function of the `math` module in Python. The explicit conversion to float is no longer needed as this `sqrt` returns afloating-point number.

```
def python_sum(n):
    from math import sqrt
    return sum( sqrt(1-(k/n)^2) for k in range(1, n+1) )/n
4*python_sum(1000)
```

Now we time the function again with `time`.

```
t2 = cputime()
python_sum(10^4)
ct2 = cputime(t2)
print('time for python_sum :', ct2)
```

We obtain `time for python_sum :  0.029598` and because this has now become a small computation, we may as well use `timeit`.

```
timeit('python_sum(10^4)')
```

which confirms with `25 loops, best of 3:  23.3 ms per loop` that that time has dropped from seconds to milliseconds.

```
print('speedup :', ct1/ct2)
```

shows `speedup :  91.0461855531`.

1.10.3 Vectorization with numpy

Instead of the `sqrt` of pure Python, we could also use the `sqrt` and `sum` of numpy. These numpy functions allow that we give arrays on input.

The *vectorization* of code is the replacement of a Python for loop in `for k in range(1, n+1)` by a loop executed by numpy functions.

```
def numpy_sum(n):
    from numpy import sqrt, sum, arange
    x = arange(n)/float(n)
    return sum(sqrt(1-x**2))/n
4*numpy_sum(1000)
```

Applying `timeit` to benchmark the function.

```
timeit('numpy_sum(10^4)')
```

shows `625 loops, best of 3:  108 µs per loop` so the time is now expressed in *microseconds* instead of milliseconds.

To compare against the pure Python sum, we sum one million items.

```
t3 = cputime()
python_sum(10^6)
ct3 = cputime(t3)
print('time for python_sum :', ct3)
t4 = cputime()
numpy_sum(10^6)
ct4 = cputime(t4)
print('time for numpy_sum :', ct4)
```

From the Python code we obtain `time for python_sum :  2.696452` and with numpy we get `time for numpy_sum :  0.019973`. Let us compute the speedup.

```
print('speedup :', ct3/ct4)
```

which shows `speedup :  135.004856556`. With numpy vectorization we again obtained a hundredfold improvement.

1.10.4 Cython code

The advantage of Cython over vectorization is that the Cython code is almost identical as the Python code. In this particular example we had to replace the ^ by the ** for the exponentiation.

```
%%cython
def cython_sum(n):
    from math import sqrt
    return sum( sqrt(1-(k/n)**2) for k in range(1, n+1) )/n
```

The two links returned when evaluating a cell with Cython code are the generated C code and an html file with the annotated version of the Cython program. To see whether it works we do.

```
print(4*cython_sum(1000))
```

and then we benchmark the code with `timeit`.

```
timeit('cython_sum(10^4)')
```

We obtain `25 loops, best of 3:  26.2 ms per loop`. Running a larger example with `time`

```
t5 = cputime()
cython_sum(10^6)
ct5 = cputime(t5)
print('time for the cython_sum :', ct5)
```

shows `time for the cython_sum :  2.651237` so the Cython code is not really faster than the original Python code, because in both functions `python_sum` and `cython_sum` the same functions in the Python C library are executed.

The code can be made faster if we declare the variables to be of C data types and the we use the C version of the sqrt function.

```
%%cython
cdef extern from "math.h":
    double sqrt(double)
```

(continues on next page)

(continued from previous page)

```
def cython_sum_typed(long n):
    cdef long k
    return sum( sqrt(1-(k/float(n))**2) for k in range(1, n+1) )/n
```

We first check the correctness.

```
print(4*cython_sum_typed(1000))
```

and now we will see improved timing results.

```
timeit('cython_sum_typed(10^4)')
```

With timeit we obtain `625 loops, best of 3:  1.09 ms per loop` and then we run `time` again.

```
t6 = cputime()
cython_sum_typed(10^6)
ct6 = cputime(t6)
print('time for cython_sum_typed :', ct6)
```

The output is `time for cython_sum_typed :  0.112308` and compared to the first Cython code, we compute the speedup.

```
print('speedup :', ct5/ct6)
```

which prints `speedup :  23.6068401182`.

1.10.5 Assignments

1. Perform all computations on your computer. Make a table with execution times and speedups.
2. Perform all computations on your Sage notebook account or on SageMathCloud. Indicate which online version you ran. Make a table with execution times and speedups.
3. Write a Python function `python_sum` which takes on input a positive integer n and which returns the floating-point value of

$$\frac{1}{n}\sum_{k=1}^{n} \ln\left(1+\frac{k}{n}\right).$$

   Write a more efficient version `numpy_sum` using vectorization.

   Time the two versions to illustrate the efficiency.
4. Take the Python function `python_sum` of the previous exercise and apply Cython to make the function more efficient. Time your Cython version and compare with `python_sum` to illustrate the efficiency.

Manipulating expressions is one of the core tasks of computer algebra. The goal of this part is to introduce the main tools and concepts to use SageMath to manipulate expressions.

1.11 Lecture 11: Univariate and Multivariate Polynomials

There may not be much difference at the low level of expressions where we store parameters as ordinary variables, at the mathematical level polynomials in several variables are different from polynomials in one variable that have symbols as coefficients. In this lecture we make the important connection between factoring and root finding, symbolically as well as numerically.

1.11.1 Polynomials as Expressions

Let us start with a definition in plain words. A univariate polynomial is a finite sum of terms, where every term is a coefficient multiplied with a monomial, and where every monomial is a power of the variable, by default, call this variable `x`. All coefficients in the polynomial are of the same type.

```
p = x^4 - 4*x^2 - 7*x + 9
print(p, 'has type', type(p))
print('the degree :', p.degree(x))
```

The degree of an expression returns the highest power of the variable given as an argument of the degree method applied to the expression. Can we also select the coefficients of the expression?

```
print('the coefficient of x^2 :', p.coefficient(x,2))
print('all coefficients :', [p.coefficient(x,k) for k in range(p.degree(x)+1)])
print('the leading coefficient :', p.leading_coefficient(x))
```

If we select all coefficients of `p`, then we get a list of lists. Each list gives the coefficient and the corresponding power.

```
print('all coefficients :', p.coefficients())
print('all coefficients in x :', p.coefficients(x))
```

If we want to see the terms of the polynomial symbolically, then we can ask for the operands.

```
print('the operands of p :', p.operands())
print('the topmost operator of p :', p.operator())
```

We see that we may view the expression p as a sum. If we ask for the roots, then see what happens!

```
print(p.roots())
```

We see the roots expressed with the `sqrt()` function. If we are not happy with this output, then we have to change the ring.

```
print('the real roots :', p.roots(ring=RR))
print('the complex roots :', p.roots(ring=CC))
```

Our polynomial has two real roots and two complex conjugated roots.

1.11.2 Univariate Polynomials

We can explicitly declare a polynomial ring and coerce our expression for p into this ring.

```
P.<x> = PolynomialRing(QQ)
q = P(p)
print(q, 'has type', type(q))
```

Another, shorter, and perhaps more natural way of writing this conversion is

```
R = QQ[x]
r = R(p)
print(r, 'has type', type(r))
```

If we were no longer interested in our polynomial q, then we could just as well pick any random quartic polynomial.

```
print(QQ[x].random_element(degree=4))
```

But let us continue with our q. If we compute the degree and coefficients, then we may not give the argument x anymore.

```
print(q.degree())
print('coefficients of nonzero terms :', q.coefficients())
print('all coefficients :', q.coeffs())
print('the list of all coefficients :', q.list())
print('a dictionary representation :', q.dict())
```

The keys of the dictionary are the exponents of those monomials that appear with nonzero coefficient. The dictionary representation could be very convenient for *sparse* polynomials, when only relatively few monomials appear with nonzero coefficient, relative to the degree of the polynomial.

We can ask if the polynomial is irreducible.

```
print(q.is_irreducible())
print(factor(q))
```

and build a field extension to add a root of q to QQ

```
K.<a> = QQ.extension(q)
print(K)
```

Now we can view the original expression as a polynomial over the new field, extended with a root of q.

```
q2 = K[x](q)
print(q2, 'has type', type(q2))
```

See if q2 factors or not.

```
print(q2.is_irreducible())
print(factor(q2))
```

We see that adding one algebraic number is not enough to factor completely in linear factors. Observe that we must explicitly coerce the nonlinear factor into a polynomial with coefficients in K.

```
f2 = q2/(x-a); print(f2, type(f2), type(K[x](f2)))
L.<b> = K.extension(K[x](f2))
print(L)
```

We take the original polynomial in the new extended coefficient ring and factor.

```
q3 = L[x](q)
print(factor(q3))
```

To select the factor to continue with our field extensions, we convert the factorization to a list.

```
lf = list(factor(q3))
print(lf)
```

Note that the elements in the list are tuples, so we must remove the multiplicities.

```
f3t = lf[2]; print(f3t, 'has type', type(f3t))
f3 = L[x](f3t[0]); print(f3, 'has type', type(f3))
```

Now make our last field extension!

```
M.<c> = L.extension(f3)
print(M)
```

As the polynomial now factors completely, we have solved the polynomial symbolically, expressing the roots as algebraic numbers a, b, and c.

```
q4 = M[x](q)
print(factor(q4))
```

The polynomial `q` is now factored completely as a product of linear factors, as we see `(x - c) * (x - b) * (x - a) * (x + c + b + a)`. Observe the connection between factoring and root finding.

```
print(q4.roots())
```

and we see `[(c, 1), (b, 1), (a, 1), (-c - b - a, 1)]`. We can still find the roots numerically of our quartic.

```
crts = q4.roots(ring=CC)
print(crts)
```

Because the coefficient in the term with x^3 is zero, the sum of the roots in both the symbolic and the numeric representation must be zero. We can see this easily from the symbolic representation, but let us verify this on the numerical representation.

```
print(sum(n for (n,1) in crts))
```

To recapitulate, we distinguish between two main forms of root finding, one is symbolic, the other numeric. The default numeric field is the field of complex numbers, whereas symbolically we extend the field of rational numbers with sufficiently many symbols to represent all roots.

1.11.3 Multivariate Polynomials

Polynomials in several variables are declared similarly as polynomials in one variable. The quotes around the names are needed if we have not used or declared them explicitly before as variables. We take a random polynomial of degree 4, and with terms we can give an upper bound on the number of terms in the polynomial.

```
R.<x,y> = PolynomialRing(QQ)
p = R.random_element(degree=4,terms=10)
print(p, 'has type', type(p))
```

We can select the monomials, get its variables, and its degrees.

```
print('the monomials :', p.monomials())
print('corresponding coefficients :', p.coefficients())
print('the variables :', p.variables())
print('the degree in x :', p.degree(x))
print('the degree in y :', p.degree(y))
print('the degree of p :', p.degree())
```

The order of the monomials is important.

```
print(R.term_order())
```

The default order appears to be `Degree reverse lexicographic term order`. To change the ordering of the monomials in the polynomial, we coerce p into a another ring. In a lexicographic order, all monomials in which `x` occurs come first.

```
Rlex.<x,y> = PolynomialRing(QQ,order = 'lex')
print(Rlex); print(Rlex.term_order())
print(Rlex(p))
```

We can view a polynomial in several variables as a polynomial in one variable by collecting terms. Because of the type of argument of `polynomial()`, the selection of the tuple of the outcome of `p.variables()` is needed.

```
print('as polynomial in x :', p.polynomial(p.variables()[0]))
print('as polynomial in y :', p.polynomial(p.variables()[1]))
```

1.11.4 Assignments

1. Execute the following sequence of commands: `var('x'); p = prod([x-k for k in range(20)]); print p; q = p.expand()` and then type `print q.roots()` and `print q.roots(ring=CDF)`. Compare the differences between the output of the two `roots`. Did you expect to see those differences?
2. Declare `x` as a polynomial variable over the rational number with the statement `x = polygen(QQ)`.
   1. Give the Sage command(s) to compute the greatest common divisor of the polynomials $p = 2x^5 + 11x^4 + 14x^3 + 11x^2 + 12x$ and $q = 2x^5 + 5x^4 + 7x^3 + 8x^2 + 5x + 3$.
   2. How can Sage compute the cofactors k and ℓ so that $\gcd(p, q) = kp + \ell q$?
   3. Finally, give the Sage commands to verify the relation $\gcd(p, q) = kp + \ell q$ for the k and the ℓ that were found.

   Hint: do `help(xgcd)`.
3. Do `x = polygen(RR); print factor(x^2 - 2.25)` and `x = polygen(QQ); print factor(x^2 - 9/4)`. Explain the differences in the outcomes of the two factor commands.

4. Consider the polynomial $p = 2x^5 + 9x^4 + 16x^3 + 15x^2 + 12x + 9$. Write p as a product of linear factors:
   1. symbolically, by adding sufficiently many formal roots; and
   2. numerically, by finding all complex roots of p.
5. Consider the polynomial $p = 2xz^4 + xz^3 + 2yz^2$. Give the Sage commands to bring p in the forms
   (a) $2z^4x + z^3x + 2z^2y$,
   (b) $2xz^4 + xz^3 + 2z^2y$, and
   (c) $2yz^2 + 2xz^4 + xz^3$.
6. Consider the polynomial $p = x^3 + 4x + 7$ over a finite field of 17 elements.
   (a) Give the Sage commands to show that p is irreducible over this finite field.
   (b) Add sufficiently many formal roots to this finite field so that p factors as a product of linear polynomials.
   (c) Give all relevant Sage commands. Write the final factorization of p.
7. Consider the statements `var('x')` and `QQ['x']`.

   What is the main difference between the roles of `'x'` after these statements? Start your answer with a precise description on the effect of each statement. Illustrate the difference.

   When should you use `var('x')`, when `QQ['x']`?

   What can you do after `QQ['x']` but not after `var('x')`?

1.12 Lecture 12: Rational Functions and Conversions

One of the main problems in computer algebra is expression swell. In an exact calculation, *expression swell* happens when the numbers and or expressions grow exponentially in size.

A rational number is simplified immediately, but in rational expressions removing the greatest common denominator may not always result in a smaller expression and may even lead to expression swell. If we consider $\frac{x^d-1}{x-1}$, for any degree d then the reason becomes clear.

1.12.1 Rational Expressions

SageMath recognizes rational expressions as of type `fraction_field_elements` of `sage.rings`.

```
x = polygen(QQ)
p = x^3 - 1; q = x^2 - 1; r = p/q
print(r, 'has type', type(r))
```

What is remarkable is that the rational expression is normalized automatically. The normalization of a rational expression is the removal of the greatest common divisor of numerator and denominator. We see that the common factor `x - 1` is removed in r.

```
print(factor(p)); print(factor(q))
```

This automatic normalization may lead to expression swell.

```
f = (x^1000 - 1)/(x-1); print(f)
```

We see an expression with 1,000 terms! Instead of printing the entire expression, we may just as well only ask for just the number of terms. The problem is that `f` is still not a polynomial and we have to take its fraction before getting the number of coefficients.

```
print(type(f))
fn = f.numerator()
print(type(fn))
print(len(fn.coefficients()))
```

We can freeze an expression, by conversion to a Symbolic Ring (SR).

```
g = SR(p)/q; print(g);
```

This shows the original expression `(x^3 - 1)/(x^2 - 1)`. We can ask for the denominator, with or without normalization

```
print(g.denominator(normalize=False))
print(g.denominator(normalize=True))
```

The same for the numerator, with or without normalization

```
print(g.numerator(normalize=False))
print(g.numerator(normalize=True))
```

We may represent a rational expression with numerator and denominator in factored form, or not.

```
fn = g.numerator(normalize=False).factor()
fd = g.denominator(normalize=False).factor()
print(SR(fn)/g.denominator(normalize=False))
print(g.numerator(normalize=False)/SR(fd))
```

This prints `(x^2 + x + 1)*(x - 1)/(x^2 - 1)` and `(x^3 - 1)/((x + 1)*(x - 1))`.

Factoring or expanding the numerator, factoring or expanding the denominator defines four normal forms of rational expressions. If we really do not want to remove common factors, then we must explicitly convert to strings.

```
print(str(fn) + '/' + '(' + str(fd) + ')')
```

which shows `(x^2 + x + 1)*(x - 1)/((x + 1)*(x - 1))`.

1.12.2 Conversions

The coercion to a symbolic ring SR on a polynomial shows the so-called Horner form of the polynomial.

```
x = SR.var('x')
p = x^4 + 2*x^3 + 3*x^2 + 4*x + 5
print(p.horner(x))
```

The Horner form gives an efficient way to evaluate a polynomial, which requires as many multiplications as additions, for the example polynomial `p`, the Horner form is `(((x+2)*x + 3)*x + 4)*x + 5`.

The disadvantage of working with x = polygen(QQ) is that we cannot select a random element from the ring.

```
P.<x> = PolynomialRing(CC)
rn = P.random_element(degree = 4)
rd = P.random_element(degree = 4)
f = rn/rd
print(f, 'has type', type(f))
```

To evaluate a rational expression efficiently, we may consider a partial fraction decomposition.

```
print(f.partial_fraction_decomposition())
```

1.12.3 Assignments

1. Assign to `r` the expression `(4*x^2 + x)/(x^2 + 4)`. Use the methods `operands()` and `operator()` to draw the expression tree of `r`.
2. Consider the rational expression `p = (x^4 + x^3 - 4*x^2-4*x)/(x^4 + x^3 - x^2 - x)`. What are the commands to transform `p` into `(x + 2)*(x + 1)*(x - 2)/(x^3 + x^2 - x - 1)`?
3. What is the partial fraction decomposition of `p = (x^4 + x^3 - 4*x^2-4*x)/(x^4 + x^3 - x^2 - x)`?
4. Consider the polynomial p in x with rational coefficients:

$$p = \frac{1}{12}x^8 + \frac{93}{4}x^7 - x^6 + 2x^5 + \frac{5}{39}x^4 + \frac{1}{5}x^3 + x^2 - \frac{1}{5}x + \frac{1}{2}.$$

   Without retyping p, convert p into the polynomial

$$q = \frac{1}{2}x^8 - \frac{1}{5}x^7 + x^6 + \frac{1}{5}x^5 + \frac{5}{39}x^4 + 2x^3 - x^2 + \frac{93}{4}x + \frac{1}{12}.$$

   which has the same coefficients as p, but swapped. If c_k is the coefficient with x^k in p, then in q, the monomial x^{8-k} has coefficient c_k, for $k = 0, 1, \ldots, 8$.

1.13 Lecture 13: Representation of Expressions

We have already covered expression trees derived from fast callable objects, but a fast callable object is just one representation that is geared for evaluation in hardware arithmetic. In this lecture we examine how expressions in Sage are stored internally.

1.13.1 Expression Trees

Let us consider again the structure of expressions. We generate a random polynomial with rational coefficients.

```
R.<x,y,z> = PolynomialRing(QQ)
set_random_seed(2018)
p = R.random_element(terms=4, degree=8)
p
```

Note that we fixed the seed of the random number generator, so we will always see the same polynomial printed.

```
x*y^2*z^4 + 1/5*x^5*y + 2*x*y^2*z^2 + x^3
```

Unfortunately, the methods `operands()` and `operator()` do not work on a multivariate polynomial of this type. We convert `p` to a general SageMath expression by evaluation in new symbolic variables. Because we will not use the polynomial ring `R` anymore, we choose the same letters `x`, `y`, and `z` for the new variables.

```
x, y, z = var('x,y,z')
sq = p(x=x,y=y,z=z)
print(sq, 'has type', type(sq))
```

Then we see that `sq` has the same value as `p`, but is of type `sage.symbolic.expression.Expression`. We can then ask to see the operands and the operators.

```
print(sq.operands())
print(sq.operator())
```

What is printed is `[x*y^2*z^4, 1/5*x^5*y, 2*x*y^2*z^2, x^3]` and `<function add_vararg at 0x1192011b8>`. Observe that the list of operands has four elements and that the operator is `add_vararg`, an addition with a variable number of arguments. Therefore: *expression trees of general Sage expressions are NOT binary*.

We start drawing the expression tree in a top down fashion.

```
oplabels = [str(op) for op in sq.operands()]
operands = [LabelledOrderedTree([], label=op) for op in oplabels]
exptree = LabelledOrderedTree(operands, label='+')
ascii_art(exptree)
```

The top level of the expression tree is shown in Fig. 1.14.

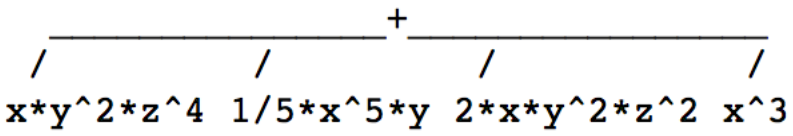

Fig. 1.14: The top level of the expression tree of a multivariate polynomial.

We can now apply the same operations to the leaves.

```
oplabels = [op.operands() for op in sq.operands()]
operands = [op.operator() for op in sq.operands()]
print(oplabels)
print(operands)
```

All the operators are multiplication, except for the last operator. Because the operands will be used as labels in the new expression tree, we convert to strings.

```
stroperands = [[str(x) for x in L] for L in oplabels]
print(stroperands)
```

and we obtain a list of lists:

```
[['x', 'y^2', 'z^4'], ['x^5', 'y', '1/5'], ['x', 'y^2', 'z^2', '2'], ['x', '3']]
```

With the list of lists of strings, we make a list of leaves and use the leaves as the internal nodes. The last leaf is dealt with separately.

```
leaves = [[LabelledOrderedTree([], label=x) for x in op] for op in stroperands]
nodes = [LabelledOrderedTree(leaf, label='*') for leaf in leaves[0:3]]
node3 = LabelledOrderedTree(leaves[3], label='^')
nodes.append(node3)
for node in nodes:
    print(ascii_art(node))
```

The four nodes are shown in Fig. 1.15.

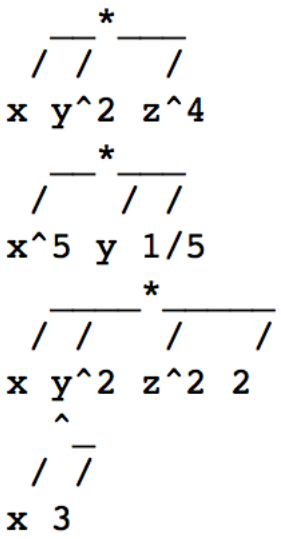

Fig. 1.15: The four nodes at the top level elaborated in the expression tree.

Now we can redefine the expression tree.

```
exptree = LabelledOrderedTree(nodes, label='+')
ascii_art(exptree)
```

The redefined expression three is shown in Fig. 1.16.

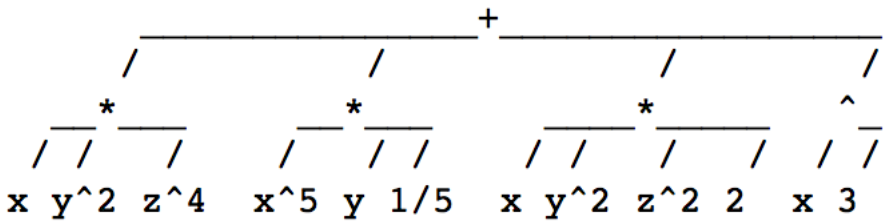

Fig. 1.16: The redefined expression tree with four nodes.

We have five leaves left to elaborate, which are all powers of variables: `y^2`, `z^4`, `x^5`, `y^2`, `z^2`.

```
leafx = LabelledOrderedTree([], label='x')
leafy = LabelledOrderedTree([], label='y')
leafz = LabelledOrderedTree([], label='z')
leafpow2 = LabelledOrderedTree([], label='2')
leafpow4 = LabelledOrderedTree([], label='4')
leafpow5 = LabelledOrderedTree([], label='5')
```

Now we can define the nodes which represent the powers.

```
nodeypow2 = LabelledOrderedTree([leafy, leafpow2], label='^')
ascii_art(nodeypow2)
```

The other nodes are defined similarly.

```
nodexpow5 = LabelledOrderedTree([leafx, leafpow5], label='^')
nodezpow2 = LabelledOrderedTree([leafz, leafpow2], label='^')
nodezpow4 = LabelledOrderedTree([leafz, leafpow4], label='^')
```

Apparently there is no way to substitute nodes in a tree. We will manually replace the appropriate entries in the list of leaves and redefine the expression tree.

```
print(leaves)
```

The print statement shows

```
[[x[], y^2[], z^4[]], [x^5[], y[], 1/5[]], [x[], y^2[], z^2[], 2[]], [x[], 3[]]]
```

Now we redefine the leaves as follows.

```
newleaves = [[leaves[0][0], nodeypow2, nodezpow4], \
             [nodexpow5, leaves[1][1], leaves[1][2]], \
             [leaves[2][0], nodeypow2, nodezpow2], leaves[3]]
print(newleaves)
```

and then redefine to obtain the complete expression tree.

```
nodes = [LabelledOrderedTree(leaf, label='*') for leaf in newleaves[0:3]]
node3 = LabelledOrderedTree(leaves[3], label='^')
nodes.append(node3)
exptree = LabelledOrderedTree(nodes, label='+')
ascii_art(exptree)
```

The complete expression tree is shown in Fig. 1.17.

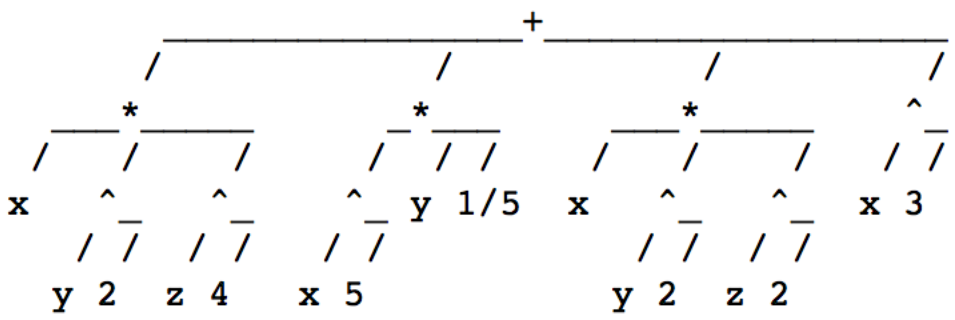

Fig. 1.17: The complete expression tree of a multivariate polynomial.

1.13.2 Evaluation of Expressions

The form of the expression matters when it comes to evaluation. For fast evaluation, we convert to a fast callable object.

```
f = fast_callable(sq, vars=['x','y','z'])
```

Observe the difference in the ways to evaluate:

- with `f` we do not use the variable names as key words,
- with `p` we do use the variable names as key words.

```
print(f(1.0,2.0,3.0))
print(p(x=1.0, y=2.0, z= 3.0))
```

Both forms of the expression give the same value `397.400000000000`. To time the evaluation, we use `timeit`.

```
timeit('f(1.0,2.0,3.0)')
```

We obtain as output `625 loops, best of 3:  11.6 µs per loop`.

```
timeit('p(x=1.0,y=2.0,z=3.0)')
```

yields `625 loops, best of 3:  92.9 µs per loop`. Even already on a such a small example, the fast callable object is much more efficient.

To see the internal structure of the fast callable object `f`, we apply the method `op_list()` to it.

```
f.op_list()
```

and we see the list of low level instructions which can be interpreted as a stack.

```
[('load_arg', 0),
 ('load_arg', 1),
 ('ipow', 2),
 'mul',
 ('load_arg', 2),
 ('ipow', 4),
 'mul',
 ('load_arg', 0),
 ('ipow', 5),
```

(continues on next page)

(continued from previous page)

```
('load_arg', 1),
'mul',
('load_const', 1/5),
'mul',
'add',
('load_arg', 0),
('load_arg', 1),
('ipow', 2),
'mul',
('load_arg', 2),
('ipow', 2),
'mul',
('load_const', 2),
'mul',
'add',
('load_arg', 0),
('ipow', 3),
'add',
'return']
```

The data structure which represents a fast callable object corresponds to a *binary* tree. In a binary tree, every internal node (that is not a leaf) has exactly two children. One reason why fast callable objects are *fast* is because they are limited to numerical values. The elementary numerical operations such as addition, multiplication, exponentiation are all binary operations and therefore, the expression tree defined by a fast callable object is a binary tree.

1.13.3 Assignments

1. Consider `p = 3*x^4 - 6*x^3 + 5*x^2 + 9*x - 7`. Draw the expression tree for `p`. Also give all Sage commands with their output used to make your drawing.
2. Consider `p = 3*x^4 - 6*x^3 + 5*x^2 + 9*x - 7`. Compute the Horner form `q` for `p` and draw the expression tree for `q`. Also give all Sage commands with their output used to make your drawing.
3. Consider `p = 3*x^4 - 6*x^3 + 5*x^2 + 9*x - 7` and its evaluation at `math.pi`. Compare with `timeit` the efficiency of the original expression, the Horner `q` form of `p`, and the fast callable objects of `p` and `q`.
4. Consider the expression:

$$x + \frac{y}{x^2 - y + 1}.$$

   Draw the internal representation of this expression.

1.14 Lecture 14: Substitution, Expansion, and Factorization

Substitution is one of the fundamental tools to work with expressions. Expansion is the counterpart of factorization.

1.14.1 Substitution

As a first application of substitution, we can replace an expression by a new variable to prevent expansion when we manipulate expression. Substitution is normally executed in sequence, but expressions are callable objects and calling an expression by keyword arguments performs the substitution simultaneously, as required when we want to permute the variables in an expression. Knowing the list of operands in an expression is useful. With string manipulations we can perform a pure syntactical substitution.

Suppose we would like to rewrite $(x+y)^2 + \frac{1}{(x+y)^2}$ into $\frac{(x+y)^4+1}{(x+y)^2}$.

We consider then an expression in `x` and `y` that we want to bring on the same denominator.

```
var('x,y')
p = (x+y)^2 + 1/(x+y)^2
print(p, 'has type', type(p))
```

If we want to bring the expression on a common denominator, then we could try the `factor()` method.

```
p.factor()
```

The output is `(x^4 + 4*x^3*y + 6*x^2*y^2 + 4*x*y^3 + y^4 + 1)/(x + y)^2`. The `factor` expands the numerator, which is not what we want. To shield `(x+y)^2` from expanding, we will save `x+y` into a new variable `z`. Then we will factor the expression in `z`. After this, we substitute `z` back to `x + y`.

```
var('z')
q = p.subs({(x+y): z})
print('after substitution of x+y into z :', q)
fq = q.factor()
print('after factoring :', fq)
qq = fq.subs(z=x+y)
print('after substitution of z into x+y :', qq)
```

And we see printed as output:

```
after substitution of x+y into z : z^2 + 1/z^2
after factoring : (z^4 + 1)/z^2
after substitution of z into x+y : ((x + y)^4 + 1)/(x + y)^2
```

To view the expression nicely typeset, we do

```
qq.show()
```

and then we see $\frac{(x+y)^4+1}{(x+y)^2}$.

Observe that even when we submit a dictionary as argument of the `subs()` method, the substitution happens in sequence and not simultaneously. Suppose we want to permute the variables in an expression. For example, we want to replace `a` by `b`, `b` by `c`, and `c` by `a`. Then the expression `a + 2*b + 3*c` turns into `b + 2*c + 3*a` via a simultaneous substitution, that is: if the substitution is executed simultaneously.

The simultaneous substitution of the variables in an expression is illustrated in Fig. 1.18.

The sequential substitution of the variables in an expression is illustrated in Fig. 1.19.

Expressions are callable objects and when we evaluate by keyword argument, then the substitution is executed simultaneously.

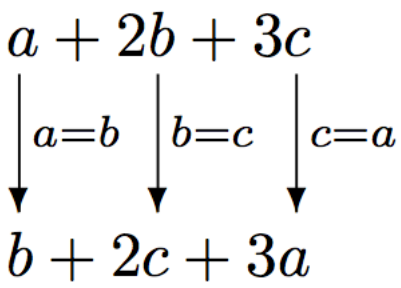

Fig. 1.18: Simultaneous substitution of the variables in an expression as done by `e(a=b, b=c, c=a)` for `e = a + 2*b + 3*c`.

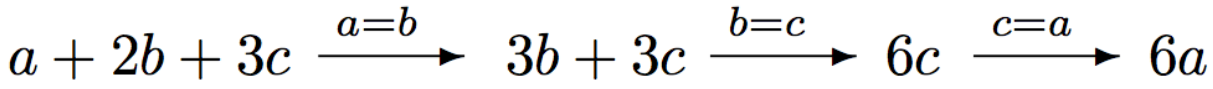

Fig. 1.19: Sequential substitution of the variables in an expression, as done by `e.subs(a=b).subs(b=c).subs(c=a)` for `e = a + 2*b + 3*c`.

```
var('a,b,c')
e = a + 2*b + 3*c
print('the expression :', e)
print('substitution with dictionary argument :', e.subs({a:b, b:c, a:c}))
print('evaluation by keyword arguments :', e(a=b,b=c,c=a))
```

The result of `e.subs({a:b, b:c, a:c})` turns out to be the same as `e(a=b, b=c, c=a)` as those statements both perform the substitution simultaneously.

Returning to the first application of substitution on `p`, if we had tried to replace `(x+y)^2` by `z`, then we would have noticed that only the numerator changed. Looking at the operands of `p` explains why this is.

```
print(p)
print(p.operands())
```

and we see that the list of operands is `[(x + y)^2, (x + y)^(-2)]`.

With string manipulation, we can perform a pure syntactical substitution.

```
s = str(p)
print(s)
t = s.replace('(x + y)^2', 'z')
print(t)
```

which then shows the *string* `z + 1/z`.

Knowing the list of operands, we can substitute `(x+y)^2` by `z^2` and `(x+y)^(-2)` by `z^(-1)`.

```
p.subs({(x+y)^2:z, (x+y)^(-2):z^(-1)})
```

which will show the *expression* `z + 1/z`.

1.14.2 Expansion

When we give an expression in factored form, such as $(a+b+c)(x^3+9x+8)$ then we see that Sage does not expand the expression automatically. Why not? The main reason is expression swell.

```
var('a,b,c,x')
p = (a + b + c)*(x^3 + 9*x + 8)
print(p)
```

If we want to expand the expression, then we apply the `expand()` method.

```
print(p.expand())
```

The `expand()` did too much, it expanded everything and we have lost the structure of the polynomial in `x`. Suppose we want to keep the factor `(a + b + c)` intact, what do we do then? Well, we declare a new variable and substitute the expression `a + b + c` to this new variable before calling the `expand()` method.

```
var('d')
dp = p.subs({a+b+c:d})
print(dp)
```

Now we expand and then replace `d` with the original `a + b + c`.

```
edp = dp.expand()
print(edp)
pp = edp.subs({d:(a+b+c)})
print(pp)
```

and we see `d*x^3 + 9*d*x + 8*d` and `(a + b + c)*x^3 + 9*(a + b + c)*x + 8*a + 8*b + 8*c`.

There is an alternative and shorter way to obtain the above result. We view `p` as a polynomial in `x` with coefficients in `QQ[a,b,c]`. Converting `p` into the ring `QQ[a,b,c][x]` is straightforward:

```
q = QQ[a,b,c][x](p)
```

The outcome of `print(q)` is the same as the result of the above `print(pp)`. The conversion of an expression into an element of a polynomial ring of course only works if the expression is a polynomial.

1.14.3 Factorization

The opposite to expand is factor. We distinguish between exact, symbolic, and numeric factorization. A complete exact factorization is only possible if all the roots are rational numbers.

```
f = factor(x^2 - 1)
print(f, 'has type', type(f))
```

and we see the expression `(x + 1)*(x - 1)`. In a symbolic factorization, we add a formal root, a so-called algebraic number. For example, if we start with the rational numbers, then we extend `QQ` with a root of an irreducible polynomial, using the symbol `a` in the extended number field `K`.

```
y = polygen(QQ)
q = y^2 + 2
print(factor(q))
print(q.is_irreducible())
```

(continues on next page)

(continued from previous page)

```
K.<a> = QQ.extension(q)
kq = (K[y])(q)
print(factor(kq))
```

Then the symbolic factorization is `(x - a) * (x + a)`. The numerical factorization happens always over a complex field.

```
z = polygen(CC)
print(factor(z^2 + 2))
```

and the numerical (approximate) factorization is `(x - 1.41421356237310*I) * (x + 1.41421356237310*I)`.

1.14.4 Assignments

1. Give the Sage commands to transform $(x+y)^2 + \frac{1}{x+y}$ into $\frac{(x+y)^3+1}{x+y}$ and vice versa.
2. Give the Sage commands to transform $x^2 + 2x + 1 + \frac{1}{x^2+2x+1}$ into $\frac{(x+1)^4+1}{(x+1)^2}$ and vice versa.
3. Give the Sage commands to transform $x^3 - xy^2 - yx^2 + y^3 + x^2 - y^2$ into $(x^2 - y^2)(x - y + 1)$.
4. Give the Sage commands to transform $(x + z^2 + 1)(y - z^2 - 1)$ into $xy - x(z^2 + 1) + (z^2 + 1)y - (z^2 + 1)^2$.

1.15 Lecture 15: Normalizing Expressions

Deciding whether two expressions represent the same mathematical object is an important problem in symbolic computation. We distinguish between the canonical and a normal form. Rewriting expressions using collection and sorting is a way of normalization. The quick, numerical, and probabilistic way to check whether two expressions are mathematically equivalent is by evaluation at a random point.

1.15.1 Normal and Canonical Form

Expressions can take many forms. We call two expressions to be equal if they are mathematically the same.

```
var('x,y')
e1 = x*(1+y)
e2 = x + x*y
print('e1 = ', e1)
print('e2 = ', e2)
print('e1 == e1 : ', e1 == e2)
```

From the last `print` we see that the equality operator `==` does not evaluate to `True` or `False`. We could compare identities, but this would only result in `True` if both names would refer to the same object,

```
print(id(e1) == id(e2))
```

which is not the case as we see `False` printed. If we are comparing expressions, then we could compare operands and operators.

```
print(e1.operands(), e1.operator())
print(e2.operands(), e2.operator())
```

But then we see `[x, y + 1] <built-in function mul>` for `e1` and `[x*y, x] <built-in function add>` for `e2`. At the data level, the two expressions are different.

What is we would take the difference of the two expressions?

```
print('e1 - e2 :', e1 - e2)
```

which then shows `e1 - e2 :  x*(y + 1) - x*y - x` so Sage does not simplify. To check for equality, we can normalize, fully expand the expressions.

```
ee1 = e1.expand()
print('e1 expanded :', ee1)
print('e2 :', e2)
print('e1 expanded - e2 :', ee1 - e2)
```

Then the difference between the expanded `e1` and the `e2` leads to `0`.

Expanding polynomials in several variables, for a given order of the variables and an order of the monomials, gives *a normal form*. Normalization is the process of bringing a mathematical expression in a normal form. For rational expressions we have seen that we can normalize numerator and denominator by removing common factors. A normal form is usually not unique, as we can then still represent numerator and denominator in expanded or factored form, which could lead to four different forms for the same expression.

The canonical form is *the* unique form of an expression. For polynomials in one variable, we can fully expand the polynomial, remove superfluous terms (like x - x) and then sort the monomials by degree in descending order (highest degree first).

1.15.2 Rewriting Multivariate Polynomials

For a multivariate polynomial, once we fix the order of the variables, we can order the terms lexicographically, first all monomials which contains the first variable, in order of degree in that variable, before all terms that do not contain the first variable.

```
R.<x,y,z> = PolynomialRing(QQ, order='lex')
p = R.random_element(degree=5, terms=20)
p
```

The default order is the degree lexicographic order, which places the terms with the highest degree first. Terms with the same degree are ordered lexicographically, first come the terms where the degree in the first variable is highest.

```
S.<x,y,z> = PolynomialRing(QQ, order='deglex')
q = S(p)
print(q, 'has type', type(q))
```

The statement `S(p)` converts the polynomial `p` into an object of the ring `S` where the terms are sorted in the degree lexicographical order.

We can rewrite the polynomial as a polynomial in one variable, where the coefficients are polynomials in the other variables. We illustrate this on a random polynomial.

```
P = QQ[x,y,z]
q = P.random_element(degree=5,terms=20)
print(q, 'has type', type(q))
```

We see that multivariate polynomials in Sage are implemented via `libSINGULAR`. SINGULAR is a computer algebra system for computational algebraic geometry.

We can write a polynomial in several variables recursively as a polynomial in one variable with its coefficients again polynomials in the same form.

To write the polynomial q as a polynomial in z, with coefficients polynomials in y that have coefficients in x, we do

```
QQ[x][y][z](q)
```

This is another normal form for multivariate polynomials.

1.15.3 A Numerical Test on Equality

There is a numerical probability-one test n the equality of expressions. First we generate a random point, choosing random complex numbers for `x` and `y`.

```
rx = CC.random_element()
ry = CC.random_element()
print('a random point :', (rx, ry))
```

And then we evaluate the expressions.

```
v1 = e1(x=rx, y=ry)
v2 = e2(x=rx, y=ry)
print(v1 - v2)
```

we see `0.000000000000000` With probability one (we could have picked a point on some common factor of `e1 - e2`), a number (close to) zero will guarantee equality of expressions. We can increase our confidence in this test by generating more random points and increasing the working precision of the computations.

1.15.4 Assignments

1. Consider the polynomial $p = (x^2 + xy + x + y)(x + y)$.
    1. Order the monomials in p in total degree order, using $x > y$.
    2. Order the monomials in p in pure lexicographic order, using $x > y$.
2. Consider the polynomial $p = (x^2 + xy + x + y)(x + y)$. Rewrite p as a polynomial in x with coefficients in y.
3. Consider the expressions $x + 1$ and $(x^2 - 1)/(x - 1)$.
    1. Are these expressions symbolically the same? Give the Sage commands to illustrate your answer.
    2. Verify the equality numerically. Show how a numerical equality test can go wrong.

1.16 Lecture 16: Review of the First 15 Lectures

In the first 15 lectures we explored the extensive number system of Sage, manipulated polynomials and general expressions.

Below is a first, preliminary list of questions to review. Consider also the quizzes and homework assignments.

1. Explain the difference between `1.0 + 10**(-32)` and `1 + 10**(-32)`.
2. The Gelfond-Schneider constant is $2^{\sqrt{2}}$. Give all Sage commands
    1. to compute a rational approximation for $2^{\sqrt{2}}$ accurate with 5 decimal places;

   2. to show that the accuracy of this approximation is indeed 5 decimal places;
   3. to compute a sequence of 10 consecutive rational approximations, starting with 5 decimal places and increasing in accuracy with one decimal place in each step of the sequence.

3. Explain the difference between `x = polygen(QQ)` and `var('x')`.
4. Consider the polynomial $p = 97x^{45} - 62x^{46} - 73x^{31}$. What is the best way to evaluate p at hardware floating point numbers? Give the relevant Sage commands and compare between the straightforward evaluation of the expression for p.
5. Define a function `whatis` which stores definitions. Below is a session with this function:
   1. `whatis('computer algebra')` returns

      ```
      call again with the definition for computer algebra
      ```

   2. `whatis('computer algebra', 'the study of algorithms in symbolic computation')`

      stores the definition for `'computer algebra'`

   3. `whatis('computer algebra')` returns

      ```
      the study of algorithms in symbolic computation
      ```

6. Consider the polynomial $p = 25x^3 + 12x^2 + 26x + 4$ over a finite field of size 29, similar to working modulo 29.
   1. Does the polynomial factor over this field?
   2. Give the Sage commands to add sufficiently many formal roots to the field so p has three roots over the extended number field.
   3. Write the symbolic factorization of p in linear factors below.
7. Write a Python function `python_sum` which takes on input a positive integer n and which returns the floating-point value of

$$\frac{\pi}{n}\sum_{i=1}^{n-1}\cos\left(-\frac{\pi}{2}+i\frac{\pi}{n}\right).$$

   Write a more efficient version `numpy_sum` using vectorization.

   Time the two versions to illustrate the efficiency.

8. Explain how rational expressions are normalized. Illustrate with an example.
9. Type `var('x,y'); q = (x^2 - y)/(y^2 - x)` and draw the expression three of `q`.
10. Give the Sage commands to transform $(x+(z^2+1))(y-(z^2+1))$ into $xy+(-z^2-1)x+(z^2+1)y-z^4-2z^2-1$.

1.17 Lecture 17: the First Midterm Exam

The first midterm exam covers the first two parts of the course.

1.17.1 Questions on the Spring 2017 First Midterm Exam

The list of questions on the Spring 2017 first midterm exam are below. The exam is open book, open notes and open computer. All answers to the questions must be handwritten, submitted on paper.

1. Let N be the number $\exp(\pi)$.
    1. Write a rational approximation A for N, accurate with 3 decimal places.
    2. Give the Sage command(s) to verify that A approximates N indeed with 3 decimal places.
    3. Compute a list of consecutively more accurate approximations for N.

       The list should start with the number A and have length 10.

       Do not write the list, write only the Sage command(s) to compute this list.
2. Let $p = x^3 + x + 1$ be a polynomial over the finite field (also called a Galois field) of five numbers.
    1. Write the Sage commands to define p and to show that p does not factor over this finite field.
    2. Use p to extend the finite field of five numbers.

       Write the factorization of p over this field extension.

       Give all relevant Sage commands you used to obtain this factorization.
3. What is a symbolic ring? Give a good example of a use of a symbolic ring.
4. Consider a polynomial with integer coefficients in three variables, x, y, and z.
    1. Define one normal form of such a polynomial.
    2. Illustrate your definition with an example of a random polynomial.
    3. What is the application of a normal form?
5. Let $q = (x^2 + x - 1)/(x^2 - 2)$. Draw the expression tree of q.

   Do not write any Sage commands.
6. We have covered a couple of different ways to substitute. Describe two different ways and illustrate each way with an example. Explain on the examples why one way to substitute is appropriate for the example instead of the other way to substitute.
7. Let $p = (x^2 - 3y)(x + 2y)$ and $q = x^3 + 2x^2y - 3xy - 6y^2$.

   Give the Sage commands to
    1. transform p into q; and
    2. transform q into p.

1.17.2 Questions on the Fall 2018 First Midterm Exam

The list of questions on the Fall 2018 first midterm exam are below. The exam is open book, open notes and open computer. All answers to the questions must be handwritten, submitted on paper.

1. Let N be the natural logarithm of 10.
    1. Give a floating-point approximation A of N, accurate with 5 decimal places.
    2. Compute a rational approximation R for N, accurate with 5 decimal places. Write the value for R. Write the Sage command(s) and its output to verify that R approximates N indeed with an accuracy of 5 decimal places.

3. Give the Sage command to compute a list of the first 10 rational approximations of N. Do not write the entire list, write only its last element.

2. Let `x` and `y` be variables, declared via `x, y = var('x,y')`.

   Explain the difference between `y = 2; x = y` and `y = 2; x = 'y'`.

   Explain the relations between `x` and `y` in both cases.

3. Let p be the polynomial $p = x^8 + 2x^7 + 5x^6 - 3x^5 - 4x^4 - x^3 + x^2 + x - 2$.

   What is the smallest number of arithmetical operations needed to evaluate p?

   What is the fastest way to evaluate p in hardware arithmetic?

   1. Give all Sage commands to convert p into an expression best suited for fast evaluation. Do not write the output of the Sage commands.
   2. Give timing results to compare the straightforward evaluation of p with the evaluation of the form for fast evaluation.

4. For this question, write all relevant Sage commands and their output.

   Define the polynomial $p = 9x^3 - 72x^2 - 78x + 3$ over the field of rational coefficients.

   1. Is p irreducible? If not, what are the factors of p?
   2. Extend the field of rational numbers with sufficiently many formal roots so p factors as a product of linear polynomials over the field extension.

5. Type `x, y = var('x, y'); q = (x-y+1)/(x^2+3*y)` and draw the expression tree of `q`.

   Give some relevant Sage commands you used to obtain your drawing.

6. For polynomials in several variables explain the difference between the pure lexicographic and the degree lexicographic order.

   Illustrate the difference with good examples.

7. What are the commands to transform $\dfrac{(x-y)^2}{(x+y)^2} + 1$ into $\dfrac{(x-y)^2 + (x+y)^2}{(x+y)^2}$?

1.17.3 Questions on the Spring 2019 First Midterm Exam

The list of questions on the Spring 2019 first midterm exam are below. The exam is open book, open notes and open computer. All answers to the questions must be handwritten, submitted on paper.

1. Consider $c = \sqrt{\pi/3}$.
   1. Give a rational approximation to c, accurate with 5 decimal places.
   2. What is the actual error of your rational approximation?
   3. Give a list of rational approximations of c. The first number of the list is the same rational number as the one above, accurate to 5 decimal places. The last rational number has an accuracy of 10 decimal places.
2. For `x = var('x')`, why does Sage not automatically simplify `sqrt(x^2)` to `x`?

   Illustrate your explanation with a good example.
3. Consider the efficient evaluation of $p = x^6 + 2x^5 + 3x^4 + 4x^3 + 5x^2 + 6x + 7$.
   1. Construct the Horner form for p.

      How many arithmetical operations does it take to evaluate p?

2. What is the most efficient way to evaluate p with hardware arithmetic?

   Compare this most efficient way to the straightforward way to evaluate p.

4. Give all relevant commands for this problem. Consider the polynomial $p = x^4+x^2+1$ over the field $\mathbb{Z}_2 = \{0, 1\}$, the field of bits, 0 and 1.

   1. Is p irreducible? If not, then what are its factors?
   2. Extend the field $\mathbb{Z}_2$ with sufficiently many formal roots so p factors as a product of linear factors. Write the factorization of p.

5. Type `x,y = var('x,y')` followed by `q = (x^2 - y + 1)/(y^2 - x + 1)`.

   Draw the internal representation of `q`.

   Give some relevant commands used to make your drawing.

6. Explain the meaning of `QQ['x,y,z']`, `QQ['x']['y,z']`, and `QQ['z']['y']['x']`.

   Explain the differences between `QQ['x,y,z']`, `QQ['x']['y,z']`, and `QQ['z']['y']['x']`.

   Illustrate the differences with good examples.

7. Write the commands to transform $x + \dfrac{(x-y)^5}{(x+y)^5}$ into $\dfrac{(x+y)^5 \cdot x + (x-y)^5}{(x+y)^5}$.

1.17.4 Questions on the Summer 2022 First Midterm Exam

The exam in the summer 2022 took 100 minutes and was administered online, in two different versions. Questions had to be uploaded into one single Jupyter notebook into gradescope.

The list of the nine questions on the first version is below.

1. Let $N = \cos(\pi/5)$.

   1. Compute a rational approximation for N, accurate to 2 decimal places.
   2. Verify that the accuracy of your rational approximation is indeed 2 decimal places.
   3. Compute the sequence of the first ten consecutive rational approximations for N, accurate with 2, 3, up to 11 decimal places.

2. Explain the difference between $1.0 + 10^{-32}$ and $1 + 10^{-32}$.

   What would you need to do for SageMath to return the *same*, *correct*, and *real* value of those two sums?

3. Declare `r` and `theta` as variables in the `RR` domain.

   Define `n` as the expression `r*exp(I*theta)`. What are the real and imaginary parts of `n`?

4. Compute the Horner form of $p = x^7 - 7x^5 - 2x^4 - x^3 - 4x^2 - 2x + 1$.

   Explain the use of the Horner form.

   What is its advantage of the Horner form over representing p as above?

5. Explain the difference between `eval(x)` and `eval(preparse(x))`.

   Give a good example of `x` to illustrate the difference.

6. Consider the function

```
def python_sum(n):
    """
    Returns the floating-point value of the sum of (3*k/n)**2,
    for k from 1 to n-1, multiplied by 3/n.
    """
    return float((3/n)*sum([(3*k/n)**2 for k in range(1, n)]))
```

   1. Time the execution of `python_sum(10**4)`.
   2. Explain why the execution is slow.
   3. Apply vectorization to write a faster version. Verify the correctness.
   4. Time the execution of your vectorized function for `n = 10**4` and compare with your timing of the original `python_sum(10**4)`.

7. Consider the polynomial $p = x^3 - x^2 - 3$.

   Compute the exact, symbolic, and numeric factorization for p.

8. Consider the expression:

$$\frac{a - 2b}{3a + b + 2c},$$

   for a, b, c defined as `a, b, c = var('a, b, c')`.

   Draw the internal representation of this expression.

9. Transform $(y+1)^6 - x^4$ into $((y+1)^3 + x^2)((y+1)^3 - x^2)$.

The list of the nine questions on the second version is below.

1. Let $N = \log(\pi/2)$.
   1. Compute a nearby rational approximation for N, with the denominator bounded by 100.
   2. What is the accuracy of your nearby rational approximation?
   3. Compute the sequence of the first ten nearby rational approximations for N, where the size of the denominator of the k-th approximation is bounded by 10^k, $k = 1, 2, \ldots 10$.
2. Explain the difference between 0 and 0.000.

   Give a good example of a calculation which results in 0.000, to illustrate the difference with a result of a computation equal to 0.
3. Make `eqn` so `print(eqn)` shows `cos(pi) == -1`.

   Without retyping `eqn`, change `eqn` so `print(eqn)` shows `-1 == -1`.
4. Consider the statements

```
a, b, c = var('a, b, c')
a = b; b = 2; c = b
```

   Describe the relations between the three variables.

   Execute the commands to verify the relations.
5. What does the `preparse()` function do?

   Illustrate your explanation with a good example where `preparse()` is needed.
6. Consider the function

```
def python_sum(n):
    """
    Returns the floating-point value of the sum of exp(-k/n)
    for k from 1 to n-1, multiplied by 1/n.
    """
    return float((1/n)*sum([exp(-k/n) for k in range(1, n)]))
```

   1. Time the execution of `python_sum(10**4)`.
   2. Explain why the execution is slow.
   3. Apply vectorization to write a faster version. Verify the correctness.
   4. Time the execution of your vectorized function for `n = 10**4` and compare with your timing of the original `python_sum(10**4)`.

7. Consider the polynomial $p = x^3 + x^2 + 3$.

   Compute the exact, symbolic, and numeric factorization for p.

8. Draw the binary expression tree defined by the fast callable

$$\frac{a + 2b}{3a - b + 2c}.$$

9. Transform $((y + 1)^3 + x^2)((y + 1)^3 - x^2)$ into $(y + 1)^6 - x^4$.

CHAPTER 2

Part III: Calculus

The third part of the course consists of a sequence of seven lectures. We call it calculus as we work with functions, used for differentiation, integration, and approximation. Giving a dictionary with as default value an empty dictionary as last argument of a recursive functions works great for memoization.

2.1 Lecture 18: Defining Mathematical Functions

Expressions in SageMath are callable objects and for fast evaluation in machine numbers we have *fast_callable* objects. Although we may define functions with the Python def syntax, we can differentiate, integrate, and plot Sage functions. The simplest discontinuous functions are step functions.

2.1.1 Functions in SageMath and in Python

We can define functions as in Python, but also more directly.

```
f(x,y) = sin(x) + e^cos(y)
print(f, 'has type', type(f))
```

The print statement shows `(x, y) |--> e^cos(y) + sin(x)` and we see that a function is an expression.

```
print(f(2,pi))
print(f.integrate(x))
```

This shows `e^(-1) + sin(2)` and `(x, y) |--> x*e^cos(y) - cos(x).` Observe thus that, as we integrate a function, the result is again displayed as a function.

Let us explore the difference with Python functions.

```
def g(x,y):
    return sin(x) + e^cos(y)
print(g, 'has type', type(g))
```

Now the type of g is that of function. We cannot integrate a Python function, the command below does not work.

```
g.integrate(x)
```

because the function object has no attribute integrate. But because we are defining functions in SageMath, we can evaluate the function symbolically.

```
e = g(x,y)
print(e, 'has type', type(e))
```

and we see e^cos(y) + sin(x) has type <type 'sage.symbolic.expression.Expression'>

So we can turn an expression into a function.

```
h(x,y) = e
print(h, 'has type' ,type(h))
print(h.integrate(x))
```

We see that h is defined as a function that we can integrate.

2.1.2 Step Functions

Can we define step functions? Step functions are the simplest discontinuous functions.

```
print(unit_step, 'has type', type(unit_step))
print('at -3 :', unit_step(-3))
print('at 4 :', unit_step(4))
```

The unit_step is defined in the class sage.functions.generalized.FunctionUnitStep. For negative x values, the value of the unit step is zero, for positive x, the value is one. Let us look at the plot, for x between -1 and +1.

```
plot(unit_step, -1, 1, aspect_ratio=1)
```

Setting the aspect_ratio to 1 forces the plot to take the same scale on the horizontal as on the vertical axes. The unit step function is shown in Fig. 2.1.

The derivative of the unit step function is the Dirac delta function.

```
diff(unit_step(x),x)
```

which shows dirac_delta(x).

We can take the definite integral of the unit step function.

```
integral(unit_step(x), (x, -3, 3))
```

and, as expected, the integral returns 3.

Another important discontinuous function is the Kronecker delta, which is a function in two variables returning 1 if both arguments are equal and returning 0 otherwise.

```
print(kronecker_delta(1,2))
print(kronecker_delta(1,1))
```

and we see 0 and 1 printed.

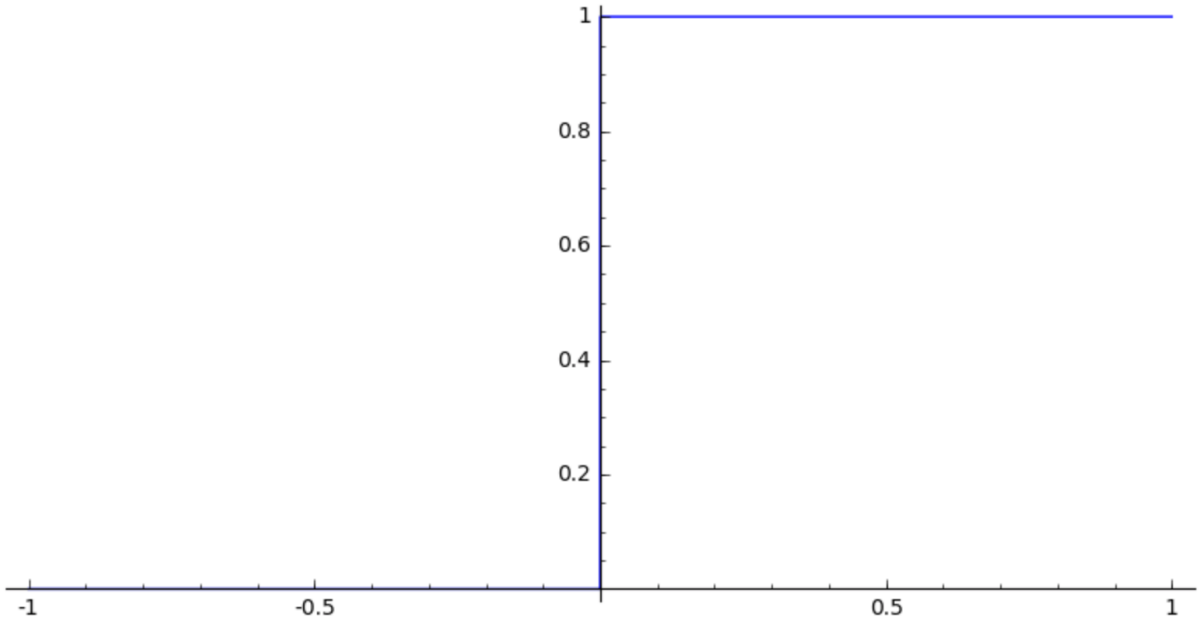

Fig. 2.1: The unit step function.

2.1.3 Piecewise Functions

We can make piecewise functions. For example,

$$f(x) = \begin{cases} 1 & if \quad x < 1 \\ \infty & if \quad x = 1 \\ x^2 & if \quad x > 1 \end{cases}$$

To define this piecewise function, we use `piecewise`.

```
f = piecewise([((-infinity, 1), 1), ([1,1], infinity), ((1, infinity), x^2)] )
print(f)
```

2.1.4 Combining Functions

We can combine functions. Suppose we want to make a 3-step staircase function, defined as follows: f(x) = 0 for x < 0, f(x) = 1/3 for x in [0,1), f(x) = 2/3 for x in [1,2) and f(x) = 1 for x >= 2.

```
f(x) = unit_step(x)/3 + unit_step(x-1)/3 + unit_step(x-2)/3
plot(f(x),(x,-1,3),aspect_ratio=0.5)
```

The figure is shown in Fig. 2.2.

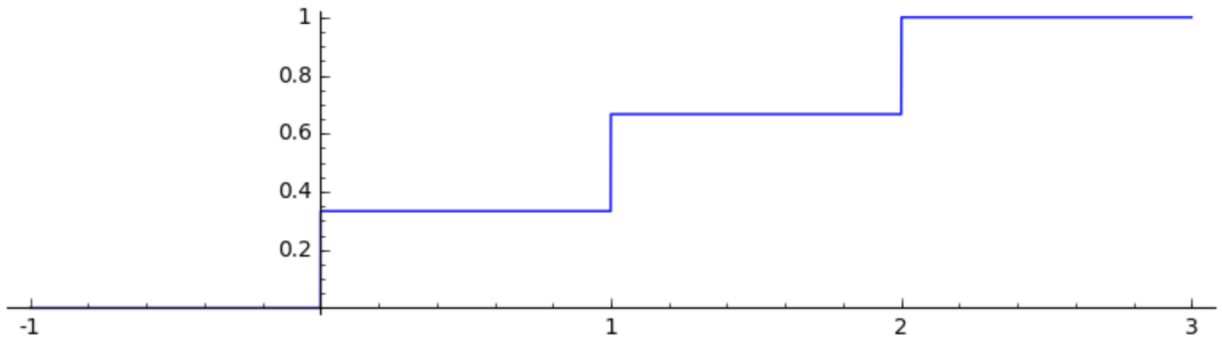

Fig. 2.2: A staircase function.

2.1.5 Assignments

1. Define the following function:

$$f(x) = \begin{cases} 3x+1 & if \quad x\ odd \\ x/2 & if \quad x\ even \end{cases}$$

   Apply `f` recursively starting at a random integer number n, computing $f(n)$, then $f(f(n))$, $f(f(f(n)))$, and so on. Do you observe a pattern?

2. Run the code below in a SageMath session. Explain what happens. Hint: consider the evaluation of a Python function at a variable in turning an expression into a SageMath function.

```
def h(x):
    if x<2:
        return 0
    else:
        return x-2
plot(h(x), (x, -2, 2))
```

3. Define a piecewise linear function that is zero for negative values of x and takes the value x/2 for positive values of x.

4. Make a hat function `f(x)` with definition `f(x) = 0` for `x` less than -1 or larger than 1; and `f(x) = 1` for `x` in the interval `[-1,+1]`. Show with a plot that your definition is correct.

5. Consider the formula

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

   Make a function `f(k)` that returns the value of what is in the $(\cdot)$ in the sum for π.

   Use this function in a list comprehension to compute the first 100 hexadecimal digits of π.

   How accurate is this approximation for π? Write the magnitude of the error.

2.2 Lecture 19: Recursive Functions

Recursion is a very powerful tool to define function in a compact way. Unfortunately, a direct implementation of a recursive function may lead to very inefficient code. Using dictionaries in Python we can apply memoization and obtain efficient recursive functions without having to rewrite the recursion into an iteration.

2.2.1 Memoization in Python

The Fibonacci numbers are defined recursively. The first two numbers are zero and one. The next numbers are the sum of the two previous ones. To compute the n-th Fibonacci number, we follow the three rules listed below.

1. $F(0) = 0$
2. $F(1) = 1$
3. $F(n) = F(n-1) + F(n-2)$, if $n > 1$.

The Python function below follows those three rules in a nested branching statement.

```
def fibonacci(n):
    """
    Returns the n-th Fibonacci number.
    """
    if n == 0:
        result = 0
    elif n == 1:
        result = 1
    else:
        result = fibonacci(n-1) + fibonacci(n-2)
    return result
```

Let us look at the first 10 Fibonacci numbers:

```
[fibonacci(n) for n in range(11)]
```

and this shows the list of the first 10 Fibonacci numbers: `[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]`.

The problem is that it gets very efficient as n grows.

```
timeit('fibonacci(20)')
```

We see `25 loops, best of 3:  15.8 ms per loop` as executed on a 3.1 GHz Intel Core i7 Mac OS X 10.12.6. Let us continue:

```
timeit('fibonacci(21)')
```

shows `25 loops, best of 3:  24.7 ms per loop` and we do it once more, for the 22-nd Fibonacci number:

```
timeit('fibonacci(22)')
```

and we see `5 loops, best of 3:  40.3 ms per loop`. Observe that 15.8 + 24.7 = 40.5. The time to compute the 22-nd Fibonacci number equals the sum of the time to compute the 20-th and the 21-st Fibonacci number. The timings thus also follow a similar pattern as the Fibonacci number are increase exponentially.

If we examine the tree of function calls, then we see why this definition leads to a very inefficient function. We abbreviate fibonacci() by f().

```
f(5)
 +-- f(4)
 |    +-- f(3)
 |    |    +-- f(2)
 |    |    |    +-- f(1) = 1
 |    |    |    +-- f(0) = 0
 |    |    +-- f(1) = 1
 |    +-- f(2)
 |         +-- f(1) = 1
 |         +-- f(0) = 0
 +-- f(3)
      +-- f(2)
      |    +-- f(1) = 1
      |    +-- f(0) = 0
      +-- f(1) = 1
```

Even for a very simple case as the 5-th Fibonacci number, we see that the f(3) gets computed thrice and f(2) twice.

If we draw the tree with `LabelledBinaryTree`, starting at the leaves `F(0)` and `F(1)`, then we arrive at the iterative version of the algorithm.

```
L0 = LabelledBinaryTree([], label='F(0)')
L1 = LabelledBinaryTree([], label='F(1)')
L2 = LabelledBinaryTree([L0, L1], label='F(2)')
ascii_art(L2)
```

The two base cases are displayed in Fig. 2.3 as they occur in the computation of `F(2)`.

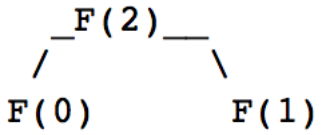

Fig. 2.3: The computation of `F(2)` as `F(0) + F(1)`.

Consider the computation of `F(3)`, `F(4)`, and `F(5)`, following the iterative algorithm to compute `F(5)`.

```
L3 = LabelledBinaryTree([L1, L2], label='F(3)')
L4 = LabelledBinaryTree([L2, L3], label='F(4)')
L5 = LabelledBinaryTree([L3, L4], label='F(5)')
ascii_art(L5)
```

The result of `ascii_art(L5)` is shown in Fig. 2.4.

With default arguments we can have functions store data. In the memoized version of the recursive Fibonacci function we use a dictionary to store the values for previous calls. The keys in the dictionary are the arguments of the function and the values are what the function returns. With each call to the function, the dictionary is consulted and only if there is no key for the argument, then the body of the function is executed.

```
def memoized_fibonacci(n, D={}):
    """
    Returns the n-th Fibonacci number, using D to
    memoize the values computed in previous calls.
```

(continues on next page)

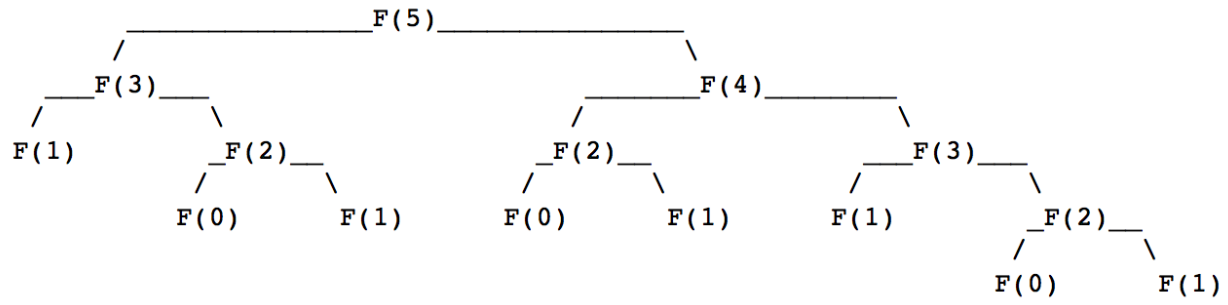

Fig. 2.4: The tree of function calls in the computation of `F(5)`.

(continued from previous page)

```
    """
    if n in D:
        return D[n]   # dictionary lookup
    else:
        if n == 0:
            result = 0
        elif n == 1:
            result = 1
        else:
            result = memoized_fibonacci(n-1) + memoized_fibonacci(n-2)
    D[n] = result     # store the new result
    return result
```

Now if we time this new function.

```
timeit('memoized_fibonacci(20)')
```

we see `625 loops, best of 3:  590 ns per loop`.

```
[memoized_fibonacci(n) for n in range(30)]
```

In the memoized version, we have replaced a number of function calls that grows exponentially in n by an amount of storage that grows *linearly* in n. For memoization to work, it is important that the anount of storage remains proportional to the dimension of the problem.

2.2.2 Memoization in SageMath

In symbolic computing we define functions to compute expressions. Chebyshev polynomials are orthogonal polynomials, available in SageMath via the function `chebyshev_T`.

```
t3 = chebyshev_T(3, x)
print(t3, 'has type', type(t3))
```

and we see `4*x^3 - 3*x has type <type 'sage.symbolic.expression.Expression'>`.

The 3-terms recurrence to define a Chebyshev polynomial of degree n in the variable x is

1. $T(0, x) = 1$,
2. $T(1, x) = x$, and

3. $T(n,x) = 2xT(n-1,x) - T(n-2,x)$.

The straightforward definition based on this 3-terms recurrence is below in the function `T`. Observe the application of `expand()`. We normalize the Chebyshev polynomials into the fully expanded form.

```
def T(n, x):
    """
    Returns the n-th Chebyshev polynomial in x.
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    else:
        return expand(2*x*T(n-1, x) - T(n-2, x))
```

To test the function, we compute the third Chebyshev polynomial.

```
T(3, x)
```

and indeed, we see `4*x^3 - 3*x`. We can check the efficiency of the implementations.

```
print('the Chebyshev in Maxima :')
timeit('chebyshev_T(10,x)')
print('our direct function T :')
timeit('T(10,x)')
```

The output is

```
the Chebyshev in Maxima : 625 loops, best of 3: 799 µs per loop
our direct function T : 25 loops, best of 3: 10.2 ms per loop
```

We apply the memoization technique from Python to the SageMath function `T` and call the memoized function `mT`. Our `T` had two parameters:

1. `n` is the degree of the polynomial, and
2. `x` is the variable in the polynomial.

Therefore, the keys in the dictionary will be a tuple `(n, x)`.

```
def mT(n, x, D = {}):
    """
    Returns the n-th Chebyshev polynomial in x,
    using memoization with the dictionary D.
    """
    if (n, x) in D:
        return D[(n, x)]  # dictionary lookup
    else:
        if n == 0:
            result = 1
        elif n == 1:
            result = x
        else:
            result = expand(2*x*mT(n-1, x) - mT(n-2, x))
    D[(n, x)] = result    # store the new polynomial
    return result
```

To check its correctness, we do again `print(mT(3,x))` to see `4*x^3 - 3*x`. Let us check how much faster the memoized version is.

```
timeit('mT(10,x)')
```

and we obtain `625 loops, best of 3:  491 ns per loop`.

2.2.3 Assignments

1. The Bell numbers $B(n)$ are defined by $B(0) = 1$ and

$$B(n) = \sum_{i=0}^{n-1} \binom{n-1}{i} B(i),$$

for $n > 0$. They count the number of partitions of a set of n elements.

Write a recursive function to compute the Bell numbers. The binomial coefficient $\binom{n-1}{i}$ is computed by `binomial(n-1,i)`. Make sure your procedure is efficient enough to compute `B(50)`.

2. The Stirling numbers of the first kind $c(n, k)$ satisfy the recurrence

$$c(n, k) = -(n-1)c(n-1, k) + c(n-1, k-1), \text{ for } n \geq 1 \text{ and } k \geq 1,$$

with the initial conditions that $c(n, k) = 0$ if $n \leq 0$ or $k \leq 0$, except $c(0, 0) = 1$.

   1. Write an *efficient recursive* function, call it `stirling1` to compute $c(n, k)$.

      The n must be the first argument of `stirling1` while k is its second argument, e.g.: for $n = 100$ and $k = 33$, `stirling1(100,33)` should return $c(100, 33)$.

   2. How many digits does the number $c(100, 33)$ have? Give also the SageMath command(s) to obtain this number.

3. The n-th Chebychev polynomial is also often defined as $\cos(n \arccos(x))$.

   Give the definition of the function `C` which takes on input the degree n and a *value* for x.

   Thus `C(n,x)` returns $\cos(n \arccos(x))$ while `C(10,0.512)` returns the value of the 10-th Chebychev polynomial at 0.512. Compare this value with `chebyshev_T(10,0.512)`.

4. Let `L(n,x)` denote a special kind of the Laguerre polynomial of degree `n` in the variable `x`.

   We define `L(n,x)` by three rules:

   1. `L(0,x) = 1`,
   2. `L(1,x) = x`, and
   3. for any degree `n > 1`: `n*L(n,x) = (2*n-1-x)*L(n-1,x) - (n-1)*L(n-2,x)`.

   Write a SageMath function `Laguerre` that returns `L(n,x)`. Make sure your function can compute the 50-th Laguerre polynomial.

5. Denote the composite Trapezoidal rule for $\int_a^b f(x)dx$ using 2^n intervals by `T(n,f,a,b)`.

   We can define `T(n, f, a, b)` recursively by two rules:

   1. `T(0, f, a, b) = (f(a) + f(b))*(b-a)/2` and
   2. `T(n, f, a, b) = T(n-1, f, a, (a+b)/2) + T(n-1, f, (a+b)/2, b)`, for `n > 0`.

   Do the following.

1. Write a recursive SageMath function for `T`.
2. Explain how you can define `T` so that `f` is never evaluated twice at the same point.

   Illustrate using `n = 5` in `T` for the numerical approximation of $\int_0^1 \cos(x)dx$.

6. The Bernoulli polynomials are defined by

$$B_0(x) = 1, \quad B_k(x) = k\left(\int_0^x B_{k-1}(t)dt - \int_0^1 \left(\int_0^x B_{k-1}(t)dt\right)dx\right), k > 0.$$

   1. Write an efficient recursive function `B` which returns $B_k(x)$ in expanded form. The arguments of `B` are the degree k and the name of the variable x in the polynomial $B_k(x)$. Thus, to compute $B_{50}(x)$, we type `B(50,x)`.
   2. What is the coefficient of x^{49} in the polynomial $B_{50}(x)$? Give also the SageMath command to obtain this coefficient.

2.3 Lecture 20: Computing with Functions

List comprehensions in Python take the form `[f(x) for x in L]` where `L` is some list and `f` some function. The result is a new list that contains all function values of `f(x)` for all elements `x` in the list `L`. In SageMath we can apply list comprehensions to build expressions of arbitrary size and shape.

2.3.1 List Comprehensions

We can use a list comprehension to create a sequence of 20 variables, starting from `x01`, `x02`, .., `x20`.

```
v = [var('x' + '%02d' % k) for k in range(1,21)]
v
```

Observe the string concatenation with the proper formatting of the integer index. This leads to the list `[x01, x02, x03, x04, x05, x06, x07, x08, x09, x10, x11, x12, x13, x14, x15, x16, x17, x18, x19, x20]`. Now we raise every variable to the power three.

```
p = [x^3 for x in v]
p
```

and we computed `[x01^3, x02^3, x03^3, x04^3, x05^3, x06^3, x07^3, x08^3, x09^3, x10^3, x11^3, x12^3, x13^3, x14^3, x15^3, x16^3, x17^3, x18^3, x19^3, x20^3]`. We add up the powers into a sum to make an expression.

```
e = sum(p)
print(e, 'has type', type(e))
```

The sum leads to the expression `x01^3 + x02^3 + x03^3 + x04^3 + x05^3 + x06^3 + x07^3 + x08^3 + x09^3 + x10^3 + x11^3 + x12^3 + x13^3 + x14^3 + x15^3 + x16^3 + x17^3 + x18^3 + x19^3 + x20^3`. We can return to the list representation via the `operands()` method.

```
ope = e.operands()
print(ope, 'has type', type(ope))
```

and this gives a list `[x01^3, x02^3, x03^3, x04^3, x05^3, x06^3, x07^3, x08^3, x09^3, x10^3, x11^3, x12^3, x13^3, x14^3, x15^3, x16^3, x17^3, x18^3, x19^3, x20^3]`. Raising the k-th operand to the power k goes as follows.

```
r = [ope[k]^k for k in range(len(ope))]
r
```

and then we see `[1, x02^3, x03^6, x04^9, x05^12, x06^15, x07^18, x08^21, x09^24, x10^27, x11^30, x12^33, x13^36, x14^39, x15^42, x16^45, x17^48, x18^51, x19^54, x20^57]`. Note that all operands are expressions.

```
print(r[2], 'has type', type(r[2]))
print('its degree in' , v[2], ':', r[2].degree(v[2])
```

With the proper selection of the variable in the argument of `degree()` we get the right degree, which is `6` for `r[2]`. We can compute the degrees in their variables.

```
[(r[k]).degree(v[k]) for k in range(len(ope))]
```

which leads to `[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, 57]`.

Suppose we want to select those operands of degree less than 13. Recall the unit_step() function.

```
print(r[2], ':', unit_step(13 - r[2].degree(v[2])))
print(r[8], ':', unit_step(13 - r[8].degree(v[8])))
```

and we see `x03^6 :  1` and `x09^24 :  0`. Now we can remove all operands with degree higher than 13.

```
filter = lambda x, k: x*unit_step(13 - x.degree(v[k]))
s = [filter(r[k],k) for k in range(len(r))]
s
```

and we see `[1, x02^3, x03^6, x04^9, x05^12, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]` and then build the expression again with `sum()`.

```
t = sum(s)
t
```

and we obtain `x05^12 + x04^9 + x03^6 + x02^3 + 1`.

2.3.2 Composing Functions

Consider the trajectory of a projectile in space modeled by a parabola, subject to the following constraints. At time t = 0 it is launched from the ground and at time t = 45 it hits the ground 120 miles further. Assuming constant horizontal speed we create a function f(t) to give the altitude of the projectile in function of time. First we model the shape of the trajectory.

```
y(x) = x*(120 - x)
plot(y(x), (x, 0, 120), aspect_ratio=1/100, thickness=3, color='red')
```

The parameters `thickness` and `color` embellish the plot, adjusting the default value for the `aspect_ratio` to `1/100` alters the display of the shape of the trajectory, so we do not have to worry about units. The plot is displayed in Fig. 2.5.

Now we want a function in time t. The assumption of constant horizontal speed implies that x is just a rescaling of t. This means that when t = 45, x must be 120.

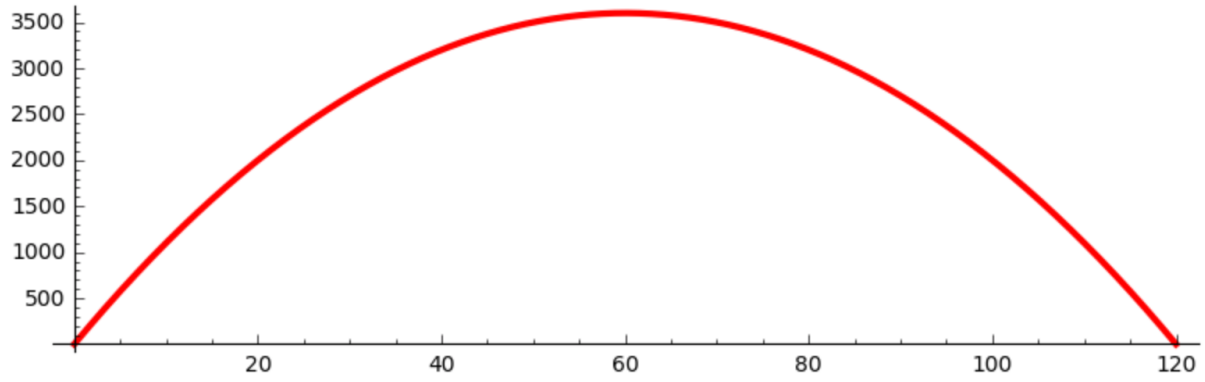

Fig. 2.5: A parabolic trajectory.

```
x(t) = 120/45*t
print('x at 0 :', x(0))
print('x at 45 :', x(45))
```

and indeed, we see `x at 0 :  0` and `x at 45 :  120` printed. Then the altitude is given as the composition of y after x.

```
f(t) = y(x(t))
print('altitude at 0 :', f(0))
print('altitude at 22.5 :', f(22.5))
print('altitude at 45 :', f(45))
```

as altitudes at 0, 22.5, and 45, we respectively see `0`, `3600.00000000000` and `0`. With the composition of functions we have separated the geometric shape of the trajectory and its evolution in time. Suppose we want to halve the speed.

```
h(t) = t/2
hf(t) = f(h(t))
print('altitude at 45 :', hf(45))
print('altitude at 90 :', hf(90))
```

and the prints confirm that the projectile reaches its peak at 45 and falls to the ground at 90.

We can also make a 3d-plot of the space curve.

```
parametric_plot3d([x(t), 0, f(t)], (t, 0, 45), thickness=5, color='red')
```

and this shows the following image:

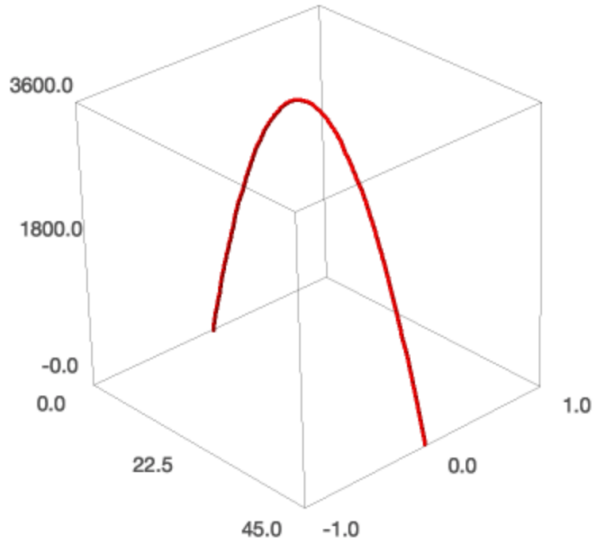

Fig. 2.6: The parabolic trajectory plotted in three dimensions.

2.3.3 Functions Returning Functions

We can define functions that return functions. Suppose we are interested in solving equations with Newton's method.

```
def newton_step(f, x):
    """
    Returns the function that will perform one Newton step
    to solve the equation f(x) = 0.
    """
    n(x) = x - f(x)/diff(f,x)
    return n
```

Let us try it out on the square function, so we can compute the square root of two.

```
sqr(x) = x^2 - 2
our_sqrt = newton_step(sqr, x)
our_sqrt
```

Then we see `x |--> x - 1/2*(x^2 - 2)/x`. Observe that if we start at an integer value, we get rational approximations for the square root of two.

```
our_sqrt(our_sqrt(our_sqrt(2)))
```

Starting at 2, three iterations of Newton's method yields `577/408`. For a repeated application of a function, we multiply strings.

```
s = 'our_sqrt('*5 + '2' + ')'*5
s
```

We will then execute `our_sqrt(our_sqrt(our_sqrt(our_sqrt(our_sqrt(2)))))` and we evaluate the expression.

```
v = eval(s)
v
```

This gives `886731088897/627013566048`. To compare with an actual approximation for sqrt(2).

```
print(v.n(digits=30))
print(sqrt(2).n(digits=30))
```

and we see

```
1.41421356237309504880168962350
1.41421356237309504880168872421
```

2.3.4 Assignments

1. Type `L = [randint(0,99) for i in range(10)]` to generate a list of 10 random numbers between 0 and 99. Give the SageMath commands for the following operations.

   (a) divide every element in the list by 100;

   (b) convert the list to a list of floating-point numbers of a precision of 3 decimal places;

   (c) select all elements in the list that are larger than 0.5;

   (d) compute the sum of the elements in the list.

2. The command `is_prime()` applied to a number returns `True` if the number is prime and `False` otherwise. Use `is_prime()` to make a list of all primes less than 1000. How many 3-digit primes are there?

3. Do `P.<x> = PolynomialRing(RR, sparse=False)` and then

   `q = P.random_element(degree=10)`.

   Select from `q` all terms with positive coefficients and make a new polynomial with those terms.

4. For a positive integer n, and a list X of n variables X = `[ x1, x2, .. , xn ]`, define the function

$$H(n, X, k, c) = 2nX[k] - cX[k]\left(1 + \sum_{i=0}^{n-1} \frac{i+1}{i+3} X[i]\right) - 2n,$$

   where c is some variable parameter. Show the definition of your function is correct for a list of 9 variables.

5. Let n and k be positive natural numbers with $k < n$ and consider the polynomial $\sum_{i=0}^{n-1} \prod_{j=0}^{k-1} z_{(i+j)} \bmod n$.

   Define a function that takes on input a list of variables (where n is the length of the list) and a value for k. On output is the expression as defined by the polynomial above. Hint: use `prod` to make the product of a list of expressions.

   Show the result of your function on a sequence of 10 variables and for $k = 3$.

6. Execute the `newton_step` function on the `our_sqr` function, defined again as `x^2 - 2`. Newton's method has the property of converging quadratically, that is: in each step the correct number of decimal places doubles. Starting at `2`, perform 10 steps with Newton's method and verify the progess of the accuracy of the rational approximations for the square root of 2? Do you observe quadratic convergence?

7. Define a Python function `makeHorner` which takes on input the coefficients of a polynomial and the symbol of the variable in the polynomial. For example, if `C` is `[c0, c1, c2, c3, c4]` and the variable is `x`, then `makeHorner(C, x)` returns

```
(((c4*x + c3)*x + c2)*x + c1)*x + c0
```

Your function `makeHorner` should directly build the polynomial and not apply the `horner()` to an expanded polynomial.

2.4 Lecture 21: Symbolic and Numeric Differentiation

We can take derivatives symbolically, of expressions and functions. Numerically, we work with difference formulas. Some functions are defined implicitly, as the solution of some equation. Declaring some variables as functions of others, we can derive the equation and solve for the derivative. This is implicit differentiation.

2.4.1 Symbolic Differentiation

We can compute the derivative of an expression with `diff`, which is an alias for the `derivative` method.

```
print('the derivative of cos(x) is', diff(cos(x),x))
```

and we see that `the derivative of cos(x) is -sin(x)`. For repeated differentiation, we have an extra argument.

```
[diff(cos(x), x, k) for k in range(5)]
```

and this list comprehension returns `[cos(x), -sin(x), -cos(x), sin(x), cos(x)]`. We can compute derivatives of functions.

```
f(x,y) = x^2*y + 2*x*y + x
print('f =', f)
print('the derivative of f is', f.diff())
```

The function `f = (x, y) |--> x^2*y + 2*x*y + x` has as its derivative `(x, y) |--> (2*x*y + 2*y + 1, x^2 + 2*x)`. We see that the derivative of a function in two variables returns a function that returns a tuple of two expressions. This tuple is the *gradient* of `f`.

```
print(f.diff(x))
print(f.diff(y))
```

and we see the two partial derivatives `(x, y) |--> 2*x*y + 2*y + 1` with respect to `x` and `(x, y) |--> x^2 + 2*x` with respect to `y`. What if we take the derivative twice?

```
f.diff().diff()
```

we get

```
[    (x, y) |--> 2*y (x, y) |--> 2*x + 2]
[(x, y) |--> 2*x + 2        (x, y) |--> 0]
```

The matrix of second order derivatives of a function in several variables is called the *Hessian*. To compute an element in the matrix we give names of variables as arguments to the diff.

```
f.diff(x).diff(y)
```

and we obtain $\frac{\partial f}{\partial x \partial y}$ as `(x, y) |--> 2*x + 2`.

2.4.2 Numerical Differentiation

Numerical differentiation is available in `scipy.misc`, in the derivative method.

```
from scipy.misc import derivative as numdif
print('exact derivative :', -sin(1.0))
print('1st order approx :', numdif(func=cos, x0=1.0, dx=1.0e-2))
```

and we then see

```
exact derivative : -0.841470984807897
1st order approx : -0.841456960361603
```

The numerical differentiation uses central differences. With extrapolation we can improve the accuracy.

```
ND = [numdif(func=cos, x0=1.0, dx=1.0e-2, order=k) for k in range(3, 12, 2)]
for a in ND: print(a)
print(-sin(1.0))
```

what is printed is below

```
-0.841456960361603
-0.841470984527404
-0.841470984807883
-0.841470984807891
-0.841470984807884
-0.841470984807897
```

2.4.3 Implicit Differentiation

We can declare derivatives symbolically. Assume one variable `y` is a function of another variable `x`.

```
x = var('x')
y = function('y')(x)
dy = y.diff(x)
print(dy, 'has type', type(dy))
```

Then `dy` is a symbolic expression, shown as `diff(y(x),x)`. We need this formal declaration of a function in implicit differentiation.

```
circle = x^2 + y^2 - 1 == 0
print(circle, 'has type', type(circle))
```

That `y` is a function of `x` shows in the display of the equation `x^2 + y(x)^2 - 1 == 0`. This equation defines how `y` depends on `x`. Then we differentiate the equation.

```
dc = circle.diff(x)
dc
```

The equation we now can solve for `dy` is

```
2*y(x)*diff(y(x),x) + 2*x == 0
```

Thus we solve for the derivative.

```
yx = solve(dc, dy)
yx
```

The solution is

```
[
diff(y(x),x) == -x/y(x)
]
```

The right hand side is the formula for the slope of the tangent line at a point on the circle, as shown in Fig. 2.7.

2.4.4 Plotting the Tangent Line

Let us make the plot shown in Fig. 2.7.

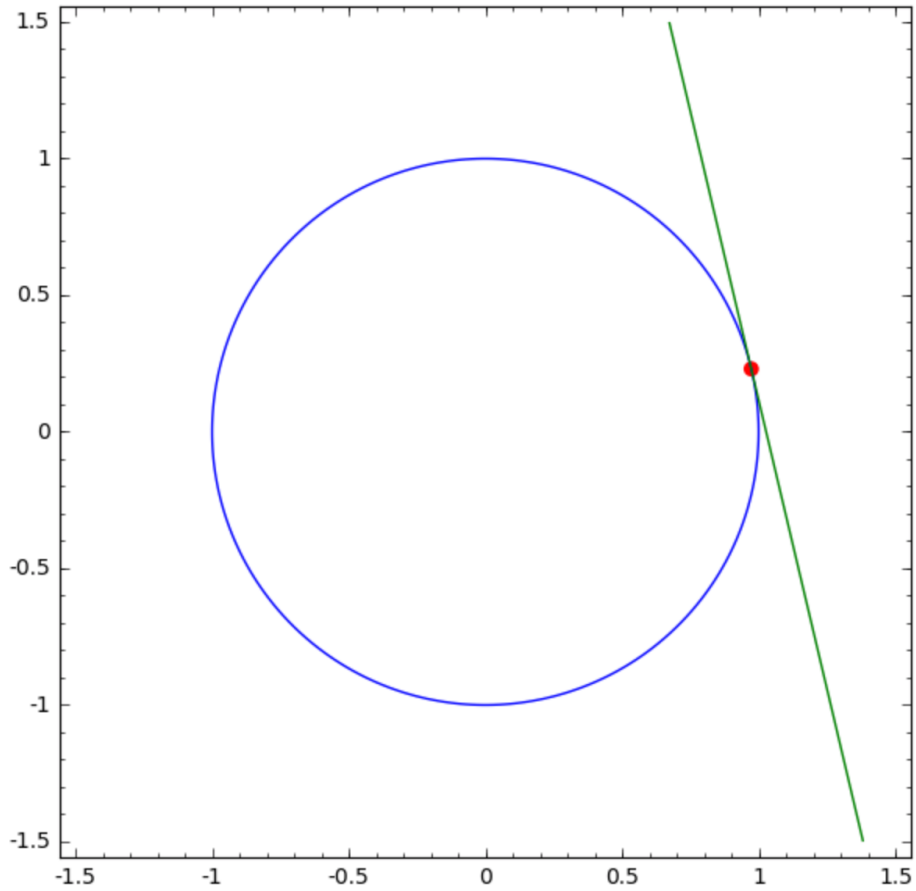

Fig. 2.7: The tangent line to a point on a circle.

We start by selecting a random point on the circle, randomly chosen in the first positive quadrant. Applying the polar representation of the unit circle, as $(x = \cos(\theta), y = \sin(\theta))$ for some angle θ, we generate a random number in the interval $[0, \pi/2]$.

```
angle = RR.random_element(0, pi/2)
circlepoint = (cos(angle), sin(angle))
```

To compute the slope of the tangent line, we first select the formula for the slope, from the list `yx` computed above.

```
slopeformula = yx[0].rhs()
```

The formula for the slope is `-x/y(x)` and we will evaluate this formula at the coordinates of the point on the circle, with coordinates in `circlepoint`. We must apply sequential substitution, first replacing `y(x)` by the value of the y-coordinate of the point, and then replacing `x`. A simultaneous substitution will leave the symbol `y` as a symbol.

```
slope = slopeformula.subs({y(x):circlepoint[1]}).subs(x=circlepoint[0])
```

To formulate the equation of the tangent line, we introduce the new variable `Y` because `y` is already defined.

```
Y = var('Y')
tangentline = Y - circlepoint[1] - slope*(x-circlepoint[0])
```

Then the plotting instructions which yield Fig. 2.7 are below.

```
tangent = implicit_plot(tangentline, (x,-1.5,1.5), (Y,-1.5,1.5), color='green')
circle = implicit_plot(x^2+Y^2 - 1, (x,-1.5,1.5), (Y,-1.5,1.5))
plotpoint = point((circlepoint[0], circlepoint[1]), size=50, color='red')
show(circle+tangent+plotpoint)
```

2.4.5 Assignments

1. Give the SageMath command to compute $\dfrac{\partial^8 f}{\partial^5 x \partial^3 y}$ for $f(x,y) = e^{2x+\cos(y)}$.
2. Consider the curve defined by the equation `3 + 2 x + y + 2 x^2 + 2 x y + 3 y^2 = 0`. Locally we can view `y` as a function of x, that is: `y = y(x)`. Compute formulas for the first and the second derivative of `y` with respect to `x`.
3. Consider the curve defined by the equation $x^3 - y^2 + x = 0$. The point P with coordinates $\left(\dfrac{1}{2}, \dfrac{\sqrt{10}}{4}\right)$ satisfies the equation and thus lies on the curve. Compute the first and second derivatives of the curve evaluated at P. Give the values and the SageMath commands.
4. Consider the plane curve defined by $p(x,y) = x^4y^2 - x^2y^3 + x - y = 0$. Locally at the point $(1,1)$, we view y as a function of x, that is: as $y(x)$.

   Compute $\dfrac{dy}{dx}$. What is the value for $y'(1) = \left.\dfrac{dy}{dx}\right|_{x=1}$?
5. Consider the cissoid of Diocles as defined by $x^3 - (2-x)y^2 = 0$. What is the slope of the tangent line at $(1,1)$?

2.5 Lecture 22: Integration and Summation

Integrals occur everywhere in science and engineering. In case there is no symbolic antiderivative, we can approximate the value for a definite integral with numerical techniques.

2.5.1 Indefinite and Definite Integrals

The counterpart to derivative is integral.

```
integral(cos(x),x)
```

and we see `sin(x)`. We can verify that integration is anti-differentiation and differentiation is anti-integration.

```
print(diff(integral(cos(x),x),x))
print(integral(diff(cos(x),x),x))
```

in both cases we see `cos(x)`. Not every expression has a symbolic antiderivative.

```
e = cos(exp(x^2))
i = integral(e,x)
print('the anti-derivative of', e, 'is', i)
```

and Sage prints

```
the anti-derivative of cos(e^(x^2)) is integrate(cos(e^(x^2)), x)
```

We can look for the definite integral of the expresion.

```
d = integral(e, (x, 0, 1))
show(d)
```

and we see $\int_0^1 \cos\left(e^{(x^2)}\right)\, dx$.

Numerical integration gives an approximation for the definite integral.

```
d.n()
```

and we see `0.112823456937`.

2.5.2 Assisting the Integrator

We can compute definite integrals symbolically.

```
var('a, b')
integral(x^2, (x, a, b))
```

The cubic polynomial `-1/3*a^3 + 1/3*b^3` equals the area under the parabola `y = x^2` for all `x` ranging from `a` till `b`. That does not work always so well. Sometimes we have to assist the integrator.

```
var('a, b')
integral(1/x^2, (x, a, b))
```

The exception triggered is `ValueError` with the comment that `Computation failed since Maxima requested additional constraints; using the 'assume' command before evaluation *may* help (example of legal syntax is 'assume(b-a>0)', see `assume?` for more details)`

Before we continue to examine the exception thrown by Sage, let us not see if we cannot apply the fundamental theorem of calculus: (1) compute the antiderivative; and then (2) make the difference of the antiderivatives evaluated at the end points.

```
ad = integral(1/x^2,x)
print(ad)
print(ad(x=b) - ad(x=a))
```

The antiderivate is `-1/x` and evaluated at the end points, this gives us `1/a - 1/b`. So why does Sage now complain about the original definite integral? The answer is in the following plot, recall that the definite integral gives the area under the curve defined by `1/x^2` for all `x` in `[a,b]`. Let us take `[-1, 1]` for the interval.

```
plot(1/x^2, (x, -1, 1), ymax=100)
```

The parameter `ymax=100` puts a bound on the y-value. The plot is shown in Fig. 2.8.

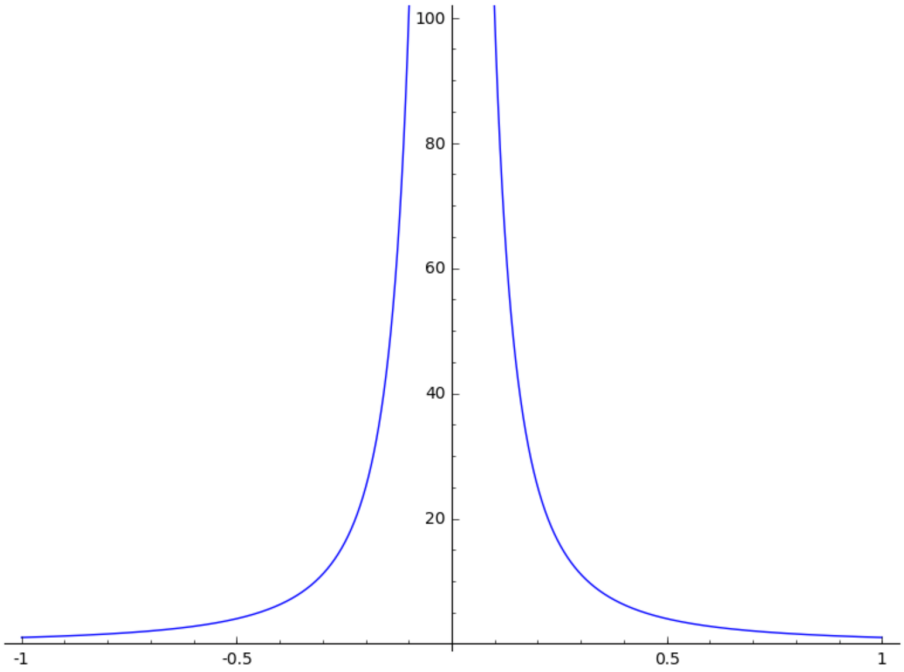

Fig. 2.8: The function `1/x^2` has an asymptote at `x=0`.

The plot shows there is something happening in the interval `[-1, 1]`. Now that we understand that the fundamental theorem of calculus does not apply for just *any* interval `[a,b]`, we can look at the exception Sage raised when we asked for the definite integral.

The question we see in the worksheet is

```
Is b-a positive, negative or zero?
```

We see that Sage asks about the sign of b-a. We can add assumptions to variables.

```
assume(a > 0, b > a)
integral(1/x^2, (x, a, b))
```

and then we have the value `1/a - 1/b`. We can check the assumptions.

```
assumptions()
```

and we then see `[a > 0, b > a]`. Assumptions are removed with `forget`.

```
forget(b > a)
print(assumptions())
forget(a > 0)
print(assumptions())
```

We then see printed:

```
[a > 0]
[]
```

2.5.3 Symbolic Summation

Sage can find explicit expressions for formal sums. Suppose we want to sum k for k going from 1 to n, denoted as $\sum_{k=1}^{n} k$.

```
var('k, n')
s = sum(k, k, 1, n)
print(s)
f = s.factor()
print(f)
```

What is printed is `1/2*n^2 + 1/2*n` and `1/2*(n + 1)*n` With range and the ordinary Python sum, we can verify the formulas, say for n = 100.

```
print(sum(range(1, 101)))
print(f(n = 100))
```

and in both cases we see `5050` as the sum.

We end with a more interesting formal sum. Infinity is expressed as `oo`.

```
sum(1/k^2, k, 1, oo)
```

and thus we see that $\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}$. With list comprehensions, we can verify the symbolic sum with a numerical example.

```
L = [float(1/k^2) for k in range(1, 100001)]
sqrt(6*sum(L))
```

and then we see `3.14158310433`.

2.5.4 Assignments

1. Compute $\int x^n e^x dx$ for a general integer n. Check the result for some randomly chosen values for n.
2. Let F be the *function* defined by $F(T) = \int_1^T \frac{\exp(-t^2 T)}{t} dt$.
   a. Define the Sage function F to compute $F(T)$. Use it to compute a *numerical* approximation for $F(2)$. Compare the approximation for $F(2)$ to the approximation for $\int_1^2 \frac{\exp(-2t^2)}{t} dt$.
   b. Compute the derivative function of F. What is a numerical approximation of $F'(2)$? Compare the value $F'(2)$ to $\frac{F(2+h)-F(2)}{h}$ for sufficiently small values of h.
3. Compute $\int_0^\infty \frac{\ln(x)}{(x+a)(x-1)} dx$ for positive a.
4. Show that $\sum_{k=1}^{n} k^3 = \frac{1}{4} n^2 (n+1)^2$.
5. Compare the results of $\int \left(\int \frac{x-y}{(x+y)^3} dx \right) dy$ with $\int \left(\int \frac{x-y}{(x+y)^3} dy \right) dx$. Why are the expressions different?

   Compute $\int_0^1 \left(\int_0^1 \frac{x-y}{(x+y)^3} dx \right) dy$ with $\int_0^1 \left(\int_0^1 \frac{x-y}{(x+y)^3} dy \right) dx$. Is there a correct value for the integral?

2.6 Lecture 23: Series, Approximations, and Limits

The manipulation of series is a form of hybrid exact-approximation computation. We start with Taylor series and them move to general power series. The accuracy of Taylor series is only local, for a better global approximation we use Padé approximations, which are rational expressions.

2.6.1 Taylor Series

Let us construct a Taylor series for `sin(x)` of order 6 at `x = 0`.

```
tsin = taylor(sin(x), x, 0, 6)
print(tsin, 'has type', type(tsin))
```

and we see `1/120*x^5 - 1/6*x^3 + x` That the result of `taylor` is an expression makes it easy for evaluation. Close to zero, this will approximate the true sine function up to 6-th order.

```
a = tsin(0.01)
b = sin(0.01)
print(a, b, abs(a-b))
```

which shows the numbers `0.00999983333416667`, `0.00999983333416666`, and `1.73472347597681e-18` as the magnitude for the error. We can do this pure formally, on an arbitrary function `f` in `x`, at some point `a`.

```
x, a = var('x, a')
f = function('f')(x)
tf = taylor(f(x), x, a , 4)
print(tf, 'has type', type(tf))
```

so we get the general form of a Taylor expansion of any function `f` in `x` at `a` as

```
1/24*(a - x)^4*D[0, 0, 0, 0](f)(a) - 1/6*(a - x)^3*D[0, 0, 0](f)(a)
+ 1/2*(a - x)^2*D[0, 0](f)(a) - (a - x)*D[0](f)(a) + f(a)
```

The Fibonacci numbers arise as the coefficient of the Taylor series of a particular rational function, called a *generating function*. To see the first 10 Fibonacci numbers, we can do

```
g = x/(1-x-x^2)
tg = taylor(g(x), x, 0, 10)
print(tg)
print(tg.coefficients())
```

The Taylor series is `55*x^10 + 34*x^9 + 21*x^8 + 13*x^7 + 8*x^6 + 5*x^5 + 3*x^4 + 2*x^3 + x^2 + x ``` and the coefficients (with corresponding exponents) is are in the list ``[[1, 1], [1, 2], [2, 3], [3, 4], [5, 5], [8, 6], [13, 7], [21, 8], [34, 9], [55, 10]]``. Taylor series are defined for expressions in several variables as well. For example, consider cos(x)*sin(y) at x = pi/2 and y = 0.

```
x, y = var('x, y')
h = cos(x)*sin(y)
taylor(h,(x,pi/2),(y,0),5)
```

and we get `-1/48*(pi - 2*x)^3*y - 1/12*(pi - 2*x)*y^3 + 1/2*(pi - 2*x)*y`.

2.6.2 Taylor Series in SymPy

Let us see what systems Sage uses to compute the Taylor series.

```
from sage.misc.citation import get_systems
get_systems('taylor(sin(x), x, 0, 6)')
```

and we see `['ginac', 'Maxima']`.

Also SymPy exports Taylor series in a more Pythonic way with generators.

```
from sympy.series import series
series(sin(x), x0=0, n=10)
```

and the tenth order Taylor series for `sin(x)` at zero is `x - x**3/6 + x**5/120 - x**7/5040 + x**9/362880 + O(x**10)`. If instead of `n=10` for the order, we give `None` as argument, we obtain a generator:

```
g = series(sin(x), x0=0, n=None)
print(g, 'has type', type(g))
```

and we see that `g` is a generator object. We can use a generator with the `next()` function to get the next term in the series.

```
print(next(g))
print(next(g))
```

or in a list comprehension to get the next 5 terms in the series

```
[next(g) for k in range(5)]
```

2.6.3 Power Series

In Sage, Taylor series are not power series, but we can turn any polynomial into a power series.

```
x = var('x')
q = x^3 + 3*x + 8
pq = q.power_series(QQ)
print(pq, 'has type', type(pq))
```

and then `pq` is shown as `8 + 3*x + x^3 + O(x^4)` and of type `sage.rings.power_series_poly.PowerSeries_poly`. Why would we want to turn a polynomial into a power series? Well, unlike polynomials, every power series with a nonzero leading constant coefficient has a multiplicative inverse and there is a division operator for power series.

```
ipq = 1/pq
print(ipq, 'has type', type(ipq))
print(pq*ipq)
```

The inverse of the series `pq` leads to `1/8 - 3/64*x + 9/512*x^2 - 91/4096*x^3 + O(x^4)` of the same type as `pq`. The multiplication leads to `1 + O(x^4)`. The order of a series is also known as its precision and we can query the precision by the `prec()` method.

```
print(pq.prec(), ipq.prec())
```

So both series are of precision `4`. Note that the order is not equal to the degree.

```
pq.degree()
```

While the order is `4`, the degree is `3`.

Truncating to a series of lower precision happens with `truncate_powerseries()`.

```
tipq = ipq.truncate_powerseries(2)
print(tipq, 'has type', type(tipq))
```

1/8 - 3/64*x + O(x^2) <type 'sage.rings.power_series_poly.PowerSeries_poly'> We can turn a power series into a polynomial with the `polynomial()` method, or we can truncate to a certain precision.

```
print(ipq.polynomial())
print(ipq.truncate(2))
```

With `polynomial()` we see `-91/4096*x^3 + 9/512*x^2 - 3/64*x + 1/8` and `truncate(2)` gives `-3/64*x + 1/8`.

With `r = p.reverse()` we compute a power series `r` so that when `r` is substituted in `p`, a series of order `k`, we obtain `x + O(x^k)`. The series `p` should not have a nonzero constant term. The coefficients of a series can select with `list`. Of our example `pq`, we substract list(pq)[0] to remove the constant term.

```
print(pq)
print('the coefficients of the series :', list(pq))
pq1 = pq - list(pq)[0]
r = pq1.reverse()
print(r)
```

What is printed as `r` is `1/3*x - 1/81*x^3 + O(x^4)`. Let us verify the reversion of the series by substitution: Note that it works both ways.

```
pq1(x = r)
r(x = pq1)
```

and we see `x + O(x^4)` twice.

2.6.4 Approximations

Padé approximations are rational approximations.

```
z = PowerSeriesRing(QQ, 'z').gen()
pd = exp(z).pade(4,4)
print(pd, 'has type', type(pd))
```

and then we see `(z^4 + 20*z^3 + 180*z^2 + 840*z + 1680)/(z^4 - 20*z^3 + 180*z^2 - 840*z + 1680)`. as an object of the class `sage.rings.fraction_field_element.FractionFieldElement_1poly_field`. How good are the approximations, let us evaluate at `3.14`.

```
print(pd(3.14))
print(exp(3.14))
```

and we see that we have two decimal places correct:

```
23.0681517426949
23.1038668587222
```

Let us compare a power series approximation for $\sin(x)$ with a Padé approximation.

```
tsin = taylor(sin(x),x,0,10)
psin = tsin.power_series(QQ)
tsin
```

The Taylor series of order 10 is $x - 1/6x^3 + 1/120x^5 - 1/5040x^7 + 1/362880x^9 + O(x^{10})$. To use as a polynomial we truncate.

```
polsin = psin.truncate()
```

The polynomial is not such a good approximation for $sin(x)$.

```
sp = plot(sin(x),x,0,2*pi)
pp = plot(polsin(x),x,0,2*pi,color='red')
(sp+pp).show(aspect_ratio=1/4)
```

The plot is shown in Fig. 2.9. Observe that after $x = 5$, the Taylor series diverges fast.

We can compute a rational approximation for $sin(x)$, where degree of numerator and denominator are both of degree 7.

```
z = PowerSeriesRing(QQ, 'z').gen()
pd = sin(z).pade(7,7)
pd
```

The output is `(-479249/18361*z^7 + 7540776/2623*z^5 - 234376560/2623*z^3 + 1644477120/2623*z)/(z^6 + 453960/2623*z^4 + 39702960/2623*z^2 + 1644477120/2623)`.

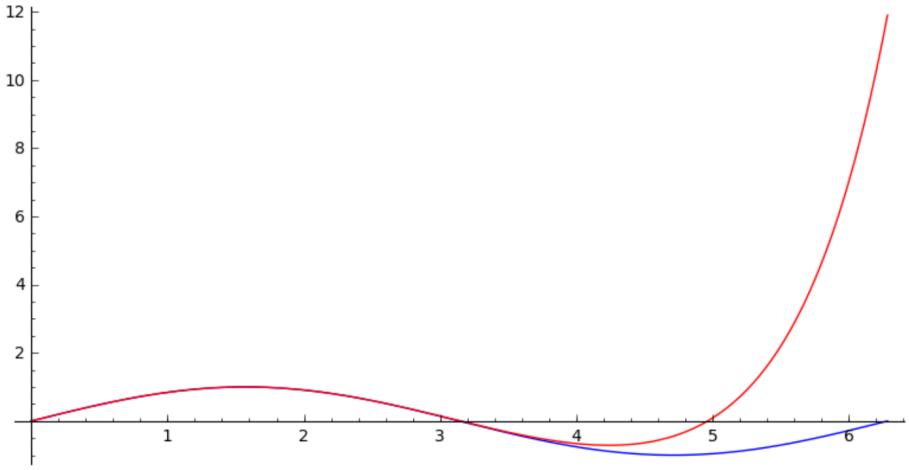

Fig. 2.9: The plot of the sine function and a 10-th order Taylor approximation.

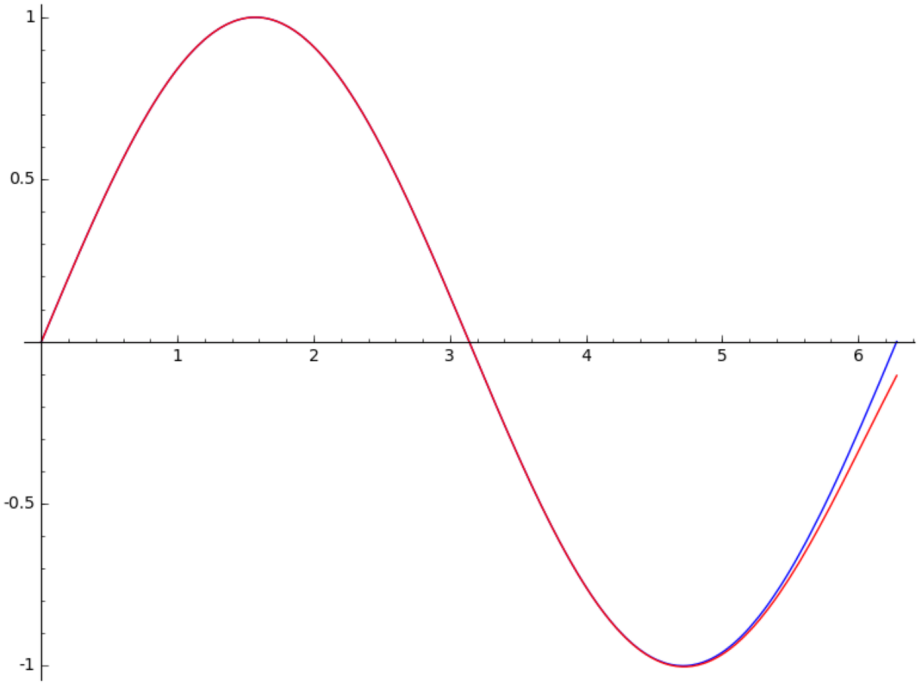

Fig. 2.10: The plot of the sine function and a 7/7-Padé approximation.

2.6.5 Limits

We can take limits with the limit command. For example, consider the limit definition of the transcendental constant $e = \lim_{k \to \infty} \left(1 + \frac{1}{k}\right)^k$.

```
var('k')
f = (1+1/k)^k
f.limit(k=oo)
```

and as output we see `e`.

2.6.6 Assignments

1. Make a 10-th order Taylor series of $\arctan(x)$ about $x = 0$. Compute the difference between the value of Taylor series evaluated at 0.3 and $\arctan(0.3)$.
2. Use the series method of SymPy to make a generator for the Taylor series of `x/(1-x-x^2)` that has the Fibonacci numbers as coefficients. Apply the generator in a list comprehension to compute the first 50 Fibonacci numbers.
3. Use Sage to verify that $\frac{1}{\sqrt{1-4x}} = \sum_{n \geq 0} \binom{2n}{n} x^n$ in the following way:
   (a) Compute a 7-th order Taylor series of of $\frac{1}{\sqrt{1-4x}}$ about $x = 0$.
   (b) Compute $\sum_{n \geq 0} \binom{2n}{n} x^n$ for $n = 7$.
4. The binomial coefficient $\binom{n+k}{k}$ counts all choices of k elements from a set of $n + k$ elements and can be defined via a Taylor series expansion of $g(z) = \frac{1}{(1-z)^{n+1}}$ at $z = 0$. Verify this definition for $n = 10$ and all values for k ranging from 0 to 10.
5. The Catalan numbers arise as the coefficients in the Taylor expansion of the generating function $\frac{1 - \sqrt{1-4x}}{2x}$.
   Give the Sage command to compute a Taylor series of the generating function at $x = 0$ of order 10. Compare with the output of `catalan_number(10)`.
6. Compare the accuracy of a Padé approximation for $\sin(x)$ of (4,4) with the accuracy of a fourth order Taylor series for $\sin(x)$. Compare with a plot of $\sin(x)$ on the interval [0, 2].

2.7 Lecture 24: Symbolic-Numeric Computation

With interval arithmetic we compute floating-point approximations *and* estimates for the errors. The manipulation of symbolic expressions is used to formulate equations to solve, which we illustrate via the application of the Lagrange multiplier technique to a constrained optimization problem.

2.7.1 Interval arithmetic

There is one important type of arithmetic we have not covered yet, and that is interval arithmetic.

```
R = RealIntervalField()
R
```

and R is a `Real Interval Field with 53 bits of precision`. By default, the `RealIntervalField()` without argument (also abbreviated as RIF) works with double precision floating-point arithmetic. We can coerce or convert into this field.

```
api = R(pi)
print(api, 'has type', type(api))
print(api.str(style='brackets'))
```

and we respectively see `3.141592653589794?` (observe the question mark at the end) and the interval `[3.1415926535897931 .. 3.1415926535897936]` as a `RealIntervalFieldElement` of `sage.rings.real_mpfi`.

Interval arithmetic provides operations to add, substract, multiply, and divide intervals. The width of the interval gives an upper bound on the roundoff error. With interval arithmetic we compute at the same time a floating-point approximation and an estimate for the error on this approximation.

To motivate the use of interval arithmetic, we take an example from the paper of Stefano Taschini in the proceedings of SciPy 2008. The problem is to evaluate

$$f(x,y) = (333.75 - x^2)y^6 + x^2(11x^2y^2 - 121y^4 - 2) + 5.5y^8 + x/(2y)$$

at $(77617, 33096)$.

To calculate the exact value, we convert the floating-point coefficients to rational numbers.

```
F(x,y) = (QQ(333.75) - x**2)*y**6 + x**2*(11*x**2*y**2 - 121*y**4 - 2)
+ QQ(5.5)*y**8 + x/(2*y)
print F
(A, B) = (77617, 33096)
exact = F(A, B)
print('the exact value :', exact, RDF(exact))
```

and the exact value is `-54767/66192` or approximated in double precision `-0.827396059947`. With increasing multiprecision, we may gain more confidence in the results.

```
f53 = F(A.n(53), B.n(53))
print(f53)
for k in range(54, 105, 10):
    print(F(A.n(k), B.n(k)))
```

and we see

```
3.62381792172646e20
-8.18209863729137e20
3.64375473385373696e17
1.46852011835392000000e14
6.72051363840000000000000e10
9.62723840000000000000000000e7
-196610.000000000000000000000000
```

Let us continue.

```
for k in range(100, 130, 5): print(F(A.n(k), B.n(k)))
```

and then we get the following:

```
3.997694000000000000000000000e6
65534.000000000000000000000000000
-2.000000000000000000000000000000000
-2.000000000000000000000000000000000
-2.000000000000000000000000000000000
-0.750000000000000000000000000000000000
```

The sequence of repeated `-2.0` values is especially troublesome as one would reach the wrong conclusion that the precision is enough once the sequence has stabilized. This trial and error process is not really encouraging…

```
R = RealIntervalField(130)
z = F(R(A), R(B))
print(R(z).str(style='brackets'))
```

and we get

```
[-0.828125000000000000000000000000000000000000 ..
 -0.820312500000000000000000000000000000000000]
```

2.7.2 Symbolic-Numeric Factorization

In a numeric factorization of a polynomial in one variable, we compute the complex roots of the polynomial. In a symbolic factorization, we extend the field of rational numbers till the polynomial can be written as a product of linear factors. A symbolic-numeric factorization uses the *complex interval field*, that is: the roots of the polynomial are computed with complex interval arithmetic in a specific precision. In SageMath, a symbolic-numeric factorization is computed using the `QQbar` set of numbers.

```
P.<x> = PolynomialRing(QQbar)
```

The above statement defines `P` as the univariate polynomial ring in `x` over the algebraic field. Any expression in `x` will be interpreted as belonging to `P`, as is the case with `q` below.

```
q = x^3 + x + 1
factor(q)
```

We see the three factors of `q` with the roots written in the interval notation and the `I` imaginary unit. To see the roots of `q` over `QQbar` we can do

```
q.roots(QQbar)
```

and we same the same intervals as in the output of `factor(q)`.

If we want a more accurate symbolic-numeric factorization, for example with 200 bits of precision, then we first define a new complex interval field before computing the roots in that new defined field, as done by the code below.

```
C = ComplexIntervalField(prec=200)
q.roots(C)
```

Grabbing the output of `q.roots(C)` defines then the factors in a symbolic-numeric factorization. Computing the product shows the *backward error* of this computation, that is: the polynomial near to `q` for which `q.roots(C)` are the exact roots.

2.7.3 Constrained Optimization

The method of Lagrange multipliers allows to solve a constrained optimization problem.

```
var('x,y,z')
target = x^2 + 2*x*y*z - z^2
constraint = x^2 + y^2 + z^2 - 1 == 0
print('optimize', target, 'subject to', constraint)
```

and we get the printed message `optimize 2*x*y*z + x^2 - z^2 subject to x^2 + y^2 + z^2 - 1 == 0`. To apply the method of Lagrange multipliers, we need to compute the gradients. For this, we convert the expressions into functions. Observe that the constraint was given as an equation, so we must select the left hand side of the equation when we convert the expression to a function.

If we want that the target evaluates at 2, then we see there is no solution, see Fig. 2.11 made with the commands below.

```
tplot = implicit_plot3d(target-2, (x,-2,2), (y,-2,2), (z,-2,2), color='red')
cplot = implicit_plot3d(constraint.lhs(), (x,-2,2), (y,-2,2), (z,-2,2))
(tplot+cplot).show()
```

If we want solutions, then the contraint and target must touch, which is the case when the value of the target equals one. Let us make the plot, see Fig. 2.12.

```
tplot = implicit_plot3d(target-1, (x,-2,2), (y,-2,2), (z,-2,2), color='red')
cplot = implicit_plot3d(constraint.lhs(), (x,-2,2), (y,-2,2), (z,-2,2))
(tplot+cplot).show()
```

```
ft(x,y,z) = target
fc(x,y,z) = constraint.lhs()
print('target as function :', ft)
print('constraint as function :', fc)
gt = ft.diff()
gc = fc.diff()
print('gradient of target :', gt)
print('gradient of constraint :', gc)
```

and the results are

```
target as function : (x, y, z) |--> 2*x*y*z + x^2 - z^2
constraint as function : (x, y, z) |--> x^2 + y^2 + z^2 - 1
gradient of target : (x, y, z) |--> (2*y*z + 2*x, 2*x*z, 2*x*y - 2*z)
gradient of constraint : (x, y, z) |--> (2*x, 2*y, 2*z)
```

Now we set up the system.

```
w = var('w')
eqs = [gt(x,y,z)[k] - w*gc(x,y,z)[k] == 0 for k in range(3)]
eqs.append(constraint)
eqs
```

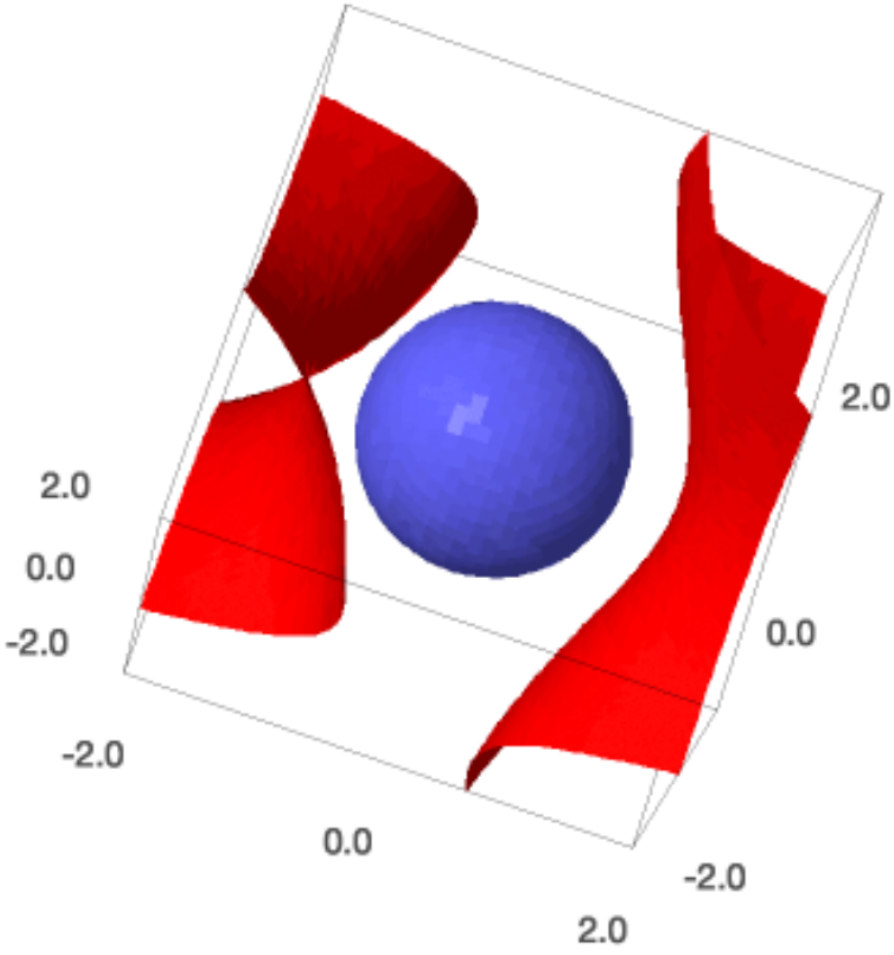

Fig. 2.11: The blue sphere (constraint) does not touch the red sheets (target at 2).

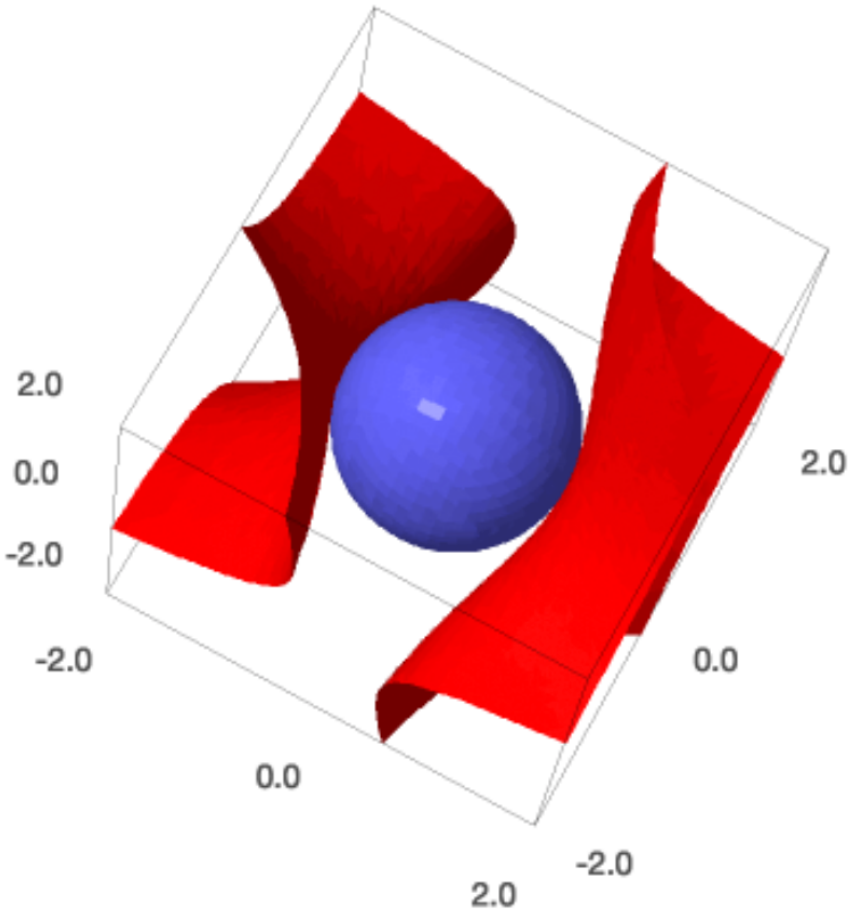

Fig. 2.12: The blue sphere (constraint) touches the red sheets (target at 1).

The polynomial system we obtain is thus

```
[-2*w*x + 2*y*z + 2*x == 0, -2*w*y + 2*x*z == 0,
2*x*y - 2*w*z - 2*z == 0, x^2 + y^2 + z^2 - 1 == 0]
```

and then we just do solve…

```
sols = solve(eqs,[x,y,z,w])
print(sols)
print('found', len(sols), 'solutions')
```

to see that we found the candidate optimal solutions.

2.7.4 Assignments

1. Do `x = RIF(-1, +1); print 1/x`. Interpret the result.
2. Compare the evaluation of $p = x^2 - 4x$ and $q = x(x-4)$ at the interval $[1, 4]$. Which form of the expression, p or q leads to the most accurate result? Justify your answer.
3. Find those points on the surface $x^2 - xy + y^2 - z^2 = 1$ closest to the origin. Set up the system with Lagrange multipliers and solve.
4. Consider the surface defined by $g(x, y) = x^2 + 3xy + 2y^2 + 7x = 0$. Find those points (x, y) that satisfy $g(x, y) = 0$ and that are closest to the origin.
   (a) Define this problem as a polynomial system.
   (b) Bring this system in triangular form. How many solutions can this system have? Justify your answer.
   (c) Find all *real* solutions.
5. Consider the function $f(x) = 1 - 10x + 0.01e^x$. Use interval arithmetic to show that $f(x) = 0$ has a solution in the interval $[9, 10]$.
6. Compute a symbolic-numeric factorization of $p = x^4 + 3x^3 + 2x^2 + x + 9$ with 20 bits of precision. Use the brackets style to display the roots. What is backward error?

CHAPTER 3

Part IV: Plotting and Solving Equations

In this part we start to use SageMath for plotting and to solve problems. We model a mathematical problem, apply advanced solvers, and visualize the setup and the computational results. This part of the course ends with the second midterm exam.

3.1 Lecture 25: Two Dimensional Plots

Plotting is one of the main strengths of computer algebra systems. We distinguish between plotting formulas and functions. Similar to function and expressions in one variable, curves in the plane are best plotted when given in parametric form. For curves with singularities at the origin, the conversion into a polar form may lead to much better results, because then we can factor out the singular point in the polar formulation.

A plane curve may be given in four different ways. Which command to use depends on the definition of the curve.

1. As the graph of a function $y = f(x)$: use `plot`.
2. Implicitly, as an equation $f(x, y) = 0$, use `implicit_plot`.
3. In parametric form, as $(x(t), y(t))$: use `parametric_plot`.
4. In polar coordinates, as $r = f(t)$: use `polar_plot`.

3.1.1 Plotting Formulas and Functions

The `plot` command is for expressions or functions in one variable, that is: we want to see all points (x, y) defined by the function $y = f(x)$, for all x in some interval. Note there is a difference in syntax between plotting expressions and functions. We first plot an expression.

```
x = var('x')
ex = exp(-x^2)*sin(pi*x^3)
plot(ex, x, -2, 2, figsize=4)
```

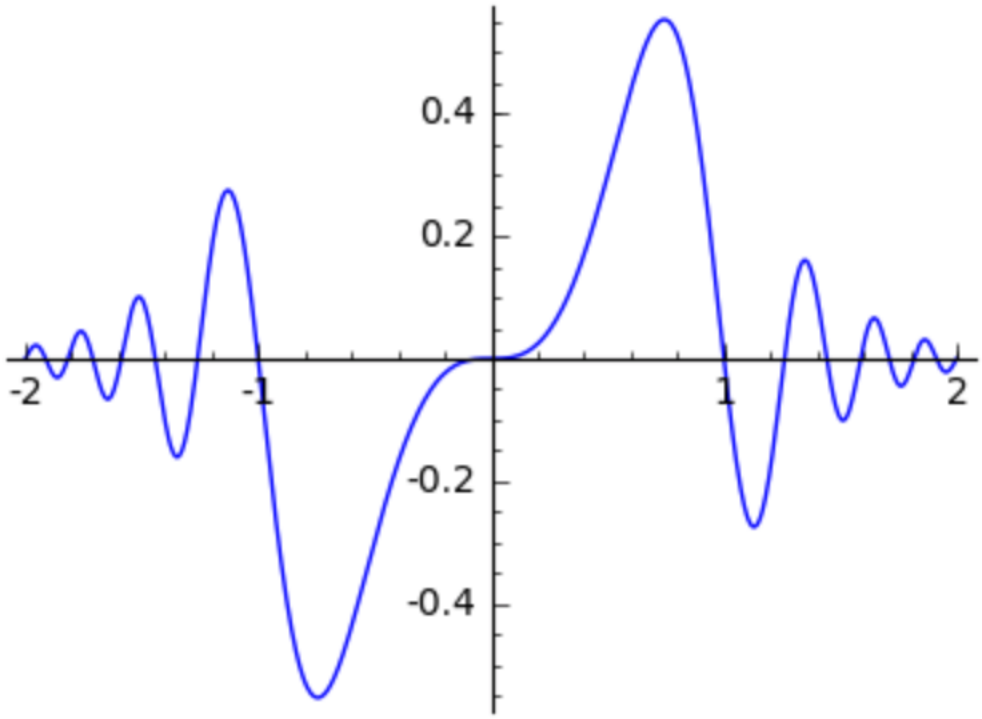

Fig. 3.1: An example plot of $\exp(-x^2)\sin(\pi x^3)$.

The plot produces the plot in Fig. 3.1.

With a function we do not need to specify the variable name.

```
ef(x) = ex
plot(ef, -2, 2, figsize=4)
```

The above plot also shows Fig. 3.1. We can save plots for later viewing. Another application is to overlay several plots. For the expression above, we will compute the amplitudes separately, as shown in Fig. 3.2.

```
aplus = plot(exp(-x^2), x, -2, 2, color='red')
amin = plot(-exp(-x^2), x, -2, 2, color='green')
explot = plot(ex, x, -2, 2)
graph = aplus + amin + explot
graph.show(figsize=4)
```

We already use plot on a function with an asymptote (1/x^2, remember?) and that was not pretty, but it did what we wanted to see. For a more pretty picture of a function with an asymptote we can provide extra arguments of plot to limit the viewing window with `ymax` and `ymin`.

```
plot(1/x^2, -2, +2, ymax = 4, figsize=4)
```

The plot is shown in Fig. 3.3.

A pole is a root of the denominator, which gives rise to a vertical asymptote. Setting the option detect_poles to True shows no vertical lines. If instead of True we give 'show' as value for the option, we see the vertical asymptotes.

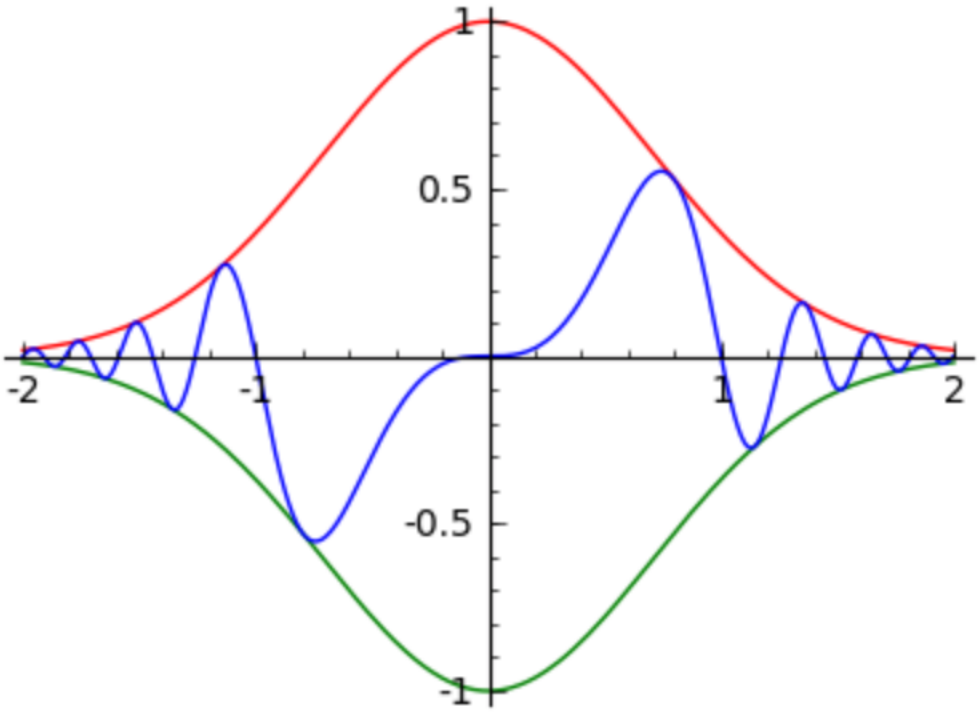

Fig. 3.2: The curve $\exp(-x^2)\sin(\pi x^3)$ with amplitudes $\exp(-x^2)$ and $-\exp(-x^2)$.

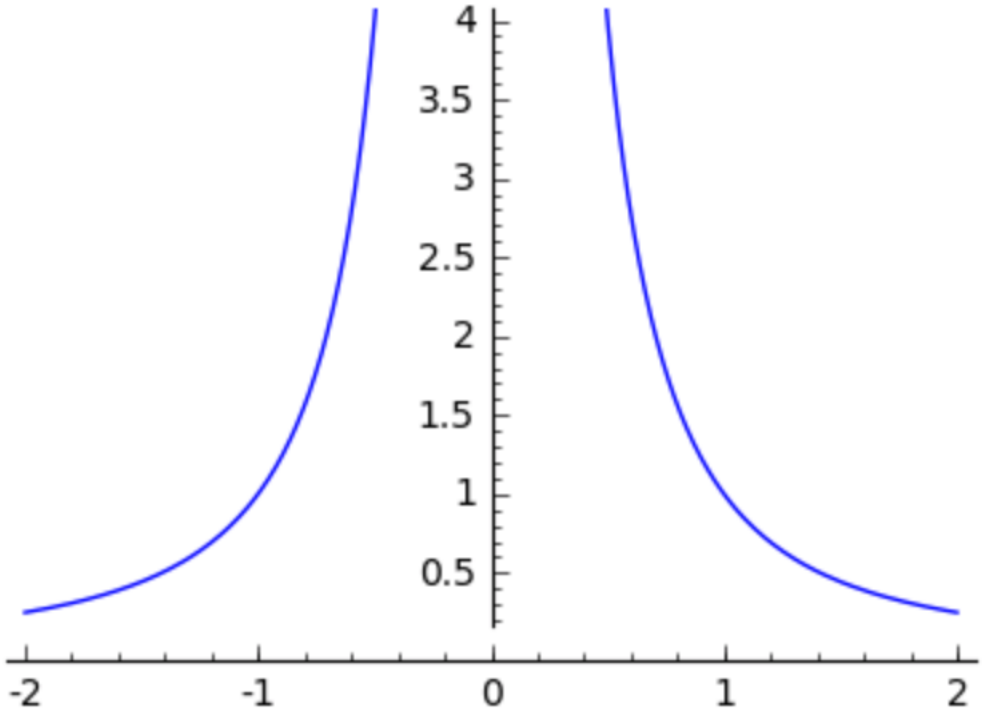

Fig. 3.3: A plot of an asymptote with a limiting viewing window.

```
plot(1/(x^3 - x),-2,2,ymin=-5,ymax=5,detect_poles='show',figsize=4)
```

In Fig. 3.4 we see the vertical asymptotes in the plot.

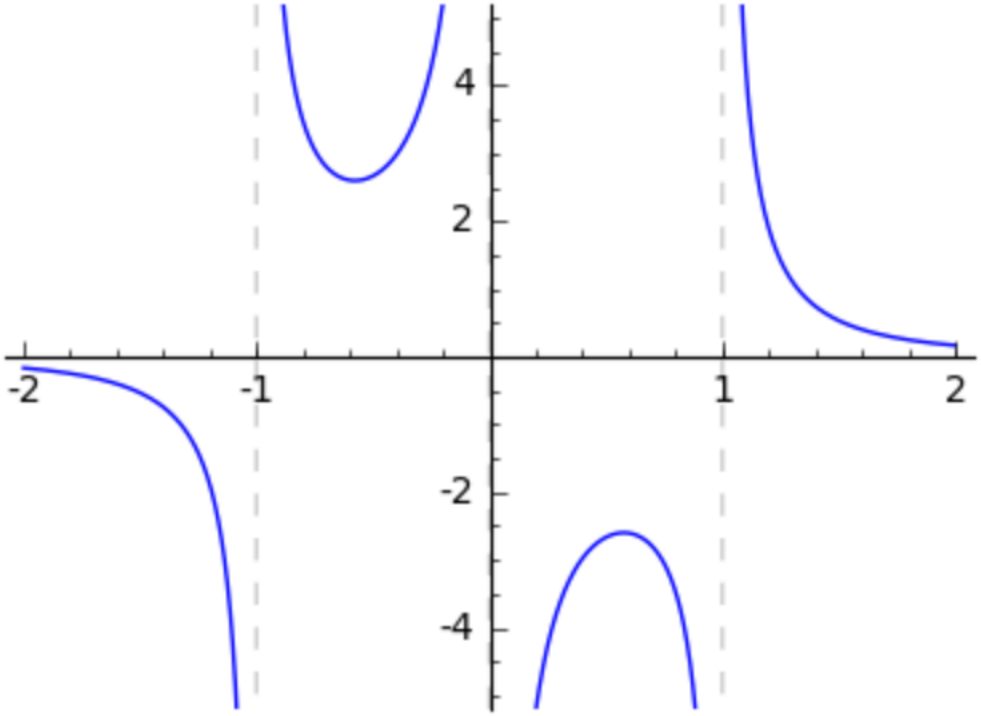

Fig. 3.4: A plot of a function with multple poles.

3.1.2 Curves in the Plane

Some curves are given by explicit representations for the coordinates $(x(t), y(t))$, as functions in some variable t. We plot such representations via `parametric_plot`. The Lissajous curve is defined in a parametric way, shown in Fig. 3.5.

The plot command is below.

```
t = var('t')
parametric_plot((sin(2*t), sin(3*t)), (t, 0, 2*pi), figsize=4)
```

Some curves may be given implicitly, via an equation $f(x, y) = 0$. For the implicit forms, we use the `implicit_plot`. The curve below is "invented" by James Watt, first shown in Fig. 3.6 and Fig. 3.7.

```
x, y = var('x, y')
f = (x^2+y^2)^3 + 5.12*(x^2+y^2)^2 - 5.15*(x^4-y^4) - 14.7456*y^2
implicit_plot(f, (x, -2, 2), (y, -2, 2)).show(figsize=4)
implicit_plot(f, (x, -2, 2), (y, -2, 2), plot_points = 2000, \
    color='red').show(figsize=4)
```

Although the plot made with 2000 points looks very good, we might get the impression that the curve passes twice through the origin.

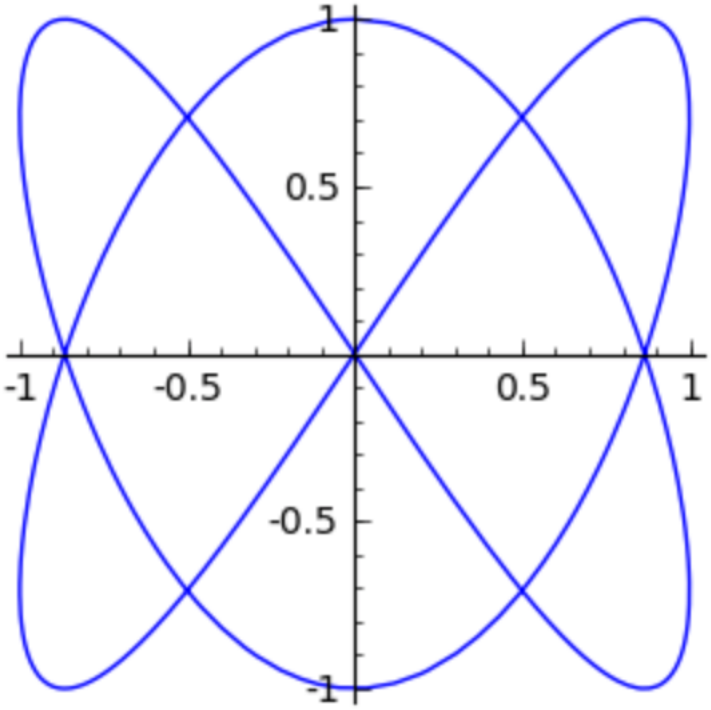

Fig. 3.5: A plot of a Lissajous curve.

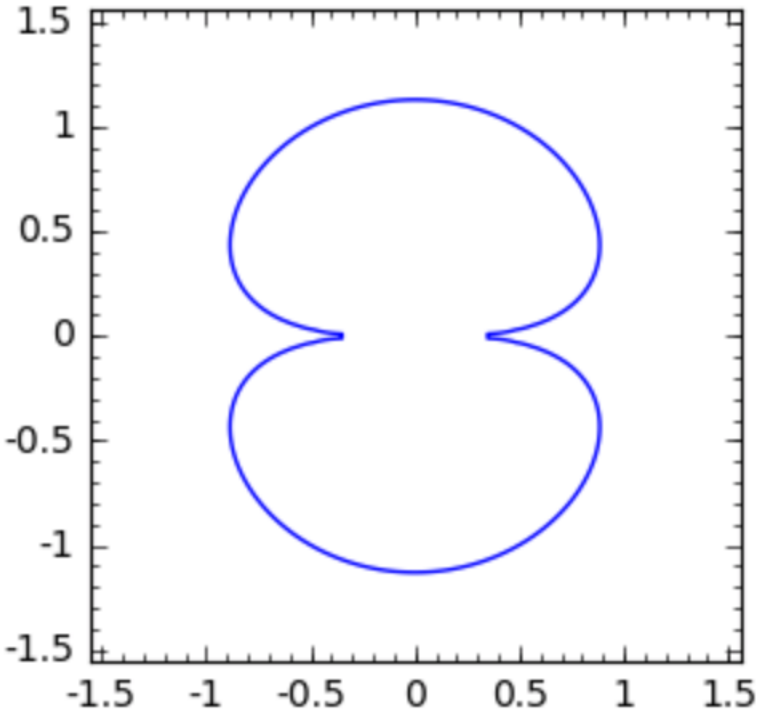

Fig. 3.6: An implicit plot of $f = (x^2 + y^2)^3 + 5.12(x^2 + y^2)^2 - 5.15(x^4 - y^4) - 14.7456y^2$.

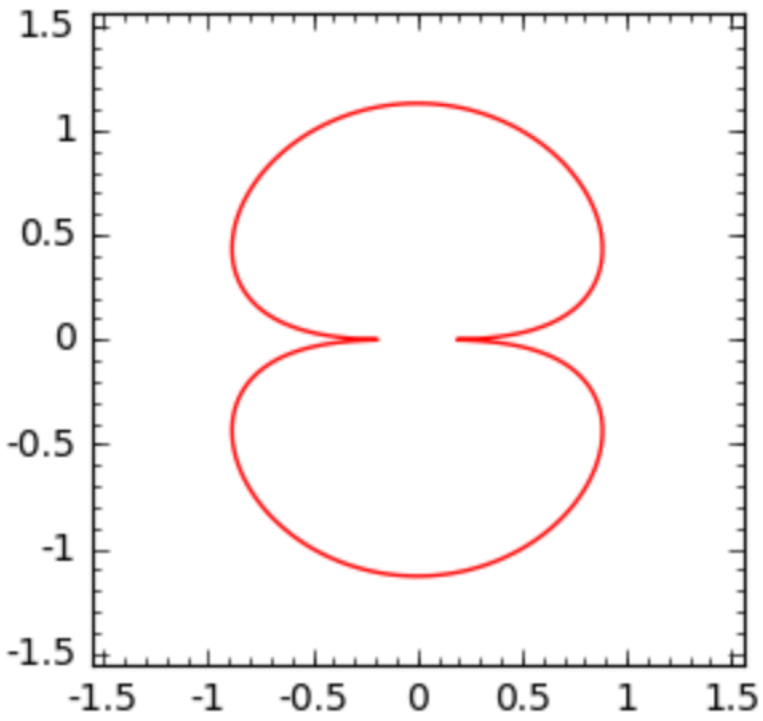

Fig. 3.7: Another implicit plot of $f = (x^2+y^2)^3 + 5.12(x^2+y^2)^2 - 5.15(x^4-y^4) - 14.7456y^2$, using 2000 plotting points.

We will convert to polar coordinates. With polar coordinates, every point is represented by a radius and an angle. The radius is the distance of the point to the origin. The angle is the angle the vector ending at the point makes with the horizontal axis.

```
r, t = var('r, t')
g = f.subs({x: r*cos(t), y:r*sin(t)})
g
```

Before solving this expression for r, it is good to simplify.

```
sg = g.trig_simplify()
sg
```

Even without factoring we see that r = 0 is a double root.

```
s = solve(sg,r)
```

We select the first solution and plot the two parts in different colors. For curves given in polar coordinates, as $r = f(t)$, we use `polar_plot`, providing a range for t.

```
s0 = s[0].rhs()
first = polar_plot(s0, (t, 0, pi), axes=False, color='red')
second = polar_plot(s0, (t, pi, 2*pi), axes=False, color='green')
(first+second).show(figsize=4)
```

The polar plot is shown in Fig. 3.8.

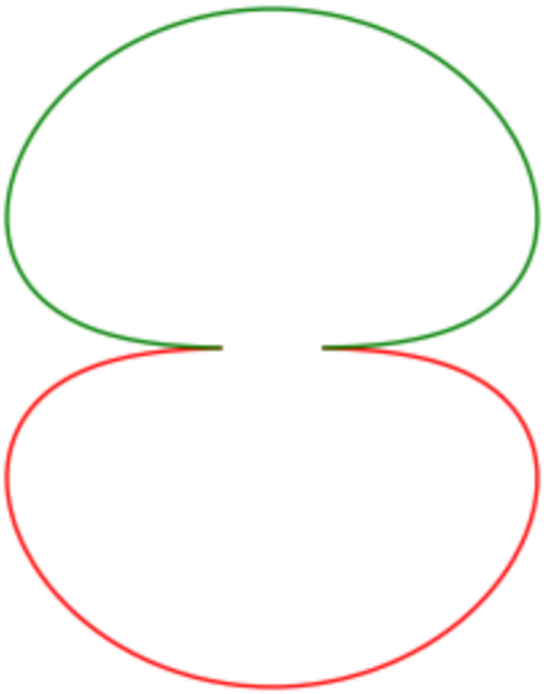

Fig. 3.8: The two halves of the polar plot, for $t \in [0, \pi]$ and for $t \in [\pi, 2\pi]$.

Is that all? Well, did we not forget the r = 0 component? The point (0, 0) is an isolated singularity of the real curve.

```
p0 = point((0, 0), color='blue', size=50)
(p0+first+second).show()
```

The complete polar plot is shown in Fig. 3.9.

3.1.3 Assignments

1. The *neoid* is defined by $r = at + b$ in polar coordinates. Make a plot for $a = 0.2$ and $b = 0.5$, for $t = 0 \ldots 6\pi$.
2. Consider the curve defined by $f(x, y) = 4xy^2 + 2x^3 - x^2 = 0$ and use SageMath
   (a) to convert the equation $f(x, y) = 0$ from rectangular into polar coordinates;
   (b) to solve the equation obtained in (a) for the radius;
   (c) to make the plot using the solution(s) obtained in (b).
3. The folium of Descartes is defined by $x^3 + y^3 = 3xy$ and has tangent $x + y + 1 = 0$.
   (a) Draw the folium in rectangular coordinates with enough points in the plot, high enough to see a nice smooth curve around the origin. Take $[-2, +2]$ as the range for both x and y.
   (b) Make a plot for the tangent $x + y + 1 = 0$, using a color different from the plot of the folium. Display both plots on the same figure window. You may want to extend the original range $[-2, +2]$ to see the folium better approaching the tangent.
   (c) Convert the folium into polar coordinates and plot. Be careful in choosing a good range for the parameter t. Compare the number of plot points you needed in (a) with the number of plot points that are needed in

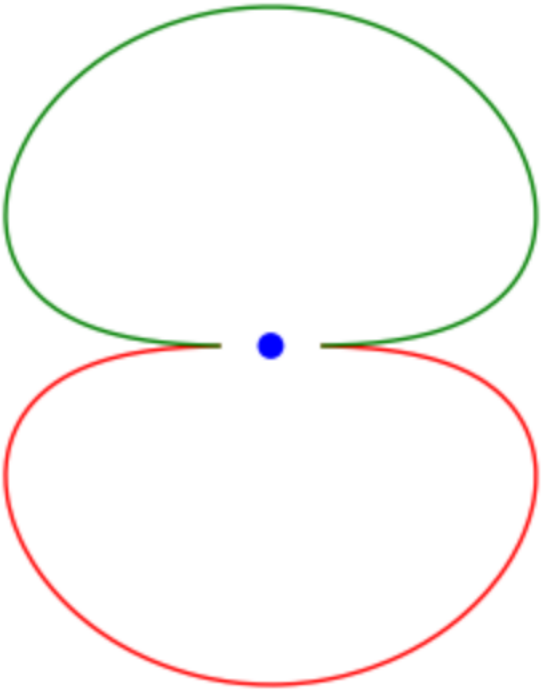

Fig. 3.9: The complete polar plot with the origin added as a point.

the polar form.

4. The snail of Pascal is the curve defined as $(x^2 + y^2 + 2y)^2 - (x^2 + y^2) = 0$.

   (a) Plot this curve for $x \in [-2, +2]$ and $y \in [-3, +1]$. How many points do you need for the plotted curve to have no gaps?

   (b) Compute the polar coordinates for the curve. Plot this curve in polar coordinates.

5. Cassini ovals are sets of points for which the product of the distances to two fixed points are constant. Taking the points on the x-axis with coordinates $(-a, 0)$ and $(+a, 0)$, then for a constant c, the Cassini ovals satisfy $((x - a)^2 + y^2) \times ((x + a)^2 + y^2) = c^4$.

   (a) Make several plots for various choices for a and c. Distinguish the cases $0 < a < c$ and $c < a < c^2$.

   (b) Find the representation of these curves in polar coordinates.

   (c) Concerning the quality of the plots, is it easier for SageMath to work with the original rectangular coordinates or are the polar coordinates better suited?

6. Consider the curve defined by the equation $f(x, y) = (x^2 + y^2)^5 - 16x^2y^2(x^2 - y^2)^2 = 0$.

   (a) Give the command to make a plot of this curve, for x and y in the range from -1 to $+1$. How many points do you need to obtain a good plot?

   (b) Give the commands to transform this curve into polar coordinates and to plot the curve. How many times does the curve pass through (0,0)?

3.2 Lecture 26: Plotting in Three Dimensions and Beyond

The default installation of SageMath may not come with the Java code (Jmol) needed to manipulate graphics objects. A work around (is needed on Windows computers) is to use `tachyon` as the viewer. In particular, if `p` is the result of a plotting command, render the plot `p` as

```
show(p, viewer='tachyon')
```

Of course, as always, the SageMath Cell Server and CoCalc are equally valid alternatives.

In three dimensions, we distinguish between surfaces and space curves. A surface may be given in three different ways. Which command to use depends on the definition of the surface.

1. Given as a function $z = f(x, y)$, use `plot3d`.
2. Implicitly as an equation $f(x, y, z) = 0$, use `implicit_plot3d`.
3. In parameter form as $(x(s, t), y(s, t), z(s, t))$, use `parametric_plot3d`.

Space curves may be given in two different ways. Which command to use depends on the definition of the space curve.

1. In parameter form as $(x(t), y(t), z(t)$, use `parametric_plot3d`.
2. Implicitly, as the intersection of two equations $f(x, y, z) = 0$ and $g(x, y, z) = 0$, use `implicit_plot3d` twice.

Observe there are no special commands for space curves, the `parametric_plot3d` and `implicit_plot3d` are used differently.

3.2.1 Surface Plots

A surface can be defined as $z = f(x, y)$, where for each point in the plane with coordinates (x, y) the corresponding height z is defined by an expression or function in x and y. In this case we use `plot3d`. For example, to plot the surface defined by $z = \cos(xy)$, for $(x, y) \in [-\pi, +\pi] \times [-\pi, +\pi]$ we do:

```
x, y = var('x,y')
plot3d(cos(x*y), (x, -pi, pi), (y, -pi, pi))
```

The plot is shown in Fig. 3.10.

We observe that the plot is rather flat and this is because the values for x and y range from -3.14 to $+3.14$, where as the $\cos(\cdot)$ takes values between -1 and $+1$. To make the plot less flat, we change the `aspect_ratio` and we make the plot spin, as follows:

```
plot3d(cos(x*y), (x, -pi, pi), (y, -pi, pi), \
    aspect_ratio=(1,1,2), spin=1)
```

We can change the viewing angle with the methods `rotateX()` and `rotateZ()` for example.

```
plot3d(cos(x*y), (x, -pi, pi), (y, -pi, pi), \
    aspect_ratio=(1,1,2)).rotateX(pi/2).rotateZ(pi/4)
```

The result of the rotation of the viewing angles is shown in Fig. 3.11.

By default, the value for opacity is 1, we can make the plot more transparant by lowering that value. We can also increase the number of plot points.

```
plot3d(cos(x*y), (x, -pi, pi), (y, -pi, pi), aspect_ratio=(1,1,2), \
    plot_points=[60,60], opacity=0.75).rotateX(pi/10).rotateY(-pi/10)
```

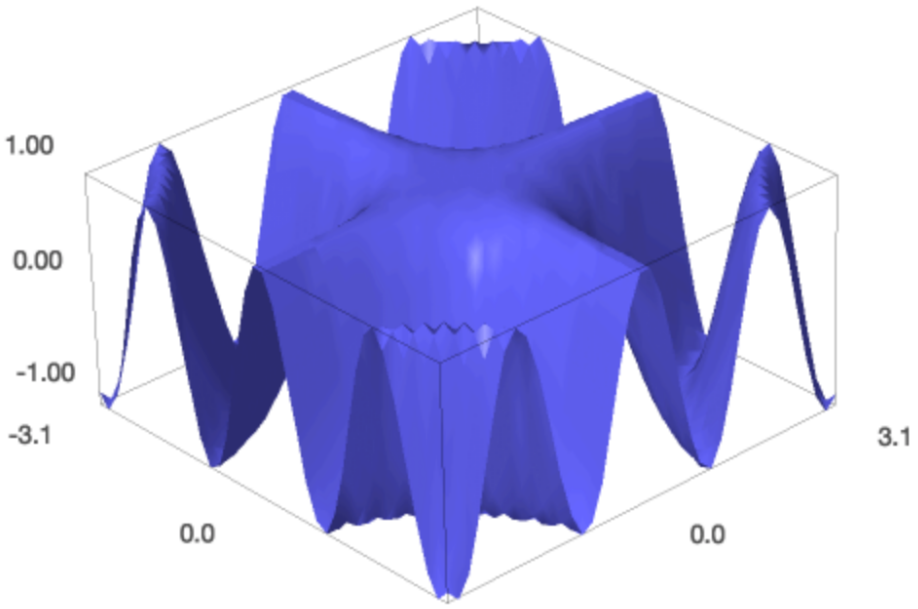

Fig. 3.10: The plot of the surface $z = \cos(xy)$.

See Fig. 3.12 for the corresponding picture.

Instead of trying to guess the best number of plot points, we can set `adaptive` to `True`. Setting color to `'automatic'` chooses a rainbow of colors, we can set the number of colors with `num_colors`.

```
plot3d(cos(x*y), (x, -pi, pi), (y, -pi, pi), adaptive=True, \
    aspect_ratio=(1,1,2), color='automatic', num_colors=256)
```

The colorful plot is shown in Fig. 3.13.

Surfaces may be given in parametric form by three expressions in two variables, say u and v, as $x = f_x(u,v), y = f_y(u,v)$, and $z = f_z(u,v)$. Then we use `parametric_plot3d`. The example below comes from the SageMath documentation.

```
u, v = var('u, v')
fx = cos(u)*(4*sqrt(1-v^2)*sin(abs(u))^abs(u))
fy = sin(u) *(4*sqrt(1-v^2)*sin(abs(u))^abs(u))
fz = v
heart = parametric_plot3d([fx, fy, fz], (u, -pi, pi), (v, -1, 1), \
    color='red').rotateX(-pi/9).rotateY(-pi/2).rotateZ(pi/4)
show(heart, frame=False)
```

Observe how we can prevent the bounding box from appearing. The surface is shown in Fig. 3.14.

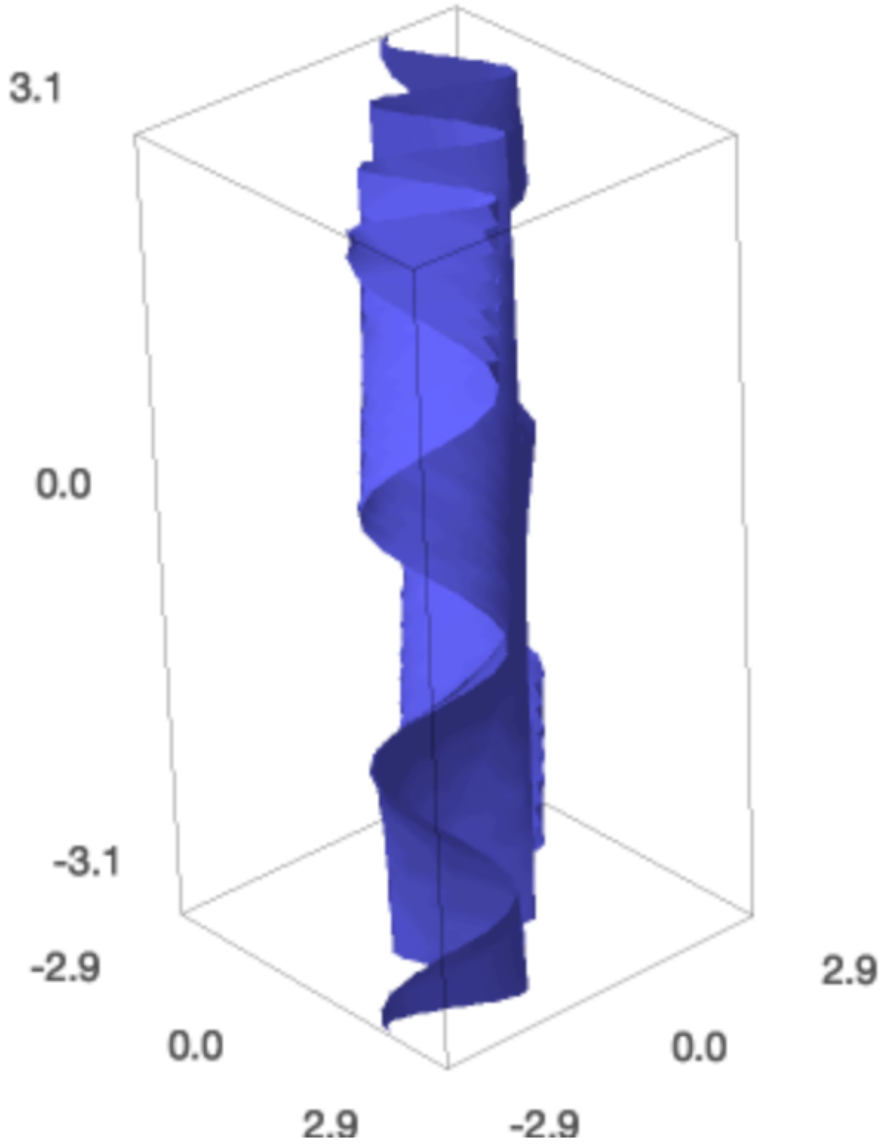

Fig. 3.11: The plot of the surface $z = \cos(xy)$ with adjusted viewing angles.

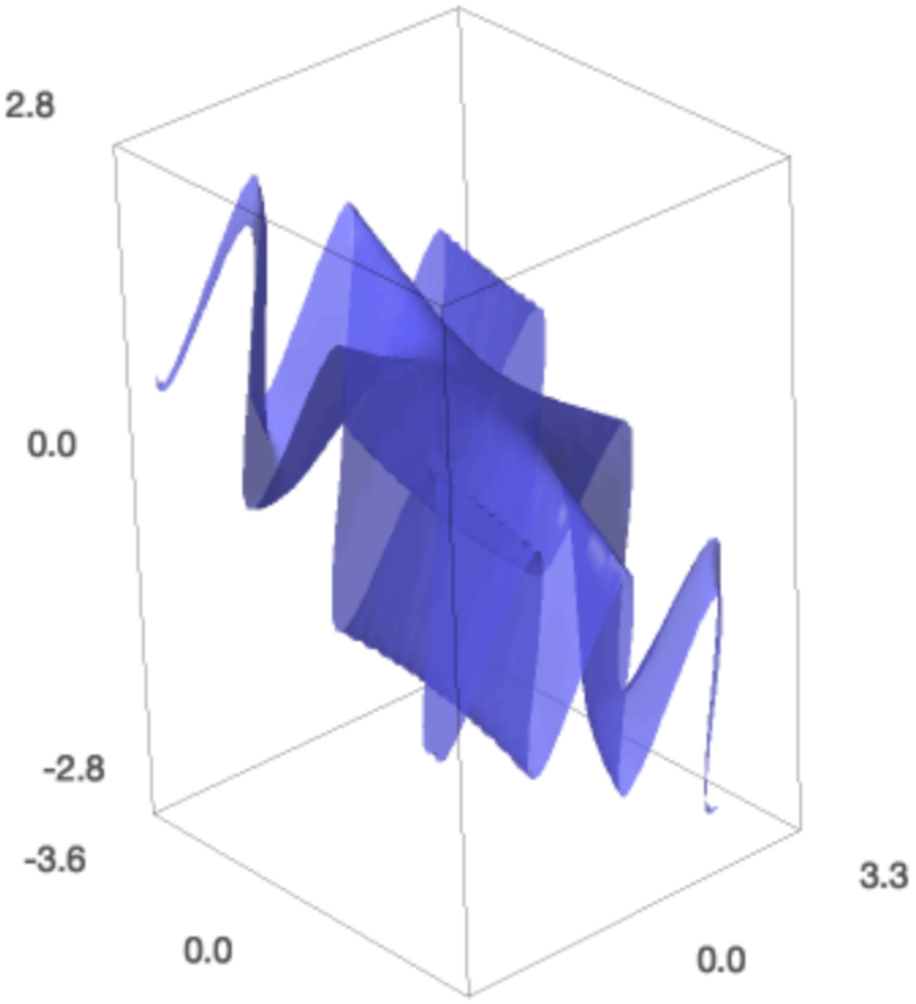

Fig. 3.12: The plot of the surface $z = \cos(xy)$ with adjusted viewing angles, defined number of plot points, and an opacity factor.

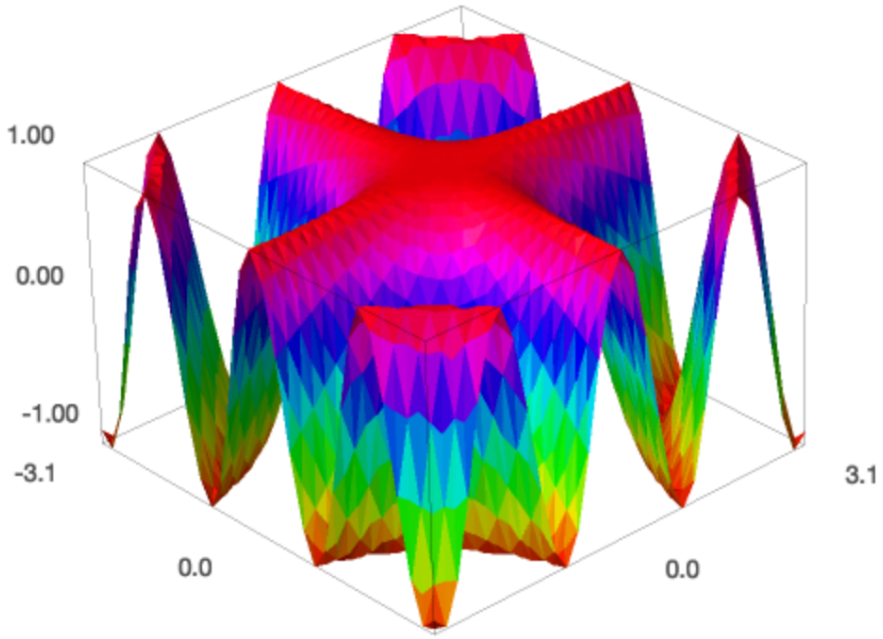

Fig. 3.13: The plot of the surface $z = \cos(xy)$ with an adaptive choice of plot points and with many colors.

Fig. 3.14: The plot of a parametric surface.

3.2.2 Space Curves

A space curve is defined by a parametric plot of 3 functions in one variable. For example, the twisted cubic is defined by $x = t, y = t^2, z = t^3$.

```
t = var('t')
twisted = (t, t^2, t^3)
parametric_plot3d(twisted, (t, -1, 1))
```

We can make the curve look red and thicker.

```
pt = parametric_plot3d(twisted, (t, -1, 1), color='red', thickness=5)
show(pt)
```

The result of the `parametric_plot3d` is shown in Fig. 3.15.

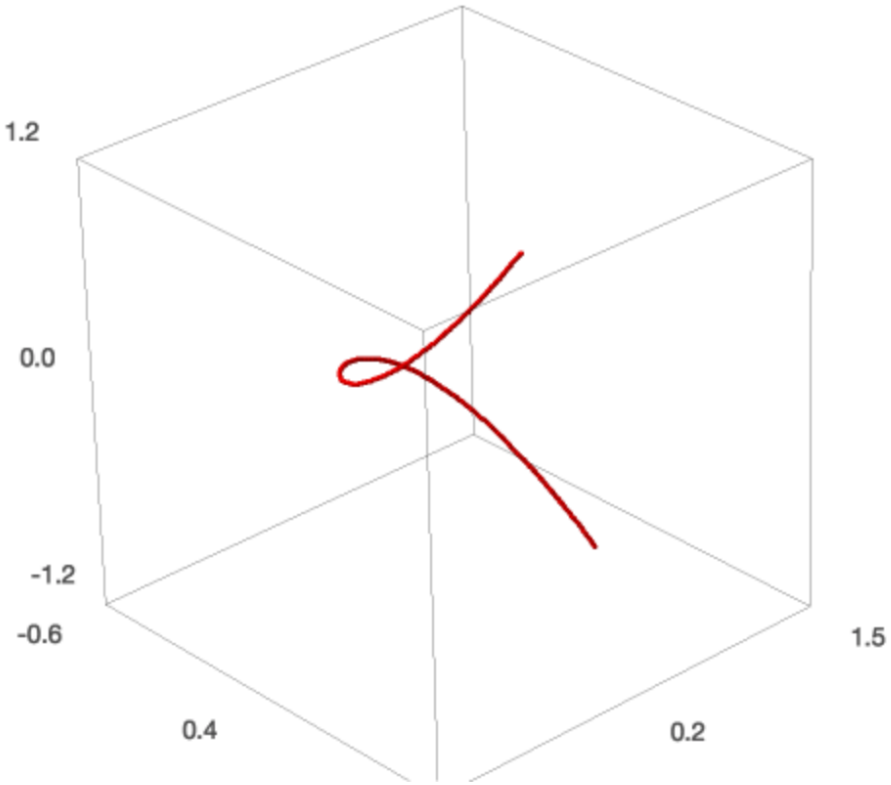

Fig. 3.15: A plot of the space curve (t, t^2, t^3), the twisted cubic.

There is a bounding box attribute associated to the plot.

```
pt.bounding_box()
```

We need to know the bounding box if we want to add to the plot, so we know which ranges for the three coordinates to choose. The twisted cubic is the intersection of a parabolic and a cubic cylinder. The parabolic cylinder has its base in the (x, y)-plane and from the parametric representation of the twisted cubic $(x = t, y = t^2)$, we can derive the implicit equation as $x^2 - y = 0$.

```
x, y, z = var('x,y,z')
c2 = implicit_plot3d(x^2 - y,(x,-1,1),(y,0,1), (z,-1,1))
show(pt+c2)
```

The result of the `implicit_plot3d` is shown in Fig. 3.16.

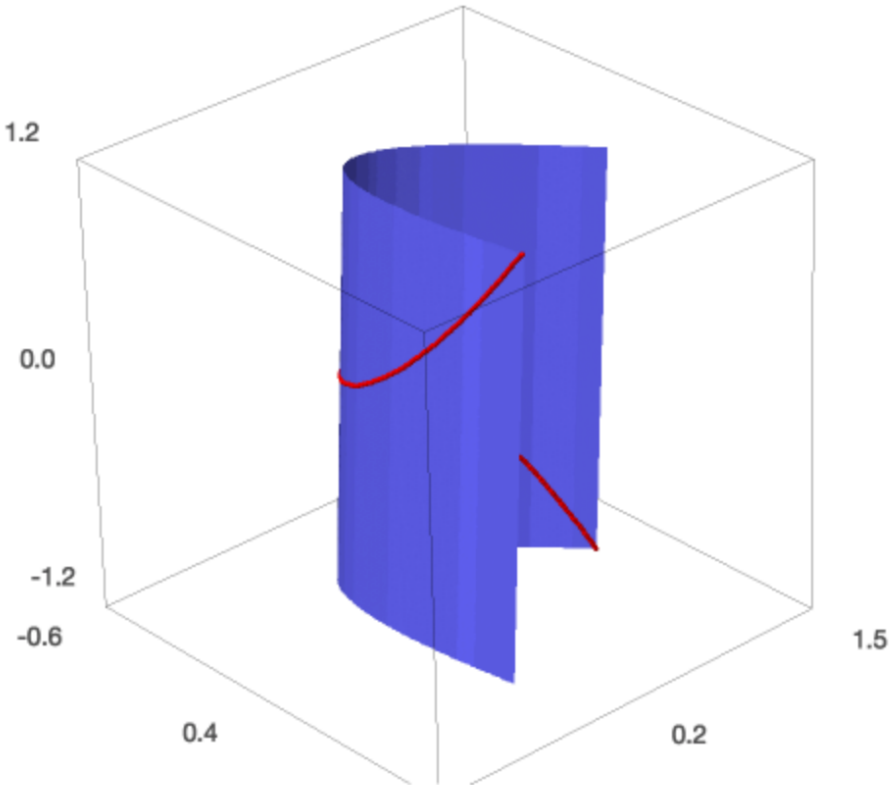

Fig. 3.16: The twisted cubic (t, t^2, t^3) on the parabolic cylinder $x^2 - y = 0$.

The cubic cylinder has its base in the (x, z)-plane and its equation can be derived from the parameter representation for the twisted cubic, $(x = t, z = t^3)$ so the equation is $x^3 - z = 0$.

```
c3 = implicit_plot3d(x^3 - z,(x,-1,1),(y,0,1), (z,-1,1), color='green')
show(pt+c2+c3)
```

We then see the two cylinders with the twisted cubic as their intersection.

3.2.3 Four Dimensional Plots with Colormaps

With colormaps we can plot in four dimensions. Let us first explore the use of colormaps. We can color surfaces with a colormap. As an example we take the Moebius strip. First we plot it without color.

```
from sage.plot.plot3d.parametric_surface import MoebiusStrip
ms = MoebiusStrip(3,1,plot_points=200).rotateX(-pi/8)
show(ms)
```

The figure is shown in Fig. 3.17.

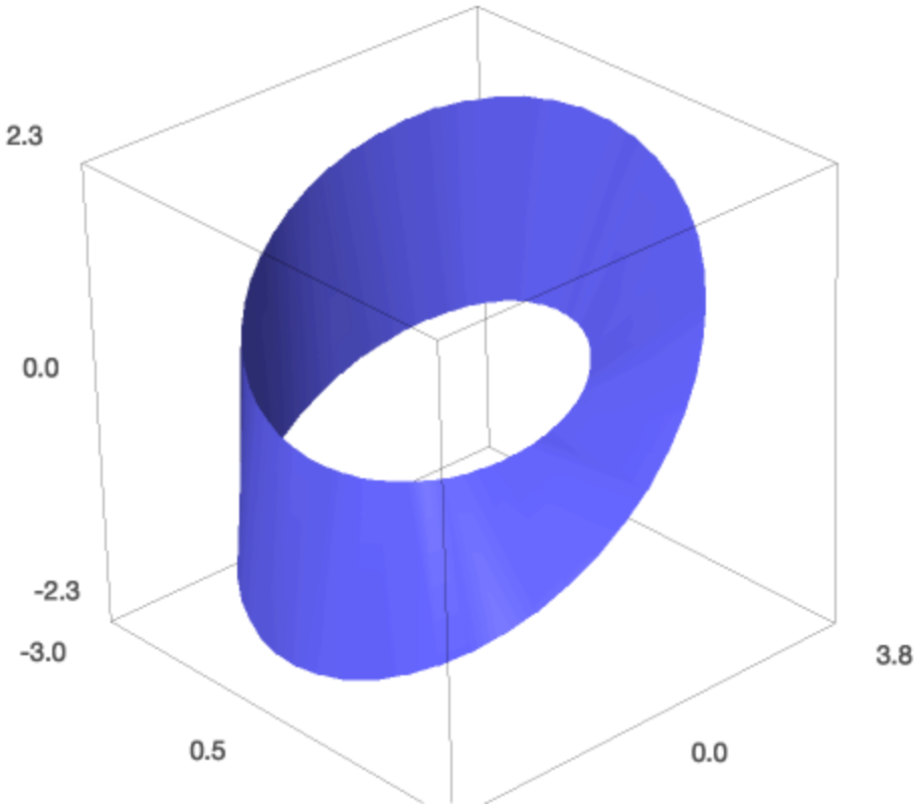

Fig. 3.17: The Moebius strip is a one-sided surface.

Now we apply a colormap.

```
cm = colormaps.ocean
def c(x,y): return sin(x*y)**2
mscm = MoebiusStrip(3,1,plot_points=200,color=(c,cm)).rotateX(-pi/8)
show(mscm, frame=False, viewer='tachyon')
```

The Moebius strip shown in colors is displayed in Fig. 3.18.

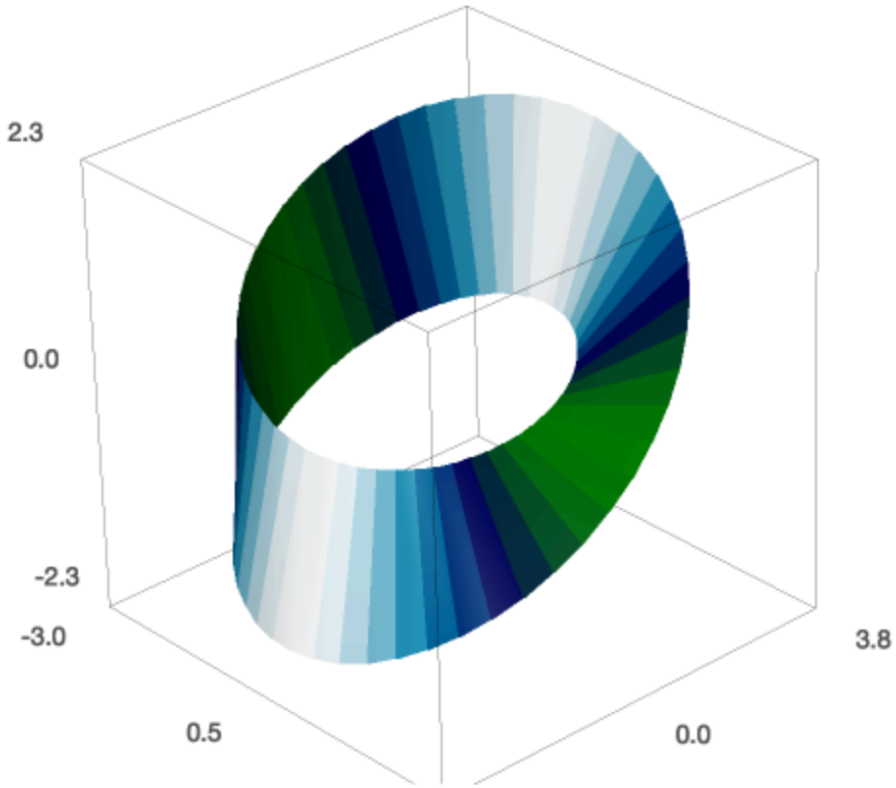

Fig. 3.18: A colored one sided surface, the Moebius strip.

We can use colormaps to make four dimensional plots. Consider the cubic root of a complex number.

```
u, v = var('u,v')
w = u + I*v
z = w^3
x = real_part(z)
y = imag_part(z)
```

We consider any complex number `w`. Because `u` and `v` are real numbers, so we can simplify their real and imaginary parts.

```
D = {real_part(u):u, imag_part(u):0, real_part(v):v, imag_part(v):0}
xx = x.subs(D)
yy = y.subs(D)
print(xx)
print(yy)
```

and then we see `u^3 - 3*u*v^2` and `3*u^2*v - v^3` as the expressions for `xx` and `yy`. We can now plot the surface using the expressions for `xx` and `yy` as functions of `u` and `v`, just as `parametric_plot3d((xx, yy, u), (u, -1, 1), (v, -1, 1))`.

Why does this represent the cubic root? Well, the height of the surface is `u`, the real part of the complex number `w` we started with. The `xx` and `yy` are the real and imaginary parts of the `z`, where `z` was obtained by taking the number `u + I*v` to the third power.

Going to polar coordinates produces a nicer plot.

```
r, t = var('r,t')
rt = {u:r*cos(t), v:r*sin(t)}
px = xx.subs(rt)
py = yy.subs(rt)
```

and now we make the plot as

```
parametric_plot3d((px, py, r*cos(t)), (r, 0, 1), (t, 0, 2*pi), adaptive=True)
```

The plot is shown in Fig. 3.19.

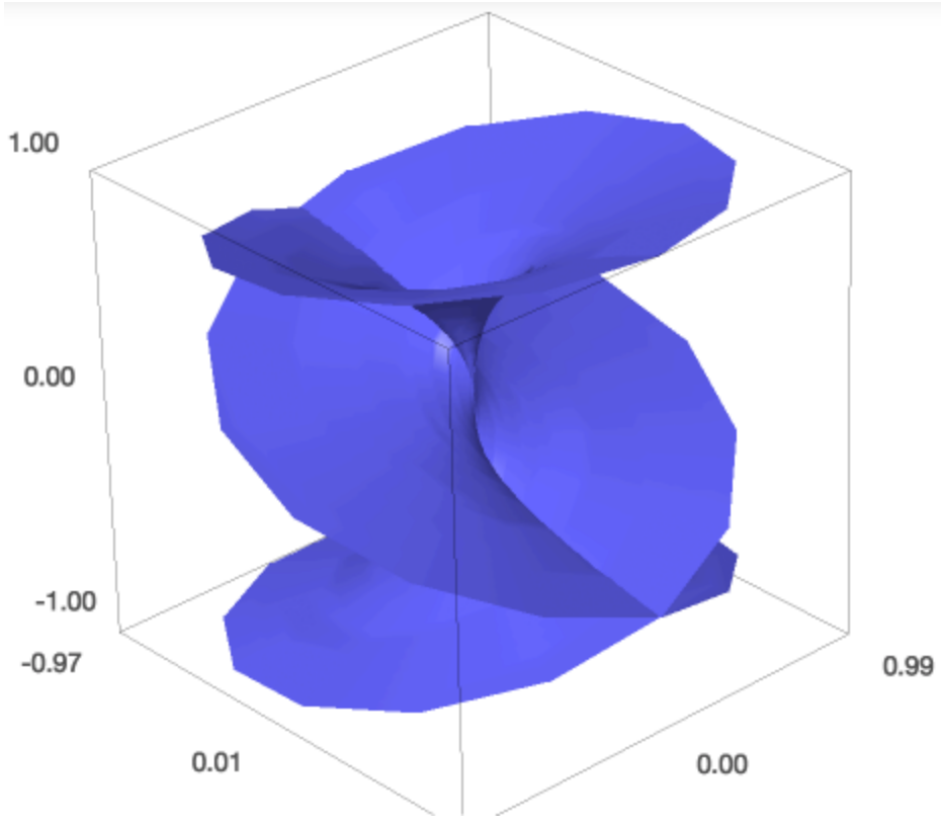

Fig. 3.19: A plot of the real part of the cubic root surface.

Now we would like as color to use the imaginary part, `v = r*sin(t)`.

```
cm = colormaps.autumn
def c(r,t): return r*sin(t)
cr = parametric_plot3d((px, py, r*cos(t)), (r,0,1), (t, 0, 2*pi), \
```

(continues on next page)

(continued from previous page)

```
    adaptive=True,color=(c,cm)).rotateX(pi/10).rotateY(-pi).rotateZ(pi/2)
show(cr, frame=False, viewer='tachyon')
```

and this produces the plot shown in Fig. 3.20.

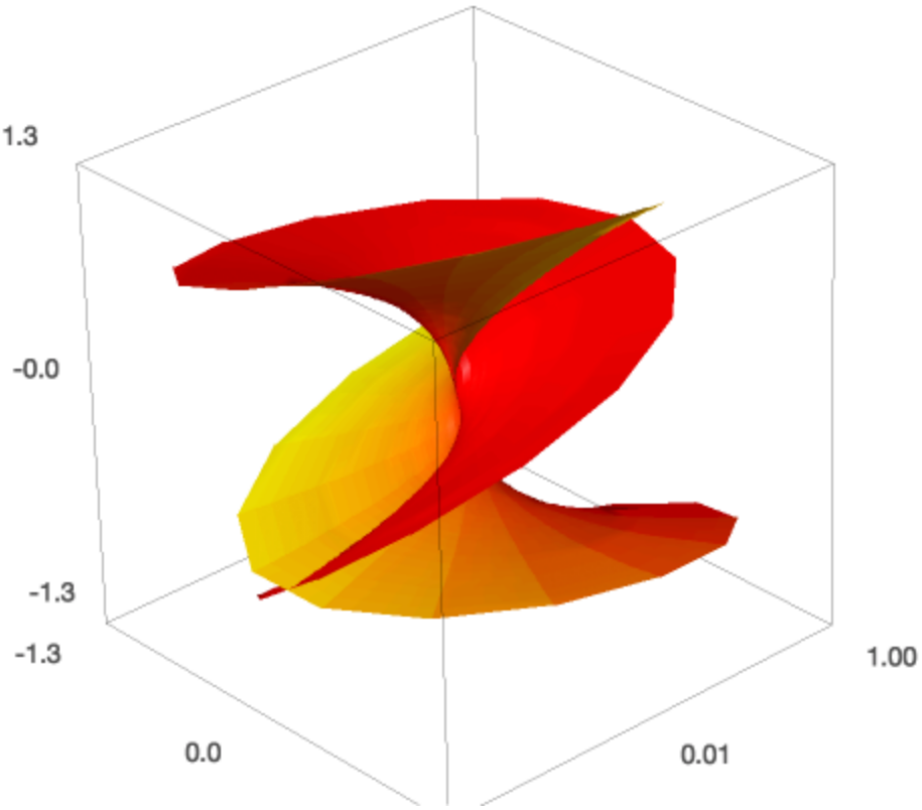

Fig. 3.20: The plot of a Riemann surface.

The height of the surface is the real part and the color of the surface represents the imaginary part of the cubic root. This is a four dimensional plot, also called a Riemann surface.

3.2.4 Assignments

1. Consider $h = 2\cos(0.4x)\cos(0.4y) + 5xye^{-(x^2+y^2)} + 3e^{-((x-2)^2+(y-2)^2)}$. Choose appropriate ranges for x and y so that the plot of this surface shows three peaks.
2. The *Viviani curve* is a space curve defined by $x = \cos^2(t)$, $y = \cos(t)\sin(t)$, and $z = \sin(t)$. Give the SageMath command to plot this space curve.
3. The *Viviani curve* is defined as the intersection of the sphere $x^2 + y^2 + z^2 = 1$ and the cylinder $x^2 + y^2 = 1$. Plot the two surfaces and emphasize their intersection adding the result of the plot of the previous exercise.
4. A torus knot is defined by $r = 2 + 4/5\cos(7t)$, $x = r\cos(4t)$, $y = r\sin(4t)$, and $z = \sin(7t)$. Plot this knot with SageMath.
5. Consider the surface defined by the equation $f(x, y, z) = x^3 - y^2 - z^2 = 0$.

1. Give the SageMath command to plot this surface, for $x \in [0, 1]$ and for $y, z \in [-1, 1]$. What do you see at (0,0,0)? Describe.
2. Replace z by zero and transform the curve defined by $f(x, y, 0) = 0$ into polar coordinates. Write the SageMath command to graph the curve in polar coordinates. Use good bounds for the range of t.

3.3 Lecture 27: Animations

An animation is defined by a list of plots, which will make the frames of the movie. We will show various examples of animations with the several plotting commands we have covered so far.

As in the previous lecture, our browser is google chrome and we use the Sage Math Cloud.

3.3.1 Animating Plots

Let us start with a two dimensional plot of a sine function with a vanishing amplitude. The plot is shown in Fig. 3.21.

```
t = var('t')
f = exp(-t^2)*sin(2*pi*t)
plot(f, (t, -2, 2))
```

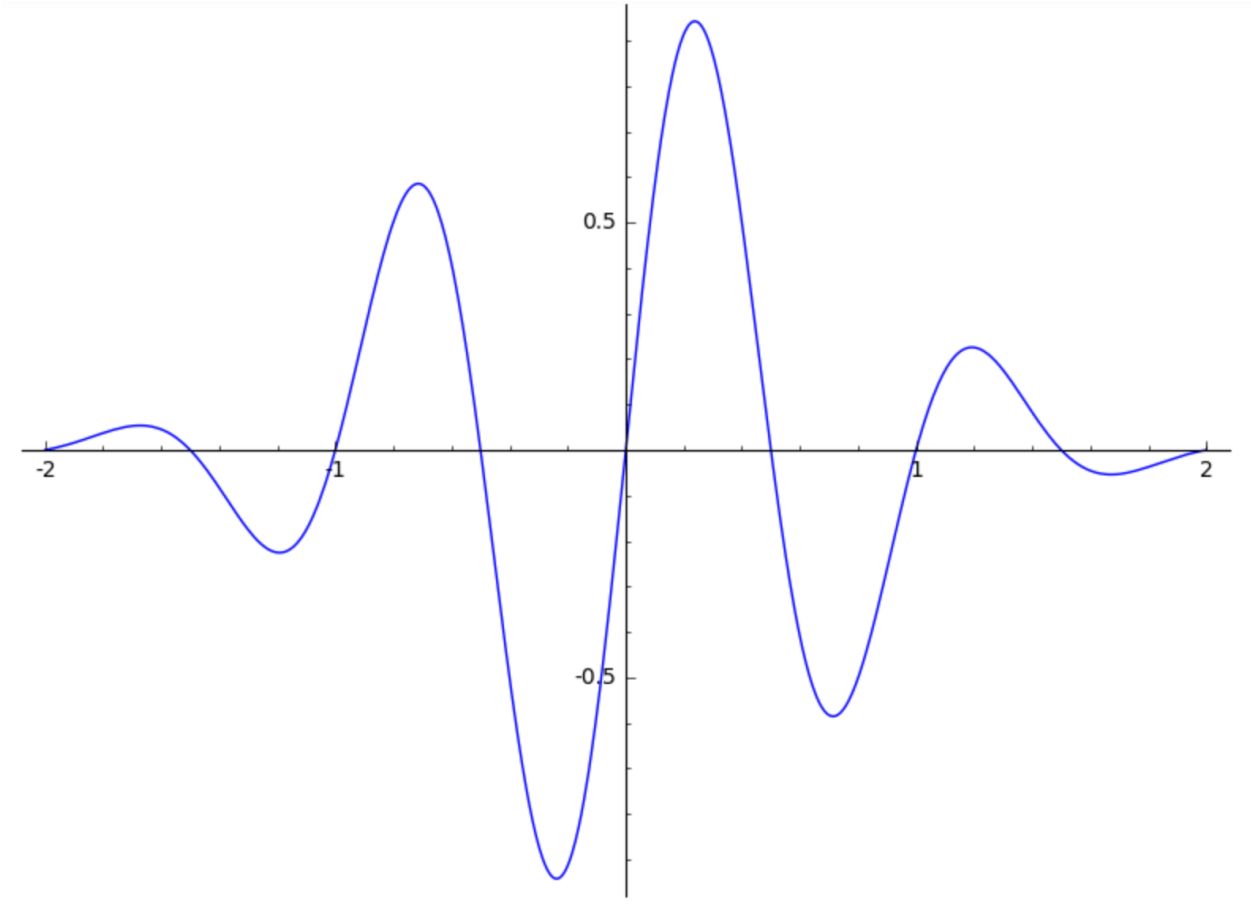

Fig. 3.21: The first frame in an animation.

The argument of the sine function is set so the frequency equals one, for `t` going from `0` to `1`, the argument of sine goes for 0 to `2*pi`. Suppose we want to see the evolution as we increase the frequency from 1 to 10. With a list comprehension, we make a sequence of plots. This list defines the frames in our animation. To test whether we did not make any mistakes in setting up the list of plots, we render the first frame. What we see is the same as in Fig. 3.21.

```
frames = [plot(exp(-t^2)*sin(2*k*pi*t), (t, -2, 2)) for k in range(1,11)]
frames[0].show()
```

And then, we just call the `animate` command on the list of frames.

```
animate(frames)
```

That is it! We can regulate the speed of the animation by providing a value for the parameter `delay`, for example: `animate(frames, delay=10)`. By default, the animation runs in an infinite loop. Giving a value for the parameter `iterations` fixes the number of iterations, for example: `animate(frames, iterations=4)`.

An animation can be turned into a `graphics_array`, suitable for viewing all frames (or selection of the frames) at once. We assign the `animate(frames)` to `a` and select the first 5 frames of the animation.

```
a = animate(frames)
g = a[:5].graphics_array(ncols=5)
show(g,figsize=[8,2], axes=False)
```

The first five frames of the animation are shown in Fig. 3.22.

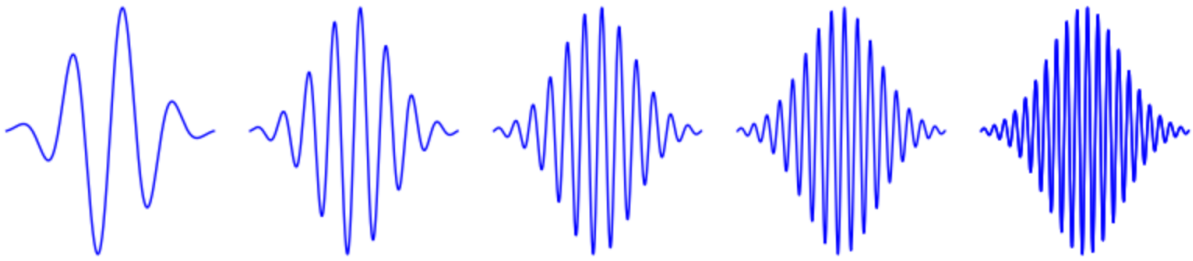

Fig. 3.22: The first five frames in an animation, via a graphics_array.

We can save an animation as a gif file:

```
a.save('ouranimation.gif')
```

The name between quotes may need be extended with the absolute path name if we do not want the animation to be saved in the current folder.

3.3.2 Designing an Animation

Interesting animations consists of several moving parts. Consider the animation of the tangent line to a circle. The ten frames are shown in Fig. 3.23.

For the first frame, we draw the unit circle and the first point at angle $2\pi/10$ and coordinates $(\cos(2\pi/10), \sin(2\pi/10))$.

```
unitcircle = circle((0,0),1)
firstpoint = point((cos(2*pi/10), sin(2*pi/10)), color='red',size=50)
(unitcircle+firstpoint).show(figsize=3)
```

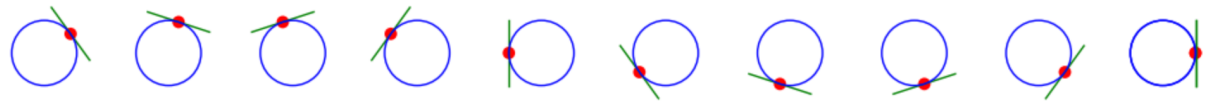

Fig. 3.23: Ten frames in the animation of the tangent lines to a circle.

The circle and point for the first frame are shown in Fig. 3.24.

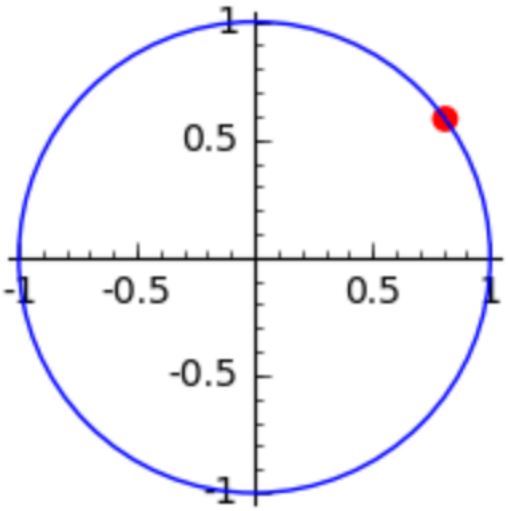

Fig. 3.24: A point on a circle in the first frame of the animation.

Now we make a list of points. The angle will be $2kpi/10$, for k ranging from 1 to 10. In the arguments of the `animate` command, we must specify values for the boundaries of the viewing window, for the `xmin`, `xmax`, `ymin`, `ymax`, because otherwise the point remains fixed and the coordinate axes are moving.

```
movingpoints = [point((cos(2*k*pi/10), sin(2*k*pi/10)), \
   color='red', size=50) for k in range(1,11)]
a = animate(movingpoints,xmin=-1,xmax=1,ymin=-1,ymax=1)
a.show(iterations=3)
```

For the tangent line, we know that vectors perpendicular to $(\cos(t), \sin(t))$ are $(-\sin(t), \cos(t))$ and $(\sin(t), -\cos(t))$. These two perpendicular vectors are added to $(\cos(t), \sin(t))$ to compute the end points of the tangent line. We define functions to make the points because we will generate lists of plots to make the frames of the animation.

```
pt(k) = (cos(2*k*pi/10), sin(2*k*pi/10))
v1(k) = (-sin(2*k*pi/10), cos(2*k*pi/10))
v2(k) = (-v1(k)[0],-v1(k)[1])
A(k) = ((pt(k)[0]+v1(k)[0]),pt(k)[1]+v1(k)[1])
B(k) = ((pt(k)[0]+v2(k)[0]),pt(k)[1]+v2(k)[1])
```

The function `pt` defines the coordinates of the moving point. The functions `A` and `B` given the end points of the tangent line. Now we are ready to plot the first frame in the animation.

```
tangentline1 = line([A(1), B(1)],color='green')
(unitcircle+firstpoint+tangentline1).show(figsize=4)
```

The first frame of the animation is shown in Fig. 3.25.

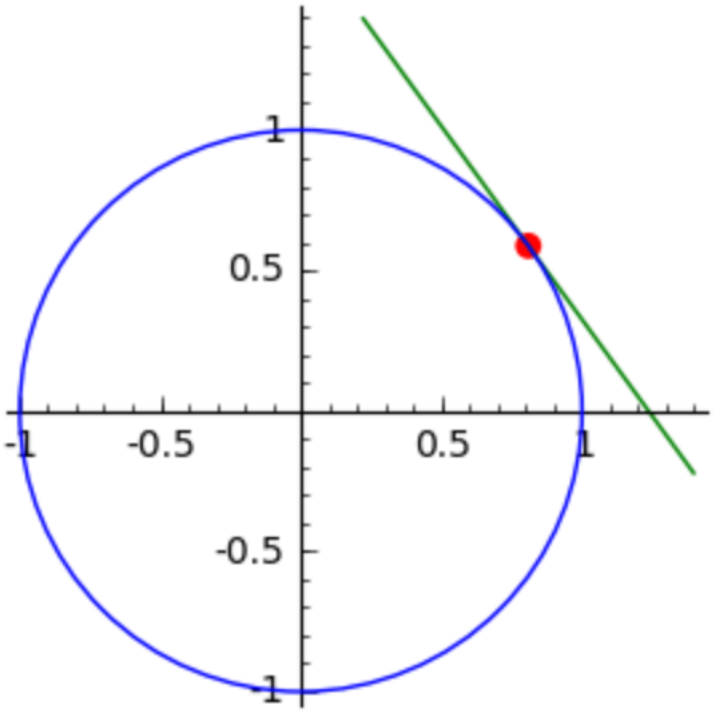

Fig. 3.25: The tangent to the first point on the circle in the animation.

The extra work to define the functions pays off as the definition of the 10 frames for the moving tangent lines is straightforward. Observe how the plot for the unit circle is added to each plot of a new tangent line.

```
movinglines = [unitcircle+line([A(k), B(k)], color='green') \
   for k in range(1,11)]
a = animate(movinglines,figsize=4,xmin=-1.5,xmax=1.5,ymin=-1.5,ymax=1.5)
g = a.graphics_array(ncols=10)
g.show(axes=False, figsize=[10,10])
```

The outcome of the above commands are shown in Fig. 3.26.

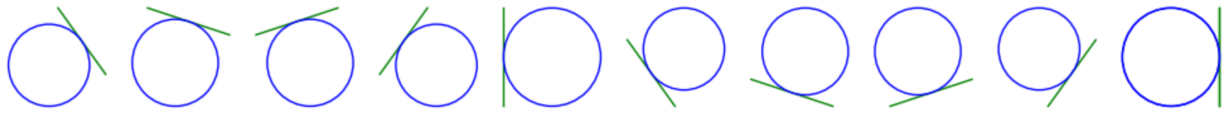

Fig. 3.26: The ten frames in the plot of tangents to the unit circle.

Then finally, we can make the animation.

```
frames = [movinglines[k]+movingpoints[k] for k in range(10)]
a = animate(frames, figsize=4, xmin=-1.5, xmax=1.5, ymin=-1.5, ymax=1.5, \
            axes=False)
a.show()
```

The frames in the animation are shown in Fig. 3.23.

3.3.3 Animating Surfaces

A first natural animation of a surface is to make a spin plot, for various orientations of the coordinate axes. For example, consider the parabolic cylinder defined by $x^2 - y = 0$. We let the x-axis rotate, with 20 frames, for the angle $2k\pi/10$, for math:k ranging from 1 to 20.

```
x, y, z = var('x, y, z')
Xframes = [implicit_plot3d(x^2 - y, (x, -1, 1), (y, 0 , 1), \
           (z, -1, 1)).rotateX(k*2*pi/10) for k in range(1, 21)]
Xa = animate(Xframes)
Xa.show()
```

and the animation shows a rotating parabolic cylinder. Replace the `rotateX` by `rotateY` or `rotateZ` to have the cylinder tumbling in a different direction.

Our next animated surface is a rotating plane. We consider the family $z = (1 - k/20)x + k/20y$, for k ranging from 0 to 20.

```
x, y = var('x,y')
frames = [plot3d((1-k/20)*x + k/20*y, (x,-1,1), (y,-1,1)) \
    for k in range(0, 21)]
animate(frames, xmin=-1, xmax=1, ymin=-1, ymax=1, zmin=-1, zmax=1)
```

For $k = 0$ we have $z = x$ and for $k = 20$ we have $z = y$.

3.3.4 Animating Space Curves

A space curve in parameter form is defined by three functions in one parameter, defined over some range. We can visualize the growing of the space curve by letting the parameter range grow, as illustrated by the following knot.

```
reset()
t = var('t')
r = 2 + 4/5*cos(7*t)
z = sin(7*t)
curve = [r*cos(4*t), r*sin(4*t), z]
parametric_plot3d(curve,(t, 0, 2*pi), thickness=5)
```

The above plot uses the range `(t, 0, 2*pi)` for `t` to range in the entire interval, which closes the knot. To view the growth of the knot, define the frames as follows:

```
frames = [parametric_plot3d(curve,(t,0,2*k*pi/20), thickness=5) \
          for k in range(1,21)]
a = animate(frames, xmin=-3, xmax=3, ymin=-3, ymax=3, zmin=-3, zmax=3)
a.show()
```

Observe that the number of plot points remains the same. In a better animation, the number of plot points would increase with each frame.

3.3.5 Assignments

1. Make an animation of ten frames of the plot of $\exp(-t^2)\sin(2\pi t)$, for t starting at -2 and ending at $-2+2k/5$, for k ranging from 1 to 10.

   The frames of the animation are displayed in Fig. 3.27.

Fig. 3.27: An animation of a plot with increasing right bound for the variable.

2. The *neoid* is defined by $r = at + b$ in polar coordinates.

   Give the Sage commands to produce an animation of 10 frames, for $a = 0.2$ and for b going from 0.1 to 1, and for $t = 0 \ldots 6\pi$.

3. Make an animation of a spiral, using the formulas $x = t\cos(t)$, $y = t\sin(t)$, and $z = t$. Let there be 30 frames in your animation, with $t = 1, 2, \ldots, 30\pi$.

4. The *Viviani curve* is a space curve defined by $x = R\cos^2(t)$, $y = R\cos(t)\sin(t)$, and $z = R\sin(t)$, where R is some parameter.

   Make an animation for the curve for R going from 1 to 10 (thus using ten frames). Give all Sage commands you use to make this animation.

5. Make an animation to show the drawing of the golden rectangle, using 14 frames shown in Fig. 3.28.

3.4 Lecture 28: Solving Equations

In this lecture we solve equations. We start with solving a polynomial in one variable where the coefficients depend on a parameter. Solving this polynomial symbolically requires the application of `solve` many times. Our running example of a system of polynomial equations is the Apollonius circle problem. A Groebner basis of an ideal in a polynomial ring with lexicographic term order is a triangular basis.

3.4.1 Polynomials in One Variable

Let us consider a polynomial with a parameter `A`.

```
A, x = var('A, x')
equ = A^2*(x^2+x+1)-A*(2*x-3) - x^2-3*x+2
equ
```

Now we can solve for x and obtain symbolic expressions in `A` as solutions.

```
s = solve(equ, x)
s
```

Consider the solutions:

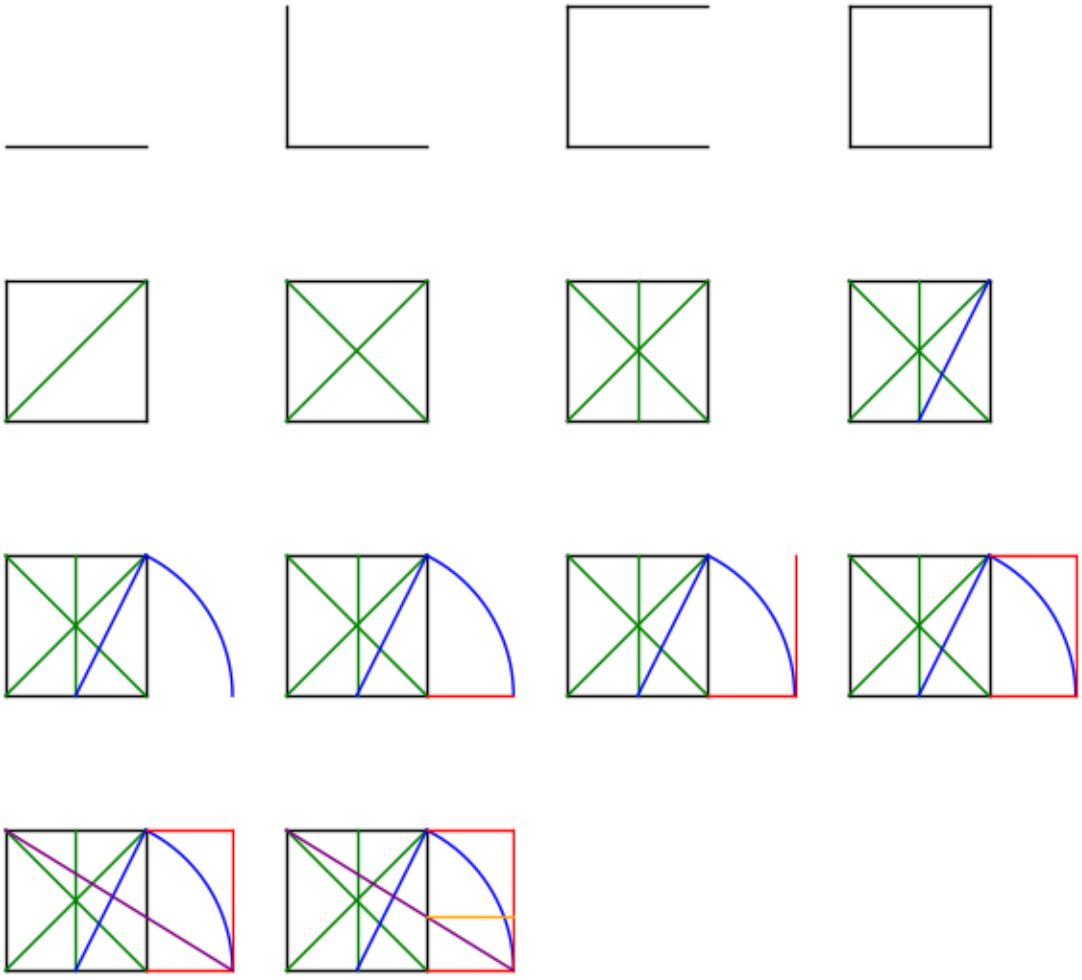

Fig. 3.28: Drawing the golden rectangle with ruler and compass.

```
[
x == -1/2*(A + sqrt(-3*A^2 - 10*A + 17) - 3)/(A - 1),
x == -1/2*(A - sqrt(-3*A^2 - 10*A + 17) - 3)/(A - 1)
]
```

The solutions were obtained via the quadratic formula. But what if we solved for `A`?

```
sA = solve(equ, A)
sA
```

and we get `A == -1` and `A == (x^2 + 3*x - 2)/(x^2 + x + 1)` as solutions. Could `A = -1` be a special parameter?

```
equ.subs(A=-1)
```

We see `0` and thus, yes, indeed: `A = -1` is a special parameter. We just saw that the equation vanishes when `A` is set to `-1`. This means that any value for `x` is a solution. Could there be other special values for `A`? What happens when we plug in `A = 1` in the equation?

```
equA1 = equ.subs(A = 1)
equA1
```

When plug in `A = 1` we get `-4*x + 6`. Recall the factor `A - 1` in the general solution we first computed. So for `A = 1`, the solutions we computed before in s are not valid, but there is still a solution, just one single solution.

```
solve(equA1, x)
```

and that solution is `x == (3/2)`. Another way we could have detected the special values for `A` is via the leading coefficient of `x` in the equation `equ`.

```
q = equ.coefficient(x,2)
q
```

Collecting the terms in `x^2` gives `A^2 - 1` So the values that kill the leading coefficient of `equ` are

```
solve(q, A)
```

which of course gives `A == -1` and `A == 1`. But then why does one parameter give such different behaviour for the solution `x` than the other? Let us look at the linear coefficient of the equation in `x`.

```
q1 = equ.coefficient(x,1)
q0 = equ.coefficient(x,0)
print(q1)
print(q0)
```

and the equations `q1` and `q0` are respectively `A^2 - 2*A - 3` and `A^2 + 3*A + 2`. Then we solve the expressions and we see that `A = -1` is a common solution to both the linear and constant coefficient.

```
print(solve(q1, A))
print(solve(q0, A))
```

The solutions of `q1` are `A == 3` and `A == -1`, while the solutions of `q0` are `A == -2` and `A == -1`.

Count the many times we did `solve` to solve one polynomial equation with parameters and realize that solving equations with parameters symbolically is a much harder problem than solving a polynomial with a numerical coefficients.

3.4.2 Solving Systems of Polynomial Equations

We turn our attention to systems of polynomial equations and solve the circle problem of Apollonius: *Given three circles, find all circles that are tangent to the three given circles.* An instance of this problem in shown in Fig. 3.32. Before we get to this, let us first do some exploratory computations with one circle touching the unit circle.

To derive the equations, consider the unit circle and a random circle inside the unit circle. We first plot the unit circle. Because we will solve the equations, we work with `implicit_plot()`, rather than with the command `circle()`.

```
x, y = var('x,y')
u = x^2 + y^2 - 1
p0 = implicit_plot(u, (x, -1, 1), (y, -1, 1))
p0.show()
```

We consider a random circle inside the unit circle. A random circle inside the unit circle has a random center. For reproducible and predictable results, we set the seed of the random number generator.

```
set_random_seed(2018)
cx = RR.random_element(0, 0.5)
cy = RR.random_element(0, 0.5)
print(cx, cy)
p1 = point((cx, cy), size=50, color='red')
(p0 + p1).show()
```

The plot is shown in Fig. 3.29.

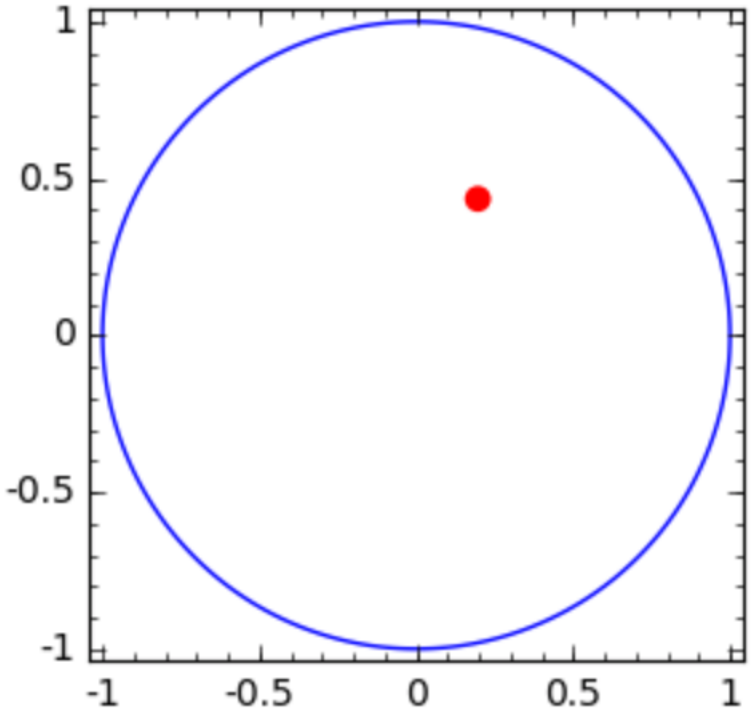

Fig. 3.29: A random point in the unit circle.

Now the radius of the circle that touches the unit circle is ...

```
r = 1 - sqrt(cx^2 + cy^2)
print(r)
p2 = implicit_plot((x - cx)^2 + (y - cy)^2 - r^2, (x, -1, 1), (y, -1, 1), \
    color='red')
(p0 + p1 + p2).show()
```

The first circle touching the unit circle from the inside is shown in Fig. 3.30.

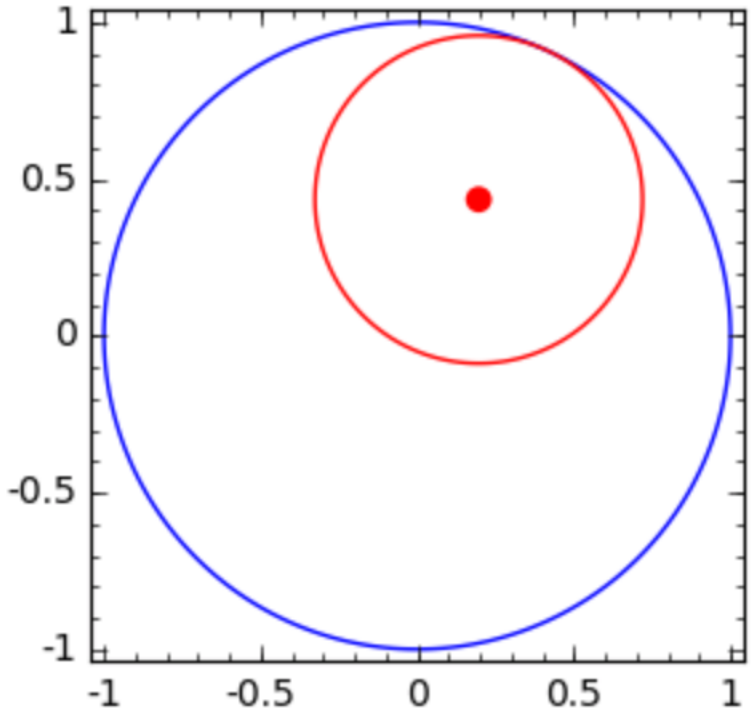

Fig. 3.30: A circle touching the unit circle from the inside.

We know there is another solution... How to find it?

```
r2 = 1 + sqrt(cx^2 + cy^2)
p3 = implicit_plot((x - cx)^2 + (y - cy)^2 - r2^2, (x, -2, 2), (y, -2, 2), \
    color='green')
(p0 + p1 + p2 + p3).show()
```

The other circle which touches the unit circle from the outside is shown in Fig. 3.31.

Now we turn our attention to the circle problem of Apollonius. We need three circles on input: The first circle will be the unit circle. The second circle has center (2, 0) with radius 2/3. The third circle has center (1, 1) with radius 1/3.

```
(c1x, c1y, r1) = (0, 0, 1)
(c2x, c2y, r2) = (2, 0, 2/3)
(c3x, c3y, r3) = (1, 1, 1/3)
c1 = (x-c1x)^2 + (y-c1y)^2 - r1^2
c2 = (x-c2x)^2 + (y-c2y)^2 - r2^2
c3 = (x-c3x)^2 + (y-c3y)^2 - r3^2
```

(continues on next page)

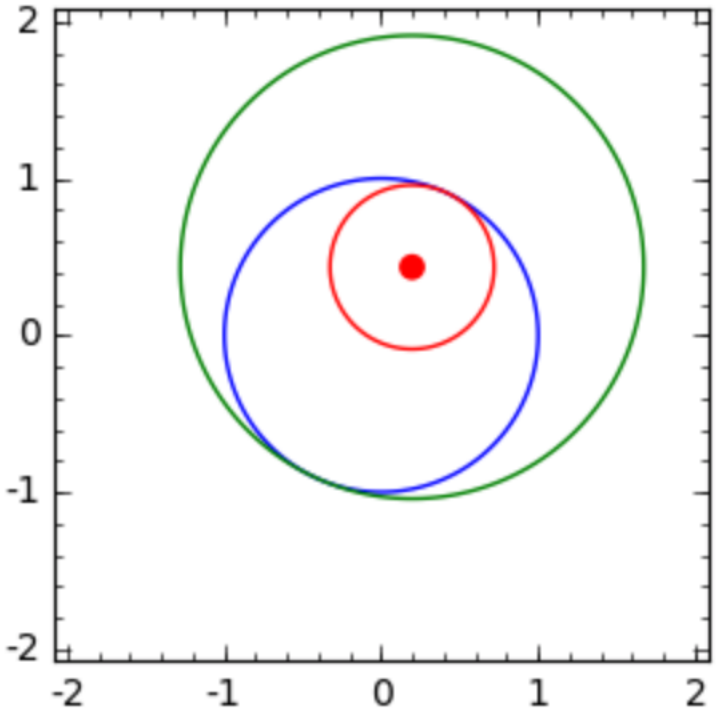

Fig. 3.31: The second circle touching the unit circle from the outside.

(continued from previous page)

```
xr = (x, -1, 3); yr = (y, -1, 2)
p1 = implicit_plot(c1, xr, yr)
p2 = implicit_plot(c2, xr, yr)
p3 = implicit_plot(c3, xr, yr)
(p1 + p2 + p3).show()
```

The input to the Apollonius circle problem is shown in Fig. 3.32.

Now we look for the circle that will be tangent to all three circles.

```
r = var('r')
e1 = (x - c1x)^2 + (y - c1y)^2 - (r - r1)^2
e2 = (x - c2x)^2 + (y - c2y)^2 - (r - r2)^2
e3 = (x - c3x)^2 + (y - c3y)^2 - (r - r3)^2
eqs = [e1, e2, e3]
eqs
```

and now we solve the system. We set the option `solution_dict` to `True` because this works better to select the coordinates of the solution.

```
sols = solve(eqs, (x, y, r), solution_dict=True)
sols
```

We see there are two solutions, computed exactly.

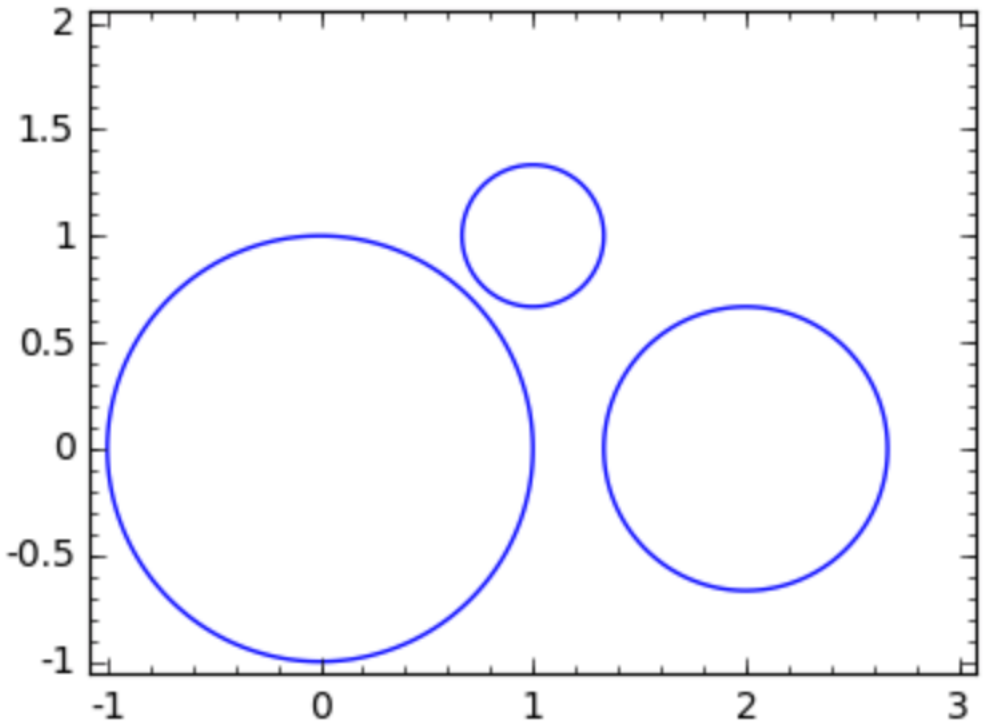

Fig. 3.32: Find all circles which touch three given circles.

```
s0x = sols[0][x]
s0y = sols[0][y]
s0r = sols[0][r]
print(s0x, s0y, s0r)
sc0 = implicit_plot((x-s0x)^2 + (y-s0y)^2 - s0r^2, (x, -2, 3), (y, -3, 2), \
    color='red')
s1x = sols[1][x]
s1y = sols[1][y]
s1r = sols[1][r]
print s1x, s1y, s1r
sc1 = implicit_plot((x-s1x)^2 + (y-s1y)^2 - s1r^2, (x, -2, 3), (y, -3, 2), \
    color='red')
(p1 + p2 + p3 + sc0 + sc1).show()
```

The plot of two circles touching three given circles in shown in Fig. 3.33.

Note that this problem has actually 8 solutions in total. What we computed are only types. Instead of `(r-r1)`, we should also consider `(r+r1)`.

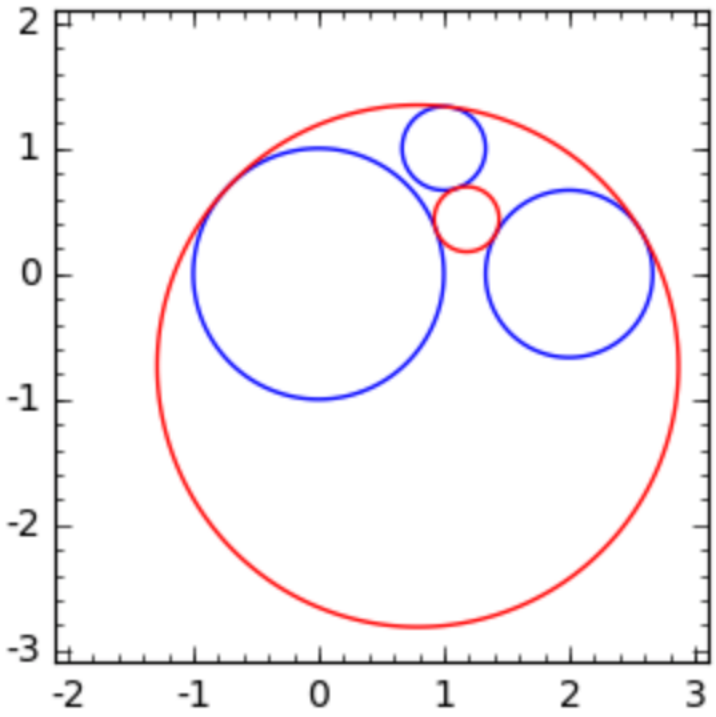

Fig. 3.33: Two of the eight circles touching three given circles.

3.4.3 Groebner Bases

We saw that the system was solved exactly, via what looked like the quadratic formula. We will cast the polynomial equations in a polynomial ring with lexicographic order.

```
PR.<x,y,r> = PolynomialRing(QQ, order='lex')
print(PR)
q1 = PR(e1)
print(q1, 'has type', type(q1))
```

We see that `PR` is a `Multivariate Polynomial Ring in x, y, r over Rational Field` and the polynomial `q1` has type `MPolynomial_libsingular` so we will be using the computer algebra system Singular. to work with the polynomials of the Apollonius circle problem. We also cast the other two polynomials into this ring and define the ideal generated by the three polynomials. An ideal generated by polynomials in a ring consists of all combinations of the generators with elements in the ring.

```
q2 = PR(e2); q3 = PR(e3)
qs = [q1, q2, q3]
J = Ideal(qs)
J
```

A *Groebner basis* of an ideal in a polynomial ring with *lexicographic* term order is a *triangular* basis.

```
B = J.groebner_basis()
B
```

The Groebner basis for the ideal generated by the polynomials of the Apollonius circle problem is

```
[x + 1/6*r - 41/36, y + 1/2*r - 11/36, r^2 - 71/39*r - 253/468]
```

We see that the last polynomial is a quadratic polynomial in one variable, that is `r`. Recall that `r` was the last variable in our lexicographic term order. The other two polynomials are linear and we can express `x` and `y` as functions of `r`.

We can ask for the solution set defined by the polynomials in the ideal `J` by asking for its variety:

```
J.variety()
```

If we provide no argument to `variety()` then the default field is `QQ` and only the rational solutions will be returned. For more general approximations, we provide the extended field `QQbar`.

```
J.variety(QQbar)
```

The returned solutions are approximated by intervals.

3.4.4 Assignments

1. Consider the polynomial equation $x^2a^2 - 12x^2 + a^2x - 3\sqrt{3}ax + 6x + a^2 - \sqrt{3}a - 6 = 0$.

   Give the Sage command to solve the equation with x as variable.

   For which values of the parameter a does the equation have less than two solutions?

   Find the solution for this special value of the parameter.

   For which values of the parameter a does the equation have infinitely many solutions?

   Give the Sage commands you used to test your answer.

2. There are eight different solutions to the Apollonius circle problem, which can be obtained by considering instead of `r - r1` also `r + r1` and likewise for the radii `r2` and `r3`. Make a plot that shows all eight different solutions as shown in Fig. 3.34.

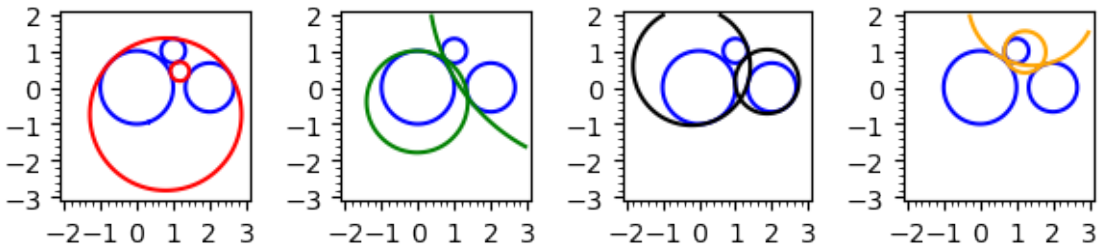

Fig. 3.34: The four pairs of solutions to the problem of Apollonius.

3. Solve the system

$$\begin{cases} x^2 + y^2 - 5 = 0 \\ 9x^2 + y^2 - 9 = 0 \end{cases}$$

   Make a plot of the two algebraic curves to confirm the values found for the intersection points.

   How many solutions did you find? How many solutions do you see?

4. As an illustration of the technique of Lagrange multipliers we derived the system in the list below:

```
[-2*w*x + 2*y*z + 2*x == 0, -2*w*y + 2*x*z == 0,
2*x*y - 2*w*z - 2*z == 0, x^2 + y^2 + z^2 - 1 == 0]
```

Compute a Groebner basis for a ideal generated by the four polynomials in the list above, with a lexicographic term order.

Explain how you can compute the number of solutions from the triangular structure of the Groebner basis.

5. A comet approaches earth in an ellipsoid path, defined by $(x-2)^2/6 + y^2/2 - 1 = 0$, where the earth is at the center $(0,0)$ of our coordinate system.

   We want to know the locations where the comet is closest to earth.

   (a) Set up the polynomial system to find those locations.

   (b) Solve the polynomial system and find the coordinates on the ellipse closest to the origin.

   (c) Make a plot of the ellipse and the solution found in (b).

3.5 Lecture 29: Linear Algebra

In this lecture we do linear algebra: we solve linear equations. The formalism to describe linear equations is via matrices. Let us first then look at the matrix types.

3.5.1 Matrices and Linear Systems

With the data in a list of lists, we can make a matrix.

```
A = Matrix(ZZ,[[1,2],[3,4]])
print(A, 'has type', type(A))
```

Some matrices are defined by formulas. For example, let us make a 3-by-4 integer matrix where the (i,j)-th entry is $\frac{1}{i+j}$, for i and j both going from 1 to n. Let us make a 3-by-4 matrix with this formula.

```
B = Matrix(QQ, 3, 4, lambda i,j: 1/((i+1)+(j+1)))
B
```

Observe that we had to change the formula because in Python as in Sage we count from zero and we do not want 1/0 as the first element in the first row.

We can make a random matrix that way.

```
C = Matrix(ZZ, 3, 3, lambda i,j: randint(0,10))
C
```

We can make a random matrix over the rational numbers. In the example below we bound the size of numerator and denominator by 100.

```
D = random_matrix(QQ, 4, num_bound=100, den_bound=100)
D
```

A random vector can be made just as easily.

```
v = random_vector(QQ, 4)
print(v, 'has type', type(v))
```

We can solve linear systems with the backslash operator (as in MATLAB).

```
x = D\v
print(x)
print(D*x)
```

We thus solved with `x = A\b` the linear system `A*x = v`.

3.5.2 Matrices over Fields

One way to solve linear systems is via a LU factorization. An LU factorization writes the matrix as a product of a lower with an upper triangular matrix.

```
A = random_matrix(QQ, 4, num_bound=100, den_bound=100)
print('A =\n', A)
P, L, U = A.LU()
print('L =\n', L)
print('U =\n', U)
print('P =\n', P)
print('P*L*U =\n', P*L*U)
print('A =\n', A)
```

How does the LU factorization relate to solving linear systems? Well, because `A = P*L*U`, the system `A*x = b` becomes `P*L*(U*x) = b`, or if we set `y = U*x`, then we first solve `P*L*y = b` and then solve `U*x = y`. Let us try if that works on the example, for a random `b`.

```
b = random_vector(QQ, 4)
PL = P*L
y = PL\b
x = U\y
print(x)
print(A\b)
```

We see that `x` and `A\b` show the same vector.

Another important decomposition is the eigenvector-eigenvalue decomposition, which for general matrices, diagonalizes the matrix.

```
A = random_matrix(QQ, 4, 4)
print('A =\n', A)
e = A.eigenvalues()
print('the eigenvalues :\n', e)
```

We see that the eigenvalues seem numerical:

```
type(e[0])
```

but they are of type `sage.rings.qqbar.AlgebraicNumber`. The eigenvalues are roots of a polynomial, the *characteristic polynomial* of the matrix `A`.

```
p = A.charpoly()
print(p)
print(p.roots(QQbar))
print(p.roots(CC))
```

For an eigenvector z of a matrix with eigenvalue L, we have A*z = L*z. Let us verify this property for the first eigenvalue and eigenvector. The eigenvectors are stored in the columns of the matrix P.

```
D, P = A.right_eigenmatrix()
```

Let us verify the relations between the matrices.

```
print('D =\n', D)
print('the eigenvectors :\n', P)
print('P*D =\n', P*D)
print('A*P =\n', A*P)
```

We have that A*P = P*D, where D is a diagonal matrix. Since P is invertible, we have D = P^(-1)*A*P.

```
p1ap = P.inverse()*A*P
print(p1ap)
print(D)
print(D-p1ap)
```

The last output is the matrix of very small numbers.

For an eigenvector z of a matrix with eigenvalue L, we have A*z = L*z. Let us verify this property for the first eigenvalue and eigenvector. The eigenvectors are stored in the columns of the matrix P.

```
v = P[:,0]
print(v)
Av = A*v
Dv = D[0,0]*v
print(Av)
print(Dv)
```

We see that both vectors Av and Dv are the same.

3.5.3 Matrices over Rings

Recall that in a ring we cannot divide. We can then still solve linear systems with *fraction-free row reduction*.

The Hermite normal form H of a matrix A is an upper triangular form obtained by multiplying A with a unimodular transformation U. A unimodular matrix U has determinant 1 or -1.

```
A = Matrix(ZZ, 4, 4, lambda i,j: randint(1,10))
print('A =\n', A)
H, U = A.hermite_form(transformation=True)
print('H =\n', H)
print('U =\n', U)
print('U*A =\n', U*A)
```

We can check the determinants via the det() method.

```
print('det(A) =', A.det())
print('det(U) =', U.det())
print('det(H) =', H.det())
```

We see that det(U*A) = det(U)*det(A).

The Hermite normal form is a triangular matrix. The Smith normal form is a diagonal matrix. In addition to the diagonal matrix D, the `smith_form()` method returns two unimodular matrices U and V.

```
D, U, V = A.smith_form()
print('D =\n', D)
print('U =\n', U)
print('V =\n', V)
print('U*A*V =\n', U*A*V)
```

3.5.4 Resultants

With a lexicographic Groebner basis we can eliminate variables and obtain a triangular form of the system. An alternative construction is to use resultants, which are available via Singular. We use Singular when we declare a polynomial ring.

```
PR.<x,y> = PolynomialRing(QQ)
print(PR)
p = x^2 + y^2 - 5
q = 9*x^2 + y^2 - 9
print(type(p), 'has type', type(q))
```

We see that PR is then a `Multivariate Polynomial Ring in x, y over Rational Field` and both p and q are of type `MPolynomial_libsingular`.

Now we can eliminate one of the variables, say x. We eliminate from p the variable x with respect to q, or equivalently, we eliminate from q the variable x with respect to p.

```
print(p.resultant(q,x))
print(q.resultant(p,x))
```

We see the same polynomial `64*y^4 - 576*y^2 + 1296` twice. Similarly, we can eliminate y via `p.resultant(q, y)` and `q.resultant(p,y)`. With the resultant we can thus compute the coordinates of the roots.

Resultants are determinants of the so-called Sylvester matrix.

```
sp = p.sylvester_matrix(q, x)
print(sp)
print(sp.det())
```

The output of `sp.det()` corresponds to `p.resultant(q, y)`.

3.5.5 Plotting Matrices

We can visualize large sparse matrices via a plot. As an example, we will generate a random matrix of bits.

```
dim = 32
rm = Matrix(dim, dim, [GF(2).random_element() for k in range(dim*dim)])
rm.str()
```

For dimensions larger than 32, even plotting with the string method will not give a good view of the pattern of the matrix, whereas a matrix plot scales much better.

```
matrix_plot(rm, figsize=4)
```

The plot is shown in Fig. 3.35.

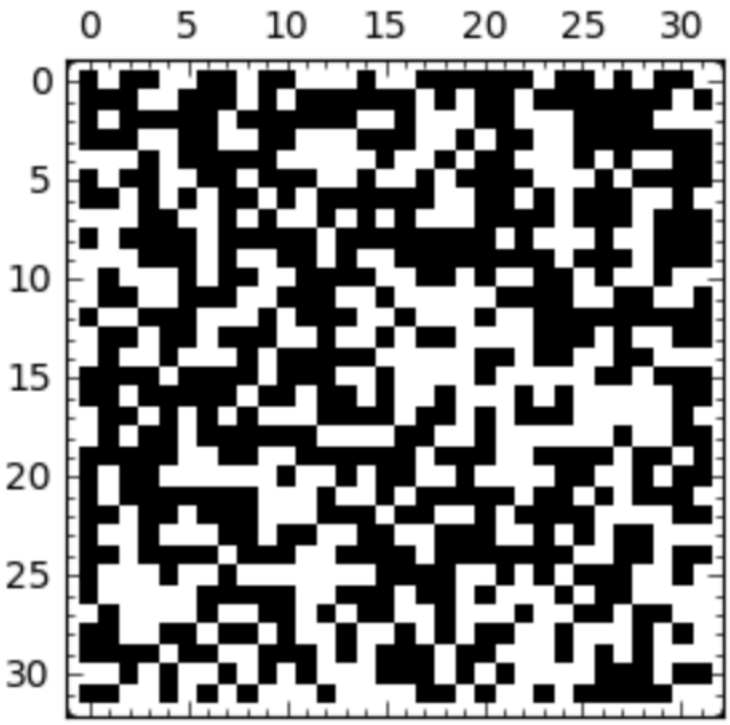

Fig. 3.35: The plot of a random 32-by-32 matrix.

3.5.6 Assignments

1. The (i, j)-th entry in an *n*-by-*n* Vandermonde matrix is defined as x_i^{j-1}, for *i* and *j* both ranging from 1 to *n*. Give the Sage command(s) to make a 4-by-4 Vandermonde matrix.
2. Compute the determinant of a 4-by-4 Vandermonde matrix (see the previous exercise). Show that it factors as the product of all differences $(x_i - x_j)$ for $j > i$ and all *i* from 1 to 4.
3. Given a polynomial *p* in one variable, the *companion matrix of p* as eigenvalues the roots of *p*. We will verify this property on a random polynomial.
   (a) Generate a *monic* random polynomial *p* of degree five. A monic polynomial has its leading coefficient equal to one. Use the Sage command `companion_matrix` (look up the help page on its use) to generate the companion matrix of *p*.
   (b) Compute the eigenvalues of the companion matrix and compare the eigenvalues with the output of `p.roots(CC)`.
4. The (i, j)-th entry of an n-by-n Lehmer matrix is $\frac{\min(i,j)}{\max(i,j)}$, for i and j both ranging from 1 to n.

   Generate a 5-by-5 Lehmer matrix and verify that all its eigenvalues are positive.

5. The (i, j)-th entry of the Pascal matrix is defined by $\begin{pmatrix} i+j-2 \\ j-1 \end{pmatrix}$. Make a sparse 32-by-32 matrix, setting the even entries in the Pascal matrix to zero, and the odd entries to one. The `matrix_plot` should show the Sierpinski gasket.

6. A Hankel matrix of dimension n has first row $\mathtt{H}[1], \mathtt{H}[2], \ldots, \mathtt{H}[n]$.

   The j-th element on the i-th row of the matrix equals $\mathtt{H}[1 + ((i + j - 2) \bmod n)]$, for i and j ranging both from 1 to n. Give the Sage command to define a symbolic Hankel matrix H_5 of dimension 5. How many terms does the determinant of H_5 have?

3.6 Lecture 30: Solving Differential Equations

In Sage, differential equations can be solved with `desolve`. As the `help(desolve)` shows, the `desolve` solves first or second order linear ordinary differential equations via the computer algebra system `maxima`.

We distinguish between initial value and boundary value problems. The pendulum problem is an example of an initial value problem, the heat diffusion problem is an example of a boundary value problem.

3.6.1 The Pendulum Problem

As our running example of a differential equation, consider a pendulum. We have a mass attached to a string, subject to gravity. The variable is `theta(t)`, the angle of deviation from the vertical position of the string, as a function in time.

```
t = var('t')
theta = function('theta', t)
theta
```

In the formulation of the Ordinary Differential Equation (ODE), the gravitational constant is denoted by `g`, and `L` is the length of the string. Applying Newton's laws leads to an equation of second order, involving the second derivative. A more geometrically precise model would have `-g*sin(theta)`, but for small angles theta we simplify `sin(theta)` to `theta`.

```
g, L = var('g, L', domain='positive')
```

Without declaring `g` and `L` as `positive` the solver will throw an exception. If the `domain='positive'` is omitted, then one can add the assumptions via `assume(g > 0, L > 0)`.

```
ode = L*diff(theta,t,2) == -g*theta
ode
```

The command to solve a differential equation is `desolve`. We have to tell `desolve` what is the independent variable, given below in the argument `ivar = t`.

```
sol = desolve(ode, theta, ivar=t)
sol
```

The answer we get is

```
_K2*cos(sqrt(g)*t/sqrt(L)) + _K1*sin(sqrt(g)*t/sqrt(L))
```

The `_K1` and `_K2` in the expression for the solution are general constants. We have two degrees of freedom, as expected for a second order equation. We can formulate an Initial Value Problem. As initial conditions, we assume that we move

the mass to an angle of `pi/10`, with velocity zero. Suppose that at `t = 0`, we have an initial angle of `pi/10`, thus: `theta(0) = pi/10` and its derivative is `0`, so we have `D[0](theta)(0) = 0`.

```
sol = desolve(ode, theta, ivar=t, ics=[0, pi/10, 0])
sol
```

and we see `1/10*pi*cos(sqrt(g)*t/sqrt(L))` as a general symbolic solution to the initial value problem. To plot the solution, we supply values for the gravitational constant `g` and the length of the string `L`.

```
s = sol.subs(g=10, L=3)
print(s)
plot(s, (t, 0, 2*pi))
```

The plot is shown in Fig. 3.36.

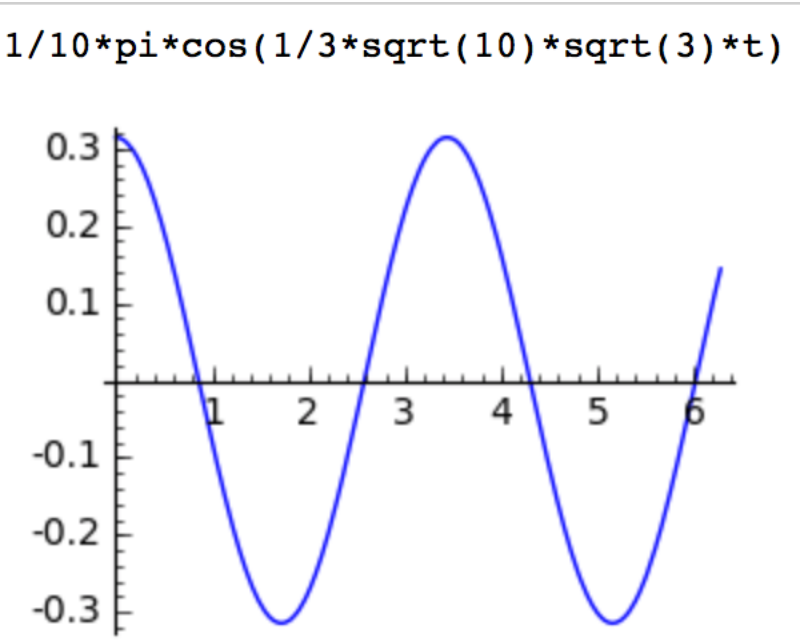

Fig. 3.36: A specific solution to the pendulum problem.

3.6.2 Plotting the Slope Field

If we solve an initial value problem for particular initial conditions, then we see only one particular solution. With the plot of the slope field we can visualize the whole family of solutions to the problem.

Consider the initial value problem

$$\frac{dy}{dt} = \frac{t^2 + t - y}{t}, \quad y(0) = 1, y'(0) = 3.$$

In formulating the problem, we assign the right hand side of the differential equation in a separate variable `rhs` because we will need this `rhs` in the plot of the slope field.

```
t = var('t')
y = function('y')(t)
```

(continues on next page)

(continued from previous page)

```
rhs = (t^2 + t - y)/t
ode = diff(y,t) == rhs
```

We solve the `ode` for `y`, with initial conditions in `ics = [1, 3]`, expressing that $y(0) = 1$ and $y'(0) = 3$.

```
sol = desolve(ode,y,ics=[1,3])
sol.show()
```

The solution `sol` is $\frac{2t^3+3t^2+13}{6t}$ and we notice the asymptote at $t = 0$. Therefore, for a nice plot of the solution, we give lower and upper bounds for the graph in `ymin` and `ymax`, restricting the viewing window to $[-50, 50]$. The range for t is $[-13, 13]$.

```
plotsol = plot(sol,(t,-13,13), ymin=-50, ymax=50, color='red')
plotsol.show()
```

The plot of the solution is shown in Fig. 3.37.

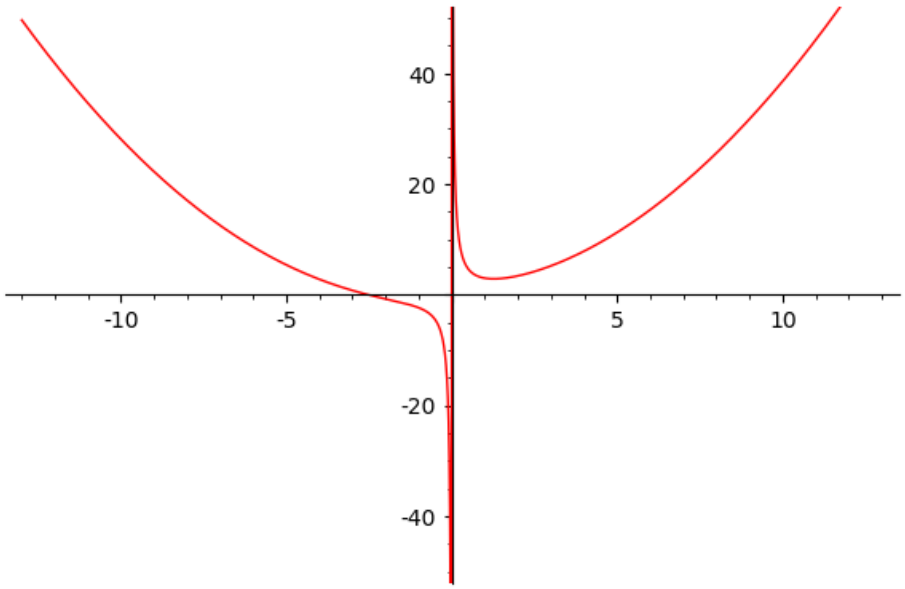

Fig. 3.37: A plot of a particular solution of an initial value problem.

To plot the slope field we need to take the right hand side of the ordinary differential equation (we saved this on purpose separately in the variable `rhs`) and replace the `y` by the solution we found.

```
plotslopes = plot_slope_field(rhs.subs({y:sol}), \
                              (t, -13, 13), (y, -50, 50), \
                              headlength=4, headaxislength=4, color='green')
plotslopes.show()
```

With the line breaks separating the arguments of `plot_slope_field` we indicate the three different types of parameters:

1. the right hand side of the ODE, with substituted `y` by the solution,
2. the ranges $t \in [-13, 13]$ and $y \in [-50, 50]$, and
3. the optional arguments for the size and color of the arrows.

The outcome of the above `plot_slope_field()` command is shown in Fig. 3.38.

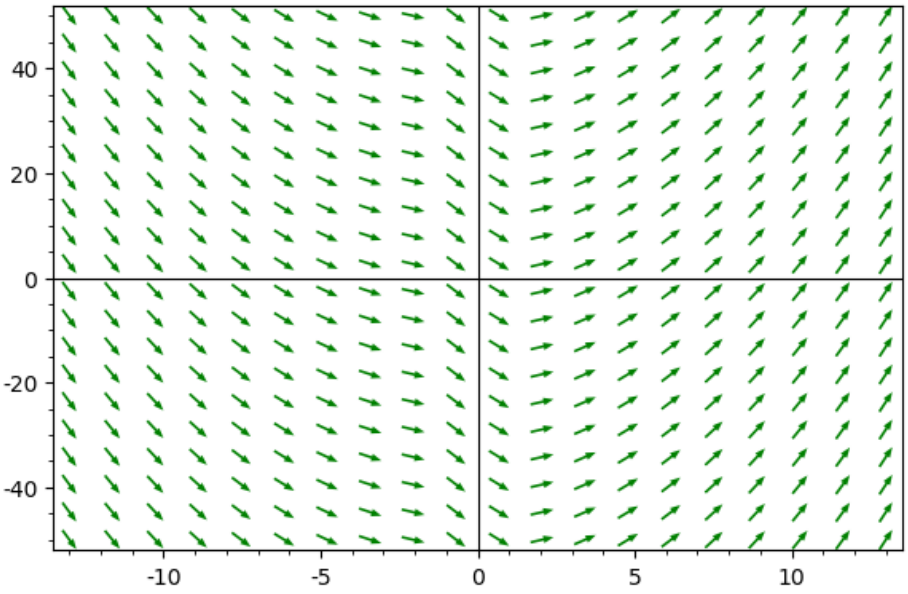

Fig. 3.38: The slope field of an initial value problem.

To plot both the particular solution and the slope field as shown in Fig. 3.39, we type `(plotsol+plotslopes).show()` and

3.6.3 Laplace Transforms

One symbolic way to solve differential equations is to use Laplace transforms. Consider a simple model of a swinging pendulum.

```
t = var('t')
theta = function('theta')(t)
g, L = var('g, L', domain='positive')
ode = L*diff(theta, t, 2) == - g*theta
sol = desolve_laplace(ode, theta, ivar=t)
```

The solution format has the symbols `theta(0)` and `D[0](theta)(0)` for the initial conditions $\theta(0)$ and $\theta'(0)$. Storing the initial conditions in a dictionary is convenient for later substitution to obtain a particular solution.

```
init = {theta(t=0): pi/10, diff(theta,t).subs({t:0}): 0}
sol.subs(init)
```

which gives `1/10*pi*cos(sqrt(L*g)(t/L)` as the solution with parameters `L` for the length of the string and `g` for the gravitational constant.

The Laplace transform brings the problem from the time domain into the frequency domain, where the problem can be solved better. We solve the same problem again, but now with the explicit computation of the transform, the solving in the frequency domain, and the application of the inverse Laplace transform. The symbol in the frequency domain is typically `s`.

```
s = var('s')
Lode = ode.laplace('t', 's')
```

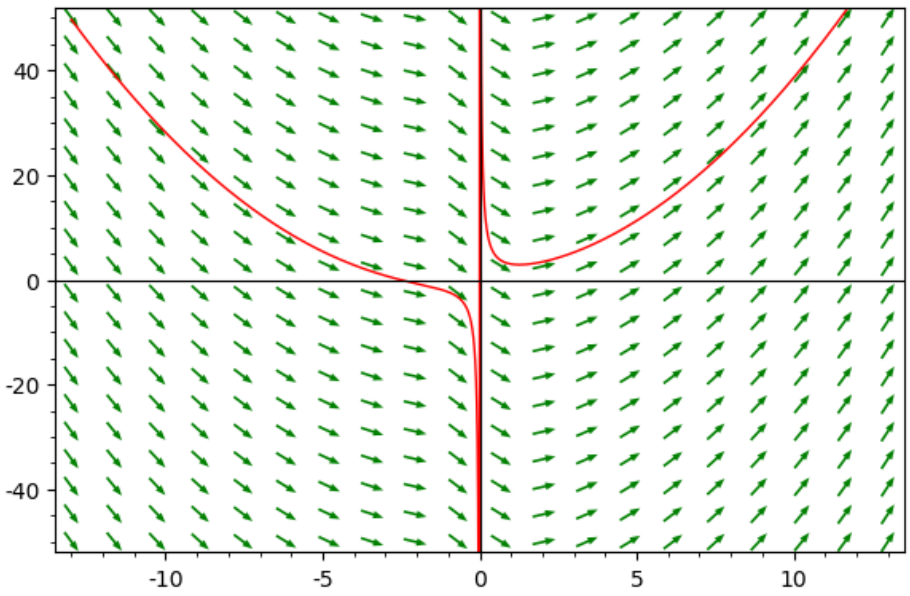

Fig. 3.39: A particular solution and the slope field of an initial value problem.

The `Lode` has `laplace(theta(t), t, s)` as the symbolic representation for the Laplace transform of `theta(t)`. To manipulate this object better, we abbreviate this transform by `T` via a substitution before solving. Observe that `Lode` is a *linear* equation in the Laplace transform.

```
T = var('T')
Teq = Lode.subs({laplace(theta, t, s): T})
sol = solve(Teq, T)
```

Selecting the right hand side of the solution, we substitute the initial conditions.

```
Ts = sol[0].rhs().subs(init)
ft = inverse_laplace(Ts, s, t)
```

In the last step, the solution was transformed from the frequency domain to the time domain.

3.6.4 Numerically Solving a First Order System

In this section we consider another model of the pendulum, which includes damping. Set $g/L = 10$ and assume the damping constant to be 1.2. The damping is proportional to the velocity.

$$\frac{d^2}{dt^2}\theta(t) = -1.2\frac{d}{dt}\theta(t) - 10\sin(\theta(t)).$$

To turn this is into a system of first order differential equations, we introduce

$$\frac{d}{dt}\theta(t) = v(t)$$

and obtain

$$\frac{d}{dt}v(t) = -1.2v(t) - 10\sin(\theta(t))$$

Observe that in the last two equations, a first order derivative appears at the left, while on the right only the unknown functions $\theta(t)$ and $v(t)$ appear. The format of a system of first order equations is needed for a numerical solver, which will return numerically values.

```
theta, v = var('theta, v')
rhs = [v, -1.2*v - 10*sin(theta)]
initc = [RR(pi)/10, 0.0]
```

In addition to the initial conditions, we must provide a range for the time interval.

```
endtime = 2*RR(pi)
step = 0.1
trange = srange(0, endtime, step)
```

The numerical solver applies `odeint()` of SciPy.

```
sol = desolve_odeint(rhs, initc, trange, [theta, v])
```

The `sol` is a matrix of two columns. The first column contains the values for `theta`, the second column has the values for the velocity `v`. After selecting the values for `theta`, we can plot.

```
angles = sol[:, 0]
pts = [(x, y) for (x, y) in zip(trange, angles)]
p = line(pts)
p.show()
```

The plot is shown in Fig. 3.40.

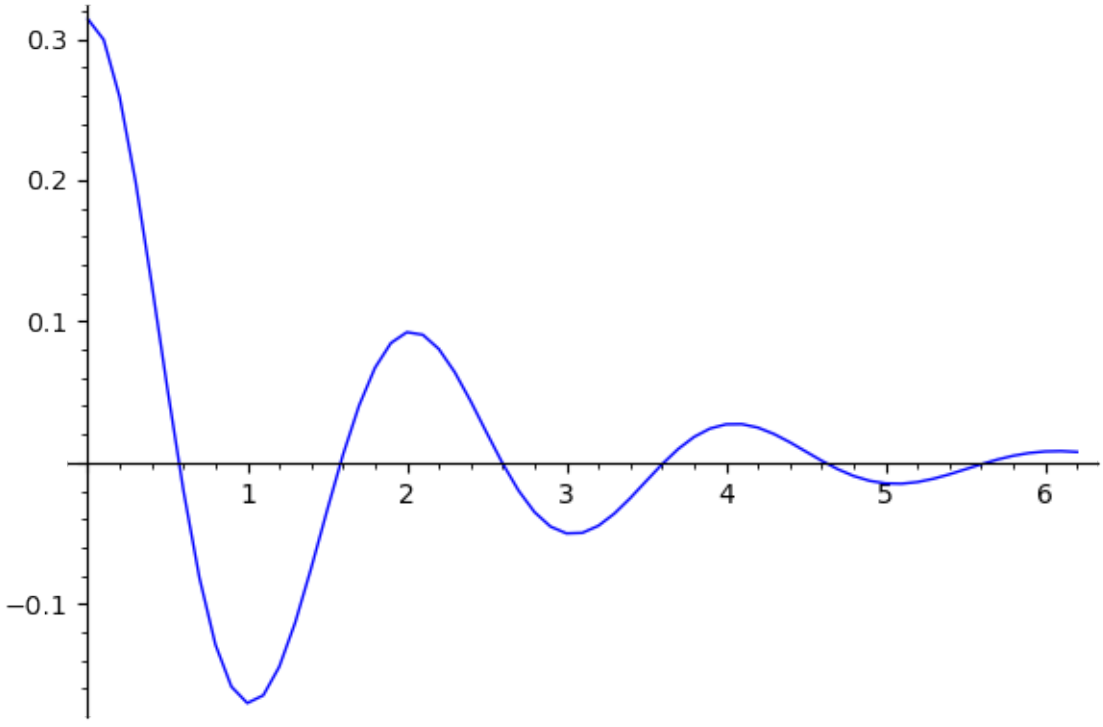

Fig. 3.40: Trajectory of a pendulum with damping.

3.6.5 Heat Diffusion

As an example of a boundary value problem, consider the diffusion of heat in a rod. We have a rod of length one. The rod experiences temperature differences at its ends. We are interested in the equilibrium temperature distribution, which in its simplest form is expressed as a second order differential equation.

```
x = var('x')
u = function('u', x)
ode = diff(u, x, 2) == 0
ode
```

As boundary conditions we impose u(0) = 2.5 and u(1) = 3.1.

```
sol = desolve(ode,u, ivar=x, ics=[0, 2.5, 1, 3.1])
sol
```

and we find as solution `3/5*x + 5/2`.

3.6.6 Assignments

1. Solve the following initial value problem:

$$y'' - y = 0, \quad y(0) = 1, \quad y'(0) = 0.$$

   Note that y' denotes the first derivative and y'' the second derivative with respect to some independent variable.

2. Solve the ordinary differential equation $3y^2y' + 16x = 12xy^2$, where y is a function of x.

   Describe the form of the computed solution. How can you verify that the expression returned is actually a solution to the equation?

3. Consider the initial value problem which models the motion of a pendulum $\frac{d\theta}{dt} = -g\theta(t)/L$, for $g = 10$ and $L = 3$. The initial values are $\theta(0) = \pi/10$ and $\theta'(0) = 0$. Make the plot of the slope field for $t \in [0, 2\pi]$ and $\theta \in [-0.4, 0.4]$. Add to this plot the particular solution to this problem.

4. Solve the initial value problem

$$\frac{d^2y}{dx^2} + 2\frac{dy}{dx} + 3y = \sin(x), \quad y(0) = 1, \quad y'(0) = 0.$$

   Plot the solution for $x \in [0, 20]$ and describe the shape of the solution.

5. Apply Laplace transforms to solve

$$\frac{d^2}{dt^2}y(t) + y - \sin(2t) = 0,$$

   in the following ways:

   1. Compute first the general solution with `desolve_laplace` and then substitute the initial values $y(0) = 2$ and $y'(0) = 1$ to obtain a particular solution.

   2. Instead of `desolve_laplace`, compute the Laplace transform of the equation, solve for the Laplace transform $Y(s)$ of $y(t)$, followed by the application of the inverse Laplace transform and the same initial values as before.

3.7 Lecture 31: Polyhedral and Unconstrained Optimization

Constrained optimization with Lagrange multipliers was covered at the end of the calculus chapter. Polyhedral optimization asks for the optimal value of a linear function, subject to constraints defined by linear inequalities. The simplex method solves polyhedral optimization problems defined in normal forms. When solving unconstrained optimization problems, the best we can hope to compute are local optima.

3.7.1 Polyhedra

A convex combination of two points is the line segment that has the two points as its ends. Given a set of points, the convex hull of the point is the set of all convex combinations of the points. For points in the plane, this convex hull is a polygon. We can draw a polygon by giving the vertices (or corner points).

Every polyhedron can be defined in two ways:

1. As the convex hull of a list of points.
2. As the intersection of half planes.

The *V-representation* of a polyhedron P are the vertices that span P (excluding points in the interior). The *H-representation* of a polyhedron P are the linear inequalities that define the half planes of the intersection that cuts out P.

Let us verify this on 10 random points with integer coordinates between 10 and 99.

```
set_random_seed(20220722)
xvals = [ZZ.random_element(x=10, y=99) for _ in range(10)]
yvals = [ZZ.random_element(x=10, y=99) for _ in range(10)]
pts = [[x, y] for (x, y) in zip(xvals, yvals)]
```

The `pts` contains the list of the points that span the polygon. A polyhedron in the plane spanned by finitely many points is called a polygon.

```
P = Polyhedron(vertices = pts)
P.plot()
```

The polygon is shown in Fig. 3.41.

```
P.Vrepresentation()
```

The V-representation of `P` consists of four vertices, as we can see from the four corners in Fig. 3.41.

```
P.Hrepresentation()
```

The H-representation of `P` consists of four linear inequalities, as we can see from the four edges in Fig. 3.41.

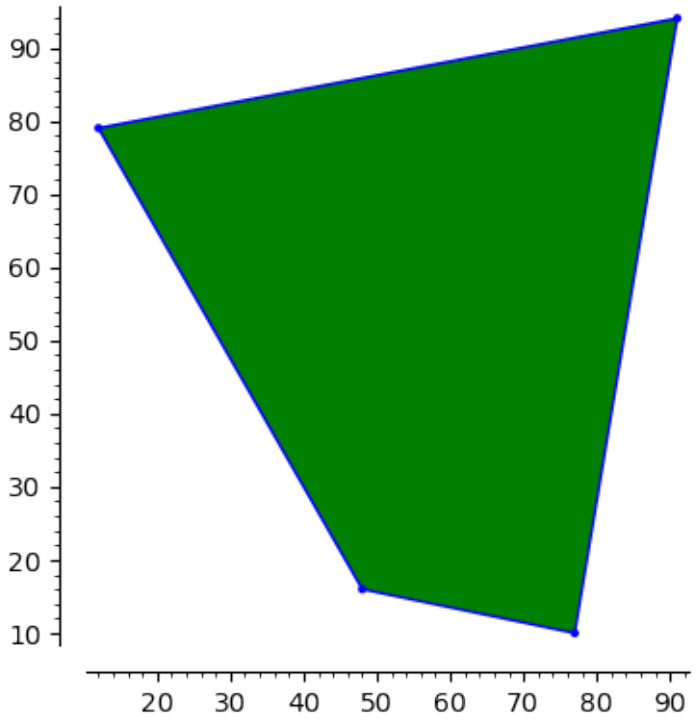

Fig. 3.41: A polygon generated by ten random integer points.

3.7.2 Linear Programming

Linear programming optimizes linear functions subject to linear constraints. The following example of a linear program is taken from the Sage documentation.

$$\begin{array}{l} \max x + 5y \\ \text{subject to} \\ \left\{ \begin{array}{rcl} x + 0.2y & \leq & 4 \\ 1.5x + 3y & \leq & 4 \end{array} \right. \end{array}$$

where the variables x and y are both real and positive. In Sage this can be solved as follows:

```
p = MixedIntegerLinearProgram(maximization=True)
x = p.new_variable(nonnegative=True)
p.set_objective(x[1] + 5*x[2])
p.add_constraint(x[1] + 0.2*x[2], max=4)
p.add_constraint(1.5*x[1] + 3*x[2], max=4)
p.solve()
```

and we see `6.666666666666666` as the solution. This solution is the value of the objective at the coordinates of the solution.

The coordinates of the solution can be obtained as follows.

```
p.get_values(x)
```

and we obtain the dictionary `{1: 0.0, 2: 1.3333333333333333}`, implying that with `0.0` as the value for `x[1]` and `1.3333333333333333` as the value for `x[2]` the objective attains the optimal value `6.666666666666666`.

Let us look at this problem a bit in greater detail.

```
A = ([1, 0.2], [1.5, 3])
b = (4, 4)
c = (1, 5)
P = InteractiveLPProblem(A, b, c, ["x1", "x2"], problem_type="max", \
              constraint_type="<=", variable_type=">=")
P.plot()
```

This linear programmig problem is shown in Fig. 3.42.

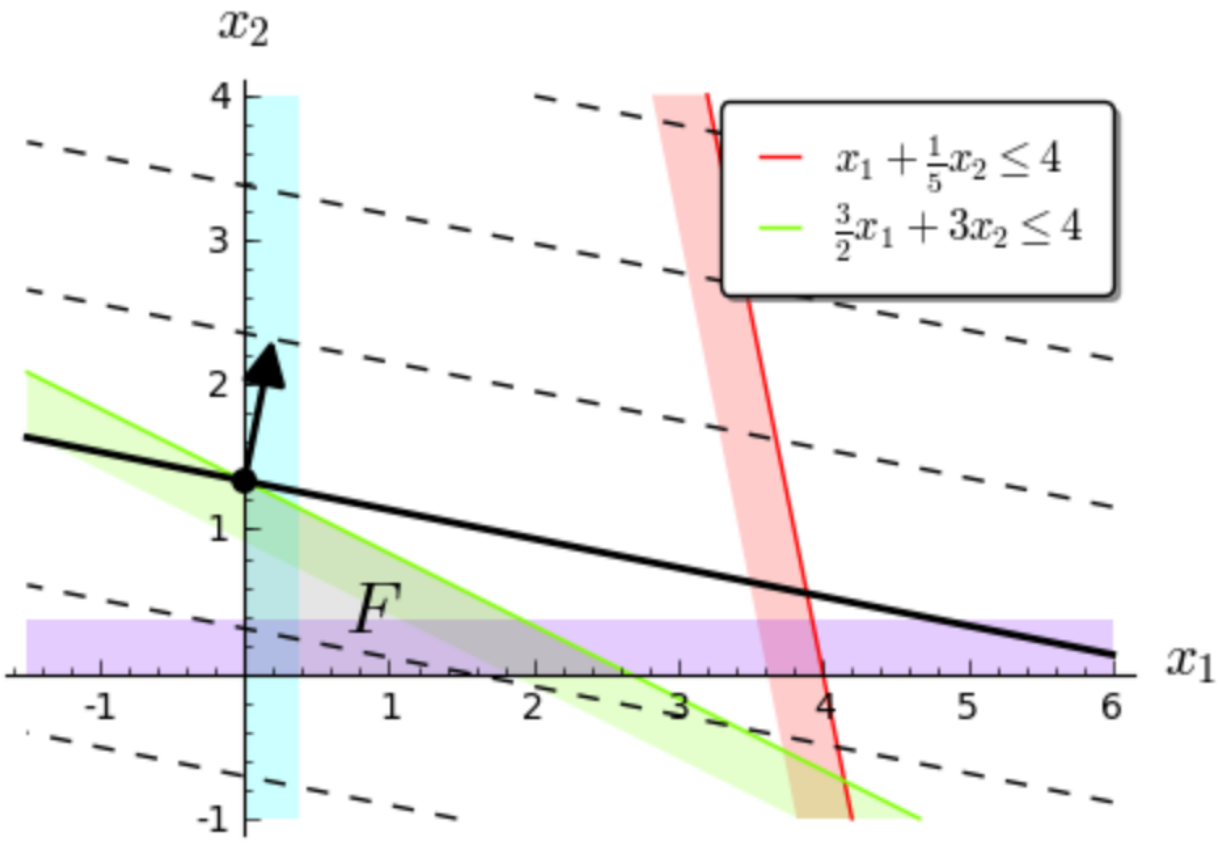

Fig. 3.42: A linear programming problem with the function to optimize drawn in black.

We first convert the problem to its standard form.

```
P = P.standard_form()
show(P)
```

The standard form is shown in Fig. 3.43.

The interactive run of the simplex method shows the succession of dictionaries. Let us start with the initial dictionary.

```
D = P.initial_dictionary()
show(D)
```

The initial dictionary is in Fig. 3.44.

We ask for the solution, whether the solution is optional, and the value of the objective function.

$$
\begin{aligned}
\max \quad & x_1 + 5.000000000000000x_2 \\
& x_1 + 0.2000000000000000x_2 \leq 4.000000000000000 \\
1.500000000000000x_1 + & \ 3.000000000000000x_2 \leq 4.000000000000000 \\
x_1, x_2 \geq 0 &
\end{aligned}
$$

Fig. 3.43: The standard form of the LP problem of Fig. 3.42.

$$
\begin{array}{rrrrr}
x_3 = & 4.000000000000000 - & & x_1 & - 0.2000000000000000x_2 \\
x_4 = & 4.000000000000000 & - 1.500000000000000x_1 & - & 3.000000000000000x_2 \\
\hline
z = & 0 + & & x_1 & + 5.000000000000000x_2
\end{array}
$$

Fig. 3.44: The initial dictionary to solve the LP problem in Fig. 3.42.

```
print(D.basic_solution())
print(D.is_optimal())
print(D.objective_value())
```

We see `(0.000000000000000, 0.000000000000000)`, `False`, and `0` as the output of the `print` statements. The simplex algorithm consists in swapping variables. Variables in the basis are swapped with variables not in the basis. We can ask for the possible variables which can enter the basis.

```
print(D.basic_variables())
print(D.nonbasic_variables())
etr = D.possible_entering()
etr
```

The output of the `print` statements is `(x3, x4)`, `(x1, x2)`, and `[x1, x2]`. We select the first of the possible entering variables. After setting the entering variable, we ask for the possible variables which can leave the basis.

```
D.enter(etr[0])
lev = D.possible_leaving()
lev
```

There is only one possible leaving variable `x4`. We select the leaving variable and update the dictionary.

```
D.leave(lev[0])
D.update()
show(D)
```

The updated dictionary is shown in Fig. 3.45.

Should we continue? Let us check if the dictionary is optimal and what the objective value is.

```
print(D.is_optimal())
print(D.objective_value())
```

We see `False` and `2.66666666666667` printed, so we have to continue. To continue, let us ask for the possible variables which may enter.

$$\begin{array}{rl} x_3 = 1.33333333333333 & +0.666666666666667x_4 + 1.80000000000000x_2 \\ x_1 = 2.66666666666667 & -0.666666666666667x_4 - 2.00000000000000x_2 \\ \hline z = 2.66666666666667 & -0.666666666666667x_4 + 3.00000000000000x_2 \end{array}$$

Fig. 3.45: The updated dictionary after swapping an entering with a leaving variable.

```
etr = D.possible_entering()
etr
```

There is only one entering variable `x2`. After selecting the entering variable, we ask for the possible variables which may leave.

```
D.enter(etr[0])
lev = D.possible_leaving()
lev
```

The leaving variable is `x1`. After selecting the leaving variable, we update the basis.

```
D.leave(lev[0])
D.update()
show(D)
```

The updated dictionary is shown in Fig. 3.46.

$$\begin{array}{rl} x_3 = 3.73333333333333 & +0.0666666666666667x_4 - 0.900000000000000x_1 \\ x_2 = 1.33333333333333 & - 0.333333333333333x_4 - 0.500000000000000x_1 \\ \hline z = 6.66666666666667 & - 1.66666666666667x_4 - 1.50000000000000x_1 \end{array}$$

Fig. 3.46: The updated dictionary after swapping `x2` for `x1`.

Then we check again for optimality.

```
print(D.is_optimal())
print(D.objective_value())
```

and we see `True` and `6.66666666666667` as the objective value.

3.7.3 Unconstrained Minimization

We covered the technique of Lagrange multipliers already to solve a constrained optimization problem. As an example we take a sum of squares.

```
x, y = var('x,y')
f(x,y) = (3+x-y^2)^2 + (x - 1)^2 + (y - 1)^2
pf = plot3d(f(x,y), (x, -2, 3), (y, -2, 3), adaptive=True, color='automatic',opacity=0.5)
pf.show()
```

The function `f(x,y)` defines the surface $z = f(x, y)$. Minimizing this function has a geometric interpretation. The surface is displayed in Fig. 3.47.

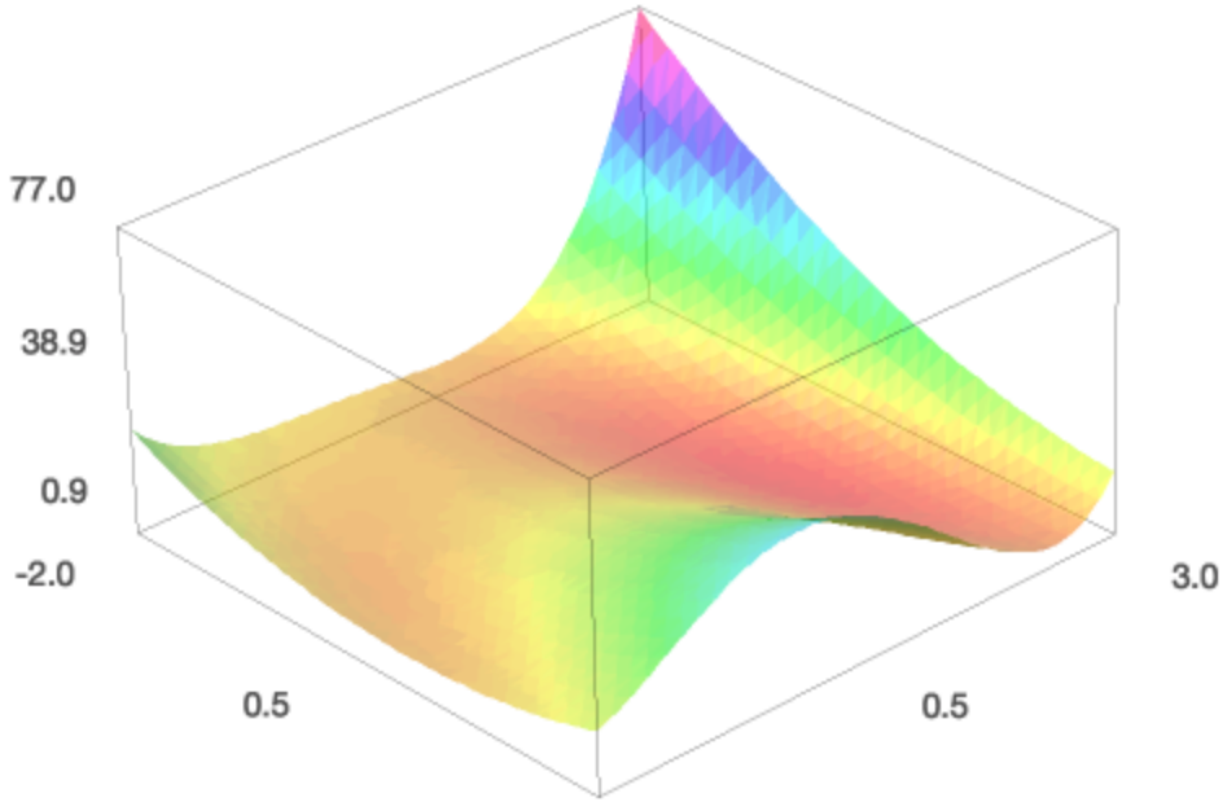

Fig. 3.47: Looking for the location with the smallest height on a surface.

We can look for a minimum. We need to give an initial guess.

```
p = minimize(f, x0=[0.0, 0.0], verbose=True)
print('minimum at', p, 'with value', f(p[0], p[1]))
```

Then Sage prints

```
Optimization terminated successfully.
         Current function value: 0.882789
         Iterations: 10
         Function evaluations: 14
         Gradient evaluations: 14
minimum at (0.766044177107, 1.87938504258) with value 0.882788808286
```

To get an idea how good this minimum is we take a look at where the location on the minimum on the surface.

```
pt = point((p[0], p[1], f(p[0],p[1])), size=20, color='black')
(pt+pf).show()
```

The point on the surface is shown in Fig. 3.48.

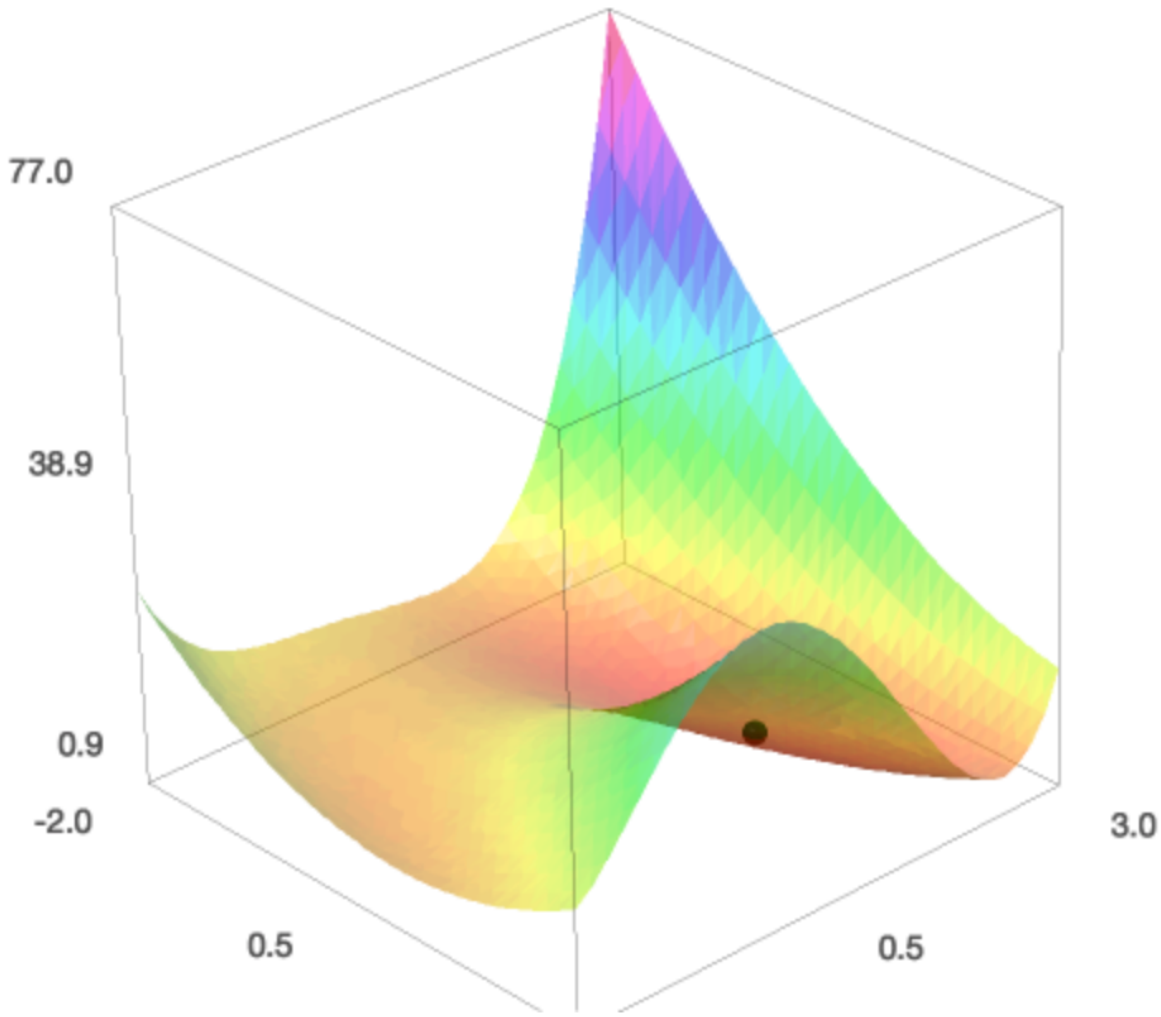

Fig. 3.48: The computed minimum shown as a black point on the surface.

3.7.4 Assignments

1. Suppose General Motors makes a profit of $100 on each Chevrolet, $200 on each Buick, and $400 on each Cadillac. These cars get 20, 17, and 14 miles a gallon respectively, and it takes respectively 1, 2, and 3 minutes to assemble one Chevrolet, one Buick, and one Cadillac. Assume the company is mandated by the government that the average car has a fuel efficiency of at least 18 miles a gallon. Under these constraints, determine the optimal number of cars, maximizing the profit, which can be assembled in one 8-hour day. Give all Sage commands and the final result.
2. Take the gradient of `f(x,y) = (3+x-y^2)^2 + (x - 1)^2 + (y - 1)^2` and solve the polynomial system defined by setting the gradient to zero. The solutions of this system contain the minima of the function. Do you find the outcome of `minimize` among the solutions of the system?
3. Suppose an invester has a choice between three types of shares. Type A pays 4%, type B pays 6%, and type C pays 9% interest. The investor has $100,000 available to buy shares and wants to maximize the interest, under the following constraints:

   1. no more than $20,000 can be spent on shares of type C;
   2. at least $10,000 of the portfolio should be spent on shares of type A.

   Answer the following questions:

   1. Give the mathematical description of the optimization problem.
   2. Bring the problem into the standard form.
   3. Solve the linear programming problem.
4. Maximize $2x_1 + x_2$ subject to $-x_1 + 2x_2 \leq 8$, $4x_1 - 3x_2 \leq 9$, $2x_1 + x_2 \leq 13$, $x_1 \geq 0$, and $x_2 \geq 0$.

3.8 Lecture 32: Review of Lectures 18 to 31

Below is a first, preliminary list of equations to review in preparation of the second midterm exam. Consider also the quizzes and homework assignments.

1. Count the number of points with integer coordinates (x, y), in the region defined by the inequalities $0 \leq x < 20$, $0 \leq y < 20$, $y \geq x/2$, and $y \leq 3x$. Give the Sage commands (not the output) for the three stages below.

   (a) Generate a list L of integer points (i, j) for i and j ranging between 0 and 19.

   (b) Select from the list L those points in the cone $y \geq x/2$ and by $y \leq 3x$.

   (c) Count the number of points in the cone. Write also the number below.
2. For some parameter t, consider the sequence recursively defined as:

$$F_n = (1-t)F_{n-1} + tF_{n-2}, \text{ for } n > 1,$$

   where $F(0) = a$ and $F(1) = b$. Using the recursive definition write an efficient Sage function F to compute F_n as `F(a, b, t, n)`. What is the result of `F(a, b, 0.3, 100)`?
3. Consider the function $f(t) = \int_0^t (1 - e^x)dx$, for $t \geq 0$. Define this function in Sage. What is $f'(1)$?
4. The function $g(x,t) = \dfrac{1-t^2}{1-2xt+t^2}$ is a generating function for the Chebyshev polynomials. Compute a Taylor series approximation for $g(x, t)$ around $t = 0$ of order 10. Select the coefficient of t^8 and compare with the output of `chebyshev_T(8, x)`. Is there a difference between the two?
5. Consider the point $(1, 1)$ on the curve $f(x, y) = x^2 - y^3 - x + y = 0$.

(a) Give the Sage command(s) to compute a Taylor series about the point $(1, 1)$ where the term of the error is of second order.

(b) Compute the slope of the tangent line of the curve at the point $(1, 1)$ and use the slope to determine the tangent line. Write the equation of the tangent line.

Verify that the equation for the tangent line corresponds to the first-order Taylor series at $(1, 1)$.

6. Consider the curve $x^4 - 3xy + y^4 = 0$. Give all Sage commands to

   (a) to make a plot for x and y both ranging between -2 and $+2$;

   (b) to convert the curve into polar coordinates; and

   (c) to plot the curve in polar coordinates.

7. Consider $p = 5x^2a^2 + 61x^2a + 66x^2 + 10xa^2 + 121xa + 121x + a^2 + 15a + 44$, as a polynomial in x with parameter a.

   (a) Find the roots of p.

   (b) For which values of the parameter a is the answer valid?

   (c) Give the Sage command(s) to treat the special case(s).

   (d) As you can see the polynomial p is shown in expanded form. Give the Sage command(s) to ''un-expand", i.e.: what is the command which reveals better the structure of p?

8. Let a and b be positive numbers. Consider $f = x^2/a + y/b$ and the unit circle $x^2 + y^2 = 1$. Give all Sage commands to determine the number of candidate extremal values of f on the unit circle. Use a lexicographic Groebner basis to compute a triangular form of the equations for this problem.

9. Give the Sage commands for the following tasks. Create a 5-by-5 matrix A over the rationals where the (i, j)-the element is $1/(i + j)$ (for i and j both from 1 to 5). Define b as a vector of length 5 of ones. Solve the system defined by $Ax = b$. Verify that $b - Ax$ equals zero.

10. Consider the intial value problem $dy/dt = 2 - 6y$, $y(0) = -1$.

    (a) Solve this problem and plot the solution trajectory for $t \in [0, 2]$.

    (b) Plot the slope field for $t \in [0, 2]$ and $y \in [-1, 0.5]$. Place also the particular solution computed in (a) on the plot.

11. Minimize $x + 3y$

    subject to $x \geq 2$, $y \geq 1$, $x + 2y \leq 8$, $x + y \leq 6$.

    Formulate the linear programming problem and solve it.

3.9 Lecture 33: the Second Midterm Exam

The second midterm exam covers parts three and four of the course.

3.9.1 Questions on the Spring 2017 Second Midterm Exam

The list of questions on the Spring 2017 second midterm exam are below. The exam is open book, open notes and open computer. All answers to the questions must be handwritten, submitted on paper.

1. Consider the formula

$$\ln(2) = \sum_{k=1}^{\infty} \frac{1}{k2^k}.$$

   Make a function `f(k)` that returns the value of $1/(k2^k)$.

   Use this function in a list comprehension to compute the first 100 bits of $\ln(2)$.

   How accurate is this approximation for $\ln(2)$? Write the magnitude of the error.

2. The recurrence

$$S(n) = \frac{1}{n}((6n-9)S(n-1) - (n-3)S(n-2))$$

   is valid for $n > 1$.

   For $n = 0$ and $n = 1$, we have $S(0) = 1$ and $S(1) = 1$.

   Use the recurrence for S to define an *efficient* function to compute $S(n)$.

   How many decimals do you see in the number $S(100)$?

3. Consider the curve $y^4 - y^3 + 2x^2y^2 + 3x^2y + x^4 = 0$.

   1. Give the command(s) to make a plot of this curve, for $x \in [-1, +1]$ and $y \in [-0.7, +1]$.
   2. Write the representation of this curve in polar coordinates. Give all relevant commands you used.
   3. What is the command to plot this curve in polar coordinates?

4. Consider the Taylor series of $\log(1+x)$ at $x = 0$.

   Use `sympy` to define a generator for the next term in the Taylor series.

   Illustrate the use of this generator to compute a list of the first 10 terms.

5. Let $f(x, y) = x^4 + x^2y^2 + y^4 - x(x^2 + y^2)$ and $P = (2/3, 2/3)$.

   Compute the slope of the tangent line of the curve defined by f at the point P.

   Write the value for the slope and the commands you used.

6. Consider the polynomial system defined by $p = x^3y - 3x^2 + 9$ and $q = 4xy^2 + xy - 7$.

   Construct a lexicographic Groebner basis for the ideal defined by the polynomials p and q.

   How many solutions does the polynomial system have?

7. Let A be an n-dimensional matrix with its elements indexed by i and j, defined by:

   1. For $i = j$, the (i, j)-th element of A is zero.
   2. For $i \neq j$, the (i, j)-th element of A is $+1$ if $i + j$ is even and -1 if $i + j$ is odd.

   Write all commands for the following:

   1. Define the matrix A, for $n = 17$. Let $\mathbf{b}$ be a 17-dimensional vector of ones.
   2. Solve the linear system $A\mathbf{x} = \mathbf{b}$. Verify the residual: the norm of $\mathbf{b} - A\mathbf{x}$.

      Write the sum of the coordinates in the solution $\mathbf{x}$.

3.9.2 Questions on the Fall 2018 Second Midterm Exam

The list of questions on the Fall 2018 second midterm exam are below. The exam is open book, open notes and open computer. All answers to the questions must be handwritten, submitted on paper.

1. Define a function f which takes on input a list of variables $[x_1, x_2, \ldots, x_n]$ and an index k. The function $f([x_1, x_2, \ldots, x_n], k)$ returns the expression

$$x_k \sum_{\substack{i=1 \\ i \neq k}}^{n} x_i^2 - x_k + 1 = x_k \left(-x_k^2 + x_1^2 + x_2^2 + \cdots + x_n^2 \right) - x_k + 1.$$

   Write the definition of the function f.

2. The number of ways to put parentheses in a chain of multiplications of n matrices of compatible dimension equals $f(n)$. The recursive definition for $f(n)$ is

$$f(1) = 1 \quad \text{and for } n > 1, \quad f(n) = \sum_{k=1}^{n-1} f(k) f(n-k).$$

   1. Write an efficient recursive definition of the function f.
   2. How many decimal places does the number $f(100)$ have?

3. Consider the curve defined by the equation $xy - x^2 - y^3 + 1 = 0$ and the point $(1, 1)$.
   1. Compute the equation of the tangent line to the curve at the point.
   2. What is the slope of the tangent line?

4. Write the commands to set up the system that needs to be solved to compute the point on the curve $xy - x^2 - y^3 + 1 = 0$ closest to $(0, 0)$. Do not solve the system.

5. Consider the curve defined by $(x^2 + y^2)(y^2 + x(x + 1)) - 4xy^2 = 0$.
   1. Make a plot of the curve for x and y both in $[-3/2, 3/2]$.

      Choose sufficiently many points to obtain a smooth plot.
   2. Convert the formulation of the curve into polar coordinates.
   3. Write the command to make the plot in polar coordinates.

   How many times does the curve pass through the origin?

6. Consider $p = x^3y + 2xy - 3y^2 + 1$ and $q = 2xy^2 - 5y^2 + 7y - 3$ as polynomials with rational coefficients.
   1. Write all relevant commands to compute a Groebner basis for a lexicographical term order.
   2. How many solutions with complex coordinates do you expect? Explain!

7. The (i, j)-th entry in an n-dimensional Redheffer matrix, for i and j both ranging between 1 and n, is one if i divides j or if $j = 1$, and zero otherwise. Write all relevant commands for the following.
   1. Define a 10-by-10 Redheffer matrix stored as a matrix A.
   2. Define a right hand side vector $\mathbf{b}$ of all ones. Solve the linear system $A\mathbf{x} = \mathbf{b}$. Compute the residual vector and its norm.

3.9.3 Questions on the Spring 2019 Second Midterm Exam

The list of questions on the Spring 2019 second midterm exam are below. The exam is open book, open notes and open computer. All answers to the questions must be handwritten, submitted on paper.

1. For any two positive natural numbers n and d, consider

$$f_k(x_1, x_2, \ldots, x_n) = \left(\prod_{i=0}^{d} x_{[(k+i) \bmod (n+1)]} \right) - x_k + 1 = x_k x_{k+1} \cdots x_{(k+d) \bmod (n+1)} - x_k + 1,$$

   for $k = 1, 2, \ldots, n$. Give a definition of a function `f` which takes on input $X = [x_1, x_2, \ldots, x_n]$, a list of n variables, and k, n, d. The function `f`:math:(*X,k,n,d*) returns $f_k(x_1, x_2, \ldots, x_n)$.

2. Bessel polynomials are defined recursively as $B_0(x) = 1$, $B_1(x) = x + 1$, and $B_n(x) = (2n-1)xB_{n-1}(x) + B_{n-2}(x)$, for $n > 1$.

   Write the definition of an *efficient, recursive* function `B` which takes on input n and x. `B`(n, x) returns $B_n(x)$. What is the coefficient of x in $B_{50}(x)$?

3. Consider the point $P = (2, 1)$ on the curve $x^2 - y^3 - y - 2 = 0$.

   Compute the slope of the tangent line at P to the curve. Write all relevant Sage commands. What is the value of the slope?

4. Describe the functionality of the `assume` command.

   Give an example of a good application. What problem does `assume` solve?

5. Consider the curve defined by $x^6 + 3x^4y^2 + 3x^2y^4 + y^6 - 4x^5 + 4xy^4 + 4x^4 - 12x^2y^2 = 0$.

   1. Write the instruction to draw the curve as given by the polynomial above.

      Does the curve pass through the origin?

   2. Convert the polynomial into polar coordinates.

      Write the expression of the curve in polar coordinates.

   3. Plot the curve in polar coordinates.

      How may times does the curve pass through the origin?

6. Consider the surface $x^2 + y^2 - z^2 = 0$ and the point $P = (3, 2, 1)$.

   Find the point on the surface closest to P.

   1. Write the commands to set up the system that needs to be solved.
   2. What solving command(s) do you use?
   3. What is the (approximate) distance of the closest point?

7. Maximize $8x_1 + 11x_2$ subject to $x_1 + x_2 \leq 8$, $-x_1 + 3x_2 \leq 12$, $x_1 \geq 0$, and $x_2 \geq 0$.

   Write the commands to define this problem and then solve it.

   What is the value of the optimal solution?

3.9.4 Questions on the Summer 2022 Second Midterm Exam

The exam in the summer 2022 took 100 minutes and was administered online, in two different versions. Questions had to be uploaded into one single Jupyter notebook into gradescope.

The list of the nine questions on the first version is below.

1. Consider the general formula, for positive integers k, n, and d:

$$f_k(x_1, x_2, \ldots, x_n) = x_k^d + \sum_{i=1}^{d} \left[x_{(k+i) \bmod n} \left(1 - x_{(k+i+1) \bmod n} \right) \right] - 1.$$

   Give a definition of a function `f` which takes on input $X = [x_1, x_2, \ldots, x_n]$, a list of n variables, and k, d, n.

   The function `f`(X, k, d, n) returns $f_k(x_1, x_2, \ldots, x_n)$.

   Show the output of your function for $d = 3$, $n = 8$, and $k = 5$.

2. The sequence of Fibonacci polynomials is defined as

$$f_0(x) = 0, f_1(x) = 1, \text{ and } f_n(x) = x f_{n-1}(x) + f_{n-2}(x), \text{ for all } n > 1.$$

   Write the definition of an *efficient, recursive* function `F` which takes on input n and x.

   `F`(n, x) returns $f_n(x)$. Compute `F`$(50, x)$.

3. Consider the point $P = (2, 1)$ on the curve $x^2y^3 - y^2 - 2x + 1 = 0$.

   Compute the slope of the tangent line at P to the curve.

4. Explain the difference between a symbolic factorization and a symbolic-numeric factorization of a polynomial in one variable.

   Illustrate the difference with a good example.

5. Plot the surface defined by $x(u, v) = 2u/(u^2 + v^2 + 1)$, $y(u, v) = 2v/(u^2 + v^2 + 1)$, and $z(u, v) = (u^2 + v^2 - 1)/(u^2 + v^2 + 1)$, for u and v both in the interval $[-4, 4]$.

   Describe what you see. Can you name the surface?

6. Consider the n-dimensional tridiagonal matrix A_n with x on the diagonal, and ones on the upper and lower diagonal. For example, for $n = 5$:

$$A_5 = \begin{bmatrix} x & 1 & 0 & 0 & 0 \\ 1 & x & 1 & 0 & 0 \\ 0 & 1 & x & 1 & 0 \\ 0 & 0 & 1 & x & 1 \\ 0 & 0 & 0 & 1 & x \end{bmatrix}.$$

   1. Write the instructions to define A_n.

      Plain typing all elements in all rows and columns will receive no credit, your solution should scale well for any dimension.

   2. Compute the determinant of A_5.

7. Consider the surface $z = x^2 - y^2$ and the point $P = (2, 1, 3)$.

   Find the point on the surface closest to P.

   1. Formulate the target function (square of the distance to P) and solve the problem.
   2. What is the (approximate) distance of the closest point?

8. What is the Laplace transform of a function?

   Illustrate your explanation with a good example.

9. Maximize $2x_1 + x_2$

   subject to $-x_1 + 2x_2 \leq 8$, $4x_1 - 3x_2 \leq 9$, $2x_1 + x_2 \leq 13$, $x_1 \geq 0$, and $x_2 \geq 0$.

   Define this problem and then solve it.

The list of the nine questions on the second version is below.

1. Consider the general formula, for positive integers k, n, and d:

$$f_k(x_1, x_2, \ldots, x_n) = x_k^d + \sum_{i=1}^{d} \left(\frac{x_{(k+i) \bmod n}}{x_{(k+i+1) \bmod n}} \right) - 1.$$

   Give a definition of a function `f` which takes on input $X = [x_1, x_2, \ldots, x_n]$, a list of n variables, and k, d, n.

   The function `f`(X, k, d, n) returns $f_k(x_1, x_2, \ldots, x_n)$.

   Show the output of your function for $d = 3$, $n = 8$, and $k = 5$.

2. The sequence of Lucas polynomials is defined as

$$\ell_0(x) = 2, \ell_1(x) = x, \text{ and } \ell_n(x) = x\ell_{n-1}(x) + \ell_{n-2}(x), \text{ for all } n > 1.$$

   Write the definition of an *efficient, recursive* function `L` which takes on input :math:*n* and x.

   `L`(n, x) returns $\ell_n(x)$. Compute `L`$(50, x)$.

3. Consider the point $P = (3, 1)$ on the curve $x^2 - 2xy + y^3 - 4 = 0$.

   Compute the slope of the tangent line at P to the curve.

4. Explain the difference between a numeric factorization and a symbolic-numeric factorization of a polynomial in one variable.

   Illustrate the difference with a good example.

5. Plot the space curve defined by $(\pm\sqrt{1 - t^4}, t^2, t)$, for $t \in [-1, +1]$, using the color red.

   Demonstrate with a plot that this curve lies in the intersection of the cylinders $x^2 + y^2 - 1 = 0$ and $y - z^2 = 0$, using blue and green to color the cylinders.

6. Consider the *n*-dimensional tridiagonal matrix A_n with $2x$ on the diagonal, and ones on the upper and lower diagonal. For example, for $n = 5$:

$$A_5 = \begin{bmatrix} 2x & 1 & 0 & 0 & 0 \\ 1 & 2x & 1 & 0 & 0 \\ 0 & 1 & 2x & 1 & 0 \\ 0 & 0 & 1 & 2x & 1 \\ 0 & 0 & 0 & 1 & 2x \end{bmatrix}.$$

   1. Write the instructions to define A_n.

      Plain typing all elements in all rows and columns will receive no credit, your solution should scale well for any dimension.

   2. Compute the determinant of A_5.

7. Consider the surface $z = x^2 - y^2$ and the point $P = (3, 2, 1)$.

   Find the point on the surface closest to P.

   1. Formulate the target function (square of the distance to P) and solve the problem.

2. What is the (approximate) distance of the closest point?

8. Consider the initial value problem

$$\frac{d^2y}{dx^2} + 2\frac{dy}{dx} + 3y = \sin(x), \quad y(0) = 1, \quad y'(0) = 0.$$

Define this problem and solve it.

Plot the solution for $x \in [0, 20]$ and describe the shape of the solution.

9. What is the V-representation of a polygon?

Illustrate your explanation with a good example.

CHAPTER 4

Part Five : Advanced Topics

The goal of this part is to look more how SageMath connects to other software packages and to cover some advanced topics, and to deepen our knowledge about earlier issues. We start by making interactive web pages. Then we cover sympy, numpy, and scipy. The last lectures are spent on GAP, PARI/GP, Singular, and R.

4.1 Lecture 34: Building Interactive Web Pages

We can view an interactive web page as a graphical user interface, with the main difference that we need a browser to run the interaction.

At UIC, the computer `people.uic.edu` offers web hosting to students, staff and faculty. If Apache runs on your computer, then you can run the web pages by pointing your browser to localhost.

We follow chapter 6 in the book of Gregory Bard and work through the tangent-line interact.

4.1.1 Drawing Tangent Lines to a Curve

The code below is adapted from *Sage for Undergraduates* by Gregory Bard, pages 285 in Figure 1.

```
f(x) = x^3 - x
df(x) = diff(f(x), x)
def tangent_at_point(x0):
    """
    Shows the tangent line at the point x0,
    using f and its derivative as global variables.
    """
    y0 = f(x0)
    slope = df(x0)
    # y - y0 = slope*(x - x0) implies
    # y = slope*x - slope*x0 + y0
    b = y0 - slope*x0
```

(continues on next page)

(continued from previous page)

```
    P1 = plot(f, -2, 2, color='blue', ymin=-6, ymax=6, gridlines='minor')
    P2 = plot(slope*x + b, -2, 2, color='tan', ymin=-6, ymax=6)
    P3 = point((x0, y0), color='red', size=50)
    P = P1+P2+P3
    P.show(figsize=3)
```

To test the script we run the following cases in separate cells.

```
tangent_at_point(0.5)
tangent_at_point(1.5)
tangent_at_point(-1.75)
```

The three figures for the x-coordinates 0.5, 1.5, and -1.75 are respectively displayed in Fig. 4.1, Fig. 4.2, and Fig. 4.3.

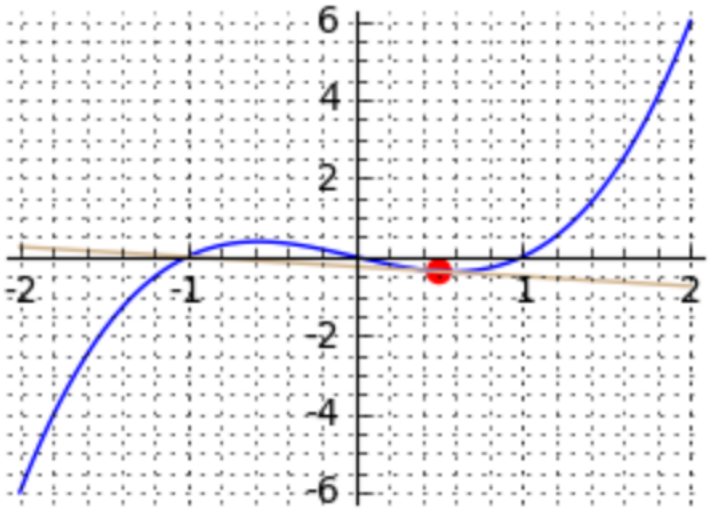

Fig. 4.1: The tangent line of $x^3 - x$ for $x = 0.5$.

We want to attach a scale to the `x0` parameter of the function. When the user then moves the scale, the point on the curve will move and the tangent line at that new point will be shown. To the code we add `@interact` before the function definition. In the header of the function we add the slider.

The parameter `x0` of the function is extended into `x0 = slider(-2, 2, 0.1, -1.5, label='x-coordinate')`. The first two arguments, `-2` and `2` determine the range of the slider, so `x0` is constrained to take values in the interval with bounds `-2` and `2`. The granularity of the slider is defined as `0.1`, the third parameter of the `slider()`. The fourth parameter sets the initial value `-1.5` of `x0`.

```
f(x) = x^3 - x
df(x) = diff(f(x), x)
@interact
def tangent_at_point(x0 = slider(-2, 2, 0.1, -1.5, label='x-coordinate')):
    """
    Shows the tangent line at the point x0,
    using f and its derivative as global variables.
    """
    y0 = f(x0)
    slope = df(x0)
```

(continues on next page)

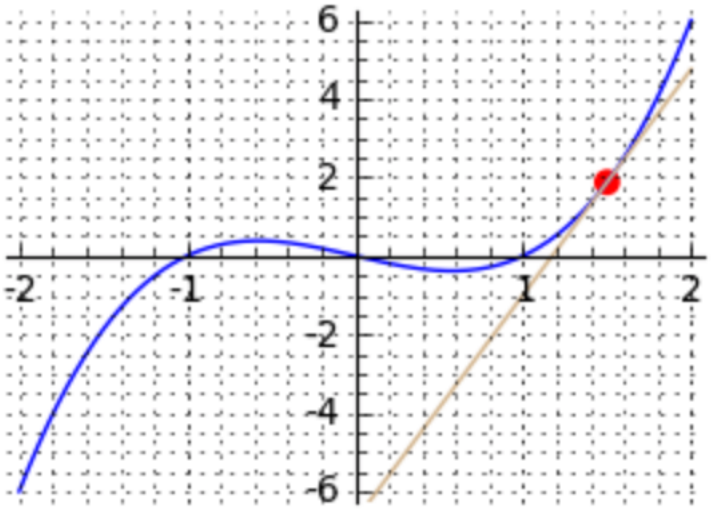

Fig. 4.2: The tangent line of $x^3 - x$ for $x = 1.5$.

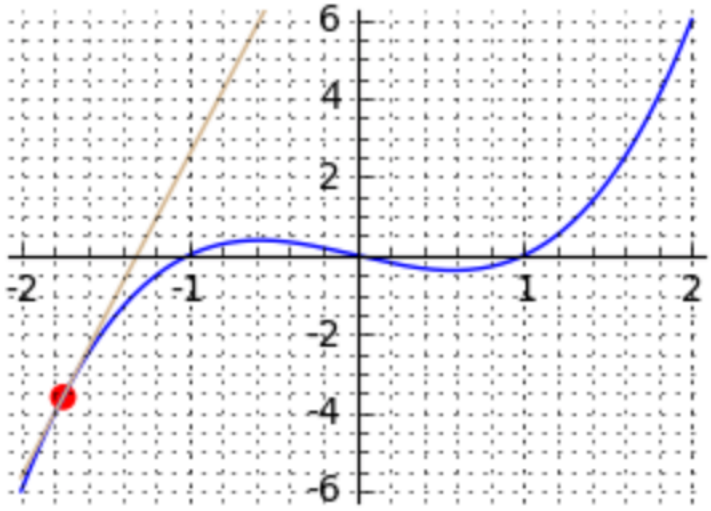

Fig. 4.3: The tangent line of $x^3 - x$ for $x = -1.75$.

(continued from previous page)

```
    # y - y0 = slope*(x - x0) implies
    # y = slope*x - slope*x0 + y0
    b = y0 - slope*x0
    P1 = plot(f, -2, 2, color='blue', ymin=-6, ymax=6, gridlines='minor')
    P2 = plot(slope*x + b, -2, 2, color='tan', ymin=-6, ymax=6)
    P3 = point((x0, y0), color='red', size=50)
    P = P1+P2+P3
    P.show()
```

If we execute the code in the cell, then we can already test the interact as it would run in a web page. The interact is shown in Fig. 4.4.

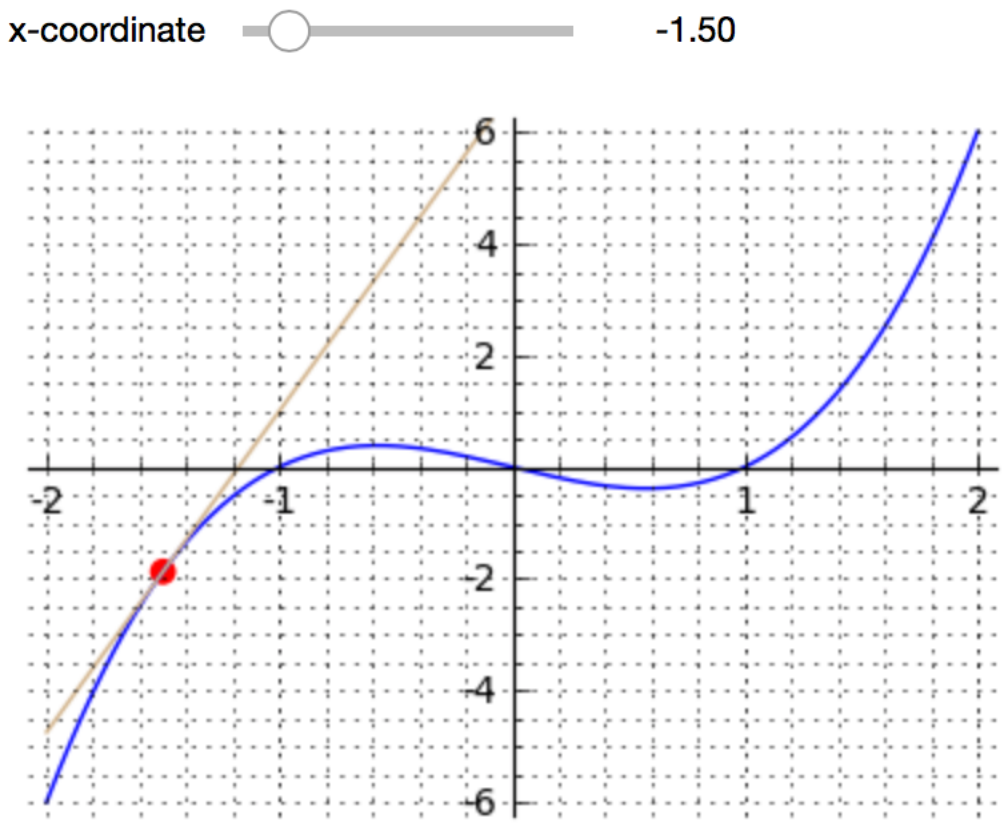

Fig. 4.4: An interact with a slider to set the x-coordinate of a point.

4.1.2 Making the Web Page

On page 294 of *Sage for Undergraduates* by Gregory Bard, we have a template of a web page, based on work of Jason Grout. Once we insert the code in the last cell of the previous section, we are ready to publish the interactive web page.

The computer `people.uic.edu` which can host your web pages is a Linux computer. You can login to this computer with your netid. Your web site is accessible via the URL `https://netid.people.uic.edu` where the `netid` in the URL is replaced by your netid. Below is a description of the steps to deploy the interact on your web site.

1. Login to `people.uic.edu` via `ssh`, available among the internet tools on the lab computers.
2. The files which defined your web pages should be placed in the folder `public_html`. Make sure that this folder is accessibile. To make `public_html` accessible, execute the command `chmod +x public_html` at the command line.
3. You can edit the html file locally on your computer and then transfer the file into the `public_html` folder on `people.uic.edu` with a secure copy which can be found among the internet tools on the lab computers.

Note that you account on `people.uic.edu` needs to be activated. Point your browser to `people.uic.edu` to activate.

4.1.3 Drop Down Menu

With `selector` we add a drop down menu to let the user choose from a range of options. For example, to select a color.

```
fun = x^3 - x
rng = (x, -2, 2)
@interact
def colorplot(colrgb=selector(['red', 'green', 'blue'], label='color: ')):
    """
    Plots the expression defined by fun over the range in rng.
    The parameters fun and rng must be defined before calling colorplot.
    The user can select the color.
    """
    P = plot(fun, rng, color=colrgb)
    P.show()
```

The result of the interact is shown in Fig. 4.5.

4.1.4 Assignments

1. Consider the function $f(x) = \exp(-x^2)\sin(2k\pi x)$ for $x \in [-2, +2]$ and for k ranging from 1 to 10. Make an interactive web page that plots $f(x)$ and with a slider so the user can set k.
2. When plotting in three dimensions we can rotate the axes, for example with the `rotateX` method. Make an interactive plot3d method where the argument for `rotateX` can be manipulated with a slider.
3. Cassini ovals are all points at distance c from the points with coordinates $(-a, 0)$ and $(a, 0)$, and satisfy

$$((x-a)^2 + y^2)((x+a)^2 + y^2) = c^4.$$

   Define an interact that plots the Cassini ovals for x and y in $[-4, 4]$ which lets the user select the parameters (a, c) from the tuples (1,2), (2,1), and (2,3).
4. The coordinates of the Lissajous curve are defined by

$$x(t) = \sin(2t), \quad y(t) = \sin(3t), \quad t \in [0, 2\pi].$$

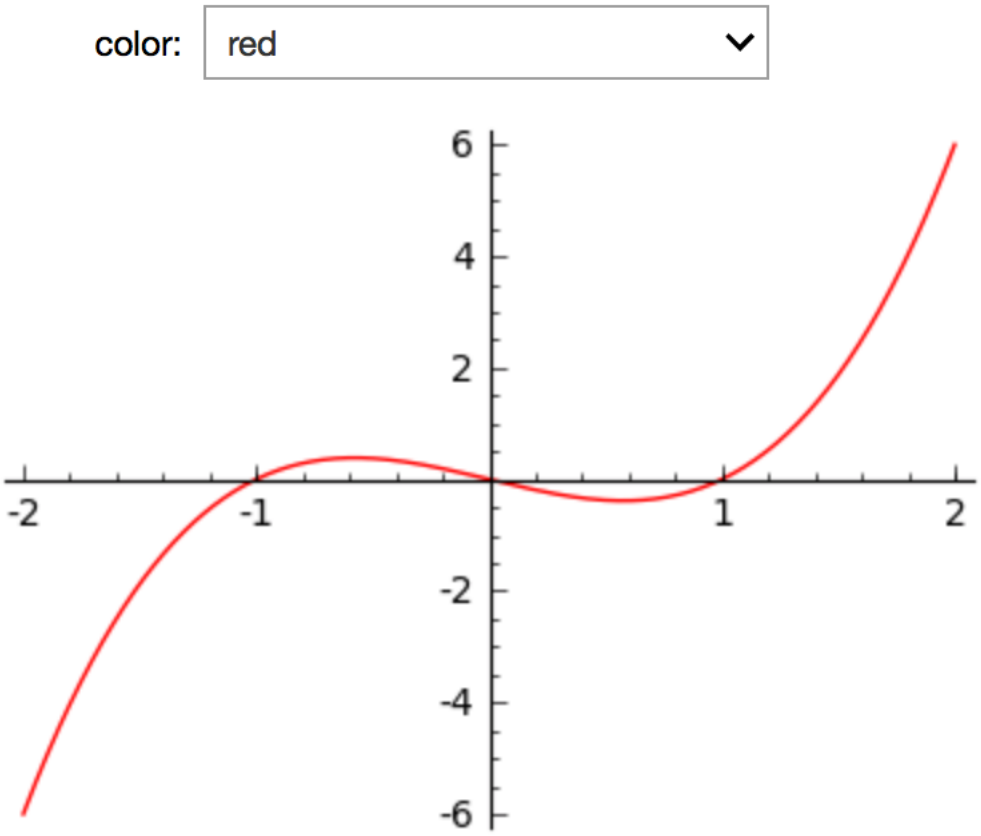

Fig. 4.5: An interact with a selector to let the user select a color.

Write the code to define an interact to plot this curve.

- The interact allows the user to set the end of the range for t with a slider.
- The range always starts at 0. The smallest value for the end of the range is $\pi/20$. The largest value of the end of the range is 2π.
- The end value for the range is incremented by $\pi/20$.
- The initial value for the end of the range is $2\pi - \pi/10$.

4.2 Lecture 35: an Application of Interact

In this lecture we consider an application of interact. We build a model of a four bar mechanism. The mechanism is attributed to Chebyshev and it transmits straight to circular motion. We build the interact in three stages, first the crank, then we solve a polynomial system to determine the coordinates of the connector point. In the third stage we add the coupler point and the fourth bar. Our interact is shown in Fig. 4.6.

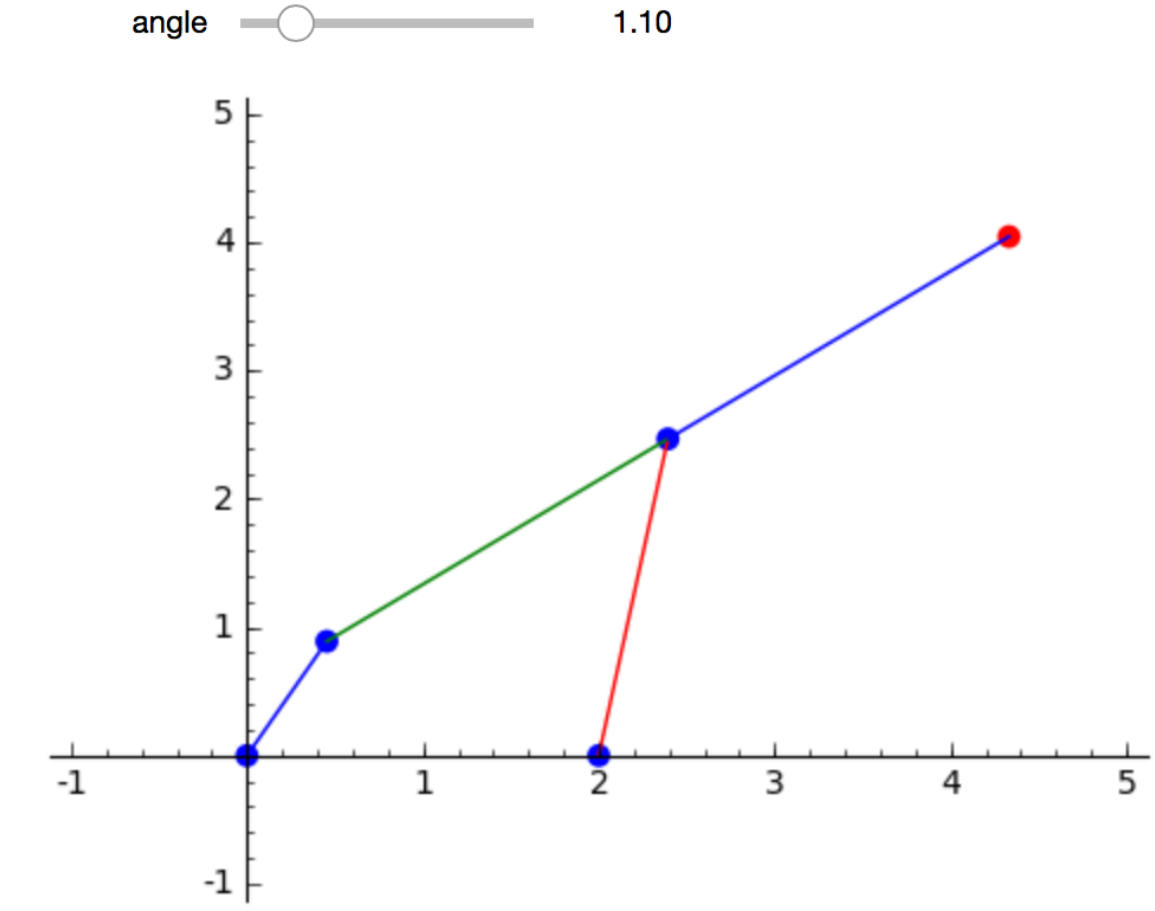

Fig. 4.6: An interact to draw the Chebyshev 4-bar mechanism.

To learn more about the Chebyshev mechanism, see for example *Kinematics, Dynamics, and Design of Machinery* by Kenneth J. Waldron and Gary L. Kinzel, second edition, John Wiley & Sons, 2003.

4.2.1 Turning a Crank

Let us start with the making of an interact for a crank. The parameter in the slider is an angle, which ranges from 0 to 2π, with increments of $\pi/20$, with the initial position at $\pi/10$. Each time the user touches the slider, the line between `(0,0)` and `(cos(angle), sin(angle))` is drawn.

```
@interact
def show_crank(angle = slider(0, 2*pi, pi/20, pi/10, label='angle')):
    """
    Shows the drawing of a crank.
    The user can slide the angle to turn the crank.
    """
    center = (0,0)
    endpnt = (cos(angle), sin(angle))
    pltcnt = point(center, size=50)
    pltend = point(endpnt, size=50)
    crank = line([center, endpnt])
    (pltcnt+crank+pltend).show(xmin=-1, xmax=1, ymin=-1, ymax=+1)
```

Observe the ranges for `x` and `y` in the `show()`. The result of the interact is shown in Fig. 4.7.

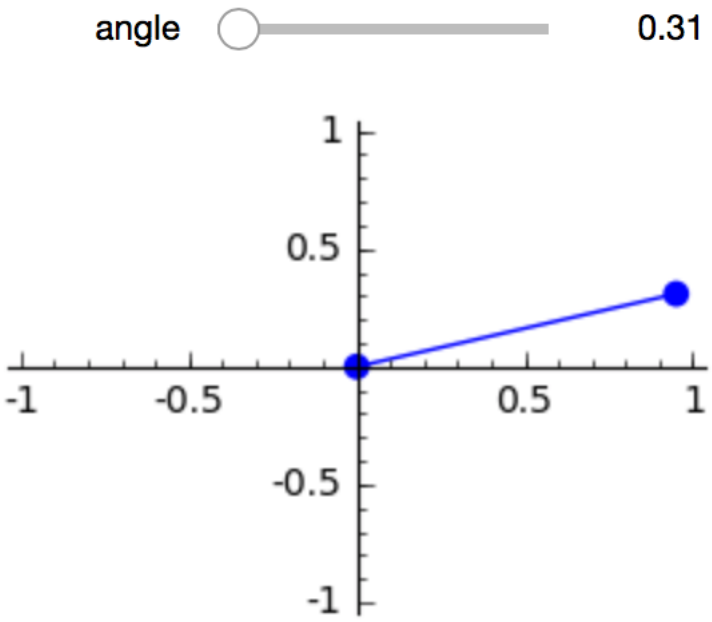

Fig. 4.7: The slider of the interact drives a crank.

4.2.2 Computing Coordinates of the Connector Point

There is a bar connected to the end point of the crank. The end point of that bar is connected to another bar, of length 5/2, that originates at (2,0). To find the coordinates of the connector point, we intersect two circles.

```
PR.<x,y,c,s> = PolynomialRing(QQ, order='lex')
p = (x-c)^2 + (y-s)^2 - (5/2)^2
q = (x-2)^2 + y^2 - (5/2)^2
print('p = ', p)
print('q = ', q)
```

With the resultant we get symbolic values for the coordinates, in function of the end point coordinates (c, s) of the crank.

```
ry = p.resultant(q,x)
rx = p.resultant(q,y)
print(ry)
print(rx)
```

To solve the polynomials for x and y, we must convert back to the Symbolic Ring (SR).

```
x, y, c, s = var('x,y,c,s')
sx = solve(SR(rx),x)
sy = solve(SR(ry),y)
print('sx = ', sx)
print('sy = ', sy)
```

We take the second solution as coordinates for x and the first solution for the coordinates for y. This was done by trial and error.

```
xv = sx[1].rhs()
yv = sy[0].rhs()
print('x = ', xv)
print('y = ', yv)
```

To use the formulas for the coordinates of the connector point, we make functions to substitute the arguments of the functions into the formulas for the coordinates of the connector point. These functions are used in our second interact.

```
fx(cs, sn) = xv.subs(c=cs, s=sn)
fy(cs, sn) = yv.subs(c=cs, s=sn)
@interact
def connector(angle = slider(0, 2*pi, 0.1, pi/2, label='angle')):
    """
    Shows the drawing of a crank, where the user can slide the angle
    to turn the crank.  Also the connector point is drawn.
    """
    center = (0,0)
    c = cos(angle)
    s = sin(angle)
    endpnt = (c, s)
    (x, y) = (fx(c,s), fy(c,s))
    pltcnt = point(center, size=50)
    pltend = point(endpnt, size=50)
    connector = point((x,y), size=50)
```

(continues on next page)

(continued from previous page)

```
    crank = line([center, endpnt])
    bartwobase = point((2,0),size=50)
    bartwo = line([(2,0), (x,y)], color='red')
    barthree = line([endpnt, (x,y)], color='green')
    bars = crank + bartwo + barthree
    pnts = pltcnt + connector + pltend + bartwobase
    plt = bars + pnts
    plt.show(xmin=-1, xmax=3, ymin=-1, ymax=+3, figsize=3)
```

The result of the interact is shown in Fig. 4.8.

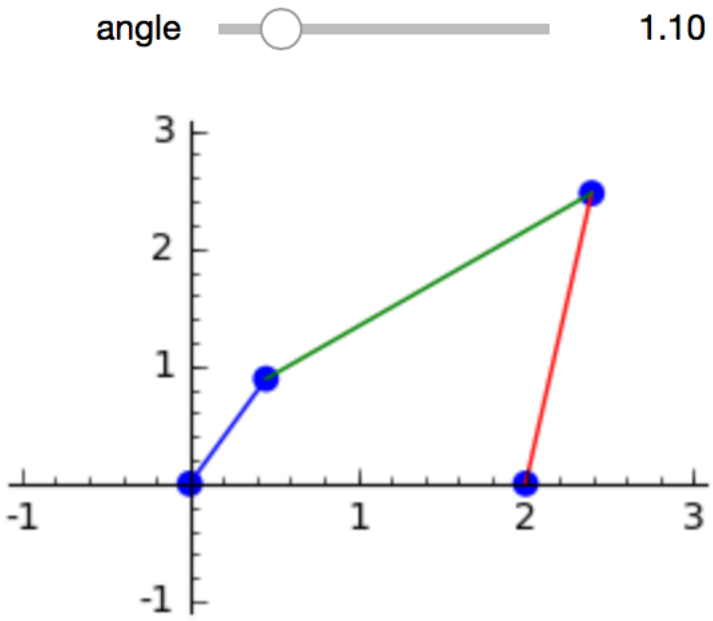

Fig. 4.8: An interact for a 4-bar mechanism, the slider drives the crank.

4.2.3 Adding the Coupler and Fourth Bar

The final addition is the coupler point and the fourth bar. The fourth bar lies in the extension of the end point of the crank and the connector point. As the fourth bar has the same length as the bar between the end point of the crank and the connector point, to obtain the coordinates of the coupler point, we add the differences between connector and end point of the crank to the coordinates of the connector point.

```
@interact
def coupler(angle = slider(0, 2*pi, 0.1, pi/2, label='angle')):
    """
    Shows the drawing of a crank, where the user can slide the angle
    to turn the crank.  Also the connector point is drawn,
    with the coupler point and fourth bar.
    """
    center = (0,0)
    c = cos(angle)
```

(continues on next page)

(continued from previous page)

```
    s = sin(angle)
    endpnt = (c, s)
    (x, y) = (fx(c,s), fy(c,s))
    dx = x - c
    dy = y - s
    xx = x + dx
    yy = y + dy
    pltcnt = point(center, size=50)
    pltend = point(endpnt, size=50)
    connector = point((x,y), size=50)
    coupler = point((xx,yy), size=50, color='red')
    crank = line([center, endpnt])
    bartwobase = point((2,0),size=50)
    bartwo = line([(2,0), (x,y)], color='red')
    barthree = line([endpnt, (x,y)], color='green')
    barfour = line([(x,y), (xx,yy)], color='blue')
    bars = crank + bartwo + barthree + barfour
    pnts = pltcnt + connector + pltend + bartwobase + coupler
    plt = bars + pnts
    plt.show(xmin=-1, xmax=5, ymin=-1, ymax=+5, figsize=5)
```

As we move the crank, we see that the coupler point traces a straight line. Thus the mechanism translates the rotational motion of the crank to the straight motion of the coupler point. The result is shown in Fig. 4.6.

4.2.4 Deploying the Interactive Web Page

When we insert the code into a web page, we must ensure that the entire context is defined. For this application, this means that we declare the variables `c` and `s`, and also define the formulas `xv` and `yv`.

4.2.5 Assignments

1. Add an extra parameter to the interact that shows the crank. The extra parameter is the length L of the crank, so the end point has coordinates (L*cos(angle), L*sin(angle)).
2. As a continuation of the previous exercise, find the coordinates of the connector point, symbolically in function of the variables c, s, and L.
3. Still as a continuation of the previous exercises, extend the interact of the connector with an extra slider for the length of the crank.
4. Finally, extend the interact of the coupler point with a slider for the length of the crank.
5. Extend the interact of the coupler point so that the path of the coupler point is drawn with a list_plot for the angle going from zero till the position determined by the current value of the angle.

4.3 Lecture 36: Symbolic Computation with sympy

SymPy is pure Python package for symbolic computation. The *pure* means that, unlike in SageMath, no modifications to the Python scripting language have been made. SymPy is integrated in SageMath and at times we need to be aware of how to use its functionality properly. SageMath is licensed under the GNU GPL, whereas the license of SymPy is BSD.

4.3.1 sympy outside SageMath

Outside SageMath, we can compute with SymPy as in the Sage cell server, point your browser to the URL <http://live.sympy.org>.

A comparison of SymPy versus SageMath is given at <https://github.com/sympy/sympy/wiki/SymPy-vs.-Sage>.

SymPy is one of the core packages of the SciPy stack <http://www.scipy.org>.

4.3.2 sympy inside SageMath

We explicitly applied sympy to define a generator for the terms in a Taylor series.

If we use sympy in SageMath, then we may have to be explicit about the use of its functions. For example, the `var` in SageMath is different from the `var` in sympy.

```
import sympy
x = sympy.var('x')
type(x)
```

We see that the type of `x` is `sympy.core.symbol.Symbol`.

Observe the difference with the `var` that we are normally using in SageMath:

```
y = var('y')
type(y)
```

and the type is `sage.symbolic.expression.Expression`.

If we use the `sqrt()` on a variable that is a sympy symbol, then we get a symbolic expression of SageMath on return.

```
print(type(sqrt(x)))
print(type(sqrt(y)))
```

In both cases we see the same type. We can make a generator object on the `sqrt(1-x)`. That works.

```
tsqrtx = sympy.series(sqrt(1-x),x,0,n=None)
print(tsqrtx)
print([next(tsqrtx) for k in range(10)])
```

But it will not work on `sqrt(1-y)`.

```
tsqrty = sympy.series(sqrt(1-y),y,0,n=None)
print(tsqrty)
next(tsqrty)
next(tsqrty)
```

Although `tsqrty` is also some type of generator, the second application of the generator triggers the `StopIteration` exception.

So for the generating function for the Catalan numbers, we must use a sympy symbol `x` and not a SageMath variable `y`.

```
g = (1 - sqrt(1-4*x))/(2*x)
cg = sympy.series(g,x,0,n=None)
print([next(cg) for k in range(11)])
print(catalan_number(10))
```

4.3.3 Pattern Matching

Another feature better in SymPy than in SageMath is pattern matching, which can be useful when combined with substitution. To run this code we can also open a Terminal session, select Misc, and select python.

```
reset()
from sympy import *
x = Symbol('x')
y = Wild('y')
d = (10*x**3).match(y*x**3)
d
```

On return is a dictionary that has `y_` as key with `10` as corresponding value, which expresses the match between `y` and `10` in the expressions `10*x**3` and `y*x**3`.

4.3.4 Solving Recurrence Relations

SymPy can solve recurrence relations. Consider the Fibonacci numbers, defined as $f(n) = f(n-1) + f(n-2)$. We import `rsolve` and declare `f` as a function of `n`.

```
reset()
from sympy import Function, rsolve
from sympy.abc import n
f = Function('f')
```

The equation $f(n) = f(n-1) + f(n-2)$ is entered as `f(n+2) - f(n+1) - f(n)` and assigned to the variable `Fib`. This variable is the first argument of `rsolve` and we solve for `f(n)`.

```
Fib = f(n+2) - f(n+1) - f(n)
print('solving', Fib)
s = rsolve(Fib, f(n))
s
```

The solution `s` has the following form: `C0*(1/2 + sqrt(5)/2)**n + C1*(-sqrt(5)/2 + 1/2)**n`. Because we have a two terms relation, for a unique solution, we need to initial conditions, as an additional dictionary argument.

```
s2 = rsolve(Fib, f(n), {f(0): 0, f(1): 1})
s2
```

and the solution `s2` equals `sqrt(5)*(1/2 + sqrt(5)/2)**n/5 - sqrt(5)*(-sqrt(5)/2 + 1/2)**n/5`. As a verification, let us apply the formula to compute the first 10 Fibonacci numbers. But first, we need to convert the SymPy `s2` into a SageMath expression:

```
fs2 = SR(s2)
```

Now we can evaluate and expand the expressions involving `sqrt(5)`, which are then automatically simplified.

```
[fs2(n=k).expand() for k in range(10)]
```

4.3.5 The Nearest Algebraic Number

SymPy depends only on `mpmath` and `mpmath` exports the `pslq` function which allows to compute the nearest algebraic number, given a floating-point approximation. Let us illustrate this with $\sqrt{2}$ which is a root of `x^2 - 2`.

```
from mpmath import pslq
s2 = RR(sqrt(2))
pwrs = [s2**k for k in range(3)]
```

In `pwrs` are the first three powers of `s2` starting at the zeroth power.

```
cff = pslq(pwrs, verbose=True)
cff
```

The output in `cff` is the list `[2, 0, -1]` which are the coefficients of the polynomial $2 - x^2$. This is the polynomial for which `s2` is an approximate roots with 53 bits of precision.

4.3.6 Assignments

1. What happens if you ask SageMath or sympy to compute a Taylor series expansion of `sqrt(x)` at `x = 0`? Explain the result.
2. The cost $T(n)$ of a merge sort on a list of n numbers is governed by the following recurrence relation:

$$T(n) = 2T\left(\frac{n}{2}\right) + n - 1, \quad T(1) = 0.$$

   Use rsolve to find an explicit solution for $T(n)$. Simplify so you can see $T(n)$ is $O(n \log(n))$.

4.4 Lecture 37: Numerical Computation with numpy and scipy

Many problems in scientific computing reduce to numerical linear algebra and numerical algorithms to solve differential equations. The packages numpy and scipy are standalone packages for use with Python and they are part of the core packages in the SciPy stack, see <http://www.scipy.org>. For plotting in Sage, we have Matplotlib, <http://www.matplotlib.org>, which is also part of the SciPy distribution.

Both numpy and scipy are integrated in Sage. The main functionality of numpy is that it extends Python with matrices and multidimensional arrays. The package scipy bundles several standard scientific mathematical libraries. Examples of such libraries are BLAS and LAPACK for numerical linear algebra, QUADPACK for numerical integration, and ODEPACK for the numerical solution of ordinary differential equations.

4.4.1 Numerical Solving of Systems of Linear Equations

We have been using numpy already in our plotting.

```
from sage.misc.citation import get_systems
get_systems("point([0,0], size=50).show(figsize=1, axes=False)")
```

The output is shown in Fig. 4.9.

Out[1]: ['numpy']

Fig. 4.9: The point() method is executed by numpy.

From the output `['numpy']` we see that numpy is credited with the display of the point.

To visualize large data sets, we can use matrix_plot.

```
import numpy as np
A = np.random.normal(0, 1, (20, 20))
matrix_plot(A, cmap='hsv', colorbar=True)
```

The result is shown in Fig. 4.10.

We have already a random coefficient matrix `A`. Let us generate a random right hand side vector `b` and solve a linear system `A*x = b`. The random numbers are uniformly distributed in the interval `[-1,+1]`.

```
b = np.random.uniform(-1, 1, (20, 1))
x = np.linalg.solve(A, b)
v = b - A*x
r = np.linalg.norm(v)
r
```

What we see printed as the value of `r` is `18.1786344846`. The residual `r` of a linear system is the norm of `b - A*x`, but something went wrong, because the residual is too large! What is the problem? Let us look at the type of `A`, `b`, and `x`.

```
print('type(A) :', type(A))
print('type(b) :', type(b))
print('type(x) :', type(x))
```

In all three cases, we see `numpy.ndarray` as the type. For the operator arithmetic to work properly, we must convert to the proper matrix types.

```
mA = np.matrix(A)
print(type(mA))
mb = np.matrix(b)
print(type(mb))
mx = np.matrix(x)
print(type(mx))
```

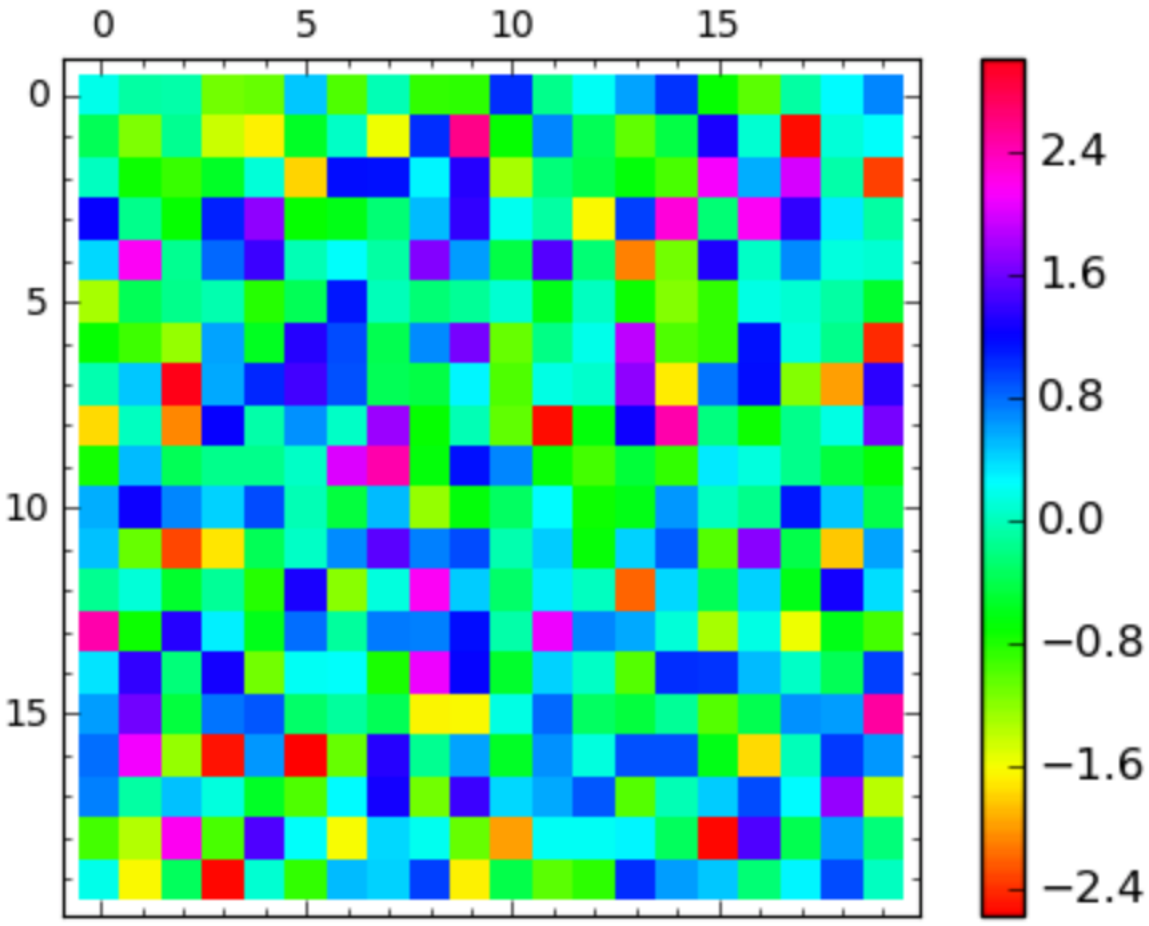

Fig. 4.10: A plot of a matrix of normally distributed numbers with a colorbar.

The variables `mA`, `mb`, and `mx` are instances of the class `numpy.matrixlib.defmatrix.matrix`. Now we can compute the residual correctly.

```
v = mb - mA*mx
r = np.linalg.norm(v)
r
```

As we see for the value of `r` the number `2.37857348102e-15` we see that the solution `x` of the linear system `A*x = b` is accurate within the machine precision for hardware doubles.

The fundamental data type in a MATrix LABoratory is a matrix and numpy gives matrices to Python. Vectorization is a technique to formulate linear algebra operations with vector and matrix arithmetic. The arithmetic is performed on dedicated data structures by optimized and fine tuned libraries.

4.4.2 Numerical Integration

Many expressions do not have symbolic antiderivatives.

```
f = exp(sin(x))
a = integral(f, x, 0, 1)
a
```

Sage then returns `integrate(e^sin(x), x, 0, 1)`. It would have been better to set the `hold` flag to True, as in `integrate(e^sin(x), 0, 1, hold=True)`. Then with `a.n()` we can get a numerical approximation. Sometimes we may directly go for a numerical approximation and apply quadrature rules.

```
from scipy.integrate import quad
d = quad(f, 0, 1, full_output=1)
d
```

Not only is the numerical approximation returned, but also an estimate for the error and the number of function evaluations. The first two numbers in the output are `(1.6318696084180513, 1.8117392124517587e-14,` which respectively give the approximation of the integral and the estimated error on the approximation.

4.4.3 Rational Approximations

Often we use a combination of sympy and scipy. For example, the computation of a Padé approximation starts from a Taylor series. We can compute Taylor series with sympy and Padé approximations with scipy.

```
from sympy import var, sin, series
x = var('x')
s = series(sin(x), x, 0, n=None)
terms = [next(s) for _ in range(6)]
terms
```

To extract the coefficients of the terms, we do

```
c = [sterms.coeff(x, k) for k in range(6)]
c
```

The coefficients of the series are `[0, 1, 0, -1/6, 0, 1/120]`. We import the `pade` command from scipy.misc. The first argument of `pade` is the list with the coefficients of the Taylor series. The second argument is the degree of the denominator of the rational approximation. So if we want a denominator of degree 2, we proceed as follows.

```
from scipy.interpolate import pade
p = pade(c, 2)
p
```

On return we get numpy polynomials, which have a nice string representation.

```
print(type(p[0]))
print('numerator :\n', p[0])
print('denominator :\n', p[1])
```

Starting from a Taylor series of order So the rational approximation is `(-0.1167*x^3 + x)/(0.05*x^2 + 1).`

The numpy polynomials are callable, so plotting is no problem and we compare with the plot of the actual sin(x) function.

```
sp = plot(sin(x), (x, -pi, pi), color='red')
pp = plot(p[0](x)/p[1](x), (x, -pi, pi))
(sp+pp).show()
```

The plot is displayed in Fig. 4.11.

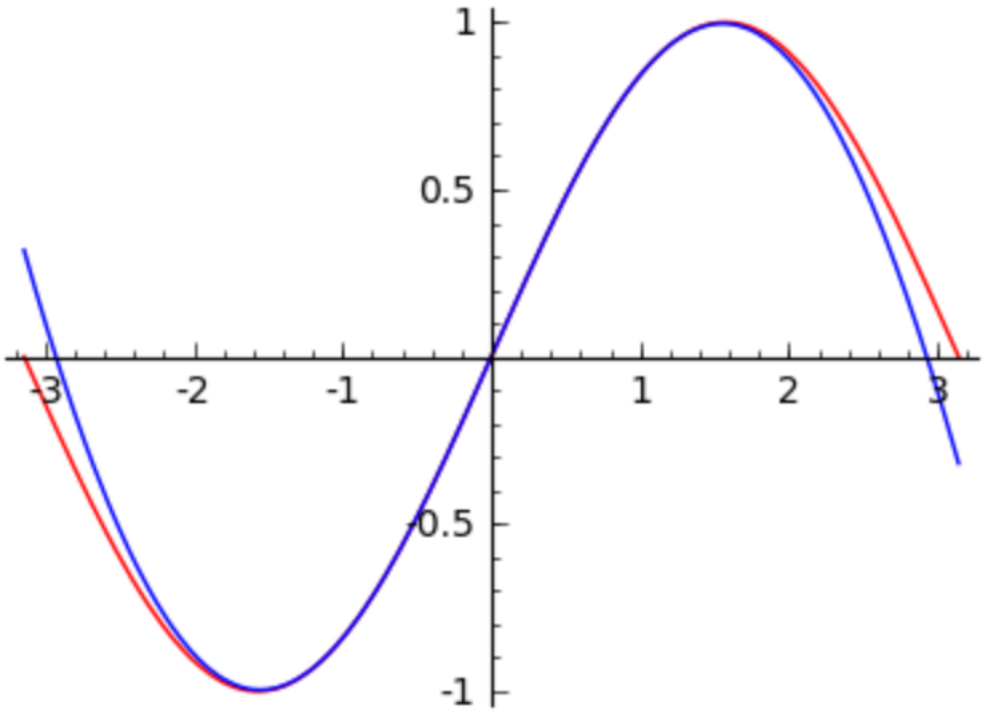

Fig. 4.11: A [3,2]-Padé approximation for the $\sin(x)$ function.

The differences between the rational approximation and the sine function become noticeable only when x is less than -pi/2 or larger than pi/2.

We can use the function to bring the polynomials from scipy into SageMath objects.

```
fp0 = p[0](x)
print(fp0)
print(type(fp0))
sfp0 = SR(fp0)
print(sfp0, 'has type', type(sfp0))
```

The type of the expression is in this way converted by SR() from `sympy.core.mul.Mul` to a symbolic Sage expression.

4.4.4 Numerical Solving of Ordinary Differential Equations

We end with the numerical solving of an ordinary differential equation, taking the pendulum problem:

$$\frac{d^2\theta}{dt^2} = -\frac{d\theta}{dt} - 9.8\sin(\theta)$$

We need to define a system of first-order differential equations, introducing an extra variable velocity which is the derivative of theta. Denote $v(t) = \frac{d\theta}{dt}$, then the second order ordinary differential equation becomes a system of two first order ordinary differential equations.

$$\begin{cases} \frac{d\theta}{dt} &= v \\ \frac{dv}{dt} &= -\frac{d\theta}{dt} - 9.8\sin(\theta). \end{cases}$$

We could define the right hand side of the system as below.

```
t, theta, velocity = var('t, theta, velocity')
oderhs = [velocity, -velocity-9.8*sin(theta)]
oderhs
```

The right hand side of the system has to be defined as a function.

```
def pend(y, t):
    return [y[1], -y[1]-9.8*sin(y[0])]
```

The symbolic evaluation of this function returns the same as the `oderhs` from above. Then we import `odeint` of the `scipy.integrate` package.

```
from scipy.integrate import odeint
from scipy import linspace
npts = 1000 # number of points
tspan = linspace(0, 10, npts)
sol = odeint(pend, [pi/10, 0], tspan)
points = [(tspan[k], sol[k][0]) for k in range(npts)]
list_plot(points, size=1)
```

The outcome of list_plot() is shown in Fig. 4.12.

A phase portrait shows the displacements and the velocities. With a list comprehension we define the coordinates for the list plot.

```
coords = [(sol[k][0], sol[k][1]) for k in range(npts)]
list_plot(coords, size=1, figsize=4)
```

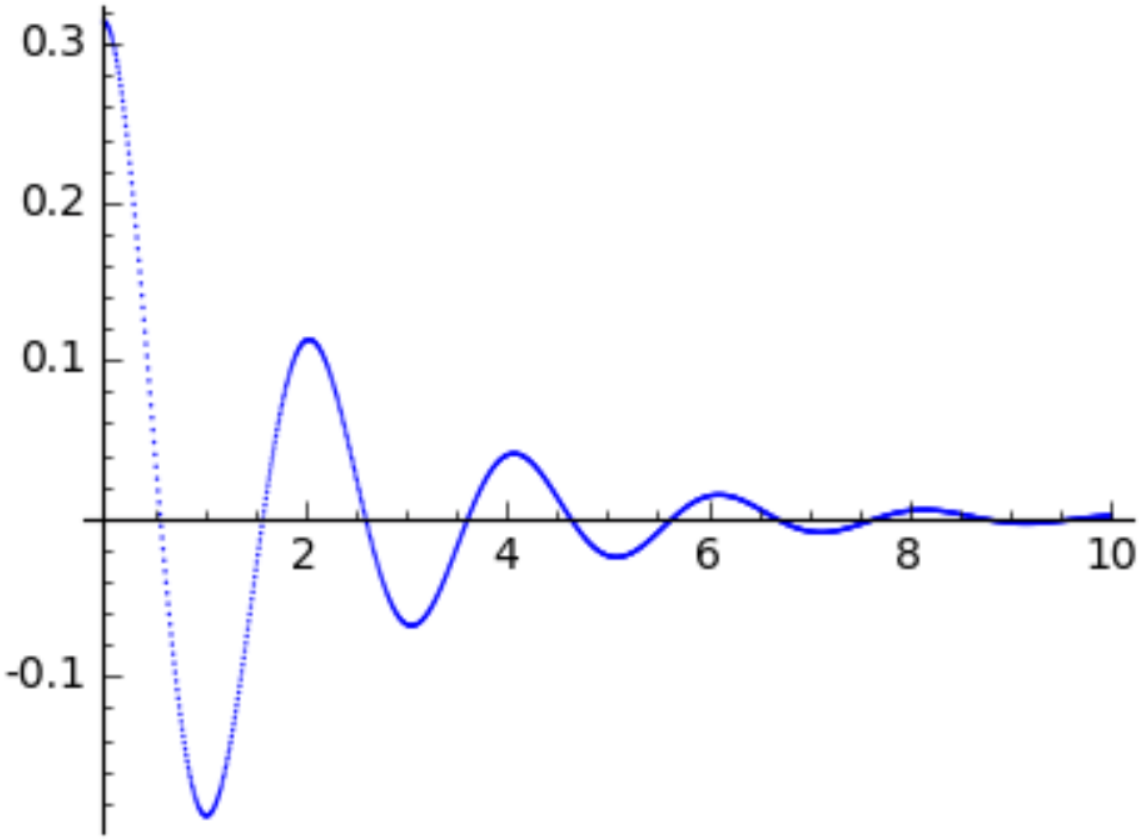

Fig. 4.12: The evolution of the displacement angle of the pendulum over time.

The phase portrait is shown in Fig. 4.13. The spiral shown in this figure should be interpreted as an inward moving spiral, as the pendulum comes to a halt. We see that the rightmost point corresponds to the point with coordinates $(\pi/10, 0)$ which matches the initial displacement angle and the initial velocity. As the trajectory spirals first down, the velocity vector is negative and we move towards the origin. At the leftmost end, we already notice the effect of the damping as the leftmost point is closer to the origin than the rightmost point.

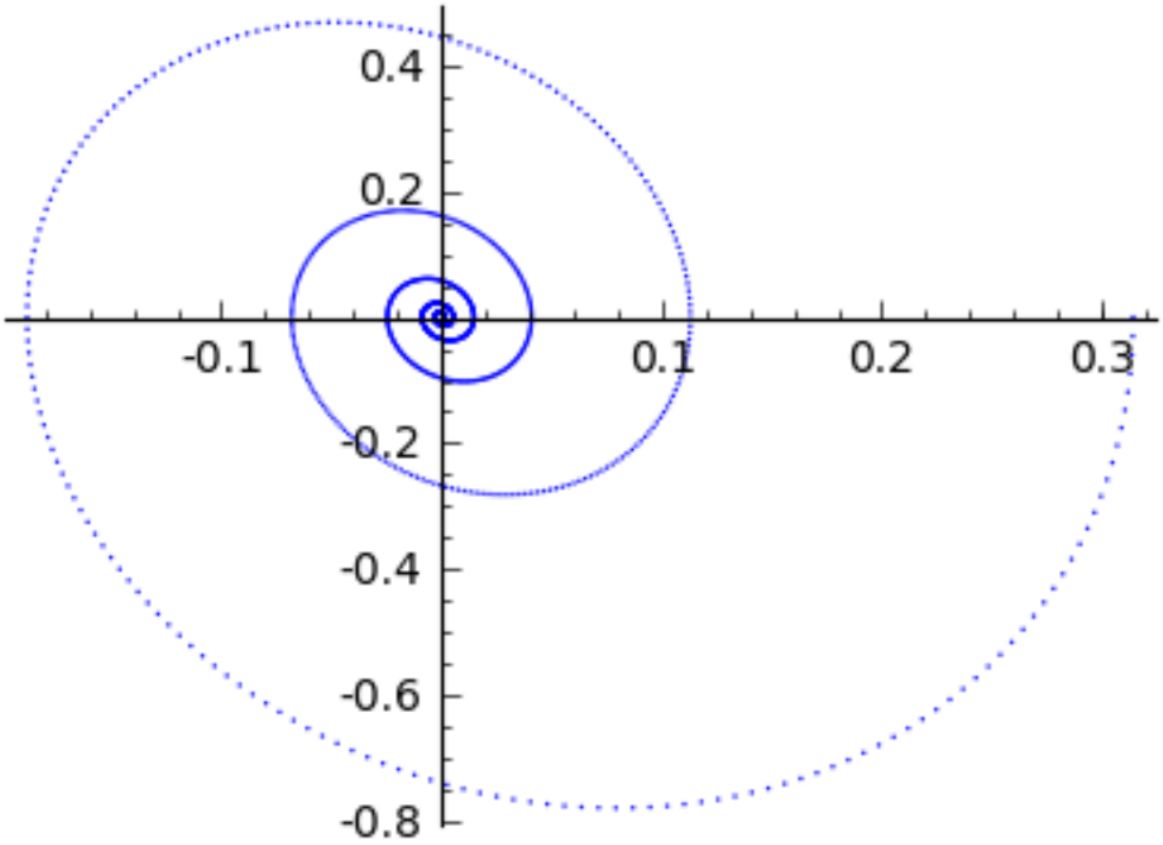

Fig. 4.13: The phase portrait of a damped pendulum.

4.4.5 Assignments

1. Define upper triangular matrices `A` that have ones on the diagonal and above the diagonal, for dimensions 2, 4, 8, 16, etc… The corresponding right hand side vector `b` is a vector of ones. Solve the linear systems `A*x = b` and report the residual, doubling the dimension in each case. How high can the dimension be before the residual becomes larger than `1.0e-8`?
2. Let the n-dimensional matrix A, where the (i, j)-th element is defined as $i^2 + j^2$, for i and j both ranging from 1 to n. Do the following with numpy:
   1. Define A as a numpy matrix, for $n = 17$.

      Define $\mathbf{b}$ as a 17-dimensional vector of ones.
   2. Solve the linear system $Ax = \mathbf{b}$ with numpy. Verify the residual.

      What is the sum of the coordinates in the solution $\mathbf{x}$?

3. Use numpy to do the following.

   Define a tridiagonal 5-by-5 numpy matrix `A`: 3 on the diagonal and 1 just below and 1 just above the diagonal. Do *NOT* type in the 25 elements of `A`. Printing `A` shows:

```
[[3 1 0 0 0]
 [1 3 1 0 0]
 [0 1 3 1 0]
 [0 0 1 3 1]
 [0 0 0 1 3]]
```

   Take as right hand side b a vector of all ones.

   Solve the system `A*x = b` and write the norm of the residual `b - A*x`.

4. Consider the function `f(x) = exp(-x^2)*sin(2*k*pi*x)` for increasing values of `k`, where `k` is 2, 4, 8, 16, 32, 64. Apply `quad` of scipy to approximate the integral of `f(x)` over the interval `[-2, +2]`. Report the number of function evaluations as `k` increases.

5. Compute a rational Padé approximation for `cos(x)`, based on a Taylor series of order 6. Make a plot to compare the accuracy between the rational approximation and the `cos(x)`.

6. Compute a rational Padé approximation for $\tan(x)$, following the steps below.

   (a) Generate a Taylor series of $\tan(x)$ at $x = 0$ order 6. Compute its list of coefficients.

   (b) Use the coefficients to compute a Padé approximation where the denominator has degree 2. Write the rational expression you obtain.

   (c) What is the difference between the value of $\tan(\pi/4)$ and the approximation at $\pi/4$?

7. Consider the following system of differential equations:

$$\begin{cases} \frac{d}{dt}r(t) = 1.1\,r(t) - 0.5\,r(t)f(t) \\ \frac{d}{dt}f(t) = -0.75\,f(t) + 0.25\,r(t)f(t) \end{cases} \qquad r(0) = 3, f(0) = 2.$$

   This system can be used to model the evolution in time t of two species, with $f(t)$ the size of the predator population (f = fox) and $r(t)$ the size of the prey population (r = rabbit).

   1. Define the right hand side vector in a function for use in odeint to solve the system, with the time span for t going from 0 to 8.
   2. With these two functions, make a plot of the solution trajectories, for t going from 0 to 8.
   3. Use the numerical solutions to a phase portrait.

4.5 Lecture 38: Introduction to Julia

Julia is a programming language which aims to combine the high productivity of scripting languages with the high performance of compiled languages. The paper [BEKS17] describes the language.

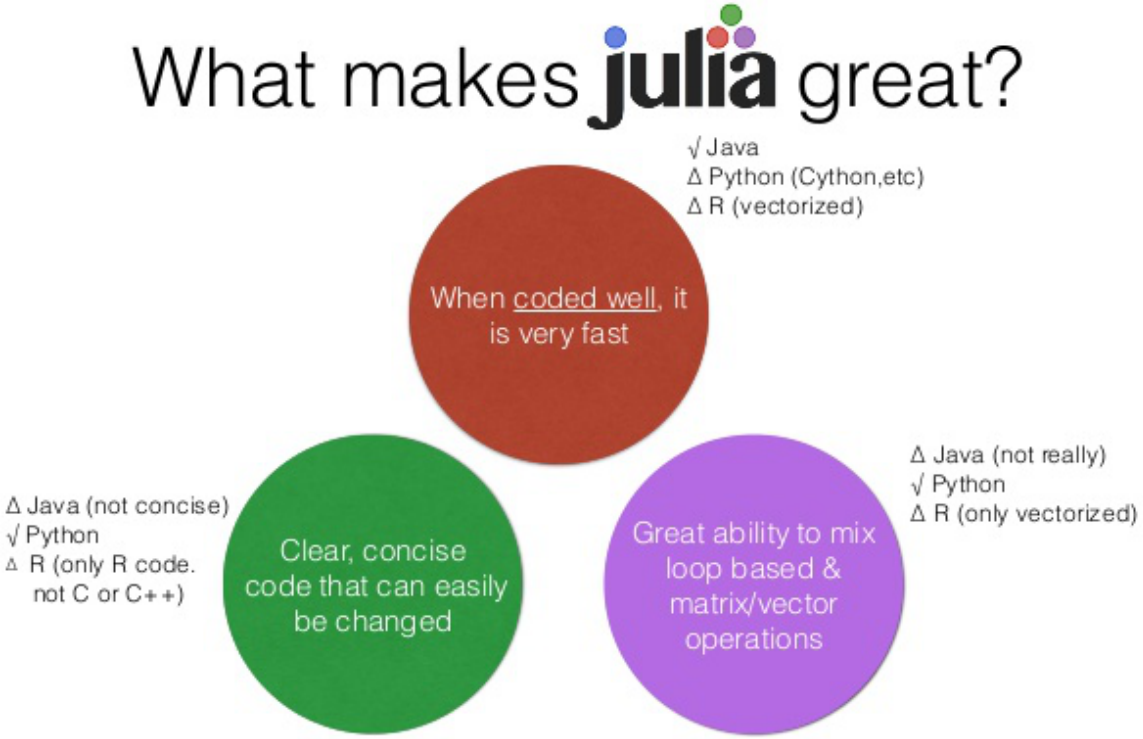

Fig. 4.14: The logo of Julia, from the Software Engineering Daily web site.

4.5.1 Getting Started

In the cloud, at CoCalc: https://cocalc.com you can open a Jupyter notebook with a Julia kernel.

To install Julia on your own computer, follow the instructions at <https://julialang.org>. To execute Julia programs in a Jupyter notebook, do `using IJulia` followed by `notebook()` in the REPL of Julia, that is: when running in a Terminal or PowerShell window. With `using` one can install `Plots`, `SymPy` and `Symbolics`. The `SymPy.jl` is a wrapper to the Python package `SymPy`, whereas `Symbolics` [GMCGSER22] is a computer algebra system written in Julia.

To demonstrate symbolic computation with Julia, consider the problem of computing the slope of the tangent line at a point on a circle.

4.5.2 Implicit Differentiation with SymPy.jl

In the equation of the circle $x^2 + y^2 - 1 = 0$, the variable x is the independent variable and the variable y depends on x. We will obtain the slope via implicit differentiation, as $\frac{d}{dx}y(x)$.

```
using SymPy
```

If `SymPy` is not yet installed, then you will be prompted to install.

The declarations of the variables result in `dy` to be assigned the unknown slope $\frac{d}{dx}y(x)$.

```
x = Sym("x")
y = SymFunction("y")
dy = diff(y(x), x)
```

Observe that `y` is declared as a symbolic function, in *any* symbol. If we want to use `y` when we define the equation, then we *must* use `y(x)` to indicate that the function `y` depends on `x`.

```
c = x^2 + y(x)^2 - 1
dc = diff(c, x)
dydx = solve(dc, dy)
```

The `solve` returns an `Array` which starts in Julia with index one. The slope is then obtained as

```
symbolicslope = dydx[1]
```

and equals `-x/y(x)`.

Let us now take a random point on the circle, generated by a random angle, using the polar representation.

```
angle = 2*pi*rand()
(a, b) = (cos(angle), sin(angle))
sd = Dict(x => a, y(x) => b)
```

The coordinates of the point `(a, b)` are stored in a dictionary, which is a convenient data structure for substitution. Observe that, as Julia is geared towards numerical computing, the `pi` is a 64-bit floating-point approximation for π. The numerical value for the slope is then obtained via substitution as follows:

```
slope = symbolicslope.subs(sd)
```

The numerical value in `slope` serves as a reference to check the alternative computation of the slope via the Taylor series.

The `series` of `SymPy` works for one variable only, but it works well on purely symbolic expressions, so we can compute the expression for the tangent line by running `series` twice. Because we continue the session, we declare new symbols `X` and `Y` for the variables in the circle equation and `A` and `B` for the coordinates of the point.

```
X, Y, A, B = Sym("X, Y, A, B")
nc = X^2 + Y^2 - 1
cx = SymPy.series(nc, X, A, 2)
```

In `cx` is the Taylor series expansion of the expression stored in `nc` about `X = A`, of order `2`. To convert the `cx` into a symbolic expression on which we can apply `series` again, we select the coefficient to make the expression `px` (a polynomial as truncated series).

```
px = cx.coeff(X, 0) + cx.coeff(X - A, 1)*(X - A)
cy = series(px, Y, B, 2)
cY = cy.subs(Dict(A => a, B => b)
```

Then the slope is obtained as

```
- cY.coeff(X, 1)/cY.coeff(Y, 1)
```

which agrees with the value earlier assigned to `slope`.

4.5.3 Implicit Differentiation with Symbolics.jl

We will now compute the value of the slope a third time.

```
using Symbolics.jl
@variables x, y(x)
```

Observe that with `Symbolics` (opposite to what we did with `SymPy`), we declare `y(x)` so `y` is explicitly defined as depending on `y`. This implies that typing `y` suffices and, unlike with `SymPy`, we do not need to type `y(x)` when we define the equation of the circle.

In order not to confuse with the `Differential` of `SymPy`, which we used earlier in this notebook, we declare that we use the `Differential` of `Symbolics`.

```
D = Symbolics.Differential(x)
dy = D(y)
```

As before, `dy` is $\frac{d}{dx}y(x)$.

```
c = x^2 + y^2 - 1
dc = D(c)
```

The differential operator `D` does not expand automatically, we have to ask for it.

```
eq = expand_derivatives(dc)
SBslope = Symbolics.solve_for(eq,dx)
```

In `SBslope` is the expression `-x/y(x)` so the symbolic part of this problem is thus indeed solved. What is left is the substitution, again using a dictionary, mapping the coordinates `(a, b)` of the random point to the symbols `x` and `y`.

```
nsd = Dict(x => a, y => b)
substitute(SBslope, nsd)
```

and we see again the value that was earlier assigned to `slope`.

References

4.6 Lecture 39: Parallel Computing in Julia

All computers are parallel. We distinguish between three types of parallel computing.

1. Distributed memory parallel computing.
2. Shared memory parallel computing.
3. Acceleration with Graphics Processing Units.

Parallel programs are evaluated by speedup and performance.

1. Speedup is the serial time over parallel time.
2. The number of floating-point operations per second (flops) measures the performance.

Julia supports all the three types listed above, but in this lecture we focus on multithreading.

4.6.1 Parallel Symbolic Computing

In symbolic computing we compute expressions. We view the expressions as functions to evaluate at many points. The evaluation at different points happens independently, in parallel.

Two of the most commonly used expressions are polynomials and power series. A typical application is interpolation.

1. A polynomial of degree d in one variable has $d + 1$ coefficients and is determined uniquely by $d + 1$ function values at distinct points.
2. Power series are computed via the Fast Fourier Transform.

4.6.2 Parallel Computing with Julia

We apply multithreading in a Jupyter notebook,

in a kernel installed with the environment variable set to 16 threads.

```
julia> using IJulia
julia> installkernel("Julia (16 threads)", env = Dict("JULIA_NUM_THREADS"=>"16"))
```

If the kernel was installed correctly, then in a code cell of a Jupyter notebook running this kernel, we can verify with

```
using Base.Threads
nthreads()
```

that indeed we have `16` threads available.

A quick way to do parallel programming is to call software libraries which support multithreading. We consider the matrix-matrix multiplication, as executed by `mul!()` of BLAS, where BLAS stands for the Basic Linear Algebra Subroutines.

Two issues we must consider.

1. Choose the size of the matrices large enough.

2. The time should not include the compilation time.

```
using LinearAlgebra
n = 8000
```

The choice of `8000` as a sufficiently large problem can be justified by considering the `peakflops`.

For the matrix-matrix multiplication, we generate 3 random matrices

```
A = rand(n, n); B = rand(n,n); C = rand(n,n);
```

suppressing the output with the semicolon and set the number of threads to 2 with

```
BLAS.set_num_threads(2)
```

Then we run with a timer

```
@time mul!(C, A, B)
```

which reports also the percentage of time spent on compilation. To avoid this overhead, we run again and then double the number of threads and then run again:

```
@time mul!(C, A, B)
BLAS.set_num_threads(4)
@time mul!(C, A, B)
```

With four threads, the time dropped from 9.96 seconds to 6.46 seconds, done on an Intel i9-9880H 2.30GHz CPU in a laptop running Windows, with Julia 1.7.2.

The command `peakflops` computes the peak flop rate using double precision `gemm!`. For the computer used in this experiment, `peakflops(8000)` returned `1.9e11` (or 190 billion flops), which is more than what the `peakflops(4000)` or `peakflops(16000)` reported. So the dimension `n = 8000` is justified.

4.6.3 Parallel Numerical Integration

We can estimate π, via the area of the unit disk:

$$\int_0^1 \sqrt{1-x^2}dx = \frac{\pi}{4}.$$

A Monte Carlo method proceeds as follows:

1. Generate random uniformly distributed points with coordinates $(x, y) \in [0, +1] \times [0, +1]$.
2. We count a success when $x^2 + y^2 \leq 1$.

By the law of large numbers, the average of the observed successes converges to the expected value or mean, as the number of experiments increases.

The random number we apply is very simple:

```
myrand(x::Int64) = (1103515245x + 12345) % 2^31
```

and illustrates the easy at which Julia functions are defined. The function executed by each thread in the experiment is listed below:

```
"""
function estimatepi(n)

Runs a simple Monte Carlo method
to estimate pi with n samples.
"""
function estimatepi(n)
    r = threadid()
    count = 0
    for i=1:n
        r = myrand(r)
        x = r/2^31
        r = myrand(r)
        y = r/2^31
        count += (x^2 + y^2) <= 1
    end
    return 4*count/n
end
```

The `r = thread(id)` initializes the seed for the random number generator with the identification number of each thread. It is very important that every thread generates a different sequence of random numbers. Each thread writes in a different location in an array, defined next:

```
nt = ntreads()
estimates = zeros(nt)
```

Then the parallel for loop below defines the execution:

```
import Statistics
timestart = time()
@threads for i=1:nt
    estimates[i] = estimatepi(10_000_000_000/nt)
end
estpi = Statistics.mean(estimates)
elapsed16 = time() - timestart
```

The total number of 10 billion random points is divided by the number of threads. Running this same loop for a smaller number of threads allows to compute the speedup. The speedup of 16 over 8 threads was observed to be `1.5`.

4.7 Lecture 40: Computational Group Theory with GAP

GAP stands for Groups, Algorithms and Programming. We can run GAP explicitly in Sage via `gap` or open a terminal session with GAP.

There are many groups one can explore with GAP. We start with the permutation groups. As an application, we can apply GAP commands to analyze Rubik's cube.

4.7.1 Permutation Groups

We can define permutation groups via transpositions. For example, to swap 1 with 2, we can define a transposition as follows.

```
a = gap("(1, 2)")
print(a, 'has type', type(a))
```

We can convert a `GapElement` into a Sage element, but we get just a tuple back, which is not what we may want.

```
print(a.sage())
print(type(a.sage()))
```

We see that the `sage.interfaces.gap.GapElement` turns into the type `tuple` for which no group operations are defined.

We read `(1, 2)` as *1 is mapped to 2* and *2 is mapped to 1*. A transposition is its own inverse.

```
gap('Inverse(%s)' % a)
```

The way to assign GAP variables in a Sage worksheet is via the `set` method. Then we can retrieve the values of the GAP variables with `get`.

```
gap.set('a', '(1,2)')
gap.get('a')
```

Let us check that `a` is its own inverse:

```
gap.get('a*a')
```

and indeed, we see `()`, which is the identity permutation.

Let us make another transposition, `b`, and multiply `b` with `a`.

```
gap.set('b', '(1, 3)')
gap.set('c', 'a*b')
gap.get('c')
```

What is printed for `c` is `(1,2,3)`. The cycle notation for `c` means that *1 is mapped to 2*, *2 is mapped to 3*, and *3 is mapped to 1*. To verify this notation, consider *(x, y, z)* and the application of `a`: *(y, x, z)*, followed by `b`: *(z, x, y)*, so we see that *x* moved from first to second position, *y* from second to third, and *z* from third to first. Note that if we first apply (1, 3) to *(x, y, z)*, we get *(z, y , x)*, followed by (1, 2), then we end up with *(y, z, x)*, so `a*b` is not equal to `b*a`.

We can have GAP generate the group generated by `a` and `b`.

```
gap.set('g', 'Group(a,b)')
gap('g')
```

Then we see that g corresponds to Group([(1,2), (1,3)]). We can list all elements in a group with an enumerator.

```
print gap('Size(g)')
gap.set('e', 'Enumerator(g)')
e
```

We see there are six elements in the group. Now we can use the enumerator to list all elements of the group.

```
gap.set('G', 'List([1..Size(g)], i -> e[i])')
gap.get('G')
```

The elements in the group are listed as [(), (1,2,3), (1,3,2), (2,3), (1,2), (1,3)].

The multiplication table stores the results of all multiplications for each pair of elements in the group. We see exactly one 1 on every row and every column, so each element in the group has an inverse.

```
T = gap('MultiplicationTable(G)')
for t in T: print(t)
```

And the multiplication table is

```
[ 1, 2, 3, 4, 5, 6 ]
[ 2, 3, 1, 6, 4, 5 ]
[ 3, 1, 2, 5, 6, 4 ]
[ 4, 5, 6, 1, 2, 3 ]
[ 5, 6, 4, 3, 1, 2 ]
[ 6, 4, 5, 2, 3, 1 ]
```

4.7.2 Decomposition of Permutations

Any permutation can be written in terms of the generators of the group. Let us verify this statement. We consider a random element of the full symmetric permutation group of 10 elements. We call this group S10.

```
gap.set('S10', 'SymmetricGroup(10)')
gap.get('S10')
```

and we see SymmetricGroup([1 .. 10]), a group of size 10!. We generate a random element of this group:

```
gap.set('r', 'Random(S10)')
gap.get('r')
```

and our random element r is (1, 3, 7)(2,10, 6, 5, 4)(8, 9). Can we write r in terms of the generators of S10? Let us see what the generators of S10 are.

```
gap.get('GeneratorsOfGroup(S10)')
```

and we see there are only two generators, as the list shown is [(1, 2, 3, 4, 5, 6, 7, 8, 9,10), (1,2)]. We assign the list to S10gens and then select from the list the two generators (lists in GAP start at position 1), assigning the generators to x and y.

```
gap.set('S10gens', 'GeneratorsOfGroup(S10)')
gap.set('x', 'S10gens[1]')
gap.set('y', 'S10gens[2]')
```

Then we define the group generated by x and y as follows;

```
gap.set('G', 'FreeGroup("x", "y")')
gap.get('G')
```

nand we see Group([x, y]). We declared G as a FreeGroup because there are no relations between the generators x and y. Our problem is now to write r in terms of x and y. To accomplish this, we define a group homomorphism between G and S10.

```
gap.set('hom', \
        'GroupHomomorphismByImages( G, S10, GeneratorsOfGroup(G), GeneratorsOfGroup(S10))
↪' )
print gap.get('hom')
```

The print statement confirms that the operation succeeded. Then we can ask GAP to compute the representation of r in terms of the generators of G.

```
gap.set('p', 'PreImagesRepresentative(hom, r)')
gap.get('p')
```

and we obtain y^-1*x^-1*y*x*y*x^-2*y^-1*x^6*(x*y)^4*x^5*(y*x^-1)^2*y*x^2*(y*x^-1)^2*x^-3 as the way r is defined in terms of x and y. But how can we verify that p is really equal to (1, 3, 7)(2,10, 6, 5, 4)(8, 9)? We encountered this problem in symbolic computation before, how do we compute that two expressions represent the same object?

Consider the following instructions:

```
xrep = gap.get('x')
print(xrep, 'has type', type(xrep))
```

and what is printed is (1, 2, 3, 4, 5, 6, 7, 8, 9,10) <type 'str'> confirming that xrep is a string. To solve our verification problem, we will replace in p all instances of x and y by their string representations, using the replace() method on Python strings.

```
yrep = gap.get('y')
prep = gap.get('p')
rpx = prep.replace('x',xrep)
rpxy = rpx.replace('y',yrep)
```

Now we have one long complicated string in rpxy. We can ask GAP to evaluate the expression defined in rpxy.

```
gap(rpxy)
```

and what is printed is (1, 3, 7)(2,10, 6, 5, 4)(8, 9), the random element r we started with. This confirms the writing of r as p, an expression in the generators x and y.

4.7.3 Rubik's Cube

A three dimensional combination cube was invented in 1974 by Ernoe Rubik. As each face rotates independently, the colors of the side mix up. Rubik's cube is shown in Fig. 4.15.

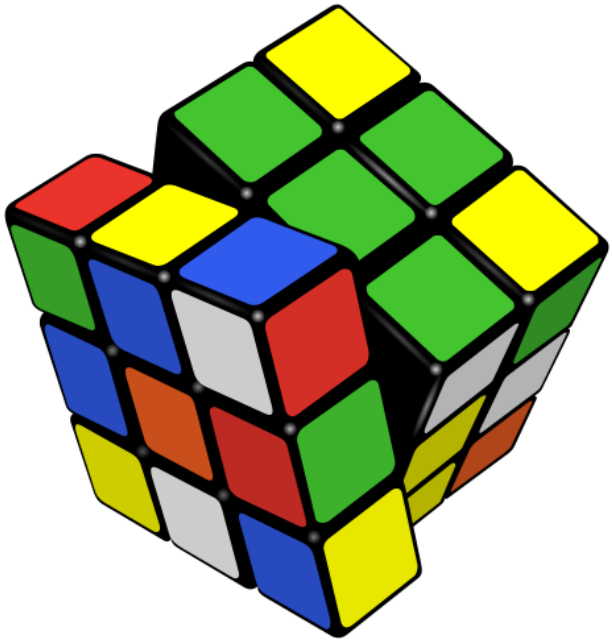

Fig. 4.15: Rubik's cube.

One of the examples of applications of GAP is described at <http://www.gap-system.org/Doc/Examples/rubik.html> by Martin Schoenert. To define the group actions on the cube, we need to number the tiles as shown in Fig. 4.16.

The state of the cube is described by a permutation on 48 numbers.

Turning the front face is described by the permutation

```
(17,19,24,22)(18,21,23,20)( 6,25,43,16)( 7,28,42,13)( 8,30,41,11)
```

which consist of 5 cycles, with meaning

```
(17,19,24,22) :  17 -> 19 -> 24 -> 22 -> 17,
(18,21,23,20) :  18 -> 21 -> 23 -> 23 -> 20,
( 6,25,43,16) :   6 -> 25 -> 43 -> 16 ->  6,
( 7,28,42,13) :   7 -> 28 -> 42 -> 13 ->  7,
( 8,30,41,11) :   8 -> 30 -> 41 -> 11 ->  8.
```

Turning top, left, right, rear, and bottom are respectively defined by

```
( 1, 3, 8, 6)( 2, 5, 7, 4)( 9,33,25,17)(10,34,26,18)(11,35,27,19),
( 9,11,16,14)(10,13,15,12)( 1,17,41,40)( 4,20,44,37)( 6,22,46,35),
(25,27,32,30)(26,29,31,28)( 3,38,43,19)( 5,36,45,21)( 8,33,48,24),
(33,35,40,38)(34,37,39,36)( 3, 9,46,32)( 2,12,47,29)( 1,14,48,27), and
(41,43,48,46)(42,45,47,44)(14,22,30,38)(15,23,31,39)(16,24,32,40).
```

To analyze Rubik's cube we define a group with six elements. To rotate the front of the cube we apply the front action.

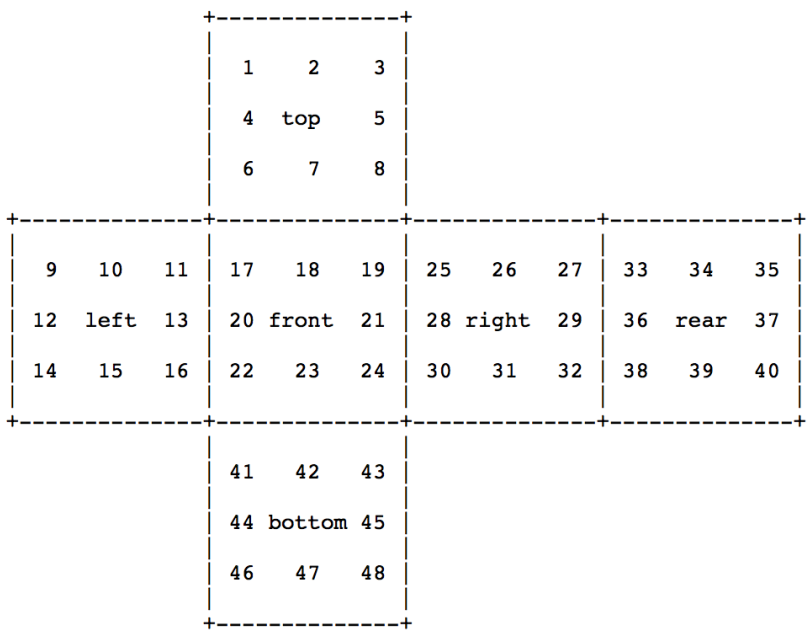

Fig. 4.16: The numbers on Rubik's cube to define the group actions.

```
gap.set('front', '(17,19,24,22)(18,21,23,20)( 6,25,43,16)( 7,28,42,13)( 8,30,41,11)')
gap.get('front')
```

Similarly, we define the top, left, right, rear, and bottom. We then define the group as generated by these six elements.

```
gap.set('top', '( 1, 3, 8, 6)( 2, 5, 7, 4)( 9,33,25,17)(10,34,26,18)(11,35,27,19)')
gap.set('left', '( 9,11,16,14)(10,13,15,12)( 1,17,41,40)( 4,20,44,37)( 6,22,46,35)')
gap.set('right', '(25,27,32,30)(26,29,31,28)( 3,38,43,19)( 5,36,45,21)( 8,33,48,24)')
gap.set('rear', '(33,35,40,38)(34,37,39,36)( 3, 9,46,32)( 2,12,47,29)( 1,14,48,27)')
gap.set('bottom', '(41,43,48,46)(42,45,47,44)(14,22,30,38)(15,23,31,39)(16,24,32,40)')
gap.set('cube', 'Group( front, top, left, right, rear, bottom)')
gap.get('cube')
```

We would like to know the size of the group.

```
size = gap('Size(cube)')
print(size)
print(size.sage().factor())
```

The size of the group is 43252003274489856000. Notice that we first have to convert the number to a Sage object before we can apply factor to see `2^27 * 3^14 * 5^3 * 7^2 * 11`.

As in the previous section, we will now decompose a random element `r`. The random element represents some arbitrary state of the cube. The decomposition of `r` tells how to obtain `r` from the six generators of the cube. With this decomposition we can restore the cube from the state `r` to its original state.

```
gap.set('r', 'Random(cube)')
gap.set('G', 'FreeGroup( "front", "top", "left", "right", "rear", "bottom")')
```

Now we define the group homomorphism:

```
gap.set('hom', 'GroupHomomorphismByImages( G, cube, GeneratorsOfGroup(G),␣
↪GeneratorsOfGroup(cube))' )
```

and then finally, we obtain the decomposition:

```
gap.set('p', 'PreImagesRepresentative(hom, r)')
gap.get('p')
```

4.7.4 Assignments

1. The dihedral group of order $2n$ consists of the isometries of a regular n-gon: (1) the n rotations through angles $2\pi k/n$ for k from 0 to $n-1$; and (2) the n reflections about lines through the center and either through a vertex or bisecting an edge. Use the GAP command `DihedralGroup` to generate the dihedral group of size 10. List all elements of this group.
2. Make an interact to visualize the moves in the Rubik's cube. There are six sliders, one for turning each side of the cube. After each move of the slider, the current state of the cube is printed as a permutation of 48 numbers.
3. As a continuation of the previous exercise, instead of printing the state of the cube as one vector of numbers, design a drawing of the six sides of the cube with numbered tiles and different colors for each side.
4. Study <http://www.gap-system.org/Doc/Examples/rubik.html> to understand how to decompose any element in the group in terms of its generators. Illustrate your understanding with a meaningful computation in a Sage worksheet.
5. After successfully completing the previous three exercises, augment the interact that allows the user to arbitrarily slide the sides of the cube with a solver. The solver applies GAP to turn the current state of the cube into its solved state.
6. Deploy the interact of the previous exercise on your personal web page at <http://people.uic.edu>.

4.8 Lecture 41: Higher Arithmetic with Pari/GP

PARI stems from Pascal ARIthmetic (although switched to C, now a pun on pari de Pascal), and GP means the Great Programmable calculator, distributed under the GNU General Public License.

PARI/GP is a widely used computer algebra system designed for fast computations in number theory (factorizations, algebraic number theory, elliptic curves...), but also contains a large number of other useful functions to compute with mathematical entities such as matrices, polynomials, power series, algebraic numbers, etc., and a lot of transcendental functions.

4.8.1 Calculating with GP in Sage

Let us first see how we can call GP in a Sage session. We first set the precision to 100 decimal places.

```
gp('default(realprecision, 100)')
pi100 = gp('Pi')
print(pi100, 'has type', type(pi100))
spi100 = pi100.sage()
print(spi100, 'has type', type(spi100))
```

The type of `pi100` is a `GpElement` and after application of the `sage()` method we obtain a real MPFR number.

We can make ASCII plots. In a Sage worksheet, in Settings the `Number of word-wrap columns` had to be changed to 100 for the plot to show up well.

```
ap = gp.eval("plot(x=0,2*Pi,sin(x))")
ap
```

The ASCII plot is shown in Fig. 4.17.

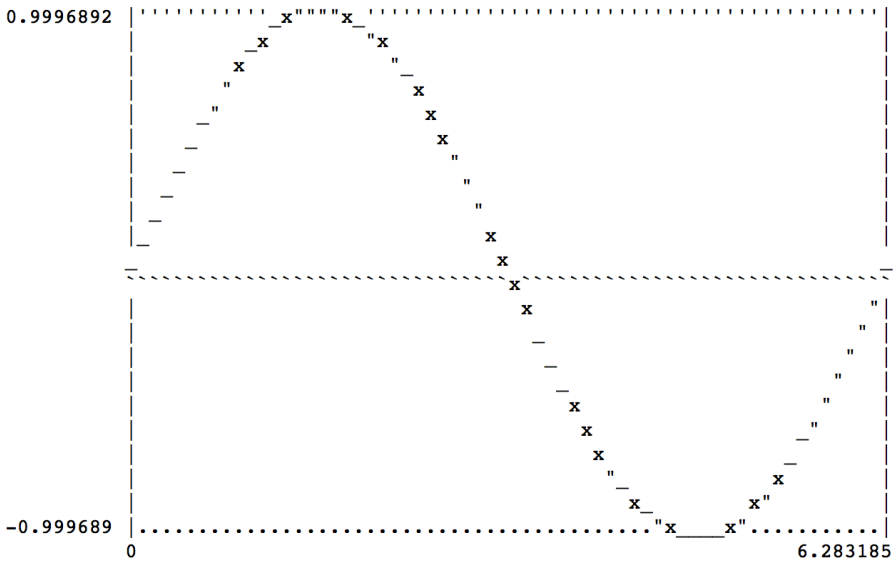

Fig. 4.17: An ASCII plot of sin(x) for x from 0 to 2*Pi.

To approximate π we could use

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$$

but this series converges very slowly. We check a slow sum to evaluate pi below.

```
gp('default(realprecision, 10)')
gp('sum4pi = 4*sum(k=0,10000,(-1.0)^k/(2*k+1))')
sum4pi = gp('sum4pi')
print(sum4pi)
print(pi.n(digits=10))
```

and we see `3.141692644` and `3.141592654`. We can accelerate a slowly alternating sum. First we reset the default precision to 100 decimal places.

```
gp('default(realprecision, 100)')
print(gp('sum4pi2 = 4*sumalt(k=0,(-1.0)^k/(2*k+1))'))
print(gp('Pi'))
```

and we see the same decimal expansion twice.

Rational approximations can be obtained with `bestappr()`.

```
gp('bestappr(Pi)')
```

As second argument of `bestappr()` we can give the size of the denominator.

```
[gp('bestappr(sqrt(2),10^%d)' % k) for k in range(1,11)]
```

We obtain the list

```
[7/5,
99/70,
1393/985,
8119/5741,
114243/80782,
665857/470832,
9369319/6625109,
131836323/93222358,
768398401/543339720,
6333631924/4478554083].
```

4.8.2 The Cauchy Integral Formula

The number of roots of $f(z)$ in a disk $C_{a,r}$ in the complex plane centered at $z = a$ and with radius r is

$$\frac{1}{2\pi i} \oint_{C_{a,r}} \frac{f'(z)}{f(z)} dx$$

We first define a polynomial. Notice the relationship between functions and polynomials, which is very similar as in Sage.

```
gp('f(z) = z^3 - z')
print gp('f')
gp('df = deriv(f(z),z)')
print(gp('df'))
gp('g(x) = substpol(df,z,x)')
print(gp('g'))
```

The `substpol` is a way to turn an expression into a function, without retyping the expression. To be sure that we have functions, we evaluate at a point.

```
print(gp('f(2)'))
print(gp('g(2)'))
```

Now we apply the Cauchy integral formula.

```
gp('intcirc(z=0, 0.5, g(z)/f(z))')
```

There is one root in a disk centered at z = 0 of radius 0.5. Because f is a cubic polynomial, if we enlarge the radius enough, we know the radius of the disk that contains all roots.

```
gp('intcirc(z=0, 1.5, g(z)/f(z))')
```

4.8.3 Expansions and Series

The syntax for expansions and series is interesting.

```
print(gp('142 + O(2^10)'))
print(gp('142 + O(3^10)'))
print(gp('142 + O(10^10)'))
```

and we see three different expansions of 142 as 2 + 2^2 + 2^3 + 2^7 + O(2^10), and 1 + 2*3 + 2*3^3 + 3^4 + O(3^10), and the familiar 2 + 4*10 + 10^2 + O(10^10).

With `truncate()` we can evaluate the series.

```
print(gp('c = 142 + O(5^10)'))
print(gp('truncate(c)'))
```

The 2 + 3*5 + 5^3 + O(5^10) is turned into 142.

The same syntax applies to Taylor series, about x = 0, of tenth order.

```
gp('t = sin(x) + O(x^10)')
```

The `truncate()` on t will return a polynomial.

```
gp('truncate(t)')
```

4.8.4 Integer Relation Detection

We can detect integer relations between floating-point approximations. As a test we take the relation `log(15) = log(3*5) = log(3) + log(5).`

```
gp('u = [log(15.0), log(3.0), log(5.0)]')
gp('u')
```

We have the list `[2.708050201, 1.098612289, 1.609437912]` of three floating-point numbers.

```
gp('r = lindep(u)')
```

The result `[-1, 1, 1]~` indicates that `-log(15) + log(3) + log(5) = 0.` The flag ~ at the end is there because the output is a column vector.

```
gp('type(r)')
```

And we see `t_COL` as the type. We can look for algebraic dependencies between numbers, which is equivalent to calling lindep on the list `[1, x, x^2, .. , x^k].`

```
gp('x = 3^(1/6)')
```

Let us check if GP can detect from the value `1.200936955` that `x` is an algebraic number, that is: that `x` is a root of a polynomial.

```
gp('algdep(x,6)')
```

On return we have `x^6 - 3`.

4.8.5 Assignments

1. Compute the first 10 best rational approximations for the transcendental number $e = \exp(1)$. The k-th number in the sequence has k decimal places in the denominator.
2. Is computing the Taylor series of `sin(x)*cos(x)` the same as multiplying the Taylor series of `sin(x)` with the Taylor series of `cos(x)`? In your answer, calculate with Taylor series of order 10.
3. Use the Cauchy integral formula to show that `sin(3*Pi*x)` has 2 roots in a disk of the complex plane centered at the origin and with radius 0.5.
4. Consider `r = 1.2599210498948732.` Find the algebraic number that is closest to `r.`

4.9 Lecture 42: Computing with Polynomials in Singular

Singular is a computer algebra system for polynomial computations, with special emphasis on commutative and non-commutative algebra, algebraic geometry, and singularity theory. Singular is integrated in Sage.

There are three ways to use Singular in Sage:

1. We can launch a Singular terminal window, or run a worksheet in Singular mode, so every command will be interpreted by Singular.
2. With `singular()` we have a Pythonic interface to Singular.
3. The input to `singular.eval()` is a string with a Singular command. The output is a string representation of a Singular object.

When we define a polynomial ring in Sage, then we are working with Singular objects.

4.9.1 Polynomials

We used Singular when we made polynomial rings.

```
R.<x,y> = PolynomialRing(QQ, order='lex')
R
```

In Sage we see `Multivariate Polynomial Ring in x, y over Rational Field` but a Singular user would get more technical information about the ring. We can get this as follows.

```
R._singular_()
```

and then we see as output

```
//   characteristic : 0
//   number of vars : 2
//        block   1 : ordering lp
//                  : names    x y
//        block   2 : ordering C
```

Polynomials in x and y automatically belong to this ring.

```
p = 4*y^2 - 4 + 4*x*y + x^2
q = 4*x^2 - 4*x*y + y^2 - 4
print(p, 'has type', type(p))
print(q, 'has type', type(q))
```

The type of both `p` and `q` is `sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular`. To see what we are solving for, we can plot the two curves defined by `p` and `q`.

```
cp = implicit_plot(p,(x,-2,2),(y,-2,2),color='red')
cq = implicit_plot(q,(x,-2,2),(y,-2,2),color='green')
(cp+cq).show(figsize=4)
```

The plot is shown in Fig. 4.18.

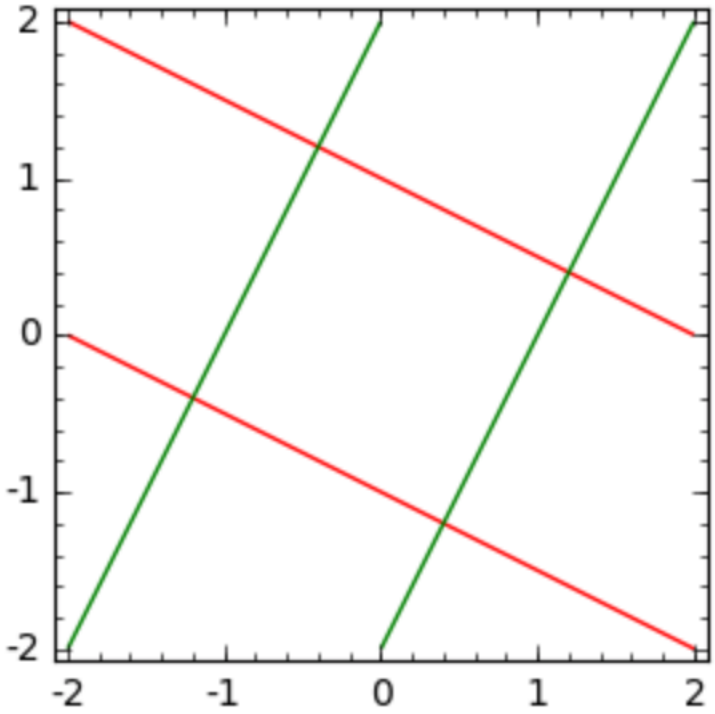

Fig. 4.18: Intersecting the green curve with the red curve.

We see that the curves break up in lines. Algebraically, we see the linear factors. Printing the outcome of `factor(p)` and `factor(q)` gives

```
(x + 2*y - 2) * (x + 2*y + 2)
(-2*x + y - 2) * (-2*x + y + 2)
```

Note that the objects we created actually belong to the Sage rings. We can declare Singular objects as follows.

```
sp = singular.new('4*y^2 - 4 + 4*x*y + x^2')
print(sp, 'has type', type(sp))
```

The type of `sp` is `sage.interfaces.singular.SingularElement`. Then on `sp` we can apply the sage() method

```
ssp = sp.sage()
print(ssp, 'has type', type(ssp))
```

The type of `ssp` is the same as `p`.

We can load a Singular library to solve polynomial systems.

```
singular.lib('solve.lib')
command = 'solve([%s, %s])' % (p, q)
print(command)
sols = singular.eval(command)
sols
```

From the output we see that we obtained four solutions. The four solutions are returned in a string, as the string representation of a list in Singular. We see that the solve in Singular solves over the complex numbers. What is returned is a string, as is always the case with the output of `singular.eval()`.

4.9.2 Reducing Polynomials with a Groebner basis

The polynomials in the ring `R` define an ideal.

```
J = Ideal([p, q])
J
```

The ideal `J` consists of all polynomial combinations we can make with its generators `p` and `q`. And then we can compute a Groebner basis.

```
G = singular.groebner(J)
G
```

Because the term order was lexicographic, the result is triangular. We can solve `x` in terms of `y`.

```
singular.reduce(x,G)
```

The reduction of `x` with respect to the Groebner basis `G` is `-125/48*y^3+41/12*y`. We see that any polynomial in `x` reduces to a polynomial in the monomial basis `1`, `y`, `y^2`, `y^3`. So with respect to this particular Groebner basis `G` every polynomial in the ring has a normal form. The `singular.reduce()` normalizes polynomials with respect to a Groebner basis. We explain next how this normalization relates to solving a polynomial system.

To set up a multiplication matrix we reduce all multiples of `y` with respect to this monomial basis.

```
L = [singular.reduce(y*y^k, G) for k in range(4)]
L
```

We see that `L` is `[y, y^2, y^3, 8/5*y^2-144/625]` The coefficients of these polynomials define a matrix. We first convert the list `L` to a list of Sage elements.

```
S = [e.sage() for e in L]
M = [[e.coefficient({y:k}) for k in range(4)] for e in S]
M
```

The type of the matrix will be over the real numbers.

```
m = Matrix(RR, 4, 4, M)
m
```

The matrix `m` is *a companion matrix*, its eigenvalues give the values for the `y`-coordinates of the solutions.

```
sy = m.eigenvalues()
sy
```

Observe that the eigenvalues are sorted. To get the corresponding `x`-coordinates we use the reduction of `x` with respect to `G`.

```
px = singular.reduce(x, G)
spx = px.sage()
sols = [(spx.subs(y=v), v) for v in sy]
sols
```

We now see the solutions as a list of tuples with values for the `x` and `y` coordinates.

4.9.3 Assignments

1. Consider the polynomial system

$$\begin{cases} x^2y - y^3 + 4x + 2 &= 0 \\ x^3y + 3x^2 - 4y - 9 &= 0. \end{cases}$$

   Use a lexicographic Groebner basis to compute the number of solutions.

   What is the number of solutions? Justify your answer.

2. The crank of length L in a four bar mechanism is centered at $(0, 0)$ and has end points $(L\cos(t), L\sin(t))$ for some angle t. We abbreviate $\cos(t)$ by c and $\sin(t)$ by s. With the introduction of c and s we replace the explicit dependency on t by the implicit equation $c^2 + s^2 - 1 = 0$. The coordinates (x, y) of the connector point are at the a distance r of a point with coordinates $(a, 0)$. The connector pont is a distance R from the end point of the crank. Then we have the system

$$\begin{cases} (x-a)^2 + y^2 - r^2 = 0 \\ (x - Lc)^2 + (y - Ls)^2 - R^2 = 0 \\ c^2 + s^2 - 1 = 0 \end{cases}$$

   Solve this system for x and y symbolically, leaving the six parameters L, c, s, a, r, and R as symbols

   1. Declare a polynomial ring in eight variables with lexicographic order with rational coefficients. Use the three polynomials to define an ideal in this ring.
   2. Compute a Groebner basis. Examine the structure of the basis. If it does not look triangular to you, then you may have to change the order of the variables as defined in the polynomial ring.
   3. Find explicit formulas for the coordinates x and y in terms of the six parameters of the problem. Is there *a discriminant* expression that is zero for bad values of the parameters?

4.10 Lecture 43: Statistical Computing with R

R is a language and environment for statistical computing and graphics. We can run R from a terminal session or evaluate cells in a Sage notebook interface with R.

4.10.1 Computations with R

The R in Sage is a little `r`.

```
c = r('choose(5,2)')
print(c, 'has type', type(c))
print(c.sage())
```

With the `sage()` we convert an `RElement` into a corresponding Sage expression.

We can evaluate R commands given as strings to `r.eval()`. The result of this evaluation is a string.

```
r.eval('choose(100, 30)')
```

Notice that '[1] 2.937234e+25' contains a floating-point number. R is not a computer algebra package, unlike Sage. In Sage we compute binomials as follows.

```
print(binomial(5,2))
print(binomial(100,30))
```

To estimate π we can run a Monte Carlo simulation. We can use that

$$\int_0^1 \sqrt{1-x^2}dx = \frac{\pi}{4}$$

to estimate π. We generate one million uniformly distributed points in the interval [0,1] and compute the mean of the height of the points.

```
data = r('u <- runif(1000000,min=0,max=1)')
estpi = r('4*mean(sqrt(1 - u^2))')
estpi
```

We see the estimate `3.142004`. To check, we use `integrate`.

```
f = r('integrand <- function(x) { sqrt(1-x^2) }')
i = r('integrate(integrand, lower=0, upper=1)')
i
```

The output is `0.7853983 with absolute error < 0.00011`.

4.10.2 Plotting Data

We generate 100 random normally distributed numbers, sort the numbers, and then make a stem and leaf plot.

```
r.eval('rn = rnorm(100)')
print r.eval('sort(rn)')
p = r.eval('stem(rn)')
p
```

The stem and leaf plot is stored as a string in `p`.

```
The decimal point is at the |

-3 | 4
-2 | 72
```

(continues on next page)

(continued from previous page)

```
-1 | 9665544332100
-0 | 99988887777666555544443333333222211 0
 0 | 00111122222233333445556666778899 9
 1 | 000224455569
 2 | 036
```

We illustrate the r.call() and making a histogram of data.

```
r.quartz() # to plot on Mac OS X
data = r.call('rnorm', 100)
r.call('hist', data)
```

The data for the histogram is below. The histogram plot is in Fig. 4.19.

```
$breaks
[1] -3.0 -2.5 -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0  2.5
$counts
[1]  1  1  6  8  9 14 21 20 15  3  2
$density
[1] 0.02 0.02 0.12 0.16 0.18 0.28 0.42 0.40 0.30 0.06 0.04
$mids
[1] -2.75 -2.25 -1.75 -1.25 -0.75 -0.25  0.25  0.75  1.25  1.75  2.25
```

Making a box plot of the data is straightforward.

```
r.call('boxplot', data)
```

The data is shown below and the plot is in Fig. 4.20.

```
$stats
           [,1]
[1,] -2.0077439
[2,] -0.4777200
[3,]  0.2852553
[4,]  0.9061189
[5,]  2.2852012
$n
[1] 100
$conf
           [,1]
[1,] 0.06660879
[2,] 0.50390188
$out
[1] -2.620645
```

The middle line in the plot indicates the medium. What is in the box are the points between the first and third quartile of the data. The horizontal lines at bottom and top show respectively the minimum and maximum of the data. Outliers are marked by circles.

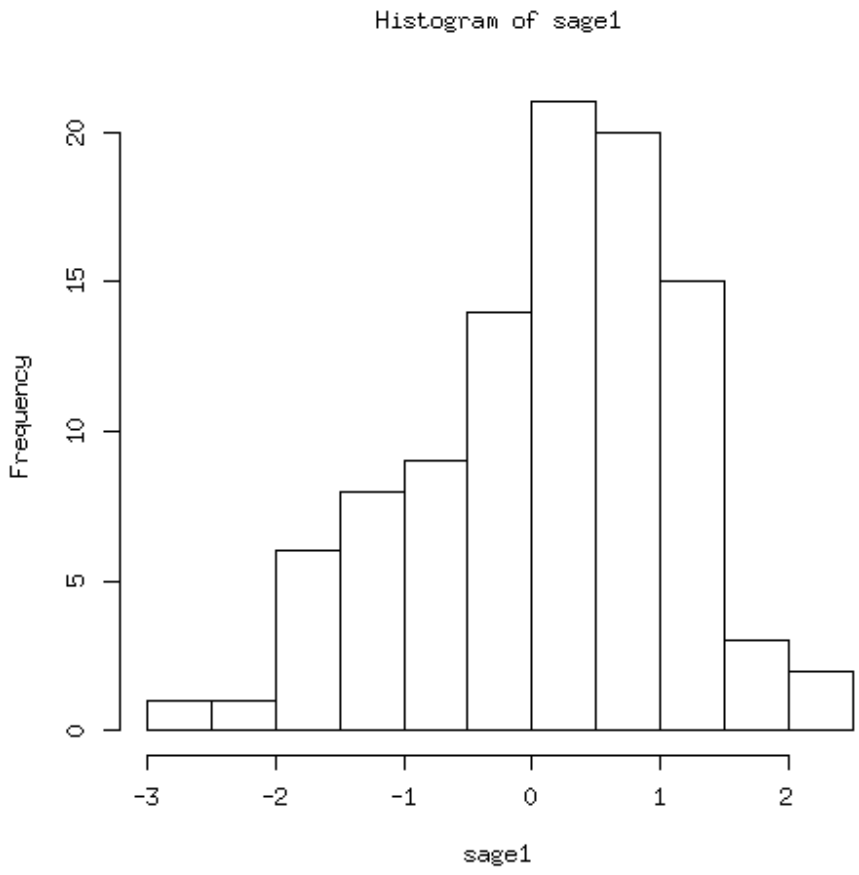

Fig. 4.19: A histogram of 100 random, normally distributed, data points.

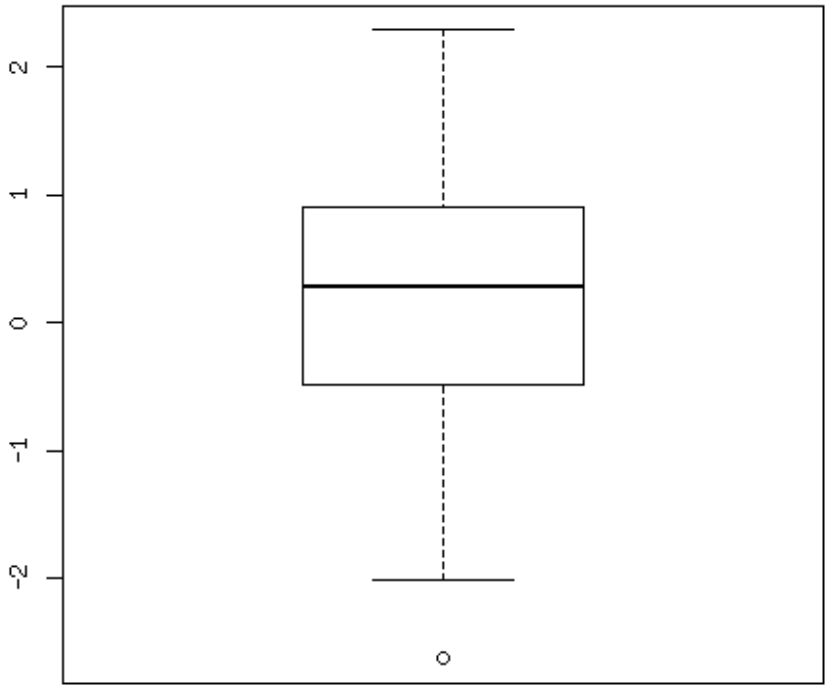

Fig. 4.20: A box plot of 100 random, normally distributed, data points.

4.10.3 Fitting Linear Models

To fit linear models, we apply `lm`. As a test we generate 50 points in [0, 10] as `x`. We set `y` equal to `x` with some random noise added. Observe the assignment in `r`.

```
r('x <- 10*runif(50)')
r('y <- x + rnorm(50)')
r.quartz()
r('plot(x,y)')
```

The plot is shown in Fig. 4.21.

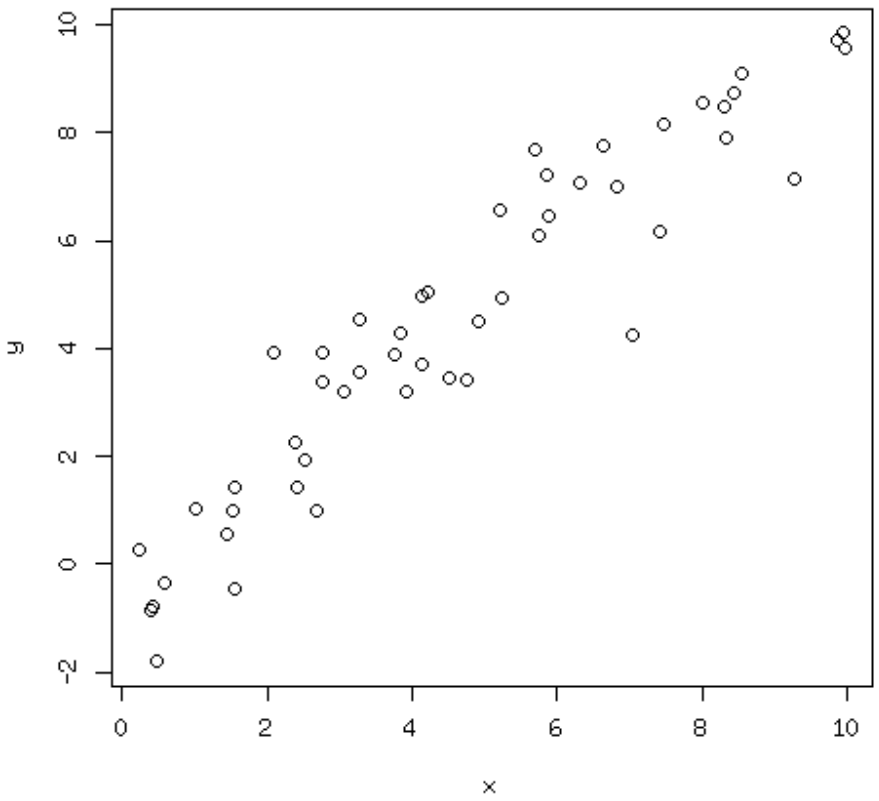

Fig. 4.21: A plot of 50 points on a line with random noise added.

Now we do the fit with the command `lm`.

```
r('fit <- lm(y ~ x)')
```

For the output we can read off the intercept `b` and the slope `a` for the linear fit `y = a*x + b`. See below:

```
Call:
lm(formula = y ~ x)
```

(continues on next page)

(continued from previous page)

```
Coefficients:
(Intercept)            x
     0.1891       0.9573
```

We can ask for a summary.

```
r('summary(fit)')
```

The summary of the `fit` is below.

```
Call:
lm(formula = y ~ x)

Residuals:
     Min       1Q   Median       3Q      Max
-1.84906 -0.75229 -0.07965  0.40908  2.46077

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.18907    0.28316   0.668    0.508
x            0.95734    0.04729  20.245   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 1.014 on 48 degrees of freedom
Multiple R-squared:  0.8952, Adjusted R-squared:  0.893
F-statistic: 409.8 on 1 and 48 DF,  p-value: < 2.2e-16
```

CHAPTER 5

Reviews

The last week of classes is devoted to reviews. The problems listed in this part are representive for the questions on the final exam. However, this list is not exhaustive. Also review the problems on the quizzes, homework assignments, and midterm exams.

5.1 Lecture 44: Third Review

The question on the third review focus on the number system and basic data structures. This review corresponds to the first review, to prepare for the first midterm exam.

5.1.1 Our First Steps

1. Explain the difference between RDF and RR. Illustrate with an example.
2. Give a list of ten rational approximations for $\sqrt{3}$, accurate with 2, 3, up to 11 decimal places.
3. Consider $\mathbb{Z}_{31}$.
    1. What is the multiplicative inverse of 7?
    2. Show that $15x^5 + 4x^4 + 23x^3 + 26x^2 + 6x + 1$ is irreducible over $\mathbb{Z}_{31}$.
4. Expand `exp(I*2*Pi/k)` as `cos(2*Pi/k) + I*sin(2*Pi/k)`.
5. Consider $q = \cos(x^3 - 1) + 3\sin(y) - z^7$. Draw the expression tree of q.
6. Save a random matrix to file, destroy its reference, and load then the saved matrix back into SageMath.
7. Generate optimized code to evaluate $p = 79x^{298} + 56x^{205} + 49x^{164} + 63x^{121} + 57x^{119} - 59x^{42}$.
    1. How many operations are needed to evaluate p?
    2. Compare with the cost of a direct evaluation of p.

8. Consider the right hand side of

$$\int_1^2 \frac{1}{x}dx \approx \frac{1}{2n}\left(1+\frac{1}{2}\right) + \frac{1}{n}\sum_{k=1}^{n}\left(\frac{1}{1+k/n}\right).$$

   1. Write a Python function `P` to define the evaluation of the right hand side of the above sum, for any value for n which is the input parameter to the function `P`.

      Compare the value of `P(10000)` with the exact value of the integral.

      Time the execution of the function for $n = 10,000$.

   2. Use vectorization to define a more efficient function `F` which computes the same sum as `P`, and which also takes n as an input parameter.

      Compare the value of `F(10000)` with the exact value of the integral.

      Time the execution of the function for $n = 10,000$.

   Compare the result of the timings.

5.1.2 Polynomials and Rational Expressions

1. Draw the internal representation of $xy(x-y)$.
2. Consider $p = x^3 - x - 2$ and give all commands for an exact, numeric, and symbolic factorization. In particular, answer the following questions.
   1. How can you write p as an *exact* product of linear factors, with *exact* complex numbers?
   2. Compute a *numerical* factorization of p over the complex numbers.
   3. Define a *symbolic* (i.e.: formal) factorization of p, declaring sufficiently many roots.
   4. Compute a *symbolic-numeric* factorization, extending the field of rational numbers with complex intervals.
3. Give all commands to transform $(x-y)(x+y)$ into $(x+y)x-(x+y)y$.
4. Consider $r = \dfrac{79x^5+56x^4+49x^3+63x^2+57x-59}{45x^5-8x^4-93x^2+43x-62}$.
   1. Convert r into a form which is more efficient to evaluate.

      Compare the number of arithmetical operations needed to evaluate r in this more efficient form with the number of arithmetical operations to evaluate r in its given form.

   2. Compare this number of operations with the Horner forms of numerator and denominator for r.
5. Explain why normal forms are so important to symbolic computation.

   What can we do if a normal form is too expensive to compute? Illustrate with a good example.

5.2 Lecture 45: Fourth Review

The question on the fourth review focus on

1. function definitions, differentiation, integration;
2. plotting in two and three dimensions;
3. solving linear, differential, polynomial equations, linear programming.

The material corresponds to the second review, which prepared for the second midterm exam.

5.2.1 Calculus

1. Write a function to make polynomials in a system. The k-th polynomial in the system is

$$f_k(x_1, x_2, \ldots, x_n) = x_k + \sum_{i=1}^{n-k} x_i x_{k+i}, \quad k = 1, 2, \ldots, n.$$

   For example, for $n = 8$, the polynomials are

```
x1*x2 + x2*x3 + x3*x4 + x4*x5 + x5*x6 + x6*x7 + x7*x8 + x1
x1*x3 + x2*x4 + x3*x5 + x4*x6 + x5*x7 + x6*x8 + x2
x1*x4 + x2*x5 + x3*x6 + x4*x7 + x5*x8 + x3
x1*x5 + x2*x6 + x3*x7 + x4*x8 + x4
x1*x6 + x2*x7 + x3*x8 + x5
x1*x7 + x2*x8 + x6
x1*x8 + x7
x8
```

2. Define a piecewise function `int_inv_cub` which as function of the end point b always returns the correct value of $\int_{-1}^{b} \frac{1}{x^3} dx$.

3. The arc length of continuous function $f(x)$ over an interval $[a, b]$ can be defined as $\int_a^t \sqrt{1 + [f'(x)]^2}$.

   1. Compute the arc length of the positive half of the unit circle, i.e.: $f(x) = \sqrt{1 - x^2}$ (answer $= \pi$).
   2. Create a function (call it `arc_length`) in t which returns a 10-digit floating-point approximation of the arc length of the positive half of the circle, for $x \in [0, t]$.

4. Consider the recurrence relation

$$h(n) = 5h(n-1) - 6h(n-2), \quad \text{for } n \geq 2, \quad \text{with } h(0) = 1 \text{ and } h(1) = -2.$$

   Answer the following the questions.

   1. The generating function $g(x) = \dfrac{1 - 7x}{1 - 5x + 6x^2}$ defines $h(n)$ as the coefficient with x^n in the Taylor expansion of $g(x)$. Use $g(x)$ to define h as a function (call it `t`) of n which gives the value of $h(n)$.
   2. Write a function to compute $h(n)$, directly using the recurrence relation from above. Make sure your function can compute $h(120)$. Compare with the result of (a).

5. The Legendre polynomials are defined by

$$P_0(x) = 1, \quad P_1(x) = x, \quad P_n(x) = \frac{2n-1}{n} x P_{n-1}(x) - \frac{n-1}{n} P_{n-2}(x), \text{ for } n \geq 2.$$

   Write a efficient recursive function `legendre` to compute $P_n(x)$. The function `legendre` takes on input the degree n and the variable x.

   Compare the output of your `legendre` (50, x) with the `legendre_P` (50, x).

6. Consider the point $P = (1, 1)$ on the curve defined by $xy - 2x + 1 = 0$.

   Compute the slope of the tangent line to the curve at P in two ways:

   1. with implicit differentiation,
   2. with a Taylor series.

5.2.2 Plotting and Solving Equations

1. Suppose we want to plot the curve $x^4 + x^2y^2 - y^2 = 0$ for x and y both between -1 and $+1$.
   1. Sampling this curve as given in rectangular coordinates, how many samples do we need to take from the curve to obtain a nice plot?
   2. Convert the curve into polar coordinates and plot. Give all commands used to obtain the plot. How many samples of the curve are needed here?
2. Solve $x^2a^2 - 2x^2a - 3x^2 - xa^2 + 4xa - 3x + a^2 + 2a - 15$ for x for all values of the parameter a.

   Be as complete as possible in your description of the solution.
3. Find the point with real coordinates on the curve $xy - 2x + 3 = 0$ closest to the origin.
4. Consider the system

$$\begin{cases} x^2 - 2y^2 - 1 &= 0 \\ xy - 2x - 3 &= 0. \end{cases}$$

   How many real solutions does this system have?
5. Consider $y'' + 6y' + 13y = 0$, with $y(\pi/2) = -2$ and $y'(\pi/2) = 8$.
   1. Find an exact solution to this initial value problem and use this to create a function f which returns a numerical 10-digit floating-point approximation of the solution.
   2. Solve this initial value problem numerically. Compare the solution with the value for $y(2)$ and also with $f(2)$ obtained in (a).
6. A 5-by-5 variable Toeplitz matrix has the following form:

```
[t0 t1 t2 t3 t4]
[t8 t0 t1 t2 t3]
[t7 t8 t0 t1 t2]
[t6 t7 t8 t0 t1]
[t5 t6 t7 t8 t0]
```

   for the symbols in the list `[t0, t1, t2, t3, t4, t5, t6, t7, t8]`.

   For general dimension n, the (i, j)-th element of the Toepliz matrix T is

$$T_{(i,j)} = \begin{cases} j - i & \text{if} \quad j \geq i \\ j - i + 2n - 1 & \text{if} \quad j < i. \end{cases}$$

   Give the command(s) to define a variable Toeplitz matrix, for any dimension n.
7. Maximize $x_1 + x_2$ subject to $-x_1 + 2x_2 \leq 8$, $4x_1 - 3x_2 \leq 8$, $2x_1 + x_2 \leq 14$, $x_1 \geq 0$, and $x_2 \geq 0$.

   Write the commands to define this problem and then solve it. What are the values of x_1 and x_2 at the optimal solution?

5.3 Lecture 46: Fifth Review

For the last part of the course took, we can divide the topics roughly as:

1. building interactive web pages;
2. the numpy, scipy, and sympy stack;
3. introduction to Julia
4. GAP, PARI/GP, Singular and R.

Below are some additional, preliminary questions.

1. Consider the curve defined by $r = \sin(8t)$. Make an interact to plot this curve.

   The range for t always starts at zero. The end of the range for t is controlled by a slider. The initial value for the end is $\pi/2$. The increment for the end value is $\pi/40$.

2. Use numpy to solve a 5-by-5 tridiagonal system $A\mathbf{x} = \mathbf{b}$.

   1. The diagonal element of A is 5, the elements just above and below the diagonal are one. Everywhere else the matrix is zero.
   2. Define a 5-dimensional right hand side vector $\mathbf{b}$ of ones. Solve the system $A\mathbf{x} = \mathbf{b}$ and compute the residual.

3. Consider the permutations $a = (1,4)(2,3)$ and $b = (4,5)(3,6)$.

   1. What is $a \star b$?
   2. What is the size of the group generated by a and b?

4. Use the Cauchy integral formula to compute the number of complex roots in a disk centered at 0 and with radius 1.1 of $(x+1)\sin(2x)$.

   Give the number of roots in that disk of the complex plane.

5. Consider the polynomial system defined by the polynomials $p = x^2y - 2x + 3$ and $q = xy^2 - 2y + 3$.

   1. Bring the system in triangular form. Use this triangular form to determine the number of complex solutions.
   2. If N is the number of solutions, compute the companion matrix of the system. The rows of this matrix are the reductions of the products of y with y^k for k ranging between 0 and $N-1$. Show that the y-coordinates of the solutions are the eigenvalues of this companion matrix.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Bibliography

[Bard15] G. Bard: *Sage for Undergraduates*. American Mathematical Society, 2015. Available via `www.gregory-bard.com/books.html`.

[Heck96] A. Heck: *Introduction to Maple*. Second Edition, Springer-Verlag, 1996.

[Stein12] W. Stein: *Sage for Power Users*. Available via `https://wstein.org/books/sagebook`.

[Zimm18] P. Zimmermann, A. Casamayou, N. Cohen, G. Connan, T. Dumont, L. Fousse, F. Maltey, M. Meulien, M. Mezzarobba, C. Pernet, N.M. Thiery, E. Bray, J. Cremona, M. Forets, A. Ghitza, and H. Thomas: *Computational Mathematics with SageMath*. SIAM 2018. Available as `dl.lateralis.org/public/sagebook/sagebook-ba6596d.pdf`.

[BS10] B. Erocal and W. Stein: The Sage project: Unifying free mathematical software to create a viable alternative to Magma, Maple, Mathematica, and MATLAB. In *Mathematical Software - ICMS 2010*, Springer-Verlag, 2010.

[KRPetal16] T. Kluyver, B. Ragan-Kelley, F. Pérez, B.Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, and Jupyter Development Team: Jupyter Notebooks — a publishing format for reproducible computational workflows. In F. Loizides and B. Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents, and Agendas*, pages 87–90. IOS Press, 2016.

[BEKS17] J. Bezanson, A. Edelman, S. Karpinski, V.B. Shah. A fresh approach to numerical computing. *SIAM Review* 59(1): 65-98, 2017.

[GMCGSER22] S. Gowda, Y. Ma, A. Cheli, M. Gwozdz, V.B. Shah, A. Edelman, and C. Rackauckas. *High-performance symbolic-numerics via multiple dispatch*, arxiv:2105.03949v3 [cs.CL] 5 Feb 2022.

Index

U

V